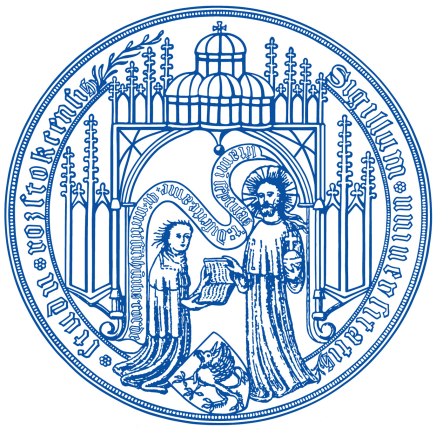
Big Data Analytics für die effiziente Aktivitätserkennung und -vorhersage in Assistenzsystemen

Dissertation
zur
Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
der Fakultät für Informatik und Elektrotechnik
der Universität Rostock



vorgelegt von:	Dennis Marten
geboren am:	04.05.1988
vorgelegt am:	14.06.2021
Disputationstermin:	10.11.2022
Erstgutachter:	Prof. Dr. rer. nat. habil. Andreas Heuer (Universität Rostock)
Zweitgutachter:	Prof. Dr.-Ing. Thomas Kirste (Universität Rostock)
Drittgutachter:	Prof. Dr.-Ing. Wolfgang Lehner (TU Dresden)

https://doi.org/10.18453/rosdok_id00004113

Inhaltsverzeichnis

Kapitel 1

Einführung

Assistenzsysteme sind integraler Bestandteil des modernen Alltags und haben sich in den letzten Jahren und Jahrzehnten in einer Vielzahl neuer Anwendungsgebiete und -szenarien etabliert. Der Erfolg dieser Systeme beruht maßgeblich auf der Ergründung immer komplexer werdender Erkennungs- und Vorhersagemodelle. Die Güte dieser wird vor allem durch die über die Jahre kontinuierlich ansteigenden Mengen an relevanten Trainingsdaten bestimmt. Ursachen hierfür sind maßgeblich die technische Entwicklung von Sensor- und Vernetzungstechnik, mit der eine signifikante Vergünstigung einherging, [1] und der voranschreitende allgemeine Digitalisierungsprozess.

Um mit dem schnellen Anstieg zu verarbeitender Datenmengen umgehen zu können, hat sich im Zuge datengetriebener Anwendungen ein gesteigerter Fokus von Forschungs- und Entwicklungsteams auf transparent-skalierbares massiv-paralleles Rechnen gebildet. Einer der wohl einschlägigsten Meilensteine dieser Bewegung war die Vorstellung des MapReduce-Frameworks, welches seit seiner initialen Publikation im Jahr 1998 [2] Basis einer Vielzahl parallel agierender Frameworks für Big-Data-Anwendungen (vergleiche Abschnitt 2.4) geworden ist.

Nahe dem Höhepunkt der Aufmerksamkeit des MapReduce-Programmiermodells um 2010 untersuchte eine Gruppe um Turing-Award-Gewinner *Michael Stonebraker* in [3] Performance-Unterschiede paralleler relationaler Datenbanksysteme (PRDBS) (vergleiche Abschnitt 2.3.5) mit denen von MapReduce-Implementationen in Apache Hadoop (siehe Kapitel 4.3.1). Die zugrundelegende Prämisse war hierbei die, dass PRDBS seit Jahrzehnten bereits Anwendung im Bereich der verteilten Verarbeitung enormer Datenmengen fanden und im Laufe der Zeit deren zugrundeliegenden Techniken stetig gepflegt und weiterentwickelt wurden. Aus diesen Gründen wurde vermutet, dass Datenbanksysteme großes Potenzial für die Anwendung in Big-Data-Szenarien haben. In den Tests wurden die Laufzeiten der folgenden drei klassischen Szenarien

[1] Goldmann Sachs schätzte etwa im Jahr 2016, dass mehr als eine Drittelung des durchschnittlichen Preises von Sensoren im Kontext von Internet-of-Things-Anwendungen im Zeitraum von 2004 bis 2020 zu erwarten war [1].

	Hadoop	DBMS-X (Rowstore)	Vertica (Columnstore)
Grep	284s	194s	108s
Web Log	1146s	740s	268s
Join	1158s	32s	55s

Tabelle 1.1: Experimentalergebnisse der Untersuchungen aus [3]. Verglichen wurden Hadoop, ein namentlich nicht genanntes paralleles DBS, sowie das System Vertica.

verglichen:

1. Grep-Task
2. Web-Log-Task
3. Verbundberechnung

Hierbei beschreibt der Grep-Task die Suche einer dreistelligen Zeichenfolge in insgesamt 100 Milliarden Dokumenten mit einem Terabyte Gesamtgröße, welches auf 100 Knoten mit jeweils 10 Gigabyte berechnet wurde. Der Web-Log-Task entspricht einer SQL-Aggregation mit `Group By` (vergleiche Abschnitte 2.3.3 und 2.3.4) auf Besucherdaten eines web server logs. In diesem wurden 2 Terabyte Daten aus 155 Millionen Dokumenten genutzt und in 20 Gigabyte Partitionen auf 100 Knoten verteilt. Das abschließende Problem entsprach der Berechnung eines Verbundes mit Zwischenaggregation und Filterung zwischen den Besucherdaten des Web-Logs und einer 100 Gigabyte Tabelle von PageRank-Daten von 18 Millionen URLs. Die Ergebnisse der Experimente sind in Tabelle 1.1 dargestellt.
Aus dieser lässt sich leicht entnehmen, dass die Datenbanklösungen „DBMS-X“ und Vertica die von Hadoop/MapReduce stets unterbieten. Selbst der Grep-Task, welcher klassischerweise als ein Standardbeispiel MapReduce-affiner Aufgaben bezeichnet wird, kann hierbei mehr als doppelt so schnell gelöst werden. Bei der Verbundberechnung wurden sogar Performance-Abstände um zwei Ordnungen beobachtet. In [3] werden eine Vielzahl möglicher Gründe für diese Ergebnisse, wie zum Beispiel wiederholtes Parsen textuell hinterlegter Daten in Hadoop, sowie bessere interne Anfageoptimierung, Zwischenergebnisverwaltung und Datenkompression in Datenbanksystemen, angegeben.
Diese Ergebnisse konnten im Laufe des vorliegenden Forschungsprojektes durch eine Studentengruppe der Universität Rostock in [4] im moderneren Stand der Technik bestätigt werden (vgl. Abschnitt 10.2.3 für detailliertere Beschreibungen). In diesem wurden keine MapReduce-Implementationen, sondern Umsetzungen der drei Anwendungsfälle auf den Parallelisierungsplattformen Apache Flink und Apache Spark (vgl. Abschnitt 4.3.2) getestet. Diese Systeme erweitern das vergleichsweise starre MapReduce-Programmiermodell durch mehrere Operato-

ren und erlauben die parallele Verarbeitung von Operatorfolgen in Form gerichteter azyklischer Graphen.

Aus diesen Ergebnissen wird eine Untersuchung zur parallelen Verarbeitung von Methoden aus dem Kontext von Assistenzsystemen in PRDBS mit Datenbeständen im Big-Data-Bereich motiviert. Mithilfe paralleler Datenbanken wird hierbei auf eine transparente Unterstützung des Entwicklungsprozesses solcher Systeme abgezielt. Wie sich zeigen wird, ist hierfür eine Ausweitung klassischer SQL-Anwendungen in den Bereich des Machine Learning und der Linearen Algebra nötig. Zugehörige Entwicklungen wurden hierbei über die Jahre in verschiedenen Ansätzen angestoßen, wobei der Fokus hauptsächlich auf der systemspezifischen Einführung zusätzlicher Funktionalitäten lag (vgl. Abschnitt 4.4). Eine systematische und weitreichende Untersuchung von Machine-Learning- und Lineare-Algebra-Methoden auf SQL wurde jedoch ausgespart. Dieser Ansatz birgt aufgrund der Standardisierung der weitverbreiteten Anfragesprache SQL ebenfalls großes Potenzial und wird in dieser Arbeit verfolgt. Motivierend hierzu hat sich etwa der zweifache Test-of-Time-Award-Preisträger *Dan Suciu* in einem Interview im Dezember 2017 in [5] geäußert. In diesem bekräftigt er wiederholt den Nutzen massiv-parallelisierter Linearer Algebra auf SQL- anstatt auf MapReduce-Systemen[2].

Neben möglicher Performance-Vorteile gegenüber klassischen Big-Data-Umgebungen werden sich hierbei eine Vielzahl weiterer Vorteile zeigen, wie Aspekte der Datensicherheit, die Ausnutzung logischer und physischer Optimierung, der Systemunabhängigkeit oder der Langlebigkeit von Implementationen durch die Berechnung mittels SQL in PRDBS. Zusätzlich wird sich zeigen, dass die Verwendung des standardisierten SQL-Kerns eine nachträgliche Erweiterung durch zusätzliche SQL-Schnittstellen ermöglicht, die beispielsweise zur Wahrung von Privatheitsaspekten genutzt werden können. Letztere sind etwa im Kontext von Assistenzsystem von besonderer Bedeutung, da diese oftmals dauerhaft personenbezogene Daten mittels Sensoren erheben [6].

Um eine detailliertere und besser nachvollziehbare Beschreibung der Rahmenbedingungen, sowie Unterstützungsmöglichkeiten des Entwicklungsprozesses von Assistenzsystemen durch die Verarbeitung in Datenbanksystemen zu ermöglichen, werden, nach einer kurzen Gliederung der Arbeit und Einordnung zugehöriger wissenschaftlicher Publikationen, im folgenden Kapitel wesentliche problembezogene Begrifflichkeiten eingeführt. Mit diesen werden im darauf folgenden Kapitel 3 eine detailliertere Diskussion zum wissenschaftlichen Konzept und dem Anwendungspotenzial im Kontext von Assistenzsystemen im Big-Data-Bereich geführt.

[2] Originalzitat zur Nachfrage, ob Lineare-Algebra-Funktionalitäten auf SQL anstatt MapReduce umgesetzt werden sollten: ¨ No, I really meant on top of SQL, on top of relational languages. I think they are here to stay. People have tried to replace them for many years . . . ¨

Gliederung und Einordnung wissenschaftlicher Publikationen

Im Folgenden wird zunächst eine Auflistung der Publikationen, die im Zuge der vorliegenden Arbeit veröffentlicht wurden, präsentiert. Daraufhin wird eine kurze Beschreibung der Gliederung der vorliegenden Arbeit gegeben, indem zusätzlich eine inhaltliche Einordnung der Publikationen vorgenommen wird.

Publikationen

Im Zuge des Dissertationsprojekts wurden insgesamt sieben Publikationen (vier Konferenzbeiträge und drei Artikel in Open Journals) als Erstautor veröffentlicht. Die Beiträge sind folgend in chronologischer Reihenfolge aufgelistet:

1. „A framework for self-managing database support and parallel computing for assistive systems“ [7] in *Proceedings of the 8th ACM International Conference on PErvasive Technologies Related to Assistive Environments* (2015)

2. „Transparente Datenbankunterstützung für Analysen auf Big Data“ [8] in *Proceedings of the 27th GI-Workshop Grundlagen von Datenbanken* (2015)

3. „Machine Learning on Large Databases: Transforming Hidden Markov Models to SQL Statements“ [9] in *Open Journal of Databases, Volume 4* (2017)

4. „Sparse and Dense Linear Algebra for Machine Learning on Parallel-RDBMS Using SQL“ [10] in *Open Journal of Big Data, Volume 5* (2019)

5. „Calculating Fourier Transforms in SQL“ [11] in *Advances in Databases and Information Systems - 23rd European Conference* (2019)

6. „Database Support for Automotive Analysis“ als Shortpaper [12] in *Proceedings of the Conference on „Lernen, Wissen, Daten, Analysen“* und als Langversion in Form eines technischen Berichts [13] (2019).

7. „Scalable In-Database Machine Learning for the Prediction of Port-to-Port Routes“ [14] in *Journal für Mobilität und Verkehr, Volume 6* (2020)

Im Laufe der vorliegenden Arbeit wird in den einleitenden Absätzen von Kapiteln oder Abschnitten erneut darauf hingewiesen, wenn diese inhaltlich auf eine oder mehrere der aufgelisteten Publikationen aufbauen.

Gliederung

Die vorliegende Arbeit zielt auf die Unterstützung des Entwicklungsprozesses von Assistenzsystemen mittels Datenbanktechnologie ab. Hierfür wird die Arbeit wie folgt gegliedert.

Zunächst werden in Kapitel 2 Begrifflichkeiten zu Assistenzsystemen, zur Aktivitäts- und Intentionserkennung, zu relationalen Datenbanksystemen und zu Big Data grundlegend eingeführt. Hierbei wird insbesondere der Aufbau ersterer und der Zusammenhang zwischen den vier genannten Gebieten erläutert.

Mit der Etablierung dieser Begrifflichkeiten und der klassischen Anwendungsgebiete paralleler Datenbanksysteme wird in Kapitel 3 ein Konzept zur Unterstützung von Entwicklern von Assistenzsystemen vorgestellt. In diesem wird eine vollständige Verarbeitung von Verfahren der Aktivitätserkennung und -vorhersage im SQL-Kern innerhalb paralleler Datenbanksysteme propagiert. Das zugehörige Framework bildet den Untersuchungsschwerpunkt der vorliegenden Arbeit und wurde erstmalig in [9], in der hier diskutierten Form, näher erläutert. Mit der Erörterung von Anforderungen und damit verbundener Bedingungen, wird abschließend der von dort an weitere Verlauf zur systematischen Untersuchung des Konzepts motiviert und näher beleuchtet. Aus diesem Grund wird die hier beschriebene Gliederung nur in Kurzform weiter geführt.

Nach der Konzeptbeschreibung wird zunächst der Stand der Technik und der Stand der Forschung in Kapitel 4 erörtert. Hierbei werden im weiteren Verlauf nötige Systeme, Bibliotheken und Funktionalitäten, sowie Forschungsprojekte, die Datenbanksystemtechnologie und wissenschaftliches Rechnen beziehungsweise Machine Learning verbinden, beschrieben.

Mit der Abgrenzung zum Stand der Technik und Forschung werden in Kapitel 5 zunächst Hidden-Markov-Modelle (HMM), als eines der wesentlichen Vertreter von Machine-Learning-Verfahren für Assistenzsysteme, eingeführt. Hierbei wird insbesondere gezeigt, dass etablierte Lösungsverfahren dieser maßgeblich auf Basisoperatoren der linearen Algebra bestehen. Teilaspekte hiervon wurden erstmalig in [9] beschrieben und ausgewertet. Die effiziente Umsetzung in relationalen Systemen wurde in [10] durch Tests evaluiert.

Mit der Etablierung wichtiger Grundoperatoren zur Berechnung von HMM-Methoden werden in Kapitel 6 Datenrepräsentationsaspekte, wie etwa eine Diskussion zur Framework-Architektur (inkl. Datenbanksystemwahl) und geeigneter Relationenschemata für Matrizen und Vektoren, geführt. In [7, 8] wurden diesbezügliche Aspekte zur Datenbanksystemwahl erörtert und experimentell ausgewertet. In [9] wurde eine Diskussion zur Architektur für vollständige SQL-Unterstützung geführt. Eine kurze Abhandlung geeigneter Relationenschema wurde in [10] gegeben.

Basierend auf der Darstellung von Matrizen und Vektoren in Datenbanksystemen werden in Kapitel 7 mehrere Aspekte der Überführung von Lineare-Algebra-Operatoren in Standard-SQL erörtert und streckenweise experimentell evaluiert. Teile der Ergebnisse des Kapitels wurden

hierbei in [9] und [10] vorgestellt.
Nachdem Möglichkeiten zur Darstellung und Komposition von Lineare-Algebra-Operatoren in SQL etabliert worden sind, wird in Kapitel 8 ein Ansatz zur effizienten Parallelisierung dieser Operatoren auf parallelen relationalen Datenbanksystemen mittels Anfragezerlegung vorgestellt. Das zugehörige Konzept wurde in kürzerer Form in [10] dargelegt.
In Kapitel 9 werden die Ergebnisse zur parallelen Berechnung dieser Operatoren aus [10] vorgestellt und diskutiert. Zusätzlich wird deren Performance auf einer etablierten State-of-the-Art Big-Data-Umgebung evaluiert und mit denen der parallelen Datenbanksysteme verglichen. Zur Verdeutlichung der Erweiterbarkeit des vorgestellten Konzeptes werden zudem die Berechnung von Fourier-Transformationen und Zeitreihenanalysen in SQL aufgeführt. Die Ergebnisse und Ausführungen sind hierbei aus [11] und [12, 13] zusammengefasst dargestellt. Abschließend wird ein Projekt zur SQL-basierten, datenbankinternen Vorhersage von Schiffsrouten mittels Markov-Ketten vorgestellt. Die Ausführungen stellen eine Zusammenfassung von [14] dar. Diese Beispielanwendung bestärkt die Nützlichkeit des in Kapitel 3 diskutierten Konzeptes im industriellen Kontext.
Im finalen Kapitel 10 wird eine Zusammenfassung und Einordnung der erarbeiteten Ergebnisse gegeben. Zusätzlich werden mögliche künftige Betrachtungen und erste Erkenntnisse zu diesen vorgestellt.

Kapitel 2

Grundlegende Begriffe

In diesem Kapitel werden zunächst grundlegend die Begriffe Assistenzsysteme, Aktivitäts- und Intentionserkennung, Big Data, (parallele) relationale Datenbanksysteme und deren Beziehungen untereinander erläutert. Basierend auf diesen Ausführungen wird im folgenden Kapitel 3 ein Konzept für eine effiziente Aktivitätserkennung und -vorhersage in Assistenzsystemen mittels paralleler relationaler Datenbanksystemen vorgestellt, welches den Untersuchungsschwerpunkt der vorliegenden Arbeit bestimmt.

2.1 Assistenzsysteme

Assistenzsysteme werden heutzutage bereits zur Verbesserung oder Erleichterung vielfältigster Situationen genutzt und sind schwer aus dem modernen Lebensalltag wegzudenken. Im Zeitalter des Internet of Things stehen vor allem komplexere und vernetzte Applikationsmöglichkeiten im Fokus der Industrie und Forschung. Mögliche Anwendungsszenarien variieren hierbei stark. Sie reichen etwa vom autonomen Fahren von Automobilen, sprachlicher Computernavigation bis hin zu Anwendungen im Gesundheits- und Pflegesektor, wie der Unterstützung und Auswertung von Ernährungsgewohnheiten [15] oder der Sturzerkennung in Pflegeheimen [16].
Assistenzsysteme arbeiten generell gemäß einer mehrschichtigen pyramidenförmigen Architektur, wie sie in Abbildung 2.1 dargestellt ist [6, 17, 18]. Die Pyramidenform bezieht sich hierbei auf die tendenziell abnehmenden Datenmengen von der Sensorschicht hin zum Nutzer an der Spitze.
In der untersten Schicht werden stetig heterogene Daten durch diverse Sensoren und Ortungskomponenten gesammelt. Da hierbei die zu assistierenden Personen dauerhaft beobachtet werden, sind diese Daten oftmals personenbezogen. Aus diesem Grund sollte eine ungefilterte Weiterreichung in obere Schichten, im Sinne der Datenprivatheit, minimiert werden. Ansätze hierfür wurden etwa in [19–21] präsentiert, in denen beispielsweise intelligente Sensoren genutzt wer-

Abbildung 2.1: Pyramidenförmige Architektur von Assistenzsystemen. Die Grafik ist [6] entnommen.

den, um Selektionen von Datenanfragen höherer Schichten bereits nahe am Sensor oder lokal nahegelegenen Prozessor durchzuführen.
Um die generierten Daten nutzen zu können, werden diese in der nächsten Schicht mittels Gerätekopplung und Aufbereitung durch Techniken des Internet of Things homogenisiert.
Da die Datenmenge selbst mit Vorfilterung extrem groß werden kann, müssen die homogenisierten Daten komprimiert und verteilt gespeichert werden. Dies wird klassischerweise mittels verteilter oder paralleler Datenbanksysteme realisiert.
Das Assistenzsystem kann zu diesem Zeitpunkt die aktuelle Situation beziehungsweise Handlungen oder Intentionen von Personen anhand der verteilten Daten ableiten. Abhängig von der Datengröße, Vorfilterungen und den zugehörigen Berechnungsmodellen geschieht dies lokal oder auch mittels Cloud-Computing.
Die Erkennung von Aktivitäten, Situationen oder Intentionen ist die Grundlage, um sinnvolle unterstützende Informationen oder Aktionen dem Nutzer zur Verfügung stellen zu können. Dies kann etwa durch optische oder akustische Signale, wie beispielsweise textuelle Anweisungen oder Sprachnachrichten, geschehen.
Abseits der Architektur eines Assistenzsystems wird in dieser Arbeit maßgeblich Fokus auf die Entwicklungsphase solcher Systeme gesetzt. Im Gegensatz zu der Nutzungsphase, in der Daten kontinuierlich gesammelt, gefiltert, gespeichert und teilweise wieder aus den Datenbanken gelöscht werden, werden in der Entwicklungsphase Daten in Testläufen gesammelt und nach der Homogenisierung und Kopplung im Allgemeinen einmalig in Datenbanken hinterlegt.

Wie in Abbildung 3.1 angedeutet, werden in der Entwicklungsphase deutlich mehr Sensoren genutzt. Durch lange Testläufe wird dann eine Datenbasis geschaffen, aus denen eine effiziente und deutlich reduzierte Kombination von Sensoren für die Nutzungsphase gewonnen werden soll. Dies ist wichtig zur Reduzierung der Stromdatenmenge, um so nötige Hardwarekosten und die Kommunikation sensibler Daten zu mindern. Zum anderen ist eine Ausdünnung der Sensoren nötig für die Nutzbarkeit des tatsächlichen Assistenzsystems. Als Beispiel hierfür sei etwa auf das MARIKA-Projekt [22] verwiesen. In diesem wurde ein Assistenzsystem für mobile Pflegekräfte entwickelt, welches eine automatische Dokumentation des Pflegeprozesses erstellt. Hierbei ist eine sichere Erkennung der Aktionen und Handlungen des Pflegepersonals nötig. Um dies zu ermöglichen, wurden in der Entwicklungsphase Pflegekräfte mit Sensorplatinen an der dominanten Hand, dem oberen Rücken und der Hüfte, während der Arbeit ausgestattet. In [23] wurde zusätzlich gezeigt, dass wesentliche Aktionen bereits über Sensoren am Handgelenk identifiziert werden, welches eine deutliche Steigerung der praktischen Umsetzbarkeit und Reduktion nötiger Sensoren zur Folge hätte.

Die vorliegende Arbeit setzt nach der Homogenisierung der gesammelten Daten an und bezieht sich folglich auf die Datenverwaltungs- und der darauf aufbauenden Schicht zur Erkennung von Aktivitäten, Intentionen und Situationen.

Wie bereits beschrieben, können die zu speichernden Daten personenbezogen und sehr groß sein, so dass eine ungefilterte Kommunikation dieser ineffizient sein kann oder Aspekte der Datenprivatheit verletzt. Um diesen Problemen entgegenzuwirken, wird in dieser Arbeit untersucht, inwiefern parallele relationale Datenbanksysteme zumindest teilweise Funktionen der Erkennung via der Standardanfragesprache SQL intern umsetzen können. Eine detaillierte Darstellung und Diskussion des Konzepts wird in Kapitel 3 gegeben.

2.2 Aktivitäts- und Intentionserkennung

Die Basis für sinnvolle Hilfestellungen ist die korrekte Erfassung der aktuellen Situation und der wahrscheinlichen Intentionen der zu assistierenden Personen (Agenten). In den folgenden Abschnitten wird eine kurze Beschreibung über automatische sensorgestützte Aktivitäts- und Intentionserkennung gegeben. Hierbei werden klassische Abläufe, von der Datenerstellung bis zur eigentlichen Aktivitätsvorhersage, diskutiert und eine kurze Motivation für die Untersuchung von Hidden-Markov-Modellen in Kapitel 5 im Kontext von Assistenzsystemen gegeben. Für eine detailliertere Einführung in Aktivitäts- und Intentionserkennung sei auf die hier maßgeblich genutzte Literatur [24], [25] und [26] verwiesen.

2.2.1 Aktivitätserkennung

Die Aktivitätserkennung hat sich zu einer der Kerndisziplinen des Feldes der Mensch-Computer-Interaktion entwickelt [24]. Eines der wesentlichen Ziele ist es hierbei, Systeme Informationen über das Verhalten von Nutzern zu geben, um diese in der darauf folgenden Vorhersage bei geplanten künftigen Aktivitäten (ihren Intentionen) zu unterstützen. Mit Voranschreiten der technischen Entwicklung von Sensoren und Batterien wurden Untersuchungen zur Aktivitätserkennung von anfangs noch starren Umgebungen ab Ende der 1990-iger Jahre zu Setups mit am Körper getragenen Sensoren erweitert [24]. Dies hat zu einer beträchtlichen Expansion potenzieller Anwendungsgebiete geführt. Neben Office-Szenarien, wie etwa die Unterstützung von Meeting-Szenarios [27, 28], Szenarien in der Unterhaltungsbranche und anderen ist einer der wesentlichen Forschungsfelder der Gesundheits- und Pflegesektor. Als ein Forschungsschwerpunkt hat sich hierbei die Erkennung von Aktivitäten des Alltags, wie beispielsweise die Zubereitung und Einnahme gesunder Nahrung [15, 29, 30] oder dem Monitoring von Alzheimerpatienten [31], entwickelt. Kernherausforderungen bei der Entwicklung von Systemen zur Aktivitätserkennung wurden in [24] in drei wesentliche Gruppen aufgeteilt. Die erste umfasst hierbei grundlegende Herausforderungen, wie sie auch in allgemeinen Mustererkennungen existieren. Dazu zählen beispielsweise

- das akkurate Klassifizieren einer Aktivität, die verschiedene ähnliche Formen annehmen kann,
- das Unterscheiden von Klassen von Aktivitäten, die sich bezüglich deren Sensordarstellung ähneln und
- das Unterscheiden zwischen relevanten und unwichtigen Aktivitäten, beziehungsweise Rauschen (die NULL-Klasse; vgl. [24]).

Die zweite Gruppe beschreibt Herausforderungen die speziell bei Aktivitätserkennung für Menschen auftreten. Dies umfasst etwa

- eine klare, durch Sensoren gut detektierbare, Definition der Charakteristiken menschlicher Aktivitäten (bzw. der Klassen),
- das Trainieren von Systemen mit stark unterschiedlich frequent auftretenden Aktivitäten,
- die Erstellung von Sensordaten mit annotierten Zuständen („ground truth“; vgl. [24]), und
- die generelle Erstellung großer, qualitativ hochwertiger, standardisierter (und kostenintensiver) Sensordatenmengen für menschliches Verhalten.

Die letzte Gruppe beschreibt Herausforderungen, die konkrete Systeme handhaben müssen. Hierzu zählen etwa Ausfälle von Sensoren oder eine große Sensitivität solcher bei widrigen Umgebungsverhältnissen. Zudem balancieren vor allem Echtzeitsysteme einen Trade-Off bezüglich Genauigkeit, Rechenleistung und Latenz.
Eines der entscheidenden Aspekte dieser Szenarien ist die erfolgreiche Erkennung von Gesten: eine der wichtigsten nonverbalen Kommunikationsmöglichkeiten im Bereich der Mensch-Computer-Interaktion [32]. Gestenerkennung wurde vor allem durch die Entwicklung kostengünstiger und kleiner Beschleunigungssensoren ermöglicht. Hierbei kommen Sensorarmbänder oder Smartwatches (siehe etwa [23,32]) zum Einsatz, um den zu assistierenden Personen zu ermöglichen, ihre Hände frei zu nutzen. Die Erkennung von Gesten wird oft durch Hidden-Markov-Modelle umgesetzt, da diese komplexe zeitliche Beziehungen gut modellieren können und deren Verhalten gut verstanden wird [32]. Aus diesem Grund und wegen deren Eignung für die Intentionserkennung (siehe folgender Abschnitt) werden diese in Kapitel 5 näher eingeführt und als Basismodell in dieser Arbeit für die Überführung von Funktionalitäten der Aktivitätserkennung und -vorhersage in (parallelen) Datenbanksystemen genutzt.

Aktivitätserkennungspipeline

Im Folgenden wird eine klassische Unterteilung der wesentlichen Phasen von Systemen der Aktivitätserkennung beschrieben. Eine mögliche Architektur ist die in Abbildung 2.2 dargestellte Aktivitätserkennungspipeline (im Original: „Activity Recognition Chain“) aus [24]. Da die Aktivitätserkennung eines der Kernaspekte von Assistenzsystemen ist, ist die Pipeline eingebettet in die vorgestellte Architektur von Assistenzsystemen aus Abbildung 2.1 in Abschnitt 2.1.
So werden analog zur Pyramide in den ersten beiden Phasen Daten durch mehrere Sensoren erfasst, gekoppelt und homogenisiert. Dies beinhaltet auch das Löschen von Artefakten, die Umrechnung von Rohdaten in aussagekräftige physikalische Einheiten oder die Interpolation von Sensordaten zur Handhabung von Sensoren mit unterschiedlichen Abtastraten. Im Zuge des vorgestellten Forschungsprojektes konnte in [12, 13] im Kontext von Automobildaten (vgl. Abschnitt 9.4) gezeigt werden, dass solche Konvertierungen und Interpolationen auch in (objekt-)relationalen Datenbanksystemen effizient umgesetzt werden können.
Im nächsten Schritt, der Datensegmentierung, werden die vorverarbeiteten Daten so in zeitliche Intervalle aufgeteilt, dass diese mit großer Wahrscheinlichkeit wichtige Informationen über Aktivitäten beinhalten. Dieser Prozess ist äußerst komplex und hat wesentlichen Einfluss auf die Qualität des Erkennungsprozesses. In [24] werden drei wesentliche Ansätze (Sliding Window, energiebasiert und Nutzung externer Informationen und Sensoren) diskutiert, auf welche hier nicht näher eingegangen wird.
Nach der Segmentierung der Daten werden aus den Sensordaten aussagekräftige, charakteristische Kenngrößen in Vektorform abgeleitet: die Feature-Vektoren. Einer der wesentlichen Aspekte

dieses Schrittes ist die Reduzierung der zu handhabenden Datenmenge. Features können automatisch berechnet oder durch Expertenwissen hergeleitet werden. Im Allgemeinen ist es hierbei sinnvoll Features gemäß deren Einfluss auf einzelne Aktivitäten zu clustern [24]. Dieses Konzept der Nachbarschaft wird etwa in Abschnitt 8.1.3 bei der Partitionierung dünn besetzter Matrizen berücksichtigt. In der Forschung und Anwendung haben sich verschiedene Typen von Features, wie etwa einfache statistische Funktionen, etabliert. Für genauere Ausführungen zu diesen sei erneut auf [24] verwiesen. Eine Vergrößerung des Feature-Raums, also dem Hinzufügen linear unabhängiger Feature-Vektoren, führt im Allgemeinen dazu, dass mehr Trainingsdaten nötig werden. Hierdurch reichen viele Anwendungsfälle bereits bei verhältnismäßig niedrigen Dimensionen in den Big-Data-Bereich (vergleiche Abschnitt 2.4).

Die letzte Phase der Aktivitätserkennungspipeline umfasst das Trainieren des Modells oder der genutzten Modelle und dem eigentlichen Klassifizierungsprozess. Die Trainingsphase wird durch die Wahl der genutzten Machine-Learning-Modelle bestimmt. Die Parameter dieser werden aus der Menge der Feature-Vektoren und der zugehörigen annotierten Zuständen („ground truth“) berechnet. Die Klassifizierung kann in zwei wesentliche Phasen aufgeteilt werden. Im ersten Schritt werden bezüglich jedes Feature-Vektors den Aktivitätsklassen Konfidenzwerte (auch: „Scores“) zugeordnet. Für Bayessche-Ansätze (etwa Hidden-Markov-Modelle) entsprechen diese etwa Wahrscheinlichkeitswerten. Im zweiten Schritt werden die Scores genutzt um auf eine Aktivitätsklasse zu schließen. Klassischerweise wird hierfür der maximale Konfidenzwert genutzt, welcher im Falle von Bayesschen-Ansätzen demnach der wahrscheinlichsten Klasse entspricht. Neben der Klassifizierung können mit diesen Werten auch Aussagen darüber gegeben werden, inwiefern dem Erkennungsprozess zu trauen ist. Sind etwa alle Konfidenzwerte sehr gering, ist es wahrscheinlicher, dass eine Aktivität der NULL-Klasse beobachtet wurde. In [24] wird zusätzlich auf Literatur verwiesen, in der die Scores als Input für andere Inferenzmethoden genutzt werden, um komplexere Strukturen zu finden.

In der Entwicklungsphase von Systemen zur Aktivitätserkennung ist zudem die Evaluation der Qualität des Erkennungsprozesses ein wesentlicher und komplexer Bestandteil. In dieser werden meist Optimierungsprozesse zur Minimierung anwendungsbezogener Metriken genutzt. Für eine genauere Beschreibung verschiedener Formen dieser sei auf die Literatur verwiesen. Es gibt drei wesentliche Fehler bei der Erkennung von Aktivitäten: Eine Möglichkeit ist das Erkennen von Aktivitäten, die nicht tatsächlich in den Daten existieren. Eine Zweite ist es, dass relevante Aktivitäten übersehen werden und schließlich die Zuordnung einer erkannten Aktivität in eine falsche Klasse. In vielen Szenarien werden bei der Optimierung Metriken eingesetzt um „false negatives“ zu minimieren, auch wenn dies zu einem Anstieg der „false positives“ führt. Allerdings kann eine zu hohe Rate letzterer dazu führen, dass Nutzer das Vertrauen in das Erkennungssystem verlieren und deren Nutzung einstellen.

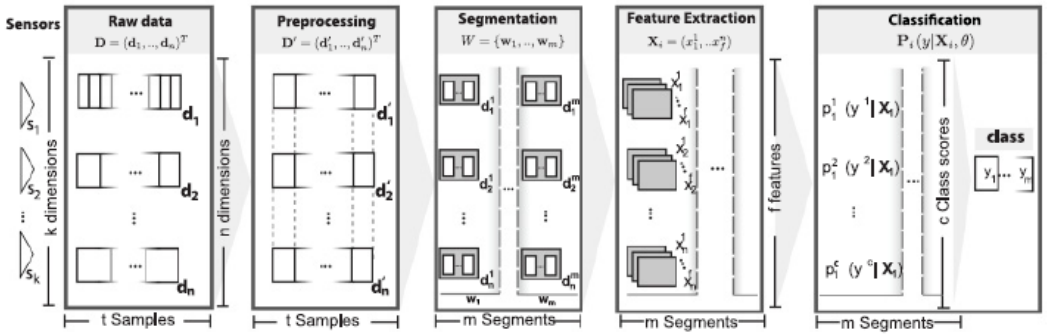


Abbildung 2.2: Struktureller Aufbau des Prozesses zur Aktivitätserkennung: Die Aktivitätserkennungspipeline (Im Original: „Activity Recognition Chain“). Grafik entnommen aus [24].

2.2.2 Intentionserkennung

Können Aktivitäten zu assistierenden Personen erfolgreich von einem System erkannt werden, ist es nötig, diese Handlungen so zu analysieren, dass das System die Intentionen des Nutzers proaktiv unterstützen kann. Intentionserkennung (im englischen „intention recognition“ oder auch „goal recognition“) wird in diesem Kontext verstanden als der Prozess des Ableitens der Intentionen von Agenten, basierend auf vorher beobachten Aktivitäten und deren Einfluss auf die Umgebung [25]. Dieses Feld ist verwandt mit dem der „plan recognition“, welches zusätzlich den Plan erkennt, den die beobachtete Person verfolgt, um deren Intention zu erfüllen. Der Unterschied ist hierbei, dass die Ziele der Intentionserkennung a priori nicht bekannt sind und auch wieder vorzeitig aufgegeben werden können. Beide Erkennungsansätze nutzen klassischerweise als Input Mengen von Intentionen, sowie pro Intention eine Menge von Plänen, wie diese erfüllt werden können. Als Beispiel hierfür sei etwa auf [26] verwiesen, in dem ein Plan für das Kochen von Tee und deren Überführung in Hidden-Markov-Modellen diskutiert wird. In [25] werden mögliche Umsetzungen dieser in zwei wesentliche Gruppen aufgeteilt: den Konsistenz- (im Original: „consistency approach“) und den probabilistischen Ansätzen. Bei Konsistenzansätzen wird untersucht, welche Intention „konsistent“ mit den beobachteten Aktivitäten ist, das heißt, ob Pläne für Intentionen existieren, die mit den Beobachtungen übereinstimmen. Problematisch an diesen Ansätzen ist, dass diese im Falle, dass die Beobachtungen mit Plänen zu verschiedenen Ansätzen korrespondieren, nur schwer zwischen diesen entscheiden können [25]. Dieses Problem kann in probabilistischen Modellen gelöst werden. Hier wird in diesem Fall die wahrscheinlichste Intention, basierend auf statistischen Nachweisen beziehungsweise im Modell eingebundener subjektiver Ansichten [25], gesucht. Solche Modelle werden typischerweise durch Bayessche Netze oder Hidden-Markov-Modelle umgesetzt. Für eine detailliertere Einführung in das Gebiet der Intentionserkennung und eine Diskussion ihrer Herausforderungen sei hier noch einmal auf [25]

und [26] verwiesen.
Wie bereits im vorigen Abschnitt beschrieben stützen wir uns hier auf die mehrfach erwähnten Hidden-Markov-Modelle, welche in [25], [24] und [26] als potenzielle und verbreitete Ansätze für Aktivitäts- und Intentionserkennung genannt werden.

2.3 Relationale Datenbanksysteme

Relationale Datenbanksysteme bilden die Klasse der kommerziell erfolgreichsten und verbreitetsten Datenbanksysteme [33]. Die theoretischen Grundlagen für diese wurden bereits in den 1970er Jahren gelegt und sind seitdem fester Bestandteil der Forschung und Industrie. Um eine bessere Einordnung und Diskussion dieser Systeme im Kontext von Assistenzsystemen zu geben, werden in diesem Abschnitt Basiskonzepte relationaler Datenbankensysteme in Kürze vorgestellt. Neben den theoretischen Grundlagen zur Datenrepräsentation und Datenverarbeitung durch Anfragesprachen (speziell SQL) wird zudem auf Konzepte verteilter Datenbanksysteme und klassischer Datenverteilungsstrategien eingegangen. Im Folgenden werden wesentliche Begriffe (paralleler) relationaler Datenbanken definiert. Diese und zugehörige Erläuterungen basieren (insofern nicht anders angegeben) auf Ausführungen in [33], [34], [35] und [36].

2.3.1 Relationenmodell

Die theoretische Grundlage relationaler Datenbanksysteme bildet das Relationenmodell. Dieses ist das Realisierungsmodell zur Implementierung von Datenbankentwürfen [33]. Es beinhaltet insbesondere Konzepte zur Strukturierung der Daten.
Vereinfacht gesagt, kann das Relationenmodell als Menge von Tabellen („Relationen“) mit einer Menge zusätzlicher Bedingungen angesehen werden. Die Werte einzelner Spalten gehören hierbei stets dem gleichen Datentyp an. Um dies formal darstellen zu können, werden zunächst die Begriffe des Universums, der Attribute und Domäne eingeführt.

Definition 1 (Universum, Attribut, Attributwerte, Domäne und Wertebereich [33])**.** Sei $\mathcal{U}$ eine nicht-leere, endliche Menge, das *Universum* der Attribute. Ein Element $A \in \mathcal{U}$ heißt *Attribut*. Sei $\mathcal{D} = \{D_1, \ldots, D_m\}$ eine Menge endlicher, nicht-leerer Mengen mit $m \in \mathbb{N}$. Die Elemente D_i ($i \in \{1, \ldots, m\}$) werden *Domäne* genannt. Es existiert eine total definierte Funktion dom: $\mathcal{U} \mapsto \mathcal{D}$. dom($A$) heißt der *Wertebereich* von A. Ein $w \in$dom(A) wird *Attributwert* genannt.

Mit dieser Grundlage ist es nun möglich, Relationen und deren Struktur formal zu definieren.

Definition 2 (Relationenschema, Relation und Tupel [33])**.** Eine Menge $R \subseteq \mathcal{U}$ über einem Universum $\mathcal{U}$ heißt *Relationenschema*. Eine *Relation* r über $R = \{A_1, \ldots, A_n\}$ (kurz $r(R)$) mit

$n \in \mathbb{N}$ ist eine endliche Menge von Abbildungen

$$t : R \mapsto \bigcup_{i=1}^{m} D_i$$

die *Tupel* genannt werden, wobei $t(A) \in \text{dom}(A)$ gilt. Dabei ist $t(A)$ die Restriktion der Abbildung t auf $A \in R$.

Im einfachsten Sinne ist dann eine Datenbank eine Menge von Relationen und ein Datenbankschema eine Menge der zugehörigen Relationenschemata.

Definition 3 (Datenbankschema, Datenbank [33])**.** Eine Menge von Relationenschemata $S := \{R_1, \ldots, R_p\}$ mit $p \in \mathbb{N}$ heißt *Datenbankschema*. Ein *Datenbankwert* (kurz: *Datenbank*) über einem Datenbankschema S ist eine Menge von Relationen

$$d := \{r_1, \ldots, r_p\},$$

wobei $r_i(R_i)$ für alle $i \in 1, \ldots, p$ gilt. Eine Datenbank d über S wird mit $d(S)$ bezeichnet, eine Relation $r \in d$ heißt *Basisrelation*.

Dieser bis hier etablierte Begriff der Datenbank bildet die Basis für die Datendarstellung und -ablage in relationalen Datenbanken. Jedoch wurde bis zu diesem Zeitpunkt nicht auf Beziehungen zwischen Attributen und Relationen oder Anforderungen an diese eingegangen. Eine der wesentlichsten Formen von Attributbeziehungen in Relationen sind die der identifizierenden Attributmengen und Schlüsseln.

Definition 4 (Identifizierende Attributmengen und Schlüssel [33])**.** Eine *identifizierende Attributmenge* für ein Relationenschema R ist eine Menge $K := \{B_1, \ldots, B_k\} \subset R$, sodass für jede Relation $r(R)$ gilt:

$$\forall t_1, t_2 \in r[t_1 \neq t_2 \Rightarrow \exists B \in K : t_1(B) \neq t_2(B)].$$

Ein *Schlüssel* ist ein bezüglich $\subset$ minimal identifizierende Attributmenge, *Primattribut* nennt man jedes Attribut eines Schlüssels. Ein *Primärschlüssel* ist ein ausgezeichneter Schlüssel.

Eine Verallgemeinerung solcher Bedingungen bilden die lokalen Integritätsbedingungen. Lokal bezeichnet in diesem Fall, dass die Bedingungen nur auf einzelnen Relationen definiert sind und nicht Relationen übergreifend auf Datenbankebene. Konzeptuell werden diese Bedingungen durch Abbildungen umgesetzt, durch welche die Gültigkeit einzelner Relationen überprüft werden können.

Definition 5 (Lokale Integritätsbedingungen [33])**.** *Lokale Integritätsbedingungen* $\mathcal{B}$ sind eine

Menge von Abbildungen

$$B \ni b : \{r \mid r(R)\} \mapsto \{\texttt{true}, \texttt{false}\},$$

die alle möglichen Relationen r zu einem Relationenschema R auf `true` oder `false` abbilden.

Neben den Schlüsselbedingungen sind etwa Einschränkungen des Wertebereichs einzelner Attribute, wie `NOT NULL`, klassische Beispiele für lokale Integritätsbedingungen. Mit diesen Einschränkungen lassen sich die Schema-Begriffe und die der Datenbank erweitern.

Definition 6 (Erweitertes Relationenschema, lokal erweitertes Datenbankschema und Datenbank [33])**.** Sei R ein Relationsschema und B lokale Integritätsbedingungen, dann heißt

$$\mathcal{R} := (R, \mathcal{B})$$

erweitertes Relationenschema. Eine Relation r über $\mathcal{R}$ muss dann alle lokalen Integritätsbedingungen erfüllen

$$\forall b \in \mathcal{B} : b(r) = \texttt{true}.$$

Eine Menge lokal erweiterter Relationenschemata

$$S := \{\mathcal{R}_1, \ldots, \mathcal{R}_p\}$$

mit $p \in \mathbb{N}$ heißen lokale erweitertes Datenbankschema. Eine Datenbank über S ist eine Menge von Relationen $d := \{r_1, \ldots, r_p\}$ (auch $d(S)$) mit $r_i(\mathcal{R}_i)$ für alle $i \in \{1, \ldots, p\}$. Eine Relation $r \in d$ wird *Basisrelation* genannt.

Als letzte Erweiterung des Datenbankkonzepts werden Bedingungen eingeführt, welche relationenübergreifend agieren. Ein klassisches Beispiel hierfür sind Fremdschlüsselbedingungen (siehe etwa [33]), die jedoch im Laufe der vorliegenden Arbeit keinen weiteren Einfluss besitzen und daher hier vernachlässigt werden.

Definition 7 (Globale Integritätsbedingungen, global erweitertes Datenbankschema [33])**.** Eine Menge von Abbildungen

$$\Gamma := \{\gamma \mid \gamma : \{d \mid d(S)\} \mapsto \{\texttt{true}, \texttt{false}\}$$

nennt man eine Menge von globalen Integritätsbedingungen für das Datenbankschema S. Das Tupel

$$\mathcal{S} := (S, \Gamma)$$

heißt dann global erweitertes Datenbankschema. Eine Datenbank $d(\mathcal{S})$ über $\mathcal{S}$ ist eine Datenbank

$d(S)$ die alle globalen Integritätsbedingungen erfüllt:

$$\forall \gamma \in \Gamma : \gamma(d) = \texttt{true}.$$

Aus Gründen der Übersichtlichkeit wird die Ausformulierung der globalen Erweiterung von Datenbanken und Schemata im weiteren Verlauf vernachlässigt.

2.3.2 Relationales Datenbank- und Datenbankmanagementsystem

Mit der im vorigen Abschnitt etablierten Datenbankdefinition kann nun der Begriff des relationalen Datenbanksystems eingeführt werden.

Definition 8 (Relationales Datenbanksystem und Datenbankmanagementsystem [33])**.** Ein *relationales Datenbanksystem* ist die Kombination von *Datenbank* und *Datenbankmanagementsystem* (vgl. Abbildung 2.3), wobei letzteres als die Gesamtheit des Softwaremoduls verstanden wird, welches für die Verwaltung der Datenbank genutzt wird.

Der Funktionsumfang des Datenbankmanagementsystem, beziehungsweise die Anforderungen an diesen, ist klassischerweise angelehnt an die 1982 von *Codd* etablierten neun Anforderungspunkte, die beispielsweise in [34] näher vorgestellt werden. Eine mögliche (und vereinfachte) Darstellung der Komponenten und deren Interaktionswege ist in Abbildung 2.4 abgebildet. Diese basiert auf dem standardisierten ANSI-Sparc-Entwurfsvorschlag von 1978. Dieser bildet, neben der hier nicht näher beleuchteten 5-Schichten-Architektur, eine der signifikantesten Systemarchitekturen von Datenbankmanagementsystemen. Eine der wesentlichsten Komponenten für die Effizienz von Datenabfragen und -analysen ist deren Optimierungsschnittstelle. Da deren Einfluss im Laufe der Arbeit nicht explizit thematisiert wird, jedoch implizit großen Einfluss besitzt, wird diese im Folgenden in Kürze beschrieben. Der Optimierungsprozess wird üblicherweise aufgeteilt in einen Teil für physische und einen für logische Optimierung [35]. Letztere transformiert Operatorbäume von Anfragen mittels logischer Optimierungsregeln in solche, die effizientere Abarbeitungsfolgen von Operatoren oder auch die Vermeidung redundanter Berechnungen ermöglichen. Diese macht Anfragen in gewisser Weise robust gegenüber der Art ihrer Formulierung. Jedoch wurde bereits in [10] im Kontext dieser Arbeit gezeigt, dass die Anfrageformulierung für manche Datenbanksysteme entscheidend sein kann, um eine effiziente Strategie zu wählen. Nach der logischen folgt die physische Optimierung, in der verschiedene Strategien zur Anfrageverarbeitung entwickelt werden, aus denen im späteren Verlauf die effizienteste zur Ausführung kostenbasiert oder heuristisch ausgewählt wird. Hierbei werden insbesondere Speichertypen, wie die in Abschnitt 4.1.1 beschriebenen Indexstrukturen, berücksichtigt. Die Entwicklung von Optimierern ist sehr komplex und entscheidend für die Performance von Datenbanksystemen. Viele Hersteller haben ihre Optimierungsschritte über Jahrzehnte stetig weiterentwickelt und an Interna ihrer

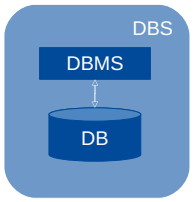


Abbildung 2.3: Beziehung zwischen Datenbank, Datenbankmanagementsystem und Datenbanksystem.

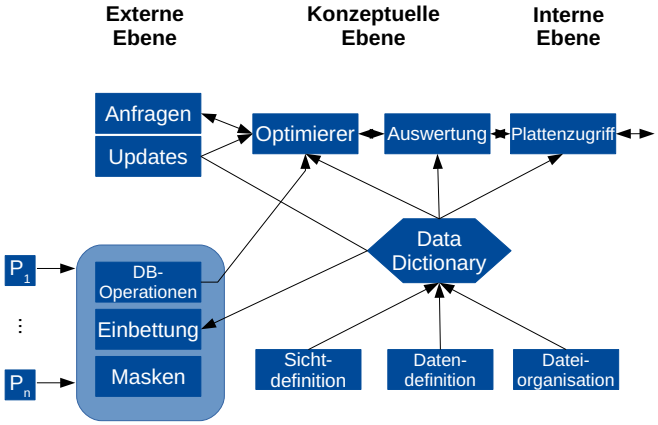


Abbildung 2.4: Vereinfachte Struktur eines Datenbankmanagementsystem. Abbildungsidee entnommen aus [34].

Datenbanksysteme angepasst. Demzufolge existieren teilweise entscheidende Unterschiede bei der Anfrageverarbeitung gleicher Anfragen in unterschiedlichen Datenbanksystemen.
Für eine Übersicht und Diskussion weiterer Architekturen, Funktionalitäten und Implementationsansätze von relationalen Datenbanksystemen sei auf [34] und [35] verwiesen.

2.3.3 Relationenalgebra

Neben den strukturellen Definitionen von Datenbanken wird zusätzlich ein Konzept für das Suchen und Ändern der Daten benötigt: die Anfragesprache. Basis für diese bilden Anfragemodelle, wobei im Kontext des Relationenmodells im wesentlichen zwischen Anfragealgebren und Anfragekalkülen unterschieden wird. Da letztere keinen weiteren Einfluss auf die folgenden Betrachtungen haben, sei für eine Einführung dieser auf [33] verwiesen. Die wohl verbreitetste Basis für Anfragesprachen auf relationalen Datenbanksystemen bildet die Relationenalgebra [33]. Diese stellt beispielsweise den konzeptuellen Kern der Anfragesprache SQL dar, welche im Folgeabschnitt 2.3.4 vorgestellt wird und ein Kernpunkt des Konzepts aus Kapitel 3 ist. Zudem wird in Abschnitt 7.1 eine erweiterte Relationenalgebra genutzt, um die Darstellbarkeit von Operatoren der linearen Algebra zu belegen. Dies wird insbesondere die Einbeziehung von „leichten“ relationalen Datenbankmanagementsystemen (Systeme mit begrenzter SQL-Funktionalität) motivieren, welche für sensornahe Verarbeitungen auf vergleichsweise schwachen Prozessoren genutzt werden können.
In der Relationenalgebra werden Relationen als abstrakte Datentypen mit einem auf diesen definierten Operatorensystem eingeführt. Eine Anfrage besteht aus einer Hintereinanderausführung

von Operationen dieses Systems. Solch ein System von unabhängigen Operatoren ist beispielsweise $\omega = \{\pi, \sigma, \bowtie, \beta, \cup, -\}$, welche im Folgenden definiert werden. Unabhängig bedeutet in diesem Kontext, dass sich keiner der Operatoren durch eine Kombination der anderen Operatoren darstellen lässt.

- Die Projektion $\pi_X(r)$ auf der Relation r mit Tupeln t und der Attributmenge X wird formal durch

  $$\pi_X(r) := \{t(X) \mid t \in r\}$$

  definiert. Es gilt hierbei zu beachten, dass das Ergebnis eine Menge ist und demnach mehrfach auftretende Ergebnisse nur einmalig dargestellt werden.

- Die Selektion σ mit der Bedingung F wird definiert durch

  $$\sigma_{\texttt{F}}(r) := \{t \mid t \in r \wedge F(t) = \texttt{true}\}.$$

  Die Bedingung F kann dabei die folgenden Formen besitzen:

  $$\texttt{Attribut} \circ \texttt{Konstante}$$

  wobei $\circ = \{=, \neq\}$ oder bei linear geordneten Wertebereichen auch $\circ \in \{\leq, <, \geq, >, =, \neq\}$ sein kann. Alternativ können auch Attributvergleiche

  $$\texttt{Attribut}_1 \circ \texttt{Attribut}_2$$

  vorgenommen werden. Die Attribute müssen in diesem Fall kompatible Wertebereiche besitzen. F kann auch mehrere Bedingungen enthalten, die logisch verknüpft sind mit $\vee, \wedge$ oder $\neg$.

- Der natürliche Verbund $\bowtie$ verbindet Tupel zweier Relation mit gleichen Attributwerten von gleichnamigen Attributen und ist formal durch

  $$r_1 \bowtie r_2 := \{t \mid t(R_1 \cup R_2) \wedge \exists t_1 \in r_1 : t_1 = t(R_1) \wedge \exists t_2 \in r_2 : t_2 = t(R_2)\}$$

  definiert.

- Die Umbenennung

  $$\beta_{B \leftarrow A}(r) := \{t' \mid \exists t \in r : t'(R - A) = t(R - A) \wedge t'(B) = t(A)\}$$

  ersetzt den Attributnamen A der Relation r durch B.

- Die Vereinigung zweier Relation r_1 und r_2 mit gleichem Schema R ist definiert durch

$$r_1 \cup r_2 := \{t \mid t \in r_1 \vee t \in r_2\}.$$

- Die Differenz zweier Relation r_1 und r_2 mit gleichem Schema R kann analog zur Vereinigung durch

$$r_1 - r_2 = \{t \mid t \in r_1 \wedge t \notin r_2\}$$

  definiert werden.

In der Literatur existieren auch weitere solcher Operatorsysteme, die bezüglich ihrer Ausdruckskraft äquivalent („relational vollständig“ vgl. [34]) sind.

2.3.4 Die Anfragesprache SQL

SQL (**S**tructured **Q**uery **L**anguage) ist eine standardisierte Datenbanksprache, welche von allen kommerziellen und freien relationalen Datenbanksystemen unterstützt wird [34]. Die Sprache findet ihren Ursprung in der 1974 entwickelten Anfragesprache SEQUEL (**S**tructured **E**nglish **QUE**ry **L**anguage), die auf dem von *Codd* eingeführten Relationenmodell, der Relationenalgebra und dem Relationenkalkül basiert. Im Zuge des Projektes System R, dem ersten Prototypen eines relationalen Datenbanksystems, wurde im Jahr 1976 die erste Erweiterung SEQUEL2 entwickelt. Dieses enthielt bereits mehr als der 1989 eingeführte Standard SQL-89 oder die 1992 veröffentliche Version SQL-92. Der Name SQL wurde erstmalig als Anfragesprache für die ersten kommerziellen relationalen Datenbanksystemen (Oracle, SQL/DS) eingeführt und setzte eine Untermenge der Funktionalitäten von SEQUEL2 um. Einer der wohl wichtigsten Kernpunkte über SQL ist deren Standardisierung, die es ermöglicht, über den Grenzen spezifischer Systeme hinaus in einer einheitlichen Datenbanksprache zu kommunizieren. In den Jahren 1982 bis 1986 wurde SQL zunächst vom *American National Standard Institute* (ANSI) genormt. Die *International Organization for Standardization* (ISO) nutzte diese als Basis und erstellte 1989 den als SQL-89 bezeichneten Standard. Dieser erweiterte eine revidierte Fassung von SQL-86 um eine Teilsprache zur Integritätssicherung [34]. Seitdem wurde der Standard in regelmäßigen Abständen erweitert. Zum aktuellen Zeitpunkt (12/2022) existieren die folgenden standardisierten Versionen: SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016 und SQL:2019 [37]. Neben den in Kapitel 3 diskutierten Vorteilen, die eine Standardisierung von SQL hervorgebracht hat, ist durch die frequente Erweiterung der Sprache eine Menge an Funktionalität entstanden, die keines der existierenden Datenbanksysteme vollständig erfüllt. Zwar unterstützen heutzutage nahezu alle Systeme eine definierte Untermenge von SQL-92 (Transition- oder Intermediate-Level), jedoch selbst dieser fast 30 Jahre alte Standard wird

von keinem Hersteller zu diesem Zeitpunkt vollkommen ermöglicht [34]. Zudem sind viele Teile der neueren Standards optional und nur in wenigen Systemen partiell umgesetzt [34]. Aus diesen Gründen wird der Begriff SQL oft Synonym für den eigentlichen Kern von SQL genutzt. Dieser entspricht im wesentlichen dem Transition-Level von SQL-92 und ist seit SQL:1999 stabil. Die Abdeckung des Kerns in relationalen Datenbanksystemen ist, wie bereits beschrieben, deutlich besser im Vergleich zu moderneren Erweiterungen im Standard.
Für eine detailliertere Beschreibung der einzelnen Standards sei aufgrund dessen Umfangs auf die Literatur verwiesen. Zur Veranschaulichung der Neuerungen ist in 2.1 eine Übersicht der Kernaspekte der Standards von SQL:1999 bis SQL:2016 dargestellt. Im Folgenden wird ein Überblick über den Aufbau von SQL und eine nähere Beschreibung des Kerns beziehungsweise der für diese Arbeit wichtigen Teile gegeben.

Tabelle 2.1: Schwerpunkte der wesentlichen SQL-Standards zwischen 1999 und 2016. Die Tabelle ist [34] entnommen.

Schwerpunkte der Standards	1999	2003	2006	2011	2016
Objektrelationale Konzepte	+	+	-	-	-
XML-Konzepte	-	+	+	-	-
OLAP/Data Warehouses	-	+	-	-	+
Data Mining (match_recognize)	-	-	-	-	+
Temporale Konzepte	-	-	-	+	-
Multimedia-Datentypen	+	-	-	-	+
NewSQL (SQL/JSON)	-	-	-	-	+

Aufbau SQL

SQL besteht aus den folgenden Teilen, die zuständig sind für verschiedene Aufgaben des Datenbankmanagementsystem:

- DDL (**D**ata **D**efinition **L**anguage) und VDL (**V**iew **D**efinition **L**anguage): Daten- und Sichtdefinitionsteil
- SSL (**S**torage **S**tructure **L**anguage): Definitionsteil für Dateiorganisationsformen und Zugriffspfade
- IQL (**I**nteractive **Q**uery **L**anguage): interaktive Anfragesprache
- DML (**D**ata **M**anipulation **L**anguage): Formulierung von Datenänderungsoperationen
- DBPL (**D**ata **B**ase **P**rogramming **L**anguage): Datenbankprogrammiersprache mit imperativen Konstrukte zur Definition von Prozeduren und Funktionen

Die Teile DDL, IQL und DML basieren maßgeblich auf dem Relationenmodell und der Relationenalgebra.
Der Anfrageteil IQL ist das offensichtliche Fundament für Datenselektion und -analysen in relationalen Datenbanken. Die Überführung von Methoden der Aktivitätserkennung und -vorhersage in Sequenzen von SQL-Anfragen ist eine der wesentlichen Untersuchungsschwerpunkte dieser Arbeit und wird maßgeblich in Kapitel 7 diskutiert. Eine grundlegende Beschreibung solcher Anfragen wird bereits im Folgeabschnitt gegeben. Für eine detaillierte Beschreibung von Klauseln und Operatoren in SQL sei auf die Literatur verwiesen.
Der Datendefinitionsteil DDL korrespondiert direkt mit dem Relationenmodell. Hier werden Operationen für das Erstellen und Löschen von Relationen und Integritätsbedingungen zur Verfügung gestellt. Für die folgenden Betrachtungen beschränken wir uns hierbei auf Basisoperationen (aus SQL-89), wie dem Erstellen von Relationen (`create table`) mit Standarddatentypen, Primärschlüsseln und einfachen Attributbedingungen, wie beispielsweise `NOT NULL` oder `check`-Klauseln für Attribute. Eine Diskussion über effiziente Relationenschemata für Matrizen und Vektoren wird in Kapitel 6 geführt.
Der Datenänderungsteil DML besteht aus den drei wesentlichen Operationen *Einfügen*, *Löschen* und *Ändern* von Tupeln in Basisrelationen. Ähnlich zu den DDL-Operationen werden in dieser Arbeit maßgeblich nur die grundlegendsten Operationen (`insert into`, `update`, `delete from`) zur Änderung von Daten genutzt.
Die Erstellung effizienter Zugriffspfade (auch „Indexstrukturen“ oder nur „Indexe“) (SSL) hat in vielen Fällen wesentlichen Einfluss auf die Performanz von Anfragen. Zwar wurde die Anweisung zur Erstellung von Indexstrukturen mit SQL-92 wieder aus dem Standard entfernt, jedoch werden diese von allen SQL-basierten relationalen Datenbanksystemen weiterhin unterstützt [34]. Eine genauere Einführung für die Umsetzung und Auswahl solcher Indexe wird in Abschnitt 4.1.1 gegeben. Zusätzlich wird eine anwendungsbezogene Untersuchung und Auswertung dieser in Abschnitt 7.4 durchgeführt. Es wird sich hierbei zeigen, dass die Nutzung von Indexstrukturen, abhängig von Daten und Methoden, wesentlich sein kann für effiziente Aktivitätsvorhersagen.
Erweiterte Konzepte, wie Sichten oder die Datenbankprogrammiersprache DBPL, werden in künftigen Betrachtungen nur einzeln einbezogen. Grund hierfür ist vor allem die hohe Variation in deren systemspezifischen Umsetzungen, beziehungsweise dem Fakt, dass solche Erweiterungen in einigen Systemen nicht unterstützt werden. Dies macht vor allem eine systemübergreifende Konzeption, wie sie in Kapitel 3 beschrieben ist, schwer. Aus diesen Gründen wird eine weitgehende Nutzung und nähere Beschreibung dieser ausgespart.

Anfrageteil IQL

Im Gegensatz zu klassischen imperativen Programmiersprachen (wie beispielsweise *C* oder *FORTRAN*) ist der Anfrageteil von SQL deklarativer Natur. Vereinfacht gesagt heißt dies, dass SQL-

Anfragen das zu lösende Problem beziehungsweise das gewünschte Ergebnis beschreiben, anstatt eine Sequenz durchzuführender Operationen zu definieren. Damit wird eine direkte Überführung von Machine-Learning-Algorithmen beziehungsweise im Allgemeinen von Verfahren des wissenschaftlichen Rechnens in SQL erschwert. Dies wird vor allem in den Kapiteln 6 und 7 näher diskutiert. Der Kern des Anfrageteils in SQL besitzt die Form

```
select Projektionsliste
from Relationsliste
[ where Bedingung ].
```

Eine Anfrage gibt als Ergebnis eine temporäre Relation zurück. Die in der `select`-Klausel aufgeführte *Projektionsliste* gibt das Schema der Ergebnisrelation an. Hierbei können neben einfachen Projektionen auch einfache tupelweise arithmetische Operationen oder Aggregatfunktionen (etwa das Summieren (`sum`) numerischer Attributswerte eines Attributs über mehrere Tupel) angegeben werden. In der `from`-Klausel wird die *Relationsliste* übergeben. Diese enthält alle Relationen, die für die Problembeschreibung von Bedeutung sind, insbesondere jene Relationen, aus denen Attributwerte selektiert werden. Die Relationen der Relationsliste können orthogonal durch Unteranfragen eingebunden und durch Verbundoperationen (`join on`) kombiniert werden. Eine einfache Trennung von Relationen durch Kommata entspricht hierbei dem kartesischen Produkt. Letzteres kann in der optionalen `where`-Klausel durch Verbundbedingungen in einen Verbund überführt werden. Zusätzlich sind hier Selektionsbedingungen, die auch geschachtelte Anfragen enthalten können, möglich. Als Beispiel seien die zwei Relationen *kunde* und *bestellung* einer Datenbank eines Warenhauses mit hier unspezifizierten Schemata betrachtet. Die folgende Anfrage gibt alle offenen Bestellungen je Kunde aus:

```
select k.id, b.bestellungsid
from kunde k join bestellung b on k.id=b.kundenid
where b.status='offen'
```

Die Aliasnamen „`k`“ und „`b`“ werden in diesem Fall der Übersicht halber eingeführt, sind für orthogonal eingeführte Relationen (temporäre Relationen, die explizit durch eine SQL-Anfrage erstellt worden sind) jedoch zwingend erforderlich. Neben dieser Basisstruktur existieren verschiedene weitere Klauseln in SQL. Wesentlich für die hier geführten Betrachtungen ist vor allem die `group by`-Klausel, welche die Gruppierung von Tupeln bezüglich gleicher Attributwertkombinationen von Attributsmengen im Kontext von Aggregatfunktionen ermöglicht. Beispielsweise berechnet die Anfrage

```
select abteilung, sum(gehalt) as personal_budget
from mitarbeiter
group by abteilung
```

die Personalkosten einzelner Abteilungen eines Unternehmens.
Es lässt sich leicht zeigen, dass SQL mächtiger als die Relationenalgebra ist. Eine Abdeckung der Operatoren der Relationenalgebra durch SQL-Klauseln und Operatoren ist in Tabelle 2.2 dargestellt. In dieser wird ebenfalls verdeutlicht, dass SQL mit einer Multimengensemantik arbeitet, das heißt dass es im Allgemeinen keine Duplikateliminierung in Relationen gibt. Die einfache Mengensemantik, die die Relationenalgebra inne hat, kann jedoch mit dem `distinct`-Operator in der `select`-Klausel erzwungen werden. Neben diesem Aspekt wird für die Abdeckung der Relationenalgebra ebenfalls die Vereinigung (`union`) von Relationen benötigt. Generell bietet SQL verschiedene Mengenoperationen, wie den Durchschnitt `intersect` oder die Differenz `except` an, um SQL-Anfragen zu verbinden.
Es gibt zahlreiche weitere Funktionalitäten (etwa rekursive Anfragen oder Window-Funktionen) die weit verbreitet in kommerziellen und freien Systemen vorhanden sind. Da jedoch eine vollständige oder umfangreiche Vorstellung von Operatoren und Funktionalitäten des Anfrageteils nicht Anspruch dieser Arbeit ist, beschränken wir uns hier darauf, aufkommende neue SQL-Operatoren und Funktionalitäten im Zuge ihrer Nutzung einzuführen.

Tabelle 2.2: Vergleich der Relationenalgebra mit dem SQL-Anfragekern. Tabelle aus [34].

Relationenalgebra	SQL
Projektion	**select distinct**
Selektion	**where** ohne Schachtelung
Verbund	**from**, **where**
	from mit **join** oder **natural join**
Umbenennung	**from** mit Tupelvariable
	as
Differenz	**where** mit Schachtelung
	except corresponding
Durchschnitt	**where** mit Schachtelung
	intersect corresponding
Vereinigung	**union corresponding**

2.3.5 Parallele relationale Datenbanksysteme

Werden die zu handhabenden Datenmengen und zugehörigen Anfragen zu groß und zeitaufwändig für zentrale Systeme, ist es in vielen Fällen nötig, verteilte Systeme einzusetzen. Für eine detaillierte Diskussion über Prinzipien verteilter Datenbanksysteme sei an dieser Stelle auf [36] verwiesen, welches die Grundlage für die folgenden Ausarbeitungen ist. Für die parallele Verarbeitung von Anfragen auf Datenbanksystemen können die folgenden Grundarchitekturen unterschieden werden:

- Multidatenbanksysteme,

- Verteilte Datenbanksysteme und
- Parallele Datenbanksysteme.

Multidatenbanksysteme (auch „föderierte Datenbanksysteme“) sind ein Verbund autonomer Datenbanksysteme (oder auch anderen Datenquellen), für die keine vorkonzipierte Strategie der Kooperation existiert. Jedes der verknüpften Datenbanksysteme fügt nur einen Teil seiner lokal vorhandenen Datenbanken der globalen Sicht bei. Außerdem können beispielsweise unterschiedliche Anfragesprachen je Knoten benötigt werden oder nur Abstufungen dieser zur Verfügung stehen. Die Kosten von Anfragestrategien sind, aufgrund der Diversität der Hardware und dem Optimierungspotenzial einzelner Systeme, nur schwer zu bestimmen.
Verteilte Datenbanksysteme sind eine Sammlung mehrerer, logisch miteinander verknüpfter Datenbanken, die über ein Netzwerk verteilt sind. Ein *verteiltes Datenbankmanagementsystem* ist dann ein Softwaresystem, welches das Management der verteilten Datenbank ermöglicht und zusätzlich die Verteilung für die Nutzer transparent umsetzt. Die einzelnen Systeme sind im Allgemeinen heterogen, das heißt, dass Leistungs- und Speichervermögen oder auch die genutzten Betriebssysteme oft unterschiedlich sind. Die Kommunikation der einzelnen Systeme geschieht ausschließlich über Netzwerkverbindungen (klassischerweise TCP). Der Übergang von verteilten Systemen zu *parallelen Datenbanksystemen* ist vergleichsweise fein. Parallele Datenbanksysteme sind Multiprozessorsysteme, das heißt, dass im Allgemeinen eine Kommunikation auch über geteilten Haupt- oder Festplattenspeicher geschieht. Es existieren jedoch auch Architekturen, welche ausschließlich über das Netzwerk kommunizieren. Der größte Unterschied zu verteilten Systemen liegt in diesem Fall vor allem in dem Aufbau der einzelnen Knoten. In parallelen Datenbanken werden, im Gegensatz zu verteilten Datenbanken, oftmals identische Prozessoren und Speicherkomponenten genutzt. Zusätzlich werden diese von einer bis mehreren Instanzen des gleichen Betriebssystems verwaltet. Aufgrund dieser Homogenität ist die Lastbalancierung und Konzeption von Optimierungen leichter möglich. Die Architektur paralleler Datenbanksysteme ist auch durch die Anforderungen der zu verarbeitenden Anfragen motiviert. So sind diese Systeme oft ausgelegt für die Verarbeitung weniger, aber sehr kostenintensiver Analyse-Anfragen (vgl. etwa OLAP-Anfragen/Data Warehouses [38]). Aus diesem Grund wird in der Konzeption aus Kapitel 3 und Architekturdiskussion aus Abschnitt 6.1 die Verwendung paralleler Systeme dargestellt. Im Folgenden werden wir hierfür näher auf den Aufbau dieser Systeme und Strategien zur Parallelisierung von Anfragen eingehen.

Architekturen paralleler Datenbanksysteme

Es werden im wesentlichen drei Architekturen von parallelen Systemen unterschieden: *Shared-Memory*, *Shared-Disk* und *Shared-Nothing*. Wie in Abbildung 2.5 dargestellt, teilen sich Prozessoren im *Shared-Memory*-Ansatz den Zugriff auf Festplatten und Hauptspeicher mittels schnel-

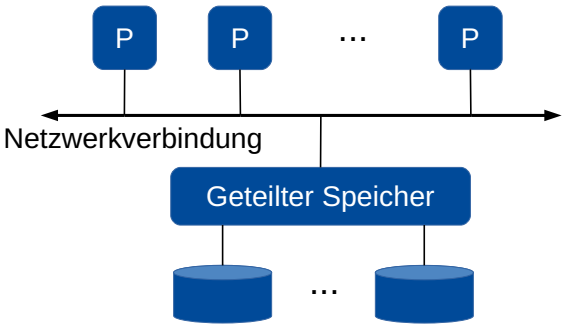


Abbildung 2.5: Shared-Memory Architektur. „P“ symbolisiert Prozessoren. Abbildungsidee entnommen aus [36].

ler Netzwerkverbindung. Zudem werden alle Knoten durch eine einzige Betriebssysteminstanz verwaltet. Der wesentliche Vorteil dieses Vorgehens besteht in seiner konzeptuellen Einfachheit: Da Meta- und Kontrollinformationen geteilt werden, ist die Implementierung ähnlich dem nicht-paralleler Systeme. Zudem ist die Lastbalancierung vergleichsweise unkompliziert. Neue Aufgaben werden den am wenigsten ausgelasteten Prozessoren zugewiesen. Die spezielle Verbindung der Prozessoren birgt jedoch auch drei wesentliche Nachteile. Zum einen wird spezielle Netzwerkhardware, die deutlich teurer sind als klassische Verbindungen, benötigt. Zudem ist dieser Ansatz nur begrenzt skalierbar, da mit jedem zusätzlichen Prozessor das Konfliktpersonal bei Plattenzugriffen erhöht wird. Abschließend ist eine niedrige Erreichbarkeit zu vermerken. Sollte ein Speicherfehler auftreten, sind womöglich mehrere Prozessoren nicht mehr in der Lage auf Daten zuzugreifen. Diese Probleme umgeht (zumindest teilweise) der in Abbildung 2.6 dargestellte *Shared-Disk*-Ansatz. In diesem hat jeder Prozessor seinen eigenen Hauptspeicher zur Verfügung und wird durch ein eigenes Betriebssystem verwaltet. Die Festplatten werden, wie bei Shared-Memory-Szenarien, über ein Netzwerk geteilt. Im Allgemeinen sind die nötigen Hardwarekosten dieser Systeme günstiger, die Erreichbarkeit und die Skalierbarkeit besser und die Lastbalancierung ähnlich einfach. Im Vergleich zum Shared-Memory-Ansatz ist die nötige Software des Managementsystems jedoch komplexer. So müssen beispielsweise, aufgrund der autonomen Hauptspeicher und der geteilten Festplatte, „lock manager“ implementiert werden, um Update-Konflikte zu vermeiden. Ferner kann der geteilte Zugriff auf dem Sekundärspeicher auch zu Performance-Problemen führen. Zum einen kann es bei parallelen Zugriffen auf gleichen Bereichen zu langsamen Zugriffszeiten kommen und zum anderen kann das Aufrechterhalten von einem konsistenten Cache zu hohen Kommunikationskosten führen.

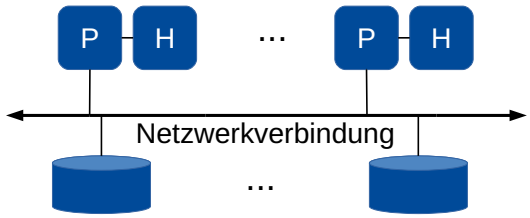


Abbildung 2.6: Shared-Disk Architektur. „P“ symbolisiert Prozessoren. „H“ symbolisiert Hauptspeicher. Abbildungsidee entnommen aus [36].

In der *Shared-Nothing*-Architektur, dargestellt in Abbildung 2.7, hat jeder Prozessor exklusiven Zugriff auf einen eigenen Haupt- und Sekundärspeicher. Zudem wird jeder Knoten durch sein eigenes Betriebssystem verwaltet. Da diese Knoten unabhängig voneinander sind und nur über ein Netzwerk kommunizieren, ähneln genutzte Techniken zur Balancierung und Anfrageverarbeitung dem der verteilten Datenbanksysteme. Die Architektur ist bedingt durch die Unabhängigkeit der Knoten, sowie deren einfache Anbindung, leicht skalierbar und mit vergleichsweise günstiger Hardware umsetzbar. Knoten können in diesem Ansatz auch örtlich entfernt sein. Zudem kann eine hohe Datenerreichbarkeit gewährleistet werden, speziell durch den Einsatz gezielter Replikationsstrategien, da Plattenfehler nur die jeweiligen Knoten beeinträchtigen. Negativ wirkt sich der Aufbau auf die Komplexität der Systemverwaltung aus. Die hohe Anzahl an Knoten erhöht den Kommunikationsaufwand zwischen den einzelnen Knoten stark. Zudem ist die Lastbalancierung schwieriger, da die Effizienz von Anfrageverarbeitungen maßgeblich von der Partitionierung abhängt. Zusätzlich wird nach einer nachträglichen Erweiterung oder Verminderung der Knotenanzahl eine komplette Repartitionierung voll verteilter Relationen benötigt.

Zudem existieren verschiedene hybride Architekturen (beispielsweise NUMA) basierend auf den drei präsentierten. Für eine detailliertere Diskussion dieser sei erneut auf [36] verwiesen. Eine Übersicht zu Vor- und Nachteilen der drei wesentlichen Ansätze ist in Tabelle 2.3 dargestellt.

Als zu favorisierenden Ansatz für skalierbare Datenbankszenarien hat *Stonebraker* bereits im Jahr 1985 in [39] die Shared-Nothing-Architektur bezeichnet. Mit Erhöhung der Datenmengen im moderneren Big-Data-Kontext wird hierbei der Aspekt der Skalierbarkeit noch entscheidender als zu jener Zeit. Moderne skalierbare Parallelisierungsframeworks, wie das in Abschnitt 4.3.1 vorgestellte MapReduce, verarbeiten große Datenmengen vor allem durch Nutzung vieler

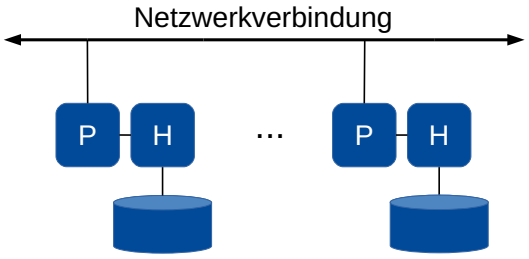


Abbildung 2.7: Shared-Nothing Architektur. „P“ symbolisiert Prozessoren. „H“ symbolisiert Hauptspeicher. Abbildungsidee entnommen aus [36].

Tabelle 2.3: Übersicht über Schlüsselaspekte der drei wesentlichen Architekturen von parallelen Datenbanksystemen. Eine detailliertere Aufschlüsselung ist in [39] zu finden.

Architektur	Shared-Memory	Shared-Disk	Shared-Nothing
Kosten	Hoch	Niedrig	Niedrig
Implementierungskomplexität	Niedrig	Hoch	Hoch
Erweiterbarkeit	Schwer	Mittel	Einfach
Erreichbarkeit	Niedrig	Mittel	Hoch
Lastbalanzierung	Einfach	Einfach	Schwer

kostengünstiger („Commodity Hardware“) Knoten. Aus diesen Gründen wird die Verwendung von Shared-Nothing-Systemen im vorgestellten Assistenzsystem-Kontext notwendig.

Parallele Anfrageverarbeitung

Um den Durchsatz oder die Latenzzeit von Anfragen zu verbessern, werden im Kontext verteilter und paralleler Datenbanken zwei wesentliche Strategien unterschieden: die *Inter-* und *Intraanfrageparallelisierung*, wobei letztere weiter aufgeteilt werden kann in *Inter-* und *Intraoperatorparallelisierung.*

Wie in Abbildung 2.8 dargestellt, werden bei der Interanfrageparalleisierung komplette Anfragen einzelnen Prozessoren zugewiesen und abgearbeitet. Klassischerweise sind solche Anfragen vergleichsweise günstig zu berechnen, so dass eine Verarbeitung auf einem einzelnen Prozessor zeitnah möglich ist.

Bei der Interoperator-Parallelität hingegen werden mehrere Operatoren von Anfragebäumen einer Anfrage parallel durch mehrere Prozessoren verarbeitet (vgl. Abbildung 2.9), während bei der Intraoperator-Parallelität aus Abbildung 2.10 ein einziger Operator durch mehrere Prozes-

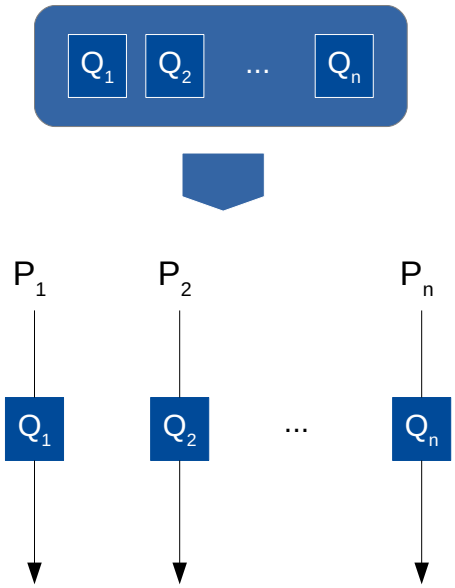


Abbildung 2.8: Schematische Darstellung von Interanfrage-Parallelisierung. Die Anfragen $Q_1, Q_2, \ldots, Q_n$ werden zeitgleich von den verschiedenen Prozessen $P_1, P_2, \ldots, P_n$ verarbeitet.

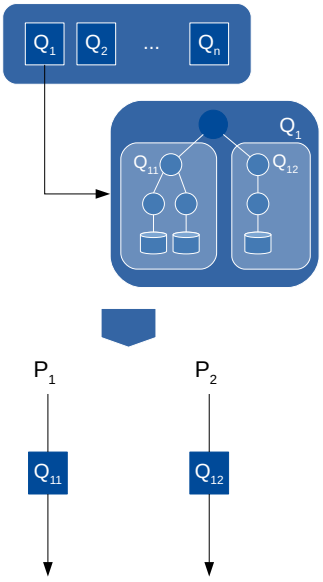


Abbildung 2.9: Schematische Darstellung von Interoperator-Parallelisierung.

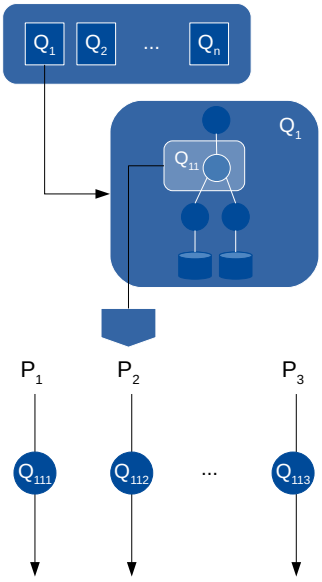


Abbildung 2.10: Schematische Darstellung von Intraoperator-Parallelisierung.

soren parallel berechnet wird. Hierbei arbeiten die einzelnen Prozessoren auf Untermengen der Input-Tupel. Da im Zuge dieser Arbeit vorrangig Intraoperator-Parallelität genutzt wird, sei eine detaillierte Beschreibung von Interoperator- und Interanfrage-Parallelität hier ausgespart. Wie in Kapitel 8 näher beschrieben wird, werden in den hier betrachteten Szenarien vor allem Anfragen parallelisiert, deren Berechnungen tupelweise arithmetische Funktionen und Aggregationsfunktionen nutzen. Letztere werden in diesem Kontext in drei wesentliche Gruppen aufgeteilt: algebraische, distributive und holistische Funktionen. Die folgende Definition basiert auf Ausführungen in [40].

Definition 9 (Distributive, algebraische und holistische Aggregatfunktion)**.** Sei ein Datensatz P in n Partitionen p_i $(i = 1, \ldots, n)$ aufgeteilt. Eine Aggregatfunktion F heißt distributiv, wenn eine weitere Aggregatfunktion G existiert, sodass das Ergebnis $F(P) = G(F(p_1), F(p_2), \ldots, F(p_n))$ ist. Hierbei bezeichnet $F(P)$ die nicht verteilte Auswertung des Datensatzes P und $G(F(p_1), F(p_2), \ldots, F(p_n))$ die Anwendung der Aggregatfunktion auf die Teilaggregate $F(p_i)$ mit $i = 1, \ldots, n$.
Ein Beispiel hierfür ist etwa die Aggregatfunktion **count**. Dieses kann verteilt berechnet werden durch die Summierung der einzelnen Partitionsgrößen. Zudem sind ebenfalls die Standardaggregate **sum**, **min** und **max** distributiv.
Algebraische Aggregatfunktionen sind algebraische Funktionen mit $M \in \mathbb{N}$ Argumente, welche distributive Aggregatfunktionen sind.
Als Beispiel sei etwa der arithmetische Durchschnitt **avg** betrachtet, welcher durch die Kombination **sum/count** bestimmt werden kann.
Aggregatfunktionen die nicht algebraisch und nicht distributiv sind heißen holistisch. Ein Beispiel hierfür ist der Median.

Wie sich bei der Analyse der Basismethoden von Hidden-Markov-Modellen in Kapitel 5 und deren Überführung in SQL im Anhang B.7 zeigen wird, sind in dem hier betrachteten Rahmen vor allem der Einsatz distributiver Aggregatfunktionen von Bedeutung.

Partitionierung und Replikation

Die Güte der Intraoperator-Parallelisierung in parallelen Datenbanksystemen (mit Shared-Nothing-Architektur) hängt maßgeblich von der Partitionierung der Daten ab. Partitionierung bezeichnet hierbei die Aufteilung einer Relation in disjunkte Partitionen und Speicherung dieser auf verschiedenen Festplatten [35]. Es wird im wesentlichen unterschieden zwischen der *vertikalen* und der *horizontalen Partitionierung*. Bei ersterer werden Relationen spaltenweise (beziehungsweise attributsweise) auf verschiedene Knoten gespeichert. Zur Rekonstruktion der ursprünglichen Relation ist es hierfür nötig, dass in jeder der einzelnen Partitionen ein Schlüsselattribut enthalten ist. Wie sich bei der Suche nach einer geeigneten Datenrepräsentation

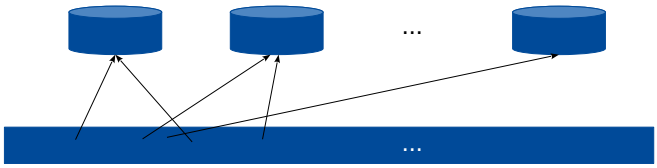


Abbildung 2.11: Round-Robin-Partitionierung. Abbildungsidee entnommen aus [36].

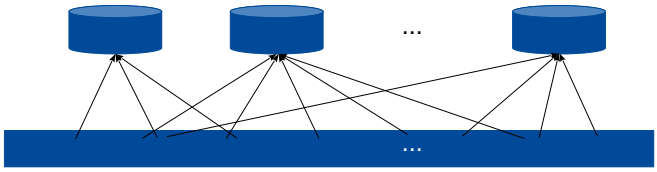


Abbildung 2.12: Hash-Partitionierung. Abbildungsidee entnommen aus [36].

von Matrizen und Vektoren in Kapitel 6 zeigen wird, ist diese Form ungeeignet für die in dieser Arbeit betrachteten Szenarien. Konträr zur vertikalen Partitionierung werden im horizontalen Absatz Relationen tupelweise auf Knoten aufgeteilt, das heißt jedes Tupel wird eindeutig einer Partition zugeordnet. Drei wesentliche Strategien zur Umsetzung solch einer Verteilung sind

- die Round-Robin-Partitionierung,
- die Bereichspartitionierung und
- die Hash-Partitionierung.

Diese werden im späteren Verlauf in Abschnitt 8.1 genutzt, um geeignete Strategien für die Intraoperator-Parallelisierung von Grundoperationen der linearen Algebra umzusetzen. Für eine kurze Beschreibung der drei Strategien sei n die Anzahl der gewünschten Partitionen. In der in Abbildung 2.11 dargestellten Round-Robin-Partitionierung wird das i-te Tupel der Partition $i \bmod n$ zugeordnet. Die Datenmenge wird in dieser Strategie optimal verteilt, was eine theoretisch optimale Beschleunigung des Lesevorgangs einer kompletten Relation ermöglicht. Allerdings wird in diesem Fall kein logischer Zusammenhang oder keine Gruppierung zwischen Tupeln, wie etwa eine Clusterbildung nach numerischen Bereichen von Schlüsselattributen oder ähnliche, genutzt. Dies kann im schlimmsten Fall bei Gruppierungsanfragen zur vollständigen

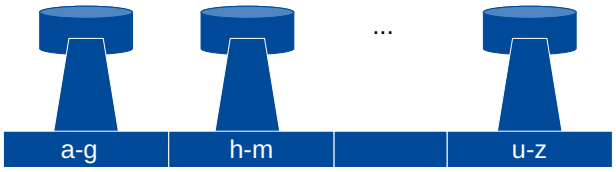


Abbildung 2.13: Range-Partitionierung. Abbildungsidee entnommen aus [36].

Online-Repartitionierung führen. Die Bereichspartitionierung adressiert dieses Problem. Sie ist die wohl verbreitetste Form der horizontalen Partitionierung [35]. Wie in Abbildung 2.13 dargestellt, wird in dieser jeder Partition ein Intervall des tatsächlich genutzten Wertebereichs eines Zielattributs (oft Schlüsselattribut) zugeordnet. Eingehende Tupel werden daraufhin den Intervallen entsprechend eingeteilt. Diese Strategie kann auch auf mehrere Attribute ausgeweitet werden. Dem Namen entsprechend können vor allem Anfragen über Bereiche oder auch exakte Werte der Partitionierungsattribute effizient verarbeitet werden. Allerdings kann die feste Bereichsdefinition schnell zu einer großen Variation der Partitionsgrößen führen, welches im Allgemeinen zu schlechter Lastbalancierung führt. Die Hash-Partitionierung aus Abbildung 2.12 verteilt Tupel nach einer Hash-Funktion auf die einzelnen Partitionen bezüglich eines oder mehrerer Attribute. Dies ermöglicht etwa effiziente Anfragen gruppierter Aggregationen, führt jedoch im Allgemeinen zu einem Aufbrechen von Clusterungen [35]. Die Lastbalancierung der Daten ist abhängig von der Hash-Funktion und den zu speichernden Daten.

Neben diesen Strategien ist zu beachten, dass eine Partitionierung vergleichsweise kleiner Relationen nicht sinnvoll ist und zu unnötigem Kommunikationsaufwand der einzelnen Knoten führt. In diesen Fällen ist etwa eine Replikation der Relationen auf jedem Knoten sinnvoll.

2.4 Big Data

Big Data ist ein noch vergleichsweise junger und nicht eindeutig definierter Begriff. Dieser wird meist im Kontext von Analysen auf großen, heterogenen und fehlerbehafteten Datenbeständen genutzt [41]. Erste Erwähnungen sind bereits in den neunziger Jahren zu finden [42], wobei moderne Interpretationen und Definitionen in der Literatur maßgeblich ab den 2010er Jahren entwickelt wurden. Klassischerweise wird Big Data durch mehrere charakterisierende (im Englischen mit „V“ beginnende) Schlagworte beschrieben und kategorisiert. Abhängig von der Literatur sind dies zwischen drei [43] und fünf verschiedene Aspekte [44], wobei der ursprüngliche Kern aus Volume, Variety und Velocity besteht. Diese können wie folgt beschrieben werden:

- **Volume**: Es werden im Kontext moderner Hardware große Datenmengen generiert, verarbeitet, analysiert und gespeichert.
- **Variety**: Eintreffende Daten sind im Allgemein heterogen und teilweise unstrukturiert, welches etwa für klassische relationale Datenbanksysteme eine Herausforderung darstellt [43].
- **Velocity**: Daten in Big-Data-Anwendungen haben einerseits eine hohe Erzeugungsrate und andererseits den Bedarf, diese auch sehr schnell - in Echtzeit - auswerten zu müssen: man spricht hier auch von Stromdatenverarbeitung.

Als zusätzlichen Aspekt führen verschiedene Quellen den Begriff „Veracity“, die „Richtigkeit“ der Daten, ein:

- **Veracity**: Erhaltene Daten können ungenau, fehlerhaft oder sogar fehlend sein. Darunter fallen neben Messfehlern auch Rauschen und Inkonsistenzen.

Abseits dieser datenbasierten Kriterien wird in wirtschaftlich orientierten Abhandlungen teilweise der Begriff „Value“ eingebracht:

- **Value**: Die Kosten von Implementation und Infrastruktur sollen einen deutlichen Mehrwert für Unternehmen einbringen. [44]

Dieser Aspekt wird hier nicht weiter verfolgt, zeigt jedoch den engen wirtschaftlichen Anwendungsbezug von Big Data. Entscheidungen, die Konzerne früher basierend auf Spekulationen treffen mussten, können heutzutage durch datengetriebene Analysen übernommen werden [41]. Mit den parallel laufenden Entwicklungen im Kontext des Internet of Things hat dies dazu geführt, dass Big Data oftmals für die Personalisierung von Anwendungen genutzt wird. So hat eine Studie aus dem Jahr 2012 von IBM und der Universität Oxford ergeben, dass 49% aller Befragten die Analyse von Konsumenten als wichtigstes Ziel im Bereich Big Data empfinden [41]. Beispielsweise versuchen große Einzelhandelskonzerne möglichst viele Informationen von Kunden zu sammeln. Mittels dieser werden dann, auf den jeweiligen Webseiten, Angebote an die aktuellen persönlichen Bedürfnisse in Echtzeit angepasst. Hierfür nennt IBM als Datenquellen etwa Social Media, Suchverläufe, Geodaten, aber auch Überwachungsvideos aus Warenhäusern und viele mehr.
Konträr zu den wirtschaftlichen Chancen dieser Ansätze ist es leicht nachzuvollziehen, dass die oftmals Cloud-basierten Umsetzungen zahlreiche Angriffspunkte für Aspekte der Datenprivatheit und des -schutzes bergen. Dies wurde beispielsweise im Kontext smarter Umgebungen in [45] diskutiert. Hier wurden zudem Assistenzsysteme als gute Beispiele für Big-Data-Anwendungen kategorisiert, da die vier Kerncharakteristiken großen Einfluss in der Systemarchitektur besitzen. Viele unterschiedliche (*Variety*) und frequent abgetastete Sensoren (*Velocity* und *Volume*)

sammeln Daten, die teilweise auch verrauscht sein können (*Veracity*). Zudem müssen in der Nutzungsphase diese in Echtzeit ausgewertet werden (*Velocity*) und in der Entwicklungsphase in großen Mengen gespeichert werden (*Volume*).
Das hier zu untersuchende Konzept aus Kapitel 3 zur Unterstützung von Assistenzsystemen setzt, wie bereits in Abschnitt 2.1 beschrieben, nach der Homogenisierung von Sensordaten an. Der Fokus der vorliegenden Arbeit liegt daher maßgeblich auf der Verarbeitung, Analyse und Speicherung großer Datenmengen (*Volume*).
Für die Handhabung solcher, wurden in den letzten Jahren in der Forschung und Industrie im Zuge von Big Data vor allem leicht skalierbare Parallelisierungsframeworks für Commodity-Hardware entwickelt. Der wohl bekannteste Vertreter hierfür ist das, in Abschnitt 4.3.1 näher beschriebene, MapReduce-Programmiermodell und das hierzu assoziierte System Apache Hadoop. In Kapitel 9 wird die in dieser Arbeit entwickelte datenbankbasierten Umsetzung evaluiert und dessen Ergebnisse mit denen des Parallelisierungsframework Apache Spark (siehe Abschnitt 4.3.2) verglichen. Letzteres baut auf Apache Hadoop auf und kann als Weiterentwicklung des MapReduce-Modells ausgelegt werden.

Kapitel 3

Konzept und wissenschaftliche Problemstellung

Mit der Etablierung wesentlicher kontextrelevanter Begrifflichkeiten im Kapitel 2, wird im Folgenden ein Framework zur Unterstützung der Entwicklungs- und Nutzungsphase von Big-Data-Assistenzsystemen vorgestellt. Hierfür werden zunächst wesentliche Anforderungen an solche Systeme zusammengeführt und mittels dieser ein datenbankbasiertes Konzept entwickelt. Nach einer Diskussion von Vorteilen und Herausforderungen, wird abschließend der weitere Verlauf der Arbeit motiviert.

3.1 Anforderungsanalyse

Ziel ist es, Entwicklern großer Assistenzsysteme zu unterstützen. Hierbei wird angenommen, dass die vorliegenden Daten und die darauf basierenden Machine-Learning-Modelle im Big-Data-Bereich liegen. Wie in Abschnitt 2.1 beschrieben wurde, wird hierbei nach der Homogenisierung der Sensordaten, das heißt ab der Schicht für verteilte Datenverwaltung der Assistenzsystempyramide aus Abbildung 2.1, angesetzt. Die aufgenommenen, angepassten und zusammengeführten Sensordaten sind persistent verteilt hinterlegt (etwa in einem parallelen Datenbanksystem) und sollen nun genutzt werden, um Modelle zur Aktivitätserkennung und -vorhersage zu lernen und anzuwenden. Aufgrund der Größe der Daten kann angenommen werden, dass die Berechnungen und Analysen mit klassischen hauptspeicherbasierten Ansätzen auf einzelnen Rechenknoten nicht durchgeführt werden können. Eine massive Parallelisierung der Methoden der Erkennungsschicht wird benötigt. Klassischerweise werden diese mit Big-Data-Umgebungen, wie Apache Hadoop oder Apache Spark (vgl. Abschnitt 4.3.2), umgesetzt. Da der in der Einleitung beschriebene Performance-Vergleich von Stonebraker et. al. in [3] gezeigt hat, dass parallele Datenbanksysteme MapReduce-Ansätze selbst in ihren vermeintlich klassische Anwendungsgebieten übertreffen

können, kann eine Untersuchung von der Berechnung von Machine-Learning-Modellen in parallelen relationalen Datenbanksystemen motiviert werden. Für solch ein Framework werden zunächst Kernaspekte beschrieben, um eine bessere Einordnung des danach folgenden Konzepts zu ermöglichen.

Datenbankinterne Analysen und Filterung

Der Pushdown von Analyseoperationen und Lernalgorithmen von Machine-Learning-Modellen in parallele (oder verteilte) Datenbanksysteme erlaubt eine Zusammenführung der Erkennungs- und Datenverwaltungsschicht der Assistenzpyramide aus Abbildung 2.1. Neben der zu untersuchenden, potenziell guten Performance im Big-Data-Bereich wird zusätzlich die Datenkommunikation zu einer externen Berechnungsschnittstelle eingespart. Dies führt zu einer zusätzlichen Beschleunigung des Berechnungsprozesses und ist ein entscheidender Aspekt im Kontext der *Datensicherheit*. So regelt das Datenbankmanagementsystem etwa den autorisierten Zugriff auf sensible Daten. Da diese zudem nicht an externe Berechnungsschnittstellen übertragen werden müssen, wird die Anzahl potenzieller Angriffspunkte auf dem Kommunikationsweg minimiert. Zusätzlich ermöglicht die Nutzung relationaler Datenbanksysteme weitere wertvolle Funktionalitäten, wie die über Jahrzehnte weiterentwickelten logischen und physischen Optimierungstechniken, Nutzung von Indexstrukturen für schnelle selektive Datenzugriffe (diese werden in Abschnitt 4.1.1 näher beschrieben), Transaktionskonzepte und weitere. Schließlich spart das potenzielle Auslassen zusätzlicher Analysesoftware ebenfalls eine Wartung letzterer aus, welches die Fehleranfälligkeit des gesamten Software-Stacks (etwa Abhängigkeitsprobleme aufgrund von Bibliotheksaktualisierungen) verringert.

SQL-basierte Implementierung

Als Anfragesprache ist der standardisierte SQL-Kern (folgend nur „SQL“) aus multiplen Gründen ein wesentlicher Punkt eines datenbankbasierten Entwicklungsframeworks. Hierbei ist zunächst der Fokus auf SQL-92-Kernfunktionalitäten zu setzen, da wie in Abschnitt 2.3.4 beschrieben ist, die Abdeckung modernerer Features von SQL nur vereinzelt in Datenbanksystemen unterstützt wird. Wie sich jedoch in den Abschnitten 6.2 oder 9.3 zeigen wird, ist die Einbeziehung neuerer Funktionalitäten, wie beispielsweise rekursive Anfragen oder Aggregatfunktionen, in manchen Fällen notwendig für effiziente Umsetzungen und daher zu berücksichtigen.
Durch die Verwendung des standardisierten SQL wird vor allem der Austausch des zugrundeliegenden Datenbanksystems ohne wesentliche Anpassung der Implementation ermöglicht. Dies kann in der Praxis entscheidend sein, etwa aus Lizenzgründen oder da beispielsweise der Wechsel zu einem, etwa auf moderner Hardware besser angepassten, relationalen System einen entscheidenden Performancegewinn verspricht (vgl. Abschnitt 6.2). Zum anderen ermöglicht das

exklusive Nutzen von SQL die Integration oder das Verknüpfen weiterer relationaler Systeme im Sinne von Multidatenbanksystemen. Dieser Ansatz kann etwa für die adaptive Erweiterung etablierter Datenbanken (inklusive gespeicherter Sensordaten) von Nutzen sein, wird jedoch im Folgenden nicht näher diskutiert. Neben dieser Systemunabhängigkeit ist ein starkes Argument für die Nutzung von SQL die Langlebigkeit der Sprache. So ist es etwa möglich, 40 Jahre alte Implementationen auf modernen Systemen zu nutzen ohne dass es Anzeichen dafür gibt, dass SQL-Systeme in naher Zukunft an Bedeutung verlieren. Die Performance der Implementation wird in diesem Fall durch die andauernde Verbesserung und Anpassung der Datenbanksysteme an moderne Technik vorangetrieben. Im Gegensatz hierzu haben in den letzten Jahren, während der rasanten Entwicklung der vergleichsweise jungen Felder der Big Data Analytics und des Cloud Computing, eine Vielzahl an neuen Systemen Auf- und Abstiege oder massive interne Umstrukturierung erfahren. Die Folge dieses Entwicklungstempos ist neben dem fokussierten Gewinn an neuer Funktionalität und Geschwindigkeit auch die, das teuer entwickelte und installierte Software innerhalb weniger Jahre nicht mehr kompatibel und anwendbar mit aktualisierten Umgebungen ist.

Neben diesen systembezogenen und praktischen Vorteilen des aktuellen und historischen Stands der Technik kann das Datenbankframework zudem aufgrund des konzeptuellen Kerns von SQL profitieren. Da SQL seine formale Basis in erweiterten Relationenalgebren, (vergleiche Abschnitt 7.1), Logikkalkülen und Logiksprachen (etwa Datalog) hat, ist es möglich Ergebnisse aus der gleichen konzeptuellen bzw. theoretischen Domäne anzubinden. Wesentliche Vertreter solcher Anwendungen sind beispielsweise jene aus der Grundlagenforschung der Bereiche der Datenprivatheit oder des Provenance Managements (vergleiche auch Abschnitt 10.2.1).

So wurde beispielsweise in [21] gezeigt, wie durch das Umschreiben von SQL-Anfragen Privatheitsaspekte sensibler Daten gesichert werden können. Diese sind vor allem in den letzten Jahren immer öfter in den Mittelpunkt öffentlicher Debatten gerückt.

Im Bereich des Provenance Managements wurde zudem in [46] beschrieben, wie der CHASE-Algorithmus — ein Verfahren zur Lösung verschiedener Datenbankprobleme der Theorie und Praxis— genutzt werden kann, um minimale Sub-Datenbanken zu errechnen. Ein Anwendungsfall hierfür ist etwa das Speichern der minimalen Menge von Tupeln großer Sensordaten, die eine komplette Rekonstruktion des ursprünglichen Datenbestands ermöglichen oder die Lösung von Why-Provenance-Fragen (vergleiche Abschnitt 10.2.1).

In [46] wurden zudem andere Anwendungsmöglichkeiten, wie beispielsweise semantische Anfrageoptimierungen in Multimediadatenbanken, des CHASE — und damit des hier diskutierten Frameworks — in Kürze diskutiert.

Die Anbindung dieser Anwendungen in einem SQL-Framework wird vor allem durch seinen Status als verbreitetster Vertreter relationaler Anfragesprache bestärkt.

Intraoperator-Parallelisierung von SQL-Anfragen

Um Auswertungen von zu Big-Data-Analysen korrespondierenden SQL-Anfragen zu ermöglichen, ist es nötig diese parallel im Datenbanksystem zu verarbeiten. In modernen parallelen und verteilten Datenbanksystemen sind bereits viele Methoden zur Interanfrage-Parallelisierung und Intraoperator-Parallelisierung implementiert. Da jedoch letztere vor allem auf die Umsetzung einfacher Aggregation für klassische Datenbankszenarien beschränkt sind, ist es nötig zu untersuchen, inwiefern gezielte Partitionierungsstrategien und Folgen (simultan gestellter) SQL-Anfragen parallele Auswertungen ermöglichen und beschleunigen können.
Wie im Zuge dieser Forschungsarbeit in [12,13] beschrieben wurde, kann sich eine Einschränkung auf parallele Systeme gegebenenfalls negativ auf Auswertungsgeschwindigkeiten auswirken, etwa durch einen dominierenden Kommunikations-Overhead bei vergleichsweise niedrigen Datenmengen. Eine Erweiterung des Framework auf zentrale Systeme ist demzufolge ebenfalls sinnvoll.

Schnittstelle für Restoperatoren und Post-Processing

Im Kontext der Entwicklung von Assistenzsystemen ist es von fundamentaler Bedeutung, dass die Berechnung innerhalb der Datenbank für Entwickler keine funktionale Einschränkung mit sich führt. Operationen, die nicht oder nur ineffizient in der Datenbank via SQL gelöst werden können, müssen auf anderen Wegen umgesetzt werden. Dies kann etwa mittels externer Schnittstellen geschehen, welche insbesondere für graphische Visualisierungen von (Teil-)Ergebnissen nötig sind. Hierbei ist zu beachten, dass die ursprünglichen Vorteile des datenbankbasierten Ansatzes nicht zu sehr eingeschränkt werden. Sensible Daten sollten beispielsweise nicht kommuniziert werden. Dies kann etwa durch im Allgemeinen nicht-invertierbare Operationen, wie etwa Aggregation, erreicht werden. Ein anderer Ansatz ist die Nutzung von Erweiterungen spezifischer Datenbanksysteme, wie die Integration von Bibliotheken oder nicht-standardisierter SQL-Operatoren und -Datentypen (vgl. Abschnitt 4.4). Dies schränkt die Systemunabhängigkeit ein, wahrt aber Aspekte der Datensicherheit oder die Einsparung von Datenkommunikation.

Wahl geeigneter relationaler Datenbanksysteme

Das Nutzen des standardisierten SQLs ermöglicht, wie bereits dargestellt, eine weitgehend unabhängige Wahl der zugrundeliegenden relationalen Datenbanksysteme. Ungeachtet dessen existieren zum Teil große Unterschiede in der Auswertungsgeschwindigkeit von Analysefunktionen. Wie in Kapitel 7 näher beschrieben wird, bilden die Qualität interner Optimierer, Funktionalitäten wie Indexstrukturen, sowie die gewählte Speicherstruktur (spalten- oder zeilenweise) eines Systems hierbei maßgebliche Kriterien für die Effizienz der Auswertung. Eine Studie zum Einfluss dieser Kriterien ist sinnvoll und etwa, je nach Weiterentwicklungen einzelner Systeme, in regelmäßigen Abständen zu wiederholen, um die adäquate Wahl der Datenbanksysteme sicher

zu stellen.

Data Provenance im Kontext von Machine Learning

Wie bereits in Abschnitt 2.1 beschrieben, werden in der Entwicklungsphase von Assistenzsystemen eine Vielzahl von Sensoren genutzt, um eine sichere Obermenge für die spätere Nutzungsphase bereitzustellen. Entwickler sind daher daran interessiert, entscheidende und unwichtige Sensoren zu detektieren und gegebenenfalls aus Gründen der Infrastrukturkosten des späteren Assistenzsystems und der Performance aus dem betrachteten Datenbestand zu entfernen. In dem Forschungsbereich des Provenance Management wird für klassische Szenarien von Datenbankanwendungen unter anderem untersucht, welche Tupel Einfluss auf betrachtete Anfrageergebnisse besitzen. Um Entwicklern zu ermöglichen, Rückschlüsse für deren Sensorwahl zu treffen, ist es nötig zu analysieren, inwiefern Techniken des Provenance Management auch für die betrachteten mathematischen Operationen Anwendung finden können oder inwiefern klassische Verfahren der Dimensionsreduktion (vgl. Feature Extraction in der Aktivitätserkennungspipeline aus Abbildung 2.2) in das SQL-Framework eingebettet werden können.

3.2 Konzept

Mit den gesammelten Anforderungen und Teilaspekten kann nun das hier untersuchte Konzept für die Entwicklungsphase von Assistenzsystemen in kompakter Form entwickelt und präsentiert werden. Der Ansatz ist insbesondere Teil des Langzeitprojekts PArADISE (**P**rivacy-**a**wa**r**e **a**ssistive **d**istributed **i**nformation **s**ystem **e**nvironment) (vgl. etwa [9]), welches in Abbildung 3.1 schematisch dargestellt ist. Das Framework unterstützt die Entwicklungs- (linke Seite) und Nutzungsphase (rechte Seite) von Assistenzsystemen unter Wahrung von Aspekten der Datenprivatheit und -sicherheit [21, 45].
Die Architektur des hier vorgestellten Konzeptes für die Entwicklungsphase ist vereinfacht auf der linken Seite der Abbildung dargestellt. Entwickler übergeben in einer zu spezifizierenden Form Methoden und Algorithmen mit Machine-Learning-Kontext. Diese werden in der „ML2PSQL“-Schicht in Sequenzen von SQL-Code, welche eine intraoperatorparallele Verarbeitung ermöglicht, überführt. Hierbei werden insbesondere SQL-92-Statements verwendet. Ausnahmen, wie beispielsweise rekursive Anfragen, werden in Kapitel 7 und 9 erläutert.
Der Übersetzungsprozess ist detaillierter in Abbildung 3.2 dargestellt und beinhaltet zusätzlich den Aspekt der Dimensionsreduktion durch Übersetzung von Feature-Extraction-Verfahren. Die Algorithmen werden hier zunächst in eine Sequenz von SQL-Anfragen übersetzt. Daraufhin werden die einzelnen Anfragen (insofern möglich) im nächsten Schritt in korrespondierende Teilanfragen aufgeteilt, die eine Intraoperator-Parallelisierung ermöglichen. Dies können beispielsweise klassische Operatoren der linearen Algebra, wie Matrix-Vektor-Multiplikationen, sein. Die Ope-

rationen werden im finalen Zerlegungsschritt jeweils zur Parallelisierung in weitere Teilanfragen aufgeteilt, die auf die Datenverteilung der jeweiligen Knoten zugeschnitten sind. Im Fall der Matrix-Vektor-Multiplikation können, wie in Abschnitt 8.1 beschrieben, durch eine geeignete Verteilung alle Suboperationen auf jedem Knoten komplett lokal ausgeführt werden. Hierfür ist neben einer effizienten Umsetzung der Partitionierung auch eine geeignete Wahl von Zugriffsstrukturen nötig.
Ist die Überführung der Algorithmen in eine Sequenz von Mengen von SQL-Anfragen abgeschlossen, werden diese durch ein paralleles relationales Datenbanksystem abgearbeitet. Sind die zu betrachtenden Datenmengen nicht groß genug, wird die Nutzung eines zentralen Systems oder die Verwendung eines einzelnen Rechenknotens bevorzugt. In diesem Fall wird der Übersetzungsprozess in SQL nach der ersten Phase ausgesetzt.
Neben der reinen Übersetzung sind zusätzliche Funktionen, wie das Scheduling der Anfragen und die Erkennung nicht-SQL-affiner-Operationen (wie beispielsweise Visualisierungen), Aufgabe der „ML2PSQL“-Schicht. Die einzelnen Schnittstellen der linken Seite von Grafik 3.1 werden in Abschnitt 6.1 im Detail bezüglich deren Funktion, Zusammensetzung und möglichen Trade-Offs genauer diskutiert. Hierbei wird insbesondere auch auf die Anbindung externer Softwareschnittstellen eingegangen.
Wie auf der rechten Seite von Abbildung 3.1 zu erkennen ist, ist eine erweiterte Architektur für die Nutzungsphase großer Assistenzsysteme im PArADISE-Framework vorgesehen. In dieser Phase werden SQL-Anfragen vorwiegend vertikal verteilt, um personenbezogene Daten so nahe wie möglich an den eigentlichen Sensoren zu filtern und zu verarbeiten. Die Restoperationen (in der Abbildung „Remainder Query“), werden nach einer möglichen Anpassung der Anfragen im Sinne der Datenprivatheit (vgl. [21]), analog zum Konzept der Entwicklungsphase, auf einem parallelen System verarbeitet.

3.3 Diskussion

Das vorgestellte Konzept hat das Potenzial, alle im Vorfeld gesammelten Anforderungen zu erfüllen und so den Entwicklungsprozess und Teile der Nutzungsphase von Assistenzsystemen entscheidend zu unterstützen.
Der Kernaspekt ist hierbei die effiziente Übersetzung beziehungsweise die Übersetzbarkeit der Analyse- und Machine-Learning-Algorithmen. Solche Verfahren werden klassischerweise in imperativen Programmiersprachen umgesetzt, welches im Gegensatz zu dem mengenbasierten und im Kern deklarativen SQL steht. Dies ist insbesondere daher untersuchungswürdig, da SQL erst mit der Einführung von rekursiven Anfragen und Fensterfunktionen in SQL:2003 Turing-vollständig wurde [47]. Wie in Abschnitt 4.4 beschrieben, orientieren sich viele aktuelle Forschungsprojekte und Datenbanksysteme auf die nicht-standardisierte Erweiterung von Datenbankfunktiona-

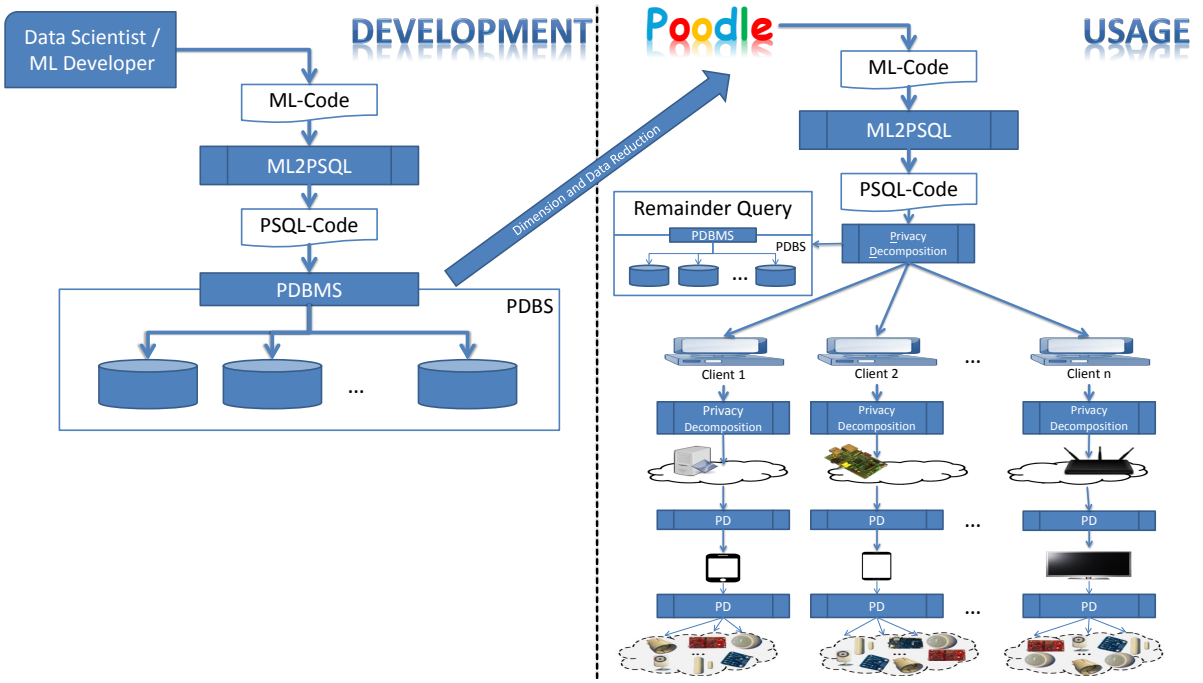


Abbildung 3.1: Die Phasen des PArADISE-Projekts: Auf der linken Seite ist die horizontal verteilte Architektur der Entwicklungsphase dargestellt. Die rechte Seite zeigt die vertikal verteilte Architektur der Nutzungsphase. Der zentral gelegene Pfeil repräsentiert den Selektionsprozess geeigneter Sensoren aus der Entwicklungsphase für die spätere Nutzungsphase. Grafik entnommen aus [9].

litäten. Offizielle Erweiterungen des SQL-Standards, die imperative Programmiersprachenkonstrukte (definiert in SQL/PSM [34]) einführen, sind nur spärlich in aktuellen Datenbanksystemen standardkonform unterstützt. Zudem schränken die zugehörigen Prozeduren oftmals Optimierungsfunktionalitäten der Systeme ein, selbst wenn Teile der Prozeduren in SQL geschrieben sind. All diese Lösungen vermögen es in einigen Szenarien, die Performance klassischer SQL-Lösungen zu übertreffen, bieten jedoch keine verbreitete und standardisierte Schnittstelle.

Da pure SQL-Lösungen im Kontext von Assistenzsystemen und Machine Learning größtenteils unerforscht sind, wird der Untersuchungsschwerpunkt in dieser Arbeit auf die Überführung solcher Algorithmen in SQL gesetzt. Hierfür muss zunächst gezeigt werden, dass verlustfreie Übersetzungen möglich sind und deren Performance mit anderen etablierten Systemen und Ansätzen verglichen werden, um eine Einordnung des SQL-Ansatzes zu ermöglichen. Ein Aspekt hierbei ist, dass die vergleichsweise starre SQL-Anfragestruktur unter Umständen ungeeignete Anfragepläne für klassische Teiloperationen der Algorithmen liefert. Zudem ist zu untersuchen, ob und wie solche Anfragen effizient im Sinne der Intraoperatorparallelität in parallelen Datenbanksystemen umgesetzt werden können, um eine effiziente Skalierung für datenintensive Systeme zu gewährleisten.

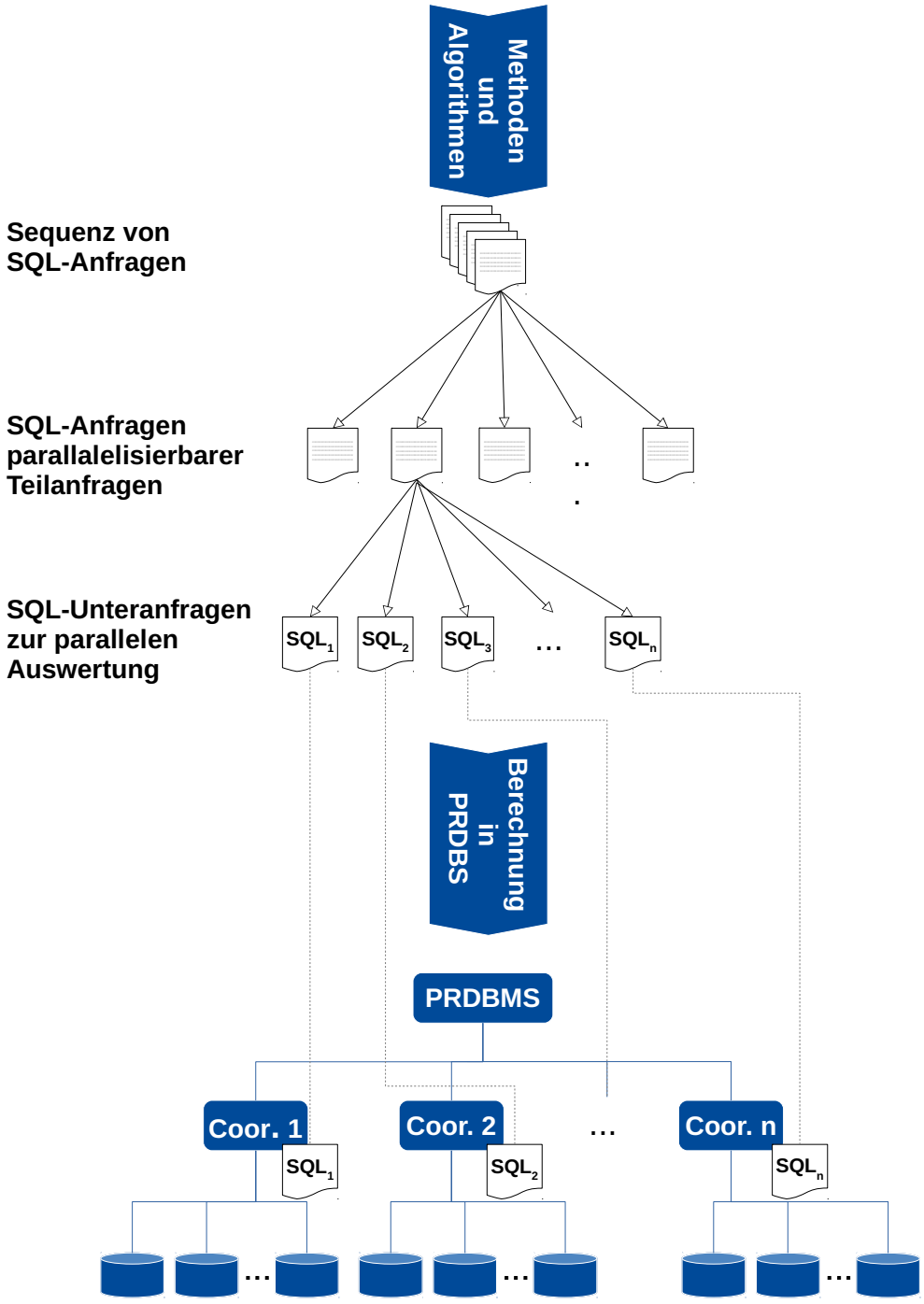


Abbildung 3.2: Grafische Darstellung des Konzepts zur effizienten Aktivitätserkennung und -vorhersage in Assistenzsystemen. Die Architektur des parallelen Datenbanksystems ist angelehnt an die GTM-Coordinator-Datanode-Architektur von Postgres-XL aus Abschnitt 4.3.3.

Tabelle 3.1: Zusammenfassung von Vorteilen und Herausforderungen des vorgestellten Konzepts.

Vorteile	Nachteile & Herausforderungen
Skalierbare Parallelisierung	Überführung imperativer Methoden in mengenbasiertes, deklaratives SQL
Unabhängigkeit vom Datenbanksystem	Wenige Untersuchungen zu parallelem wissenschaftlichen Rechnen in SQL
Langlebigkeit der Implementation	Kommerzielle parallele Datenbanksysteme oft teuer
Datensicherheit	Performance lokal & parallel zu untersuchen
Datenprivatheit möglich	Nutzbarkeit von Provenance-Management-Techniken zu untersuchen
Transparente Optimierung	

Es ist bei dieser Untersuchung jedoch vor allem zu bedenken, dass das PArADISE-Framework Aspekte der Datensicherheit und (durch modulare Erweiterungen) der Datenprivatheit ermöglichen soll. Dies motiviert, speziell bei der Nutzung personenbezogener sensibler Daten, eine Verwendung dieses Ansatzes auch bei leichten Einbußen in der Verarbeitungsgeschwindigkeit. Aus diesem Grund ist, wie bereits beschrieben, das Anwendungsfeld zudem auf zentrale Datenbanksysteme mit moderaten Datengrößen erweiterbar.
Eine kurze Übersicht über die Vorteile und Herausforderungen des vorgestellten Konzepts ist in Tabelle 3.1 dargestellt.

3.4 Weiterer Verlauf

Zur Untersuchung des vorgestellten Konzepts wird der weitere Verlauf der Arbeit wie folgt strukturiert: Im folgenden Kapitel werden zunächst etablierte Datenbanktechniken und -funktionalitäten beschrieben, die für den später folgenden Übersetzungsprozess von Bedeutung sind. Daraufhin wird eine Kurzbeschreibung der weit verbreiteten *Basic Linear Algebra Subprograms* (BLAS) gegeben. Diese werden im weiteren Verlauf mehrfach als Referenz für Datenstrukturen und Berechnungsansätze des wissenschaftlichen Rechnens genutzt. Daraufhin werden etablierte Big-Data-Umgebungen und (parallele) Datenbanksysteme vorgestellt, die zur Evaluation in den Abschnitten 9.1 und 9.2 genutzt werden. Abschließend wird ein State-of-the-Art-Überblick zu wissenschaftlichem Rechnen und Machine Learning in Kombinationen mit Datenbanktechnologien gegeben. Hierbei werden vor allem Systeme und Forschungsprojekte beschrieben, die zusätzliche Funktionalitäten in den jeweiligen Systemen ermöglichen.
In dem darauf folgenden Kapitel 5 werden für die initiale Untersuchung essenzieller Verfahren für Assistenzsysteme die bereits in Kapitel 2 motivierten Hidden-Markov-Modelle (HMM) näher

eingeführt. Hierbei werden, nach einer Definition und zugehörigen Beispielen, grundlegende Probleme und deren Lösungsverfahren vorgestellt. Es wird sich hierbei zeigen, dass diese maßgeblich durch Basisoperationen der Linearen Algebra beschrieben werden können. Nach einer Analyse nötiger Operationen für die Überführung in SQL, wird durch eine kleine Erweiterung der gewonnenen Operatormenge eine weitgehende Abdeckung der BLAS-Bibliothek erreicht. Damit wird durch die Überführung dieser Menge eine Abdeckung zahlreicher Szenarien des wissenschaftlichen Rechnens und damit eine hohe Funktionalität des Datenbankkonzepts erreicht. Aus diesem Grund liegt der Fokus in den darauf folgenden Untersuchungen maßgeblich auf der Untersuchung dieser Operatormenge.

In Kapitel 6 wird eine Diskussion möglicher Architekturen für die Umsetzung des vorgestellten Frameworks durchgeführt. Hierbei werden zusätzlich Aspekte zur Wahl geeigneter relationaler Datenbanksysteme, wie deren interne Systematik zur Speicherung von Relationen, für den beschriebenen Lineare-Algebra-Kontext diskutiert. Unter Berücksichtigung der etablierten Operatormenge wird daraufhin eine Analyse und experimentelle Studie möglicher Schemata für Vektoren und Matrizen durchgeführt. In dieser wird insbesondere zwischen dicht und dünn besetzten Daten unterschieden und mögliche Bezüge zur Zeitreihenverwaltung diskutiert.

Mit der etablierten Darstellung von Matrizen und Vektoren werden in Kapitel 7 die Übersetzung der Basisoperationen in einer anfangs definierten erweiterten Relationenalgebra und SQL-92 gezeigt. Dies soll insbesondere die Anwendbarkeit auf relationalen Systemen mit eingeschränkter Funktionalität, so wie sie teilweise auf leistungsschwacher Hardware mit räumlicher Nähe zu Sensoren genutzt werden können, motivieren. Daraufhin wird eine Diskussion der Effizienz der Übersetzung und Verarbeitung in relationalen Systemen geführt. Hierbei werden insbesondere Vergleichsrechnungen zu BLAS-Implementationen vorgenommen. In den beiden abschließenden Abschnitten wird die effiziente Komposition der entwickelten Anfragen und die Nutzung von Indexstrukturen diskutiert.

Mit der Etablierung der Übersetzung werden in Kapitel 8 Möglichkeiten zur effizienten Verarbeitung der Basisoperationen in parallelen relationalen Datenbanksystemen diskutiert. Hierfür werden klassische Parallelisierungsstrategien für Lineare-Algebra-Operatoren vorgestellt und eine Überführung dieser auf Datenbanksysteme untersucht. In dieser Analyse werden neben der Umsetzbarkeit zugehöriger Partitionierungsstrategien auch die zugehörige parallele Anfragevearbeitung diskutiert. Mit diesen Überlegungen wird ein Ansatz zur Intraoperator-Parallelisierung mittels Anfragezerlegung vorgestellt, welcher im darauf folgenden Evaluationskapitel ausgewertet wird.

Nach dieser Evaluation werden diese Ergebnisse im Kapitel 9 genutzt, um die Datenbanklösung mit der Verarbeitung der Big-Data-Umgebung Apache Spark zu vergleichen. Anschließend werden zwei Szenarien vorgestellt, die die Erweiterung der Anwendung des Frameworks verdeutlichen sollen. Zunächst wird hierfür die Berechnung von Fourier-Transformationen in SQL be-

schrieben und ausgewertet, worauf abschließend eine Diskussion zur Zeitreihenverwaltung und -analyse im Automotive-Kontext in Datenbanksystemen folgt. Das Kapitel wird mit einer Beschreibung eines industriellen Projektes abgeschlossen. In diesem werden Vorhersagen von Schiffsrouten mittels Markov-Ketten in relationalen Datenbanken berechnet. Das Beispiel zeigt den praktischen Nutzen des hier diskutierten SQL-basierten Ansatzes.
Im finalen Kapitel 10 werden, nach einer Zusammenfassung und Einordnung der hier vorgestellten Ergebnisse, darauf aufbauende Forschungsansätze vorgestellt und teilweise erste zugehörige Erkenntnisse präsentiert. Hierzu zählen etwa die Dimensionsreduktion im Kontext von Assistenzsystemen und deren Zusammenhang zum Provenance Management oder das Lernen von HMM-Parametern aus annotierten Testdaten.

Kapitel 4

Stand der Technik und Forschung

Um eine Abgrenzung und Einordnung des vorgestellten Konzepts und deren Umsetzung zu ermöglichen, wird in diesem Kapitel der aktuelle Stand der Technik und Forschung bezüglich relevanter Anwendungsgebiete aufgearbeitet. Hierfür wird zunächst eine Abhandlung über Datenbankfunktionalitäten geführt, die Einfluss auf die Umsetzung und Performance SQL-basierter Implementationen besitzen und Gegenstand späterer Untersuchungen sind. Daraufhin wird die Referenz-Bibliothek der *Basic Linear Algebra Subprograms* (BLAS) vorgestellt, welche ein de-facto Standard für lokale (und teilweise verteilte) Berechnungen wesentlicher Subroutinen des wissenschaftlichen Rechnens sind. Darauf folgend wird das in der Einführung bereits erwähnte *MapReduce*-Programmiermodell beschrieben und Apache Spark als erweiterndes Big-Data-System präsentiert. Abschließend werden Projekte kategorisiert und vorgestellt, die Datenbanktechniken und die Berechnung von Linear-Algebra-Operationen oder Machine-Learning-Verfahren innerhalb von Datenbanksystemen ermöglichen.

4.1 Datenbanktechniken und -implementationsaspekte

In diesem Abschnitt werden etablierte Datenbanktechniken und Details zu deren Umsetzung vorgestellt. Hierfür wird zunächst eine Übersicht über klassische Indexstrukturen gegeben, welche in Abschnitt 7.4 im gegebenen Kontext evaluiert werden. Daraufhin wird eine kurze Abhandlung bezüglich etablierter Verbundimplementationen in relationalen Datenbanksystemen präsentiert. Die Ansätze werden vor allem in Abschnitt 7.2 genutzt, um dort betrachtete Performance-Auswertungen von SQL-Implementationen zu erklären.

4.1.1 Indexstrukturen

Relationen einer Datenbank werden in physischen Dateien („data files“) gespeichert. Die jeweiligen Tupeln sind auf Datenbankseiten (kurz „Seite“), die eine oder mehrere Blöcke der genutzten

Platten umfasst, hinterlegt [35]. Nach einer Tupelanfrage werden die zugehörigen Seiten vom Dateisystem des Betriebs- oder Datenbanksystems (abhängig vom genutzten Datenbanksystem) geladen und verarbeitet. Wie in Abbildung 4.1 dargestellt, bestehen Seiten hierbei aus jeweils einem Header, sequenziell angeordneten Datensätzen und unbelegten Bytes.

Um bei einer Selektion spezifischer Tupel nicht jede Seite der zugehörigen Relationen untersuchen zu müssen („full table scan“), werden die Datenbankseiten typischerweise entsprechend organisiert. Hierfür nutzen Datenbanksysteme unter anderem Indexstrukturen (auch Indexdateien). Diese sind zusätzliche Datenstrukturen, die einen effizienten Zugriff auf Tupeln bezüglich konkreter Werte oder Wertebereiche spezifizierter Attribute oder Mengen von Attributen („Suchschlüssel“) ermöglicht. Solche Strukturen fordern generell einen Trade-Off bezüglich des potenziellen Laufzeitgewinns selektiver Anfragen und dem nötigen Extraaufwand, der durch die Verwaltung dieser zusätzlichen Strukturen bei Änderungen des Datenbestandes der ursprünglichen Relationen entsteht.

Indexstrukturen werden im Allgemeinen durch Indexdateien realisiert, die aus Paaren von Suchschlüsselwerten und Zeigern, die auf die entsprechenden Tupel in den physischen Seiten verweisen, bestehen. Die Strukturen werden vorrangig in primär und sekundär aufgeteilt. Primäre Indexe beeinflussen hierbei direkt den Speicherort der Daten in den physischen Dateien, wohingegen sekundäre Indexe die vorliegende Datenspeicherung nutzen. Zusätzlich werden im Fall primärer Indexstrukturen dicht (im englischen „dense“) und dünn besetzte („sparse“) Strukturen unterschieden. Dichte Indexe besitzen für jeden Dateneintrag einen eigenen Eintrag in den Indexdateien. Umgekehrt gilt dies für dünn besetzte Indexstrukturen, bei denen nicht jeder Eintrag, sondern nur einzelne Seiten von Tupeln referenziert werden, nicht. In diesen Fällen müssen die Daten bezüglich der Suchschlüsselwerte sortiert sein. Nach der Referenzierung können die gesuchten Tupel aus den jeweiligen Seiten per Binärsuche durchsucht werden. Alternativ können mehrstufig dünn besetzte Indexe, wie in Abbildung 4.3 dargestellt, verwendet werden. Dünn besetzte Indexdateien sind offenbar kleiner und günstiger in deren Verwaltung, benötigen jedoch Zusatzoperationen um jeweilige Tupel zu selektieren. Zur Veranschaulichung sind in Abbildung 4.2 dicht und dünn besetzte Primärindexstrukturen auf „sequential data files“ dargestellt. Letztere beschreiben den definierenden Umstand, dass die Einträge in den physischen Dateien der jeweiligen Relation nach einem Attribut (klassischerweise dem Primärschlüssel) sortiert wurde. Solche sortierten Daten und primäre Indexstrukturen werden in Abschnitt 7.4 und bei der Wahl der Relationenschemata für Matrizen und Vektoren in Abschnitt 6.4 diskutiert.

In den folgenden Unterabschnitten werden spezifische etablierte sekundäre Indexstrukturen vorgestellt, die Schwerpunkt der Untersuchungen zur Nutzung von Indexstrukturen im Kontext von Hidden-Markov-Modellen und linearer Algebra in Abschnitt 7.4 sind. Sekundäre Indexe sind konzeptuell stets dicht, da keine Aussagen über benachbarte Tupel in den Seiten möglich sind. Die Werte der Suchschlüssel sind, wie in Abbildung 4.4 dargestellt, bei diesen ebenfalls in

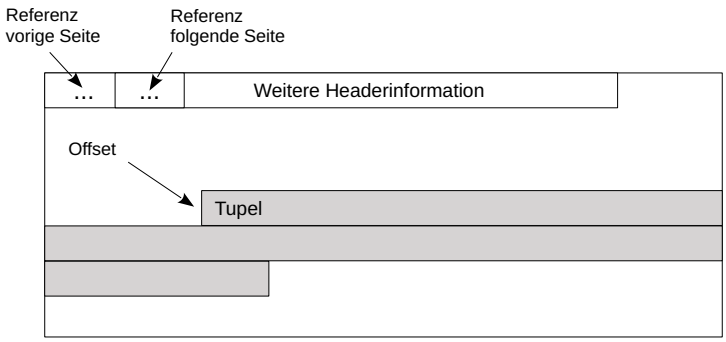


Abbildung 4.1: Struktur einer Seite. Abbildungsidee aus [35] entnommen.

den Indexdateien sortiert. Da die Relationentupel nicht geordnet sind, können sich in diesem Fall jedoch Tupel mit gleichen Suchschlüsselwerten auf unterschiedlichen Datenbankseiten befinden, welches in ungünstigen Fällen zu hohen IO-Kosten führen kann. Für erweiterte allgemeine Konzepte und effiziente Implementationstechniken von Indexstrukturen sei auf [35] und [48] verwiesen. Im Folgenden werden nun zunächst B-Bäume und Hash-Indexstrukturen als Vertreter etablierter sekundärer Indexstrukturen betrachtet. Abschließend werden GIN-Indexstrukturen vorgestellt, die räumliche und mehrdimensionale Zugriffe ermöglichen und in Abschnitt 7.4 in PostgreSQL evaluiert werden.

B- und B+-Bäume

B- und B+-Bäume bilden eine Familie mehrstufiger Indexstrukturen in relationalen Datenbanksystemen. B+-Bäume zählen hierbei zu der wohl verbreitetsten Variante [48] und werden aus diesem Grund im Folgenden vorrangig behandelt. B+-Bäume organisieren ihre Knoten in balancierte Baumstrukturen, das heißt alle Wege von der Wurzel zu einem Blatt besitzen die gleiche Länge. Wie in Abbildung 4.5 dargestellt sind B+-Bäume klassischerweise in drei Ebenen aufgebaut: der Wurzel, den Zwischenschichten und den Blättern. Jeder Knoten besteht aus n Suchschlüsselwerten und $n + 1$ Zeigern, wobei n so gewählt wird, dass der Block eine Datenbankseite bestmöglich ausfüllt. Die Suchschlüsselwerte einer Stufe sind hierbei über alle Knoten von links nach rechts sortiert hinterlegt. Zur effizienten Aktualisierung der Struktur nach einzelnen Änderungen des Datenbestands können die Blätter auch nicht vollkommen belegt werden. Die Füllstände betragen abhängig von der Anwendung zwischen 50% und 100 %. Im Allgemeinen wird davon ausgegangen, dass die Wurzel mindestens zwei Zeiger besitzt (d.h. die Relation enthält mehr als zwei Tupel). B+-Bäume passen die Anzahl der Indexstufen beziehungsweise Zwischenschichten anhand der Relationengröße automatisch an.

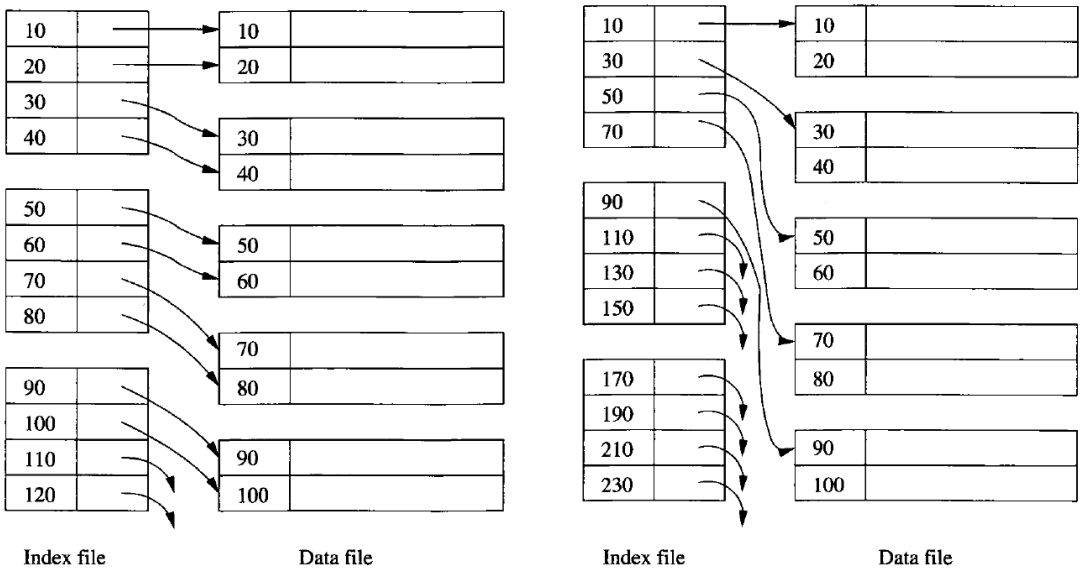


Abbildung 4.2: Beispielhafte Darstellung von dicht (links) und dünn besetzten Indexdateien (rechts) auf sequential data files. Die Grafiken wurden aus [48] entnommen und leicht angepasst.

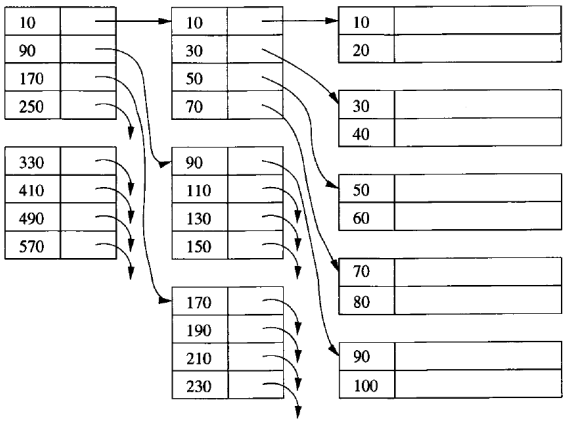


Abbildung 4.3: Beispielhafte Darstellung einer mehrstufigen dünn besetzten Primärindexstruktur. Die Grafik wurde aus [48] entnommen.

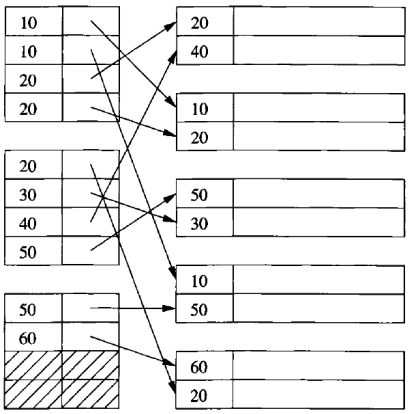


Abbildung 4.4: Beispielhafte Darstellung eines Sekundärindex. Die Grafik wurde aus [48] entnommen.

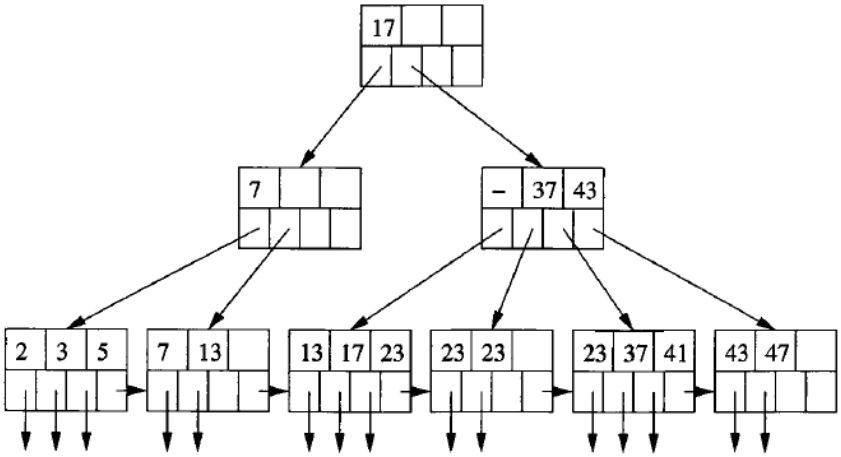


Abbildung 4.5: Beispielhafte Darstellung eines B+-Baums. Die Grafik wurde aus [48] entnommen.

Die Suchschlüsselwerte der Blätter sind Attributwerte aus der ursprünglichen Relation, wobei der letzte Zeiger eines Blattes das nächste rechts-liegende Blatt referenziert. Die übrigen verweisen jeweils auf die zugehörigen Einträge in den physischen Dateien der Relation. Die Zeiger der inneren Knoten und der Wurzel verweisen auf Knoten der nächsten Stufe. Wenn $K_1, \ldots, K_m$ die Suchschlüsselwerte bezeichnen, referenziert der erste Zeiger von links auf den ersten Knoten der nächsten Baumstufe, der die größten Werte hält die kleiner als K_1 sind. Der i-te Zeiger ($i < 1 \leq m$) eines Knotens verweist auf den Knoten der nächsten Stufe, dessen Werten zwischen K_{i-1} und K_i liegen.

B+-Bäume können in verschiedenen Formen und Varianten umgesetzt werden. Abhängig von der Sortierung der Tupel können diese dicht oder dünn besetzt sein. Die Suchschlüsselwerte in Blättern können in duplizierter oder eindeutiger Form stehen. Da in dieser Arbeit nur B+-Bäume in dichter Form auf eindeutigen Daten angewandt werden, sei für eine Diskussion verschiedener Varianten auf [48] verwiesen.

Mit diesen Spezifikationen kann die Selektion von Tupeln mit dem Suchschlüsselwert K entsprechend einfach gemäß der Baumstruktur umgesetzt werden. Ist ein Blatt erreicht worden, kann mittels des zugehörigen Zeigers das Tupel der Relation selektiert werden. Andernfalls werden die Knoten von der Wurzel an stufenweise mittels den jeweils zugehörigen Zeigern (gemäß $K \in [K_{i-1}, K]$ oder $K < K_1$) durchschritten. B+-Bäume sind durch ihre Referenzierung von Nachbarblättern ebenfalls geeignet für effiziente Bereichsselektionen. Werden etwa Anfragen bezüglich eines Suchschlüsselintervalls $[K_a, K_e]$ formuliert, wird zunächst der Eintrag zu K_a, wie oben beschrieben, selektiert. Von diesem Punkt werden schrittweise solange die rechts-liegend referenzierten Tupel selektiert (auch Blatt-übergreifend mittels der letzten Zeiger) bis der zugehörige Suchschlüsselwert $K > K_e$ ist.

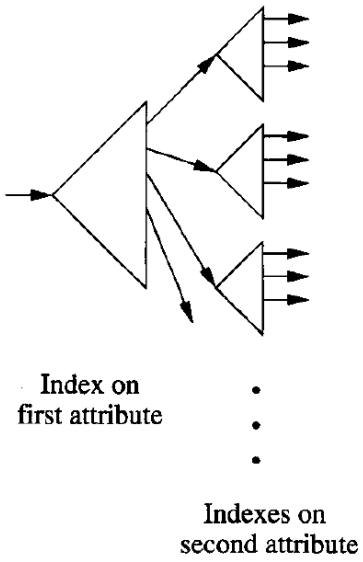


Abbildung 4.6: Beispielhafte Darstellung eines mehrattributigen B+-Baums. Die Grafik wurde aus [48] entnommen.

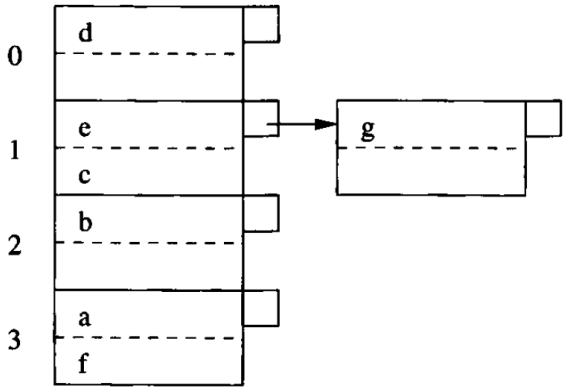


Abbildung 4.7: Beispielhafte Darstellung eines Hash-Index mit Überlaufseite im hash bucket zum Wert 1. Die Grafik wurde aus [48] entnommen.

Der Aufbau und die Aktualisierung von B+-Bäumen wird hier, aufgrund dessen fehlenden Einflusses für die kommenden Betrachtungen, ausgespart.
B+-Bäume können auch (wie in Abschnitt 7.4 untersucht) für multidimensionale Schlüsselmengen genutzt werden. Wie in Abbildung 4.6 dargestellt, kann dies über ein einfache Schachtelung der B+-Bäume geschehen. Das heißt, dass die Blätter des i-ten Baums der mehrdimensionalen Indexstruktur einen Zeiger auf einen weiteren Baum enthalten. Dieser Baum ordnet die Einträge des $i+1$-ten Schlüsselattributs unter Berücksichtigung der in den vorigen i Bäumen etablierten Teilschlüsselwerte. Diese Art der Kombination ist offensichtlich vergleichsweise speicherintensiv und deren Effizienz ist abhängig von der Reihenfolge der genutzten Schlüsselattribute einer Anfrage. Wird etwa das erste Schlüsselattribut in einer Anfrage („partial match query“) nicht spezifiziert, müssen potenziell alle Subbäume durchlaufen werden, welches abhängig vom Szenario sehr ineffizient sein kann.

Hash-Indexe

Hashverfahren können in verschiedener Art und Weise in Datenbanksystemen genutzt werden. Ein wesentliches Anwendungsgebiet sind Hash-Indexstrukturen. Hierbei können grundlegend Hauptspeicher- und Sekundärspeichervarianten unterschieden werden.
In Hauptspeicher-Indexes wird eine Hashfunktion $h : \mathbf{dom}(K) \mapsto \{0, \ldots, B-1\}$ genutzt, welches einem Suchschlüsselwert $k \in \mathbf{dom}(K)$ einem Integer-Wert zwischen 0 und $B-1$ zuweist. Hierbei

ist der Funktionswert stellvertretend für den zugewiesenen *hash bucket*, welche jeweils in einem Array („bucket array“) der Länge B angeordnet sind. Jeder bucket hält hierbei Zeiger, die auf den Header einer verketteten Liste der per Suchschlüssel zugewiesenen Einträge/Tupel verweisen. Der Suchschlüssel K kann in diesem Fall auch mehrdimensional sein.
In Sekundärspeicher-Indexen werden im Allgemeinen Mengen an Tupeln verwaltet, die nicht im Hauptspeicher gehalten werden können. Hierbei wird in einem bucket kein Zeiger auf einen Header, sondern eine Datenbankseite genutzt. In dieser werden die per Hashfunktion zugewiesenen Einträge gespeichert. Wird ein neuer Eintrag einer vollen Seite zugewiesen, können zusätzlich „Überlaufseiten“ (englisch: overflow blocks) hinzugefügt und verkettet werden (vgl. Abbildung 4.7). Analog werden beim Löschen von Einträgen mit Suchschlüssel K die jeweiligen Einträge aus dem bucket des Hashwerts $h(K)$ gesucht und gelöscht. Leerstehende Überlaufseiten können in diesem Fall entfernt werden.
Hashindexe zeichnen sich vor allem durch geringe IO-Kosten für direkte Selektion von Tupeln bezüglich der Suchschlüssel aus. Soll ein Tupel mit Suchschlüssel K selektiert werden, wird nur die Identifizierung des buckets mittels der Berechnung von $h(K)$ benötigt und nur eine Durchsuchung der jeweiligen Seite oder Seiten. Ähnlich positiv verhalten sich die Kosten für das Einfügen und Löschen von Einträgen.
Ein Problem, das bei Hash-Indexstrukturen auftreten kann, ist, dass bei kontinuierlich steigenden Datenmengen die Anzahl der verketteten Datenbankseiten ebenfalls steigt. In diesen Szenarien steigt der IO-Aufwand für das Finden einzelner Einträge folglich. Zusätzlich einschränkend ist, dass Hash-Indexe durch ihre Konzeption keine Bereichsanfragen oder effiziente Suchen nach minimalen oder maximalen Werten unterstützen, da sie (etwa im Gegensatz zu B+-Bäumen) keine verkettete Ordnungsbeziehung der Einträge bezüglich ihrer Suchschlüsselwerte modellieren.

GIN-Indexstrukturen

Die vorgestellten und weitverbreiteten B+-Bäumen und Hash-Indexe sind keinesfalls eine erschöpfende Darstellung existenter Indexstrukturen in relationalen Datenbanksystemen. Eine weitere Familie von Indexstrukturen ist der *Generalized Inverted Index* (GIN) [50], welche in dem hier mehrfach genutzten System PostgreSQL (vgl. Abschnitt 4.3.3) unterstützt werden. Diese Strukturen sind für einen effizienten multidimensionalen Zugriff konzipiert. Der Term „Generalized“ bezieht sich in diesem Fall darauf, dass GIN-Indexe keine vordefinierte Vergleichsoperatoren besitzen [49]. Diese müssen von Nutzern, abhängig von den genutzten Datentypen, definiert werden. Im Gegensatz zu geschachtelten B+-Bäumen verwalten GIN-Indexstrukturen zusammengesetzte Schlüssel derart, dass diese auf die Unterstützung von Anfragen mittels Teilschlüssel ausgelegt sind. Die Struktur speichert hierbei Paare der Form (key, posting list), wobei „key“ einem einzelnen Schlüsselelement entspricht und „posting list“ aus einer Menge von Zeilenidentifizierern zum jeweiligen Schlüssel besteht [50]. Da verschiedene Schlüsselelemente in

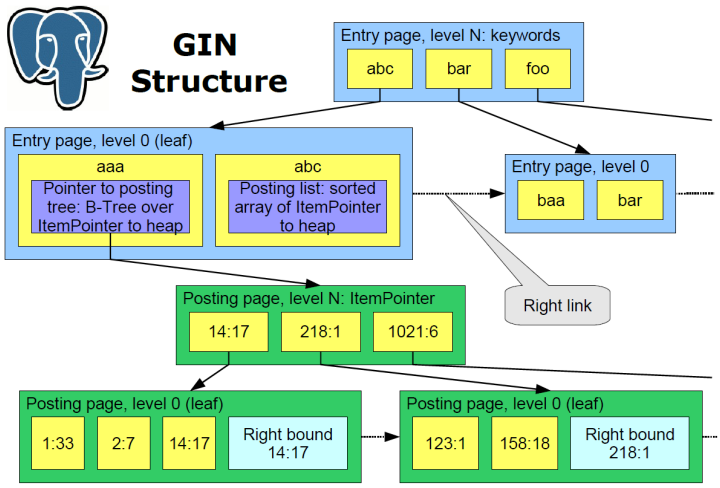


Abbildung 4.8: Beispieldarstellung eines GIN-Indexes. Die Grafik wurde aus [49] entnommen.

einer Zeile auftreten können, treten Identifizierer auch mehrfach im Index auf. Mögliche Werte eines Schlüsselelements werden hierbei jedoch nur einmalig gespeichert, sodass die Größe von GIN-Indexstrukturen im Allgemeinen vergleichsweise klein sind. Wie in Abbildung 4.8 dargestellt, werden GIN-Indexstrukturen intern als B-Baum bezüglich der Teilschlüsselwerte umgesetzt. Die Blätter des Baums besitzen zwei mögliche Einträge. Der erste sind die posting lists, die eine sortierte Liste von Referenzen zu den physischen Einträgen der Tupel des zugehörigen Schlüsselwerts enthalten. Existieren mehr Referenzen als in einem Indextupel gespeichert werden können, werden die Referenzen in einem B-Baum („posting tree“) angeordnet (vgl. Abbildung 4.8). Mehrattributige GIN-Indexstrukturen werden über einen einzelnen B-Baum bezüglich der zusammengesetzten Schlüssel (Spaltennummer, Schlüsselwert) umgesetzt.

4.1.2 Verbundimplementationen

Der Verbund ist einer der wesentlichen Basisoperatoren relationaler Datenbanksysteme und besitzt maßgeblichen Einfluss auf die Performance von Anfragen, die Relationen miteinander verknüpfen. Im Folgenden werden die drei wesentlichen Grundverfahren zur Berechnung von Verbunden beschrieben: Der Nested-Loop-Verbund, der Merge-Verbund und der Hash-Verbund. Im Allgemeinen sind hoch-optimierte Versionen dieser Verfahren in etablierten Datenbanksystemen implementiert. Details zu diesen Umsetzungen sind hierbei nur in Teilen veröffentlicht. Für die in dieser Arbeit nötigen Analysen ist eine Beschränkung auf die Grundverfahren jedoch ausreichend. Der Einfluss der verschiedenen Verbundtechniken auf Anfragen des hier diskutierten Kontexts werden etwa im Abschnitt 7.2 evaluiert. Die hier dargestellten Ausarbeitungen basieren auf [51].

Der Nested-Loop-Verbund

Algorithmus 1 Die Grundstruktur des Nested-Loop-Verbunds bezüglich des natürlichen Verbunds $r_A \bowtie r_B$ mit der zugehörigen Verbundmenge $I = R(r_A) \cap R(r_B)$.

```
for each t_a ∈ r_A do
    for each t_b ∈ r_B do
        if t_a(I) = t_b(I) then
            put t_a‖t_b
        end if
    end for
end for
```

Das konzeptuell einfachste Verfahren zur Berechnung von Verbunden ist der Nested-Loop-Verbund, welcher seinem Namen entsprechend die Verbundbedingung jeder möglichen Tupelkombination zweier Relationen über geschachtelte Schleifen prüft. Die zugehörige Struktur ist in Algorithmus 1 dargestellt. In diesem bezeichnet **put** die Hinzunahme der miteinander entsprechend verschmolzenen Tupel t_a und t_b ($t_a \| t_b$) in die Ergebnisrelation. Aufgrund des geschachtelten Durchlaufens der Relationen ergeben sich in diesem Fall Berechnungskosten der Form $\mathcal{O}(|r_A| \cdot |r_B|)$. Existiert auf der Verbundmenge $I = R(r_A) \cap R(r_B)$ in r_B eine Indexstruktur kann die Berechnung eines Gleichverbundes deutlich beschleunigt werden, da zu jedem Wert von $t_a(I)$ aus r_A mögliche zugehörige Tupel aus r_B mittels Index selektiert werden. Im Fall von Baumindexstrukturen (etwa ein B+-Baum) können die Kosten des Nested-Loop-Verbundes zu $\mathcal{O}(|r_A| \log(|r_B|))$ reduziert werden.
Der Algorithmus kann zusätzlich verbessert werden, indem über die Datenbankseiten anstatt über die einzelnen Tupel der Relation iteriert wird. In diesem Fall werden alle Tupel aus den jeweils gelesenen Datenbankseiten (über geschachtelte Schleifen) untersucht, welches den Verbund in einer vierfach geschachtelte Schleife berechnet. Da in diesem Fall jede Datenbankseite nur einmalig gelesen wird, sind die IO-Kosten im Allgemeinen niedriger, als die der klassischen Struktur. Die Komplexitätsklasse verbleibt hierbei jedoch gleich.
Der Nested-Loop-Verbund ist im Allgemein die ineffizienteste Implementationsvariante, wird aber teilweise als Submethode in komplexeren Verbundverfahren genutzt [48]. Gegensätzlich zu den im Folgenden vorgestellten Merge- und Hashverbund ermöglicht dieser Ansatz jedoch auch die Berechnung allgemeinerer Verbunde und beschränkt sich nicht nur auf Gleichverbunde (Verbunde der Form $r.x = s.y$).

Der Merge-Verbund

Der Mergeverbund ist eine erweiterte Verbundimplementation, die in verschiedenen Szenarien im Laufe der Arbeit (etwa bei Datenmengen, die die Hauptspeichergröße überschreitenden)

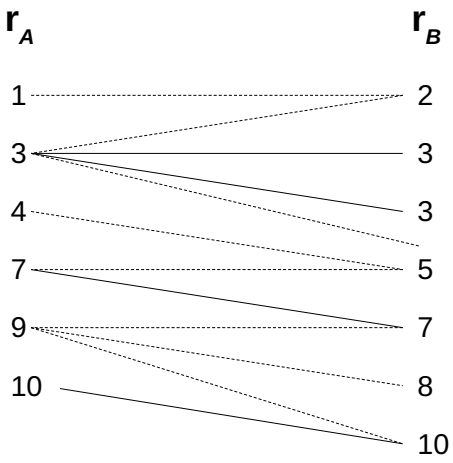


Abbildung 4.9: Beispielhafte Darstellung des Merge-Verbunds sortierter Relationen r_A und r_B. Durchgezogene und gestrichelte Linien beschreiben erfolgreiche beziehungsweise nicht-erfolgreiche Vergleiche bezüglich der Verbundbedingung. Die Grafikidee ist aus [51] entnommen.

die präferierte Verbundimplementation der genutzten Datenbanksysteme ist. Der Algorithmus basiert grundlegend auf der Sortierung der Relationen über den Verbundattributen. Die Grundstruktur des Verfahrens ist in Algorithmus 2 und in Abbildung 4.9 dargestellt. Wie zu erkennen ist, arbeitet das Verfahren durch schrittweise Vergleiche die Elemente beider Relationen ab. Im Falle des Mehrfachauftretens von Verbundattributwerten werden die zugehörigen Werte der jeweils anderen Relation mehrfach durchlaufen. Die Kosten des Verfahrens unterscheiden sich hierbei stark bezüglich der vorliegenden Relationenstruktur. Sind die Verbundattribute in einem der beteiligten Relationen ein Schlüssel (sowie es etwa im Fall elementweiser Matrixoperationen in SQL der Fall ist [vgl. Abschnitt 7.1]) ergeben sich bei vorab sortierten Relationen (etwa durch Indexstrukturen) Kosten von $\mathcal{O}(|r_A|+|r_B|)$. Im unsortierten Fall erhöhen sich die Kosten aufgrund der nötigen Sortierung der Relationen auf $\mathcal{O}(|r_A|\log(|r_A|)+|r_B|\log(|r_B|))$. Gilt für die Verbundmenge keine Schlüsselbeziehung (etwa bei der mehrfachen Existenz von Verbundattributwerten) verschlechtert sich die Komplexitätsklasse deutlich. Im worst case (alle Verbundattributwerte sind identisch (Kreuzprodukt)), iteriert das Verfahren ähnlich zum Nested-Loop-Verbund über die Tupel. Dies führt dementsprechend zu Kosten von $\mathcal{O}(|r_A| \cdot |r_B|)$.

Der Hash-Verbund

Der letzte wesentliche Vertreter von Grundimplementationen ist der Hashverbund, der seinem Namen entsprechend Hash-Techniken zur Berechnung des Verbunds ausnutzt. Die Grundstruk-

Algorithmus 2 Die Grundstruktur des Merge-Verbunds bezüglich des natürlichen Verbunds $r_A \bowtie r_B$ und der zugehörigen Verbundmenge $I = R(r_A) \cap R(r_B)$. Hier bezeichnet $t_r^{(i)}$ das i-te Tupel einer nummerierten Relation r.

if not sorted **then**
  sort$_I(r_A)$
  sort$_I(r_B)$
end if
$i_A = 1, i_B = 1$
while $i_A \leq |r_A| \wedge i_B \leq |r_B|$ **do**
  if $t_{r_A}^{(i_A)} < t_{r_B}^{(i_B)}$ **then**
    i_A++
  else if $t_{r_A}^{(i_A)} > t_{r_B}^{(i_B)}$ **then**
    i_B++
  else
    $i = i_B$
    while $t_{r_A}^{(i_A)} = t_{r_B}^{(i_B)} \wedge i_A \leq |r_A|$ **do**
      $i_B = i$
      while $t_{r_A}^{(i_A)} = t_{r_B}^{(i_B)} \wedge i_B \leq |r_B|$ **do**
        put $t_{r_A}^{(i_A)} \| t_{r_B}^{(i_B)}$
        i_B++
      end while
      i_A++
    end while
  end if
end while

Algorithmus 3 Die Grundstruktur des Hash-Verbunds bezüglich des natürlichen Verbunds $r_A \bowtie r_B$ und der zugehörigen Verbundmenge $I = R(r_A) \cap R(r_B)$. Ohne Beschränkung der Allgemeinheit wird in diesem Fall $|r_A| \leq |r_B|$ angenommen.

for $t_a \in r_A$ **do** ▷ Build-Phase
  Erstelle eindeutigen Hashbucket bezüglich $h(t_a(I))$ und sortiere t_a ein
end for
for $t_B \in r_B$ **do** ▷ Probe-Phase
  for $t \in$ Hashbucket mit Hashwert $h(t_b(I))$ **do**
    if $t(I) = t_b(I)$ **then**
      put $t \| t_b$
    end if
  end for
end for

tur des Verfahrens läuft wie in Algorithmus 3 in zwei wesentlichen Phasen ab: der Build- und der Probe-Phase. In der Build-Phase wird jedes Tupel t_a der kleineren Relation r_A in ein im Hauptspeicher liegenden Hashbucket bezüglich des Hashwerts $f(t_a(I))$ einer Hashfunktion f eingeordnet. Dabei sollte die Hashfunktion so gewählt sein, dass jedem Tupel aus r_A genau ein Hashbucket eindeutig zugeordnet wird. In der Probe-Phase werden für jedes Tupel $t_b \in r_B$ die Tupel des Hashbuckets bezüglich des Hashwerts $f(t_b(I))$ auf Erfüllung der Verbundbedingung getestet und gegebenenfalls das verschmolzene Tupel der Ergebnisrelation hinzugefügt. Durch das einmalige Durchlaufen der Relationen ergeben sich hierbei Kosten von $\mathcal{O}(|r_A| + |r_B|)$.
In der Grundstruktur des Hashverbunds aus Algorithmus 3 wird angenommen, dass die Hashbuckets der kleineren Relation r_A komplett in den Hauptspeicher passen. Ist dies nicht der Fall, wird das Verfahren auf Partitionen von r_A und abhängig von der Variante auch teilweise auf r_B umgesetzt. Für einen Überblick solcher erweiterten Verfahren sei etwa auf [51] verwiesen.

4.1.3 Vererbung

In Abschnitt 9.1 wird eine Anfragezerlegungstechnik evaluiert, die eine PostgreSQL- bzw. Postgres-XL-spezifische Vererbungsfunktionalität nutzt. Vererbungskonzepte werden im SQL-Standard durch

1. einen klassisch relationalen und
2. einen objektrelationalen

Ansatz ermöglicht. Im Folgenden werden diese näher beschrieben. Die Darstellungen basieren auf Ausführungen aus [34]. Anschließend wird die in Postgres unterstützte Version in Kürze vorgestellt.
Der relationale Ansatz wurde in SQL:2003 eingeführt und erweitert die **create table**-Klausel um die Möglichkeit Schemata existierender Relationen zu erben und zu erweitern. Hierfür wird die **like**-Klausel in der Form

```
create table kind_relation (
    like master_relation,
    attr_1 datentyp_1,
    attr_2 datentyp_2,
          ⋮
    attr_n datenyp_n
)
```

genutzt. Die Kind-Relation enthält alle Attribute der Master-Relation und `attr_1`,...,`attr_n`. Für die objektrelationale Vererbung sind im Standard die miteinander gekoppelte

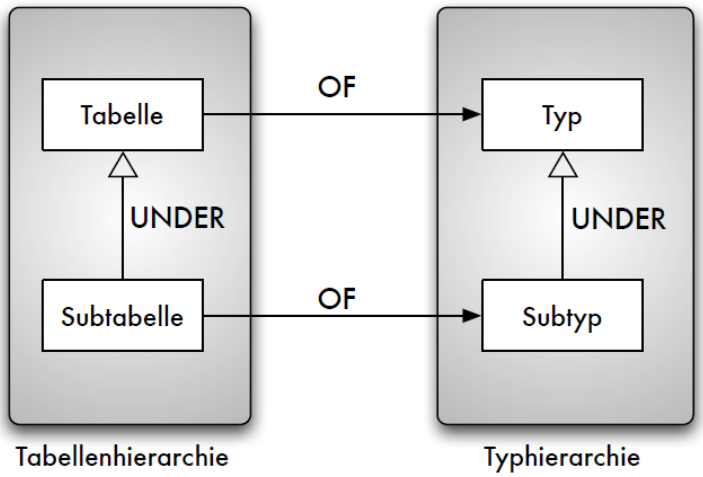


Abbildung 4.10: Die Kopplung der intensionalen und extensionalen Hierachie von Tabellen. Abbildung aus [34] entnommen.

- intensionale Hierachie und
- die extensionale Hierachie

zu unterscheiden. Wie in Abbildung 4.10 zu erkennen ist, beschreibt die intensionale Hierachie eine Typhierachie auf Objekttypen, wo hingegen die extensionale Hierachie eine Hierachie von Objekttabellen darstellt. Eine erbende Relation einer Master-Relation muss einem direkten oder indirekten Untertyp des entsprechenden Obertyps entsprechen. In SQL:2003 wird die Definition eines Untertyps mittels **create type** ... **under** umgesetzt, wohingegen die konkrete Instanzierung eines Typs durch **create table** ... **of** definiert wird.
Konträr hierzu unterstützt PosgreSQL und -XL (vergleiche Abschnitt 4.3.3) eine systemspezifische, kompaktere Variation der objektrelation Vererbung, welche von dem Standard abweicht [52, 53]. Die spezifische Vererbung wird über die **inherits**-Klausel umgesetzt, welche bei der Erstellung von Relationen in der Form

```
create table kind_relation (
    attr_1 datentyp_1,
    attr_2 datentyp_2,
          ⋮
    attr_n datenyp_n
) inherits master_relation [to data_node_name]
```

genutzt werden kann. Hierbei erbt die Kind-Relation alle Attribute der Master-Relation. Zu-

sätzlich werden die Attribute `attr_1`, ... `attr_n` definiert, die nicht in der Master-Relation existieren. In PostgreSQL kann eine Kind-Relation von mehreren Relationen erben. Attribute, die in mehreren Master-Relationen existieren, kommen nur einmalig in der Kind-Relation vor. Eine SQL-Anfrage

```
select *
from master_relation
```

liefert in diesem Fall nicht nur die Tupel, die sich in der Master-Relation befinden, sondern auch jene, die in der/den Kind-Relationen enthalten sind. Dieser Umstand wird in Abschnitt 9.1 genutzt, um eine Form der verteilten und parallelen Matrixmultiplikation in Postgres-XL zu implementieren. Hierbei wurde ebenfalls die Postgres-XL-Erweiterung **to** der **create table**-Klausel genutzt. Diese lässt den Nutzer definieren, auf welchen Datanodes (siehe Abschnitt 4.3.3) die Kind-Relationen hinterlegt werden sollen.

4.2 Basic Linear Algebra Subprograms (BLAS)

Die *Basic Linear Algebra Subprograms* (BLAS) sind eine Referenzimplementation, welche dem Namen entsprechend eine Basismenge von Operatoren der linearen Algebra zur Verfügung stellt. Zusammen mit der hier nicht näher betrachteten, darauf aufbauenden Referenzimplementation der *Linear Algebra Packages* (LAPACK), welche erweiterte Methoden des wissenschaftlichen Rechnens, wie beispielsweise Eigenwertverfahren und iterative Lösung von linearen Gleichungssystemen, zur Verfügung stellt, bilden die zugehörigen Routinen und die Schnittstellen den de-facto Standard gängiger Umgebungen des wissenschaftlichen Rechnens [54].
Es existieren verschiedene Implementationen von BLAS, wovon viele von diesen Software- und Hardware-optimiert sind und zusätzliche Funktionalitäten wie LAPACK-Methoden und weitere verbreitete Verfahren zur Verfügung stellen. Beispiele hierfür sind etwa *Intels Math Kern Library* [55, 56], *Apples Accelerate Framework* oder *CuBLAS* für NVidia GPUs. Solche Implementationen werden insbesondere von verbreiteter Software des wissenschaftlichen Rechnens (und Machine Learning), wie R [57], Matlab [58], Octave [59] oder Pythons Pakete SciPy [60] und NumPy [61] genutzt. Neben diesen werden die Implementationen auch teilweise lokal in Big-Data-Umgebungen wie beispielsweise Apache Sparks MLlib (vgl. Abschnitt 4.3.2) integriert.
Die ursprüngliche BLAS-Bibliothek wurde in den 1970er Jahren [62] entwickelt und im Laufe der Jahrzehnte mehrfach um Matrix-Vektor-Operatoren erweitert [63–65], dessen effiziente Verarbeitung durch neue technische Möglichkeiten, wie das Aufkommen von Vektorprozessoren und hierarchischen Speicherarchitekturen, ermöglicht wurden [66]. Ferner wurden Anwendungsbereiche und -szenarien mit der Zeit erweitert: So wurde in den *Sparse Basic Linear Algebra Subprograms* (SparseBLAS [66]) die Verarbeitung dünn besetzter Matrizen, die etwa in Big-Data-Szenarien

oftmals auftreten, bezüglich einer geeigneten Untermenge der ursprünglichen BLAS-Operatoren bereitgestellt. Hierfür wurden insbesondere angepasste Datenrepräsentationen und Algorithmen eingeführt. In den *Parallel Basic Linear Algebra Subprograms* [67] (PBLAS) wird eine verteilte Berechnung der wesentlichen Matrix-Vektor-Operationen mittels MPI ermöglicht, wobei lokal ebenfalls sequenzielle BLAS-Routinen genutzt werden.
Die weite Verbreitung der BLAS-Implementationen und das große Anwendungsfeld, beziehungsweise das Potenzial der Basisoperatoren effizient Methoden des wissenschaftlichen Rechnens und des Machine Learnings umzusetzen, machen die BLAS-Bibliothek zu einem gut geeigneten Referenzpunkt für die Etablierung einer Basisfunktionalität von Lineare-Algebra-Operationen in relationalen Datenbanksystemen. Wie sich in Abschnitt 5.4 zeigen wird, ermöglicht diese Überführung insbesondere die Umsetzung von Lösungsverfahren der Grundprobleme von Hidden-Markov-Modellen in SQL.
Abschließend wird ein kurzer Überblick bezüglich der zur Verfügung gestellten Operatoren und deren Kategorisierung in BLAS präsentiert. Die Informationen und Grafiken sind [66] entnommen. Zusätzlich werden genutzte Datenrepräsentationen von Matrizen aus (Sparse)BLAS gesondert in Abschnitt 6.3 vorgestellt und als Ausgangspunkt zur Entwicklung von Relationenschemata für Matrizen und Vektoren verwendet.
BLAS teilt die Operatormenge in drei wesentliche Level auf. Die meisten Operationen können für reelle oder komplexe Datentypen in einfacher oder doppelter Genauigkeit ausgeführt werden. Wie bereits beschrieben, wird nicht jede der Operationen anwendungsbedingt in Sparse BLAS oder PBLAS unterstützt. Für eine exakte Unterscheidung sei auf die angegebene Literatur verwiesen. Im Folgenden werden die drei Level in Kurzform aufgeführt.
Das erste Level umfasst skalare und Vektoroperationen. Diese lassen sich in die vier wesentlichen Kategorien [65]

1. Reduktionsoperatoren (vgl. Abbildung 4.11)
2. Vektor-Rotationen (vgl. Abbildung 4.12)
3. Vektor-Operationen (vgl. Abbildung 4.13)
4. Datenmanagement (vgl. Abbildung 4.14)

unterteilen. Die Reduktionsoperatoren beinhalten insbesondere Normen, Skalarprodukte und die Aggregatfunktionen **sum**, **min**, **max**, **argmin** und **argmax**, welche in den Lösungsverfahren der Grundprobleme von Hidden-Markov-Modellen aus den Abschnitten 5.2 und 5.3 genutzt werden.
Das zweite Level beinhaltet die in Abbildung 4.15 dargestellten Matrix-Vektor-Operationen. Hierbei handelt es sich vorrangig um Kombinationen skalierter Matrix-Vektor-Produkte, Vektoradditionen oder dyadischen Produkten $\mathbf{x}\mathbf{y}^T$. Eine speziellere Operation ist das Lösen linearer

Gleichungssystemen $\mathbf{x} = \alpha T\mathbf{x}$ mit Dreiecksmatrix T. Die Operatoren des dritten Levels bestehen aus reinen Matrix-Operationen, die in die Kategorien

1. $\mathcal{O}(n^2)$-Matrix-Operationen
2. $\mathcal{O}(n^3)$-Matrix-Matrix-Operationen
3. Datenmanagement (vgl. Abbildung 4.18)

aufgeteilt werden. Die erste Kategorie beinhaltet hierbei verschiedene Matrixnormen, Skalierungen und Matrixadditionen. Die zweite Kategorie beinhaltet maßgeblich verschiedene Varianten allgemeiner und spezieller Matrizenprodukte und Lösungsverfahren für multiple lineare Gleichungssysteme mit Dreiecksmatrix T. Die abschließende dritte Kategorie bietet Routinen zum Kopieren, Transponieren und Permutieren von Matrizen.

4.3 Parallele Systeme für Big-Data-Anwendungen

In diesem Abschnitt wird zunächst eine kurze historische Übersicht und Kategorisierung klassischer Big-Data-Systeme vorgestellt. Daraufhin wird der Aufbau und die Funktionalität des Big-Data-Systems Apache Spark und des parallelen relationalen Datenbanksystems Postgres-XL präsentiert. Beide Systeme werden in Kapitel 9 zu Vergleichszwecken genutzt.

4.3.1 Übersicht und Kategorisierung klassischer Big-Data-Systeme

Im Folgenden wird ein kurzer Überblick über die Entwicklung klassischer Big-Data-Systeme gegeben. Historischer und konzeptueller Ursprung dieser bildet das MapReduce-Framework, welches aus diesem Grund in Kürze beschrieben wird. Daraufhin wird eine Kategorisierung moderner Systeme in Kürze beschrieben. Die Ausarbeitungen stützen sich maßgeblich auf [68]. Dort können zusätzlich Informationen für Anwendungen und Kontext gefunden werden.

Das MapReduce-Framework

Das MapReduce-Framework wurde im Zuge aufkommender Probleme entwickelt, die eine massive Parallelisierung aufgrund großer zu verarbeitender Datenmengen benötigten. Beispiele hierfür sind etwa der *PageRank*, der in Beispiel 3 näher beschrieben wird, oder Analysen auf Graphdaten im Kontext sozialer Medien. Erstmalig veröffentlicht wurde das Konzept im Jahr 2004 in [69].
Das Modell unterscheidet sich im Vergleich zu vorher etablierten Ansätzen der Parallelisierung darin, dass diese — anstatt auf Supercomputern oder auf Clustern mit spezieller Hardware — auf "commodity hardware" (kostengünstige Standardhardware) aufbaut. Hierbei wird typischerweise

Dot product	$r \leftarrow \beta r + \alpha x^T y$
Vector norms	$r \leftarrow \|\|x\|\|_1, r \leftarrow \|\|x\|\|_{1R}$,
	$r \leftarrow \|\|x\|\|_2$,
	$r \leftarrow \|\|x\|\|_\infty, r \leftarrow \|\|x\|\|_{\infty R}$
Sum	$r \leftarrow \sum_i x_i$
Min value & location	$k, x_k, ; k = \arg\min_i x_i$
Min abs value & location	$k, x_k, k = \arg\min_i(\|Re(x_i)\| + \|Im(x_i)\|)$
Max value & location	$k, x_k, ; k = \arg\max_i x_i$
Max abs value & location	$k, x_k, k = \arg\max_i(\|Re(x_i)\| + \|Im(x_i)\|)$
Sum of squares	$(ssq, scl) \leftarrow \sum x_i^2$,
	$ssq \cdot scl^2 = \sum x_i^2$

Abbildung 4.11: Die Level-1-Reduktionsoperatoren aus BLAS.

Reciprocal Scale	$x \leftarrow x/\alpha$
Scaled vector accumulation	$y \leftarrow \alpha x + \beta y$,
Scaled vector addition	$w \leftarrow \alpha x + \beta y$
Combined axpy & dot product	$\begin{cases} \hat{w} \leftarrow w - \alpha v \\ r \leftarrow \hat{w}^T u \end{cases}$
Apply plane rotation	$\begin{pmatrix} x & y \end{pmatrix} \leftarrow \begin{pmatrix} x & y \end{pmatrix} R$

Abbildung 4.13: Die Level-1-Vektor-Operationen aus BLAS.

Matrix vector product	$y \leftarrow \alpha Ax + \beta y$
	$y \leftarrow \alpha A^T x + \beta y$
	$x \leftarrow \alpha Tx,\ x \leftarrow \alpha T^T x$
Summed matrix vector multiplies	$y \leftarrow \alpha Ax + \beta Bx$
Multiple matrix vector multiplies	$\begin{cases} x \leftarrow T^T y \\ w \leftarrow Tz \end{cases}$
	$\begin{cases} x \leftarrow \beta A^T y + z \\ w \leftarrow \alpha Ax \end{cases}$
Multiple mv mults & low rank updates	$\begin{cases} \hat{A} \leftarrow A + u_1 v_1^T + u_2 v_2^T \\ x \leftarrow \beta \hat{A}^T y + z \\ w \leftarrow \alpha \hat{A} x \end{cases}$
Triangular solve	$x \leftarrow \alpha T^{-1} x,\ x \leftarrow \alpha T^{-T} x$
Rank one updates	$A \leftarrow \alpha x y^T + \beta A$
and symmetric ($A = A^T$)	$A \leftarrow \alpha x x^T + \beta A$
rank one & two updates	$A \leftarrow (\alpha x) y^T + y(\alpha x)^T + \beta A$

Abbildung 4.15: Das BLAS-Level 2.

Matrix matrix product	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha A^T B + \beta C$
	$C \leftarrow \alpha AB^T + \beta C,\ C \leftarrow \alpha A^T B^T + \beta C$
	$C \leftarrow \alpha AB + \beta C,\ C \leftarrow \alpha BA + \beta C$
Triangular multiply	$B \leftarrow \alpha TB,\ B \leftarrow \alpha BT$
	$B \leftarrow \alpha T^T B,\ B \leftarrow \alpha BT^T$
Triangular solve	$B \leftarrow \alpha T^{-1} B,\ B \leftarrow \alpha BT^{-1}$
	$B \leftarrow \alpha T^{-T} B,\ B \leftarrow \alpha BT^{-T}$
Symmetric rank k & $2k$	$C \leftarrow \alpha AA^T + \beta C,\ C \leftarrow \alpha A^T A + \beta C$
updates ($C = C^T$)	$C \leftarrow \alpha AJA^T + \beta C,\ C \leftarrow \alpha A^T JA + \beta C$
	$C \leftarrow (\alpha A)B^T + B(\alpha A)^T + \beta C,$
	$C \leftarrow (\alpha A)^T B + B^T(\alpha A) + \beta C$
	$C \leftarrow (\alpha AJ)B^T + B(\alpha AJ)^T + \beta C,$
	$C \leftarrow (\alpha AJ)^T B + B^T(\alpha AJ) + \beta C$

Abbildung 4.17: Die $\mathcal{O}(n^3)$-Matrixoperatoren aus dem BLAS-Level 3.

Generate Givens rotation	$(c, s, r) \leftarrow \text{rot}(a, b)$
Generate Jacobi rotation	$(a, b, c, s) \leftarrow \text{jrot}(x, y, z)$
Generate Householder transform	$(\alpha, x, \tau) \leftarrow \text{house}(\alpha, x)$,
	$H = I - \alpha u u^T$

Abbildung 4.12: Die Level-1-Rotationssoperatoren aus BLAS.

Copy	$y \leftarrow x$
Swap	$y \leftrightarrow x$
Sort vector	$x \leftarrow \text{sort}(x)$
Sort vector & return index vector	$(p, x) \leftarrow \text{sort}(x)$
Permute vector	$x \leftarrow Px$

Abbildung 4.14: Die Level-1-Datenmanagementoperatoren aus BLAS.

Matrix norms	$r \leftarrow \|\|A\|\|_1, r \leftarrow \|\|A\|\|_{1R}, r \leftarrow \|\|A\|\|_F$,
	$r \leftarrow \|\|A\|\|_\infty, r \leftarrow \|\|A\|\|_{\infty R}$,
	$r \leftarrow \|\|A\|\|_{max}, r \leftarrow \|\|A\|\|_{maxR}$
Diagonal scaling	$A \leftarrow DA,\ A \leftarrow AD$
	$A \leftarrow D_L A D_R$
	$A \leftarrow DAD$
	$A \leftarrow A + BD$
Matrix acc and scale	$B \leftarrow \alpha A + \beta B,\ B \leftarrow \alpha A^T + \beta B$
Matrix add and scale	$C \leftarrow \alpha A + \beta B$

Abbildung 4.16: Die $\mathcal{O}(n^2)$-Matrixoperatoren aus dem BLAS-Level 3.

Matrix copy	$B \leftarrow A$
	$B \leftarrow A^T$
Matrix transpose	$A \leftarrow A^T$
Permute Matrix	$A \leftarrow PA,\ A \leftarrow AP$

Abbildung 4.18: Die Level-3-Datenmanagementoperatoren aus BLAS.

Alle Tabellenausschnitte sind [66] entnommen.

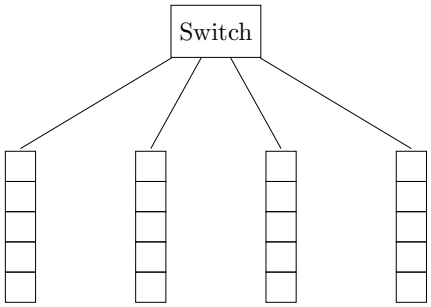


Abbildung 4.19: Knotenorganisation in distributed file systems. Mehrere Knoten sind in einem rack organisiert. Racks sind über Breitbandnetzwerk oder Switch verbunden. Idee entnommen aus [68].

eine sehr hohe Anzahl handelsüblicher Rechner genutzt, welche mit günstigen Switches oder Ethernet verbunden sind. Aufgrund dieser vergleichsweise fragilen Architektur wird kritischen Fehlern durch Knotenausfällen mittels gezielter Replikation entgegengewirkt.
Die Basis zum Arbeiten auf gemeinsamen Datensätzen wird in diesem Modell über verteilte Dateisysteme („distributed file systems“) umgesetzt.

Das Distributed File System (DFS)

Das DFS bildet die lokalen Dateisysteme der agierenden Knoten auf ein geschlossenes ganzes Dateisystem ab und macht dieses zugänglich für die jeweiligen Berechnungsknoten. Die wohl bekannteste und weit-verbreitetste Implementation ist das *Hadoop Distributed File System* (HDFS), welche im Zuge der MapReduce-Implementation Apache Hadoop entwickelt wurde.
In verteilten Dateisystemen werden Berechnungsknoten in „racks“ organisiert (vergleiche Abbildung 4.19). Die Knoten eines racks (klassischerweise zwischen 8 und 64 [68]) sind untereinander etwa über ein Gigabit-Ethernet-Netzwerk verbunden. Racks werden über eine weitere Stufe von Netzwerken mit hoher Bandbreite oder etwa per Switch verbunden.
Die Architektur ist derart konzipiert, dass transparent eine gute Lastverteilung sichergestellt wird und Knotenausfälle im Allgemeinen keine kritischen Folgen für laufende Berechnungen mit sich führen. Dateien werden hierfür in (typischerweise) 64 Megabyte große Teilstücke (”chunks”) aufgeteilt und repliziert in verschiedenen racks gespeichert. Dies gewährleistet nicht nur, dass Daten bei einem Knotenausfall nicht verloren gehen, sondern auch, dass zugehörige Aufgaben innerhalb eines anderen Subnetzwerkes (rack) ausgelagert werden können. Die Pfade der einzelnen

chunks im Dateisystem werden für jede Datei in einer kleineren Datei namens „master node“ oder „name node“ dokumentiert, welche ebenfalls repliziert abgelegt werden. Dieses Konzept führt zu einen erheblichen Overhead für jede Datei und ist darauf ausgelegt, mit sehr großen Datenmengen (hunderte Gigabytes bis Terabytes) zu agieren. Ferner sollten zu behandelnde Daten aufgrund der häufigen Replikation nur selten manipuliert werden.

Programmiermodell und Funktionsweise

Zur Implementation von Programmen werden vom Nutzer, dem Modellnamen entsprechend, nur die zwei wesentlichen *Map*- und *Reduce*-Funktionen spezifiziert. Das System setzt die parallele Verarbeitung, Koordination und Behandlung von Knotenausfällen transparent um (vergleiche Abbildung 4.20). Hierfür erhalten mehrere Map-Tasks chunks von Inputdateien, welche sie laut programmierter Map-Funktion in Schlüssel-Wert-Paare (k,v) überführen. Die chunks der Inputdateien bestehen aus einer Menge von Elementen eines beliebigen Typs wie beispielsweise Tupel oder etwa Dokumente. Der Begriff „Schlüssel“ ist hierbei nicht zu verwechseln mit dem einer eindeutig identifizierenden Attributmenge von Tupeln im Kontext relationaler Datenbanksysteme, sondern ist als Gruppenidentifikator ähnlich eines Hash-Werts zu verstehen. Die errechneten Schlüssel-Wert-Paare werden vom *Master-Controller*-Prozess auf dem Masterknoten gesammelt und dort nach Schlüssel sortiert. Zusätzlich werden hier lokal tasks erstellt, gelöscht und überwacht. Im Allgemeinen ist der Masterknoten nicht an der eigentlichen Berechnung von Map- oder Reduce-Tasks beteiligt, kann aber auch gegebenenfalls als Rechenknoten eingeteilt werden. Nach der Sortierung werden die Paare mehrerer Map-Tasks gesammelt und in die Form $(k,[v_1,\ldots,v_n])$ gebracht. Es werden daraufhin r (Anzahl von Reduce-Tasks) lokale Dateien erstellt, in welche die Schlüssel mit ihren gruppierten Werten mittels (selbst programmierbarer) Hash-Funktion aufgeteilt werden. Jede Schlüssel-Wert-Gruppierung wird hierbei in genau eine Datei geschrieben und an genau einen Reduce-Task verschickt. Diese Phase wird auch als „Shuffle“-Phase bezeichnet. Die Reduce-Tasks verarbeiten die erhaltenen Schlüssel-Wert-Gruppierungen gemäß der vom Nutzer programmierten Funktion. Hierbei können keine bis mehrere Schlüssel-Wert-Paare den Output darstellen, welche pro Reduce-Task in eine Datei im DFS hinterlegt werden. Die Form des Outputs ermöglicht es hierbei, den Output als Input einer anderen MapReduce-Iteration zu nutzen, wodurch Schleifen und ähnliches modelliert werden können. Typischerweise werden Rechenknoten entweder als Reduce-Worker oder als Map-Worker eingeteilt, allerdings ist es auch möglich, beides auf einem Knoten durchzuführen. Im Folgenden wird das wohl gängigste einführende Beispiel zum MapReduce-Programmiermodell vorgestellt: Das Zählen einzelner Wörter in einer Vielzahl von Dokumenten. Dieses Szenario war insbesondere Bestandteil der in der Einführung beschriebenen Vergleichstests von *Stonebraker et al.* aus [3].

Beispiel 1 (Wordcount)**.** Eines der klassischen Beispiel von MapReduce-Implementierungen stellt das Zählen einzelner Wörter in mehreren Dokumenten dar. Hierbei entsprechen Dokumente den einzelnen Elementen und alle vorkommenden untereinander verschiedenen Wörter entsprechen jeweils Schlüsseln. Es existieren zwei wesentliche Beschreibungen der Map-Tasks. Bei der ersten erstellt die Map-Funktion aus den Dokumenten Paare der Form

$$(\mathrm{w}_1,1),(\mathrm{w}_2,1),...$$

wobei w_i dem $i-$ten Wort entspricht. Das heißt, wenn ein Wort m-mal in den Eingangsdokumenten eines Map-Tasks vorkommt, existieren m verschiedene Output-Tupel mit dem Wert 1. In der Reduce-Phase würden Reduce-Tasks pro Wort alle erhaltenen Tupel aller Map-Tasks aggregieren und im Dateisystem hinterlegen. Im zweiten Ansatz wird die Anzahl der Wörter noch innerhalb der Map-Funktion aggregiert, so dass etwa das Tupel (w_i,m) erzeugt wird. Dieser Ansatz ist offensichtlich effizienter im Sinne der Kommunikationskosten. Allgemeiner kann gesagt werden: wenn die Reduce-Funktion assoziativ und kommutativ ist (wie beispielsweise die Summation), kann ein Teil der Funktion in die Map-Funktion geschoben werden, welches sich oftmals positiv auf die Netzwerkkosten auswirkt. Generell ist es für Implementationen nicht sinnvoll, viele Knoten für wenig Reduce-Funktionen zu nutzen, da die Reduce-Tasks sehr unterschiedlich aufwändig sein können, was etwa zu langen Wartezeiten einzelner Knoten führen könnte. Zusätzlich kann auch eine hohe Anzahl an Knoten sich ungünstig auf die Performance auswirken, da jede Map-Task für jeden Reduce-Task eine Datei erstellen muss. Ferner muss jede Map-Task für jede Reduce-Task eine Datei schreiben, welches bei einer großen Anzahl von Reduce-Tasks zu erheblichem Overhead führen kann. In [68] werden weitere Umsetzungsmöglichkeiten für Verfahren und Methoden in MapReduce diskutiert. Insbesondere werden hierbei die im Verlauf der Arbeit näher betrachteten Basis-Lineare-Algebra-Operatoren, wie (dünn besetzte) Matrix-Vektor- und Matrizenmultiplikationen, untersucht.

Behandlung von Knotenausfällen

Der ungünstigste Fall entsteht, wenn der Masterknoten (einzelne Instanz; nur zuständig für Scheduling [master controller Prozess]) ausfällt, da dann der gesamte MapReduce-Job neugestartet werden muss. Dies ist der einzige single point of failure im Konzept. Fällt etwa ein Knoten mit Map-Workern aus, welches durch regelmäßiges anpingen der Knoten detektiert wird, müssen alle Map-Tasks neu berechnet werden. Dies liegt daran, dass der gesamte Output der Map-Funktionen lokal geschrieben wird. Der Master organisiert die Tasks auf andere Worker um, setzt die Map-Tasks auf idle und sendet den Reduce-Tasks die neue Input-Adressen. Fällt hingegen ein Knoten mit Reduce-Workern aus, setzt der Master lediglich die Reduce-Tasks auf idle und plant neue Worker ein.

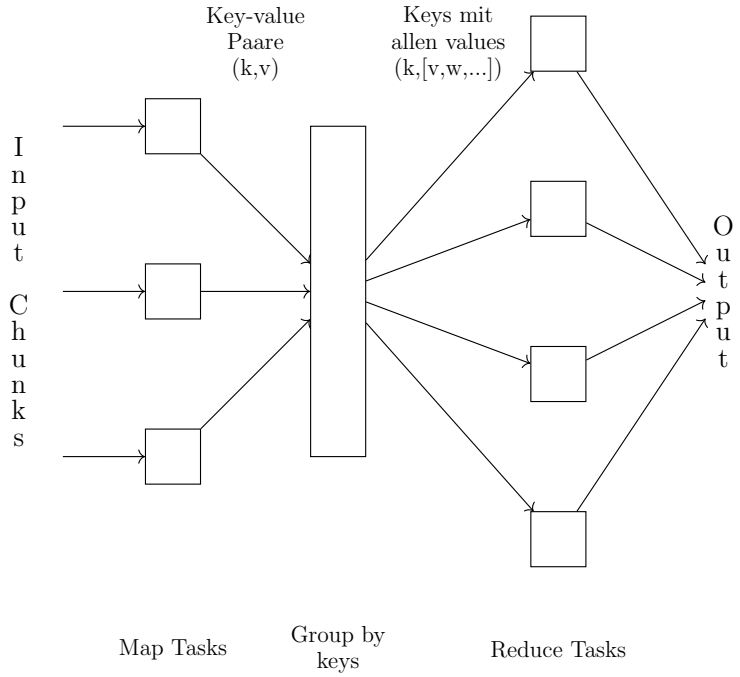


Abbildung 4.20: Schematische Darstellung eines MapReduce-Verfahrens. Idee entnommen aus [68].

Erweiterungen von MapReduce: Workflow-Systeme

Mit ansteigender Popularität datenintensiver Szenarien und des MapReduce-Frameworks sind eine Vielzahl von Entwicklungen und Forschungsprototypen einhergegangen, die verschiedene aufsetzende Schnittstellen diskutiert und umgesetzt haben. Beispiele hierfür sind etwa

- *Apache Hive* [70], welches Data-Warehouse-Funktionalitäten inklusive SQL-ähnlicher Sprache und Indexstrukturen auf Apache Hadoop umsetzt(e).
- *SimSQL*, welches ein paralleles relationales System auf MapReduce umsetzte und Zusatzfunktionalitäten wie statistische Simulationen [71] oder parallele Lineare-Algebra-Operationen [72] (vgl. weitere Ausführungen in Abschnitt 4.4) einbettet.
- *SystemML*, welches initial von IBM als Machine-Learning-Aufsatz auf MapReduce entwickelt wurde [73, 74] und durch ein R-ähnliche Sprache transparente Parallelität in gewohnter Data-Science-Umgebung ermöglicht.

Im Kontext von Machine-Learning-Verfahren wurde jedoch vergleichsweise schnell festgestellt, dass das klassische MapReduce-Modell ungeeignet für die Verarbeitung solcher Verfahren ist [75]. Gründe hierfür sind etwa der oftmals iterative Charakter von Machine-Learning-Verfahren, der im Kontrast zum vergleichsweise starren Map-Shuffle-Reduce-Konstrukt steht. Big-Data-Projekte setzen heutzutage daher primär auf MapReduce-erweiternden Workflow-Systemen auf, welche sich aufgrund der beschriebenen Einschränkungen entwickelt haben. Diese Systeme bauen im Allgemeinen ebenfalls auf verteilten Dateisystemen auf, nutzen große Menge von Berechnungsknoten zur transparenten parallelen Berechnung einer kleinen Menge von strukturierten Funktionen und setzen transparent in der Laufzeit Strategien zur Behandlung von Knotenausfällen um [68]. Workflow-Systeme werden in [68] in zwei wesentliche Gruppen aufgeteilt:

1. Systeme, die Daten mittels azyklischer Netzwerke von Funktionen umsetzen, wobei jede Funktion durch eine Menge von Tasks umgesetzt wird.

2. Systeme, die Daten mittels Graphenmodellen umsetzen, indem auf jedem Knoten Operationen durchgeführt werden und Zwischenergebnisse an benachbarte Knoten weiter geschickt werden.

Prominenter Vertreter der letzteren Gruppe ist Google Pregel [76], welcher gleichzeitig Pionier dieser Gruppe von Systemen ist. Im Folgenden fokussieren wir uns anwendungsbezogen jedoch auf erstere Gruppe. Solche Workflow-Systeme erweitern das starre 3-Phasen-Map-Shuffle-Reduce-Konstrukt durch eine Menge von Funktionen, welche durch einen azyklischen Workflow dargestellt werden können. Konträr zum MapReduce-Modell können durch diesen Ansatz Zwischenergebnisse effizient und einfach mehrfach genutzt werden. Ein Beispiel ist hierfür in Abbildung 4.21 dargestellt. Dort ist das durch „f“ berechnete Zwischenergebnis Input der Funktion „i“ und der Funktion „j“. Die Kommunikation geschieht analog zum MapReduce-System über Dateien, wobei Zwischenergebnisse einzelner Operatoren im Allgemeinen nicht ins verteilte Dateisystem geschrieben werden. Die Funktionen führen ihre Operationen im Falle einer einzelnen Inputdatei (etwa „g“) auf jedem Eintrag dieser gesondert aus. Im Falle mehrerer eingehender Dateien (etwa „i“) rechnen die Funktionen auf einer (zu spezifizierenden) Kombination der verschiedene Inputs, wobei jedes einzelne Element maximal einmal genutzt wird [68].
Ebenfalls analog zu MapReduce wird eine Funktion in mehrere Tasks aufgeteilt, die jeweils durch eine große Anzahl an Knoten auf den jeweiligen Teilen der Inputdaten verarbeitet werden. Ein Master-Controller-Prozess ist hierbei für die zugehörige Verteilung anfallender Berechnungen zuständig. Dies bezieht ebenfalls die zeitgerechte Koordination der Zwischenergebnisdateien ein. Letztere werden hierbei durch einen Task erst dann produziert, wenn dieser erfolgreich abgeschlossen wurde. Im Falle eines Fehlers kann so der Task vom Master-Controller-Prozess auf einem anderen Knoten neugestartet werden.

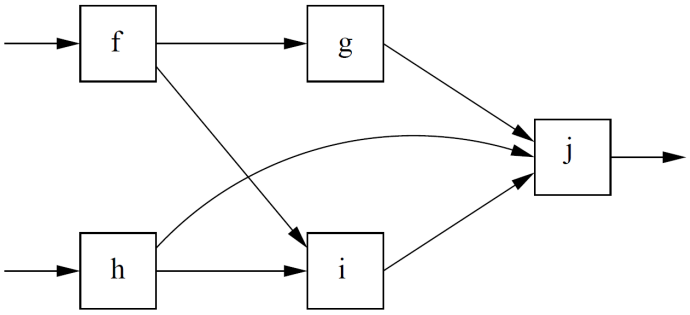


Abbildung 4.21: Schematische Darstellung eines azyklischen Datenflussgraphen. Grafik aus [68] entnommen.

Die Art der Implementation, etwa die der einzelnen unterstützten Operatoren oder zusätzlicher Funktionalitäten, ist hierbei systemabhängig. Es existieren folglich mehrere Vertreter, die sich prinzipiell in die Kategorie der Workflow-Systeme einordnen lassen. Diese enthalten oftmals auch erweiternde Funktionalitäten, wie Schleifenkonstrukte, welche die Zuordnung technisch betrachtet, jedoch nicht zutreffend macht. Namhafte Beispiele dieser Art sind etwa

- Microsoft's Naiad [77],
- Apache Flink (entstanden aus dem Projekt „Stratosphere“ [78]),
- Google Tensorflow [79], und
- Apache Spark [80].

Hierbei ist mitunter in den letzten Jahren eine steigende Popularität an Apache Spark zu vernehmen [68]. So wurde beispielsweise der Entwicklungsfokus der vorher erwähnten Projekte Apache SystemML und Apache Hive von einer MapReduce-basierten Verarbeitung auf die Nutzung von Apache Spark als Backend gewechselt. Aus diesen Gründen wurde in Abschnitt 9.2 eine experimentelle Vergleichsevaluation des hier propagierten Datenbankansatzes mit Apache Spark vorgenommen. Hierfür werden in den anschließenden Unterabschnitten die Systeme Apache Spark und das genutzte parallele relationale Datenbanksystem Postgres-XL näher vorgestellt.

4.3.2 Apache Spark

Im Folgenden wird das Cluster-Computing-System Apache Spark näher vorgestellt. Das System entspringt einem Projekt der Universität von Kalifornien, Berkeley und wurde im Jahr 2010 zum Open-Source-Projekt unter BSD-Lizensierung. Apache Spark ist im Kern Vertreter

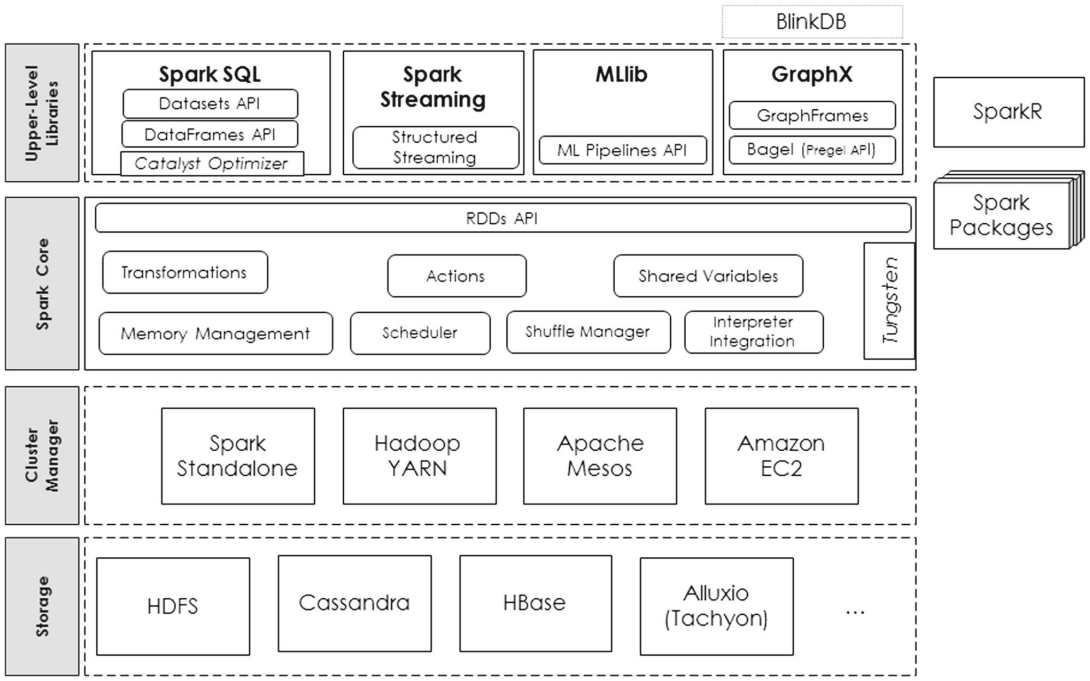


Abbildung 4.22: Architektur von Apache Spark. Grafik entnommen aus [80] .

der MapReduce-erweiternden Workflow-Systeme [68] und gilt aufgrund seiner hauptspeicherinternen Berechnungsmöglichkeiten und seiner Skalierbarkeit für Machine-Learning- und Graphproblemen als „de facto Standard für Big-Data-Analysen“ [80] (Stand 10/2016) [1]. Aus diesem Grund wurde das System genutzt, um in Abschnitt 9.2 Vergleichswerte für den hier propagierten SQL-basierten Ansatz auf parallelen relationalen Datenbanksystemen im Kontext von Lineare-Algebra-Operatoren zu erhalten. Die folgenden Ausarbeitungen beziehen sich hierbei auf [68] und [80].

Aufbau von Apache Spark

Apache Spark setzt auf Apache Hadoop auf und besteht grundlegend aus den, in Abbildung 4.22 dargestellten, vier Schichten: Storage, Cluster Manager, Spark Core und den darauf aufsetzenden High-Level-Bibliotheken. Wie in der Abbildung erkennbar ist, kann das System Daten aus verschiedenen Quellen, inklusive des eingebundenen Hadoop Distributed File System (HDFS), verarbeiten. Der Kern (Spark Core) baut auf verschiedene Cluster Manager auf, welche den Res-

[1] wörtliches Zitat: ¨Apache Spark has emerged as the de facto framework for big data analytics with its advanced in-memory programming model and upper-level libraries for scalable machine learning, graph analysis, streaming and structured data processing¨ [80]

sourcenaustausch zwischen Spark-Anwendungen regeln. Spark Core ist in Scala implementiert und bietet neben dieser auch APIs für Java, Python und R an.
Die wesentliche interne Datenabstraktion geschieht in Spark über die *Resilient Distributed Datasets* (RDDs), welche die zentrale Schnittstelle im Kern von Spark für Anwendungen und aufsetzende Bibliotheken darstellen und im folgenden Unterabschnitt näher beschrieben werden. Auf Grundlage dieser RDDs verbindet der Kern eine Engine für verteiltes Rechnen mit einem Programmiermodell, welches besonderen Fokus auf lokal hauptspeicherorientierte Berechnungen setzt.
Wie der Architekturdarstellung (Abbildung 4.22) zu entnehmen ist, wird durch die auf dem Kern aufsetzende, offiziell integrierte Funktionsbibliothek Spark SQL [81] zusätzlich die Möglichkeit der Anfragestellung mittels SQL ermöglicht. Neben dieser Bibliothek sind die Stromdatenbibliothek SparkStreaming [82], die Graphverarbeitungsbibliothek GraphX [83] und die Machine-Learning-Bibliothek MLlib [84] Bestandteil des Systems. Aufgrund seines Open-Source-Charakters existieren noch deutlich mehr Pakete (2016: >200). Diese sind jedoch nicht Teil des offiziellen Apache-Spark-Projekts.
Nach einer Beschreibung des Aufbaus und der Verarbeitung der zugrundeliegenden Resilient Distributed Datasets werden die auf dem Kern aufbauenden Bibliotheken Spark SQL und MLlib vorgestellt, welche in den Auswertungen aus Abschnitt 9.2 genutzt worden sind. Weiterführende interne Implementationsdetails und Funktionalitäten von Spark-Programmen, des Spark Cores und weitere Aspekten werden anwendungsbedingt hier vernachlässigt. Hierfür sei auf die angegebene Literatur verwiesen.

Resilient Distributed Datesets (RDDs)

Apache Spark RDDs sind eine Sammlung von Objekten eines (zusammengesetzten) Typs, die als In- und Output der internen Funktionen des (erweiterten azyklischen) Workflows genutzt werden. Die RDDs sind verteilt und werden, wie in MapReduce, in chunks (in Spark werden diese als „splits“ bezeichnet) aufgeteilt und transparent auf verschiedenen Knoten gehalten.
In Apache Spark werden zwei Typen von Funktionen des Workflows unterschieden: *Transformationen* („transformations“) und *Aktionen* („actions“). Eine Transformation ist eine Operation, die eine Funktion auf ein RDD anwendet und ein RDD als Output produziert. In einer Aktion werden Daten entweder aus einer Datenquelle (etwa dem HDFS) gelesen und in ein RDD überführt oder es wird eine RDD in ein spezifisches Dateiformat überführt und in das jeweilige umliegende Dateisystem zurückgeschrieben beziehungsweise an eine Anwendung gesendet, die das Spark-Programm aufgerufen hat. Im Gegensatz zum starren Drei-Phasen-Modell von MapReduce existiert in Spark eine deutlich größere und flexiblere Menge an Transformationen und Aktionen. Beispiele für Transformationen sind etwa *map*, *filter* oder *join* und für Aktionen etwa *reduce* oder *count*.

Ein wesentlicher Aspekt des hauptspeicherorientierten Programmiermodells ist die *lazy evaluation* in Spark. Wie bereits für Workflow-Systeme beschrieben, wird der Output einer Funktion auf einem chunk/split nur dann ausgegeben, wenn die komplette Berechnung erfolgreich verlief. Dies ermöglicht einen effizienten Umgang mit Knotenausfällen. In Spark werden die Transformationen auf RDDs erst dann ausgeführt, wenn es wirklich nötig ist (etwa bei nötiger Repartitionierung (shuffle) oder bei einem Aufruf einer Aktion). Dadurch können im besten Fall RDD-Zwischenergebnisse einzelner Transformationen nicht physisch gespeichert und verteilt werden, da sie lokal als Input von der oder den nächsten Transformation/en direkt genutzt werden können. Dieser Mechanismus ermöglicht die Umsetzung von hauptspeicherinternen lokalen Iterationen.
Dadurch bedingt wird für den Umstand eines Knotenausfalls die *lineage* der RDDs mitgeführt. Dies entspricht intern einer detaillierten Beschreibung der Transformationsfolge der chunks/splits von RDDs. Da im Vergleich zu MapReduce keine Zwischenergebnisse nach Transformationen physisch gespeichert werden, kann der Rekonstruktionsprozess deutlich komplexer sein als in solchen Systemen. Im Allgemeinen überwiegt jedoch der Performance-Gewinn, durch die Aussparung der Disk-Operationen, deutlich gegenüber dem Nachteil der komplexeren Wiederherstellung [68].

Spark SQL

Die Bibliothek Spark SQL ermöglicht optimierte relationale Berechnungen in Sparkanwendungen und auf externen Daten. Hierfür wurde die deklarative DataFrame-Schnittstelle und die erweiterbare Anfrageoptimierungsschnittstelle Catalyst (vgl. Abbildung 4.22) eingeführt. Im Folgenden werden diese, basierend auf Ausführungen in [81], beschrieben.

DataFrames sind die wichtigste Datenabstraktion in Spark SQL und stellen eine Sammlung von Tupeln bezüglich eines homogenen Schemas dar. Sie sind als das Äquivalent zu Relationen in Datenbanksystemen zu verstehen und lehnen sich strukturell an den DataFrames von R und Python an. Sie sind spaltenweise (vgl. Ausführungen zur Relationenschemawahl in Abschnitt 6.2) hinterlegt, können aber auch als Row-Objekte von RDDs transformiert werden. Auf diese Weise ist es möglich, neben den relationalen Operationen auch prozedurale RDD-Funktionen (wie beispielsweise **map**) zu nutzen. DataFrames können aus RDDs oder aus Daten externer Quellen (wie etwa Datenbanken) erstellt werden. Aufgebaute DataFrames können mit verschiedenen relationalen Operatoren, wie **where** oder **groupBy**, manipuliert werden. Analog zu RDDs werden DataFrames erst dann materialisiert, wenn diese benötigt werden (lazy evaluation). Jedes DataFrame repräsentiert hierbei ein logisches Objekt, sodass Optimierungen auch funktionsübergreifend umgesetzt werden können. Spark SQL unterstützt alle wesentlichen SQL-Datentypen, inklusive User Defined Datatypes. Neben der Nutzung relationaler Operatoren

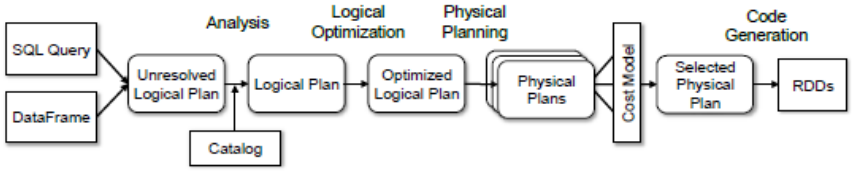


Abbildung 4.23: Die verschiedenen Phasen der Anfrageplanung in Spark SQL. Grafik entnommen aus [81]. Abgerundete Rechtecke entsprechen hierbei Bäume in Catalyst.

ist es möglich, SQL-Anfragen nach der Registrierung von DataFrames als temporäre Tabellen zu stellen. Hierbei können insbesondere UDFs in der generellen Spark-API registriert und in SQL-Anfragen genutzt werden.

Catalyst ist die erweiterbare Optimierungsschnittstelle von Spark SQL, die auf Konstrukten der funktionalen Programmierung in Scala basiert. Die Erweiterbarkeit der Schnittstelle ist, neben der möglichen Nachimplementation von Optimierungsregeln, für die Erstellung von Push-Down-Optimierungen in neue Datenquellen, wie beispielsweise relationale Datenbanksysteme, ausgelegt worden. Durch die Verwendung nativer Scala-Funktionalitäten ist die Implementation solcher Optimierungsregeln vergleichsweise unkompliziert und kompakt möglich [81].
Catalyst stellt Anfragepläne als Anfragebäume dar und setzt Optimierungsregeln maßgeblich über pattern matching um, bei dem Subbäume des Anfragebaums gefunden und geeignet ersetzt werden. Wie in Abbildung 4.23 dargestellt, werden aus den logisch optimierten Anfragebäumen („logical plan“) mehrere physische Verarbeitungspläne („physical plan“) erstellt, von denen nach einer Kostenabschätzung der effizienteste Plan ausgewählt und daraufhin in Spark ausgeführt wird.
Für weitere Implementationsdetails und zusätzlicher Funktionalitäten sei abermals auf [81] verwiesen.

MLlib

MLlib ist Apache Sparks Bibliothek für verteilte Machine-Learning-Workflows [84]. Die Bibliothek setzt skalierbare Standard-Lernverfahren für klassische ML-Problemklassen, wie Klassifizierung, Regression, kollaboratives Filtern oder etwa Dimensionsreduktion (etwa Hauptkomponentenanalysen, vgl. Anhang B.6.2) um. MLlib besteht aus den zwei Paketen *spark.mllib*, welches auf der RDD-Schnittstelle aufsetzt, und *spark.ml*, welches auf der DataFrame-Schnittstelle arbeitet [81]. Beide Pakete bieten wesentliche Funktionalitäten für Machine-Learning-Szenarien, wie etwa Feature-Extraction und -Transformation oder Modell-Training an.

Das Hauptziel des Paketes *spark.ml* ist hierbei die Umsetzung vollständiger ML-Pipelines. Dies beinhaltet etwa das Preprocessing der Daten, die Feature Extraction, das Model Fitting und die Validierung des Modells. Für die Integration von Daten und das effizienten Preprocessing, welches etwa Data Cleaning beinhaltet, können hierbei insbesondere Funktionalitäten von Spark SQL, aufgrund der Nutzung von DataFrames, einbezogen werden. [84]
Das Paket *spark.mllib* ermöglicht weitere Funktionalitäten, wie die parallele Berechnung von Statistiken oder von Lineare-Algebra-Operatoren. Letztere werden in MLlib über das Paket *linalg* [85] realisiert. Dieses bietet Matrizendarstellungen für dicht und dünn besetzte Matrizen und Vektoren an, sowie verschiedene Partitionierungsstrategien für Matrizen (etwa Coordinate- (vgl. Abschnitt 6.2), Block-, oder Zeilendarstellungen (vgl. Abschnitt 8.1)). Zusätzlich werden neben klassischen Basisoperationen, wie Matrixmultiplikationen, auch erweiterte Methoden wie Singulärwertzerlegung oder Minimierungsverfahren, wie Gradient Descent, unterstützt. Basisverfahren werden insbesondere in *linalg* über das in Abschnitt 4.2 präsentierte BLAS-Interface durch jeweilige Implementationen lokal hardwareoptimiert verarbeitet. Da im weiteren Verlauf der Arbeit sowohl dichte als auch dünn besetzte Matrixberechnungen im Fokus der hier propagierten SQL-basierten Verarbeitung auf parallelen relationalen Systemen sind, wurden die zugehörigen *linalg*-Operatoren für Vergleichsrechnungen aus Abschnitt 9.2 genutzt.
Für detaillierte Beschreibungen über Funktionalitäten (etwa Aufzählungen unterstützter ML-Modelle und -Methoden) und deren Implementierung sei auf [81] und [84] verwiesen.

4.3.3 Postgres-XL

Im Folgenden wird das quelloffene parallele relationale Datenbanksystem Postgres-XL [86] vorgestellt. Das System basiert auf dem namensverwandten PostgreSQL (auch: Postgres), welches ebenfalls in experimentellen Auswertungen im Laufe der Arbeit als Vertreter klassischer (objekt-)relationaler Systeme (mit zeilenbasierter Speicherarchitektur) genutzt wird. Dessen Ursprünge reichen bis in das POSTGRES-Projekt der Universität von Kalifornien, Berkeley aus dem Jahr 1986 zurück, welches seitdem aktiv weiterentwickelt wurde [87].
Postgres bietet sich für experimentelle Auswertungen, neben seinem freien und quelloffenen Charakter, für das hier propagierte Framework an, da es sehr nahe am SQL-Standard ist. Laut offizieller Dokumentation [87] unterstützt Postgres aktuell (Stand 12/2022) 160 von 179 Features vom SQL:2016-Kern, die zur offiziellen Konformität des Systems nötig sind. Hier wird insbesondere der objektrelationale Aspekt und die damit einhergehende Unterstützung von Array-Datentypen für die Diskussion geeigneter Relationenschemata in Abschnitt 6.2 und zur effizienten Zeitreihenverwaltung in Abschnitt 9.4 genutzt. Zusätzliche Funktionalitäten, wie detaillierte Einsichten und Möglichkeiten zur Manipulation von Anfrageplänen, die Unterstützung von Multicore-Processing, die Konfigurierbarkeit zahlreicher Systemparameter, eine vergleichsweise große Auswahl an klassischen und experimentellen Indexstrukturen (vgl. Abschnitt 4.1.1) und

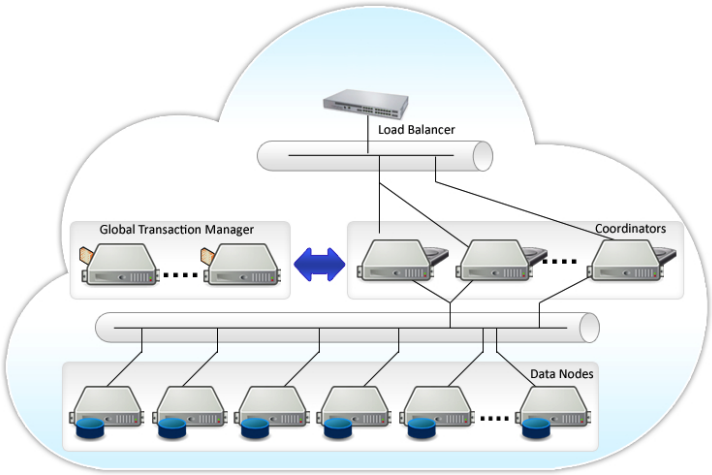


Abbildung 4.24: Grafische Darstellung eines Postgres-XL-Clusters. Grafik entnommen aus [86].

eine BSD-ähnliche-Lizenz machen das System zu einem potenziellen Vertreter für Forschungs- und Anwendungszwecke.
Im Sinne der hier diskutierten Big-Data-Szenarien wird im Folgenden die Architektur des parallelen Systems Postgres-XL vorgestellt. Die spezifische Konfiguration des hier genutzten Clusters und Systems wird in Abschnitt 9.1 vor der Beschreibung der Experimentalergebnissen beschrieben.

Architektur von Postgres-XL

Ein Postgres-XL-Cluster besteht im wesentlichen aus den drei Komponenten (vgl. Abbildung 4.24):

- Global Transaction Managers (GTMs),
- Coordinators, und
- Datanodes.

Der **GTM** ist maßgeblich verantwortlich für die Wahrung der Transaktionssicherheit und setzt diese mittels Multi-Version Concurrency Control (MVCC) (vgl. etwa [51]) um. Damit wird ein eindeutiger globaler Status von Transaktionen, die von Coordinators und Datanodes bearbeitet werden, durch GTMs sichergestellt. Postgres-XL unterstützt zudem GTM-Proxys, die Verbindungen und Interaktionen zwischen GTMs und Coordinators und Datanodes gruppieren können. Dadurch kann die aktive interne Kommunikation zwischen Instanzen signifikant verringert werden, welches neben dem Einsparen von Ressourcen auch zu Laufzeitverbesserungen führt.
Coordinators stellen die Schnittstelle zwischen Datenbank und Nutzer in Postgres-XL dar. Diese sind aufgebaut und arbeiten wie klassische PosgreSQL-Instanzen, speichern jedoch selbst

keine Daten ab. Erhält ein Coordinator ein SQL-Statement, fragt dieser eine globale Transaktionsnummer (GXID) und einen zugehörigen Snapshot des aktuellen Datenbankzustands an. Nach Erhalt entscheidet der Coordinator, welche Datanodes benötigt beziehungsweise genutzt werden sollen, um die jeweilige Anfrage zu verarbeiten. Handelt es sich um Leseanfragen auf replizierten Daten, bevorzugt der Coordinator Datanodes, die auf dem gleichen Knoten laufen. Im allgemeinen Fall werden Subanfragen des originalen Statements mit den zugehörigen Transaktionsinformationen (GXID und Snapshot) durch den Coordinator an die jeweiligen Datanodes gesendet. Gegebenenfalls führt der Coordinator nach Erhalt von materialisierten Zwischenergebnissen finalisierende Restoperationen der Anfrage aus. Ein Beispiel hierfür sind etwa algebraische und distributive Aggregatfunktionen aus Definition 9. Der Coordinator versucht bei der Erstellung von Subanfragen unter anderem folgende Operationen einer Anfrage in die Subanfrage eines Datanodes einzubinden:

- die **where**-Klausel
- Verbundberechnungen
- Projektionen
- die **order by**-Klausel

Datanodes sind im wesentlichen ebenfalls PostgreSQL-Instanzen und sind für den Kern der Anfrageverabeitung und Datenspeicherung zuständig. Relationen können hierbei auf verschiedenen Datanodes partitioniert oder repliziert werden. Als native Verteilungsstrategie werden Hash/Modulo- und RoundRobin-Strategien oder eine vollkommene Replikation unterstützt (Stand Postgres-XL 10.1). Datanodes haben keine globale Sicht auf die Datenbank. Sie arbeiten ausschließlich mit den lokal vorhandenen Daten. Eingehende Subanfragen können durch die zusätzlichen Snapshot- und Transaktionsnummern im jeweiligen Datenbankzustand ausgeführt werden.
Laut offizieller Dokumentation wird empfohlen, den GTM auf einem Knoten ohne Coordinators und Datanodes aufzusetzen. Ferner ist es sinnvoll, Coordinatoren und Datanodes gruppiert auf lokalen Knoten zu platzieren, um Netzwerkkosten zu sparen. Da die Instanzen abhängig voneinander agieren, werden bei logischer Anordnung lokale Rechenressourcen im Allgemeinen nicht blockiert (vgl. Subanfragen und Finalisierung durch Coordinatoren).
Ein weiterer hier wesentlicher Aspekt der Umsetzung von Postgres-XL ist der interne Verbindungsaufbau zwischen Coordinators und Datanodes. In Setups mit vielen solcher Instanzen können vergleichsweise große Mengen an aktiven und inaktiven Verbindungen unter diesen entstehen. Zur ressourcenschonenden Verwaltung von Verbindungen werden diese in Postgres-XL über einen Connection-Pool wie folgt umgesetzt. Wenn ein Coordinator eine Verbindung zu einem Datanode benötigt, sucht ein Pooler-Prozess nach einer möglichen Verbindung im Pool und

weist diese dann dem Coordinator zu. Nach der Nutzung der Verbindung wird diese wieder vom Pooler für spätere Anwendungen weiter verwaltet, ohne diese explizit aufzulösen. Dadurch wird eine Reduktion der Verbindungslatenz für zeitlich variierende Mengen an Verbindungen großer Setups beabsichtigt. Wie in Abschnitt 9.1 beschrieben, führte dieser Implementationsaspekt jedoch zu Einschränkungen in der Umsetzung von Anfragestrategien zur Intraoperatorparallelisierung in Postgres-XL.

4.4 Machine Learning und wissenschaftliches Rechnen in Datenbanksystemen

In diesem Abschnitt wird der Stand der Technik und Forschung bezüglich Projekten und Systemen, die wissenschaftliches Rechnen, Machine Learning (als Methodenkern der Aktivitätserkennung und -vorhersage) und Datenbanktechnologie kombinieren, erörtert.
Ein Überblick über das Forschungs- und Entwicklungsfeld wurde im Jahr 2017 von *Günemann* in [75] gegeben. Der Großteil der dort beschriebenen Projekte ist bis zum heutigen Zeitpunkt (Stand 12/2022) relevant oder Basis aufbauender Entwicklungen. In den Ausführungen wurde nicht nur auf den hier untersuchten Push-Down der Berechnung von Machine-Learning-Verfahren oder allgemeineren Methoden des wissenschaftlichen Rechnens in Datenbanksystemen eingegangen, sondern ebenfalls das Potenzial für andere Anwendungsgebiete beleuchtet. So wurden etwa Machine-Learning-Ansätze zur Selbstoptimierung von Datenbanksystemen getestet. Diese beinhalten etwa das automatische Erstellen effizienter Indexstrukturen oder Anfragepläne (vgl. Abschnitt 7.4). Andere Projekte forschen an der Vorhersage sinnvoller Konfigurationen von Database-as-a-Service-Applikationen im Cloud-Computing-Kontext. Hierbei werden etwa die Ressourcen an den vorhergesagten Workload angepasst, um entstehende Kosten und Energienutzung zu minimieren. Detaillierte Betrachtungen werden hier anwendungsbedingt ausgespart. Es sei erneut hierfür auf [75] verwiesen.
Im Folgenden liegt der Fokus auf hier fokussierten Verarbeitung von Machine-Learning-Verfahren und Methoden des wissenschaftlichen Rechnens innerhalb relationaler Datenbanksysteme. Zur besseren Abgrenzung der hier vorgestellten Ergebnisse werden zunächst Projekte, die ebenfalls Implementationsansätze mittels Standard-SQL im ähnlichen Kontext verfolgt haben, vorgestellt. Wie sich zeigen wird, umfasst dies vergleichsweise wenige systematische Ansätze, sodass die im Verlauf der Arbeit vorgenommenen Untersuchungen sich vom aktuellen Forschungsstand weitgehend klar abgrenzen.
Nach der Vorstellung der SQL-Projekte werden Untersuchungen und etablierte Systeme vorgestellt, die auf verschiedene Art und Weisen Berechnungsfunktionalitäten innerhalb der jeweiligen relationalen Datenbanksysteme ermöglichen. Solche datenbankspezifischen Anpassungen sind im Allgemeinen performanter, da sie optimiert an die Architektur des jeweiligen Datenbanksystems

angepasst werden können. Nachteilig hieran ist die spezifische Systemgebundenheit, welches etwa die Langlebigkeit von Nutzerimplementationen aufgrund fehlender Standardisierung abhängig von der Unterstützung der jeweiligen Datenbankhersteller macht.
Der zugehörige Überblick ist daher insbesondere wichtig, um nötige Trade-Offs zwischen Standardisierung und Performance abwägen zu können, insofern reine SQL-Implementationen zu ineffizient sind (vgl. beispielsweise die Diskussion zur Hauptkomponentenanalyse in SQL in Anhang B.6). Eine detailliertere Analyse solcher Trade-Offs wird bei der Diskussion möglicher Framework-Architekturen in Abschnitt 6.1 geführt.

4.4.1 Standard-SQL-Implementationen

Die systematische Untersuchung von Standard-SQL-Umsetzungen im gegebenen Kontext ist neben dem hier vorgestellten Forschungsprojekt vergleichsweise selten vorgenommen worden.

Ein nennenswertes Projekt wurde in [88] vorgestellt. In diesem wurde die Berechnung von Hauptkomponentenanalysen mittels SQL-Anfragen und User-Defined-Functions (UDFs) in Microsofts SQL Server 2008 mit hybriden Implementationen in R und Java verglichen. Hierbei wurde insbesondere ein Tridiagonilisierungsverfahren und das QR-Verfahren zur Berechnung von Eigenwerten umgesetzt. Die Laufzeitergebnisse der SQL-Implementation stellten sich im Vergleich als schwach heraus. Gründe hierfür sind, wegen der nur eingeschränkten Beschreibung der Implementation, schwierig zu benennen. Die Wahl der Relationenschemata (teilweise ähnlich dem Spaltenattributsschema aus Abschnitt 6.4), vergleichsweise niedrige Problemdimensionen, fehlende Informationen zur Nutzung von Indexstrukturen und der Systemkonfiguration, die Wahl eines zeilenorientierten Datenbanksystems („row store“, vgl. Abschnitt 6.2), sowie die hohe Anzahl aneinandergereihter, nicht geschachtelter leichtgewichtiger Einfügeanweisungen (vgl. Abschnitt 7.3) sind mögliche Problempunkte. Im Anhang B.6 ist eine vollständige SQL-Umsetzung der Hauptkomponentenanalyse, die im Zuge dieser Arbeit erstellt wurde, mit zugehöriger Laufzeit-Diskussion präsentiert. Hierbei wurde jedoch eine schneller konvergierende Variante des QR-Verfahrens mittels Shift-Strategien, sowie teilweise geschachtelte QR-Zerlegungen mittels Givens-Rotationen implementiert. Trotz des vermeintlichen Optimierungspotenzials zeigt sich dort ebenfalls, dass der hoch-iterative Charakter der Hauptkomponentenanalyse (im Falle vollständiger Eigenwertanalysen in dichtbesetzten Szenarien) als Beispiel für eine tendenziell ungünstige Methode für SQL-Implementationen beschrieben.

In **RIOT-DB** (R with I/O Transparency) [89] wurde eine modulare Erweiterung der statistischen Software R („R-Package“) entwickelt, die Klassen und einfache Operatoren derart überladen, dass diese transparent eine Berechnung zugehöriger SQL-Anfragen auf einem Datenbanksystem (in diesem Fall MySQL) umsetzt. Hierfür wurden die Variablennamen in R auf

zugehörige Relationen abgebildet und Anweisungen einer eingeschränkten Menge von Operationen zunächst in nicht-materialisierte Sichten überführt. Durch die abschließende Materialisierung konnten in diesem Fall das Schreiben von Zwischenergebnissen eingespart werden. Die Forschungsgruppe berichtete über Einschränkungen bezüglich der Erweiterung von Matrixoperationen (wie etwa Multiplikation) aufgrund von ungeeigneten Anfragepläne und deren Laufzeiten. Eine detailliertere Diskussion zur Verarbeitung solcher Operationen in SQL-Systemen wird hier in Abschnitt 7.2 vorgenommen. Die Gruppe beschrieb ebenfalls, dass die Wahl des Datenbanksystems für den gegebenen Kontext möglicherweise ungeeignet ist.

Die möglichst transparente Einbettung von Anfragen in Programmiersprachen ist hierbei ein bereits verbreitetes Feld. Ohne einen erschöpfenden Überblick zu intendieren, sei etwa auf das im .NET-Framework integrierte **LINQ** [90] verwiesen, welches aus einer eigenen Anfragesyntax transparent Anfragen für SQL-Systeme (oder auch andere Datenbanktypen, etwa XML) ableitet und an das jeweilige System kommuniziert. Nennenswert in diesem Zusammenhang ist ebenfalls das Projekt **Database Supported Haskell** [91], indem List-Comprehensions in Haskell transparent (mit vergleichsweise geringer Anpassung der Syntax) in SQL:1999-Anfragen überführt und vom Datenbanksystem (hier MonetDB, vgl. folgender Unterabschnitt) verarbeitet werden. Letztere Projekte konzentrieren sich eher auf die transparente Kooperation etablierter Programmiersprachen und relationaler Datenbanksysteme, anstatt auf systematische Untersuchungen zu SQL-basierten wissenschaftlichen Rechnungen. Die transparente Einbettung ist aber insbesondere interessant für den Kontext der Assistenzsystementwicklung (oder anderen Entwicklungsszenarien), da im Allgemeinen nicht davon ausgegangen werden kann, dass Entwickler Experten in SQL bzw. Datenbankentwicklung und -verwaltung sind (vgl. die Architekturdiskussion aus Abschnitt 6.1).
Konträr hierzu wurde im Open-Source-Projekt **SQLFlow** [92] eine Erweiterung der SQL-Syntax durch Machine-Learning-Klauseln, wie beispielweise `to train` oder `to predict`, vorgestellt. Mit dieser Erweiterung werden aus vergleichsweise kompakten Anfragen ganze Workflows (in Python) erstellt, die eine transparente Berechnung von Machine-Learning-Modellen (inklusive Data-Cleaning und Feature Extraction) auf einem variablen Software-Stack ausgewählter Datenbanksysteme, Machine-Learning-Bibliotheken, User-Interfaces und weiteren umsetzen.
Im Folgenden werden Projekte und Systeme vorgestellt, die systemspezifische Push-Down-Möglichkeiten für wissenschaftliches Rechnen nutzen. Einige setzen hierbei auf die Etablierung von Funktionalität durch das Einfügen von User Defined Functions (UDFs), die in SQL-Anfragen aufgerufen werden können. Ein standardisierter (aber nicht zwangsläufig weitverbreiteter) Weg hierfür ist die Umsetzung dieser in der Skriptsprache PL/SQL, welche ein prozedurale Erweiterung von SQL darstellt und insbesondere eine Kombination von SQL-Anfragen mit imperativen Sprachkonstrukten, wie beispielsweise Schleifen, realisieren kann (vgl. etwa [34]). In [93] wird

beschrieben, dass solche UDFs oftmals problematische Laufzeiten mit sich führen. Dies liegt etwa an dem einschränkenden internen (teilweise multiplen) Wechsel zwischen der mengenorientierten (optimierten) Anfrageplanverarbeitung der äußeren SQL-Anfrage, dem „Statement-by-Statement PL/SQL interpretation mode“ und dem daraus resultierenden Overhead. Aus diesem Grund wird von der Gruppe ein Compiler zur Überführung von PL/SQL-Funktionen in reine SQL-Statements beschrieben, welcher insbesondere auch iterative Verfahren in rekursive Anfragen überführen kann. Dies ist für das vorliegende Projekt von Bedeutung, da speziell hoch-iterative Verfahren, ohne die Nutzung von rekursiven Anfragen, problematisch in der Anfrageplanformulierung und -auswertung sein können (vgl. etwa Abschnitt 7.3 zur Kompositon von SQL-Anfragen, die rekursive Verarbeitung der Fourier-Transformation in 9.3 oder die bereits erwähnte problematische Performance dicht besetzter Hauptkomponentenanalysen in SQL im Anhang B.6).

4.4.2 Spezifische datenbankinterne Erweiterungen

In diesem Abschnitt werden Systeme und Projekte vorgestellt, die auf spezifische Art und Weise zusätzliche Funktionalitäten des wissenschaftlichen Rechnens oder Machine Learning in das jeweilige Datenbanksystem eingeführt haben. Hierbei unterscheiden sich die Ansätze von der kompletten Neuimplementation von Special-Purpose-Datenbanksystemen, zu modularen Erweiterungen durch UDFs oder internen Reimplementation/Integration einzelner Aspekte, wie etwa Datentypen, Operatoren der Relationenalgebra und zugehöriger logischer Optimierungsregeln in der Anfrageplanerstellung.
Aufgrund des vorherrschenden Forschungs- und Entwicklungsfokus auf Big Data, künstlicher Intelligenz und smarten Umgebungen ist dieses noch vergleichsweise junge Feld weit und schnelllebig. Ferner sind kommerzielle Datenbanksystemhersteller (etwa **Oracle**, **Google BigQuery** oder **Teradata**) mitunter bemüht, Details zu einzelnen Funktionalitäten im Sinne des Wettbewerbs nicht oder nur teilweise zu publizieren. Aus diesen Gründen ist eine erschöpfende Darstellung aller Ansätze im Allgemeinen schwierig. Es folgt eine Übersicht viel zitierter und einflussreicher Forschungsprojekte, Datenbanksysteme und Erweiterungen.

MonetDB ist ein relationales Datenbanksystem mit dem Fokus auf eine effiziente hauptspeicherinterne Anfrageverarbeitung. Das System gilt als eines der Vorreiter spaltenorientierter Datenbankarchitekturen (vgl. Abschnitt 6.2) und war Ausgangspunkt für Forschungen zur Entwicklung der Vektor-Anfrage-Engine X100, die SIMD-Möglichkeiten moderner CPUs ausnutzt [94]. Diese Engine ist Basis des kommerziellen System *Actian Vector*, welches wie MonetDB mehrfach im Lauf dieser Arbeit zur experimentellen Auswertung genutzt wurde. MonetDB erlaubt neben den Möglichkeiten, mittels UDFs in C++ oder Python zusätzliche Funktionalitäten des wissenschaftlichen Rechnens einzubinden, eine kommunikationsfreie Kombination mit der stati-

schen Software *R*. In [95] wird hierfür beschrieben, wie MonetDB in den zugehörigen R-Prozess eingebettet wird und durch gezielte Adressierung mittels Zeigern beide Instanzen auf demselben Adressraum arbeiten können. Dadurch wird es ermöglicht, R-Funktionen effizient in der SQL-Schnittstelle aufzurufen. Dieses Feature wurde beispielsweise in Abschnitt 6.2 im Kontext einfacher linearer Regression experimentell getestet.

Eine weitere Integration von R mit geteiltem Adressraum setzt das **RICE**-Projekt in SAP HANA [96] um. In diesem werden von R angeforderte Daten aus der Datenbank nur einmalig in den geteilten Adressraum so kopiert, dass sie von R ohne kopieren in ein *dataframe*, der nativen Datenstruktur für Tabellen in R, überführt werden können. Mithilfe dieses Ansatzes wurde eine Integration von R-Skripten als Teil des Anfrageplans realisiert, welches parallele Verarbeitung mittels multipler R-Prozesse ermöglicht.

Eine Integration etablierter Bibliotheken wurde ebenfalls in [97] vorgestellt. In diesem wurde eine Hauptkomponentenanalyse durch eine Kombination von UDFs und einer Integration einer verteilten LAPACK/BLAS-Implementation (Intels Math Kernel Library) in Microsoft SQL Server 2008 propagiert. Experimentelle Ergebnisse konnten Performance-Vorteile bezüglich der, im Zuge des gleichen Projektes vorgestellten, reinen SQL-Lösung aus [88] (siehe vorigen Abschnitt) und klassischen Server-Client-Anbindungen in R aufzeigen.
Konzeptionell ähnlich zu diesem Ansatz ist **Apache MADlib** [98], welches ein Apache-Projekt[2] ist, in dem verschiedene Machine-Learning-Methoden und einfache Lineare-Algebra-Operatoren dem Datenbankentwickler (PostgreSQL und Greenplum Database) zur Verfügung gestellt werden. Hierfür wurden zahlreiche Methoden mittels *User Defined Aggregates* (UDAs) und *User Defined Functions* (UDFs) in C++ und Python umgesetzt und in die Datenbanksysteme integriert. Als Backend der meisten Methoden wurden insbesondere auch die effizienten BLAS- und LAPACK-Implementationen (vgl. Abschnitt 4.2) genutzt. Ähnlich hierzu nutzen etwa **Oracle Data Miner** oder **SAP HANAs Predictive Analysis Library** UDFs und UDAs um Machine-Learning-Funktionalitäten zu erreichen [75].
Im Folgenden werden Systeme beschrieben, die den klassischen relationalen Kern gezielt für wissenschaftliches Rechnen und Machine Learning erweitern.

HyPer-DB ist ein Hauptspeicherdatenbanksystem, welches Machine-Learning-Verfahren auf zwei wesentliche Arten unterstützt [75,99]. Zum einen wurde die SQL-Schnittstelle um ein Schleifenkonstrukt erweitert, welches für oftmals hoch-iterative ML-Verfahren entscheidend ist. Zum anderen wurden Analyseoperatoren in deren Relationenalgebra eingeführt, die über spezielle λ-Ausdrücke parametrisiert werden können. Basierend auf diesen Konstrukten wurde in [100]

[2] `https://madlib.apache.org` (Stand: 12/2022)

eine Erweiterung der HyPer-Operatoren für die Berechnung des weit eingesetzten Gradient-Descent-Minimierungsverfahren und allgemeiner Tensor-Algebra vorgestellt. Zusätzlich wurde in [101] ein Compiler vorgestellt, der eine eigens entwickelte deklarative Sprache für kompakte Machine-Learning-Berechnungen in Python/NumPy oder in SQL/UDFs für Postgres oder HyPer überführt.

SciDB [102] ist ein paralleles Datenbanksystem mit einem Array-Datenmodell und zugrundeliegender Shared-Nothing-Architektur, welches mit dem ausdrücklichen Ziel zur Speicherung und Analyse wissenschaftlicher Daten entwickelt wurde. Das System wird über die speziell entwickelte SQL-Erweiterung AQL (Array Query Language) angesprochen und bietet vorimplementierte mathematische Datentypen und Operationen [103], sowie eine Integration der Sprache R und Python an.
Ein weiteres paralleles relationales System ist **SimSQL**, welches auf Hadoop aufsetzt. In diesem werden SQL-Anfragen nach Durchlauf einer klassischen relationalen Optimierungsschnittstelle in Java-Code zur Verarbeitung auf Hadoop überführt und umgesetzt. Im Zuge dieses Projektes wurde eine interne Erweiterung der SQL-Schnittstelle für dicht besetzte Vektor- und Matrix-Datentypen, sowie die Einführung zugehöriger Anfragepläne für parallele Lineare-Algebra-Operationen, wie die Block-Matrix-Multiplikation [72] (vgl. Ausführungen in Abschnitt 8.2), propagiert und analysiert. Die Gruppe evaluiert in Experimenten, dass mit vergleichsweise wenigen internen Anpassungen des Systemkerns relationale Systeme Laufzeitergebnisse erreichen können, die vergleichbar mit modernen Special-Purpose-Systemen, wie SciDB oder SystemML, sind.

Kapitel 5

Hidden-Markov-Modelle

Um mögliche Ansätze für die Untersuchung SQL-basierter Machine-Learning-Verfahren in relationalen Datenbanken detailliert diskutieren zu können, werden als Ausgangspunkt Hidden-Markov-Modelle (HMM) betrachtet. Wie in Abschnitt 2.2 beschrieben, finden diese häufig Anwendung bei Computer-Mensch-Interaktionen (wie etwa Assistenzsystemen). Gründe hierfür sind etwa, ihre Fähigkeit temporale Beziehungen gut modellieren zu können, die Trennung von Beobachtungen (Sensoren) und interner nicht-beobachtbarer Systemzustände, sowie eine weitgehende Etablierung fundamentaler Problemstellungen und zugehöriger Lösungsverfahren. Diese drei Punkte werden in der folgenden Einführung näher erläutert.

Als Referenz für die in dieser Arbeit betrachteten grundlegenden Probleme von HMMs wird hierbei maßgeblich das einflussreiche und viel zitierte Tutorial von *Rabiner* [104][1] genutzt. Die Lösungsverfahren der Basisprobleme werden für die Überführung in SQL-basierte Implementationen auf deren struktureller Zusammensetzung bezüglich mathematischer Operatoren untersucht. Der Übersetzungsprozess in SQL wird daraufhin in den Kapiteln 6, 7 und 8 näher diskutiert. Hierbei wird sich zeigen, dass die Lösungsverfahren durch eine überschaubare Menge einfacher Operationen der linearen Algebra und klassischer Datenbankoperationen, wie Selektionen und Aggregationen, beschrieben werden können. Aus diesem Grund wird im abschließenden Abschnitt 4.2 Bezug zu den weitverbreiteten *Basic Linear Algebra Subprograms* (BLAS) [66] genommen, welche eine Obermenge der genutzten Lineare-Algebra-Operationen enthalten und Grundlage vieler essentieller Bibliotheken und Software-Umgebungen des wissenschaftlichen Rechnens ist. Durch diesen Bezug und der damit bedingten Erweiterung der zu betrachtenden Operatormenge wird insbesondere die Menge möglicher Anwendungsszenarien für das datenbankbasierte Konzept aus Kapitel 3 deutlich erweitert.

[1] Der Artikel wurde laut der digitalen Bibliothek von IEEE (`https://ieeexplore.ieee.org/document/18626`) über 12000 mal zitiert. Stand: 12/2022

5.1 Theorie

In diesem Abschnitt werden zunächst theoretische Grundlagen über HMMs und zugehörige Notationen etabliert. Die Beschreibungen beziehen sich hierbei, wenn nicht anders gekennzeichnet, auf *Rabiners* Tutorial [104] und Ausarbeitungen in *Russel* und *Norvigs* Einführung in Künstliche Intelligenz [105].
Hidden-Markov-Modelle sind Vertreter temporaler probabilistischer Modelle und eine Spezialform von dynamischen Bayeschen Netzen (vgl. etwa [105]). Sie ermöglichen es, in deren Abstraktion der Welt oder des Systems, Aussagen über Vergangenheit, Gegenwart und Zukunft zu treffen. Dies geschieht hierbei unter Nutzung von Übergangs- und Beobachtungsmodellen (oder auch: Sensormodellen). Übergangsmodelle beschreiben die Zustandsübergangsprozesse eines zugrundeliegenden, nicht direkt beobachtbaren diskreten Systems, beziehungsweise einer diskreten Abstraktion der Welt. Mithilfe von Beobachtungen (etwa durch Sensoren) und des Beobachtungsmodells kann versucht werden, die Wahrscheinlichkeiten aktueller Zustände (auch: Belief States) des Systems zu aktualisieren. In HMMs ist der Zustandsraum des Systems diskret. Der Zustandsübergangsprozess wird hierbei durch eine (diskrete) *Markov-Kette* modelliert.

5.1.1 Markov-Ketten

Diskrete Markov-Ketten (auch: Markov-Prozess) sind spezielle stochastische Prozesse, dessen Wahrscheinlichkeitsverteilung im aktuellen Zeitschritt nur von einer begrenzten Anzahl vergangener Zustände abhängt. Für eine formalere Beschreibung sei $N \in \mathbb{N}$ die Anzahl möglicher Zustände im diskreten Systemzustandsraum $S = \{S_1, S_2, \ldots, S_N\}$ und $Q = \{q_0, q_1, \ldots, q_T\}$ eine Familie von Zufallsvariablen mit $T \in \mathbb{N}^{\geq 2}$. Hierbei beschreibt T etwa einen maximalen Zeitwert für diskrete Zeitstempel $t = 0, 1, \ldots, T$. Die Familie Q ist ein diskreter (endlicher) Markov-Prozess n-ter Ordnung ($\mathbb{N} \ni n < T$), wenn

$$\begin{aligned} &P(q_{t+1} = S_{j_{t+1}} \mid q_t = S_{j_t}, q_{t-1} = S_{j_{t-1}}, \ldots, q_0 = S_{j_1}\} \qquad (5.1) \\ =&P(q_{t+1} \mid S_{j_{t+1}} \mid q_t = S_{j_t}, q_{t-1} = S_{j_{t-1}}, \ldots, , q_{t-n} = S_{j_{t-n}}) \end{aligned}$$

gilt, wobei (5.1) als Markov-Bedingung (auch: Markov-Annahme) bezeichnet wird. Prinzipiell können die Wahrscheinlichkeitsverteilungen $P(q_{t+1} \mid q_t)$ in jedem Zeitschritt t variieren. Im Verlauf dieser Arbeit werden jedoch ausschließlich zeitinvariante Prozesse betrachtet. Dies ist eine verbreitete Annahme und lässt sich etwa durch die Zeitinvarianz von Naturgesetzen oder anderer Gesetzmäßigkeiten motivieren.
Die wohl verbreitetste Form ist hierbei die Markov-Kette erster Ordnung, die auch im Laufe der Arbeit ausschließlich betrachtet wird. Dies ist keine starke Restriktion, da es möglich ist Ketten höherer Ordnung durch das gezielte Einführen neuer Zustandsvariablen in Markov-Ketten erster

Ordnung umzuformulieren [105].

Die in der Kette auftretenden Übergangswahrscheinlichkeiten $P(q_{t+1} = S_{j_{t+1}} \mid q_t = S_{j_t})$ werden klassischerweise in einer Matrix $(a_{ij})_{ij} = A \in [0,1]^{N\times N}$, der Zustandsübergangsmatrix, mit

$$a_{ij} = P(q_{t+1} = S_j \mid q_t = S_i)$$

gespeichert. Wie sich zeigen wird, erlaubt dies insbesondere eine kompakte Implementation grundlegender Lösungsverfahren durch einfache Operationen der linearen Algebra. Mit dieser Notation lässt sich eine Markov-Kette durch das Tupel (S, A, π) beschreiben, wobei $\pi = (q_0 = S_i)_i^T \in [0,1]^N$ eine Anfangsverteilung beschreibt. Zur Veranschaulichung sei zunächst ein Wettermodell, eines der wohl verbreitetsten Einführungsbeispiele von Markov-Prozessen (vgl. etwa [104] oder [106] für ähnliche Ausführungen), vorgestellt.

Beispiel 2 (Wettermodell (Markov-Kette))**.** Es soll ein einfaches Modell erstellt werden, mit dem Voraussagen bezüglich der drei Systemzustände S_1=sonnig, S_2=bewölkt, S_3=regnerisch und dem Zustandsraum $S = \{S_1, S_2, S_3\}$ gemacht werden können. Als Zustandsübergangswahrscheinlichkeiten seien folgende Werte beobachtet (beziehungsweise gesetzt) worden

$$\begin{array}{lll} P(q_t = S_1 \mid q_{t-1} = S_1) = 0.6 & P(q_t = S_2 \mid q_{t-1} = S_1) = 0.3 & P(q_t = S_3 \mid q_{t-1} = S_1) = 0.1 \\ P(q_t = S_1 \mid q_{t-1} = S_2) = 0.3 & P(q_t = S_2 \mid q_{t-1} = S_2) = 0.5 & P(q_t = S_3 \mid q_{t-1} = S_2) = 0.2 \\ P(q_t = S_1 \mid q_{t-1} = S_3) = 0.2 & P(q_t = S_2 \mid q_{t-1} = S_3) = 0.4 & P(q_t = S_3 \mid q_{t-1} = S_3) = 0.4, \end{array}$$

welche kompakt in der Zustandsübergangsmatrix

$$A = \begin{pmatrix} 0.6 & 0.3 & 0.1 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.4 & 0.4 \end{pmatrix}$$

notiert werden können. Eine Visualisierung dieses Prozesses mittels Graphen ist in Abbildung 5.1 dargestellt.

Mit dem definierten Zustandsraum und den Wahrscheinlichkeiten können, mithilfe einer initialen Verteilung π, Wettervorhersagen getroffen werden. Es sei etwa „heute“ ($t = 0$) sonnig (d.h. $\pi = (1,0,0)$) und es soll eine Vorhersage für Übermorgen ($t = 2$) getroffen werden. Da jede Spalte $j = 1, 2, \ldots, N$ von A genau die Zustandsübergangswahrscheinlichkeiten $P(q_t = S_j | q_{t-1} = S_i)$ mit $i = 1, 2, \ldots, N$ enthält, entspricht der $j-$te Eintrag des Matrix-Vektor-Produktes $A^T\pi$ genau der aufsummierten Wahrscheinlichkeit, von einer der möglichen Ausgangszustände in den

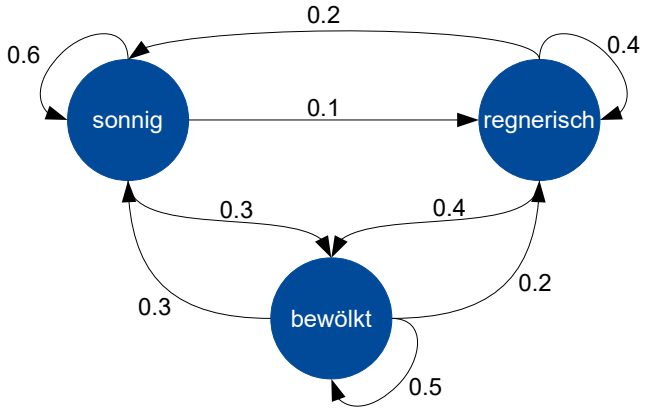


Abbildung 5.1: Graphische Darstellung des Wettermodells aus Beispiel 2 als gerichteter, zylklischer Graph.

Zustand j zu geraten. Daher kann die gewünschte Vorhersage durch

$$\begin{aligned} P(q_3 \mid q_1 = S_1) &= P(q_3 \mid q_2) \cdot P(q_2 \mid q_1 = S_1) \\ &= A^T A^T \pi \\ &= \begin{pmatrix} 0.47 & 0.37 & 0.32 \\ 0.37 & 0.42 & 0.42 \\ 0.16 & 0.21 & 0.26 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0.47 & 0.37 & 0.16 \end{pmatrix}. \end{aligned}$$

errechnet werden und mit 47-prozentiger Wahrscheinlichkeit sonniges, mit 37-prozentiger Wahrscheinlichkeit bewölktes und mit 16-prozentiger Wahrscheinlichkeit regnerisches Wetter vorausgesagt werden.

Das Wetterbeispiel dient als einfaches Beispiel zur Erklärung der Struktur und Funktionsweise von Markov-Ketten, ist jedoch nicht als realitätsnahes Beispiel zu verstehen. Konträr hierzu wird abschließend einer der wohl bekanntesten Big-Data-Anwendungsfälle von Markov-Ketten beispielhaft eingeführt: Das PageRank-Verfahren.

Beispiel 3 (PageRank und MapReduce)**.** Die Ursprünge des PageRank-Verfahrens wurden 1998 von den Google-Gründern *Page* und *Brin* gelegt [2]. Zu dieser Zeit war, bedingt durch das exponentielle Wachstum der Anzahl der Internetseiten, das Suchen nach passenden Webseiten mittels Webkatalogen an seine Grenzen gestoßen [106]. *Page* und *Brin* entwickelten hierfür ein Verfahren, welches „die Relevanz“ der Internetseiten abschätzt. In [2] beschreiben die Autoren,

dass das Nutzerverhalten als stochastischer Prozess interpretiert wird. Die Annahme hierbei ist, dass ein Nutzer, der sich auf einer zufälligen Seite befindet, auf Weblinks klickt, niemals zu einer alten Seiten zurückkehrt und nach einiger Zeit „in Langeweile verfällt“ und eine neue zufällige Seite aufruft[2]. Aus dieser Prämisse lässt sich eine Markov-Kette erster Ordnung entwickeln, mit der Menge registrierte Webseiten als Zustandsraum und deren Links als Basis für Übergangswahrscheinlichkeiten von Seite zu Seite. Die propagierte Übergangsmatrix (auch: Google-Matrix)

$$G = dS + (1-d)\frac{1}{N}\mathbf{1}\cdot\mathbf{1}^T \in [0,1]^{N\times N}$$

besteht aus zwei wesentlichen Summanden: Der erste Term $S = H + \frac{1}{N}\mathbf{a}\mathbf{1}^T$ lässt sich als regularisierte Hyperlink-Matrix beschreiben. Hierbei beinhalten die Einträge h_{ij} der Hyperlink-Matrix die Wahrscheinlichkeiten auf der $i-$ten Webseite einen Link auf die $j-$te zu nutzen. Diese Wahrscheinlichkeiten werden hierbei als gleichverteilt angenommen. Bezeichnet demnach c_i die Menge aller Links der $i-$ten Webseite, dann ist

$$h_{ij} = \begin{cases} \frac{1}{c_i} & \text{falls } i\text{-te Webseite ein Link auf } j\text{-te Webseite besitzt} \\ 0 & \text{sonst } 0 \end{cases}$$

Aufgrund der begrenzten Anzahl von Links pro Webseite ist die Matrix H offenbar im Allgemeinen sehr dünn besetzt mit einem Durchschnitt zwischen 10 und 15 Einträgen pro Zeile und einer geschätzten Webseitenanzahl im Milliardenbereich [68]. Besitzt eine Seite keine ausgehenden Links, wird angenommen, dass der Nutzer eine beliebige Webseite als nächstes ansteuert. Die Übergangswahrscheinlichkeiten sind hierbei ebenfalls gleichverteilt. Dies wird über den Term $\frac{1}{N}\mathbf{a}\mathbf{1}^T$ mit

$$a_i = \begin{cases} 1 & \text{falls } i\text{-te Webseite keine ausgehenden Links besitzt} \\ 0 & \text{sonst } 0. \end{cases}$$

und dem N-elementigen 1-Vektor $\mathbf{1}$ realisiert. Der zweite wesentliche Summand $\frac{1}{N}\mathbf{1}\cdot\mathbf{1}^T$ modelliert hierbei, dass der Nutzer aus „Langeweile“ eine beliebige neue Seite aufruft. Die beiden Summanden werden in Form eines Polygonzugs über den Dämpfungsparameter $d \in [0,1]$ verbunden. Im Falle von Googles ursprünglichem PageRank ist $d \approx 0.85$ [106], welches insbesondere eine Dominanz des Hyperlink-Terms bedeutet. Die resultierende Google-Matrix G erfüllt offenbar die Eigenschaften einer Zustandsübergangsmatrix einer Markov-Kette. Sie ist insbesondere so konstruiert, dass die stationäre Lösung $\mathbf{x}$ (der Vektor, dessen Einträge die PageRanks der

[2]Originalzitat aus [2]: „PageRank can be thought of as a model of user behavior. We assume there is a “random surfer” who is given a web page at random and keeps clicking on links, never hitting “back” but eventually gets bored and starts on another random page. The probability that the random surfer visits a page is its PageRank“

Webseiten enthält)

$$G\mathbf{x} = \mathbf{x} \tag{5.2}$$

existiert und eindeutig ist (vgl. [106] für eine detailliertere Ausführung des Verfahrens).
Die Berechnung der stationären Verteilung wird klassischerweise als Eigenwertproblem zum Eigenwert 1 interpretiert und über die Potenzmethode (vgl. etwa [107]) gelöst. Dieses entspricht in seiner einfachsten Form einer wiederholten Matrix-Vektor-Multiplikation gemäß

$$\mathbf{x}^{(i+1)} = G^T\mathbf{x}^{(i)}, \tag{5.3}$$

wobei typischerweise das Ergebnis nach jedem Schritt im Sinne der numerischen Stabilität normiert wird. Das Verfahren ist insbesondere wegen dessen Einfachheit und der vergleichsweise guten Konvergenzgeschwindigkeit, aufgrund einer garantierten guten Isolation des größten Eigenwert 1 vom Rest der Werte [108], geeignet. Insbesondere interessant in (5.3) ist die Struktur der Google-Matrix. Wird letztere in seine oben beschriebenen Summanden aufgeteilt und ist $\mathbf{x}^{(i)}$ normiert (d.h. $\sum x_k^{(i)} = 1$), gilt

$$\begin{aligned}\mathbf{x}^{(i+1)} &= G^T\mathbf{x}^{(i)} \\ &= dH^T\mathbf{x}^{(i)} + \frac{1}{N}\mathbf{1}(d\mathbf{a}^T\mathbf{x}^{(i)} + (1-d)),\end{aligned}$$

welches den Kern der Iterationen im wesentlich auf die enorm große dünn besetzte Matrix-Vektor-Multiplikation $H^T\mathbf{x}^{(i)}$ beschränkt. Die Notwendigkeit der massiven, skalierbaren Parallelisierung dieses Problems bildet den Ursprung für Googles Entwicklung von MapReduce [68,69], welches etwa in Abschnitt 4.3.1 beschrieben wird und in Kapitel 9 ausgewertet und verglichen wird mit parallelen Datenbanksystemen. Das Verfahren konvergiert in double precision typischerweise etwa nach 50 Iterationen [68]. Die spezielle Unterscheidung von dünn besetzten Problemen im Big-Data-Bereich wird auch bei der Übersetzung in SQL und der effizienten Parallelisierung in Kapitel 7 und 8 besondere Beachtung finden.

5.1.2 Hidden-Markov-Modelle

In den bisher betrachteten Markov-Ketten kann der aktuelle Zustand beziehungsweise die Zustandsverteilungen im System direkt nachvollzogen werden. Diese Prämisse trifft in vielen Anwendungsfällen jedoch nicht zu. Vielmehr wird beispielsweise in Assistenzsystemen versucht, über Sensordaten beziehungsweise daraus abgeleiteter Beobachtungen auf die versteckten („hidden“) Systemzustände zu schließen.
HMMs modellieren diesen Umstand durch das Hinzufügen eines Beobachtungsmodells (auch: Sensormodells). Die Markov-Kette $\mathbf{q} = (q_t)_{t=1,\dots,T}$ der Systemzustände wird durch einen weiteren

stochastischen Prozess $\mathbf{o} = (o_t)_{t=1,\ldots,T}$ erweitert, dessen Zufallsvariablen Werte aus dem Raum der beobachtbaren Symbole (auch: Emissionen) $V = (V_1, \ldots, V_M)$ annehmen. Die Symbole o_t könnten prinzipiell von dem aktuellen und allen vorherigen Zuständen sowie von vorangegangen Beobachtungen abhängen. Ähnlich zu der Markov-Annahme wird in diesem Fall jedoch davon ausgegangen, dass der aktuelle Systemzustand q_t entscheidenden Einfluss auf die Beobachtung o_t besitzt und demnach

$$P(o_t \mid o_{0:t-1}, q_{0:t}) = P(o_t \mid q_t) \tag{5.4}$$

angenommen. Hierbei bezeichnet $0 : t$ die Folge von $0, 1, 2, \ldots, t$ und demnach $o_{0:t}$ die Folge der Beobachtungen $o_0, o_1, \ldots, o_t$. Die Prämisse (5.4) wird als *Sensor-Markov-Annahme* [105] bezeichnet und ist in Abbildung 5.3 als Abhängigkeitsgraph visualisiert. In der Grafik ist hierbei erkennbar, dass die beobachtbaren Symbole o_t durch den Welt- bzw. Systemzustand q_t bestimmt werden. Der eigentliche Inferenzprozess geschieht jedoch in entgegengesetzter Richtung. Analog zur Zustandsübergangsmatrix der Markov-Kette können die Wahrscheinlichkeiten aus (5.4) in einer Beobachtungsmatrix $B = (b_{ij})_{ij} \in [0, 1]^{N \times M}$ mit

$$b_{ij} = P(o_t = V_j \mid q_t = S_i) \tag{5.5}$$

gespeichert werden können.
Mit den etablierten Bezeichnungen kann schließlich ein (zeitinvariantes, diskretes) Hidden-Markov-Modell durch ein Tupel

$$\lambda = (S, V, A, B, \pi)$$

beschrieben werden.
Zur Veranschaulichung wird im Folgenden das Wetter-Beispiel der Markov-Ketten aus Beispiel 2 in ein HMM erweitert.

Beispiel 4 (Erweiterung Wettermodell (Hidden-Markov-Modell))**.** Es soll das Wettermodell aus Beispiel 2 mit Zustandsraum S und Zustandsübergangsmatrix A erweitert werden. Für automatische Wettervorhersagen, speziell an mehreren entfernten Orten, ist eine Beobachtung des Wetters unzulänglich. Hier wird angenommen, dass durch Barometer gemessene Luftdruckdaten zur Verfügung stehen, aus denen auf das Wetter geschlossen werden soll. Aus Gründen der Übersichtlichkeit seien die gemessenen Luftdruckdaten auf die beiden Symbole $V = \{V_1 =$ `Luftdruck hoch`$, V_2 =$ `Luftdruck niedrig` $\}$ abgebildet. Dabei seien die folgenden Zusammenhänge zwischen Luftdruck und Wetter beobachtet worden

$$\begin{aligned}
&P(o_t = V_1 \mid q_t = S_1) = 0.9 && P(o_t = V_2 \mid q_t = S_1) = 0.1\\
&P(o_t = V_1 \mid q_t = S_2) = 0.3 && P(o_t = V_2 \mid q_t = S_2) = 0.7\\
&P(o_t = V_1 \mid q_t = S_3) = 0.1 && P(o_t = V_2 \mid q_t = S_3) = 0.9,
\end{aligned}$$

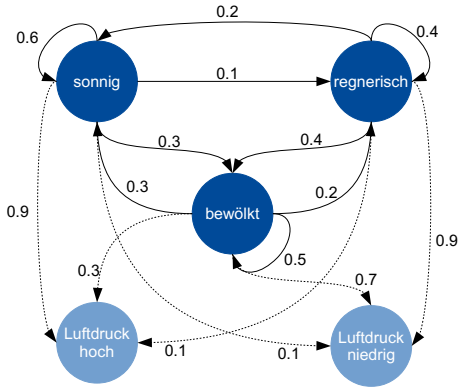


Abbildung 5.2: Graphendarstellung des HMMs aus Beispiel 4.

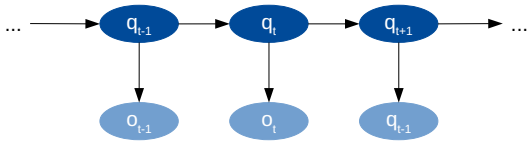


Abbildung 5.3: Abhängigkeiten von Beobachtungen und Systemzuständen im Hidden-Markov-Modell. Abbildungsidee aus [105] entnommen.

welche kompakt in der Beobachtungsmatrix

$$B = \begin{pmatrix} 0.9 & 0.1 \\ 0.3 & 0.7 \\ 0.1 & 0.9 \end{pmatrix} \tag{5.6}$$

hinterlegt werden können. Das gesamte System ist als gerichteter Graph in Abbildung 5.2 dargestellt. Um mit diesem Modell Vorhersagen oder Abschätzungen über Systemzustände treffen zu können, werden im folgenden Abschnitt Basisprobleme von HMMs und zugehörige Lösungsansätze vorgestellt. Das Szenario wird daraufhin in Beispiel 6 erneut aufgegriffen.

Neben dem veranschaulichenden Wetterbeispiel sei abschließend ein Experimental-Setup zur Aktivitätserkennung und -vorhersage in einem Meeting-Szenario, basierend auf Untersuchungen an der Universität Rostock [27, 28, 109], vorgestellt. Eine Diskussion der Überführung dieses Szenarios für SQL-basierte In-Datenbank-Verarbeitungen wurde erstmalig, im Rahmen des hier vorgestellten Forschungsprojektes, in [9] geführt.

Beispiel 5 (Meeting-Szenario [9,27,28,109])**.** Im Folgenden wird die Modellierung eines Meeting-Szenarios mehrerer Probanden in einem Smartlab durch ein HMM beschrieben. Es wurden hierfür drei Probanden („A“, „B“ und „C“) mit Sensoren ausgestattet, welche zu jedem diskreten Zeitpunkt die Ebenenpositionen (x- und y Koordinaten) aufgezeichnet haben. Für den Prozess des Lernens der HMM-Modellparameter (vgl. Problem 3 im Folgeabschnitt 5.2) wurden für jeden einzelnen Probanden zu jedem Zeitpunkt der aktuelle Zustand aus dem individuellen

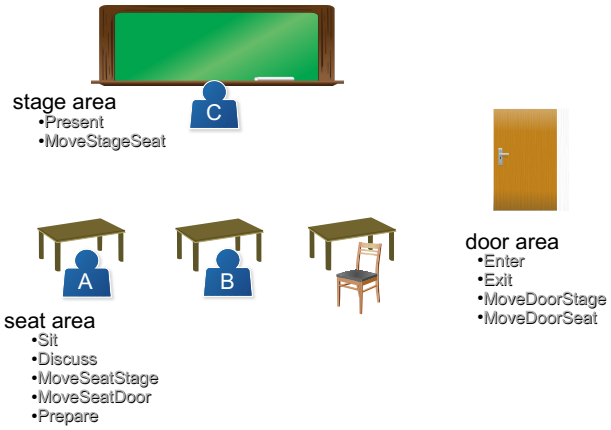


Abbildung 5.4: Beispielhafter Zustand des Meeting-Szenarios aus Beispiel 5. Die Abbildung wurde erstmalig in [9] genutzt.

Zustandsraum

$$S_{\text{Proband}} = \{\text{enter},\ \text{moveDoorStage},\ \text{moveDoorSeat},\ \text{moveSeatDoor},\ \text{moveStageSeat},\ \text{moveSeatStage},\ \text{present},\ \text{sit},\ \text{discuss},\ \text{exit}\}$$

notiert. Da in HMM nur eine einzelne diskrete Zustandsvariable für die Beschreibung des Prozess-Zustandes genutzt wird [105], ist es in diesem Fall nötig, die individuellen Zustände der Probanden per äußerem Produkt in der Form $S_{\text{A}} \times S_{\text{B}} \times S_{\text{C}}$ zu verknüpfen. Beispielsweise ist in Abbildung 5.4 der Systemzustand

$$q_t = (\text{sit}, \text{sit}, \text{present})$$

abgebildet. Mit diesen annotierten Zuständen können die HMM-Parameter A und B geschätzt werden.

Die Zustandstupel selbst sind hierbei offenbar nicht direkt durch Sensoren beobachtbar. In diesem Fall wird aus der Kombination der horizontalen Positionen der Probanden ein Beobachtungssymbol errechnet, welches das Folgern des Systemzustandes aufgrund der Wahrscheinlichkeitsverteilung ermöglicht. Befinden sich beispielsweise alle Probanden an der Tür, ist es wahrscheinlich, dass diese in den Raum ein- oder austreten wollen. Die Unterscheidung der Intention kann dann mithilfe der Position des vorherigen Zeitschrittes geschehen: Waren die Personen im vorigen Zeitschritt im Raum, ist die Wahrscheinlichkeit hoch, dass sie nun aus den Raum austreten möchten und vice versa. Hierfür wurde der Raum des Meetings in mehrere für das Szenario

essentielle Teilabschnitte

$$V_{\text{Proband}} = \{\text{in seat},\ \text{at stage},\ \text{at door},\ \text{else where}\}$$

aufgeteilt. Die Positionsdaten können daraufhin auf die jeweilige Position abgebildet werden. Hierbei werden die individuell beobachteten Emissionen des HMM ebenfalls durch äußere Produkte kombiniert.
Das vorgestellte Szenario erzeugt ein verhältnismäßig kleines HMM. Aus der Struktur sind jedoch zwei Dinge ersichtlich: die exponentielle steigende Größe des Zustandsraum und das potenzielle Auftreten dünn besetzter Übergangs- und Beobachtungsmatrizen. Die Zustandsraumgröße in diesem Beispiel beträgt

$$|S_{\text{Proband}}|^{\text{Anzahl Probanden}}$$

und steigt demnach exponentiell mit der Menge der Probanden. Selbst Meeting-Szenarien, in denen eine moderate Anzahl von Teilnehmern mit nur begrenzter Aktivitätsspielraum agieren, können hierbei schnell in den Big-Data-Bereich führen. Ähnlich zu der Google-Matrix aus Beispiel 3 ist zudem leicht eine dünn besetzte Struktur der Zustands- und auch Beobachtungsmatrix motivierbar. So erfüllen mehrere Zustände, wie beispielsweise `sit` und `present` oder auch `sit` und `exit`, keine logische Nachbarschaftsbeziehung. Dieser Effekt wird zusätzlich durch die Verbindung der individuellen Zustände im Allgemeinen verstärkt. Analog hierzu ist in Abbildung 5.4 beispielhaft dargestellt, dass nicht jede Beobachtung in einem Zustand auftreten können muss. In diesem Fall ist beispielsweise ein aktiv präsentierender Teilnehmer nicht im Tür-Bereich zu erwarten. Dies zeigt ebenfalls, dass eine Unterscheidung dünn und dicht besetzter Beobachtungsmatrizen im weiteren Verlauf sinnvoll sein kann.

5.2 Grundprobleme von Hidden-Markov-Modellen

Nach der strukturellen Beschreibung werden im Folgenden grundlegende Probleme und Lösungsansätze von HMMs vorgestellt. Diese werden daraufhin im Folgeabschnitt 5.4 für die Überführung in SQL auf deren wesentlichen Operationen strukturell untersucht. Diese werden als Basis genutzt für die hier präsentierte Diskussion zur Überführbarkeit wissenschaftlicher Methoden und HMM in SQL. Neben den Basisoperatoren, werden in Kapitel 7 zusätzlich das im Folgenden vorgestellte essenzielle Forward-Verfahren für die Diskussion der Anfrageformulierung und dem Einfluss von Indexstrukturen ausgewertet. Eine vollständige Übersetzung der hier präsentierten Algorithmen ist im Anhang B.7 aufgeführt.
Rabiner beschreibt in [104] im Kontext von HMMs die drei wesentlichen Probleme:

1. Wie hoch ist die Wahrscheinlichkeit $P(\mathbf{o} \mid \lambda)$ einer Beobachtungsfolge $\mathbf{o} = o_{1:T}$ in einem HMM λ?

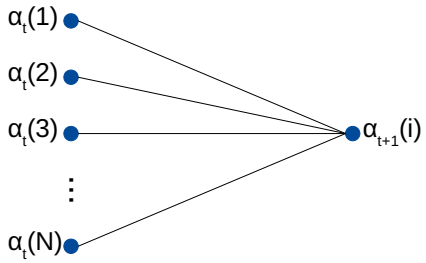


Abbildung 5.5: Grafische Darstellung der Abhängigkeit von Forward-Variablen α_t.

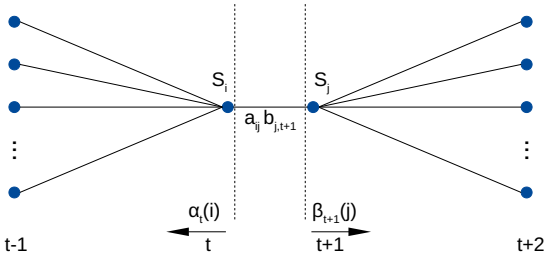


Abbildung 5.6: Grafische Darstellung der Abhängigkeiten für die Wahrscheinlichkeit, dass das System im Zustand S_i zum Zeitpunkt t und im Zustand S_j im Zeitpunkt $t+1$ ist.

2. Welche ist die wahrscheinlichste Zustandssequenz $\text{argmax}_{\mathbf{q}}(P(\mathbf{q} \mid \mathbf{o}))$ bezüglich einer Beobachtungsfolge $\mathbf{o}$?

3. Wie müssen die Parameter A, B, π gewählt werden, um das Auftreten einer Beobachtungssequenz $\mathbf{o}$ zu maximieren?

Diese Probleme werden in vielen Arbeiten um HMMs als die wesentlichen Probleme referenziert (beispielsweise [106, 110–112]) und bieten sich daher als gute Basis für einführende Betrachtungen an. Die Problemmenge ist hierbei offenbar nicht erschöpfend. So führen beispielsweise *Russell* und *Norvig* in [105] noch weitere Probleme ein. Hierzu zählen etwa die Glättung (die Berechnung von $P(q_k \mid o_{1:t})$ mit $k \in \mathbb{N}$ und $0 \leq k < t$) oder die Vorhersage (die Berechnung von $P(q_{t+k} \mid o_{1:t})$ für $k \in \mathbb{N}^{>0}$). Wie in [105] beschrieben wird, kann zumindest die Vorhersage durch leichte Anpassung des Lösungsansätze zur Berechnung von $P(\mathbf{o})$ bestimmt werden. Für nähere Erläuterungen zur Vorhersage und Glättung sei jedoch auf die angegebene Literatur verwiesen.

Problem 1: Wahrscheinlichkeit von Beobachtungsfolgen

Gegeben sei ein HMM λ und eine Beobachtungssequenz $\mathbf{o} = o_{1:T}$. Ziel ist es, die Wahrscheinlichkeit $P(\mathbf{o} \mid \lambda)$, d.h. die Wahrscheinlichkeit der Beobachtungsfolge $\mathbf{o}$, zu berechnen. Dies ist nicht nur von Bedeutung für Vorhersagen des Modells, sondern ist auch ein wesentliches Teilproblem des Lernens von HMM-Parametern (Problem 3). Beispielsweise kann die Wahrscheinlichkeit als Maß für die Güte eines HMM bezüglich einer Trainingssequenz interpretiert werden.

Der wohl intuitivste Ansatz zur Berechnung von $P(\mathbf{o} \mid \lambda)$ ist die Aufsummierung der Wahrscheinlichkeiten der gegebene Beobachtungsfolge $\mathbf{o}$ bezüglich einer möglichen Zustandssequenz

$\mathbf{q}$:

$$\begin{aligned} P(\mathbf{o} \mid \lambda) &= \sum_{\mathbf{q} \in S^T} P(\mathbf{o} \mid \mathbf{q}, \lambda) P(\mathbf{q} \mid \lambda) \\ &= \sum_{\mathbf{i} \in \{1,\ldots,n\}^T}^{T} \pi_{i_1} b_{i_1,o_1} \prod_{k=1}^{T-1} a_{i_k,i_{k+1}} b_{i_{k+1},o_{k+1}}. \end{aligned}$$

Eine direkte Berechnung benötigt hierbei $2TN^T$ Operationen, welches bereits für kleine Werte N und T sehr aufwendig ist. Um dieses Problem zu umgehen, können die *Forward-Variablen*

$$\alpha_t(i) = P((o_i)_{i=1,\ldots,t}, q_t = S_i \mid \lambda) \tag{5.7}$$

genutzt werden, welche rekursiv berechnet werden können. Das zugehörige Forward-Verfahren ist in Algorithmus 4 beschrieben. Anstatt alle möglichen Zeit-Zustandskombination einzeln zu berechnen, werden, wie in Abbildung 5.5 dargestellt, für jeden Zeitstempel die Wahrscheinlichkeiten der Zustandssequenzen mithilfe der Zwischenergebnisse des jeweiligen vorangegangen Zeitschritts berechnet. Die ursprüngliche Unbekannte $P(\mathbf{o} \mid \lambda)$ kann dann abschließend durch Aufsummierung der Wahrscheinlichkeiten $P(\mathbf{o}, q_t \mid \lambda)$ berechnet werden. Im Gegensatz zum intuitiven Ansatz nutzt dieses Verfahren lediglich N^2T Fließkommaoperationen.

Alternativ hierzu kann die Wahrscheinlichkeit $P(\mathbf{o} \mid \lambda)$ rekursiv mittels der *Backward-Variablen*

$$\beta_t(i) = P((o_i)_{i=t+1,\ldots,T} \mid q_t = S_i, \lambda)$$

der Zeit entgegengesetzt berechnet werden. Das zugehörige Verfahren ist in Algorithmus 6 dargestellt und benötigt, aufgrund seiner strukturellen Ähnlichkeit zum Forward-Algorithmus, ebenfalls N^2T Operationen.

Beide Variablen α_t und β_t sind essenzieller Bestandteil des *Baum-Welch-Verfahrens* zur Lösung des Lernproblems (Problem 3).

Wie in den Algorithmen 5 und 7 gezeigt wurde, lässt sich bereits hier beobachten, dass die Algorithmen sich kompakt in Matrix-Vektor-Schreibweise darstellen lassen. Dies ist insbesondere von Bedeutung für die in Kapitel 9 vorgestellte Übersetzung der Algorithmen in SQL.

Beispiel 6 (Forward-Verfahren für erweitertes Wettermodell)**.** Für das erweiterte Wettermodell aus Beispiel 4 soll im Folgenden beispielhaft die Wahrscheinlichkeit der Beobachtung $\mathbf{o} = (V_1, V_2)$ (V_1 =`Luftdruck hoch`, V_2 =`Luftdruck niedrig`) bestimmt werden, wobei es „heute“ sonnig ist $\pi = (1, 0, 0)^T$. Hierfür wird nun das Forward-Verfahren aus Algorithmus 5 genutzt. Der

Algorithmus 4 Das Forward-Verfahren

Input: $\mathbf{o} = (o_1, \ldots, o_T)$ ▷ Beobachtungssequenz
for $i = 1, \ldots, N$ **do**
  $\alpha_1[i] = \pi_i b_{i,o_1}$ ▷ Initialisierung
end for
for $t = 1, \ldots, T-1$ **do** ▷ Iteration
  for $i = 1, \ldots, N$ **do**
    $\alpha_{t+1}[i] := \left(\sum_{j=1}^{N} \alpha_t[j] a_{i,j}\right) b_{i,o_{t+1}}$
  end for
end for
$P(\mathbf{o}|\lambda) = \sum_{i=1}^{N} \alpha_T[i]$, ▷ Terminierung

Initialisierungsschritt für `Luftdruck hoch` ergibt zunächst:

$$\begin{aligned}\alpha_1 &= \pi \odot B_{:,1} \\ &= \begin{pmatrix}1\\0\\0\end{pmatrix} \odot \begin{pmatrix}0.9\\0.3\\0.1\end{pmatrix} \\ &= \begin{pmatrix}0.9\\0\\0\end{pmatrix}\end{aligned}$$

Darauf folgt ein rekursiver Berechnungsschritt für α_2 gemäß `Luftdruck niedrig` mit

$$\begin{aligned}\alpha_2 &= (A^T \alpha_1) \odot B_{:,2} \\ &= \left(\begin{pmatrix}0.6 & 0.3 & 0.2\\0.3 & 0.5 & 0.4\\0.1 & 0.2 & 0.4\end{pmatrix}\begin{pmatrix}0.9\\0\\0\end{pmatrix}\right) \odot \begin{pmatrix}0.1\\0.7\\0.9\end{pmatrix} \\ &= \begin{pmatrix}0.054\\0.189\\0.081\end{pmatrix}\end{aligned}$$

und der Terminierungsschritt

$$P((V_1, V_2) \mid \lambda) = 0.054 + 0.189 + 0.81 = 0.324.$$

Dies bedeutet, dass die Wahrscheinlichkeit der Beobachtung von (`Luftdruck hoch`, `Luftdruck niedrig`) in dem spezifierten HMM λ bei 32.4% liegt.

Algorithmus 5 Das Forward-Verfahren aus Algorithmus 4 in Matrix-Vektor-Form

Input: $\mathbf{o} = (o_1, \ldots, o_T)$ ▷ Beobachtungssequenz
$\alpha_1 := \pi \odot B_{:,o_1}$ ▷ Elementweise-Multiplikation $\odot$
for $t = 1, \ldots, T-1$ **do** ▷ Iteration
  $\alpha_{t+1} := (A^T \alpha_t) \odot B_{:,o_{t+1}}$
end for
$P(\mathbf{o}|\lambda) = \text{sum}(\alpha_T)$, ▷ Terminierung

Algorithmus 6 Das Backward-Verfahren

Input: $\mathbf{o} = \{o_1, \ldots, o_T\}$ ▷ Beobachtungssequenz
for $i = 1, \ldots, N$ **do**
  $\beta_T[i] := 1$ ▷ Initialisierung
end for
for $t = T-1, \ldots, 1$ **do** ▷ Iteration
  for $i = 1, \ldots, N$ **do**
    $\beta_t[i] := \sum_{j=1}^{N} a_{i,j} \beta_{t+1}[j] b_{j,o_{t+1}}$
  end for
end for
$P(\mathbf{o}|\lambda) = \sum_{i=1}^{N} \beta_1[i] b_{i,o_1} \pi_i$ ▷ Terminierung

Algorithmus 7 Das Backward-Verfahren aus Algorithmus 6 in Matrix-Vektor-Form

Input: $\mathbf{o} = \{o_1, \ldots, o_T\}$ ▷ Beobachtungssequenz
$\beta_T := \mathbf{1}$ ▷ Initialisierung
for $t = T-1, \ldots, 1$ **do** ▷ Iteration
  $\beta_t := A(\beta_{t+1} \odot B_{:,o_{t+1}})$ ▷ Elementweise-Multiplikation $\odot$
end for
$P(\mathbf{o}|\lambda) = \langle \beta_1, B_{:,o_1} \odot \pi \rangle$ ▷ Terminierung

Problem 2: Wahrscheinlichste Zustände

Im Gegensatz zum Problem 1 ist das Berechnen der wahrscheinlichsten Zustandsfolge $\mathbf{q}$ bezüglich einer Beobachtungssequenz $\mathbf{o}$ relativ und hängt von der Wahl der Optimierungskriterien ab. So führt beispielsweise die Wahrscheinlichkeitsmaximierung bezüglich jedes individuellen Zeitschritts t

$$\max_{i \in \{1,\ldots,n\}} P(q_i \mid \mathbf{o}_t, \lambda)$$

dazu, dass die entstehende Zustandssequenz möglicherweise nicht gültig in diesem Modell ist. Dies kann beispielsweise das Aufeinanderfolgen zweier Zustände S_i zu S_j sein, obwohl für deren Übergangswahrscheinlichkeit $a_{ij} = 0$ gilt.

Um dieses Problem zu umgehen, wird im *Viterbi-Algorithmus* die Maximierung der Zustandsfolge bezüglich aller Zeitschritte

$$\max_{\mathbf{q}} P(\mathbf{q} \mid \mathbf{o}, \lambda)$$

verfolgt.

Für die Berechnung werden die Hilfsvariablen δ_t definiert, welche die Wahrscheinlichkeit einer möglichen Zustandssequenz bezüglich der gegebenen Beobachtungsfolge $\mathbf{o}$ hält. Demnach gilt

$$\delta_t(i) = \max_{q_1,\ldots,q_{t-1}} P(\mathbf{q}_{1:t-1}, q_t = S_i, \mathbf{o}_{1:t} \mid \lambda)$$

oder äquivalent in rekursiver Form

$$\delta_{t+1}(j) = (\max_i \delta_t(i) a_{ij}) \cdot B_{j,o_{t+1}}.$$

Mit diesen Hilfsvariablen kann die wahrscheinlichste Sequenz ähnlich zu dem Ansatz des Forward-Verfahrens berechnet werden. Der zugehörige vollständige Algorithmus ist in Algorithmus 8 dargestellt und kann ebenfalls in Vektorform, wie in Algorithmus 9 beschrieben, überführt werden.

Beispiel 7 (Viterbi-Verfahren für das erweiterte Wettermodell)**.** Analog zu Beispiel 6 wurde die Folge (`Luftdruck hoch`, `Luftdruck niedrig`) beobachtet. Im Folgenden wird die wahrscheinlichste Zustandsfolge mittels Viterbi-Verfahren ermittelt.

Die Initialisierung der Parameter bezüglich V_1 = `Luftdruck hoch` und dem initialen Zustand

$\pi = (1, 0, 0)^T$ ergibt

$$\delta_{:,1} = \pi \odot B_{:,1} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \odot \begin{pmatrix} 0.9 \\ 0.3 \\ 0.1 \end{pmatrix} = \begin{pmatrix} 0.9 \\ 0 \\ 0 \end{pmatrix}$$

$$\psi_{:,1} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Der Iterationsschritt für $V_2 =$ `Luftdruck niedrig` folgt mit

$$\psi_{:,2} = [\text{argmax}((\delta_{t-1} \cdot \mathbf{1}^T) \odot A)]^T = \left[\text{argmax}\left(\begin{pmatrix} 0.9 & 0.9 & 0.9 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \odot \begin{pmatrix} 0.6 & 0.3 & 0.1 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.4 & 0.4 \end{pmatrix}\right)\right]^T$$

$$= \left[\text{argmax}\left(\begin{pmatrix} 0.54 & 0.27 & 0.09 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}\right)\right]^T = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$\delta_{:,2} = [\max((\delta_{t-1} \cdot \mathbf{1}^T) \odot A)]^T \odot B_{:,2}$$

$$= \left[\begin{pmatrix} 0.54 & 0.27 & 0.09 \end{pmatrix}\right]^T \odot \begin{pmatrix} 0.1 \\ 0.7 \\ 0.9 \end{pmatrix} = \begin{pmatrix} 0.054 \\ 0.189 \\ 0.081 \end{pmatrix}.$$

Daraus lassen sich die Wahrscheinlichkeit und die Zustandsfolge

$$P^* = 0.189$$
$$\mathbf{q}^* = \begin{pmatrix} \psi_{q_2^*,2} & \text{argmax}(\delta_{:,2}) \end{pmatrix} = \begin{pmatrix} \psi_{2,2} & 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 \end{pmatrix}$$

berechnen. Die wahrscheinlichste Zustandsfolge entspricht demnach (`sonnig`, `bewölkt`).

Problem 3: Modellberechnung

Um Wahrscheinlichkeiten von Beobachtungs- und Zustandsfolgen zu berechnen oder Vorhersagen zu ermöglichen, müssen offensichtlich die Modellparameter A, B und π des HMM bekannt sein. Das Lernen der Matrizen A, B und der Anfangswahrscheinlichkeit π kann auf verschiedene Weisen geschehen und ist abhängig von mehreren Aspekten, wie beispielsweise der Existenz von Trainingsdaten (Beobachtungsfolgen) mit oder ohne Annotationen (Zustände).
Beiden Szenarien werden in dieser Arbeit unterschieden, teilen sich hierbei jedoch die prinzipielle

Algorithmus 8 Das Viterbi-Verfahren

Input: $\mathbf{o} = \{o_1, \ldots, o_T\}, \lambda, I = \{1, \ldots, N\}$
for $i = 1, \ldots, N$ **do**
  $\delta_1[i] = \pi_i b_{i,o_1}$ ▷ Initialisierung
  $\psi_1[i] = 0$
end for
for $t = 2, \ldots, T$ **do** ▷ Rekursion
  for $i = 1, \ldots, N$ **do**
    $\delta_t[i] = \max_{j \in I}(\delta_{t-1}[j] a_{j,i}) b_{i,o_t}$
    $\psi_t[i] = \operatorname{argmax}_{j \in I}(\delta_{t-1}[j] a_{j,i})$
  end for
end for

$P^* = \max_{i \in I}(\delta_T[i])$ ▷ Initialisierung Output
$q_T^* = \operatorname{argmax}_{i \in I}(\delta_T[i])$

for $t = T - 1, \ldots, 1$ **do** ▷ Output
  $q_t^* = \psi_{t+1}[q_{t+1}^*]$
end for

Algorithmus 9 Das Viterbi-Verfahren aus Algorithmus 8 in Matrix-Vektor-Form

Input: $\mathbf{o} = \{o_1, \ldots, o_T\}, \lambda, I = \{1, \ldots, N\}$
$\delta_1 = \pi \odot B_{:,o_1}$ ▷ Elementweise-Multiplikation $\odot$
$\psi = \mathbf{0}$
for $t = 2, \ldots, T$ **do** ▷ Rekursion
  $\psi_t = [\operatorname{argmax}((\delta_{t-1} \cdot \mathbf{1}^T) \odot A)]^T$ ▷ Argumente der Spaltenweise-Maximierung argmax
  $\delta_t = [\max((\delta_{t-1} \cdot \mathbf{1}^T) \odot A)]^T \odot B_{:,o_t}$ ▷ Spaltenweise-Maximierung max
end for

$P^* = \max_{i \in I}(\delta_T)$ ▷ Initialisierung Output
$q_T^* = \operatorname{argmax}_{i \in I}(\delta_T)$

for $t = T - 1, \ldots, 1$ **do** ▷ Output
  $q_t^* = \psi_{t+1}(q_{t+1}^*)$
end for

Algorithmus 10 Baum-Welch-Algorithmus zur Schätzung optimaler Parameter A, B, π in Matrix-Vektor-Form.

Input: $\mathbf{o} = \{o_1, \ldots, o_T\}$ ▷ Observationssequenz
Startparameter wählen: A, B, π
for $k = 1, \ldots,$Max_Iteration **do**
  $\alpha_1 = \pi \odot B_{:,o_1}$ ▷ Forward-Variablen
  for $t = 1, \ldots, T-1$ **do**
    $\alpha_{t+1} = (\alpha_t^T A) \odot B_{:,o_{t+1}}$
  end for

  $\beta_T = \mathbf{1}$ ▷ Backward-Variablen
  for $t = T-1, \ldots, 1$ **do**
    $\beta_t = A(B_{:,o_{t+1}} \odot \beta_{t+1})$
  end for

  for $t = 1, \ldots, T$ **do** ▷ Gamma-Variablen
    $\gamma_t = \alpha_t \odot \beta_t / \langle \alpha_t, \beta_t \rangle$
  end for

  for $t = 1, \ldots, T-1$ **do** ▷ Xi-Variablen
    $\xi_t = A \odot (\alpha_t \cdot (\beta_{t+1} \odot B_{:,o_{t+1}})^T) / (\alpha_t^T A(B_{:,o_{t+1}} \odot \beta_{t+1}))$
  end for

  $\overline{\pi} = \gamma_1$ ▷ Update
  $\overline{A} = (\sum_{t=1}^{T-1} \xi_t[i,j]) / (\sum_{t=1}^{T-1} \gamma_t) \cdot \mathbf{1}^T)$ ▷ Elementweise Division /
  for $j = 1, \ldots, M$ **do**
    $(\overline{B})_{:,j} = (\sum_{t=1}^{T} \delta_{o_t, V_j} \gamma_t) / ((\sum_{t=1}^{T} \gamma_t))$ ▷ Elementweise Division /
  end for

  if Abbruchbedingung **then**
    return $\overline{A}, \overline{B}, \overline{\pi}$
  else
    $A = \overline{A}, B = \overline{B}, \pi = \overline{\pi}$
  end if
end for

Interpretation [104]:

$$\pi_i = \text{Erwartete Frequenz von Zustand } S_i \text{ bei } t = 1 \tag{5.8}$$

$$a_{ij} = \frac{\text{Erwartete Anzahl von Übergängen von } S_i \text{ zu } S_j}{\text{Erwartete Anzahl von Übergängen ausgehend von } S_i} \tag{5.9}$$

$$b_{ij} = \frac{\text{Erwartete Anzahl von Zustand } S_i \text{ und gleichzeitiger Beobachtung } V_j}{\text{Erwartete Anzahl von Zuständen in } S_i} \tag{5.10}$$

Sei zunächst angenommen, dass annotierte Trainingsdaten vorhanden sind. In diesem Fall können die aus den Sensordaten berechneten Beobachtungen in Zeitreihen der Form (Zeitstempel, Zustand, Beobachtung) beziehungsweise (t_i, q_i, o_i) generiert werden. Mittels dieser Folgen kann dann der vorherige Ansatz durch Auszählen von Tupeln und Tupelfolgen durch

$$\begin{aligned} \pi_i &= \frac{\#q_1 = S_i}{\#\text{Trainingsläufe}} \\ a_{ij} &= \frac{\#q_t = S_i \wedge q_{t+1} = S_j}{\#q_t = S_i} \\ b_{ij} &= \frac{\#q_t = S_i \wedge o_t = V_j}{\#q_t = S_i}. \end{aligned}$$

bestimmt werden. Dieser Ansatz wurde in [9] im Zuge der vorgestellten Arbeit erstmalig auf Datenbanksystemen evaluiert. Die zugehörigen Ergebnisse und deren Einordnung sind in Abschnitt 10.2.2 in Kurzform zusammengefasst.

Wie in [26] beschrieben, ist es nicht immer möglich annotierte Trainingsdaten zu erhalten. Die Erstellung dieser kann beispielsweise zu teuer oder generell unpraktikabel sein, etwa aufgrund der Menge verschiedener wesentlicher Szenarien, die aufgenommen werden müssten. In diesen Fällen ist eine Betrachtung nicht-annotierter Trainingsdaten nötig.

Hier wird versucht, die Parameter A, B und π aus einer oder mehrerer (errechneter) Beobachtungsfolgen $\mathbf{o} \in V^T$ ($T \in \mathbb{N}$) so zu berechnen, dass die Wahrscheinlichkeit dieser bezüglich des Modells λ maximiert wird. Die Schwierigkeit hierbei besteht darin, dass es keinen bekannten Weg gibt, eine analytische Lösung zu bestimmen, und ebenfalls keinen „optimalen“ Weg, die Parameter zu schätzen [104]. Als einer der verbreitetsten Ansätze zur Lösung dieses Problems wird im *Baum-Welch*-Algorithmus ein Modell realisiert, indem die Wahrscheinlichkeit $P(\mathbf{o} \mid \lambda)$ lokal maximiert wird. Dieses Verfahren basiert maßgeblich auf den im ersten Problem eingeführten Forward- und Backward-Variablen α_t und β_t.

Für eine Beschreibung des Algorithmus seien zum einen die Hilfsvariablen

$$\xi_t(i, j) = P(q_t = S_i, q_{t+1} = S_j \mid \mathbf{o}, \lambda)$$

eingeführt, welche die Wahrscheinlichkeit des Zustandsübergangs von S_i zu S_j im Zeitpunkt t

im aktuellen Modell beschreibt. Zusätzlich werden die Variablen

$$\gamma_t(i) = P(q_t = S_i \mid \mathbf{o}, \lambda)$$

betrachtet, welche die Wahrscheinlichkeit beschreibt, dass das HMM λ im Zustand S_i zum Zeitpunkt t bei einer Beobachtungsfolge $\mathbf{o}$ ist. Die Summierung von $\xi_t(i,j)$ über die Zeit t kann in diesem Fall als Maß für Zustandsübergänge von S_i zu S_j verstanden werden und demnach die Summierung von $\gamma_t(i)$ als Maß für das Auftreten von S_i. Nach 5.8 bis 5.10 können daher die Parameter mit diesen Hilfsvariablen durch

$$\begin{aligned}
\overline{\pi}_i &= \gamma_1(i) \\
\overline{a}_{ij} &= \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \\
\overline{b}_{i,k} &= \frac{\sum_{t=1}^{T} \delta_{o_t,v_k} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)}
\end{aligned}$$

dargestellt werden.

In [104] wird weiter gezeigt, dass die Hilfsvariablen insbesondere mittels der HMM-Parameter und den Forward- und Backward-Variablen

$$\xi_t(i,j) = \frac{\alpha_t[i] a_{ij} b_{j,o_{t+1}} \beta_{t+1}[j]}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_t[i] a_{ij} b_{j,o_{t+1}} \beta_{t+1}[j]}$$

und

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(\mathbf{o} \mid \gamma)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_t(i)\beta_t(i)}$$

dargestellt werden können. Diese Beziehung ist für ξ_t in Abbildung 5.6 grafisch motiviert.

Beide Variablen können analog zu den Forward- und Backward-Variablen mittels Vektoren und Matrizen durch

$$\xi_t = \frac{A \odot (\alpha_t (\beta_{t+1} \odot B_{:,o_{t+1}})^T)}{(\alpha_t^T A (B_{:,o_{t+1}} \odot \beta_{t+1}))}$$

und

$$\gamma = \frac{\alpha_t \odot \beta_t}{\langle \alpha_t, \beta_t \rangle}$$

zusammengefasst werden. Das zugehörige Verfahren ist in Algorithmus 10 dargestellt. Es wurde von *Baum et al* gezeigt, dass dieses monoton gegen ein lokales Maximum konvergiert [104]. Dies bedeutet entweder, dass die aktuellen Parameter λ_i (Modellparameter nach i Iterationen) bereits ein Extrempunkt sind oder, dass die Aktualisierung dieser keine Minderung der Wahr-

scheinlichkeit zur Folge hat:

$$P(\mathbf{o} \mid \lambda_i) \geq P(\mathbf{o} \mid \lambda_{i-1}).$$

Zudem werden bei der Aktualisierung insbesondere automatisch die stochastischen Bedingungen

$$\begin{aligned} \sum_{i=1}^{N} \pi_i &= 1 \\ \sum_{j=1}^{N} a_{kj} &= 1 \qquad \forall k \in \{1, \ldots, N\} \\ \sum_{i=1}^{N} b_{ik} &= 1 \qquad \forall k \in \{1, \ldots, M\} \end{aligned}$$

erfüllt. Die Konvergenz in lokale Maxima ist im Allgemeinen kritisch zu betrachten, da typischerweise der Optimierungsraum sehr komplex ist und viele solcher Extrema aufweist. Entscheidend für das Finden eines globalen Maximums kann hierbei etwa eine geeignete Initialisierung der HMM-Parameter durch a-priori-Wissen sein. Aufgrund der Komplexität und dem daraus folgenden Platzbedarf, wird das Baum-Welch-Verfahren anhand des erweiterten Wettermodells im Anhang B.1 präsentiert.

5.3 Numerische Stabilität

Die vorgestellten Lösungsverfahren der drei Grundprobleme können bereits bei moderaten Zustandsraumgrößen oder Längen von Beobachtungsfolgen, aufgrund langer Produktfolgen oder Summen von Wahrscheinlichkeitswerten, auf numerische Instabilitäten stoßen. Dies führt dazu, dass Zwischenergebnisse schnell Werte nahe 0 annehmen, die mit klassischen Computerarithmetiken (etwa der *double precision*) nicht mehr adäquat dargestellt werden können. Um diesem Problem entgegenzutreten, existieren mehrere mögliche Ansatzpunkte. Zwei allgemeingültige Ansätze sind etwa die Verwendung längerer Bit-Darstellungen für die Darstellung von Wahrscheinlichkeiten und das Nutzen stabilerer Berechnungsansätze von Basisoperationen. Hierzu zählt etwa die Kahan-Summation, die vor allem für die Addition vieler Summanden beziehungsweise für Summanden mit stark variierenden totalen Werten eingesetzt wird. Die Nutzung des Verfahrens wurde beispielsweise im Kontext von Big-Data-Machine-Learning in Apache SystemML in [73] propagiert. Eine SQL:1999.Implementation der Kahan-Summation mittels rekursiver Anfragen ist in Anhang B.1.1 dargestellt und wird im weiteren Verlauf der Arbeit nicht näher beleuchtet. Speziell für die vorgestellten Algorithmen wurden bereits von *Rabiner* in [104] im Jahre 1989 zwei Möglichkeiten zur Sicherung der numerischen Stabilität diskutiert. Im Folgenden werden diese näher beschrieben.

Algorithmus 11 Numerisch stabilereres Forward-Verfahren.

Input: $\mathbf{o} = (o_1, \ldots, o_T)$ ▷ Beobachtungssequenz
$\tilde{\alpha}_{\mathbf{1}} := \pi \odot B_{:,o_1}$ ▷ Elementweise-Multiplikation $\odot$
$\mathfrak{S}_1 = \|\tilde{\alpha}_{\mathbf{1}}\|_1$
$\alpha_1 = \tilde{\alpha}_{\mathbf{1}}/\mathfrak{S}_1$
for $t = 1, \ldots, T-1$ **do** ▷ Iteration
  $\tilde{\alpha}_{\mathbf{t+1}} := (\alpha_t^T A) \odot B_{:,o_{t+1}}$
  $\mathfrak{S}_{t+1} = \|\tilde{\alpha}_{\mathbf{t+1}}\|_1$
  $\alpha_{t+1} = \tilde{\alpha}_{\mathbf{t+1}}/\mathfrak{S}_{t+1}$
end for
$\log(P(\mathbf{o}|\lambda)) = \sum_{t=1}^{T} \log(\mathfrak{S}_t)$ ▷ Terminierung

Skalierung der Forward- und Backward-Variablen

Die erste Anpassung betrifft die Berechnung der Forward- und Backward-Variablen. Die Einträge dieser streben im Allgemeinen in vergleichsweise wenigen Zeitschritten gegen 0. Um Informationsverluste zu mindern, werden die Forward- und Backward-Variablen (falls nötig) in jedem Zeitschritt skaliert. Beschreibt etwa $\tilde{\alpha}_{\mathbf{t}}$ den Vektor der Forward-Variablen im Zeitschritt t nach Algorithmus 5, dann bezeichnet

$$\alpha_t = \frac{\tilde{\alpha}_t}{\mathfrak{S}_t}$$

den skalierten Vektor der Forward-Variablen mit zugehörigen Skalierungsfaktor

$$\mathfrak{S}_t = \|\tilde{\alpha}_{\mathbf{t}}\|_1 = \sum_{i=1}^{N} \tilde{\alpha}_t[i] = \langle \tilde{\alpha}_{\mathbf{t}}, \mathbf{1} \rangle. \tag{5.11}$$

Letztere sind typischerweise deutlich kleiner als 1, so dass die Werte von α_t folgerichtig größer werden und nach der Transformation in einen numerisch stabileren Bereich liegen. Ein Aussetzen der Skalierung kann konzeptionell durch $\mathfrak{S}_t = 1$ umgesetzt werden.
Werden die Backward-Variablen β_t ebenfalls mit dem Faktor $\mathfrak{S}_t$ aus (5.11) skaliert, verändern sich die Zwischenergebnisse ξ_t und γ_t des Baum-Welch-Verfahrens aufgrund einfacher Auslöschungseffekte der Faktoren nicht. Dadurch kann die numerische Stabilität erhöht werden, ohne weitere Änderungen des Verfahrens zu benötigen. Das angepasste Verfahren ist in Algorithmus 12 dargestellt.
Konträr hierzu, ist es bei dem Forward-Verfahren zur Berechnung von $P(\mathbf{o} \mid \lambda)$ nötig die Skalierungsfaktoren $\mathfrak{S}_t$ mitzuführen. Es kann jedoch durch Induktion gezeigt werden, dass

$$\alpha_t = \frac{\tilde{\alpha}_t}{\prod_{t=1}^{t} \mathfrak{S}_t}$$

gilt. Damit kann

$$\frac{\sum_{i=1}^{N} \tilde{\alpha}_T[i]}{\prod_{t=1}^{T} \mathfrak{S}_t} = 1$$

beziehungsweise

$$P(\mathbf{o} \mid \lambda) = \prod_{t=1}^{T} \mathfrak{S}_t \tag{5.12}$$

gefolgert werden. Das Produkt aus (5.12) nimmt im Allgemeinen mit bereits verhältnismäßig kurzen Beobachtungsfolgen (T) Werte nahe 0 an, die nur bedingt durch klassische Computer-Fließkomma-Repräsentationen dargestellt werden können. Um diesem Effekt entgegen zu wirken, kann das Produkt logarithmiert werden. Hierbei wird genutzt, dass der natürliche Logarithmus log auf $]0, 1[$ ein streng monoton steigende, bijektive, stetig invertierbare Funktion ist und

$$\log(a \cdot b) = \log(a) + \log(b) \quad \forall a, b \in \mathbb{R}^{>0}$$

gilt. Dadurch kann das Produkt (5.12) in

$$\log(P(\mathbf{o} \mid \lambda)) = \sum_{t=1}^{T} \log(\mathfrak{S}_t).$$

überführt werden. Die logarithmierten Werte der Wahrscheinlichkeitsfolgen, die gegen 0 streben, steigen in ihrem absoluten Wert und werden damit in numerisch stabilere Bereiche transformiert. Das angepasste Verfahren ist in Algorithmus 11 dargestellt.

Anpassung Viterbi-Verfahren

Analog zum Terminierungsschritt im Forward-Verfahren, kann das Viterbi-Verfahren von der Logarithmierung von Wahrscheinlichkeitsprodukten profitieren. Gemäß der Monotonie des Logarithmus können, anstatt der Maximierung der Wahrscheinlichkeiten $P(\mathbf{q}, \mathbf{o} \mid \lambda)$, die Maximierung der logarithmierten Werte $\log(P(\mathbf{q}, \mathbf{o} \mid \lambda))$ betrachtet werden. Die Anpassung ist in Algorithmus 13 umgesetzt.

5.4 Strukturelle Analyse der Grundoperationen

Aus den finalen Algorithmen 11, 12 und 13 ist ersichtlich, dass diese (nach deren Vektorisierung) maßgeblich durch Basisoperatoren der linearen Algebra, sowie einfachen Zuweisungen, Selektionen von Matrixspalten, skalaren mathematischen Funktionen (etwa der Logarithmus **log** [3]) und

[3] Darstellbar aus dem natürlichen Logarithmus **ln** aus SQL:2003 [113].

Tabelle 5.1: Benötigte Operationen zur Berechnung der Lösungsverfahren der Hidden-Markov-Problem aus Abschnitt 5.2. Hierbei bezeichnen A, B-Matrizen, $\mathbf{x}, \mathbf{y}$ Vektoren und a, b Skalare. Verfahren Nutzen die Akronyme: F- Forward, B-Backward, V- Viterbi, B-W - Baum-Welch, AL - annotiertes Lernen.

Operation	**Formel**	**Verfahren**
Aggregationen		
Maximales El.	$\mathbf{max}(a), \mathbf{max}$	VB-W
Argument des maximales El.	$\mathbf{argmax}(a), \mathbf{argmax}(A)$	V
Zählen	$\mathbf{count}$	AL
Vektor-Operationen		
Elementweise Vektor-Operationen	$\mathbf{x} \odot \mathbf{y}, \mathbf{x} + \mathbf{y}$	FBVB-W
Skalierung von Vektoren	$a\mathbf{x}, \mathbf{x}/a$	FBB-W
Transponierung	$\mathbf{x}^T$	FVB-W
Vektor-1-Norm / Skalarprodukt	$\|\mathbf{x}\|_1, \langle \mathbf{x}, \mathbf{y} \rangle$	FB-W
Dyadisches Produkt	$\mathbf{x}\mathbf{y}^T$	B-W
Matrix-Vektor-Produkt	$\mathbf{x}^T A, A\mathbf{x}$	FBB-W
Matrix-Operationen		
Elementweise Matrix-Operationen	$A \odot B$,$A + B$	VB-W
Diverses		
Kronecker-Delta	δ_{ij}	B-W
Logarithmierung	$\mathbf{log}(a), \mathbf{log}(\mathbf{x}), \mathbf{log}(A)$	FBV

Aggregationen (etwa **argmax**, **max**, **count**) besteht. Die letzten beiden sind insbesondere bereits seit SQL-86 Teil des Standards. Das Argument des maximalen Elements **argmax** lässt sich, abhängig vom gewählten Relationenschema, einfach durch eine geschachtelte Anfrage darstellen.[4]

Die vorgenommene Vektorisierung ist hierbei offensichtlich nicht eindeutig, so können beispielsweise die Variablen γ_t auch ohne Schleifen umgesetzt werden, insofern die Forward- und Backwardvariablen in Matrixform hinterlegt sind. Der Kern der Vektor-Operationen bleibt hierbei offenbar erhalten. Da die Überführung in das maßgeblich deklarative SQL naturgemäß nicht direkt umgesetzt werden kann, ist die Form der Vektorisierung hier nur als Motivation datenintensiver linearer Algebra zu verstehen. Eine Zusammenfassung der genutzten Operationen ist in Tabelle 5.1 dargestellt. In dieser ist ersichtlich, dass die wesentlichen Operationen (im Sin-

[4] Beschreibt etwa `f(x,y)` ein Relationenschema mit Schlüssel- bzw. Indexattribut `x` und Funktionswertattribut `y` in Anlehnung an eine Funktionsauswertung $f(x) = y$, kann $\mathbf{argmax}(f(x))$ etwa in erweiterter Relationenalgebra (vergleiche Abschnitt 7.1) durch

$$\gamma_{\mathbf{max}(x) \to x}(\sigma_{y=\gamma_{\mathbf{max}(y) \to y}(f)}(f))$$

bestimmt werden. Die äußere Maximierung wird in diesem Fall genutzt, um ein eindeutigen skalares Ergebnis, im Falle des multiplen Auftretens des maximalen Funktionswertes, zu erhalten und kann ebenfalls durch **min** ersetzt werden.

ne des Berechnungsaufwandes) die Matrix-Vektor-Produkte $A\mathbf{x}$ bzw. $\mathbf{x}^T A$, dyadische Produkte $\mathbf{x}\mathbf{y}^T$, sowie elementweise arithmetische Operationen $(+, -, /, \odot)$ auf (skalierten) Matrizen und Vektoren

$$aA \star (bB), a\mathbf{x} \star (b\mathbf{y}) \qquad (\star \in \{+, -, \odot, /\}) \tag{5.13}$$

darstellen. Die Matrizen und Vektoren arbeiten hierbei maßgeblich, bedingt durch den statistischen Hintergrund, mit Einträgen im Intervall $[0, 1]$. Wie in den Beispielen 3, 4 und 5 beschrieben, können die Matrizen, abhängig vom Modell, dicht oder dünn besetzt sein und teilweise auch regelmäßigen Besetzungsstrukturen (etwa Bandmatrizen) besitzen. Bis auf wenige Ausnahmen, wie die Logarithmierung oder dem Kronecker-Delta im Baum-Welch-Verfahren, sind die Algorithmen insbesondere direkt durch low-level-Routinen der weitverbreiteten BLAS-Bibliotheken (vgl. Abschnitt 4.2) darstellbar.
Für eine Verallgemeinerung möglicher Anwendungsgebiete des Konzepts aus Kapitel 3 kann demnach eine weitreichende SQL-Abdeckung der drei BLAS-Level motiviert werden. Dies würde nicht nur die Verarbeitung der HMM-Methoden ermöglichen, sondern auch die Berechnung datenintensive wissenschaftlicher Methoden, wie dem PageRank aus Beispiel 3 oder die in Abschnitt 9.3 beschriebenen Frequenzspektrenanalysen (etwa von auditiven Signalen) mittels Fouriertransformationen.
Auffällig bei der Operatoranalyse der HMM-Algorithmen ist das Fehlen allgemeiner Matrizenmultiplikationen, welche in BLAS etwa in den Level-3-Routinen der Form

$$C = aA^{[T|H]}B^{[T|H]} + cC$$

mit optionaler Transponierung A^T bzw. Adjungierung A^H bereitgestellt werden. Die Abwesenheit kann grundlegend als nötig für dünn besetzte HMM-Probleme erachtet werden, da durch die Multiplikation zweier dünn besetzter Matrizen im Allgemeinen eine dicht besetzte Matrix entsteht. Aufgrund der klassischerweise hohen Dimensionen dünn besetzter Matrizen wären das Produkt zu groß um effizient handhabbar zu sein.
Trotzdem ist, aufgrund deren weiten Verbreitung, eine Betrachtung allgemeiner Matrizenprodukte unabdingbar für die Überführung von Operationen des wissenschaftlichen Rechnens in SQL-Datenbanken.
Aus diesem Grund werden im folgenden Kapitel zunächst effiziente Darstellungen von Vektoren und Matrizen in (objekt-)relationalen Datenbanksystemen untersucht und verglichen. Hierbei wird zwischen dünn und dicht besetzten Matrizen unterschieden. Als initialer Schwerpunkt wird die in Tabelle 5.2 dargestellte Menge Linearer-Algebra-Operatoren der BLAS-Bibliotheken betrachtet. Die Menge überdeckt hierbei nicht die gesamte Funktionalität der Referenz-Bibliothek. Es fehlen beispielsweise Rotationen, Normen oder Lösung von linearen Gleichungssystemen

Tabelle 5.2: In Kapitel 7, 8 und 9 untersuchte fundamentale Lineare-Algebra-Operationen. Hierbei stellen a, b Skalare, A, B Matrizen und $\mathbf{x}, \mathbf{y}$ Vektoren dar.

Level-1		
Elementweise Vektor-Operationen	$a\mathbf{x} \star (b\mathbf{y})$	$(\star \in \{+, -, \odot, /\})$
Skalarprodukt		$\mathbf{x}^{[T\|H]}\mathbf{y}$
Level-2		
Matrix-Vektor-Produkt		$aA^{[T\|H]}\mathbf{x}$
Level-3		
Elementweise Matrix-Operationen	$aA \star (bB)$	$(\star \in \{+, -, \odot, /\})$
Matrix-Matrix-Produkt		$aA^{[T\|H]}B^{[T\|H]}$

$x = Tx$ mit Dreiecksmatrizen T. Ein Großteil der nicht explzit einbezogenen Operationen können jedoch (wenn auch nicht direkt optimiert) aus Kombinationen der gewählten Operatormenge umgesetzt werden. Ein einfaches Beispiel hierfür ist etwa die euklidische Norm $\| \cdot \|_2 = \sqrt{\langle \cdot, \cdot \rangle}$ (**sqrt**() ist erstmalig offiziell in SQL:2003 eingeführt worden [113]). Aus den künftigen Betrachtungen zur Datenrepräsentation von Matrizen aus Kapitel 6 sind ebenfalls die Umsetzung von Rotationen einfach ableitbar (vgl. etwa die Givens- und Householder-Rotationen in Anhang B.6). Das Ziel der Arbeit ist hier jedoch zunächst nur, neben der Überführung der HMM-Methoden, grundlegende Betrachtungen zu Vektor- und Matrizenoperationen in SQL durchzuführen, um eine weitreichende Funktionalität des präsentierten Frameworks zu erreichen. Hierdurch können beispielsweise weitere frequent genutzte Methoden zur Aktivitätserkennung und -vorhersage, wie die bereits erwähnte Fouriertransformationen (vgl. Abschnitt 9.3) oder die Hauptkomponentenanalyse (vgl. Anhang B.6) umgesetzt werden.

Algorithmus 12 Numerisch stabilerer Baum-Welch-Algorithmus zur Schätzung optimaler Parameter A, B, π

Input: $\mathbf{o} = \{o_0, \ldots, o_T\}$ ▷ Observationssequenz
Startparameter wählen: A, B, π
for $k = 1, \ldots,$Max_Iteration **do**
  $\mathfrak{S}_1 = \|\tilde{\alpha}_{\mathbf{1}}\|_1$
  $\alpha_1 = \tilde{\alpha}_{\mathbf{1}}/\mathfrak{S}_1$
  for $t = 1, \ldots, T-1$ **do** ▷ Iteration
    $\tilde{\alpha}_{\mathbf{t+1}} := (\alpha_t^T A) \odot B_{:,o_{t+1}}$
    $\mathfrak{S}_{t+1} = \|\tilde{\alpha}_{\mathbf{t+1}}\|_1$
    $\alpha_{t+1} = \tilde{\alpha}_{\mathbf{t+1}}/\mathfrak{S}_{t+1}$
  end for

  $\beta_T = \mathbf{1}/\mathfrak{S}_T$ ▷ Backward-Variablen
  for $t = T-1, \ldots, 1$ **do**
    $\beta_t = A(B_{:,t+1} \odot \beta_{t+1})/\mathfrak{S}_t$
  end for

  for $t = 1, \ldots, T$ **do** ▷ Gamma-Variablen
    $\gamma_t = \alpha_t \odot \beta_t / \langle \alpha_t, \beta_t \rangle$
  end for

  for $t = 1, \ldots, T-1$ **do** ▷ Xi-Variablen
    $\xi_t = A \odot (\alpha_t \cdot (\beta_{t+1} \odot B_{:,o_{t+1}})^T)/(\alpha_t^T A(B_{:,o_{t+1}} \odot \beta_{t+1}))$
  end for

  $\overline{\pi} = \gamma_1$ ▷ Update
  $\overline{A} = (\sum_{t=1}^{T-1} \xi_t[i,j])/((\sum_{t=1}^{T-1} \gamma_t) \cdot \mathbf{1}^T)$ ▷ Elementweise Division /
  for $j = 1, \ldots, M$ **do**
    $(\overline{B})_{:,j} = (\sum_{t=1}^{T} \delta_{o_t,V_j} \gamma_t)/((\sum_{t=1}^{T} \gamma_t))$ ▷ Elementweise Division /
  end for

  if Abbruchbedingung **then**
    return $\overline{A}, \overline{B}, \overline{\pi}$
  else
    $A = \overline{A}, B = \overline{B}, \pi = \overline{\pi}$
  end if
end for

Algorithmus 13 Numerisch stabilerer Viterbi-Algorithmus.

Input: $\mathbf{o} = \{o_0, \ldots, o_T\}, \lambda, I = \{1, \ldots, N\}$
$\delta_1 = \log(\pi) + \log(B_{:,o_1})$ ▷ Elementweise Logarithmisierung
$\psi_1 = \mathbf{0}$
for $t = 2, \ldots, T$ **do** ▷ Rekursion
    $\psi_t = [\text{argmax}((\delta_{t-1} \cdot \mathbf{1}^T) + \log(A))]^T$ ▷ Spaltenweise-Maximierung argmax
    $\delta_t = [\max((\delta_{t-1} \cdot \mathbf{1}^T) + \log(A))]^T + \log(B_{:,o_t})$ ▷ Spaltenweise-Maximierung max
end for

$\log(P^*) = \max_{i \in I}(\delta_T)$ ▷ Initialisierung Output
$q_T^* = \text{argmax}_{i \in I}(\delta_T)$

for $t = T - 1, \ldots, 1$ **do** ▷ Output
    $q_t^* = \psi_{t+1}(q_{t+1}^*)$
end for

Kapitel 6

Datenrepräsentation

Nach der Etablierung von Methoden und Operatoren, die in (parallelen) relationalen Datenbanksystem umgesetzt werden sollen, ist vor dem eigentlichen Übersetzungsprozess zunächst eine Diskussion über die Architektur und die Datenrepräsentation von Matrizen und Vektoren in SQL-Systemen nötig.

Für die Untersuchung möglicher Architekturen beziehen wir uns zum einen auf das zentrale SQL-basierte Konzept aus Kapitel 3 und zum anderen auf systemspezifische Datenbankerweiterungen, wie sie in Abschnitt 4.4 vorgestellt worden sind. Für die Wahl geeigneter relationaler Systeme wird neben Datenbankfunktionalitäten und Erweiterungen insbesondere unterschieden, inwiefern die Relationen intern attributweise (column stores) oder tupelweise (row store) gespeichert werden.

Nach der Etablierung potenzieller Architekturen werden effektive Relationenschemata für Matrizen und Vektoren untersucht. Eine adäquate Wahl der Repräsentation bildet die Basis für eine optimale Ausnutzung von Datenbankfunktionalitäten und ist Grundlage für die Aufstellung und Abarbeitung effizienter Anfragepläne im Datenbanksystem. Aus diesem Grund ist es von fundamentaler Bedeutung, für weiterführende Betrachtungen eine detaillierte Analyse potenzieller Relationenschemata im Kontext von Hidden-Markov-Modellen beziehungsweise allgemeinem wissenschaftlichen Rechnen vorzunehmen. Dafür werden zunächst Speicherschemata aus etablierten Lineare-Algebra- und darauf aufbauenden Scientific-Computing-Bibliotheken vorgestellt und deren Überführbarkeit in relationale SQL-Systemen diskutiert.

Erweiterte Konzepte wie die aktive Speicherung und Verarbeitung von Matrizen in komprimierter Form, wie es beispielsweise im Kontext von Apache SystemML in [114] vorgestellt wird, ist nicht Bestand der folgenden Untersuchungen. Die effiziente Komprimierung (etwa Lauflängenkompression) spezieller Matrizen kann jedoch bei geeigneter Struktur und Relationenschemawahl transparent durch das Datenbanksystem geschehen. Ein Beispiel in einem anderen Kontext ist hierfür die in Abschnitt 9.4 vorgestellte Zeitreihenverwaltung von Automotive-Daten. In diesem

Fall ist die transparente Datenkompression ausschlaggebend für die Umsetzbarkeit (Datenvolumen und effiziente Anfrageverarbeitung) der Datenbanklösung. Der dort diskutierte Kern der Relationenschemata wird in den folgenden Betrachtungen berücksichtigt.
Im Allgemeinen wird sich zeigen, dass die Überführung in relationale oder auch objekt-relationale Ansätze nicht eindeutig ist und die Performance dieser innerhalb verschiedener Datenbanksysteme variieren kann.

6.1 Architektur

Zur Umsetzung eines Frameworks für transparente Datenbankunterstützung im Kontext der Aktivitätsvorhersage und -erkennung in Big-Data-Assistenzsystemen wird zunächst eine Diskussion über Komponenten, Schnittstellen und nötige Funktionalitäten der Architektur geführt. Hierfür seien zunächst die wesentlichen Punkte der ursprünglichen Anforderungsanalyse des wissenschaftlichen Konzepts aus Abschnitt 3.1 in Kürze zusammengefasst:

- **Datenbankinterne Analysen und Filterung:** Wissenschaftliche Analysen und Verfahren zur Aktivitätserkennung und -vorhersage (etwa Machine Learning) sollen innerhalb (paralleler) relationaler Datenbanken auf bereits homogenisierten Daten angewandt werden. Neben der transparenten Parallelisierung und optimierten Anfrageverarbeitung durch Datenbankfunktionalitäten, wirkt sich die Minimierung von Angriffspunkten insbesondere positiv auf die Fehleranfälligkeit des Software-Stacks und auf den Schutz sensibler Daten aus.

- **SQL-basierte Implementierung:** Die Verarbeitung der Analysen soll mittels standardisierter SQL-Statements umgesetzt werden. Aufgrund der lückenhaften Verbreitung modernerer Standards ist eine Umsetzung mittels SQL-92 zu bevorzugen, um eine weitgehende Systemunabhängigkeit und Implementationslanglebigkeit garantieren zu können.

- **Intra-Operator-Parallelisierung von SQL-Anfragen:** Um die Verarbeitung von Big-Data-Anfragen zu ermöglichen, ist eine parallele Verarbeitung der Analysen nötig. Aufgrund immenser Datenmengen wird insbesondere die Parallelisierung einzelner Operatoren des wissenschaftlichen Rechnens benötigt.

- **Schnittstelle für Restoperatoren und Post-Processing:** Selbst bei vollständiger Abdeckung aller Operationen, im Sinne von Fließ- und Fixkommaoperationen, durch das Datenbanksystem benötigen Entwickler von Assistenzsystemen oder Data Scientists im Allgemeinen zusätzliche Funktionalitäten. So ist etwa die visuelle Darstellung von Ergebnissen für viele Anwendungen von zentraler Bedeutung und muss im Framework ermöglicht werden.

- **Data Provenance im Kontext von Machine Learning:** Um aus der Entwicklungsphase Rückschlüsse auf ergebnisbestimmende Sensoren, Zeiträume oder Teilergebnisse zu ziehen, sollten Techniken des Provenance Management oder der Dimensionsreduktion unterstützt werden können.

Mit diesen Punkten als Leitlinie werden im Folgenden verschiedene Architekturen mit transparenter Datenbankunterstützung diskutiert. Die Nutzung nicht-paralleler relationaler Systeme kann für verschiedene Anwendungsszenarien ebenfalls positiv sein und wird daher Bestandteil der hier vorgestellten Untersuchungen (insbesondere bei den Abhandlungen zum Übersetzungsprozess in Kapitel 7) sein. Für die strukturelle Betrachtung liegt der Fokus hier auf den parallelen Systemen. Die Umsetzung der Anforderungen kann hierbei in verschiedenen Variationen eines Basis-Frameworks geschehen. In Abbildung 6.1 ist dieses dargestellt. Im Gegenzug zu den darauf aufbauenden Varianten der Architektur, die im weiteren Verlauf diskutiert werden, erfüllt dieses alle vorher gesammelten Anforderungen. Prinzipiell werden neben einem (parallelen) relationalen Datenbanksystem eine oder mehrere Schnittstellen für die Aspekte

- der transparenten Anfrageerstellung,
- der eventuell extern koordinierten parallelen Anfragestrategien,
- dem Scheduling der Anfragen und
- der Verarbeitung von Restoperationen (etwa visuelle Darstellungen)

benötigt. Die Sinnhaftigkeit einer externen Komponente zur Koordinierung der Intraoperator-Parallelisierung durch Anfragezerlegung wird in Kapitel 8 diskutiert. Die Notwendigkeit dieser ist hierbei nicht intuitiv ersichtlich, da die Parallelisierung klassischerweise durch das parallele Datenbanksystem selbst, aus Gründen logischer und physischer Optimierungen und effizienterem Scheduling, geschehen sollte. Für nicht parallele Verarbeitungen ist diese (Teil-)Schnittstelle obsolet.
Noch vor der Parallelisierung einzelner Anfragen ist zuerst die Überführung der Methoden der Assistenzsystementwickler in ebensolche umzusetzen. Im optimalen Fall sollte die Formulierung der Probleme für Nutzer „natürlich“ sein, etwa durch einfache Formelsprachen oder weit verbreitete Programmiersprachen. Da solch eine Überführung nicht Fokus der vorliegenden Arbeit ist, werden im weiteren Verlauf nur die fundamentalen Operationen aus Abschnitt 5.4 im Sinne einer Funktionsbibliothek untersucht. Die entstandenen Anfragen sollen durch Schachtelung und Kombination als Basis für allgemeine Verarbeitungsansätze beliebiger Operationen des wissenschaftlichen Rechnens dienen. Dies beinhaltet insbesondere auch Verfahren zur Dimensionsreduktion.
Für die Kombination von Anfragen und deren Koordinierung wird eine weitere (Teil-)Schnittstelle benötigt. Wesentliche Aspekte von Anfragekompositionen werden in Abschnitt 7.3 untersucht.

Abhängig von der Datenbankfunktionalität muss beim Scheduling insbesondere beachtet werden, dass Teilergebnisse der Datenbank mit der Weiterverarbeitung auf externen Schnittstellen (etwa zur Visualisierung) koordiniert werden müssen. Wie im weiteren Verlauf näher beschrieben, steigt der Scheduling-Aufwand insbesondere, wenn aus Performance-Gründen die Verarbeitung rechenintensiver Methoden auf externe Scientific-Computing-Bibliotheken ausgelagert werden (vgl. Architektur aus Abbildung 6.2, sowie Abschnitt 7.2 für eine Diskussion über die Effizienz der Matrizenmultiplikation in SQL).
Entscheidend für den Aspekt der Effizienz solcher Verfahren ist hierbei insbesondere die Wahl eines geeigneten relationalen Datenbanksystems. Ein geeignetes paralleles System (vgl. Abschnitt 2.3.5) ermöglicht eine gleichmäßige parallele Verarbeitung von „einfach strukturierten“ Anfragen (etwa parallele Aggregationen oder Verbünde) auf meist homogenen, lokal verbundenen Rechenknoten. Prinzipiell sind auch verteilte oder Multidatenbanksysteme einsetzbar. In diesen Fällen wird das Scheduling deutlich komplexer, da die Lastbalancierung variierende Hardware und Netzleistung berücksichtigen muss. Eine Anfragezerlegung und Koordination der Kommunikation von Zwischenergebnissen muss im Falle von Multidatenbanksystemen zudem im Vorhinein geschehen. Aufgrund des hiermit verbundenen großen Aufwandes konzentrieren wir uns in den weiteren Kapiteln auf zentrale und parallele relationale Datenbanksysteme. Im parallelen Fall gehen wir von einer Shared-Nothing-Architektur aus. Neben der Komposition verschiedener Datenbankknoten ist auch die Effizienz der lokalen Systeme und deren Funktionalität von maßgeblicher Bedeutung. Eine detailliertere Diskussion hierzu wird im Folgeabschnitt 6.2 geführt.
Wie sich im weiteren Verlauf zeigen wird, existieren verschiedene Methoden des wissenschaftlichen Rechnens und Operationen der linearen Algebra, in denen reine SQL-Lösungen vergleichsweise ineffizient umsetzbar sind. Ist die Laufzeit der zugehörigen Anwendungen von kritischer Bedeutung, sind verschiedene Anpassungen des Basisframeworks denkbar. Ansatzpunkte hierbei können entweder die Auslagerung der nicht-performanten Operationen in dafür entwickelte Bibliotheken (wie etwa BLAS/LAPACK-Implementationen [vergleiche Abschnitt 4.2] oder darauf aufsetzende Software, wie R [57,115] oder Matlab [58]) oder auch die Einbeziehung weiterer spezieller nicht-standardkonformer Funktionalitäten einzelner Datenbanksysteme sein. Für letzteres können etwa die Erweiterungen aus Abschnitt 4.4 als Referenz genutzt werden. Damit ergeben sich insgesamt die folgenden drei wesentlichen, einfachen Architekturansätze:

1. Vollständige SQL-Implementation (Abbildung 6.1)
2. Auslagerung von ineffizienten Methoden (Abbildung 6.2)
3. Im Datenbanksystem integrierte Erweiterungen (Abbildung 6.3)

Die Architektur der vollständigen SQL-Implementation aus Abbildung 6.1 entspricht der Basis-Architektur und überführt alle Methoden des Machine Learnings und des wissenschaftlichen

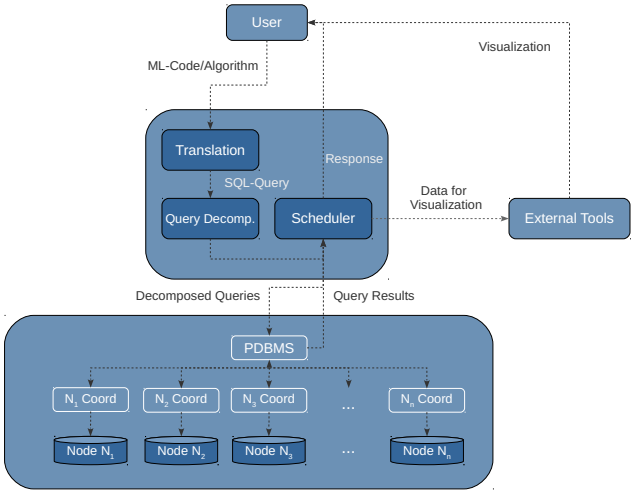


Abbildung 6.1: Architektur zur datenbankbasierten Verarbeitung von Machine Learning und wissenschaftlichem Rechnen im Big-Data-Bereich. In dieser Architektur verarbeitet das Datenbanksystem die gesamten Verfahren in Form von SQL-Anfragen. Visualisierungen und ähnliches werden ausgelagert.

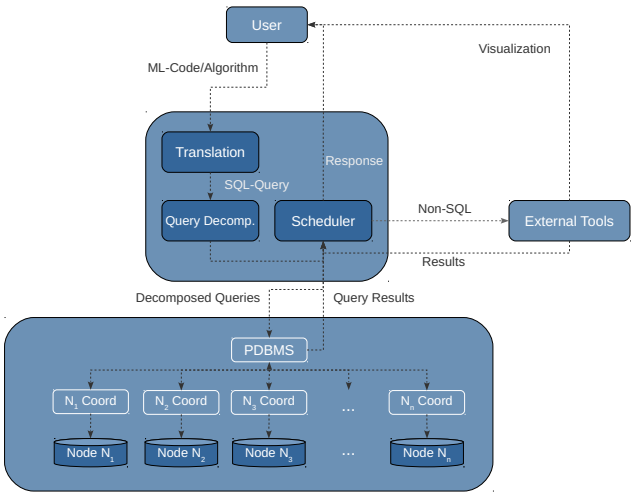


Abbildung 6.2: Architektur zur datenbankbasierten Verarbeitung von Machine Learning und wissenschaftlichem Rechnen im Big-Data-Bereich. In dieser Architektur verarbeitet das Datenbanksystem nur jene Operationen, welche hinreichend effizient in diesem berechnet werden können. Die restlichen Operationen, sowie Visualisierungen und ähnliches werden ausgelagert und mit Software, die beispielsweise auf Lineare-Algebra-Bibliotheken (LAPACK/BLAS) aufbaut, berechnet.

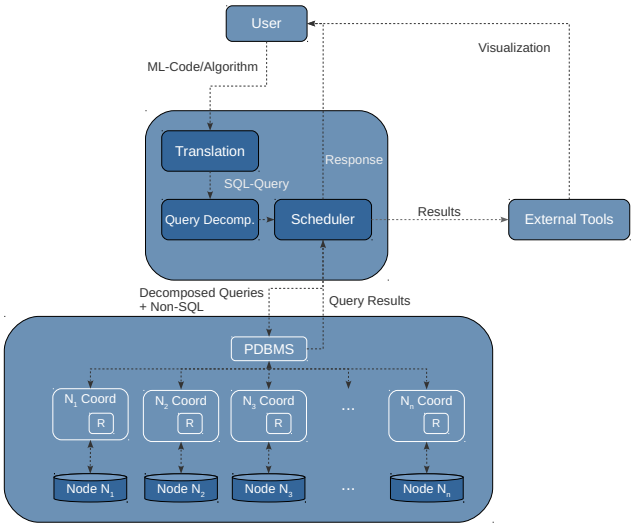


Abbildung 6.3: Architektur zur Berechnung von Machine Learning und wissenschaftlichem Rechnen im Big-Data-Bereich mittels integrierter Bibliotheken für lineare Algebra Operationen (LAPACK oder aufsetzende Systeme wie R [hier]) oder systemspezifischen Erweiterungen des SQL-Dialekts (etwa Matrix-Datentypen mit zugehöriger Anfragelogik oder durch UDFs) für die Verarbeitung ineffizienter SQL-Algorithmenanteile. Visualisierungen und ähnliches werden hierbei ausgelagert.

Rechnens in SQL-Anfragen (mit Fokus auf SQL-92). Hierfür wird in einer zu entwickelnden Schnittstelle die Überführung der Verfahren, mögliche Anfragezerlegungen zur Beschleunigung der Berechnung und das koordinierte Senden dieser an das parallele Datenbanksystem umgesetzt. Mögliche Visualisierungen werden extern, beziehungsweise lokal auf dem Client ausgeführt. Hierbei sollte sichergestellt werden, dass nur Ergebnisse visualisiert werden, die keinen direkten Rückschluss auf sensible Daten oder eine Verletzung von Datenprivatheitsaspekten ermöglicht. Die wesentlichen Vorteile dieses Ansatzes sind

- die potenzielle Langlebigkeit der Implementation durch eine weit verbreitete langlebige standardisierte Anfragesprache,
- eine transparente Verarbeitung von Hauptspeicher-übersteigenden Daten (bei geeigneter RDBS-Wahl),
- transparente Parallelisierung und mögliche funktionale Erweiterungen durch Einbinden zusätzlicher Schnittstellen zur Manipulation der SQL-Anfragen (etwa zur Wahrung der Datenprivatheit [19, 21])
- und die Einhaltung von Aspekten des Datenschutzes (Zugriffsbeschränkung und Minimie-

rung von Angriffspunkten ausserhalb der Datenbank).

Wie bereits beschrieben, kann sich hierbei die Performance einzelner Methoden nachteilig auf die Nutzbarkeit der Architektur auswirken. Andererseits kann der hohe Fokus auf Sicherheit und der möglichen Wahrung der Privatsphäre ein wesentlicher Aspekt für das Zustandekommen von Projekten und der Akquise von Daten sein. So können sich diese Punkte wohl möglich positiv auf die Bereitschaft von Personen auswirken, persönliche Daten für die Trainingsphase von Assistenzsystemen bereitzustellen.

Die zweite Architektur ermöglicht bessere Performance durch die externe Berechnung kritischer wissenschaftlicher Methoden mittels hierfür speziell optimierter Bibliotheken. Dies entspricht etwa dem klassischen Ansatz der Client-Server-Datenbankanwendungsprogrammierung. Durch die Nutzung ebenfalls standardisierter Referenzimplementationen (BLAS/LAPACK aus Abschnitt 4.2) kann in diesem Ansatz (zumindest teilweise) die Langlebigkeit und Systemunabhängigkeit gewährt werden. Die Implementation ist im Allgemeinen komplexer und fehleranfälliger, da Zwischenergebnisse der externen Bibliotheken und des Datenbanksystems koordiniert und kommuniziert werden müssen. Zudem sind (parallele) Bibliotheken des wissenschaftlichen Rechnens klassischerweise (lokal) hauptspeicherorientiert und teilweise auf sehr leistungsstarke (und teure) Hardware ausgelegt. Neben dem zusätzlichen Aufwand kann es hierdurch aus Kostengründen zur Einschränkung der Umsetzbarkeit kommen. Neben diesem Aspekt werden durch die teilweise Auslagerung der Berechnungen Vorteile der ersten Architektur, wie die potenzielle Wahrung von Privatheitsaspekten oder des Datenschutzes, beeinträchtigt. Die erhöhte Datenkommunikation und deren Überführung in die jeweilige interne Datenrepräsentationsform ist in diesem Fall zusätzlich zu beachten und kann im schlimmsten Fall teurer sein als die gewonnene Performance der eingesetzten Bibliotheken. Ein Beispiel für einen negativen Einfluss dieser ist in Abschnitt 9.4 im Kontext eines Szenarios von Automotive-Daten beschrieben.

Um die Kommunikation der (sensiblen) Daten zu meiden, bei gleichzeitiger Beschleunigung der Performance, werden in der dritten Architektur aus Abbildung 6.3 kritische Methoden durch zusätzliche Funktionalitäten innerhalb des Datenbanksystems berechnet. Dies spart zum einen teure Kommunikation und interne Datenanpassung ein, und ermöglicht zudem, dass sensible Daten nicht die Datenbank verlassen müssen. Trotz deren Standardisierung können hier auch User-Defined-Functions (UDFs) für SQL einbezogen werden, da diese etwa die Formulierung imperativer Funktionen ermöglichen und in der SQL-Schnittstelle verfügbar machen. Aufgrund ihrer spärlichen (standardkonformen) Verbreitung kann deren Nutzung zum aktuellen Zeitpunkt (Stand: 12/2022) als systemspezifisch interpretiert werden. Analog hierzu erweitern andere Ansätze dieser Kategorie (wie in Abschnitt 4.4 diskutiert) die Anfrageformulierung in SQL durch zusätzliche Scientific-Computing-Funktionalitäten. Hierbei werden entweder neue Datentypen für Matrizen und Vektoren mit zugehörigen Funktionalitäten in das Datenbanksys-

tem implementiert oder etablierte Umgebungen des wissenschaftlichen Rechnens [1] in das RDBS (oder umgekehrt) eingebettet und so deren Funktionalität für Nutzer ermöglicht. Aufgrund der systemspezifischen Schnittstellen und Erweiterungen büßt dieser Ansatz vor allem die Systemunabhängigkeit (und damit deren Austauschbarkeit), sowie eine Implementationslanglebigkeit und die Möglichkeit der Anbindung zusätzlicher Funktionalitäten durch SQL-Schnittstellen ein. Da einige der Ansätze aus dem Stand der Technik und Forschung nicht für parallele Systeme und klassische relationale Systeme entwickelt worden sind, wären in diesem Fall zusätzlich aufwändig zu implementierende Koordinierungsschnittstellen nötig. Dies wirkt sich offensichtlich negativ auf die Umsetzbarkeit bzw. die Entwicklungskosten dieser aus.
In der vorliegenden Arbeit liegt der Fokus maßgeblich auf der Untersuchung der ersten Architektur, der vollständigen Überführung in Standard-SQL, um eine weitreichende Untersuchung der Nutzbarkeit solcher Implementationen in bereits etablierten und verbreiteten Systemen zu evaluieren. Hierfür wird zunächst im Folgeabschnitt die Wahl geeigneter (paralleler) relationaler Datenbanksysteme diskutiert. Abschließend werden in diesem Kapitel adäquate Relationenschemata für Matrizen und Vektoren untersucht und ausgewertet.

6.2 Wahl des Datenbanksystems

Der wohl wesentlichste Vorteil des SQL-basierten Konzepts ist die Systemunabhängigkeit und die damit verbundene Austauschbarkeit der Datenbanksysteme und genutzter Rechenressourcen. Dies ist entscheidend, da die Wahl des (parallelen) Datenbanksystems maßgeblich die Effizienz des Frameworks bestimmt. Wie sich mehrfach im Laufe der Arbeit zeigen wird, können sich die Berechnungszeiten derselben Anfragen auf gleicher Hardware und verschiedenen relationalen Systemen bis zu mehreren Ordnungen unterscheiden. Speziell im Kontext sich weiterentwickelnder Hardware sind Anpassungen oder Neuentwicklungen von Datenbankinterna nötig. Aus diesem Grund ist die Austauschbarkeit der Systeme (oder deren Update) Basis für die Langlebigkeit des Frameworks. Da unterstützte SQL-Funktionalitäten im Allgemeinen nur erweitert werden, können Implementationen mit den über die Jahre und Jahrzehnte steigenden Hardwaremöglichkeiten (aus heutiger Sicht etwa SIMD-Ausnutzungen von CPU oder GPU) an Performance gewinnen, insofern relationale Datenbanksysteme diese ausnutzen können.
Da (hardware-orientierte) Implementationsdetails nicht im Fokus dieser Arbeit liegen, wird im folgenden Abschnitt nur eine kurze Diskussion über eine Auswahl hinreichender Kriterien zur Wahl geeigneter Datenbanksysteme aus heutiger Sicht (Stand 12/2022) geführt. Prinzipiell ist die Güte der Anfrageverarbeitung eines Systems nur schwer einsehbar für Nutzer. Dies hängt mit der Komplexität der Systeme und der fehlenden Offenlegung von Implementationsdetails zusammen. Neben der im Allgemeinen nur bedingt einsehbaren Anfrageverarbeitung (inklusive

[1] Wie etwa hier R [57, 115], angelehnt an die (hier umgekehrte) Einbettung von MonetDB in R [95]

der zugehörigen Optimierungsschnittstelle), werden hier drei wesentliche, einsehbare Kriterien angeführt:

- Die Ausnutzung moderner Hardware.
- Die Unterstützung SQL-92 übergreifender DB-Funktionalitäten (Indexstrukturen, SQL-Funktionen ab SQL:1999 oder nicht standardkonforme Erweiterungen).
- Die Unterscheidung zwischen zeilen- und spaltenorientierter Speicherung von Relationen.

Diese Kriterien beschreiben offensichtlich nicht einen vollständigen Katalog an hinreichenden Bedingungen, haben sich jedoch im Zuge der hier beschriebenen Arbeit als sinnvoll erwiesen. Neben diesen Punkten, die vor allem auf die Funktionalität und Berechnungseffizienz des Frameworks ausgelegt sind, existieren noch verschiedene andere Ansatzpunkte, die speziell im Kontext der industriellen Nutzung von Bedeutung sein können. Hierzu zählen beispielsweise finanzielle Kosten, welche sich etwa durch Lizenzkosten des Datenbanksystems oder der Anschaffung und Instandhaltung neuer Hardware ergeben. Konträr hierzu kann die Wahl des Systems auch an bereits vorhandene Hardware angepasst werden. Da die Szenarien stark variabel sind und schwer zu verallgemeinern, werden finanzielle Kosten im Folgenden vernachlässigt.

Datenbanksysteme auf moderner Hardware

Viele der (kommerziell) verbreiteten relationalen Systeme sind weit vor der Jahrtausendwende, teilweise mit Ursprung in den 1970er Jahre, entwickelt worden. Seit dieser Zeit haben sich neben den stetig steigenden Zahlen verbauter Transistoren und der Taktfrequenz von CPUs auch weitere Hardware-Konzepte etabliert, die sich positiv auf Berechnungsgeschwindigkeiten auswirken. Darunter zählen unter anderem moderne Speichergeräte wie SSDs, Multicore- und Manycore-Prozessoren oder externe (Ko-)Berechnungsinstanzen, wie beispielsweise GPUs. All diese Aspekte sind fester Bestandteil aktueller Forschung und Entwicklung im Datenbankbereich (vergleiche etwa das DFG-Schwerpunktprogramm SPP 2037: „Scalable Data Management for Future Hardware“ [116]). Einige der Aspekte, wie etwa Nutzung von Multicore-CPUs zur parallelen Berechnung von Teilanfragen, sind bereits in klassisch relationalen Systemen etabliert und können transparent zur Beschleunigung von Anfragen genutzt werden. Andere Punkte, wie die Nutzung von GPUs im Kontext von Datenbanksystemen, sind ein vergleichsweise neues Feld und aktiver Bestandteil zahlreicher Forschungsprojekte. Konzeptionell bieten die SIMD-Möglichkeiten von GPUs (und auch von CPUs) prinzipiell ein großes Potenzial für die einfachen Matrix-Vektorrechnungen, die in den folgenden Abschnitten und Kapiteln diskutiert werden. Die optimale Ausnutzung zur Kopplung von CPU und GPU ist hierbei nicht einfach umsetzbar, aufgrund vergleichsweise langsamer BUS-Anbindungen und begrenzter Speichergrößen in GPUs [117]. Aus diesem Grund sind Forschungen und Entwicklungen oftmals ausgelegt auf die

Unterstützung ausgewählter Teilprobleme, etwa von Verbundimplementationen [117] oder einfacher Hash-Aggregate [118]. Da bereits mehrere Prototypen und Erweiterungen zur Verfügung stehen, beispielsweise PG-Strom [119] für das hier mehrfach genutzte PostgreSQL (und Postgres-XL aus Abschnitt 4.3.3), sollte die Ausnutzung von GPUs potenzieller Forschungsbestandteil für künftige Betrachtungen sein und könnte sich eventuell in naher Zukunft als hinreichendes Kriterium für eine adäquate Wahl relationaler Datenbanksysteme herausstellen.

Funktionalität: SQL-Funktionen und Indexstrukturen

Im Gegensatz zur schwer einsehbaren Güte der Anfrageverarbeitung und -optimierung ist die bereitgestellte Funktionalität eines Datenbanksystems typischerweise gut dokumentiert. Da das hier untersuchte Konzept sich maßgeblich auf SQL-92 bezieht, ist die Nutzung modernerer standardisierter SQL-Funktionen optional und teilweise einschränkend im Sinne der Systemunabhängigkeit. Trotzdem kann die Einbeziehung dieser anwendungsbezogen sinnvoll sein. Vor allem iterative Verfahren können oftmals von der Nutzung von *Window*-Funktionen (ab SQL:2003 [113]) und rekursiven Anfragen (ab SQL:1999 [120]) profitieren. Ein Beispiel hierfür ist etwa die Implementation einer *Fast Fourier Transformation* (Radix-2-Verfahren), die in Abschnitt 9.3 des Evaluationskapitels diskutiert wird. Im Kontext einfacher statistischer Analysen kann zudem die Nutzung der, im Zuge der OLAP-Erweiterungen in SQL:2003 [113] eingeführten, Analyse-Aggregatfunktionen und einfachen skalaren Funktionen (etwa dem Logarithmus **log** oder der Wurzelberechnung `sqrt`) profitabel und teilweise entscheidend sein. Die wesentlichsten dieser elementweisen mathematischen Funktionen sind erst mit SQL:2003 vergleichsweise spät in den Standard eingeführt worden, gelten jedoch als Quasi-Standard in gängigen Systemen[2]. Damit ist deren Einbeziehung in das untersuchte Framework nur bedingt als Verletzung der SQL-92-Prämisse zu verstehen.
Als eine der initialen Betrachtungen des vorliegenden Forschungsprojektes wurden in [7] und [8] eine systemübergreifende Auswertung der Berechnung von linearen Regressionsparametern in SQL vorgenommen. Hierbei wurden Ansätze verglichen, die

- SQL-92-konforme Anfragepläne,
- Anfragen mittels Analyse-Aggregatfunktion aus SQL:2003 und
- SQL-Statements mit erweiterten Datenbankfunktionalitäten (in diesem Fall MonetDBs R-Integration (vgl. [95]))

nutzen. Es wurde hierfür ein Datensatz aus 4000 Tupel mit 666 Attributen gearbeitet, wobei die Parameter der linearen Regressionsgeraden bezüglich einem Zeitparameter und einem weiteren

[2]Originalzitat: „Implementation of mathematical functions is fairly standard across all RDBMS implementations in this book.“ [113]. Zu den angesprochenen Systemen zählen: Oracle, IBM DB2, Microsoft SQL Server, Sybase SQL, MySQL und PostgreSQL.

Attribut auf Teilfenstern berechnet worden sind. Hierfür wurden die 4000 Tupeln in 800 aufeinander folgende nicht überlappende Fenster à 5 Tupel (mittels Extraattribut) gruppiert. Die Berechnungen wurden für alle Systeme auf einer 2.8 GHz Octacore CPU mit 4 GB DDR2 Arbeitsspeicher und dem Betriebssystem CentOS 6.6 umgesetzt. Die Ergebnisse des Experiments sind in Abbildung 6.4 dargestellt. Für die genutzten SQL-Anfragen sei auf die angegebene Literatur verwiesen. Aus der Abbildung lassen sich mehrere Schlüsse ziehen. Zum einen ist zu erkennen, dass die Datenbanksysteme trotz gleicher Anfragen und gleicher Hardware sich in deren Berechnungsgeschwindigkeit deutlich unterscheiden. MonetDB zeigt in diesem Szenario die besten Laufzeiten. Ein wesentlicher Einflussfaktor ist hierbei die interne Speicherstruktur (MonetDB ist der einzige betrachtete column store), welche im folgenden Unterabschnitt näher beleuchtet wird. Zusätzlich zeigt sich, dass die Einbindung von R-Funktionalitäten sich sehr positiv auswirkt und, wie bereits in der Architekturdiskussion beschrieben, eine Möglichkeit ist, wissenschaftliches Rechnen zu beschleunigen. Abschließend zeigt sich ein klarer Performance-Vorteil für die Analyse-Aggregatfunktion aus SQL:2003 im Vergleich zum SQL-92-Ansatz. Problematisch hierbei kann jedoch die transparente Implementierung dieser sein. So wurde in [121] in einer systemübergreifenden Studie zur Berechnung von Varianzen in modernen Datenbanksystemen gezeigt, dass manche Systeme numerisch instabile Algorithmen zur Berechnung der besagten Aggregation verwenden. Aus diesem Grund wurde von der Nutzung dieser in den betroffenen Systemen abgeraten. Die Güte der numerischen Stabilität ist speziell im Kontext von Big Data entscheidend und durch die Datenbankhersteller zu sichern. Wie in diesem Fall zu sehen ist, kann sich dies mitunter als Vorteil, aber auch als Nachteil herausstellen. Im Zuge des vorgestellten Experiments wurde in [8] aus diesen Gründen zusätzlich eine numerisch stabilere SQL-92-Umsetzung der linearen Regression mittels QR-Faktorisierung diskutiert. Abseits von den hier durchgeführten Untersuchungen zur linearen Regression wurde von der Nutzung analytischer Aggregationsfunktion im weiteren Verlauf abgesehen.

Neben den Funktionen des SQL-Kerns, die die Anfrageformulierung direkt beeinflussen, existieren Datenbankfunktionalitäten, die keinen direkten Einfluss auf die Form von Anfragen besitzen, jedoch deren Laufzeit deutlich verbessern können. Es handelt sich hierbei um Indexstrukturen (vgl. Abschnitt 4.1.1), dessen fehlende Unterstützung ein Ausschlussargument für die Nutzbarkeit eines Datenbanksystems sein kann. So wurde in einer im Zuge dieser Arbeit geführten Studie zur Verwaltung und Analyse von Automotive-Zeitreihen in Datenbanksystemen (vgl. Abschnitt 9.4) auf die Untersuchung eines andererseits vielversprechenden parallelen Datenbanksystems aus Performance-Gründen verzichtet. Grund hierfür war die fehlende Verfügbarkeit der in diesem Fall essenziellen B-Baum-Indexstrukturen [13]. Dies verdeutlicht die Signifikanz von Indexstrukturen bei der Wahl eines geeigneten Datenbanksystems und für generelle Untersuchungen zur Laufzeitbeschleunigung von Anfrageplänen. Eine detaillierte Auswertung verschiedener gängiger Indexstrukturen im Kontext der lineare-Algebra-Operationen und der HMM-Methoden aus Ab-

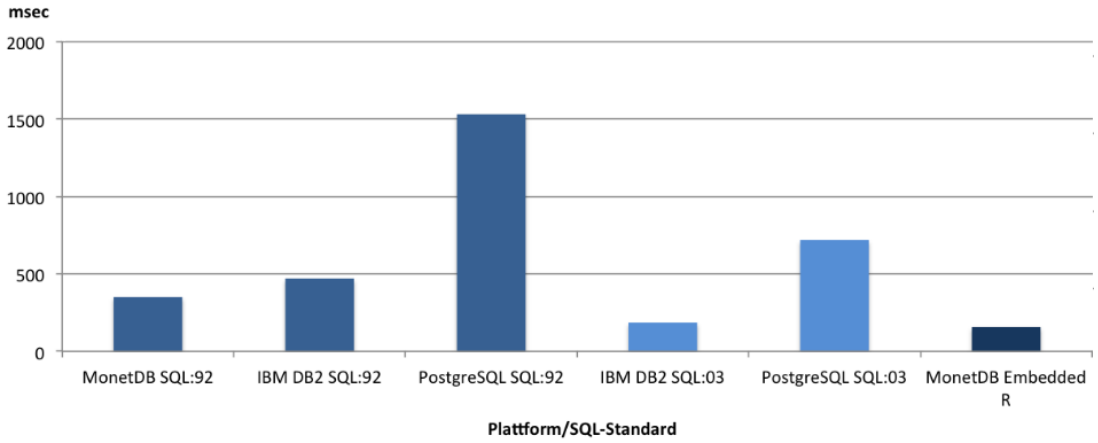


Abbildung 6.4: Laufzeitvergleich zur Berechnung linearer Regressionskoeffizienten auf verschiedenen Datenbanksystemen (PostgreSQL 9.4.1, IBM DB2 v10.1.0.0 und MonetDB v1.7). Die Darstellung ist ein Auschnitt einer Grafik aus [8].

Zeitstempel	Sensor A	Sensor B	Label
1	2	-1.7	A
2	2	-1.8	A
3	2	-1.9	A
4	3	-2.1	B

Abbildung 6.5: Beispielrelation für die Unterscheidung zwischen row stores und column stores.

schnitt 5.4 und 5.2 wird in Abschnitt 7.4 geführt.

Zeilen- und spaltenorientierte Datenbanken

Wie in den Ausführungen zum Lineare-Regression-Experiment (aus Abbildung 6.4) bereits motiviert wurde, ist die interne Speicherstruktur ein entscheidender Aspekt für die effiziente Verarbeitung datenintensiver Analysen. Im Kontext relationaler Datenbanksysteme werden im wesentlichen hierbei zeilen- und spaltenorientierte Speicherschemata, sogenannte *row stores* und *column stores*, unterschieden. Zur Verdeutlichung sei die Beispielrelation aus Abbildung 6.5 betrachtet. Die interne Hinterlegung dieser in Datenbankseiten ist für row und column stores in Abbildung 6.6 visualisiert.

In dieser ist erkennbar, dass column stores Attributwerte sequenziell hinterlegen, welches insbesondere für Aggregationen vorteilhaft sein kann. Zudem besitzen solche Systeme ein hohes Potenzial für Lauflängenkompression, welche speziell im Fall sortierter Daten (vgl. etwa die Sortierung durch Indexstrukturen in Actian Vector in Abschnitt 7.4) vergleichsweise einfach intern umsetzbar ist. Konträr hierzu ist dies in row stores aufgrund der tupelweisen Speicherung deut-

...	1	2	-1.7	A	2	2	-1.8	A
3	2	-1.9	A	4	3	-2.1	B	...

...	1	2	3	4	2	2	2	3
-1.7	-1.8	-1.9	-2.1	A	A	A	B	...

Abbildung 6.6: Konzeptuelle Darstellung der Datenbankseiten der Beispielrelation aus Abbildung 6.5 in einem row store [oben] und einem column store [unten]. In column stores werden zudem Attribute oftmals getrennt gespeichert (siehe etwa MonetDB [122]), so dass in diesem Falle vier Datenbankseiten genutzt werden würden.

lich schwieriger. Aus diesem Grund sind column stores vor allem für read-only Analyse-Anfragen auf komprimierbaren Daten prinzipiell besser geeignet. In [123] konnte dies im Zuge von Analysen (im Data-Warehouse-Kontext) experimentell belegt werden. Hierbei wurden verschiedene Techniken wie reine Indexanfragen in row stores getestet, um eine Komprimierung durch das Datenbanksystem und damit eine mögliche Anfragebeschleunigung zu erreichen. Die Ergebnisse der row stores blieben jedoch vergleichsweise weit hinter denen der column stores.
Im Kontext von Assistenzsystemen sind hierbei mehrere Vorteile von der Nutzung von column stores zu erwarten. Beispielsweise enthalten Testdaten, die mit hoher Abtastrate aufgenommen wurden, oftmals Daten, die stark komprimierbar sind. In Abschnitt 9.4 wird dies in einem ähnlichen Kontext für Zeitreihenanalysen von Automobildaten gezeigt. In diesem Fall wurden dieselben strikt relationalen Tabellen in column stores deutlich platzsparender gespeichert und schneller verarbeitet als in row stores. Allerdings zeigten sich objektrelationale Ansätze (vergleiche die folgenden Abschnitte zur Schemadiskussion) in diesem Kontext als noch effizienter. Neben der Zeitreihenverwaltung zeigt sich etwa in der Diskussion zu Relationenschemata für Matrizen aus Abschnitt 6.4, dass in column stores ein großes Potenzial zur Kompression von Zeilen- und Spaltenidentifizierungsattributen vorliegt. Ferner wird sich dort zeigen, dass die vergleichsweise kostenintensiven Anfragen der Matrix-Vektor- und Matrix-Matrix-Multiplikation (als Vertreter der etablierten Lineare-Algebra-Operationen aus Abschnitt 5.4) in den getesteten column stores schneller als in den row stores verarbeitet werden. Dies lässt sich wahrscheinlich auf den zugehörigen Anfragekern aus gruppierten Aggregationen und deren effizienteren Verarbeitung in column stores zurückführen.

6.3 Matrizendarstellungen in Lineare-Algebra-Bibliotheken

In diesem Abschnitt werden verschiedene Datenrepräsentationen für Matrizen aus etablierten Lineare-Algebra-Bibliotheken und allgemeineren bzw. darauf aufsetzenden Bibliotheken des wis-

senschaftlichen Rechnens beschrieben. Basis für die Betrachtungen dieses Abschnitts sind hierbei die in Abschnitt 4.2 vorgestellten *(Sparse) Basic Linear Subprograms* (BLAS) [66, 124] und der *Intel Math Kernel Library* (IMKL) [55, 56], wobei letzterer unter anderem eine hardwareoptimierte Implementation von BLAS und LAPACK enthält. Weitere Informationen sind den allgemeinen Diskussionen zu Matrixspeicherschemata aus [125] und [126] entnommen. In all diesen Quellen wird zwischen dicht und dünn besetzten Matrizen unterschieden. Die Schemata letzterer sind hierbei oft angepasst an spezielle Strukturen und einer begrenzten Menge an auszuführenden Operationen, die sich aus den jeweiligen Anwendungsszenarien ergeben. Daher ist die Anzahl und die Varianz der Struktur dünn besetzter Matrixschemata deutlich größer als die des dicht besetzten Falls. Beispielsweise werden in [125] 15 gängige dünn besetzte Schemata diskutiert, jedoch nur ein einzelnes für dicht besetzte Matrizen.

Zur Veranschaulichung von Speicherstrategien werden in den kommenden Ausführungen eine dicht besetzte Matrix A und Vektor $\mathbf{x}$

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n} \qquad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \tag{6.1}$$

und eine dünn besetzte Matrix

$$B = \begin{pmatrix} b_{11} & b_{12} & 0 & 0 \\ 0 & b_{22} & 0 & 0 \\ 0 & b_{32} & b_{33} & 0 \\ b_{41} & 0 & 0 & b_{44} \end{pmatrix} \tag{6.2}$$

als Beispiele genutzt. Die Matrixeinträge sind hierbei in **double precision** (8 byte) abgespeichert, können jedoch prinzipiell durch andere primitive numerische Datentypen repräsentiert werden.

Im Falle komplexer Matrix- und Vektoreinträge können diese in den Schemata durch die Aufteilung in Real- und Imaginärwerte gemäß des Isomorphismus'

$$\begin{aligned} f &: \mathbb{C} \mapsto \mathbb{R}^2 \\ f(x) &= \mathfrak{Re}(x) + i\mathfrak{Im}(x) \end{aligned}$$

durch zwei Attributwerte dargestellt werden. Dies wurde etwa zur Berechnung diskreter Fourier-Transformationen in Abschnitt 9.3 genutzt. Alternativ können die in SQL:1999 [120] eingeführten *structured user-defined types* genutzt werden, um ein komplexen Datentyp aus zwei passenden

primitiven numerischen Variablen eines Datentyps zu erstellen. Letzteres ist insbesondere bei der Nutzung von Array-Datentypen sinnvoll, um Real- und Imaginärteil eines Eintrags geclustert zu speichern. Im Sinne der SQL-92-Kompatibilität und der damit zusammenhängenden Systemunabhängigkeit ist jedoch die Aufteilung in zwei Attribute zu bevorzugen. Aufgrund dieser vergleichsweise einfachen Anpassung wird in den folgenden Ausführungen zu Gunsten der Übersichtlichkeit der Fall komplexer Matrizen und Vektoren vernachlässigt.
Da es für die Betrachtungen in dieser Arbeit nicht das Ziel ist, eine detaillierte und umfassende Überführung von Spezialschemata zu diskutieren, beziehen wir uns im Sinne der Funktionalität nur auf drei allgemeine Speicherschemata für dicht und dünn besetzte Probleme. Von diesen sind alle in (Sparse-)BLAS unterstützt. Die Überführung spezieller Schemata, wie beispielsweise dicht besetzte Bandmatrizen oder Diagonalmatrizen, ist analog zu den folgenden Ansätzen möglich und kann in zukünftigen Untersuchungen für spezielle Operationen getestet werden.

Dicht besetzte Matrizen - „Full Storage“

Der Fall dicht besetzter Matrizen wird erwartungsgemäß durch die Speicherung in einfachen Arrays, getreu der Form der Matrix $A \in \mathbb{R}^{m \times n}$, umgesetzt. Dies wird entweder über ein zweidimensionales Array oder ein eindimensionales Array

```
double[][] A
double[] A
```

realisiert, in dem bei letzterem das Element a_{ij} durch eine Sichtfunktion der Form

$$k : \{1, \ldots, m\} \times \{1, \ldots, n\} \mapsto \{1, \ldots, m \times n\} \tag{6.3}$$

$$k(i, j) = (i - 1) \cdot n + j \quad \vee k(i, j) = (j - 1) \cdot m + i \tag{6.4}$$

angesprochen wird. Abhängig von der Implementation werden die Elemente zeilen- oder spaltenweise abgespeichert. Manche BLAS-Implementationen (etwa IMKL [55]) ermöglichen beide Formen, um sequenzielle Zugriffe für einzelne Operationen zu maximieren. Für Vektoren wird analog ein eindimensionales Array

```
double[] x
```

genutzt. Im Falle symmetrischer oder Dreiecksmatrizen (vgl. etwa diskrete Fourier-Transformationen aus Abschnitt 9.3) kann dieses Schema in BLAS genutzt werden unter beschränkter Nutzung einer Dreieckshälfte. Es existiert jedoch auch ein kompakteres und im Sinne der Speicherverwaltung effizienteres Format („Packed Storage“) im BLAS-Standard. Dieses wird jedoch anwendungsbedingt hier nicht weiterverfolgt.

Zeilenindex =	1	1	2	3	3	4	4
Spaltenindex =	1	2	2	2	3	1	4
Wert =	b_{11}	b_{12}	b_{22}	b_{32}	b_{33}	b_{41}	b_{44}

Abbildung 6.7: Beispielhafte Darstellung der Beispielmatrix B aus (6.2) im Coordinate-Schema.

Dünn besetzte Matrizen

Im Fall dünn besetzter Matrizen ist die Dimensionsgröße im Allgemeinen wesentlich größer als die dicht besetzter Probleme. Die Menge von Nicht-0-Elementen pro Zeile variiert mit den Anwendungsszenarien, ist jedoch im Vergleich zu den 0-Elementen verschwindend klein. Aus diesem Grund fokussieren zugehörige Speicherschemata vor allem auf die alleinige Speicherung von Nicht-0-Elementen (mit wenigen Ausnahmen, wie etwa Bandmatrixschemata, die auch 0-Elemente innerhalb der Bandbreite hinterlegen).
In diesem Abschnitt werden die weitverbreiteten *Compressed Sparse Row* (bzw. *Compressed Sparse Column*) Speicherschemata, sowie die *Coordinate*-Darstellung beschrieben. Im Gegensatz zum Großteil anderer verbreiteter Schemata nehmen diese Strategien keine besondere Besetzungsstruktur an und sind damit prinzipiell für die Speicherung jeder dünn besetzten Matrix geeignet. In Spezialfällen, in denen bestimmte Muster der Besetzungsstruktur effizient in einer speziellen Darstellung ausgenutzt werden können, sind die allgemeinen Ansätze jedoch meist im Datenzugriff und/oder im Speicheraufwand schlechter.

Das Coordinate-Schema

Das Coordinate-Schema ist die „simpelste und flexibelste Form zur Darstellung von Matrizen“[3] [56]. Das Schema besteht aus drei Arrays, welche jeweils den Zeilenindex, den Spaltenindex und den jeweiligen Matrixeintrag aller Nicht-0-Werte strukturiert auflisten. Für die Beispielmatrix B aus (6.2) ergibt sich hierfür etwa das Speicherschema aus Abbildung 6.7. Das Schema kann auch als Key-Value-Liste für Paare der Form (i, j, v) interpretiert werden, welches sich auf natürliche Weise in ein relationales Schema überführen lässt. Die Darstellungsform ist vor allem durch seine einfache Umsetzbarkeit sehr verbreitet [125]. Durch die explizite Auflistung der Zeilen- und Spaltenindizes für jedes Element entstehen im Allgemeinen bei geordneter Auflistung viele komprimierbare Folgen von Indizes, welches sich negativ auf die Speichereffizienz auswirkt. Um diesem entgegen zu wirken, werden in den folgenden beiden Schemata eine komprimierte Speicherstrategie der Zeilen- bzw. Spaltenindizes umgesetzt.

[3] Im Originalzitat: „The coordinate format is the most flexible and simplest format for the sparse matrix representation.“ [56]

Zeilenanfang =	1	3	4	6			
Spaltenindex =	1	2	2	2	3	1	4
Wert =	b_{11}	b_{12}	b_{22}	b_{32}	b_{33}	b_{41}	b_{44}

Abbildung 6.8: Beispielhafte Darstellung der Beispielmatrix B aus (6.2) im CSR-Schema.

Compressed Sparse Row/Column (CSR/CSC)

Das Compressed-Sparse-Row-Schema (CSR) (bzw. Compressed-Sparse-Column-Schema (CSC)) kann als speicheroptimiertes Coordinate-Schema angesehen werden. In diesem Fall werden ebenfalls 3 Arrays genutzt, jedoch besitzt eines der Arrays nur die Anzahl der Zeilen m (bzw. Spalten n), anstatt der Anzahl der Nicht-0-Einträge l. Analog zum Coordinate-Schema werden in einem der Arrays alle Nicht-0-Werte gespeichert, wobei in diesem Fall diese gruppiert nach der jeweiligen Zeile im CSR und Spalte im CSC sind. Im CSR werden im zweiten l-elementigen Array, ebenso wie in der Coordinate-Darstellung, die Spaltenindizes der Nicht-0-Werte gespeichert. Bei der Hinterlegung der Zeileninformationen wird in diesem Fall jedoch im k-ten Array-Eintrag die Position des ersten Eintrags der k-ten Zeile in den beiden anderen Arrays geführt. Da im Allgemeinen m oder n deutlich kleiner als l ist, wird der benötigte Speicherplatz wesentlich reduziert. Negativ wird in der Literatur der zusätzliche Elementzugriffsschritt, aufgrund der indirekten Adressierung, für skalare Operationen in Matrix-Vektor-Multiplikationen und andere Operationen beschrieben [126][4]. Wie sich zeigen wird, erschwert dieser indirekte Schritt, sowie die nötige Ordnung von Zeilenelementen, zudem die relationale Umsetzung. Für die Beispielmatrix aus (6.2) folgt demnach die Ablage der Matrix gemäß der Darstellung in Abbildung 6.8. Die aufsteigende Ordnung der Spaltenelemente ist in diesem Fall nicht zwingend notwendig, ermöglicht jedoch in mehreren verbreiteten Verfahren effizientere Implementationen [125]. Der spaltenorientierte CSC-Fall verhält sich analog. Es lässt sich ferner leicht überlegen, dass das CSC-Speicherschema einer Matrix A^T dem CSR-Schema von A entspricht. Für das Schema existieren auch Varianten mit 4 Arrays (etwa in IMKL [55]), bei dem ein weiteres Array zusätzlich das letzte Nicht-0-Zeilenelement (bzw. -Spaltenelement) referenziert.

6.4 Relationenschemata für Matrizen

Nachdem allgemeine Strategien zur Speicherung von Matrizen im Kontext imperativer Programmiersprachen vorgestellt worden sind, soll nun eine Überführung dieser in Relationenschemata

[4] Originalzitat: „On the other hand, they are not very efficient, needing an indirect addressing step for every single scalar operation in a matrix-vector product or preconditioner solve.“ [126]

zur Repräsentation in Datenbanken diskutiert werden. Da der SQL-Kern deklarativer Natur ist und maßgeblich auf einer mengenorientierten Relationenalgebra basiert, ist das Anforderungsprofil an die Darstellung nicht identisch mit dem klassischer Scientific-Computing-Bibliotheken. Die Effektivität dieser Schemata ist abhängig von verschiedenen Aspekten, wie beispielsweise

1. dem Datenformat:
    - dicht oder dünn besetzte Matrizen / Vektoren (oder allgemeiner: Tensoren)
    - gruppierte Zeitreihen (mit und ohne label)
    - ...
2. dem Speichersystem und der Funktionalität des Datenbanksystems:
    - row store / column store
    - Unterstützung von Array-Datentypen
    - Datenkompression (etwa Lauflängenkompression, Dictionary Encoding, ...)
    - ...
3. der Art der Anwendungen / Operationen:
    - hoch-selektive oder full-table-scan-Anfragen
    - read-only oder schreibintensive Anwendungen
    - ...
4. Kompatibilität mit SQL:
    - Eignung für klassische Datenbankoperationen (Verbund, gruppierte Aggregation, Indexstrukturen, ...)
    - Selektierbarkeit von Matrixelemente, Zeilen, Spalten, Submatrizen
    - ...

Neben diesen Punkten ist vor allem der Aspekt des sequenziellen Elementzugriffs, beziehungsweise der Minimierung nötiger Zugriffe auf Datenbankseiten, für ausgewählte Operationen ein entscheidender konzeptioneller Performance-Aspekt. Im Fokus stehen hierbei etwa effiziente Selektionen und Gruppierungen einzelner Zeilen und Spalten. Diese sollen in diesem Fall entweder durch die Konstruktion des Schemas oder durch eine potenzielles sortiertes Einfügen beziehungsweise einer Clusterung der jeweiligen Einträge in den physischen Datenbankseiten (vgl. sequential files und primäre Indexstrukturen aus Abschnitt 4.1.1 und deren Auswertung in Abschnitt 7.4) erreicht werden. Hierbei sollte jedoch gleichzeitig nicht die Menge der (möglichst kompakt anwendbaren) SQL-Funktionalitäten beeinträchtigt werden.

Im Folgenden werden Relationenschemata, basierend auf den im vorherigen Abschnitt etablierten Speicherstrategien, entwickelt, diskutiert und abschließend evaluiert. Zunächst werden hierfür Ableitungen des Full-Storage-Schemas dicht besetzter Matrizen diskutiert.

6.4.1 Dicht besetzte Matrizen

Zur Umsetzung der Full-Storage-Darstellung ist eine Diskussion der Überführung von Arrays in (strikt) relationale Schemata nötig. Hierfür existieren mehrere Möglichkeiten. Zum einen können Spalten zur Identifizierung der jeweiligen Dimension genutzt werden. Dies ist etwa angelehnt an das Coordinate-Schema, welches Attribute für Zeilen- und Spaltennummern nutzt. Alternativ kann die Tabellenstruktur von Relationen direkt der Matrix-Struktur (visuell) nachempfunden werden. Hierfür wird jede Spalte durch ein eigenes Attribut repräsentiert, sodass ein Tupel einer gesamte Matrixzeile entspricht. Werden objekt-relationale Ansätze einbezogen, können Array-Datentypen genutzt werden, welches etwa Speichern ganzer Zeilen oder Spalten ermöglicht. Im Folgenden werden diese Ansätze detailliert vorgestellt und diskutiert. Besondere Betrachtung findet hierbei die Unterscheidung zwischen *column stores* und *row stores*. Um die Kompaktheit zugehöriger SQL-Anfragen einzubeziehen, werden mögliche zugehörige Anfragen für die Matrix-Matrix-Multiplikation betrachtet. Diese wird aufgrund seiner, im Vergleich zu den weiteren fundamentalen Lineare-Algebra-Operationen aus Tabelle 5.2, komplexen Struktur als Vertreter gewählt. Abschließend werden die Anfragen zu Vergleichszwecken auf den relationalen Datenbanksystemen PostgreSQL (row store), Actian's Vector (column store) und MonetDB (column store) ausgewertet.

Spaltenattribut-Darstellung

Der wohl intuitivste Ansatz zur Speicherung von Matrizen in einer relationalen Datenbank ist diese konzeptuelle Übertragung der tabellarischen Form in das Relationenschema. Hierfür wird jede Spalte (oder auch Zeitreihe) durch ein eigenes Attribut repräsentiert. Zusätzlich wird hierbei zu Identifizierungs- und Selektionszwecken ein Zeilenidentifizierungsattribut `i` (Schlüsselattribut) benötigt. Das zugehörige Schema kann dann durch

```
A (
    i int PRIMARY KEY,
    c1 double precision NOT NULL,
    c2 double precision NOT NULL,
    ⋮
    cn double precision NOT NULL
)
```

...	1	a_{11}	a_{12}	...	a_{1n}	2	a_{21}	...

...	1	2	...	m	a_{11}	...	a_{m1}	a_{12}	...

Abbildung 6.9: Beispielhafte Darstellung der Speicherung des Spaltenattribut-Schemas in Datenbankseiten eines row stores[oben] und eines column stores [unten] in sortierter Reihenfolge.

beschrieben werden, wobei `c`i die i-te Spalte beschreibt. Die zugehörige Relation für A aus (6.1) besitzt die matrixähnliche Form

i	c1	c2	...	cn
1	a_{11}	a_{12}	...	a_{1n}
2	a_{21}	a_{22}	...	a_{2n}
⋮	⋮	⋮	⋮	⋮
m	a_{m1}	a_{m2}	...	a_{mn}

,

wobei eine nach `i` geordnete Speicherung angenommen wird. Die Sortierung ist insbesondere ein Mittel um eine implizit geclusterte Speicherung von Zeilen- oder Spaltenelemente zu erreichen, welche im Kontext der Selektion dieser zu einer Minimierung der Aufrufe von Datenbankseiten führt.
Wie in allen folgenden Schemata werden Nicht-Schlüsselattribute mit **NOT NULL** gekennzeichnet, da zum einen in vielen Datenbankensystemen dies zu optimierten Elementzugriffen führen kann (siehe etwa [5] für Actian Vector) und zum anderen die Nutzung von **NULL**-Matrixeinträgen im Kontext des wissenschaftlichen Rechnens rar ist.
Wie in Abbildung 6.9 dargestellt ist, hinterlegt dieses Schema (dem Namen entsprechend) sequenziell Matrixzeilen in einem row store und Matrixspalten in einem column store. Spalten sind in diesem Fall einfach in SQL selektierbar, welches insbesondere in den HMM-Verfahren aus Abschnitt 5.4 für die Beobachtungsmatrix von Bedeutung sein kann.
Die Darstellung einzelner Spalten über Attribute führt jedoch zu mehreren potenziellen Problemen. Beispielsweise können Nutzer in vielen Datenbanksystemen nur eine begrenzte Anzahl an Attributen pro Relation nutzen. So unterstützt etwa Actian Vector nur 1024 Attribute pro Relation[6] und PostgreSQL nur Schemata, die kleiner als eine Datenbankseite[7] (Standardwert: 8kB) sind (Stand 12/2022). Von den in dieser Arbeit verwendeten Systemen ist MonetDB das einzige System, welches keine Begrenzung besitzt[8]. Unabhängig von dieser Limitierung ist das

[5] `https://communities.actian.com/s/article/Vectorwise-Table-Structures` (Stand 12/2022))
[6] `https://communities.actian.com/s/article/Vectorwise-Limits`
[7] `https://www.postgresql.org/docs/current/storage-page-layout.html`
[8] `https://www.monetdb.org/content/column-store-features`

Schema durch die Modellierung der Spalten als Attribute unflexibel bezüglich der Berechnung der Basisoperationen aus Abschnitt 5.4. So kann beispielsweise selbst die Berechnung eines Skalarprodukts als Sonderfall einer Matrizenmultiplikation $vw = \sum_{i=1}^{n} v_{1,i} w_{i,1}$ mit $v \in \mathbb{R}^{1\times n}$ und $w \in \mathbb{R}^{n\times 1}$ nicht elegant in SQL gelöst werden, da der nötige Verbund der Matrixeinträge nicht über jeweils ein identifizierendes Attribut geschehen kann. In diesem Fall müssten Elemente von v einzeln selektiert und den jeweiligen Elementen von w zugeordnet werden:

```
select sum(vT.v*w.c1)
from w join (
   select 1, v.c1
   from v
   union all
   select 2, v.c2
      ⋮
   select n, v.cn
   from v ) vT (i,v) on vT.i=w.i
```

Zur Vermeidung dieses Problems kann die Transponierung von v genutzt werden, um Verbund über `vt.i=w.i` zu ermöglichen, welches die zugehörige Anfrage zu

```
select sum(vT.c1 * w.c1)
from vt join w on vT.i = w.i
```

vereinfacht. Dies begünstigt zwar die Darstellung in SQL, führt jedoch im schlimmsten Fall zur zeitgleichen Haltung von v und v^T, welches neben der Datenredundanz auch zu Dateninkonsistenzen führen kann. Ferner bleibt die Anfrage für Matrizenmultiplikation AB mit $A \in \mathbb{R}^{k\times m}, B \in \mathbb{R}^{m\times n}$ nicht kompakt und deren Erstellung aufwändig. Analog zum oberen Beispiel nimmt hier das Produkt die Form

```
select 1, sum(AT.c1*B.c1), sum(AT.c1*B.c2),...,sum(AT.c1*B.cn)
from AT join B on AT.i=B.i
union all
select 2, sum(AT.c2*B.c1), sum(AT.c2*B.c2),...,sum(AT.c2*B.cn)
from AT join B on AT.i=B.i
   ⋮
select k, sum(AT.ck*B.c1), sum(AT.ck*B.c2),...,sum(AT.ck*B.cn)
from AT join B on AT.i=B.i
```

...	1	a_{11}	a_{12}	...	a_{1n}	2	a_{21}	...

...	1	2	...	m	a_{11}	a_{12}	...	a_{1n}	a_{21}	...

Abbildung 6.10: Beispielhafte Darstellung der Speicherung von (6.1) in sortierter Reihenfolge im Zeilen-Array-Schemas (`Ar`) in den Datenbankseiten eines row stores[oben] und eines column stores [unten].

an und kann durch die Attributdarstellung nicht von Gruppierungsfunktionalitäten von SQL profitieren. Trotz dieser augenscheinlichen Einschränkung der Anfrageformulierung, wird sich in der Evaluation zeigen, dass das Schema in manchen Setups (speziell in dem row store PostgreSQL) die schnellste Matrizenmultiplikation (ohne Anfrageerstellung) bietet.

Im Allgemeinen lässt sich das Schema konzeptuell als ungeeignet für variierende Lineare-Algebra-Operationen bezeichnen. Andererseits kann es genutzt werden, um kleinere Gruppen von Zeitreihen mit gleichem Zeitraster effizient zu speichern und zu verarbeiten, etwa durch Nutzung verschiedener Datentypen für einzelne Spaltenattribute. So wurde in [9] gezeigt, wie annotierte Sensordaten in dieser Darstellung gehalten werden können und effizient zur Berechnung von Transitionsmatrizen von Hidden-Markov-Modellen, beziehungsweise Markov-Ketten, weiterverarbeitet werden.

Array-Darstellung

Um die begrenzte Selektierbarkeit des Spaltenattribut-Schemas zu entgegnen, ohne die implizite Clusterung von Zeilen oder Spalten aufzugeben, wird nun die Zusammenfassung der Spaltenattribute durch Arrays diskutiert. Das zugehörige Schema besitzt die Form

```
Ar (
   i int PRIMARY KEY,
   row_vals double precision ARRAY NOT NULL,
)
```

und kann analog für die spaltenweise Speicherung (hier als `Ac` bezeichnet) umgesetzt werden. Dieses Schema bildet die zweidimensionale Array-Struktur des Full-Storage-Schemas am ehesten direkt nach, da die Relation etwa als nummerierte Liste von Arrays interpretiert werden kann. Die Matrix aus (6.1) kann etwa im zeilenweisen Fall durch

i	row_vals
1	ARRAY $[a_{11}, a_{12}, \ldots, a_{1n}]$
2	ARRAY $[a_{21}, a_{22}, \ldots, a_{2n}]$
⋮	⋮
m	ARRAY $[a_{m1}, a_{m2}, \ldots, a_{mn}]$

dargestellt werden, wobei abermals die Tupel geordnet nach `i` vorliegen. Durch die Nutzung von Array-Datentypen und der damit verbundenen Nutzung objektrelationaler Funktionalitäten (ab SQL:1999) ist die universelle Anwendbarkeit dieses Schemas eingeschränkt (vgl. etwa die fehlende Unterstützung in MonetDB und Actian Vector). Wie in Abbildung 6.10 zu erkennen ist, wird in diesem Schema eine sequenzielle Speicherung von Zeilen (bzw. Spalten für `Ac`) in column stores und row stores gleichermaßen gegeben, selbst wenn Tupel nicht geordnet eingefügt werden.
Im aktuellen Stand der Technik ist dieser Ansatz nur bedingt sinnvoll umsetzbar, da neben der begrenzten Unterstützung von Array-Datentypen auch elementweise arithmetische Operationen oder Aggregationen direkt auf Arrays nicht im Standard vorgesehen sind. Infolgedessen ist es nötig, Arrays vor solchen Operationen in Relationen zu transformieren. Dies geschieht laut Standard (SQL:1999/SQL:2003) mittels des `unnest`-Operators in der `FROM`-Klausel [127]. So gibt etwa die Anfrage

```
select i, av.j as j, av.v as row_vals
from Ar, unnest(row_vals) av (v,j) with ordinality
```

mit der Relation

i	row_vals
1	ARRAY $[a_{11}, a_{12}, \ldots, a_{1n}]$
2	ARRAY $[a_{21}, a_{22}, \ldots, a_{2n}]$

die Ergebnisrelation

i	j	row_vals
1	1	a_{11}
1	2	a_{12}
⋮	⋮	⋮
1	n	a_{1n}
2	1	a_{21}
2	2	a_{22}
⋮	⋮	⋮
2	n	a_{2n}

zurück. Demnach werden Relationen mit Schemata der Grundform

```
Ar (i int, v double precision ARRAY NOT NULL)
```

temporär in normalisierte Relationen der Form

```
Art (i int, j int NOT NULL, v double precision NOT NULL )
```

überführt. Dieses entspricht dem Coordinate-Schema des folgenden Unterabschnitts. Im Gegensatz zu diesem ist es hier zusätzlich nicht möglich, mit Indexstrukturen auf Spaltenelemente direkt zu verweisen. Andererseits sind durch die Zusammenfassung von Zeilen/Spalten als logische Einheit die Verbundkosten, speziell im Fall großer Arrays, deutlich niedriger. In Abschnitt 9.4 konnte im Kontext der Zeitreihenverwaltung ein deutlicher Performance-Gewinn durch das besagte Kernschema im Vergleich zum folgenden Coordinate-Ansatz erzielt werden.
Der Fall der Matrizenmultiplikation kann prinzipiell, aufgrund einer fehlenden standardisierten Überführung einattributiger Relationen in Arrays [127], nicht konsistent in Standard-SQL gelöst werden. Es werden hierfür systemspezifische Lösungen benötigt. In PostgreSQL [128] kann die Matrizenmultiplikation etwa durch

```
select i, array_agg(v) as row
from (
   select i,j,sum(v) as v
   from (
      select at.i as i,b.i as j, unnest(at.col)*unnest(b.col) as v
      from a2 at, b2 b
   ) t
   group by i,j
   order by i,j
) tt
group by i;
```

umgesetzt werden, wobei angenommen wird, dass `a2` die transponierte Matrix A ist um die Anfrageeffizienz zu erhöhen. Aus Konsistenzgründen wird hier das Ergebnis der Multiplikation in das gleiche Array-Schema wieder überführt, obwohl es bereits in der temporären Relation `tt` im Coordinate-Schema berechnet wurde. Die Operationen `array_agg` ist hierbei die Umkehrfunktion von `unnest` in PostgreSQL und überführt eine geordnete einattributige Relation in ein Array. Aus diesem Grund muss die temporäre Relation geordnet und gruppiert werden. In PostgreSQL kann der `unnest`-Operator ebenfalls in der `select`-Klausel genutzt werden, welches hier aus Gründen der Übersicht verfolgt wurde. Im Anhang B.2 wird zusätzlich die Erstellung und Nutzung einer User-Defined-Function (UDF) in PostgreSQL dargestellt, die das Skalarprodukt aus zwei Arrays direkt berechnet ohne das diese zunächst „entnestet“ werden müssen. Dieser Ansatz wird sich in der abschließenden Evaluation als performanter erweisen, ist aber aufgrund

der ebenfalls beschränkten (standardkonformen) Verfügbarkeit von UDFs in RDBS ungünstig im Sinne der Systemunabhängigkeit beziehungsweise Umsetzbarkeit.

Coordinate-Darstellung

Der dritte Ansatz zur Darstellung bezieht sich auf den Coordinate-Ansatz für dünn besetzte Matrizen aus Abschnitt 6.3 und entspricht dem wohl klassischsten strikt relationalen Ansatz zur Beschreibung von Matrizen. Zeilen- und Spaltennummer (`i` und `j`) werden als identifizierende Schlüsselmenge für den eigentlich Matrixeintrag interpretiert. Das zu dieser Key-Value-Darstellung zugehörige Relationenschema ist für Matrizen demnach von der Form

```
A (
   i int,
   j int,
   v double precision NOT NULL,
   PRIMARY KEY (i,j)
)
```

und analog für Vektoren von der Form

```
x (
   i int PRIMARY KEY,
   v double precision NOT NULL
).
```

Die Relation der Beispielmatrix aus (6.1) besitzt dann die folgende Gestalt:

i	j	v
1	1	a_{11}
1	2	a_{12}
⋮	⋮	⋮
1	n	a_{1n}
2	1	a_{21}
2	2	a_{22}
⋮	⋮	⋮
m	n	a_{mn}

Hierbei wird abermals von einer geordneten Einfügung der Tupel ausgegangen. Wie zu erkennen ist, erlaubt dieses Schema den sequenziellen Zugriff auf Zeilen und Spalten in column stores. In row stores ist eine sequenzielle Speicherung jedoch nicht möglich, welches sich insbesondere konzeptuell nachteilig auf Aggregation auswirkt. Im Kontext relationaler Datenbanksysteme birgt

...	1	1	a_{11}	1	2	a_{12}	1	...

...	1	1	...	1	2	...	m	1
1	...	1	2	...	n	a_{11}	a_{12}	...

Abbildung 6.11: Beispielhafte Darstellung der Speicherung (6.1) im Coordinate-Schema in sortierter Reihenfolge in Datenbankseiten eines row stores[oben] und eines column stores [unten].

dieses Schema weitere Vorteile. Zum einen ist es leicht erweiterbar auf höherdimensionale Tensoren durch das Hinzufügen weiterer Identifizerungsattribute der weiteren Dimensionen (vgl. etwa ξ_t in der SQL-Implementation des Baum-Welch-Verfahren in Anhang B.7). Durch die Key-Value-Struktur können schnelle Zugriffe auf einzelne Matrixeinträge oder Mengen von Matrixeinträgen durch Indexstrukturen umgesetzt werden. Dies wird in Abschnitt 7.4 diskutiert. Ferner ist die Darstellung strikt relational und ermöglicht eine „natürliche“ Selektionen von Submatrizen in SQL, welches wiederum kompakte Darstellungen von Matrixoperationen ermöglicht. Die Anfrage kann für die Matrix-Matrix-Multiplikation in SQL-92 (auch SQL-89 möglich) etwa durch die vergleichsweise simple Anfrage

```
select a.i,b.j,sum(a.v*b.v)
from a join b on a.j=b.i
group by a.i,b.j
```

umgesetzt werden. Die Struktur ermöglicht eine einfache und standardkonforme Schachtelung von Anfragen, welches effiziente Anfrageverarbeitungen komplexerer Methoden und Iterationen erlaubt. Dieser Aspekt wird in Abschnitt 7.3 näher diskutiert. Die kompakte Umsetzung der Lineare-Algebra-Operationen, mittels fundamentaler SQL-92-Operatoren, der klassisch relationale Aufbau und die Ermöglichung effizienter Indexstrukturen lassen ein großes Potenzial für universelle, systemübergreifende Implementationen von Verfahren des wissenschaftlichen Rechnens vermuten. Im Falle dicht besetzter Probleme ist der Einfluss der logischen Loslösung von Spalten oder Zeilen als strukturelle Einheit auf die Effizienz der Anfrageverarbeitung verbreiteter Operationen zu überprüfen. Andererseits erlaubt genau dies die Nutzung für dicht und dünn besetzte Matrizen (vgl. Folgeabschnitt).

Coordinate-Darstellung mit einem Attribut

Analog zur Variation der Full-Storage-Darstellung kann das Coordinate-Schema in eine Darstellung mit nur einem Identifizierungsattribut `i` umgesetzt werden. Das Schema ändert sich dann zu

```
A (
   i int PRIMARY KEY,
   v double precision NOT NULL
).
```

Die Relation der Beispielmatrix A aus (6.1) besitzt in diesem Fall die folgende Form:

i	v
1	a_{11}
2	a_{12}
⋮	⋮
n	a_{1n}
n+1	a_{21}
n+2	a_{22}
⋮	⋮
nm	a_{mn}

Wie in Abbildung 6.12 dargestellt, ermöglicht die Einsparung eines Attributs row stores die Haltung von mehreren Matrixeinträge pro Datenbankseite, welches den Datenzugriff prinzipiell verbessert. Es ergeben sich jedoch operationsabhängig höhere Kosten, da die Struktur der Verbundbedingungen teilweise komplexer ist. Zusätzlich ist die Anzahl von Zeilen und Spalten einer Matrix nicht implizit aus der Relation ablesbar und muss auf anderem Weg hinterlegt werden. Abhängig von den Methoden kann es hierbei sinnvoll sein, die Nummerierung (`i`) der Elemente geordnet nach Zeilen oder Spalten durchzuführen.
Mit der Zeilenanzahl `$m` und Spaltenanzahl `$n` kann die Matrizenmultiplikation in diesem Fall durch die Anfrage

```
select j+$n*i as i, v
from (
   select i, j, sum(v) as v
   from (
      select a.i/$m as i, b.i%$m as j, a.v*b.vs as v
      from a join b on a.i%$m=b.i/$m
   ) temp
   group by i,j
) temp2
```

umgesetzt werden, wobei die Relationen `a` und `b` zeilenweise geordnet sind. Um die nötigen Summen der einzelnen Zielmatrixeinträge zu bestimmen, muss hierbei über den jeweiligen Zeilen- und Spaltenindex gruppiert werden. Demnach wird abermals intern zur Berechnung des Matrizenprodukts das Problem auf das zweiattributige Coordinate-Schema zurückgeführt.

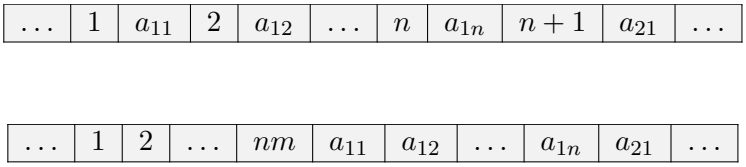


Abbildung 6.12: Beispielhafte Darstellung der Speicherung des einattributigen Coordinate-Schemas in sortierter Reihenfolge in Datenbankseiten eines row stores[oben] und eines column stores [unten].

Tabelle 6.1: Hardware-Spezifikation der Schema-Experimente für die dicht besetzte Matrizenmultiplikation.

Komponente	Wert
Prozessor	Intel Core i7-4600U CPU @ 2.10 GHz × 4
L1 Cache	32 KiB
L2 Cache	256 KiB
L3 Cache	4 MiB
RAM	12 GiB DDR3
Festplatte	TOSHIBA THNSFJ25 256 GB

Tabelle 6.2: Geänderte Parameter der PostgreSQL-Konfiguration.

Parameter	Wert
shared_buffers	3GB
temp_buffers	64MB
work_mem	1536MB
maintenance_work_mem	1535MB
max_worker_processes	8
max_parallel_workers_per_gather	8
effective_cache_size	9GB

Evaluation

Nach der Etablierung von Schemata und möglicher SQL-Anfragen für Matrix-Matrix-Multiplikation wird im Folgenden ein Experiment zur Auswertung dieser auf verschiedenen lokalen relationalen Datenbanksystemen präsentiert. Hierbei wird der open-source row store PostgreSQL 11.2, der open-source column store MonetDB-11.33.3, sowie der kommerzielle column store Actian Vector 5.1.0. getestet. Alle Systeme liefen auf einem Notebook mit der in Tabelle 6.1 beschriebenen Hardware und dem Betriebssystem Ubuntu 18.04.3 LTS. MonetDB und Actian Vector nutzen hierbei die Standardkonfiguration, während für PostgreSQL die Konfigurationsparameter aus Tabelle 6.2 angepasst worden sind.

Beide column stores unterstützen hierbei keine Array-Datentypen, so dass in diesen Fällen die Auswertung des Array-Schemas entfällt. Für letzteres wurde in PostgreSQL die vorgestellte SQL-Anfrage ausgewertet, so wie ein Ansatz, der eine in der Postgres-Sprache *plpgsql* implementierte UDF nutzt. Diese interpretiert zwei Arrays als Vektoren und berechnet ihr Skalarprodukt, so dass eine Entschachtelung der Arrays nicht nötig ist. Die UDF und deren Anfrage („psql2agg“ in Abbildungen 6.13 und 6.14) sind in Anhang B.2 dargestellt.

Um die kompaktere Variante der Anfragen für das Array- und das Spaltenattribut-Schema zu

ermöglichen, wurde hier das Produkt $C = A^T B$ berechnet, sodass ein Eintrag c_{ij} dem Skalarprodukt der $i-$ten Spalte von A und der $j-$ten Spalte von B entspricht. Die Tupel im Coordinate-Schema wurden für A und B in diesem Fall ebenfalls spaltenweise eingefügt, um prinzipiell dem Datenbanksystem eine optimierte Anfrageverarbeitung gemäß der Verbundbedingung zu ermöglichen. In Abbildung 6.13 und 6.14 sind jeweils die schnellsten Läufe aus 10 Versuchen für jede Kombination von Schema und Datenbanksystem dargestellt. Aufgrund der hohen Kosten der Anfrageerstellung wurde hierbei unterschieden zwischen der reinen Anfrageverarbeitungszeit und der Zeit, die Erstellung der Anfragen beinhaltet. Im Anhang B.2 wurden die Ergebnisse aus Abbildung 6.13 zusätzlich aus Gründen der Übersicht für jedes System getrennt dargestellt.

Aus den Ergebnissen sind verschiedene (auch nicht-intuitive) Erkenntnisse zu entnehmen. Zum einen ist erkennbar, dass das Spaltenattributschema (Schema 1) im Falle der column stores (wie intuitiv anzunehmen) ungünstig ist. Im Falle von PostgreSQL zeigte sich dieser Ansatz jedoch streckenweise sehr performant, insbesondere wenn die Anfrage bereits erstellt wurde.

Zeitgleich ist das einattributige Coordinate-Schema in PostgreSQL sehr langsam, speziell im Vergleich zum zweiattributigen Coordinate-Schema. Eine Untersuchung der Anfragepläne hat hierbei gezeigt, dass die einattributige Coordinate-Darstellung bereits bei niedrigen Dimensionen die Verbundberechnung (in diesem Fall Merge-Verbund; vgl. Abschnitt 4.1.2) auf die Festplatte auslagert, welches die Berechnungszeit stark erhöht. Der Grund hierfür ist, dass die Anfrageberechnung einen großen Bedarf an Speicherplatz aufweist. Im Falle der Dimension 300 umfasst dies beispielsweise um die 900MB Festplattenspeicher. Im Vergleich zu diesem ist der Unterschied zwischen ein- und zweiattributigen Coordinate-Schema in den column stores vergleichsweise klein und deren Reihenfolge sogar variierend. Eine detailliertere Diskussion über die Anfrageverarbeitung von Matrizenmultiplikationen ist in Abschnitt 7.2 vorgenommen.

Für das Array-Schema ist zum einen zu erkennen, dass die Nutzung von UDFs die Performance leicht verbessern kann, zum anderen aber im Falle der Matrizenmultiplikation vergleichsweise unperformant ist. Demnach können die hier vorgestellten objektrelationalen Ansätze nicht von der konzeptuell effizienteren Zusammenfassung von Zeilen oder Spalten profitieren.

Als Fazit lässt sich die geeignete Wahl von Schemata für dicht besetzte Probleme als schwierig und stark abhängig vom Datenbanksystem und deren Anfrageverarbeitung bezeichnen. Obwohl es eigentlich für dünn besetzte Probleme in Lineare-Algebra-Bibliotheken konstruiert wurde, kann aus den Ausführungen geschlossen werden, dass das zweiattributige Coordinate-Schema systemübergreifend als die geeignetste und stabilste Wahl für Lineare-Algebra-Operationen zu betrachten ist.

Eine Zusammenfassung der wesentlichen Vor- und Nachteile der Schemata ist in Tabelle 6.3 dargestellt.

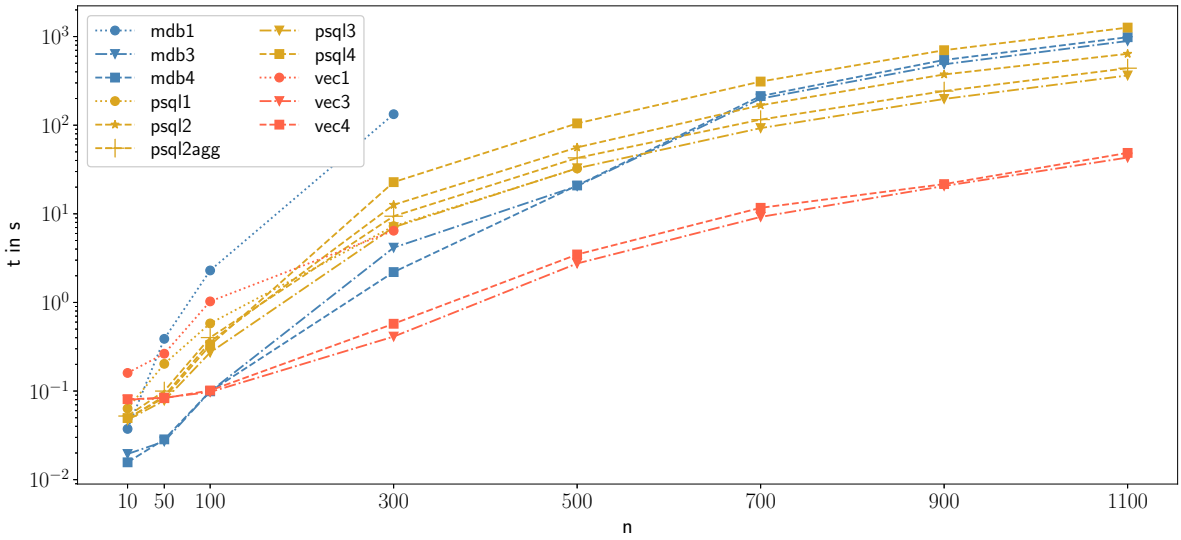


Abbildung 6.13: Experimentelle Auswertung (inklusive Anfrageerstellung) von Matrizenmultiplikationen $C = AB$ mit $A, B, C \in \mathbb{R}^{n \times n}$ mittels der verschiedenen Matrizenschemata (Spaltenattribut-1,Array-2, Coordinate(i,j)-3,Coordinate(i)-4) aus Abschnitt 6.4 in PostgreSQL (psql), MonetDB (mdb) und Actian Vector (vec).

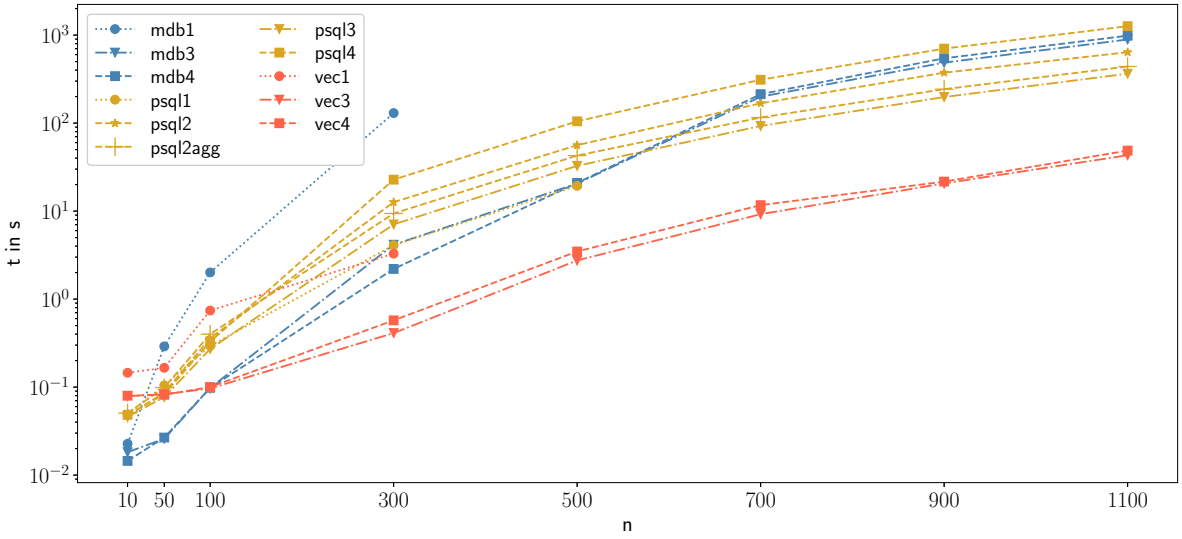


Abbildung 6.14: Experimentelle Auswertung der reinen Berechnungszeit (ohne Anfrageerstellung) von Matrizenmultiplikationen $C = AB$ mit $A, B, C \in \mathbb{R}^{n \times n}$ mittels der verschiedenen Matrizenschemata (Spaltenattribut-1,Array-2, Coordinate(i,j)-3,Coordinate(i)-4) aus Abschnitt 6.4 in PostgreSQL (psql), MonetDB (mdb) und Actian Vector (vec).

Tabelle 6.3: Zusammenfassung wesentlicher Aspekte der einzelnen Schemata für dicht besetzte Probleme. Die numerischen Werte zur Performance-Einordnung entsprechen der eingenommenen Position im Laufzeitvergleich für das jeweilige System.

	Spalten-Attribut	Array	Coordinate (i,j)	Coordinate (i)
Seq. Speicherung row store	Zeile	Zeile	Nein	Nein
Seq. Speicherung column store	Komplett	Komplett	Komplett	Komplett
Kompakte Anfragen	Nein	Mittel	Ja	Ja
SQL-92	Ja	Nein	Ja	Ja
Indexstrukturen auf bel. Elemente	Nein	Nein	Ja	Ja
Performance Postgres	1-4	3	1-2	2-4
Performance MonetDB	3	x	1-2	1-2
Performance Vector	3	x	1-2	1-2

6.4.2 Dünn besetzte Matrizen

In der Betrachtung dicht besetzter Probleme wurde bereits das Coordinate-Relationenschema, welches direkt an das eigentliche Coordinate-Schema aus Abschnitt 6.3 angelehnt ist, vorgestellt und wird daher nicht ausgiebig diskutiert. Die Nutzbarkeit für allgemeine dünn besetzte Probleme ist für das zweiattributige Schema hierbei bereits in seiner initialen Vorstellung motiviert. Dies bestärkt das große Potenzial des Schemas, da dieses für dicht und dünn besetzte Probleme eingesetzt werden kann und deren zugehörige SQL-Anfragen universell gültig sind. Die einattributige Darstellung ist durch die prinzipiell variable Anzahl an nicht-0-Zeileneinträgen nicht umsetzbar. In diesem Fall ist eine Abbildung des Identifizierungsattribut `i` auf die eigentlichen Zeilen und Spaltenindizes notwendig, welches ultimativ auf das zweiattributige Coordinate-Schema zurückführt.
Als letztes Basisschema wird die Compressed-Sparse-Row- beziehungsweise die Compressed-Sparse-Column-Darstellung auf deren Überführbarkeit in Relationenschemata untersucht. Diese werden anschließend mit dem Coordinate-Schema anhand eines dünn besetzten Matrix-Vektor-Produkt verglichen.

Compressed Sparse Row/Column (CSR/CSC)

Die ursprüngliche Darstellung des Compressed-Sparse-Row/Column-Schemas ist prinzipiell direkt durch

```
B (
   zeilenanfang int,
```

```
    j int NOT NULL,
    v double precision NOT NULL
)
```

überführbar. Dieses Schema ist jedoch ungünstig für weitere Verarbeitungen in SQL. So sind etwa im Allgemeinen die Attribute `zeilenanfang` und `j` keine Schlüsselattribute. Insbesondere nimmt `zeilenanfang` mehrere `NULL`-Werte an, welches die ursprünglich beabsichtigte Kompression der Zeilenindizes verhindert. Eine Nutzung von Indizes, die den nächsten Zeilenanfang beschreiben, hat zudem nur dann Bedeutung, wenn die Tupel geordnet und nummeriert werden. Dies kann etwa durch ein Extra-Attribut umgesetzt werden, welches mehr Speicherplatz als das Coordinate-Schema nutzt, oder in den Anfragen selbst mittels der Window-Funktion `row_number()`. Da die Anzahl von nicht-0-Elementen jedoch im Allgemeinen sehr groß ist, ist dieser Ansatz ebenfalls problematisch. Damit ist eine direkte Überführung der CSR/CSC-Darstellung nicht sinnvoll.
Die Kompression der Werte eines Attributs (etwa der Zeile) kann jedoch beispielsweise durch die folgenden objekt-relationalen Ansätze ermöglicht werden (insofern entsprechende Verfahren vom jeweiligen DBMS intern umgesetzt werden):

```
B (
    i int PRIMARY KEY,
    j int ARRAY,
    v double precision ARRAY
)
```

Zeilenkompression

oder

```
B (
    i int ARRAY,
    j int PRIMARY KEY,
    v double precision ARRAY
)
```

Spaltenkompression

In diesen werden die Zeilenindizes `i` pro Zeile beziehungsweise die Spaltenindizes `j` pro Spalte nur einmalig benötigt (und erlaubt). Diese Ansätze nutzen abermals Array-Datentypen, welche wie bereits beschrieben nur bedingt in relationalen Datenbanksystemen verbreitet sind. Dies mindert insbesondere die Systemunabhängigkeit der zugehörigen Implementation. Für die Umsetzung der Matrix-Vektor-Multiplikation $B\mathbf{w}$ ist insbesondere das Spaltenkompressions-Schema nützlich. So kann in diesem Fall das Produkt in PostgreSQL (mit der systemspezifischen **unnest**-Funktion in der **select**-Klausel) durch

```
select i, sum(v)
from (
    select unnest(b.i) as i, unnest(b.v)*w.v as v
    from b join w on b.j=w.i
) temp
group by i
```

umgesetzt werden. Die standardkonforme Umsetzung mittels **unnest** in der **from**-Klausel ist hierbei komplexer, da die beiden Arrays `i` und `v` über ihre Ordinalität verbunden werden müssen. Durch die Verminderung der Tupelanzahl und die Entschachtelung der Arrays nach dem Verbund bietet dieser Ansatz potenziell eine bessere Anfrageverarbeitung.

Evaluation

Im Folgenden wird die Auswertung eines Experiments für die dünn besetzte Matrix-Vektor-Multiplikation $\mathbf{y} = B\mathbf{w}$ bezüglich der diskutierten Relationenschemata vorgestellt. Hierfür wurde das gleiche Setup wie in der Evaluation der dicht besetzten Matrizenmultiplikation aus Abschnitt 6.4.1 (vgl. Tabellen 6.1 und 6.2) genutzt. Die Matrizen $B \in \mathbb{R}^{n \times n}$ besitzen hierbei 25 Elemente pro Zeile, die innerhalb einer Bandbreite von $n/4$ liegen[9]. Verglichen wurde das CSC-Relationenschema (Schema 1) mit der oben ausgeführten SQL-Anfrage, das CSR-Relationenschema (Schema 2) und das Coordinate-Schema (Schema 3). Die zugehörigen Anfragen der letzten beiden Ansätze sind im Anhang B.2 dargestellt. Dort wird gezeigt, dass während der Anfrage des zweiten Schemas zunächst temporär die Relation `b` in die Coordinate-Darstellung transformiert wird und dann identisch zu diesem weiterverarbeitet wird. Demnach ist zu erwarten, dass das CSR-Schema stets langsamer als das Coordinate-Schema ist. Wie in Abbildung 6.15 dargestellt ist, bestätigt sich dies für PostgreSQL. Für dieses System ist zudem zu beobachten, dass im Gegensatz zum Fall dicht-besetzter Matrixmultiplikationen objekt-relationale Ansätze sich positiv auf die Performance und den benötigten Speicher auswirken können. Das Coordinate-Schema ist hierbei konsistent geringfügig langsamer als die CSC-Variante. Generell skalieren alle Coordinate-Ansätze ähnlich, wobei Actian's Vector offensichtlich ein höheren Overhead für die Anfrageverarbeitung benötigt.
Eine Zusammenfassung der dünn besetzten Schemata ist in Tabelle 6.4 dargestellt.

Tabelle 6.4: Zusammenfassung wesentlicher Aspekte der einzelnen Schemata für dünn besetzte Probleme. Die numerischen Werte zur Performance-Einordnung entsprechen der eingenommenen Position im Laufzeitvergleich.

	CSC/CSR	Coordinate
Kompression	Ja	transparent in column stores
Kompakte Anfragen	Mittel	Ja
SQL-92	Nein	Ja
Indexstrukturen auf bel. Elemente	Nein	Ja
Performance Postgres	1	2

[9] D.h.: Wenn $b_{ij} \neq 0$ gilt, ist $\mathbf{max}(1, i - n/4) \leq j \leq \mathbf{min}(n, i + n/4)$.

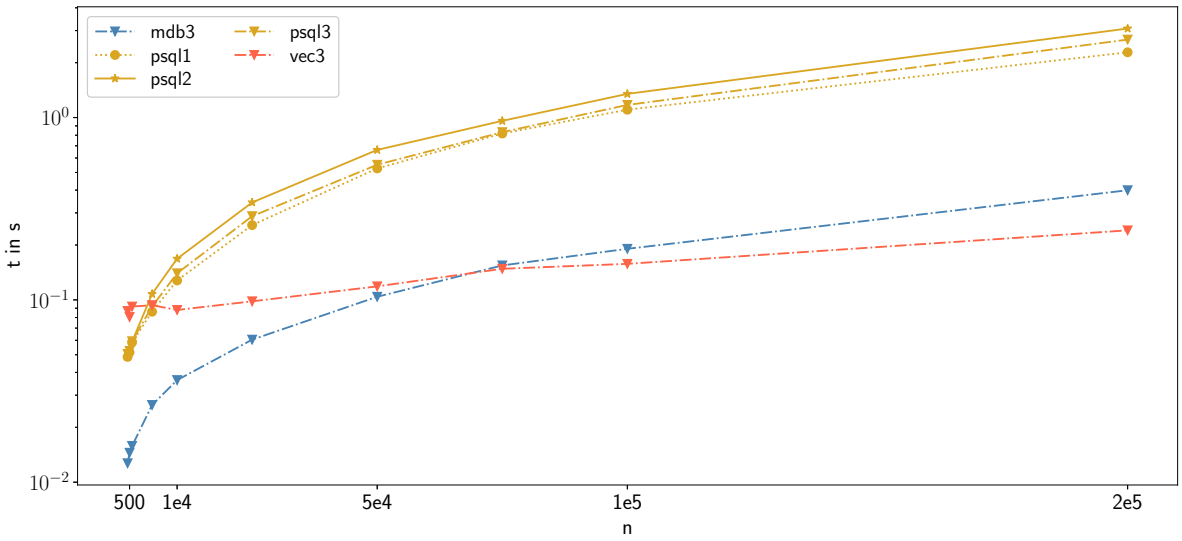


Abbildung 6.15: Experimentelle Auswertung von Matrix-Vektor-Multiplikationen $\mathbf{b} = B\mathbf{w}$ mit dünn besetzten Matrizen B der Dimension $n \times n$ mittels der verschiedenen Matrizenschemata (1-CSC, 2-CSR, 3-Coordinate) aus Abschnitt 6.4 in PostgreSQL, MonetDB und Actian Vector. Jede Zeile von B besitzt 25 Nichtnullelemente innerhalb einer Bandbreite von $n/4$.

6.4.3 Zusammenfassung und Diskussion

In diesem Abschnitt wurde gezeigt, dass vielfältige Relationenschemata für die Darstellung und Verarbeitung von Matrizen denkbar sind. Hierbei wurde gezeigt, dass, trotz eingeschränkter arithmetischer Möglichkeiten für Array-Datentypen auch objekt-relationale Darstellungen nützlich sein können. Keines der Schemata ist uneingeschränkt zu bevorzugen. Aufgrund der universellen Nutzbarkeit für dicht und dünn besetzte Probleme, der SQL-92-Konformität, der guten Selektierbarkeit von Submatrizen, möglicher Indexierung aller Matrizenelemente und der überwiegend positiven Performance wird für den Rest der Arbeit das Coordinate-Schema genutzt. Da von den fundamentalen Matrix-Operationen aus Tabelle 5.2 die kostenintensiven Matrix-Vektor- und Matrix-Matrix-Produkte vor allem gruppierte Aggregationen nutzen, sind column stores (wegen der potenziellen sequenziellen Speicherung) konzeptuell den row stores überlegen. Die in diesem Abschnitt vorgestellten Experimentalergebnisse bekräftigen diese These.

Im Vergleich zu den Umsetzungen in imperativen Sprachen wird speziell für dicht besetzte Matrizen stets mehr Speicherplatz benötigt. Dies wirkt sich direkt auf die Verarbeitungseffizienz und die Dimensionsspanne von Problemen, die hauptspeicherintern berechnet werden können, aus. Eine detailliertere Diskussion der Anfrageverarbeitung von Lineare-Algebra-Operatoren wird in

Abschnitt 7.2 vorgenommen.
Bei den Ausführungen zu dünn besetzten Matrizen wurde eine Diskussion von Schemata, die eine spezielle Besetzungsstruktur der jeweiligen Matrizen aufgreifen, ausgespart. Eine Betrachtung dieser könnte sich positiv auf verschiedene, häufig genutzte Methoden des wissenschaftlichen Rechnens auswirken. Deren Überführung in Relationenschemata ist hierbei abhängig von der jeweiligen Darstellung und dem Anwendungsfall. Beispielsweise ist die Speicherung von Tridiagonalmatrizen T (oder allgemeinen Bandmatrizen) prinzipiell durch das Schema

```
T (
    i int PRIMARY KEY,
    subdiag double precision,
    diag double precision NOT NULL,
    superdiag double precision
)
```

umsetzbar. Im Gegensatz zu den vorher diskutierten Schemata ist hier die Zulassung und Behandlung von **NULL**-Werten nötig, da Super- und Subdiagonalen ein Element weniger als die Diagonale halten. Die Darstellung ermöglicht schnellen Zugriff von Zeilenelementen der Matrix, welches sich beispielsweise bei Lösungsverfahren für lineare Gleichungssysteme $T\mathbf{x} = \mathbf{b}$ vorteilhaft auswirken kann. Hierbei ist insbesondere die iterative Natur dieser Verfahren im Kontext von SQL zu untersuchen. Dafür könnten potenziell Window-Funktionen oder rekursive Anfragen aufgegriffen und einbezogen werden.

Kapitel 7

Transformation in relationale Operatoren

Nach der Diskussion zur Darstellung von Matrizen und Vektoren in Kapitel 6 werden im Folgenden wesentliche Aspekte der Berechnung wissenschaftlicher Methoden in relationalen Datenbanksystemen untersucht. Hierbei werden zunächst die fundamentalen Lineare-Algebra-Operatoren aus Tabelle 5.2 aus Abschnitt 5.4 in SQL überführt. Zusätzlich werden hierbei Darstellungen der Operatoren in einer zunächst definierten, erweiterten Relationenalgebra entwickelt. Dies zeigt Verbindungsmöglichkeiten zu Anwendungen der Relationenalgebra, wie sie in Kapitel 3 näher beschrieben werden, auf. Zudem motiviert die Relationenalgebradarstellung eine Verarbeitung in relationalen Systemen mit eingeschränkter Funktionalität, wie sie etwa für frühzeitige Aggregationen nahe am Sensor genutzt werden können (vergleiche die vertikale Partitionierung in der Nutzungsphase des zugrundeliegenden PArADISE-Frameworks aus Abbildung 3.1 und Kapitel 3). Nach der Transformation wird die Effizienz der Verarbeitung der erstellten Anfragen in relationalen Systemen diskutiert und zur Einordnung mit der Laufleistung verbreiteter Scientific-Computing-Software für Probleme, die innerhalb des Hauptspeichers berechnet werden können, verglichen. Darauf folgend wird die geeignete Komposition einzelner Lineare-Algebra-Anfragen zu vollständigen Methoden diskutiert. Dies bezieht insbesondere iterative Verfahren ein. Abschließend wird die Nutzung von Indexstrukturen untersucht und experimentell ausgewertet. Letzteres wird sich insbesondere als essenziell für die Verarbeitung der HMM-Grundoperationen aus Abschnitt 5.4 zeigen.

7.1 Darstellbarkeit von Fundamentaloperatoren in relationalen Anfragesprachen

Im Folgenden werden die fundamentalen Operatoren aus Tabelle 5.2 in SQL-92 bezüglich des Coordinate-Relationenschemas aus Abschnitt 6.4 diskutiert. Hierbei wird zusätzlich eine Darstellung in einer erweiterten Relationenalgebra abgeleitet. Hierfür werden zunächst neue Operatoren der Relationenalgebra aus Abschnitt 2.3.3 hinzugefügt und deren ursprünglichen Definition auf Multimengen erweitert. Darauf folgend wird das neue System genutzt, um die eigentliche Übersetzung zu demonstrieren.

7.1.1 Erweiterung der Relationenalgebra

Da die Operatoren der in Abschnitt 2.3.3 definierten Relationenalgebra (

- Projektion $\pi_X(r)$,
- Selektion $\sigma_F(r)$ mit Bedingung F,
- Umbenennung $\beta_{B \leftarrow A}(r)$,
- Natürlicher Verbund $r_1 \bowtie r_2$,
- Vereinigung $r_1 \cup r_2$ und
- Differenz $r_1 - r_2$

) keine arithmetischen Berechnungen erlauben, ist eine Erweiterung der Operatormenge oder der Funktionalität einzelner Operatoren nötig. Wie sich zeigen wird, ist aufgrund der vergleichsweise kleinen und einfach strukturierten Menge der Lineare-Algebra-Operatoren nur eine vergleichsweise geringe Erweiterung erforderlich. Hierfür wird im folgenden Unterabschnitt der Gamma-Operator γ eingeführt, der für die Aggregation (in SQL-92 etwa die Summierung **sum**, der Durchschnitt `avg`, der minimale und maximale Attributwert **min**, sowie die Tupelanzahl **count**) über einem Attribut bezüglich der kombinierten Attributwerte einer zu spezifizierenden Attributmenge genutzt wird. Zusätzlich wird der Projektionsoperator π erweitert, um einfache (arithmetische) tupelweise Funktionen aus Attributen zu erstellen. Wir beziehen uns hier bei diesen Operatoren auf die viel zitierten Ausführungen in [48]. Dort werden die ursprünglichen und neu definierten Operatoren auf Multimengen, also unter Zulassung von Duplikaten, definiert. Dieser Ansatz wird insbesondere auch in SQL-Systemen aus Effizienzgründen umgesetzt. Beispielsweise müssen nach einer Projektion keine möglicherweise vorhandenen Duplikate eliminiert werden, welches demnach zusätzliche Scans temporärer Relationen einspart. Zur Umsetzung des Multimengenkonzepts ist eine leichte Anpassung der ursprünglichen Operatordefinitionen nötig. Da

diese jedoch im Folgenden keinen wesentlichen Einfluss besitzen und vergleichsweise einfach sind, sei für die Redefinition auf [48] verwiesen. Dort werden zudem weitere wesentliche Operatoren, wie der Sortierungs- und der Duplikateliminationsoperator, sowie der Operator des äußeren Verbunds, eingeführt. Da diese für die fundamentalen Operatoren nicht zwingend benötigt werden, werden diese im weiteren nicht näher beschrieben.
Für die bessere Darstellung des Gamma-Operators und der Erweiterung des Projektionsoperator sei die Beispielmatrix

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{7.1}$$

mit der zugehörigen Relation `a`

i	j	v
1	1	1
1	2	2
2	1	3
2	2	4

definiert.

Der Gruppierungsoperator γ nach [48]

Zur gruppierten Aggregation von Attributwerten einer Relation wird der Gamma-Operator der Form

$$\gamma_L(\mathrm{r})$$

eingeführt. Hierbei ist L eine Liste von Elementen, die zwei verschiedene Formen annehmen können. Entweder ist ein Element ein Attribut der Relation `r` und damit Teil der Attributliste nach der `r` gruppiert wird oder das Element ist ein Aggregationsoperator, der auf ein Attribut von `r` angewendet wird. Hierbei wird ein Pfeil zur Benennung der Aggregationsspalte analog zum β-Operator genutzt. Die Relation, die durch die Anwendung von γ_L auf `r` entsteht, wird hierbei wie folgt erstellt: Die Tupel von `r` werden in Gruppen partitioniert. Jede Gruppe besteht aus allen Tupeln die zu einer jeweilig existenten Wertkombination der Gruppierungsattribute gehört. Wenn keine Gruppierungsattribute existieren (hier gekennzeichnet durch $\emptyset$), wird die ganze Relation als eine Gruppe interpretiert. Für jede Gruppe wird genau ein Tupel erstellt. Dieses besteht aus

- den Attributwerten der Gruppierungsattribute der jeweiligen Gruppe, sowie
- den Aggregationswerten aller Gruppen-Tupel bezüglich der angegebenen Aggregationen aus L.

Für die Beispielrelation `a` der Matrix A aus (7.1) kann beispielsweise die zeilenweise Aufsummierung der Spalteneinträge durch

$\gamma_{\texttt{i},\textbf{sum}_{\texttt{(v)}\rightarrow\texttt{v}}(\texttt{a})} \Rightarrow$

i	v
1	3
2	7

dargestellt werden.

Der erweiterte Projektionsoperator nach [48]

Zur Umsetzung einfacher tupelweiser arithmetischer Rechenvorschriften bezüglich einer oder mehrerer Attribute wird im Folgenden der ursprüngliche Projektionsoperator

$$\pi_F(\mathrm{r})$$

erweitert. Die Liste F beinhaltet hierbei eine oder mehrere Ausdrücke. Jedes Element der Liste ist entweder

- ein Attribut aus `r`,
- ein Ausdruck `x`→`y`, wobei `x` und `y` Namen von Attributen sind und `x` ein Attribut aus `r` ist und in `y` benannt wird, oder
- ein Ausdruck $E \mapsto \texttt{z}$. Dabei ist E ein Ausdruck aus Attributen aus `r`, Konstanten, arithmetische- und String-Operatoren und `z` der Name für das zugehörige Ergebnisattribut.

Dies beinhaltet insbesondere auch die Möglichkeit zur Berechnung wissenschaftlicher unärer Operationen, wie etwa dem Logarithmus (vgl. Anpassungen zur numerischen Stabilität in Abschnitt 5.3) oder der Exponentialfunktion.
Als Beispiel für den erweiterten Projektionsoperator ist für die Relation `a` der Matrix A aus (7.1) die Berechnung

$$2 \cdot A + \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}$$

im Folgenden dargestellt:

$\pi_{\texttt{i,j,2*v+i}\rightarrow\texttt{v}(\texttt{a})} \Rightarrow$

i	j	v
1	1	3
1	2	5
2	1	8
2	2	10

.

7.1.2 Darstellung fundamentaler Operatoren

Mit den etablierten Erweiterungen der Relationenalgebra können nun die fundamentalen Operatoren der linearen Algebra in dieser und SQL(-92) dargestellt werden. Hierbei wird abermals eine Unterscheidung zwischen dünn und dicht besetzten Matrizen und Vektoren vorgenommen, wobei dieses aufgrund des universal einsetzbaren Coordinate-Relationenschemas vergleichsweise wenig Einfluss besitzt. Wegen der Struktur des Schemas ist es prinzipiell nötig, vor der Umsetzung fundamentaler, binärer Operatoren die Kompatibilität der Matrix- und Vektordimensionen zu testen. Dieser Aspekt wird im Folgenden vernachlässigt und kann etwa in der Matrixerstellungsphase realisiert werden. Zusätzlich wird für die Darstellung in SQL eine Umsetzung mit natürlichen Verbunden, wie es in der Relationenalgebra genutzt wird, vermieden. Es wird anstatt dessen der innere Verbund **join on** genutzt. Dies wird maßgeblich aufgrund der Lesbarkeit und Kompaktheit der Anfragen, aber zu Lasten der direkten Übereinstimmung von SQL- und Relationenalgebra-Anfragen, umgesetzt.

Elementweise Operationen $c\mathbf{x} \star (d\mathbf{y})$ und $cA \star (dB)$

Seien `x,y` und `a,b` Relationen, die stellvertretend für Vektoren $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ und Matrizen $A, B \in \mathbb{R}^{m \times n}$ stehen. Mit den Skalaren $c, d \in \mathbb{R}$ sollen die Operatoren

$$\begin{aligned} &c\mathbf{x} \star (d\mathbf{y}) && \star \in \{+, -, \cdot, /\} \\ &cA \star (dB) \end{aligned}$$

überführt werden. Sei zunächst angenommen, dass diese dicht besetzt sind und auch etwaige 0-Werte als Tupel `(i,j,0)` in der Relation hinterlegt werden. Aufgrund der Existenz von Tupeln für jeden Matrix- oder Vektoreintrag ist ein einzelner innerer Verbund genügend um die Matrix- und Vektoroperationen durchzuführen. Es ergibt sich etwa

```
select x.i as i, c*x.v⋆(d*y.v) as v
from x join y on x.i=y.i
```

$\pi_{\texttt{i,c*v⋆(d*w)}\to\texttt{v}}(\texttt{x} \bowtie \beta_{\texttt{w}\leftarrow\texttt{v}}(\texttt{y}))$

```
select a.i as i, a.j as j,
       c*a.v⋆(d*b.v) as v
from a join b on a.i=b.i and a.j=b.j
```

$\pi_{\texttt{i,j,c*v⋆(d*w)}\to\texttt{v}}(\texttt{a} \bowtie \beta_{\texttt{w}\leftarrow\texttt{v}}(\texttt{b}))$

Kritisch ist in diesem Szenario lediglich die mögliche elementweise Division durch 0, welche zu **NULL**-Werten in der Ergebnisrelation führt. Das Auftreten solcher Fälle sollte im Allgemeinen durch geeignete Berechnungsverfahren vermieden werden und wird daher hier nicht weiter verfolgt.

Der komplexere Fall ist der von dünn besetzten Vektoren und Matrizen. In diesem Fall werden Einträge, die 0 sind, nicht in den Relationen geführt, welches die Verbundkosten durch die verminderte Tupelanzahl verringert. Auf der anderen Seite sind die Anfragen zur Addition und Subtraktion nicht durch einen einfachen inneren Verbund umsetzbar, da mitunter nicht-0-Elemente konzeptuell mit 0-Elementen verbunden werden müssen. Für die Multiplikation ändert sich die ursprüngliche Anfrage, aufgrund der entfallenden Tupel in der Ergebnisrelation im Falle der Multiplikation mit einem 0-Eintrag, nicht. Analog ist der Fall der Division mit 0-Einträgen in der Matrix A und nicht-0-Einträgen in B. Der umgekehrte Fall wird ebenfalls vernachlässigt. Für die Addition und Subtraktion existieren in SQL-92 mehrere Konzepte, um die Fälle außerhalb eines inneren Verbundes einzubeziehen. Eines der kompaktesten ist die Nutzung eines vollen äußeren Verbundes in Kombination mit der SQL-92-Funktion **coalesce**, welche den ersten nicht-**NULL**-Wert einer Liste von Argumenten wiedergibt:

```
select coalesce(x.i,y.i) as i, coalesce(x.v,0.0)±coalesce(y.v,0.0) as v
from x outer join y on x.i=y.i

select coalesce(A.i,B.i) as i, coalesce(A.j,B.j) as j,
       coalesce(A.v,0.0)±coalesce(B.v,0.0) as v
from A outer join B on A.i=B.i and A.j=B.j
```

Der äußere Verbund wird in [48] als zusätzlicher Operator der erweiterten Relationenalgebra eingeführt und kann daher analog in dieser umgesetzt werden. Es ist jedoch auch möglich, die Subtraktion und Addition ohne diese und ohne die **coalesce**-Funktion umzusetzen. Hierfür wird eine Aggregation über der Vereinigung der mit den jeweiligen Skalaren multiplizierten Relationen genutzt, wobei im Falle der Subtraktion die Vorzeichen aller Werte umgekehrt werden müssen. Die entstehenden Gruppen enthalten demnach entweder 1 oder 2 Tupel. In der erweiterten Relationenalgebra führt dies für die Addition zu

$$\gamma_{\mathtt{i,j},\mathbf{sum}(\mathtt{v})\to\mathtt{v}}(\pi_{\mathtt{i,j,c*v}\to\mathtt{v}}(\mathtt{a})\cup\pi_{\mathtt{i,j,d*v}\to\mathtt{v}}(\mathtt{b})) \qquad \gamma_{\mathtt{i},\mathbf{sum}(\mathtt{v})\to\mathtt{v}}(\pi_{\mathtt{i,c*v}\to\mathtt{v}}(\mathtt{x})\cup\pi_{\mathtt{i,d*v}\to\mathtt{v}}(\mathtt{y}))$$

und für die Subtraktion zu

$$\gamma_{\mathtt{i,j},\mathbf{sum}(\mathtt{v})\to\mathtt{v}}(\pi_{\mathtt{i,j,c*v}\to\mathtt{v}}(\mathtt{a})\cup\pi_{\mathtt{i,j,-d*v}\to\mathtt{v}}(\mathtt{b})) \qquad \gamma_{\mathtt{i},\mathbf{sum}(\mathtt{v})\to\mathtt{v}}(\pi_{\mathtt{i,c*v}\to\mathtt{v}}(\mathtt{x})\cup\pi_{\mathtt{i,-d*v}\to\mathtt{v}}(\mathtt{y})).$$

Transponierung A^T und x^T

Vor der Diskussion der verschiedenen Produktvariation sei erwähnt, dass die Transponierung (der Fall der Adjungierung verläuft analog) von Matrizen durch eine einfache Umbenennung umgesetzt werden kann. Im Fall eines Vektors kann durch das Hinzufügen eines Zeilen-Attributs mit dem konstanten Attributwert 1 eine Überführung in eine Matrix simuliert werden. Es ergibt

sich daher:

```
select 1 as i,i as j, v
from x
```
$\pi_{\texttt{1 as i,i}\rightarrow\texttt{j}}(\texttt{x})$

```
select j as i, i as j, v
from a
```
$\beta_{\texttt{i}\leftarrow\texttt{k}}(\beta_{\texttt{j}\leftarrow\texttt{i}}(\beta_{\texttt{k}\leftarrow\texttt{j}}(\texttt{a})))$

Die physische Umsetzung und Sicherung der Transponierung von Matrizen oder Vektoren ist in der Praxis nur selten sinnvoll, da sie in Operationen integriert werden kann. Für die spätere Diskussion zur Komposition der Operatoren in SQL in Abschnitt 7.3 ist die Nutzung dieser Teilanfragen aber denkbar. In diesem Fall kann davon ausgegangen werden, dass durch logische Optimierung die Transponierung transparent in weitere Operationen eingebunden wird, anstatt isoliert berechnet zu werden.

Skalarprodukt $\langle \mathbf{x}, \mathbf{y} \rangle$

Mit den etablierten elementweisen Operationen lässt sich das Skalarprodukt, welches die wohl einfachste Produktvariante der Fundamentaloperatormenge ist, einfach herleiten. Die Form des (Standard-)Skalarprodukts

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i$$

lässt leicht erkennen, dass es sich im relationalen Sinne um eine Aggregation der elementweisen Multiplikation zweier Vektoren handelt. Es folgt demnach:

```
select sum(x.v*y.v) as v
from x join y on x.i=y.i
```
$\gamma_{\emptyset;\mathbf{sum}\texttt{(v)}\rightarrow\texttt{v}}(\pi_{\texttt{v}\star\texttt{w}\rightarrow\texttt{v}}(\texttt{x} \bowtie \beta_{\texttt{w}\leftarrow\texttt{v}}(\texttt{y})))$

Hierbei wird insbesondere die Nutzung von Multimengen benötigt, da nach der Projektion zur elementweisen Multiplikation eine einattributige temporäre Relation entsteht, welche mehrfach gleiche Werte aufführen kann. Prinzipiell könnte jedoch auch die Projektion das Verbund- und Schlüsselattribut beziehungsweise den Zeilenindex `i` mit einbeziehen, um eine Mengenkonformität zu erzwingen.

Aufgrund des multiplikativen Kerns ist eine Unterscheidung zwischen dicht und dünn besetzten Besetzungsstrukturen hier nicht nötig. Dies gilt auch für die folgenden Matrix-Vektor- und Matrix-Matrix-Multiplikation.

Matrix-Vektor- und Matrix-Matrix-Produkt $A\mathbf{x}$ und AB

Das Matrix-Vektor- und das Matrix-Matrix-Produkt lassen sich aufgrund von

$$(A\mathbf{x})_i = \sum_{k=1}^{n} a_{ik}x_k = \langle A^T_{i,:}, \mathbf{x}\rangle$$

beziehungsweise

$$(AB)_{i,j} = \sum_{k=1}^{n} a_{ik}b_{kj} = \langle A^T_{i,:}, B_{:,j}\rangle$$

als Verallgemeinerung oder gruppierte Anwendung des Skalarprodukts deuten. Im Kontext der relationalen Algebra lässt sich demnach die ursprüngliche Anfrage des Skalarprodukts durch eine einfache gruppierte Aggregation erweitern. Für die Matrix-Vektor-Multiplikation ist dies etwa durch

```
select a.i as i, sum(a.v*x.v) as v
from a join y on a.j=x.i
group by a.i
```

$$\gamma_{i,\mathbf{sum(v)\to v}}(\pi_{\mathtt{i},\mathtt{v*w}\to\mathtt{v}}(\mathtt{a} \bowtie \beta_{\mathtt{j}\leftarrow\mathtt{i}}(\beta_{\mathtt{w}\leftarrow\mathtt{v}}(\mathtt{x}))\ ))$$

umsetzbar und für die Matrizenmultiplikation durch

```
select a.i as i, b.j as j,
       sum(a.v*b.v) as v
from a join b on a.j=b.i
group by a.i, b.j
```

$$\gamma_{i,j,\mathbf{sum(v)\to v}}(\pi_{\mathtt{i},\mathtt{j},\mathtt{v*w}\to\mathtt{v}}(\beta_{\mathtt{k}\leftarrow\mathtt{j}}(\mathtt{a}) \bowtie \beta_{\mathtt{k}\leftarrow\mathtt{i}}(\beta_{\mathtt{w}\leftarrow\mathtt{v}}(\mathtt{b})))).$$

Fazit

Es lässt sich zusammenfassen, dass die fundamentalen Lineare-Algebra-Operationen einfach umsetzbar im SQL-Kern und in einer vergleichsweise leicht erweiterten Relationenalgebra sind. Wird zusätzlich der Logarithmus log (für die Relationenalgebra etwa in der Projektion) unterstützt, können die (numerische stabilen) Grundoperationen der HMM aus Abschnitt 5.4 und 5.3 vollständig berechnet werden[1].
Die Effizienz der Berechnung ist hierbei jedoch noch nicht geklärt. Die Anfrageverarbeitung wird zur besseren Einordnung des Ansatzes im Folgenden für wesentliche Operationen analysiert und teilweise experimentell mit Lineare-Algebra-Bibliotheken (BLAS) verglichen.

[1] Das bis jetzt nicht diskutierte Kronecker-Delta wird hier nicht näher erläutert, entspricht in diesem Fall jedoch einer einfachen Selektion.

Die explizite Darstellbarkeit der Linearen-Algebra-Operatoren in der erweiterten Relationenalgebra ist ein wichtiges Zwischenergebnis für das diskutierte Framework. Die von Dan Suciu in [5] motivierte massiv parallele Verarbeitung von linearer Algebra in SQL-Systemen (vergleiche Kapitel 1) kann mit dieser Betrachtung gefestigt werden, da die hier vorgestellte erweiterte Relationenalgebra nach [48] die wesentliche Basis von SQL-Systemen ist. Wie in Kapitel 3 näher beschrieben worden ist, zeigt die Darstellbarkeit insbesondere Anbindungsmöglichkeiten weiterer Relationenalgebra-Anwendungen aus der Datenbank-Grundlagenforschung, wie beispielsweise die Wahrung der Datenprivatheit, die semantische Optimierung oder die Umsetzung von Provenance-Management-Techniken, auf. Da SQL der prominenteste der implementierten Vertreter relationaler Systeme ist, werden im Laufe der Arbeit jedoch vorwiegend SQL-Systeme benutzt.

7.2 Diskussion zur Verarbeitung im Datenbanksystem

Nachdem die Darstellbarkeit der Lineare-Algebra-Operatoren in SQL und der erweiterten Relationenalgebra belegt worden ist, wird in diesem Abschnitt eine Analyse der Effizienz der Berechnung der zugehörigen Anfragen durchgeführt. Hierfür werden zunächst die Produktvarianten $(\langle \mathbf{x}, \mathbf{y} \rangle, A\mathbf{x}, AB)$ experimentell in SQL-Systemen ausgewertet und mit der Verarbeitungsgeschwindigkeit von `R`, einer weit verbreiteten (Hauptspeicher-)Statistik-Software, verglichen. Letztere nutzt für die Verarbeitung dieser Operatoren Implementationen des BLAS-Frameworks aus Abschnitt 4.2. Nach dieser Auswertung wird eine kurze Darstellung von Anfrageplänen und wahrscheinlichen Implementationen der Matrizenmultiplikation, als komplexester und kritischer Stellvertreter, vorgenommen. Anhand dieser Erläuterungen werden die unterschiedlichen Laufzeiten der Operationen im Vergleich zu gängiger Scientific-Computing-Software abschließend erklärt und die einhergehenden Folgen bewertet.

Vergleich von Lineare-Algebra-Operatoren in RDBS und R (BLAS)

Zunächst sollen die etablierten SQL-Anfragen der Produktvariation (die teuersten Methoden gemäß Fließkommaoperationen) experimentell ausgewertet werden und deren Laufzeiten mit denen von BLAS-Implementationen verglichen werden. Mit Bezug zu der Diskussion zur Datenbanksystemwahl aus Abschnitt 6.2, wurden hierfür erneut die column stores MonetDB (v11.27.5) und Actian Vector (5.0), sowie der row store PostgreSQL (Version 10.1 mit der Konfigurationsanpassung nach Tabelle 7.1) auf einem Notebook mit Hardwarespezifikationen nach Tabelle 6.1 genutzt. Zur Auswertung der jeweiligen BLAS-Routinen (vgl. Abschnitt 4.2) wurden diese aus der Statistik-Software R (3.4.2) aufgerufen. Der Quellcode für die Berechnung in `R` ist in Anhang B.3.1 hinterlegt. Bei der Matrix-Vektor- und der Matrix-Matrix-Multiplikation wurden hierbei die Ergebnisse in einer Relation beziehungsweise im Fall von `R` in CSV-Dateien hinterlegt. Die

Tabelle 7.1: Rekonfigurierte Parameter von PostgreSQL 10.1

Parameter	Value
shared_buffers	3GB
temp_buffers	64MB
effective_cache_size	9GB
work_mem	1536MB
maintenance_work_mem	1535MB
min_wal_size	4GB
max_wal_size	8GB
checkpoint_completion_target	0.9
wal_buffers	16MB
default_statistics_target	500
random_page_cost	4

reine Berechnungszeit ist für `R` ebenfalls dargestellt, ist jedoch nicht als fairer Vergleichswert, sondern zur besseren Abschätzung von Schreib- und Rechenzeit zu betrachten. Die Ergebnisse zur Berechnung von Skalarprodukten sind in Abbildung 7.1, der Matrix-Vektor-Multiplikation in Abbildung 7.2 und der Matrizenmultiplikation in Abbildung 7.3 dargestellt. Hierbei wurde aufgrund der niedrigen Ergebnisvarianz für jede Dimension der jeweils beste aus drei Läufen genutzt. Aus den Experimenten sind mehrere Erkenntnisse ableitbar. Zunächst ist zu erkennen, dass der row store PostgreSQL im Allgemeinen die längste Berechnungszeit benötigt und R/BLAS die niedrigste. MonetDB ist für die Berechnung von Skalarprodukten stets das schnellste Datenbanksystem und ist etwas weniger als eine Ordnung langsamer als R. Für Matrix-Operationen ist Actian Vector das performanteste Datenbanksystem, wobei der Abstand zu R/BLAS für Matrix-Vektor-Multiplikationen in etwa eine Ordnung ist, während sie für Matrizenmultiplikation näher an zwei Ordnungen grenzt. Im letzteren Fall ist zudem erkennbar, dass MonetDB vergleichsweise schwach skaliert und teilweise langsamer als PostgreSQL ist. Es ist anzunehmen, dass in diesen Fällen aufgrund von Speicherüberlaufen eine große Menge an Berechnungen von MonetDB auf die Festplatte ausgelagert werden. Neben diesem Fall und einzelnen Schwankungen kann jedoch zusätzlich festgestellt werden, dass die SQL-Implementation im wesentlichen in der gleichen Komplexitätsklasse, wie ihre R-Konterparts liegen. Abschließend lässt sich damit zusammenfassen, dass die Matrizenmultiplikation im Vergleich zu gängigen Umgebungen des wissenschaftliches Rechnens am ineffizientesten in relationalen Datenbanksystemen umsetzbar ist, während Skalarprodukte und Matrix-Vektor-Multiplikation vergleichsweise performant sind. Für die Berechnung der HMM-Operationen sind Matrizenmultiplikationen nicht nötig, welches prinzipiell positiv für die Verarbeitung in Datenbanksystemen gewertet werden kann. Trotzdem werden solche Multiplikationen in vielen weiteren wissenschaftlichen Methoden, wie etwa Eigenwertverfahren (vergleiche Hauptkomponentenanalysen in Abschnitt 10.2.1) oder et-

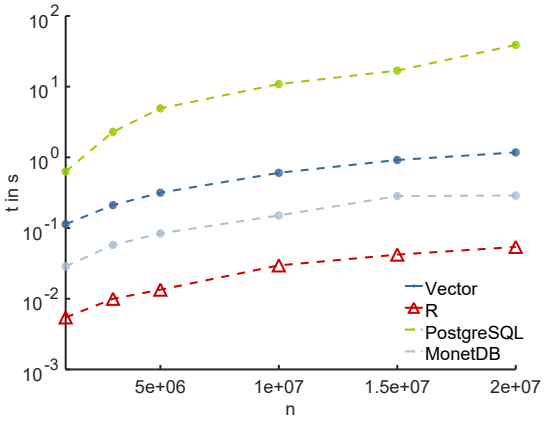


Abbildung 7.1: Laufzeiten zur Berechnung von Skalarprodukten mittels SQL-Anfragen in PostgreSQL, MonetDB und Actian Vector und mittels BLAS-Routine aus R.

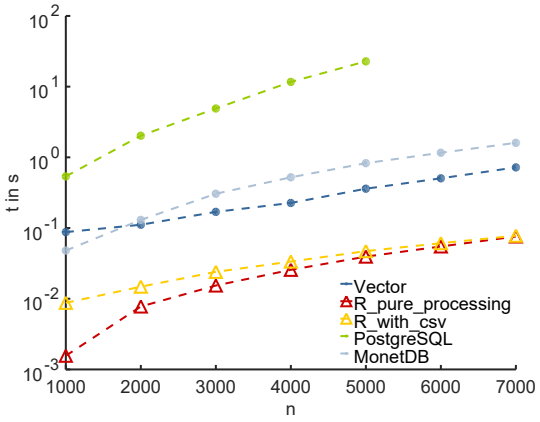


Abbildung 7.2: Laufzeiten zur Berechnung von Matrix-Vektor-Multiplikationen mittels SQL-Anfragen in PostgreSQL, MonetDB und Actian Vector und mittels BLAS-Routine aus R.

wa Lösungsverfahren für lineare Gleichungssysteme, benötigt. Dies ist demnach eine größere Restriktion der Nutzbarkeit des Frameworks für SQL-basierte Implementationen. Werden viele Matrizenmultiplikationen benötigt, kann es demnach sinnvoll sein, reine SQL-Implementationen zu vernachlässigen. Im Folgenden wird die Anfragevearbeitung von Matrizenmultiplikationen in relationalen Systemen näher untersucht und beschrieben, um eine Erklärung für den deutlichen Performance-Gap zu BLAS-Implementationen zu bieten.

Matrizenmultiplikation in relationalen Systemen: Verbundform

Aus der verhältnismäßig einfachen SQL-Anfrage und mittels der Operatoren der Relationenalgebra ist zu erkennen, dass die Kosten beider Ansätze im wesentlichen auf die gruppierte Aggregation und den Verbund der Relationen zurückgehen. Daher werden im Folgenden diese beiden Bestandteile näher beleuchtet. Klassischerweise werden in relationalen Systemen eine Teilmenge der drei Verbundimplementationen aus Abschnitt 4.1.2 unterstützt: Hash- und Merge-Verbund, sowie Nested Loop. Sei für die folgende Untersuchung von dicht besetzten Matrizen $A \in \mathbb{R}^{k \times m}$ und $B \in \mathbb{R}^{m \times n}$ mit den zugehörigen Relationen `a` und `b` ausgegangen. Die Relation `a` enthält im Coordinate-Schema offenbar km Tupel und die Relation `b` mn. Wie in Abschnitt 4.1.2 beschrieben, kann der Nested Loop als vermeintlich langsamste Variante angenommen werden. Im optimalen Fall, das heißt wenn die innere Schleife über das Verbundattribut läuft und dieses per Indexstruktur zugänglich ist, sind die Kosten $\mathcal{O}(km \log(mn))$. Im Merge- und im Hash-Verbund können hingegen lineare Kosten von $\mathcal{O}(m(k+n))$ erreicht werden. Für den Merge-Verbund müssen hierbei jedoch die Relationen bereits sortiert vorliegen (etwa durch Index-Bäume). Als Studie zur Performance dieser Verbundimplementationen wurde die Matrizenmultiplikation in

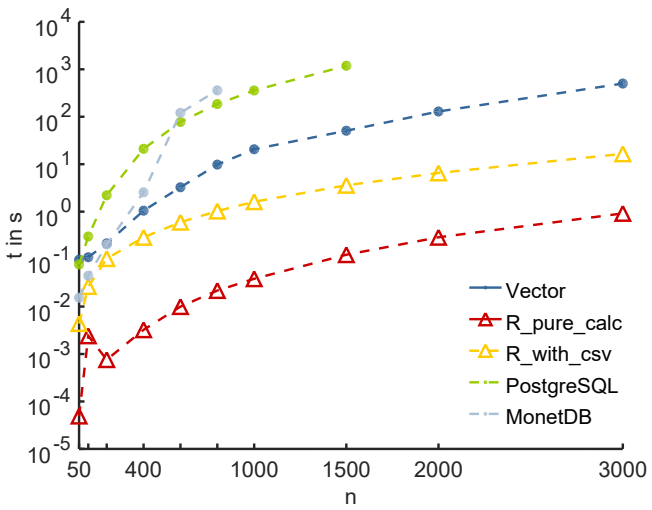


Abbildung 7.3: Laufzeiten zur Berechnung von Produkten quadratischer Matrizen $A, B \in \mathbb{R}^{n \times n}$ mittels SQL-Anfragen in PostgreSQL, MonetDB und Actian Vector und mittels BLAS-Routine aus R. Grafik erstmalig veröffentlicht in [10].

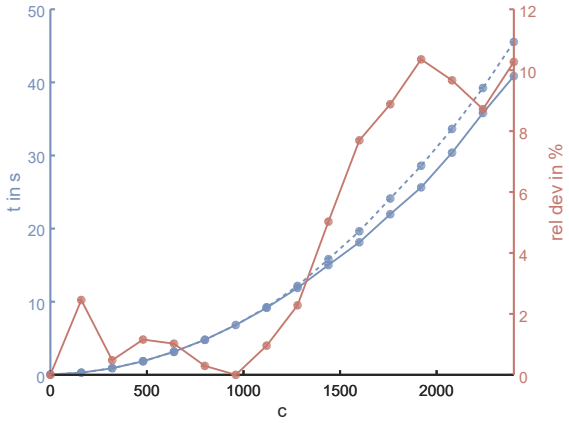


Abbildung 7.4: Berechnungszeit für das Produkt AB mittels SQL-Anfrage für Matrizen $A, B^T \in \mathbb{R}^{n \times k}$. Die durchgezogene blaue Linie beschreibt den Fall $n = 40$, $k = 20 + g(40, 20, c)$ mit g aus (7.2). Die gestrichelte blaue Linie repräsentiert den Fall von $k = 20$ und $n = 40 + c$. Die Grafik wurde erstmalig veröffentlicht in [10].

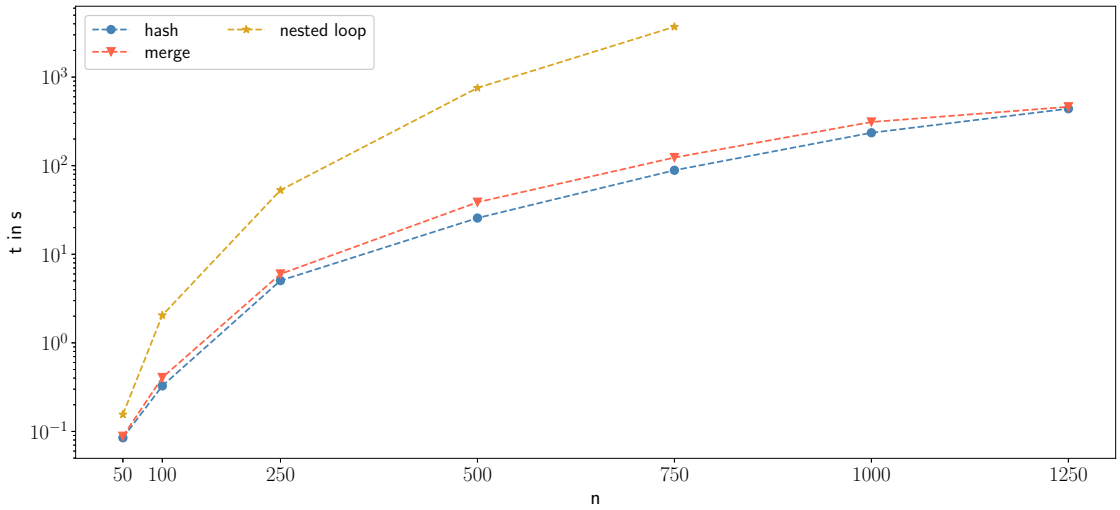


Abbildung 7.5: Laufzeiten zur Matrizenmultiplikation AB mittels SQL-Anfragen in PostgreSQL 12.1 unter Nutzung verschiedener Verbundimplementationen.

PostgreSQL 12.1 für verschiedene Dimensionen bezüglich der drei Ansätze getestet. Hierbei wurde abermals das Notebook mit den Hardware-Spezifikationen aus Tabelle 6.1 verwendet und Postgres nach Konfigurationsanpassung laut Tabelle 6.2 auf dem Betriebssystem Ubuntu 18.04.3 LTS genutzt. Für die Tests wurde auf den Relationen jeweils eine B-Baum-Indexstruktur (vergleiche Abschnitt 4.1.1) auf die Verbundattribute (`j` für Relation `a` und `i` für Relation `b`) aufgebaut, um die optimale Berechnung des Merge-Verbunds und des Nested-Loop-Ansatzes zu ermöglichen.
Die Ergebnisse sind in Abbildung 7.5 dargestellt. Wie zu erkennen ist, ist der Hash-Verbund konsistent der schnellste Ansatz, wobei der Merge-Verbund nur geringfügig langsamer ist und mit steigender Dimensionszahl näher an den Hash-Verbund aufschließt. Gründe für dieses Verhalten sind voraussichtlich der steigende Verwaltungsaufwand des Hash-Verbunds ausserhalb des Hauptspeichers (vgl. Abschnitt 4.1.2). Dies spiegelt sich insbesondere darin wieder, dass PostgreSQL den Merge-Verbund bei höherer Dimensionszahl selbst favorisiert, auch wenn der Hash-Verbund (noch) schneller ist. Die Anfrageberechnung mittels Nested-Loop-Verbund zeigt sich, trotz Nutzung von Indexstrukturen, als deutlich langsamer als die übrigen Ansätze. Insbesondere ist eine schlechtere Skalierung in diesem Experiment angedeutet, welches die vorher etablierten Kosten der Verbunde bestätigt. Damit kann zusammenfassend der Hash-Verbund als bevorzugte Verbundimplementation (für in-Hauptspeicher-Rechnungen) angesehen werden.
Aufgrund der Priorisierung dieser, wird im folgenden Abschnitt die Performance der Produktvariationen in relationalen Datenbanksystemen für Hash-Verbunde diskutiert.

Diskussion der Performance von Matrizenmultiplikationen

Wie am Anfang dieses Abschnitts gezeigt wurde, ist der relative Abstand der Verarbeitung von relationalen Systemen zu R/BLAS für die Matrizenmultiplikation am größten. Der wahrscheinlichste Grund hierfür liegt in der Art der Anfrageverarbeitung der zugehörigen (Relationenalgebra-)Operatoren. Nach der Studie in PostgreSQL ist die Verarbeitung der Aggregation mittels Hash-Verbund (im Hauptspeicherbereich) zu erwarten, wobei voraussichtlich für jeden Verbundschlüsselattributwert ein Hash-Bucket erstellt wird. Nach dem Erstellen der Buckets wird die Multiplikation und abschließende Summierung durchgeführt. Durch diese sequentielle Verarbeitungsstrategie werden Zeilen von A und Spalten von B mehrfach auf verschiedene Buckets aufgeteilt. Dadurch wird die Ausnutzung des Cache-Hit-Potenzials, welches durch den wiederholten Zugriff einzelner Elemente bzw. Zeilen und Spalten existiert und in Lineare-Algebra-Bibliotheken genutzt wird, erschwert beziehungsweise nicht ermöglicht.
Zusätzlich zeigt sich bei der Hash-Verarbeitung ein signifikantes Ungleichgewicht von Berechnungszeiten für Matrizenmultiplikationen mit unterschiedlichen Matrixdimensionen aber gleichen Fließkommaoperationskosten. Zur Erläuterung hierfür seien zwei Matrizen A, B mit $A, B^T \in \mathbb{R}^{n \times k}$ und deren Produkt AB betrachtet. Die nötigen Fließkommaoperationen belaufen sich in

diesem Fall auf $f(k,n) = n^2(2k-1)$ mit zugehöriger Kostenfunktion $f : \mathbb{N}^2 \mapsto \mathbb{N}$. Wird n (Zeilenanzahl von A, Spaltenanzahl von B) variiert folgt

$$\begin{aligned} f(k, n+c) &= (n^2 + 2nc + c^2)(2k-1) \\ &= n^2(2k-1)(1 + 2c/n + c^2/n^2) \\ &= f\left(\underbrace{k + \frac{c}{n}(2k-1) + \frac{c^2}{n^2}(k-0.5)}_{=:g(k,n,c)}, n\right) \end{aligned} \tag{7.2}$$

wobei $c \in \mathbb{N}$ ist. Letzterer Parameter wurde im folgend dargestellten Experiment genutzt um die Laufzeit der Matrizenmultiplikation mit unterschiedlichen Matrixformen jedoch gleicher benötigter Fließkommaoperationen zu vergleichen. In Abbildung 7.4 wurden hierfür die reinen Berechnungskosten (ohne konsistente Speicherung) der folgenden Fälle dargestellt:

- Viele kleine Hash-Buckets: $k = 20$ und $n = 40 + c$
- Wenige große Hash-Buckets: $n = 40$ und $k = g(k,n,c)$

Hierbei wurde der Parameter c zur Dimensionsvergrößerung variiert gemäß $c = 160 \cdot \zeta$ mit $\zeta = 1, 2, \ldots, 15$. Hierbei ist zu erkennen, dass im Fall steigender k-Werte (der Funktionswert von g) die Elementanzahl von A und B quadratisch bezüglich c wächst, im Vergleich zur linearen Abhängigkeit im ersten Fall. Demnach behandelt der zweite Fall größere und schneller ansteigende Datenmengen als der erste Fall. In der Grafik ist zu erkennen, dass mit ansteigender Bucket-Anzahl (steigendes n; gestrichelte blaue Linie) die Kosten konsistent schneller steigen, als für den Vergleichsfall anwachsender Hash-Bucket-Größen (steigendes k; durchgezogene blaue Linie), obwohl hier die zu behandelnden Matrizen weniger Elemente besitzen und mehr Cache-Hit-Potenzial, durch das mehrfache Wiederverwenden von Matrixelementen, besteht. Dieser Umstand ist vermutlich auf das erhöhte Datenmanagement zurückzuführen, welches durch das Management der großen Anzahl an Hash-Buckets entsteht.
Diese Beobachtung bestätigt die Ergebnisse der vorigen Studie von Matrix- und Vektorprodukten in SQL, indem sie aufzeigt, dass der SQL-Ansatz vor allem für Produkte schmaler Matrizen, bzw. von Produkten mit niedrigen Cache-Hit-Potenzial, vergleichsweise gut abschneidet.

7.3 Komposition von Operatoren und iterative Verfahren

Nach der Etablierung der Überführungen der Lineare-Algebra-Operatoren in SQL und einer Einordnung ihrer Performance in relationalen Datenbanksystemen wird in diesem Abschnitt die effiziente Komposition der Anfragen zu vollständigen Methoden diskutiert.
Zur effizienten Verarbeitung der HMM-Methoden aus Abschnitt 5.2, beziehungsweise allgemeinen Methoden des wissenschaftlichen Rechnens, ist eine Hintereinanderausführung der Anfragen

der Fundamentaloperatoren im Allgemeinen nicht geeignet. Vielmehr ist es nötig, eine gezielte Komposition (wie beispielsweise Pipelining) der Fundamental-Anfragen zu nutzen, um die Materialisierung von Zwischenergebnissen auf Festplatten zu vermeiden. Das Mittel der Wahl ist hierbei die Schachtelung von SQL-Anfragen, welches durch die Orthogonalität des SQL-Kerns ab SQL-92 ermöglicht wird. Dies vermeidet nicht nur das Schreiben von Zwischenergebnissen, sondern erweitert die Möglichkeiten des Datenbanksystems, logische und physische Optimierungsstrategien abzuleiten und anzuwenden. Ähnliche Effekte können konzeptuell ebenfalls durch die Erstellung aufeinander aufbauender nicht-materialisierter Sichten erzielt werden. Aufgrund der Ähnlichkeit und des zusätzlichen Verwaltungsaufwandes auf Nutzerseite, werden diese im Folgenden vernachlässigt.
Ein gutes Beispiel zur Demonstration der Effizienz der Schachtelung von Anfragen ist der Forward-Algortihmus zur Wahrscheinlichkeitsberechnung einer Beobachtungsfolge in einem Hidden-Markov-Modell (vgl. Abschnitt 5.2). Es können hier nicht nur die Zwischenergebnisse α_t weiterverarbeitet werden, sondern der Zugriff auf die Transitionsmatrix A und Beobachtungsmatrix B optimiert werden.
Zur genaueren Veranschaulichung wird nun zunächst die zeilenweise Überführung des Forward-Verfahrens aus Algorithmus 5 in SQL vorgestellt. Diese wird daraufhin genutzt, um den Performance-Gewinn der Schachtelung experimentell zu belegen. Zusätzlich werden diese Anfragen im Folgeabschnitt 7.4 verwendet, um die effiziente Nutzung von Indexstrukturen beispielhaft zu diskutieren. Es wird für die folgende Übersetzung davon ausgegangen, dass die Relationen der Vektoren `pi` und `alpha` dicht besetzt sind, dass heißt es existieren für Matrixeinträge mit Wert 0 ebenfalls Tupel. Die Beobachtungsfolge ist zudem als dicht besetzter Vektor mit Relationenschema

```
o( i int PRIMARY KEY, v double precision NOT NULL )
```

modelliert, wobei `v` jedoch Integer-Einträge nutzt. Die Initialisierung $\alpha_0 = \pi \odot B_{:,o_0}$ wird durch die folgende, als Q_0 bezeichnete, Anfrage umgesetzt.

```
select pi.i, pi.v*b.v
from pi join b
   on pi.i=b.i
where b.j=(select v from o where i=0);
```

Anfrage Q_0

Hierbei wurde zur Selektion der korrekten Spalte der Beobachtungsmatrix bereits eine Unteranfrage in der Haupt-**where**-Klausel genutzt. Ein externes Einfügen der Spalten/Beobachtungsnummern durch die Anfrageerstellungsschnittstelle ist hier ebenfalls denkbar und wurde beispielsweise in [9] angenommen. Wird ein sequenzieller Anfrageplan genutzt, muss das α_0 in

die leere Relation per `insert into` eingefügt werden.

Der $t-$te Iterationsschritt des Verfahrens kann dann aus den fundamentalen Lineare-Algerbra-Anfragen zu

```
select b.i, alpT_a.v*b.v
from (
   select a.j as i, sum(aa.v*a.v) as v
   from {alpha| Q_{t-1}} aa join a on aa.i=a.i
   group by a.j
) alpT_a join b on alpT_a.i=b.i
where b.i=alpT_a.i and b.j=(
   select v
   from o
   where i=t
)
```

Anfrage Q_t $(t = 1, \ldots, T)$

kombiniert werden. Hierbei wird unterschieden, ob die Anfragen ineinander geschachtelt werden (Q_t) oder auf einer sukzessive aktualisierten `alpha`-Relation agieren. Der Update-Prozess kann mit der `update`-Klausel und `where exists` umgesetzt werden. Für diesen Ansatz sei auf den Anhang B.3.1 verwiesen.

Die finale Aggregation zur Berechnung der Gesamtwahrscheinlichkeit wird über eine einfache Summierung

```
select sum(v) from {alpha| Q_T}
```

Anfrage Q_{T+1}

umgesetzt. Auch diese kann im sukzessiven Anfrageplan (mit `alpha`) oder in einer geschachtelten Anfrage (Q_T) genutzt werden. Für die finale Erstellung der geschachtelte Anfrage werden rekursiv die Q_t in die jeweiligen `from`-Klauseln der Anfragen Q_{t+1} eingesetzt.

Für diese beiden SQL-Überführungen wurde in [9] gezeigt, dass die iterationsübergreifende Schachtelung der SQL-Anfragen die Berechnungsgeschwindigkeit des sequenziellen Anfrageplans unterbietet. Die Ergebnisse sind in Abbildung 7.6 dargestellt. In diesen wurde eine feste 10-elementige Beobachtungsfolge in MonetDB 11.19.9. und R 3.3.1 auf dicht besetzten Matrizen $A, B \in \mathbb{R}^{n \times n}$ mit randomisiert erstellten Matrixeinträgen (hier mit $B = A$) auf dem Notebook mit den Spezifikationen aus Tabelle 6.1 ausgewertet. Dargestellt ist jeweils der Beste aus 5 Läufen. In der Abbildung ist erkennbar, dass der relative Abstand von MonetDB im geschachtelten Fall konstant geringer als eine Ordnung ist, obwohl der Kern der Kosten aus Matrix-Vektor-Multiplikationen besteht, die für sich selbst in Abschnitt 7.2 einen größeren Abstand besaßen. Insbesondere ist hier feststellbar, dass die SQL-Implementation in der selben Komplexitätsklasse, wie die der R-Implementation liegen. Dies lässt vermuten, dass durch die effiziente Schachtelung

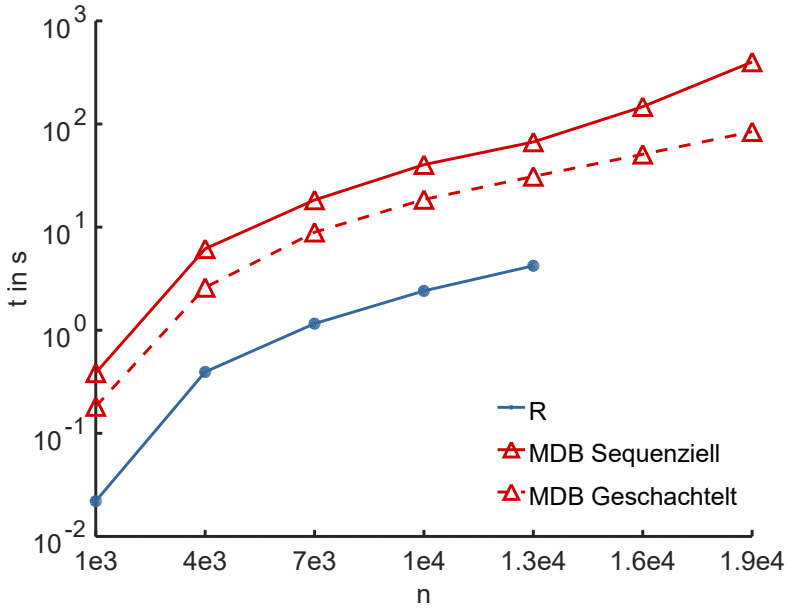


Abbildung 7.6: Berechnungszeit eines Forward-Algorithmus in R/BLAS und MonetDB mit sequenziellem und geschachteltem Anfrageplan. Die Ergebnisse wurden erstmals in [9] vorgestellt.

und Nutzung weiterer Datenbankfunktionalitäten, wie etwa Indexstrukturen (vergleiche Folgeabschnitt 7.4), der etablierte relative Abstand zu Umgebungen des wissenschaftlichen Rechnens beziehungsweise BLAS-Implementationen verringert werden kann.
Die Verarbeitungsbeschleunigung von wissenschaftlichen Methoden durch Anfrageschachtelung hat sich in einem zweiten Experiment bestätigt. In diesem wurden Matrix-Produkte der Form

$$h_d(A^{(n)}, B^{(n)}) = \underbrace{AA\ldots A}_{d \text{ mal}}\underbrace{BB\ldots B}_{d \text{ mal}} \tag{7.3}$$

bezüglich variierender Faktoranzahl ($2d$) und der jeweiligen Matrixdimensionen n ($A, B \in \mathbb{R}^{n\times n}$) in PostgreSQL 12.1 ausgewertet. Hierbei wurde die gleichen Hardware genutzt und die Konfiguration des Datenbanksystems nach Tabelle 6.2 angepasst. Der Versuch ist zudem auf die Untersuchung der folgenden Aspekte ausgelegt:

- Ist das Senden eines gesamten Anfrageplans von Anfragen der fundamentalen Operatoren stets performanter als das sukzessive Senden der einzelnen Anfragen?
- Erkennen Datenbankoptimierer Zwischenergebnisse, die die Anfrageberechnung beschleunigen können?

Der erste Aspekt zielt hierbei auf die effiziente Verarbeitung hoch-iterativer Verfahren (etwa Matrix-Faktorisierungen oder Lösungsverfahren linearer Gleichungssysteme) ab, da im SQL-

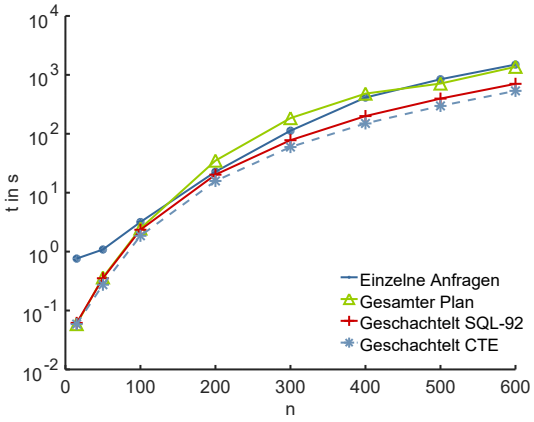


Abbildung 7.7: Vergleich der Berechnung von $h_5(A^{(n)}, B^{(n)})$ aus (7.3) in SQL mit verschiedenen Anfragestrategien.

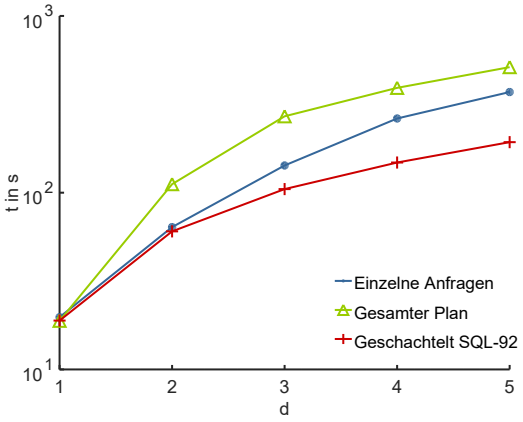


Abbildung 7.8: Vergleich der Berechnung von $h_d(A^{(400)}, B^{(400)})$ aus (7.3) in SQL mit verschiedenen Anfragestrategien.

Kern keine Schleifen-Konstrukte existieren. Prinzipiell können hierfür verschiedene alternative Ansätze getestet werden, wie etwa die Nutzung von rekursiven Anfragen (vgl. beispielsweise die Ausführungen zur Fast Fourier Transformation in Abschnitt 9.3), die Schachtelung von Anfragen, die Nutzung von *common table expressions* (CTEs) oder die Formulierung von Schleifen per UDF. Außer der Anfrageschachtelung sind diese jedoch entweder mit größerem Berechnungsaufwand verbunden oder, im Falle der Nutzung von UDFs, im Framework aufgrund der mangelnden Verbreitung des nicht vorgesehen.

Der zweite Aspekt (die transparente Erkennung und Wiederverwendung von Zwischenergebnissen des Anfrageplans) soll evaluieren, inwiefern klassische logische Optimierungsregeln relationaler Datenbanksysteme geeignet sind, einfach geschachtelte Lineare-Algebra-Anfragen in effizienter Weise zu verarbeiten. Dies lässt sich auch als Untersuchung zur Robustheit von Anfrageplänen bezüglich deren Formulierung im gegebenen Kontext interpretieren und ist insbesondere auch Teil der Untersuchung zur Nutzung von Indexstrukturen im folgenden Abschnitt.

Die experimentellen Ergebnisse sind offensichtlich abhängig von den implementierten Optimierungsregeln und demnach sehr systemabhängig. Es kann jedoch davon ausgegangen werden, dass gängige relationale Systeme einen Kern ähnlicher Optimierungsregeln besitzen. Demnach lassen Anfragepläne und relative Berechnungsgeschwindigkeiten von PostgreSQL Mutmaßungen über ähnliches Verhalten in anderen Systemen zu.

Zur Auswertung beider Aspekte wurde die Berechnung von $h_5(A^{(n)}, B^{(n)}) = A^5 B^5$ mit variierenden Matrixdimensionen n und $h_d(A^{(400)}, B^{(400)})$ mit variierender Anzahl von Matrizen $2d$ ausgewertet. Die Ergebnisse sind in den Abbildungen 7.7 und 7.8

dargestellt, wobei jeweils der Durchschnitt aus fünf Läufen genutzt wurde. In beiden Teilexperimenten wurde eine geschachtelte Anfrage in SQL-92 (Schachtelung in **from**-Klausel), sowie

ein sequenzieller Anfrageplan genutzt. Letzterer besteht aus einer Folge von Matrix-Matrix-Produkten, die zusammen das Produkt $h_d(A, B)$ berechnen. Die Anfragen wurden in einem Testszenario einzeln gesendet und in einem weiteren wurde der gesamte Anfrageplan in einem Zug an die Datenbank übermittelt. Wie sich in Abbildung 7.7 zeigt, ist der relative Kurvenverlauf dieser beiden Ansätze nicht einfacher Natur. Für niedrige Matrixdimensionen scheint der initiale Overhead der Anfrageberechnung, der etwa durch Kommunikationskosten und Anfrageplanerstellung entsteht, für das sukzessive Senden der Statements zu groß zu sein. In diesen Fällen ist ein einmaliges Senden des gesamten Plans oder eines größeren Blocks von Anfragen performanter. Dies lässt sich beispielsweise auf Matrixfaktorisierungen, wie etwa QR-Zerlegungen (vergleiche die Hauptkomponentenanalyse in Abschnitt 10.2.1) mittels Givens-Rotationen, beziehen. Bei diesen werden im Allgemeinen sehr viele Iterationen von vergleichsweise kostenarmen Matrix-Produkten durchgeführt. Zwischen $n = 200$ bis $n = 400$ kippt das Verhältnis, wobei sich in Abbildung 7.8 zeigt, dass der Abstand für $n = 400$ mit wachsender Anzahl von Matrizen d sinkt. Da sich für steigende Dimensionen das Verhältnis abermals dreht, lässt sich vermuten, dass in dem angesprochenen Fenster Auslagerungseffekte für den gesamten Plan auftreten, die für die einzelnen Anfragen (noch) nicht entstehen. Ähnliche Kurvenverläufe konnten in MonetDB in Stichproben beobachtet werden. Für die iterative Berechnung sukzessiver, unabhängiger Operationen ist demnach eine initiale Prüfung solcher Grenzen sinnvoll, um eine adäquate Größe einzelner Anfrageblöcke zu finden.

Unabhängig davon zeigt sich in beiden Abbildungen, dass die Schachtelung der Anfragen stets die bevorzugte Anfragetechnik ist und teilweise sequenzielle Anfragepläne um eine Ordnung übertrifft. Trotz der vergleichsweise guten Performance konnte in den Anfrageplänen von $h_5(A, B) = A^5B^5$ keine Ausnutzung des in Abbildung 7.9 dargestellten Umstandes

$$h_5(A, B) = (A^2)^2AB(B^2)^2$$

beobachtet werden. PostgreSQL nutzte hier unabhängig von der Dimension stets Hash-Verbunde pro geschachtelter Anfrage. Um zu testen, inwiefern die Berechnung von h_5 in Standard-SQL beschleunigt werden kann, wurde eine alternative geschachtelte Anfrage genutzt. Hierbei wurden zunächst die Produkte AA, BB und mit diesen die Produkte A^4 und B^4 mittels der in SQL:1999 eingeführten **with**-Kausel (*Common Table Expression* (CTE)) berechnet. Wie sich in Abbildung 7.7 zeigt, übertrifft diese Version konsistent die der SQL-92-Schachtelung. Aus nicht näher einsehbaren Gründen berechnet PostgreSQL die CTEs ebenfalls mittels Hash-Aggregate, jedoch den Verbund zwischen CTEs und Basisrelation mit dem in diesem Kontext langsameren Merge-Verbund (vergleiche Abschnitt 7.2). Das Einsparen von vier Matrizenmultiplikationsanfragen ist hierbei jedoch wie abgebildet profitabler.

Neben dem Performance-Gewinn können CTEs zusätzlich genutzt werden, um die Menge von

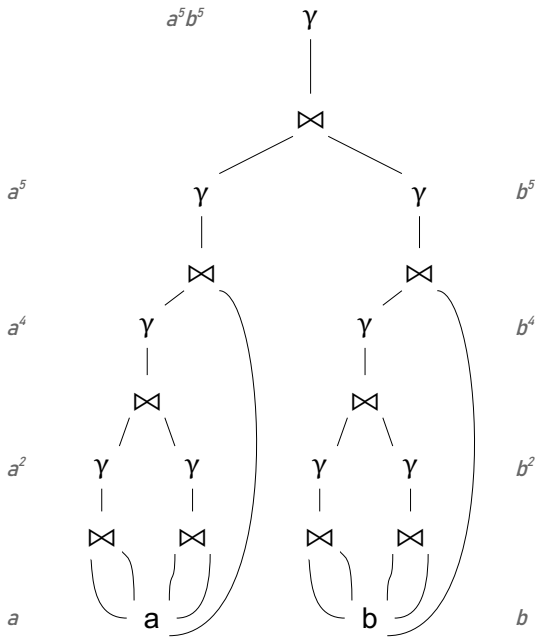


Abbildung 7.9: Vereinfachter Operatorbaum des Produkts A^5B^5 der Matrizen A, B mit zugehörigen Relationen `a` und `b`. Auf der Ebene A^2 bzw. B^2 ist zu erkennen, dass die jeweiligen Subbäume doppelt (*common subtree*) auftreten.

schachtelbaren Szenarien zu erweitern beziehungsweise kompakter zu ermöglichen. In der SQL-92-Schachtelung können etwa vorige Zwischenergebnisse genutzt werden, wenn diese in der **from**-Klausel referenziert werden. Dies führt, wie exemplarisch in Abbildung 7.10 dargestellt, dazu, dass eine Weiternutzung von Zwischenergebnissen / temporären Relationen nur durch explizite Duplikation der jeweiligen Anfragen (hier etwa Q_{11} und Q_{12}) in der geschachtelten Form möglich ist. Prinzipiell könnten die Optimierungsschnittstellen die identischen Teilbäume erkennen und die Zwischenergebnisse physisch nur einmalig berechnen. Wie sich jedoch bereits im Beispiel für $h_5(A, B)$ gezeigt hat, ist dies im Allgemeinen nicht der Fall. Mittels CTEs kann die Beispielanfrage Q_3 aus Abbildung 7.10 in der Form

with $Q_{11}, Q_{12}, Q_{21}, Q_{22}$
Q_3

vergleichsweise einfach umgesetzt werden.
Zusammenfassend lässt sich eine optimale Anfragekomposition in Standard-SQL nur als schwer vorhersehbar und sehr systemspezifisch beschreiben. Die Schachtelung von Anfragen zeigte sich als effizienteste Methode, wobei die Umsetzung dieser mittels CTEs zusätzlich den Datenbank-

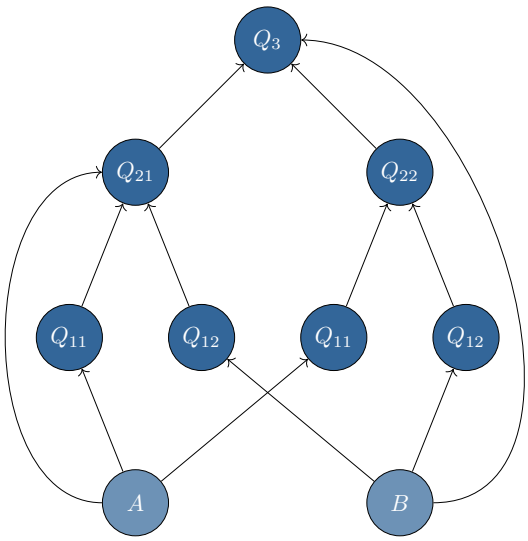


Abbildung 7.10: Beispiel eines allgemeinen Anfragegraphen für SQL-92-Anfragen. Jeder Knoten beschreibt eine SQL-Anfrage, wobei Kanten ein Auftreten des ausgehenden Knotens in der **from**-Klausel des eingehenden Knotens bezeichnen. Die Zwischenergebnisse Q_{11}, Q_{12} mit Bezug zu den Basisrelationen A und B müssen explizit neu formuliert werden, um die auf diesen aufbauenden Zwischenergebnisse Q_{21} und Q_{22} berechnen zu können.

optimierern helfen kann, sinnvolle Zwischenergebnisse zu nutzen. Letzterer Ansatz zeigte in PostgreSQL jedoch teilweise nur schwer nachvollziehbare Anfrageverarbeitungspläne und ist mitunter in manchen Datenbanksystemen nicht unterstützt. Die Verarbeitung hoch-iterativer Anfragen ist prinzipiell schwierig aufgrund fehlender Schleifen-Konstrukte und sollte in zukünftigen Betrachtungen näher untersucht werden.

7.4 Einfluss von Indexstrukturen

Neben der bereits beschriebenen Nutzung von Indexstrukturen zur Berechnung von Nested-Loop- und Merge-Verbunden wird im Folgenden eine potenzielle Beschleunigung von SQL-Anfragen im präsentierten Kontext durch diese diskutiert. Indexstrukturen sind weitverbreitete nicht-standardisierte Zugriffsstrukturen in relationalen Datenbanksystemen, die vor allem für selektive Anfragen große Verarbeitungsbeschleunigungen ermöglichen. Hier werden die in Abschnitt 4.1.1 beschriebenen Indexstrukturen in zwei getrennten Untersuchungen auf ihren Nutzen im Kontext von Hidden-Markov-Modellen und wissenschaftlichem Rechnen ausgewertet.
Die erste Studie testet klassische, weitverbreitete sekundäre Indexstrukturen bezüglich deren Potenzial, die etablierten Lineare-Algebra-Operatoren und HMM-Methoden aus den Abschnitten 5.2 und 5.3 zu beschleunigen. In der zweiten Studie werden modernere Strukturen, wie blockweise Min-Max-Statistiken, für den Lineare-Algebra-Kontext diskutiert und getestet.

B-Bäume und GIN-Indexstrukturen

Zur initialen Betrachtung wird der Einfluss der in Abschnitt 4.1.1 etablierten sekundären Indexstrukturen für den hier untersuchten Kontext diskutiert und experimentell ausgewertet. Hierfür werden zunächst die Auswirkungen dieser auf die Berechnung von Stellvertretern der Lineare-Algebra-Operatoren aus Tabelle 5.2 untersucht. Anschließend wird der essenzielle Nutzen dieser für die beschriebenen HMM-Grundmethoden begründet. Die Ergebnisse der folgenden Experimente wurden erstmalig in [10] veröffentlicht.

Für die Untersuchung der fundamentalen Operatoren wurden die etablierten B-Bäume und Hash-Indexstrukturen, sowie GIN-Indexstrukturen in PostgreSQL 10.1 (vgl. Abschnitt 4.1.1) auf dem Notebook mit den Spezifikationen aus Tabelle 6.1 untersucht. Das System wurde hierbei mit Bezug auf Tabelle 7.1 rekonfiguriert.

Es wurde unterschieden zwischen dicht und dünn besetzten Szenarien, wobei jeweils eine elementweise Operation und ein Produkt ausgewertet wurden. Als Stellverteter der elementweisen Operationen wurde die Addition $A+B$ gewählt. Für dicht besetzte Szenarien wurde die Matrixmultiplikation AB und für dünn besetzte Szenarien das Matrix-Vektor-Produkt $A\mathbf{x}$ ausgewertet. Für letztere wurden Matrizen mit unterschiedlicher (aber jeweils fester) Anzahl an Elementen pro Zeile („branching“) getestet. Die zugehörigen Anfragen entsprechen denen aus Abschnitt 7.1. Die Anfrageverarbeitung von PostgreSQL wurde derart konfiguriert, dass indexorientierte Anfragepläne erstellt wurden, auch wenn dies negative Folgen für die Berechnungsdauer hatte.

Die Nutzung von GIN-Indexstrukturen wurde aufgrund deren Kompaktheit und ihrer Fähigkeit, effizient partial match queries (Anfragen auf Teilschlüsseln) zu berechnen getestet. Durch letztere können Anfragen mit Spalten-, Zeilen- oder allgemeinen Submatrixselektionen mittels einer einzelnen Struktur umgesetzt werden.

Für die Tests wurden auf Matrizen jeweils zweidimensionale B-Bäume bezüglich (`i`,`j`) und (`j`,`i`) erstellt und getrennt hiervon GIN-Indexstrukturen auf dem Attributpaar `i` und `j`. Zusätzlich wurde getestet, ob eine Hinzunahme des Wertattributs `v` einen direkten Zugriff der Matrixeinträge über den Index ermöglicht. Letzteres hatte jedoch keinen Einfluss und wurde daraufhin vernachlässigt. Für Vektoren wurden die jeweiligen Indexstrukturen bezüglich des Zeilenattributs `i` aufgebaut.

Analog hierzu wurden Hash-Indexstrukturen erstellt, welche sich jedoch bei initialen Tests als konsistent langsamer herausgestellt haben und aus diesem Grund vorzeitig aus den Untersuchungen ausgeschlossen wurden.

Die Laufzeitergebnisse bezüglich der dicht besetzten Szenarien sind in Abbildung 7.11 und der dünn besetzten Probleme in Abbildung 7.12 dargestellt. Aus diesen Darstellungen ist zunächst erkennbar, dass keine der Ansätze mit oder ohne Indexstrukturen in jedem Fall am besten oder schlechtesten abschneidet. GIN-Indexstrukturen zeigen für dicht besetzte elementweise Ope-

rationen deutliche Performance-Nachteile, ähnlich zu denen von B-Bäumen für dicht besetzte Matrizenprodukte. Ein Grund für letzteres ist, dass Postgres Hash-Verbunde, wie bereits in Abschnitt 7.2 beschrieben wurde, effizienter umsetzt als die index-basierten Alternativen. Im Falle des dünn besetzten Matrix-Vektor-Produkts zeigt sich dieser Effekt noch deutlicher. Für dünn besetzte elementweise Operationen zeigen B-Bäume einen deutliche Beschleunigung bezüglich der Anfrageberechnung im Vergleich zu der Verarbeitung ohne Index. Dies hängt vermutlich mit dem vergleichsweise selektiven Charakter und der niedrigen Menge an Fließkommaoperationen der Anfrage zusammen. So müssen, wie bei jedem der vier Szenarien, zwar alle Tupel der Relationen gescannt und verarbeitet werden, jedoch müssen beim Verbund pro Element einer Relation kein oder nur ein Tupel aus der jeweils anderen gefunden werden. Dies kann, abhängig von der Implementation, zu einem ungünstigen Verhältnis zwischen der Anzahl und der Größe von Hash-Buckets für indexfreie Ansätze führen.
Es kann für die Linearen-Algebra-Operatoren zusammengefasst werden, dass trotz der Verarbeitung ganzer Relationen Fälle existieren, in dem eine Verwendung von Indexstrukturen sinnvoll sein kann. GIN-Indexstrukturen haben sich in diesen Testszenarien am ineffizientesten erwiesen, sind jedoch invariant bezüglich der Attributreihenfolge. Damit könnten sich diese vor allem bei der Verarbeitung höherdimensionaler Tensoren als effizient erweisen. Da Postgres für die Verarbeitung der Anfragen vor allem Hash-Verbunde nutzt, können Testergebnisse auf anderen Datenbanksystemen, die andere Verbundimplementation favorisieren, variieren. In solchen Fällen (vor allem Nested-Loop- und Merge-Ansätze) würden klassische Indexstrukturen mutmaßlich zur Anfragebeschleunigung beitragen.

Indexstrukturen im Kontext von HMM

Prinzipiell sind Hidden-Markov-Modelle gut geeignet für die Nutzung von Indexstrukturen. So sind deren Parameter (A, B, π) nach dem Lernprozess im Allgemeinen statisch, sodass konstruierte Zugriffstrukturen nach dem initialen Aufbau nicht aktualisiert werden müssen. Im Gegensatz zu den Linearen-Algebra-Operationen sind die Methoden zur Lösung der Grundprobleme der Hidden-Markov-Modelle aus Abschnitt 5.2 zusätzlich selektiverer Natur. Dies bezieht sich vor allem auf die frequent genutzte Selektion von Spalten der Beobachtungsmatrix B, welche in jeder der diskutierten Methoden mehrfach (abhängig von der Länge der Beobachtungssequenz) vorkommt. Dies führt etwa bei hohen Matrizendimensionen beziehungsweise großen zugehörigen Relationen mit wenig Zeilen- und Spalteneinträgen (etwa dünn besetzte Matrizen) zu deutlichen Beschleunigungen der Spaltenselektion gegenüber klassischen *full table scans*. Zur Veranschaulichung dieses Aspekts sei im Folgenden eine Studie zur Anfragebeschleunigung eines Iterationsschritts des Forward-Verfahrens aus Algorithmus 5 in SQL für dünn besetzte Matrizen $A, B \in \mathbb{R}^{n \times n}$ und dicht besetztem 1-Vektor $\alpha = \mathbf{1} \in \mathbb{R}^n$ (alle Einträge besitzen den Wert 1) präsentiert. Hierbei wurden ebenfalls GIN- und B-Baum-Indexstrukturen auf PostgreSQL auf

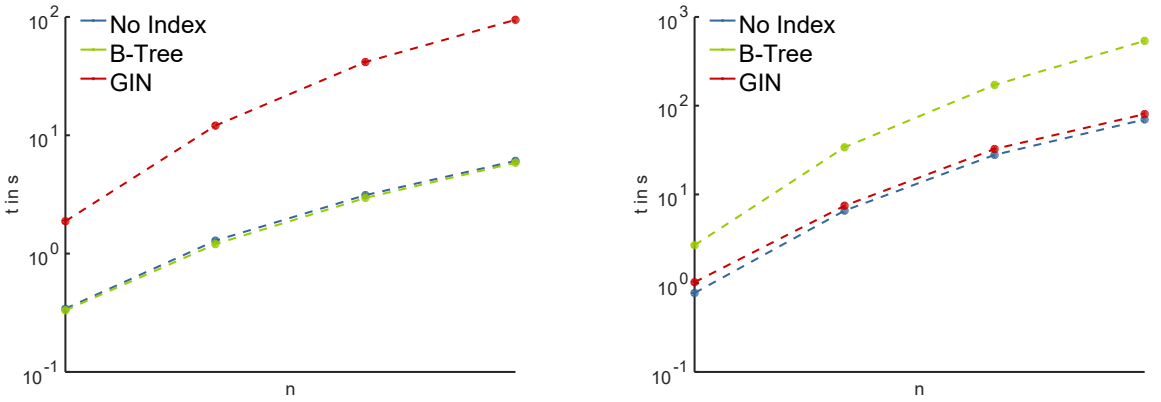


Abbildung 7.11: Vergleich dicht besetzter Matrixadditionen $A + B$ (links) und Matrizenmultiplikationen (rechts) mit und ohne Indexstrukturen. Grafiken erstmalig in [10] veröffentlicht.

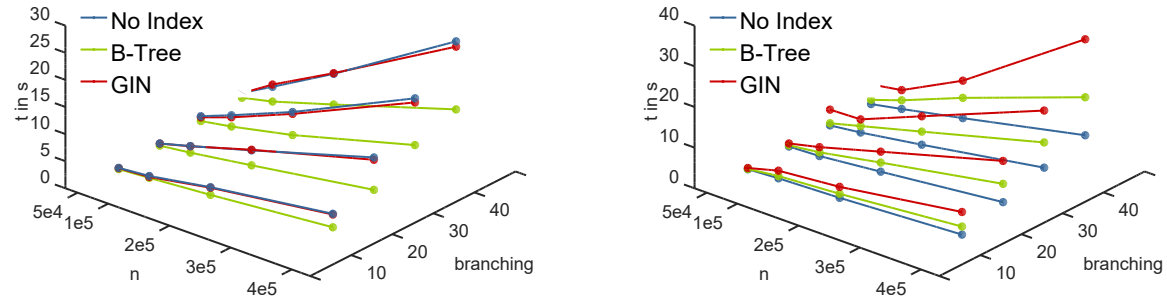


Abbildung 7.12: Vergleich dünn besetzter Matrixadditionen $A + B$ (links) und dünn besetzter Matrix-Vektor-Multiplikationen (rechts) mit und ohne Indexstrukturen. Grafik erstmalig in [10] veröffentlicht.

dem identischen Setup getestet. Die Matrizen besitzen 20 Elemente pro Zeile und sind in einer weiten Bandform strukturiert (Bandbreite $\pm 3n/40$), um etwaige Nachbarschaftsbeziehungen der Zustände zu simulieren. Da

$$\underbrace{(\mathbf{1}^T A)}_{\text{dicht besetzt}} \odot \underbrace{B_{:,k}}_{\text{dünn besetzt}}$$

gilt, ist das Experiment so konstruiert, dass die Struktur des Ergebnisvektors identisch mit der der k-ten Spalte der dünn besetzten Matrix B ist. Aus diesem Grund ist die tatsächliche Anzahl benötigter Fließkommaoperationen im Durchschnitt konstant bezüglich der Dimension n (vgl. hierfür etwa den direkten Bezug zwischen der Anzahl von Nicht-0-Spaltenelementen und der Bandbreite einer strikten Bandmatrix). Dementsprechend kann im Schnitt über mehrere Läufe

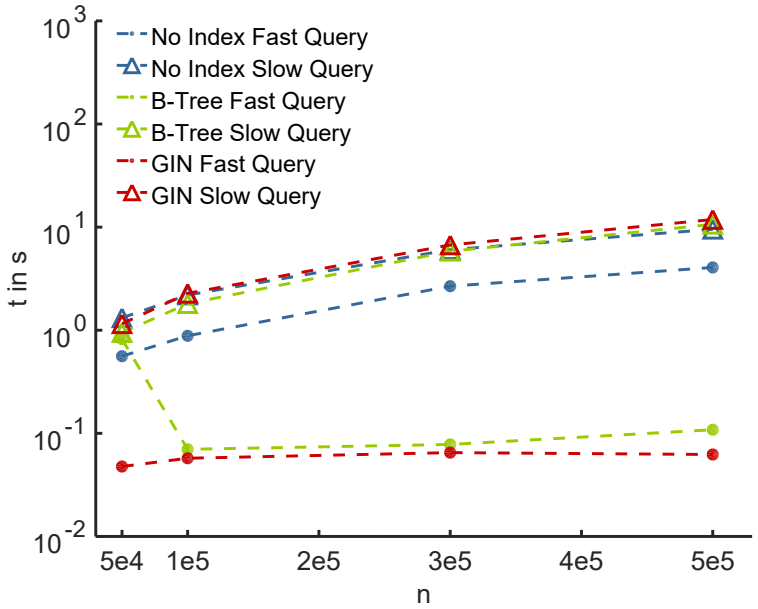


Abbildung 7.13: Berechnung des Forward-Verfahrens aus Algorithmus 5 in SQL auf dünn besetzten Zustandsübergangs- und Beobachtungsmatrizen A und B mit und ohne Indexstrukturen. Grafik erstmalig veröffentlicht in [10].

eine konstante Berechnungszeit über steigende Dimensionen erwartet werden.
Initial wurde in PostgreSQL der Iterationsschritt des Forward-Verfahrens in die Anfrage

```
select alphaTa.i as i,alphaTa.v*b.v as v
from (
    select a.j, sum(a.v*alpha.v)
    from a join alpha on a.i=alpha.i
    group by a.j ) alphaTa (i,v) join b on alphaTa.i=b.i
where b.j=k
```

überführt („Slow Query“ in Abbildung 7.13), wobei die jeweiligen Einzelanfragen der Basisoperationen einfach geschachtelt worden sind und die Selektion der k-ten Spalte von B im Sinne der Kompaktheit in der **where**-Klausel umgesetzt wurde. Wie im Falle der effizienten internen Wiederverwendung von Zwischenergebnissen von A^5B^5 aus Abschnitt 7.3, konnte PostgreSQL die klassische Schachtelung der Anfragen nicht effizient nutzen. Der Umstand, dass aufgrund der dünn besetzten Struktur von B nur die Vektoreinträge von $\alpha^T A$ berechnet werden müssen, die in $B_{:,k}$ nicht 0 sind, wurde durch das System nicht erkannt. Da 0-Einträge in der Relation `b` nicht enthalten sind, ist dieser Umstand jedoch implizit durch den inneren Verbund von `b` und `alphaTa` modelliert. In den Anfrageplänen von PostgreSQL wurde jedoch der Vektor $\alpha^T A$ komplett berechnet.

Um dies zu umgehen, ist es nötig, in der Anfrage die Besetzungsstruktur der $k-$ten Spalte von B explizit als Restriktion der zu berechnenden Einträge von $\alpha^T A$ hinzuzufügen:

```
select alphaTa.i,alphaTa.v*b.v
from (
   select a.j,sum(a.v*alpha.v)
   from a join alpha on a.i=alpha.i
   where a.j in ( select i from b where j=k )
   group by a.j ) alphaTa (i,v) join b on alphaTa.i=b.i
where b.j=k.
```

Diese Anfrage wird in Abbildung 7.13 als „Fast Query“ bezeichnet. Durch die Restriktion werden selektive Zugriffe durch die Indexstrukturen auf den beiden wesentlichen Relationen `a` und `b` ermöglicht. Wie aus der Darstellung zu entnehmen ist, profitiert diese Anfrage daher deutlich von dieser eingefügten zusätzlichen Bedingung, obwohl das jeweilige Ergebnis hierdurch nicht verändert wird. Beide Indexstrukturen, GIN und B-Bäume, erreichen den erwarteten annähernd konstanten Zeitverlauf. Im Fall von 5e4 nutzt PostgreSQL hierbei jedoch keine B-Bäume, welches die punktuell schwache Performance erklärt. In der „Slow Query“ sind die Ansätze mittels Indexstrukturen nur unwesentlich schneller als die Verarbeitung ohne zusätzliche Zugriffsstrukturen.
Zusammenfassend lassen sich Indexstrukturen als essenziell zur Berechnung des Forward-Verfahren mit dünn besetzten Matrizen beschreiben. Dies motiviert die Nutzung beziehungsweise weitere Untersuchung von Indexstrukturen für selektive wissenschaftlichen Methoden, speziell für jene, die dünn besetzte Matrizen nutzen.
Zusätzlich hat sich wiederholt gezeigt, dass trotz logischer Optimierung eine einfache Komposition der fundamentalen Operationen und Selektionen teilweise zu nicht-optimalen Anfrageplänen führt. Eine Analyse dieser, speziell im Kontext effizienter Verwaltung von Zwischenergebnissen und Indexstrukturnutzung, ist demnach bei der Umsetzung stets nötig.

Ausblick: Neuentwicklungen, Clusterung und primäre Indexstrukturen

Trotz der Etablierung und weitgehenden Verbreitung von B-Bäumen und Hash-Indexstrukturen ist die Entwicklung (und Implementierung) neuerer Strukturen ein aktives Forschungsfeld. Beispiele hierfür sind etwa Untersuchungen zum Lernen von Indexstrukturen (hier neuronale Netze) in [129] oder Indexe, die lokal gespeicherte statistische Metainformationen (etwa Min-, Max- und **NULL**-Werte in [130]) sortierter Daten erstellen und nutzen.
Die Sortierung bzw. Clusterung von Daten ist hierbei eine der wesentlichen Funktionen primärer Indexstrukturen (vgl. Abschnitt 4.1.1) und wurde insbesondere bei der Konzeption nützlicher

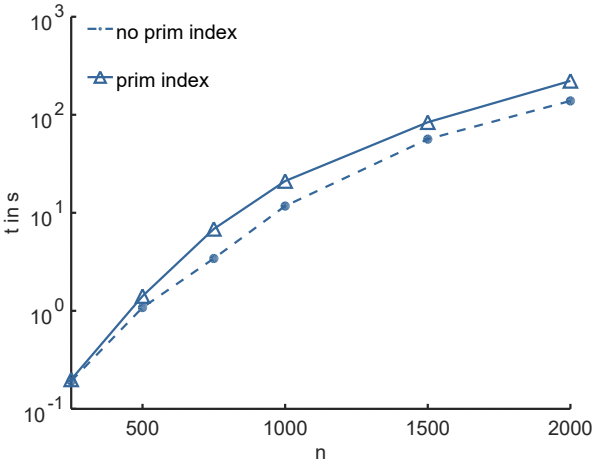


Abbildung 7.14: Einfluss von Sortierung und Min-Max-Statistiken für die SQL-basierte Matrizenmultiplikation AB in Actian Vector.

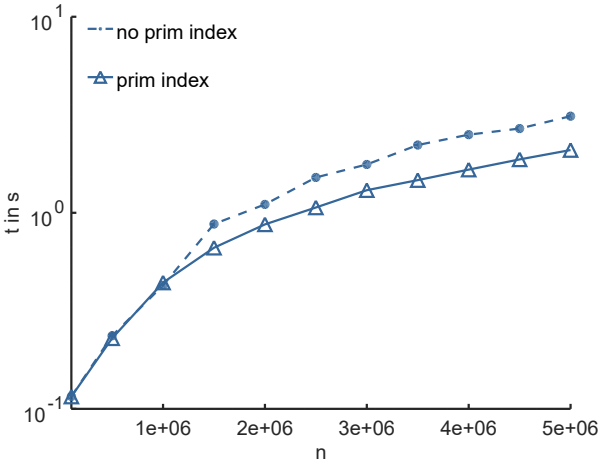


Abbildung 7.15: Einfluss von Sortierung und Min-Max-Statistiken für die SQL-basierte dünn besetzte Matrix-Vektor-Multiplikation $A\mathbf{v}$ in Actian Vector.

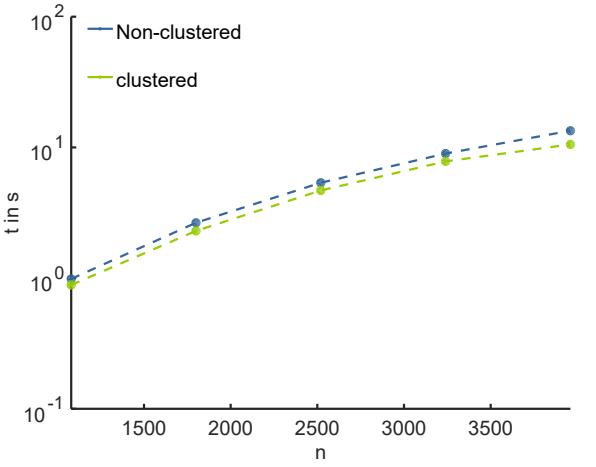


Abbildung 7.16: Darstellung der Experimentalergebnisse zur elementweisen Addition $A + B$ geclusterter quadratischer Matrizen $A, B \in \mathbb{R}^{n \times n}$ in PostgreSQL.

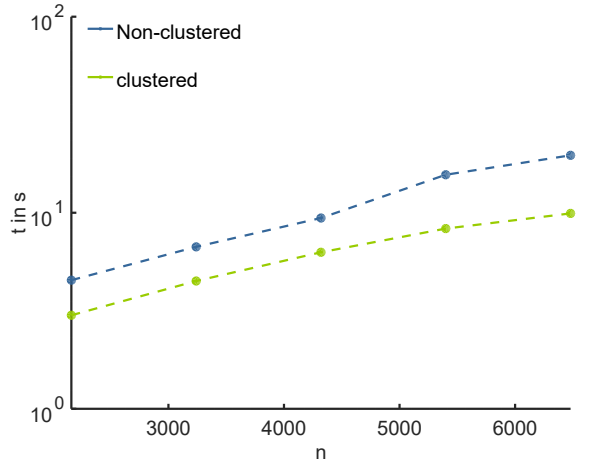


Abbildung 7.17: Darstellung der Experimentalergebnisse zur elementweisen Addition $A_{500:2000,:} + B_{500:2000,:}$ von Submatrizen geclusterter quadratischer Matrizen $A, B \in \mathbb{R}^{n \times n}$ in PostgreSQL.

Relationenschemata für Matrizen und Vektoren in Abschnitt 6.4 als eines der wesentlichen Kriterien zur effizienten Berechnung von Lineare-Algebra-Operationen beschrieben. So wurde bei der Diskussion eine sortierte Einfügereihenfolge angenommen, um implizit eine Clusterung der Zeilen beziehungsweise Spalten zu erreichen. Dies soll bei der Selektion von Submatrizen den Zugriff auf Datenbankseiten minimieren. Sind die Matrixeinträge (etwa in column stores) sortiert hinterlegt, können Zeilen, Spalten und einzelne Matrixeinträge in dicht besetzten Szenarien direkt über deren Tupelindex/Zeilennummer in der Relation angesprochen werden. Dadurch könnten die Matrizenoperationen und deren Verbunde effizient ohne Zusatzoperationen, angelehnt an klassische Implementationen in imperativen Sprachen, umgesetzt werden.
Um zu testen, wie stark sich die Clusterung auf die Anfrageverarbeitung im etablierten Lineare-Algebra-Kontext auswirkt, werden im Folgenden die Ergebnisse zweier Experimente vorgestellt. Für diese wurde abermals das Hardware-Setup aus Tabelle 6.1 genutzt.
Als initialer Test wurden Matrizenprodukte AB dicht besetzter Matrizen und Matrix-Vektor-Produkte ($A\mathbf{x}$) mit dünn besetzter Matrix A in Actian Vector 5.1.0 ausgewertet. Das System unterstützt hierbei zwei wesentliche Funktionalitäten: Zum einen werden Min-Max-Indexstrukturen für jedes Attribut in jeder Relationen standardmäßig erstellt [131]. Hierbei werden nur die minimalen und maximalen Werte aufeinanderfolgende Tupel erfasst. Die Anzahl der zusammengefassten Tupel ist hierbei so gewählt, dass jede (genügend große) Relation in 1024 Gruppen geteilt wird [132]. Die Sortierung nach Zeilen- oder Spaltenindex ist in diesem Fall durch einen primären Index umgesetzt. Mittels diesem werden optimierte, vorsortierte Merge-Verbunde (vgl. Abschnitt 4.1.1) bevorzugt bei der Anfrageverarbeitung genutzt. Da die Daten sortiert vorliegen, soll in diesem Fall weniger Speicher benötigt werden[2].
In den Abbildung 7.14 und 7.15 sind hierfür jeweils die besten Ergebnisse aus fünf Läufen dargestellt. In den Grafiken sind positive und negative Einflüsse durch den primären (Sortierungs-) Index zu beobachten. Im dicht besetzten Fall führt eine Sortierung gemäß der Verbundbedingung `a.j=b.i` zu einer signifikanten Verschlechterung der Performance. Eine Untersuchung der Anfrage

select `a.i,b.j`
from `a` **join** `b` **a.j=b.i**

durch die Actian Corporation [133] hat hierbei gezeigt, dass ohne Indexerstellung der Verbund von `a` und `b` die für das Produkt zu aggregierenden Tupeln nach a.i sortiert. Dadurch können die benötigten (`a.i`,`a.v`)-Wertkombinationen effizient im Cache gehalten und weiter verarbeitet werden. Konträr hierzu wird nach der Indexerstellung eine interne Sortierung nach `a.i,a.j` vorgenommen, wodurch das Halten der (a.i,a.v)-Wertkombinationen im Cache erschwert bzw.

[2]Originalzitat aus [131]: „Clustered indexes also affect query execution plans, causing the selection of a sort merge join over a hash join when both tables in the join are indexed on the join key. The sort merge join uses less memory than the hash join and is more efficient because data is already sorted.“

das Cache-Hit-Potenzial verringert wird. Dies ist voraussichtlich die entscheidende Ursache für die schlechtere Performance. Bei einer Nachstellung des gleichen Tests in PostgreSQL wurden ebenfalls Leistungsverschlechterungen beobachtet. In diesem Fall war der Abstand zum ungeclusterten Ansatz jedoch vergleichsweise gering und vernachlässigbar ($\approx 2\%$).

Trotz der Verarbeitungsverschlechterung im dicht besetzten Fall können dünn besetzte Matrix-Vektor-Multiplikationen etwa ab 1.5e7 Zeilen (mit bis zu 20 Elemente pro Zeile mit einer Bandbreite von 100) deutlich profitieren. Der größtenteils konstante relative Abstand ab der Dimension 1.5e7 lässt auf Auslagerungseffekte bei der Verarbeitung ohne Index schließen. Dies tritt vermutlich zum einen durch die, bereits zitierte, ressourceneffizientere Verbundimplementation mittels Merge-Verbund auf und zum anderen durch den hier gewichtigeren Aspekt der Datenselektion und -gruppierung im Vergleich zur eigentlichen Berechnung der jeweiligen Aggregate (Fließkommaoperationen).

Im zweiten Test wurde die Verarbeitung elementweiser Operationen in PostgreSQL untersucht. In diesem ist das angesprochene Verhältnis zwischen den Kosten zur Datenselektion und denen der arithmetischen Operationen vergleichsweise hoch. Es wurden hierfür abermals zwei Szenarien, dessen Ergebnisse in den Abbildungen 7.16 und 7.17 dargestellt sind, untersucht. Im ersten Test (Abbildung 7.16) wurde eine Addition zweier Matrizen $A + B$ und im zweiten (Abbildung 7.17) eine Addition der Submatrizen $A_{500:2000,:} + B_{500:2000,:}$, welche jeweils den Block der 500ten bis 2000ten Zeile enthalten, ausgewertet. Hierfür wurden die Daten ungeordnet eingefügt und die zugehörigen Relationen mit sekundärem multidimensionalen B+-Baum auf den zeilen- und spaltenidentifizierenden Attributen (`i,j`) erstellt („Non-clustered“). Nach der Auswertung dieses Ansatzes wurden die Relation bezüglich einer Postgres-spezifischen Funktionalität (`cluster on`) bezüglich der multidimensionalen B+-Bäume geclustert bzw. sortiert und erneut in getrennten Sessions ausgewertet. In den zugehörigen Grafiken ist hierbei ersichtlich, dass die volle Addition erwartungsgemäß nur geringfügig (aber konsistent) durch die Sortierung profitieren kann. Im selektiven zweiten Szenario ist ein deutlich größerer Vorteil der Clusterung erkennbar. Hierbei konnten, abhängig von den Matrixdimensionen, deutlich Einsparungen nötiger Datenbankseitenaufrufe (bis hin zur Drittelung) beobachtet werden. Bei der Durchführung des Experiments ist hierbei aufgefallen, dass die Anfrageverarbeitungsschnittstelle von PostgreSQL streckenweise ineffiziente Anfragepläne bevorzugte. So wurden, speziell beim nicht geclusterten Ansatz, streckenweise Nested-Loop und Hash-Verbund-Ansätze (vgl. Abschnitt 4.1.2) bevorzugt, die abhängig von der Dimension deutlich langsamer als Merge-Verbund-Strategien (mittels Indexstrukturen) waren. In den Abbildungen sind jeweils die mittleren Werte aus drei Läufen der effizientesten Umsetzungen dargestellt.

Zusätzlich konnte in diesem Test beobachtet werden, dass das gezielte sortierte Einfügen der Tupel in PostgreSQL zu denselben Ergebnissen und Vorteilen der expliziten Clusterung unsortierter Daten führte. Zudem sind die negativen Aspekte bezüglich der Matrizenmultiplikation

(2 % Performance-Verschlechterung) ausgeblieben, welches diesen Ansatz zum effizientesten in PostgreSQL macht. Folglich wurde im Rest der Arbeit diese Methodik verfolgt.
Als Fazit lassen sich die Effekte der Clusterung und des sortierten Einfügens von Tupeln als prinzipiell positiv bezeichnen. Vorteile sind hierbei insbesondere bei selektiven Anfragen zu erwarten. Negative Effekte bezüglich dicht besetzter Matrizenmultiplikationen (und potenziellen weiteren Operatoren) sollten in künftigen Betrachtungen weiter untersucht werden.

Kapitel 8

Parallelisierung durch Anfragezerlegung

Nach der Etablierung möglicher Überführungen der fundamentalen Lineare-Algebra-Operationen aus Abschnitt 5.4 in relationale Datenbanken wird in diesem Kapitel eine Parallelisierung dieser im Sinne der in Kapitel 3 vorgestellten Anfragezerlegung diskutiert.
Die Prämisse ist hierbei, dass die Ausführung jeder einzelnen Operation so aufwändig ist, dass eine skalierbare Parallelisierung dieser nötig ist. Dies steht etwa im Kontrast zur Interanfrage-Parallelisierung: der parallelen Verarbeitung mehrerer unabhängiger und vergleichsweise kostengünstiger Anfragen. Diese wird im Evaluationskapitel in Abschnitt 9.4 in einem Anwendungsbeispiel aus dem Automotive-Kontext [12, 13] untersucht.
Analog zu vorigen Untersuchungen wird für die folgende Diskussion zwischen dicht und dünn besetzten Problemen unterschieden. Besonderer Fokus wird hierbei auf dicht besetzte Matrizenprodukte AB und (dünn besetzte) Matrix-Vektor-Produkte $A\mathbf{x}$ gelegt, da etwa elementweise Verarbeitungen unter der Nutzung gleicher Verteilungsfunktionen vergleichsweise einfach handhabbar sind. Zusätzlich wird das Kapitel, bedingt durch deren direkte Abhängigkeit, aufgeteilt in Ausführungen zur Datenpartitionierung und der darauf aufbauenden Anfragezerlegung. Eine Evaluation der Parallelisierungsstrategien wird in Abschnitt 9.1 vorgenommen. Die Untersuchungen wurden erstmalig in kompakterer Form in [10] beschrieben.

8.1 Partitionierung von fundamentalen Operatoren

Im Folgenden werden Partitionierungsstrategien von Matrizen und Vektoren im Kontext der Intraoperator-Parallelisierung der etablierten fundamentalen Operatoren aus Tabelle 5.2 diskutiert. Grundlage hierfür sind die Coordinate-Relationenschemata aus Kapitel 6, die abgeleiteten SQL- und Relationenalgebra-Anfragen aus Abschnitt 7.1 und die klassischen Verteilungsstra-

tegien relationaler Datenbanken aus Abschnitt 2.3.5. Für die Betrachtungen werden hierbei erweiterte Replikationsstrategien zur Ausfallsicherheit vernachlässigt. Diese werden beispielsweise in [36] detailliert beschrieben und können mit den hier entwickelten Strategien kombiniert werden.
Prinzipiell ist die Wahl geeigneter Partitionierungsstrategien von zahlreichen Kritierien, wie beispielsweise den durchzuführenden Operationen, der Netzwerkanbindung oder der lokalen Berechnungsgeschwindigkeit, abhängig. Da hier parallele relationale Datenbanksysteme mit homogenen Berechnungsknoten, verbunden durch ein Hochgeschwindigkeitsnetzwerk, betrachtet werden, konzentriert sich die Untersuchung auf vergleichsweise einfache Kostenfunktionen (lokale Berechnungskosten und zu kommunizierende Tupel). Als genereller Startpunkt der Untersuchungen werden in diesem Kapitel zunächst etablierte Partitionierungsstrategien im vorgestellten Kontext für dicht und dünn besetzte Matrizen (und Vektoren) diskutiert.

8.1.1 Partitionierungsmöglichkeiten in Datenbanksystemen

Abhängig von den Modellen und Verfahren, die auf den Daten der Datenbank erstellt und berechnet werden sollen, sind verschiedene Partitionierungsformen oder Kombinationen dieser sinnvoll. Aufgrund der deklarativen Natur von SQL ist die Wahl geeigneter Verteilungsstrategien (und darauf angepasster Anfragen) sehr entscheidend für eine effiziente parallele Verarbeitung datenintensiver Operationen. Der eigentliche Verarbeitungsprozess, inklusive der Entscheidung, welche Daten auf welchem Knoten verarbeitet werden, wird durch das Datenbanksystem durchgeführt. Da Matrizen keine nativen Datentypen in SQL sind und die zugehörigen Operationen mittels klassischer relationaler Datenbankoperationen umgesetzt werden müssen, ist es nicht sicher inwiefern Datenbanksysteme mögliche lokale Berechnungsmöglichkeiten optimal ausnutzen. Zur Untersuchung dieses Problems werden in diesem Kapitel in Datenbanksystemen etablierte Partitionierungsstrategien anhand deren Kommunikationskosten und nötigen lokalen Fließkommaoperationen (FLOP - **FL**oating Point **OP**eration) für die etablierte Operatormenge, mit besonderem Fokus auf Matrixmultiplikationen $A^{[T]}B^{[T]}$ untersucht. Auf Matrizen mit komplexen Einträgen wird aus Gründen der Kompaktheit verzichtet. Diese Fälle verhalten sich jedoch analog. Die entwickelten Strategien werden in Abschnitt 8.2 genutzt, um eine Technik zur Intraoperator-Parallelisierung durch Anfragezerlegung zu entwickeln. Zudem werden die Verteilungsstrategien in Teilen in Abschnitt 9.1 evaluiert und daraufhin diskutiert, inwiefern der theoretisch erreichbare Speed-up von dem dort betrachteten parallelen Datenbanksystem Postgres-XL (vgl. Abschnitt 4.3.3) umgesetzt werden konnte.
Im Kontext des etablierten Coordinate-Relationenschemas

```
A (
   i int,
```

```
    j int,
    v double precision NOT NULL,
    PRIMARY KEY (i,j)
)
```

ist es in relationalen Systemen sinnvoll, Matrizen tupelweise bezüglich des Zeilenindexes `i`, des Spaltenindexes `j` oder einer Kombination beider zu verteilen. Eine Verteilung bezüglich der Matrixeinträge `v` kann im Fall diskreter Wertebereiche (etwa Zustandsräume) ebenfalls sinnvoll sein, ist im Allgemeinen jedoch ungeeignet und einschränkend und wird daher hier nicht weiter betrachtet. Als letzte Alternative können mögliche attributweise (vertikale) Partitionierungen aufgrund der Existenz eines einzigen Nicht-Schlüsselattributes ausgeschlossen werden.
Für die folgenden Untersuchungen werden

- die Round-Robin-Partitionierung,
- die Hash-Partitionierung und
- die Bereichspartitionierung

aus Abschnitt 2.3.5 als Vertreter klassischer Verteilungsstrategien in Datenbanksystemen genutzt.
Die Round-Robin-Verteilung (vgl. Abbildung 8.1) ist prinzipiell die mächtigste der Verteilungsansätze, da diese durch Anpassen der Einfügereihenfolge beliebige (gleichmäßig verteilte) Partitionierungen umsetzen kann. Neben dem hiermit verbundenen Extraaufwand ist jedoch das Speicherungsverhalten bei dem dynamischen Erstellen neuer Relationen aus Anfrageergebnissen (**create table as**) nur bedingt voraussehbar, welches Folgeoperationen erschwert beziehungsweise aufwändige Repartitionierungen zur Folge haben kann. Im Kontext von Matrizen lassen sich die Bereichs- und Hash-Partitionierung etwa als

- Zeilen- bzw. Spalten-Partitionierung (vgl. Abbildungen 8.2 und 8.3),
- Blockzeilen- bzw. Blockspalten-Partitionierung und
- Blockmatrizen-Partitionierung (vgl. Abbildung 8.4)

interpretieren. Diese Ansätze sind weitverbreitete Strategien für verteilte Matrixberechnungen. So sind etwa die Zeilen- und Blockpartitionierung in Apache Spark's MLlib unterstützt [134].
Ein weiterer viel beachteter Ansatz ist die in ScaLAPACK [135] (einer auf LAPACK (vgl. Abschnitt 4.2) basierenden verteilten Bibliothek) umgesetzte zyklische Blockpartitionierung (siehe [136] und Abbildung 8.5), welche hier nicht näher betrachtet wird. Die Darstellung ist prinzipiell mittels Round-Robin auch in Datenbanksystemen umsetzbar, ist aber auf die Nutzung lokal

genutzter LAPACK-Routinen und deren implementierten optimierten Ausnutzung der Speicherhierarchie ausgelegt, und wurde zum anderen bezüglich Matrix-Faktorisierungen (etwa die LU-Zerlegung) entwickelt und getestet.
Solche Faktorisierungen können hierbei als andere Menge von Lineare-Algebra-Operationen verstanden werden, da diese, im Gegensatz zu den vergleichsweise einfach strukturierten datenintensiven fundamentalen Matrix-Vektor-Operationen aus Tabelle 5.2, aus einer größeren Anzahl leichtgewichtiger und voneinander abhängigen Operationen bestehen. Wird zusätzlich die oftmals Hash-Verbund-basierte Berechnungsart in Datenbanken berücksichtigt, lässt sich vermuten, dass die zyklische Blockpartitionierung im hier diskutierten Kontext nicht von Vorteil ist. Trotzdem sollte bei möglichen künftigen Untersuchungen von Faktorisierungen in SQL-Datenbanksystemen (etwa durch rekursive Anfrageansätze) diese Strategie als potenzieller Kandidat betrachten werden.
Aus den vergleichsweise einfachen Anfrageplänen von Matrizenprodukten und elementweise Operationen in SQL (vgl. die Anfragediskussion in Kapitel 7), dargestellt in den Abbildungen 8.6 und 8.7, kann angenommen werden, dass parallele Verarbeitungsstrategien sich maßgeblich an der Definition distributiver Aggregatfunktion (vgl. Definition 9) oder der Berechnung paralleler Verbunde bzw. einzelner Gruppierungselemente von Aggregaten orientieren. Damit lässt sich etwa die Berechnung einzelner Anfragen in die folgenden Phase unterteilen:

1. Sende/Empfange benötigte Tupel an/von andere(n) Knoten (paralleler Verbund)
2. Berechne Zwischenergebnisse (Distributive Aggregation)
3. Sende Zwischenergebnisse an Knoten gemäß Verteilungsfunktion (Distributive Aggregation)
4. Berechne das finale Ergebnis aus Teilergebnissen (Distributive Aggregation / parallele Gruppierung)

Hierbei wird nicht zwangsläufig jede dieser Phasen durchlaufen. So benötigen elementweise Operationen aufgrund ihrer Einfachheit im Allgemeinen keine Berechnung von Zwischenergebnissen. Analog können gruppierte Aggregate Phase 1 bis 3 prinzipiell überspringen, falls die nötigen Daten, die im jeweiligen Knoten benötigt werden, bereits lokal vorhanden sind. Inwiefern parallele Datenbanksysteme in solchen Fällen eine ineffizientere distributive Verarbeitung vorziehen, wird in den Tests in Abschnitt 9.1, die auf den folgenden Betrachtungen aufbauen, untersucht.
Um zwischen der Effizienz einzelner Strategien zu unterscheiden, werden nun Kostenfunktionen bezüglich verschiedener Partitionierungs- und Anfragestrategien entwickelt. Die Umsetzung komplexerer Parallelisierungsansätze kann durch die Aufspaltung einzelner Anfragen in einer Menge von Unteranfragen ermöglicht werden. Dieser Ansatz wird im abschließenden Abschnitt 8.2 diskutiert.

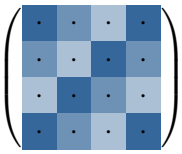


Abbildung 8.1: Round-Robin

Abbildung 8.2: Hash über Spalten

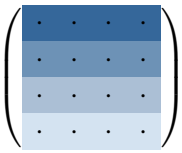


Abbildung 8.3: Hash über Zeilen

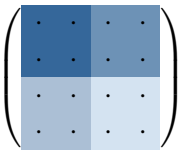


Abbildung 8.4: Blockweise - Zweidimensionale Bereichspartitionierung

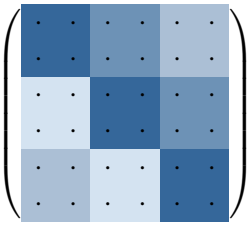


Abbildung 8.5: Zyklische Blockpartitionierung aus [136]

Notationen

Für die konzise Beschreibung von Kostenfunktionen werden zunächst nötige Notationen eingeführt. Im Folgenden beschreibt m die Anzahl der Knoten und h eine Verteilungsfunktion mit

$$\begin{aligned} & h : \{1,\ldots,m\} \mapsto \{(i,j) | i,j \in \{1,\ldots,n\}\} \\ & \bigcup_{k=1}^{m} h(k) = \{1,\ldots,n\} \otimes \{1,\ldots,n\}, \quad \forall k \in \{1,\ldots,m\} : |h(k)| = \frac{n^2}{m} \\ \text{und} \quad & \forall k_1, k_2 \in \{1,\ldots,m\} : k_1 \neq k_2 \Rightarrow h(k_1) \cap h(k_2) = \emptyset, \end{aligned}$$

die einer Knotennummer $k \in \{1,\ldots,m\}$ einer von m Partitionen einer betrachteten quadratischen Matrix $A \in \mathbb{R}^{n \times n}$ zuordnet. Im gleichen Kontext beschreibt

$$p_k \subset \{1,\ldots,n\}$$

eine Auswahl von Zeilen- bzw. Spaltenindices mit dem Komplement

$$\overline{p}_k = \{1,\ldots,n\} \backslash p_k. \tag{8.1}$$

Mit diesen Bezeichnungen können Submatrizen kompakt geschrieben werden. So beschreibt im Folgenden $A_{p_k,:}$ und $A_{:,p_k}$ die Submatrix von A, welche aus den gesamten Zeilen bzw. Spalten

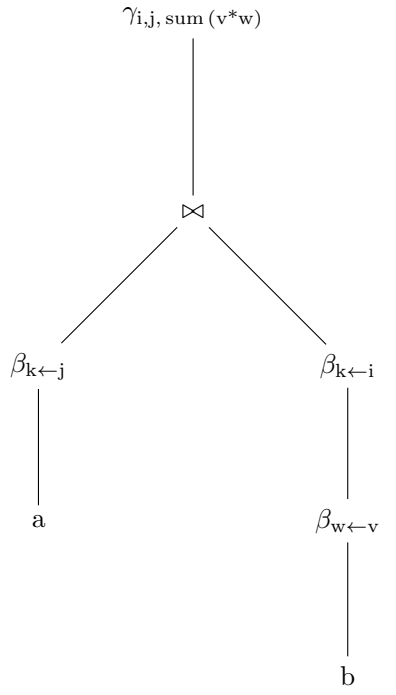


Abbildung 8.6: Anfrageplan Matrizenmultiplikation AB in SQL

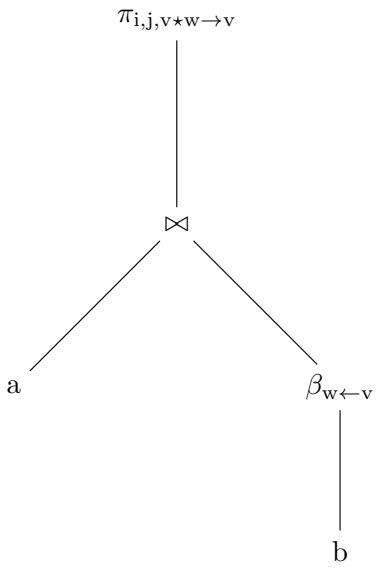


Abbildung 8.7: Vereinfachter Anfrageplan elementweiser Operationen $A \star B$ in SQL. Der Spezialfall des äußeren Verbunds (vgl. Abschnitt 7.1) ist hier vernachlässigt.

aus p_k besteht. Ist etwa $A \in \mathbb{R}^{n \times n}$ und $p_k = \{1, 7\}$, dann ist

$$A_{p_k,:} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{7,1} & a_{7,2} & \dots & a_{7,n} \end{pmatrix}$$

$$A_{:,p_k} = \begin{pmatrix} a_{1,1} & a_{1,7} \\ a_{2,1} & a_{2,7} \\ \vdots & \vdots \\ a_{n,1} & a_{n,7} \end{pmatrix}$$

$$A_{p_k,p_k} = \begin{pmatrix} a_{1,1} & a_{1,7} \\ a_{7,1} & a_{7,7} \end{pmatrix}.$$

Die Kommunikationskosten werden durch

$$\underset{\text{Operation}}{\text{Partitionierung}}\kappa$$

notiert und beschreiben die über das Netzwerk zu kommunizierenden Tupel nach jeweiliger

Partitionierungsstrategie und Operation. Analog bezeichnet

$$\overset{\text{Partitionierung}}{\underset{\text{Operation}}{\zeta}}$$

die nötigen Fließkommaoperationen pro Berechnungsknoten.

8.1.2 Dicht besetzte Probleme

Mit den etablierten Notationen werden nun Verteilungsstrategien für dicht besetzte Matrizen erörtert, die sich gemäß der 4-Phasen-Struktur in parallelen Datenbanksystemen umsetzen lassen. Hierfür werden zeilen-, spalten- und blockweise Partitionierungen untersucht. Die reine Round-Robin-Partitionierung wird in dieser Arbeit aufgrund deren Ineffizienz vernachlässigt.

Zeilen- und spaltenweise Partitionierung

Es sei zunächst die zeilenweise Partitionierung dicht besetzter quadratischer Matrizen und zugehöriger Vektoren betrachtet. Der Einfachheit halber sei für $A, B \in \mathbb{R}^{n \times n}$ und $\mathbf{x} \in \mathbb{R}^n$ angenommen, dass

$$n = m \cdot l \qquad\qquad l \in \mathbb{N} \tag{8.2}$$

gilt. Aus (8.2) lässt sich direkt ermitteln, dass pro Knoten exakt ln Matrixelemente gespeichert werden müssen um eine gleichmäßige Verteilung der Daten zu erreichen. Nicht-quadratische Matrizen oder Ansätze, in deren Fällen die Anzahl der Zeilen beziehungsweise Spalten kein Vielfaches der Knotenanzahl sind, werden hier nicht näher betrachtet, können aber etwa durch gleichmäßige Aufteilung der Restelemente analog genutzt werden.

Prinzipiell können diese Verteilungen im Fall von $n = ml$ in relationalen Datenbankensystemen mit allen der drei Basisstrategien umgesetzt werden. Für Round-Robin gilt dies etwa, bei gezielten geordneten Einfügen der Matrixelemente. Die zugehörige Verteilungsfunktion ist in diesem Fall

$$h(k) = \{(i, j) \mid k \equiv i \mod m \wedge j = 1, \ldots, n\},$$

welche ebenfalls als Hash-Verteilungsfunktion über dem Zeilenindex i interpretiert werden kann. Für die Bereichspartitionierung ist etwa

$$h(k) = \{(i, j) \mid l(k-1) < i \leq lk \wedge j = 1, \ldots, n\}$$

eine gültige Verteilung.

Im Hash-Fall ist jede gleichverteilte Partitionierung der Zeilenindizes

$$\forall k \in \{1,\ldots,m\} : p_k \subset \mathcal{P}(\{1,\ldots,n\}) \qquad |p_k| = l$$
$$\bigcup_{i=1}^{m} p_k = \{1,\ldots,n\} \qquad \forall k_1, k_2 \in \{1,\ldots,m\} : k_1 \neq k_2 \Rightarrow p_{k_1} \cap p_{k_2} = \emptyset.$$

mit der Potenzmenge $\mathcal{P}(X)$ für eine Verteilungsfunktion

$$h(k) = p_k \otimes \{1,\ldots,n\}$$

denkbar. Aus diesem Grund werden im Folgenden allgemeine Zeilen- bzw. Spaltenpartitionierungen untersucht. Es gilt zu bedenken, dass typischerweise die gleiche Verteilungsfunktion (bei gleicher Strategie; etwa Hash auf Zeilenindex) für alle Matrizen und Vektoren vom Datenbanksystem genutzt wird und auch im Folgenden angenommen werden muss, um sinnvolle Konzepte für Parallelisierungsstrategien zu untersuchen.

Das euklidische Skalarprodukt $\langle \mathbf{x}, \mathbf{y} \rangle$

Sei zunächst das euklidische Skalarprodukt

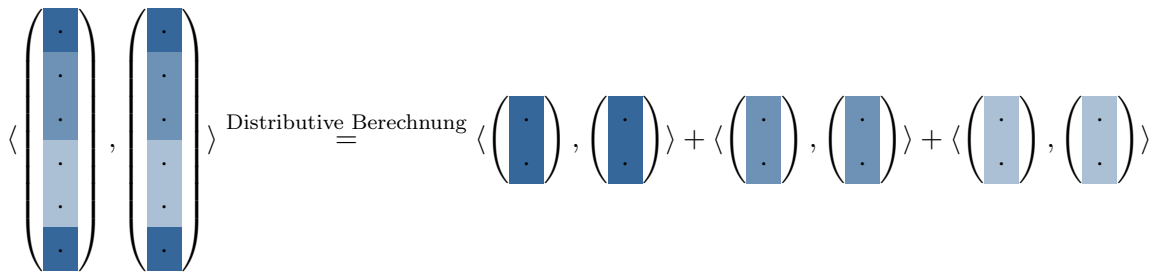
$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i$$

zweier dicht besetzter Vektoren $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ betrachtet, welches etwa als Spezialfall einer Matrizenmultiplikation interpretiert werden kann. Im Kontext dicht besetzter Probleme ist eine verteilte Berechnung von Skalarprodukten aufgrund einer vergleichsweise niedrigen Dimension im Allgemeinen nicht nötig. Da jedoch dicht besetzte Skalarprodukte auch häufig im Kontext hochdimensionaler dünn besetzter Probleme auftreten, ist eine Betrachtung sinnvoll. Da davon ausgegangen werden kann, dass die Vektoren bzgl. der gleichen Verteilungsfunktion aufgeteilt worden sind, kann aus der zugehörigen SQL-Anfrage

```
select sum(x.v*y.v)
from x join y on x.i=y.i
```

angenommen werden, dass die elementweise Multiplikation und der Verbund lokal verarbeitet wird. Damit reduziert sich das Skalarprodukt auf die Aggregation **sum** und ist dementsprechend ein Musterbeispiel für distributive Verarbeitung in parallelen Datenbanksystemen. Die Berechnung wird dementsprechend in kleinere Teilskalarprodukte aufgeteilt und bis auf eine nötige

finale Aggregation der m Teilergebnisse komplett lokal berechnet:



Die Aufteilung in Teilskalarprodukte ist insbesondere von Bedeutung, insofern dadurch Auslagerungen auf Festplatten eingespart werden können. Insgesamt werden in diesem Fall

$$\overset{z}{\underset{\langle \mathbf{x},\mathbf{y} \rangle}{\kappa}} = \underbrace{m}_{\text{Teilergebnisse pro Knoten}}$$

Tupel kommuniziert und lokal

$$\overset{z}{\underset{\langle \mathbf{x},\mathbf{y} \rangle}{\zeta}} = \underbrace{l}_{\text{Multiplikationen}} + \overbrace{l-1}^{\text{Additionen}} = (2l-1) = 2\frac{n}{m} - 1$$

Fließkommaoperationen benötigt. Zusätzlich werden in der finalen Aggregation $m-1$ Additionen berechnet.

Das Matrix-Vektor-Produkt: $\mathbf{y} = A^{[T]}\mathbf{x}$

Es soll zunächst der Fall des Matrix-Vektor-Produkts $\mathbf{y} = A\mathbf{x}$ mit der SQL-Anfrage

```
insert into y
select a.i, sum(a.v*x.v)
from a join x on a.j=x.i
group by a.i
```

betrachtet werden. Dieses kann etwa als strukturierte $n-$elementige Gruppe von Skalarprodukten der Form

$$A\mathbf{x} = (\langle A_{i,:}, \mathbf{x} \rangle)_i$$

dargestellt werden. Entgegen der vorherig erklärten verteilten distributiven Berechnung eines einzelnen Skalarprodukts, ist es in diesem Kontext für Zeilen- und Spaltenpartitionierungen sinnvoll anzunehmen, dass mehrere ganze Zeilen oder Spalten auf einem einzelnen Knoten verarbeitet werden können. Dies entspricht etwa der im Allgemeinen gültigen Aussage, dass weniger Rechenknoten als Zeilen/Spalten der Matrix A existieren. Generell können von diesem Punkt an

als potenzielle Strategien die distributive Berechnung von n Skalarprodukten/Vektoreinträgen oder die parallele lokale Berechnung ganzer Vektoreinträge unterschieden werden.

Da der Fall zeilenweiser Verteilung von A dem der spaltenweisen Verteilung von A^T entspricht, wird hierfür im Folgenden auf die explizite Betrachtung spaltenweiser Strategien verzichtet. Beschreibt nun p_k die Zeilenindizes von A und $\mathbf{x}$, die auf dem Knoten k liegen, ist es zunächst möglich dort die Zwischenergebnisse

$$\mathbf{y}_{p_k}^{(p_k)} = A_{p_k,p_k}\mathbf{x}_{p_k}$$

lokal zu berechnen. Von diesem Punkt an werden die Elemente $\mathbf{x}_{\overline{p_k}}$ auf dem Knoten k (paralleler Verbund) benötigt, um $\mathbf{y}_{p_k}^{\overline{p_k}}$ zu bestimmen.

Alternativ kann der Vektor $\mathbf{x}$ auf jedem Knoten repliziert werden. Dies führt entweder zu keinen Kommunikationskosten, falls $\mathbf{y}$ zeilenweise gemäß p_k partitioniert wird oder zu denselben Kosten, insofern das Ergebnis $\mathbf{y}$ abschließend repliziert wird, um etwa Verarbeitungsschritte effizient zu ermöglichen. Dieser Ansatz umgeht zudem mögliche zusätzliche Berechnungskosten durch die Zusammenrechnung von Teilergebnissen gemäß der distributiven Aggregation. So könnte, abhängig von der Implementation, das Datenbanksystem die fehlenden Elemente $\mathbf{x}$ asynchron empfangen und die Teilprodukte $\mathbf{y}_{p_k}^{(p_l)}$ zunächst nacheinander berechnen und gemäß der distributiven Aggregation, diese anschließend finalisieren. Hierdurch entstehen $m-1$ zusätzliche FLOPs pro Knoten und zudem mögliche Einschränkungen und Abhängigkeiten durch die Kommunikation der Tupel über das Netzwerk. Dieser Extraaufwand wird im Allgemeinen für hohe Dimensionen jedoch weitgehend vernachlässigbar sein. Kritisch könnten sich jedoch ineffiziente Strategien, wie die Kommunikation von Elementen von A oder die Berechnung von Teilergebnissen auf „falschen“ Knoten, auswirken.

Aus diesem Grund soll nun der effizientere Fall der Replikation von $\mathbf{x}$ betrachtet werden. In diesem Fall kann auf dem Knoten k das Produkt

$$\mathbf{y}_{p_k} = A_{p_{:},:}\mathbf{x}$$

komplett lokal berechnet werden. Die lokal berechneten Elemente können dann abschließend (falls benötigt) repartitioniert werden (vgl. Abbildung 8.8). Dieser datenparallele Ansatz ist mit der gängigen Replikation und der Hash-Partitionierung vergleichsweise natürlich in relationalen Systemen umsetzbar. Durch das vollständige Vorliegen der Submatrizen $A_{p_k,:}$ und des Vektor $\mathbf{x}$, wird die Berechnung des Verbunds (Phase 1) und der Aggregation (Phase 2) ohne Parallelisierung lokal mit möglicher anschließender Repartitionierung (Phase 4) umgesetzt. Als Kosten

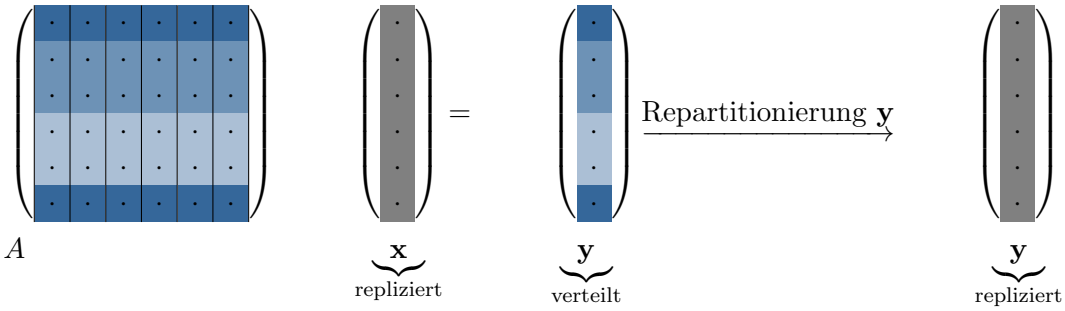


Abbildung 8.8: Beispielhafte Darstellung einer parallelen Matrix-Vektor-Multiplikation $A\mathbf{x} = \mathbf{y}$ mit zeilenweise partitionierter Matrix A und replizierten Vektoren $\mathbf{x}$ und $\mathbf{y}$.

ergeben sich in diesem Szenario eine zu kommunizierende Menge von

$$ {}^{z}_{A\mathbf{x}}\kappa = \underbrace{m(m-1)l}_{\text{Replikation } \mathbf{y}} = n(m-1) $$

Tupeln und

$$ \begin{aligned} {}^{z}_{A\mathbf{x}}\zeta &= \underbrace{l}_{\text{Zeilen } \mathbf{y}} \Big(\overbrace{n}^{\text{Multiplikationen}} + \underbrace{n-1}_{\text{Additionen}} \Big) \\ &= (2n-1)l = \frac{2n^2 - n}{m} \end{aligned} $$

FLOPs. Letzteres entspricht exakt der gewünschten gleichmäßigen Aufteilung der ursprünglichen Berechnungskosten. Trotz dieser einfachen und datenbankkonformen Strategie wird sich bei der Auswertung in Abschnitt 9.1 zeigen, dass das parallele Systeme Postgres-XL ohne weitere Einwirkungen auf Nutzerseite einen ineffizienteren distributiven Ansatz nutzt. Diese Art der Berechnung kann mithilfe der später eingeführten Anfragezerlegung in Postgres-XL jedoch „erzwungen“ werden.
Im Fall des transponierten Produktes $A^T\mathbf{x}$ mit der SQL-Anfrage

```
insert into y
select a.j, sum(a.v*x.v)
from a join x on a.i=x.i
group by a.j
```

ist die Replikationsstrategie des Vektors $\mathbf{x}$ nicht zielführend. Wie aus der Verbundbedingung `a.i=x.i` erkennbar ist, kann der parallele Verbund auch ohne eine Replikation von $\mathbf{x}$ komplett lokal umgesetzt werden (bei gleicher Verteilungsfunktion). Es wird in diesem Fall in Knoten k

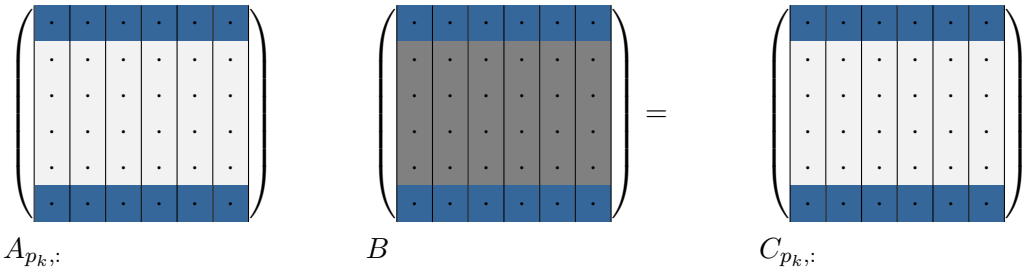


Abbildung 8.9: Schematische Darstellung lokal verfügbarer Elemente für die verteilte Berechnung von AB mit zeilenweisen partitionierten Matrizen nach Algorithmus 14. Weiße Matrixeinträge sind lokal nicht vorhanden und zudem nicht erforderlich. Graue Matrixeinträge müssen von anderen Knoten angefordert werden.

das Teilprodukt

$$(A_{p_k,:})^T \mathbf{x}_{p_k} = A^T_{:,p_k} \mathbf{x}_{p_k} = \mathbf{y}^{(p_k)}$$

berechnet. Diese Ausgangslage entspricht exakt der simultanen distributiven Berechnung von n Skalarprodukten. Dementsprechend werden die Zwischenergebnisse $\mathbf{y}^{(p_k)}_{\overline{p_k}}$ von Knoten k aus versendet und die erhaltenen Teilergebnisse $\mathbf{y}^{(p_l)}_{p_k}$ in einer finalen Aggregation zusammengeführt.

Daraus ergeben sich zunächst Kommunikationskosten von

$$\begin{aligned} {}^{z}_{A^T\mathbf{x}}\kappa &= \underbrace{m}_{\text{pro Knoten}} \; (\; \overbrace{(n-l)}^{\text{Senden von } y^{(p_k)}_{\overline{p_k}}} \;) \\ &= n(m-1) \end{aligned}$$

Tupeln, welches exakt denen des nicht-transponierten Falls mit repliziertem $\mathbf{x}$ und $\mathbf{y}$ entspricht. Analog werden lokal ebenfalls

$$\begin{aligned} {}^{z}_{A^T\mathbf{x}}\zeta &= \underbrace{\overbrace{n}^{\text{je Zeile}} (\; \overbrace{l}^{\text{Multiplikation}} + \overbrace{(l-1)}^{\text{Additionen}} \;)}_{y^{(p_k)}} + \underbrace{\overbrace{l}^{\text{je Zeile}} \; \overbrace{(m-1)}^{\text{Additionen}}}_{\sum_{\tilde{k}} y^{(p_{\tilde{k}})}_{p_k}} \\ &= n(2l-1) + l(m-1) = \frac{2n^2 - n}{m} \end{aligned}$$

FLOPs benötigt.

Algorithmus 14 Strategie zur Berechnung von $C = AB$ mit zeilenweiser Partitionierung aus Sicht von Knoten k.

1: **for** $j \in \{1, \ldots, m\} \backslash \{k\}$ **do**
2: Sende B_{p_k,p_j} an Knoten j
3: **Ende for**
4: **Erhalte** $B_{p_j,:}$ **do**
5: Berechne $C_{p_k,:} += A_{p_k,p_j} B_{p_j,:}$
6: **Ende Erhalte**

Das Matrizenprodukt $C = A^{[T]}B^{[T]}$

Es wird nun das komplexeste Produkt — das dicht besetzte Matrizenprodukt — betrachtet. Analog zum Matrix-Vektor-Produkt werden die verschiedenen Varianten AB, A^TB, AB^T und A^TB^T im Laufe des Abschnitts betrachtet. Damit schließt die explizite Untersuchung von Zeilenpartitionierungsansätzen die Fälle der Spaltenpartitionierung implizit ein.
Es sei zunächst das einfache Produkt AB mit der zugehörigen SQL-Anfrage

```
insert into c
select a.i,b.j,sum(a.v*b.v)
from a join b on a.j=b.i
group by a.i,b.j
```

betrachtet. Aus letzterer oder der Berechnungsvorschrift

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = \langle A_{i,:}, B_{:,j} \rangle$$

lassen sich mehrere Interpretationsmöglichkeiten ableiten, die im Prinzip eine Wiederverwertung der vorherig etablierten Strategien ermöglichen. So kann das Produkt etwa als n^2 Skalarprodukte oder, im Sinne des Cache-Hit-Potenzials beziehungsweise der Mehrfachverwendung von Zeilen und Spalten, als n Matrix-Vektorprodukte interpretiert werden. Trotz dieses Zusammenhangs sind Replikationsstrategien, wie sie für die Matrix-Vektor-Multiplikation motiviert wurden, in diesem Szenario aufgrund des Speicher- und Synchronisationsaufwandes ungeeignet und werden daher hier nicht näher untersucht. Ohne Replikation von A oder B ist jedoch direkt ersichtlich, dass weder durch Zeilen- noch durch Spaltenpartitionierungsstrategien eine vollständig parallele lokale Berechnung von C ermöglicht werden kann. Demzufolge ist es nötig „block-distributive Ansätze“ zu untersuchen. Es wurden hierfür zwei wesentliche Strategien in der vorliegenden Arbeit untersucht.
In der ersten Strategie werden die fehlenden Elemente einer der Matrizen angefordert, um so die lokale Berechnung der Elemente von C zu ermöglichen, die laut Verteilungsfunktion im jeweiligen

Knoten gespeichert werden müssten. Ist A etwa zeilenweise partitioniert, können durch den Erhalt aller Elemente von B im Knoten k die Zeilen von C gemäß

$$C_{p_k,:} = \underbrace{A_{p_k,p_k} B_{p_k,:}}_{\text{lokal vorhanden}} + \sum_{l \in \{1,\dots,m\} \setminus \{k\}} \underbrace{A_{p_k,p_l}}_{\text{lokal vorhanden}} \overbrace{B_{p_l,:}}^{\text{nicht vorhanden}}$$

lokal berechnet und hinterlegt werden. Das zugehörige Verfahren ist in Algorithmus 14 dargestellt. In diesem ist zu erkennen, dass die Zwischenergebnisse $A_{p_k,p_l} B_{p_l,:}$ lokal berechnet werden, welches demnach kein direkt distributiver Ansatz ist. Vielmehr wird der parallele Verbund optimal gemäß der folgenden Aggregation umgesetzt.

In dem Verfahren werden aufgrund der nötigen Kommunikation von $B_{\overline{p_k},:}$ für jeden Knoten k insgesamt

$$\begin{aligned} {}^{z}_{C=AB}\kappa &= \underbrace{m}_{\text{pro Knoten}} \overbrace{n-l}^{\text{Zeilen}} \underbrace{n}_{\text{Spalten}} \\ &= m(m-1)ln = n^2(m-1) \end{aligned}$$

Elemente kommuniziert. Jeder Berechnungsknoten berechnet ein Matrizenprodukt der Form $A_{p_k,:}B$ und benötigt demnach

$$\begin{aligned} {}^{z}_{C=AB}\zeta &= \underbrace{l}_{\text{Zeilen } B} \overbrace{n}^{\text{Spalten } B} \Big(\underbrace{n}_{\text{Mult pro Element}} + \overbrace{n-1}^{\text{Sum pro Element}} \Big) \\ &= nl(2n-1) = \frac{n^2(2n-1)}{m} \end{aligned}$$

FLOPs, welches der erwünschten optimalen linearen Ersparnis der Berechnungskosten bezüglich der Knotenanzahl entspricht.

Der zweite Ansatz — die klassisch block-distributive Methode — wird im Anhang B.4 detailliert beschrieben. In diesem wird gezeigt, dass eine Verteilung der Elemente von A zur Berechnung von Teilaggregaten $C^{(p_k)}_{p_k,:}$ und finaler lokaler Aggregation

$$C_{p_k,:} = \sum_{l \in \{1,\dots,m\}} C^{(p_l)}_{p_k,:}$$

auf dem Knoten k stets zu höheren Kommunikations- und identischen lokalen Berechnungskosten führt. Basierend auf den Anfrageplänen von Postgres-XL(siehe Anhang B.5) ist zu vermuten, dass das System diesen Ansatz jedoch präferiert. In künftigen Betrachtungen gilt es zu untersuchen, inwiefern dies umgangen werden kann, durch beispielsweise Aufteilen der Anfrage in reine

parallele Verbunde und anschließender lokaler Aggregation.

Im Anhang B.4 wird zusätzlich detailliert gezeigt, dass die Operationen $C = A^T B$, $C = AB^T$ und $A^T B^T$ (ohne Speicherung) die im bestem Fall gleichen Kosten

$$\begin{aligned} \overset{z}{\underset{C=AB}{}}\kappa = \overset{z}{\underset{C=A^T B}{}}\kappa = \overset{z}{\underset{C=AB^T}{}}\kappa = \overset{z}{\underset{A^T B^T}{}}\kappa \\ \overset{z}{\underset{C=AB}{}}\zeta = \overset{z}{\underset{C=A^T B}{}}\zeta = \overset{z}{\underset{C=AB^T}{}}\zeta = \overset{z}{\underset{A^T B^T}{}}\zeta \end{aligned}$$

besitzen. Hierbei werden insbesondere verschiedene Strategien genutzt: Im Falle $C = A^T B$ sind beispielsweise die Daten genau so partitioniert, dass für jedes Element der Matrix C genau l Tupel bereits lokal die Verbundbedingung erfüllen und so die jeweiligen Summen klassisch block-distributiv nach

$$\begin{aligned} C^{(p_k)} &= \left(A^T\right)_{:,p_k} B_{p_k,:} \\ &= \left(A_{p_k,:}\right)^T B_{p_k,:} \end{aligned}$$

berechnet werden können. Der Fall AB^T bietet ebenfalls einen Sonderfall, da die Daten bereits im nötigen „Zeilen-Spalten“-Format im Sinne der Verbundbedingung vorliegen, sodass lokal ohne Kommunikation bereits die Elemente

$$\begin{aligned} C_{p_k,p_k} &= A_{p_k,:} \left(B_{p_k,:}\right)^T \\ &= A_{p_k,:} \left(B^T\right)_{:,p_k} \end{aligned}$$

berechnet werden können. Zur Berechnung der Elemente $C_{p_k,\overline{p_k}}$ werden die zugehörigen Zeilen von B kommuniziert, wodurch die gleichen Kommunikationskosten zum Fall $C = AB$ entstehen.

Der Fall $C = A^T B^T$ ist im Vergleich zu den vorigen speziell. Die Berechnung des Produktes $A^T B^T$ kann analog zu den vorigen Fällen mit der Kommunikation von $n^2(m-1)$ Tupeln umgesetzt werden. Dies ist vor allem ersichtlich, da $A^T B^T = (BA)^T$ ist, sodass BA analog zu der Strategie für AB berechnet werden kann. Problematisch hierbei ist jedoch, dass in diesem Fall die Spalten $C_{:,p_k}$ lokal berechnet werden, jedoch die Zeilen $C_{p_k,:}$ bei gleicher Verteilungsfunktion gespeichert werden müssten. Demnach könnte eine anschließende Repartitionierung nötig werden, welche einen erhöhter Kommunikationsaufwand mit sich führen würde.

Operation	Kommunikation ${}^{z}_{*}\kappa$ in Elemente	Lokale FLOPs ${}^{z}_{*}\zeta$ nötig
$C = AB$	$n^2(m-1)$	$\frac{n^2(2n-1)}{m}$
$C = A^T B$	$n^2(m-1)$	$\frac{n^2(2n-1)}{m}$
$C = AB^T$	$n^2(m-1)$	$\frac{n^2(2n-1)}{m}$
$C = A^T B^T$	$n^2(1+1/m)(m-1)$	$\frac{n^2(2n-1)}{m}$
$A^T B^T$	$n^2(m-1)$	$\frac{n^2(2n-1)}{m}$

Tabelle 8.1: Übersicht Kostenfunktionen für Matrizenmultiplikationen bezüglich zeilenweiser Partitionierung.

Für Letzteren gilt dann

$$
\begin{aligned}
{}^{z}_{C=A^T B^T}\kappa &= \underbrace{n^2(m-1)}_{\text{Berechnung } BA} + \overbrace{m}^{\text{pro Knoten}} \underbrace{(n-l)l}_{\text{Repartitionierung } C_{\overline{p_k},p_k}} \\
&= (m-1)(n^2+nl) = (m-1)(n^2+\frac{n^2}{m}) = (m-1)n^2\left(1+\frac{1}{m}\right)
\end{aligned}
$$

Im Anhang B.4 wird zudem gezeigt, dass eine alternative Strategie, die die lokale Berechnung von $C_{p_k,:}$ ermöglicht, identische Kommunikationskosten besitzt.
Eine abschließende Zusammenfassung der Ergebnisse ist in Tabelle 8.1 dargestellt. Diese Strategien sollen zeigen, dass effiziente Ansätze für Zeilenpartitionierungen mit der Verarbeitung in parallelen Datenbanksystemen vereinbar sind. Jede der Strategien ermöglicht einen linearen Speed-Up der nötigen lokalen Fließkomma-Operationen. Zusätzlich werden die hergeleiteten Kosten genutzt, um die theoretische Überlegenheit der unten folgenden Blockpartitionierung zu demonstrieren.

Kosten elementweiser Operationen: $C = A \star B$

Abschließend wird der Vollständigkeit halber eine Diskussion der elementweisen Operationen $\star \in \{+, -, \cdot, /\}$ geführt. Der Fall $C = A \star B$ ist in diesem Fall verhältnismäßig einfach, da der Verbund komplett lokal durchgeführt werden kann. Es folgt demnach

$$
{}^{z}_{A\star B}\kappa = 0
$$

und

$$
{}^{z}_{A\star B}\zeta = nl.
$$

Der Vektor-Fall verhält sich analog.
Im transponierten Fall $C = A \star B^T$ wird voraussichtlich der parallele Verbund gemäß der Ver-

teilungsfunktion umgesetzt. Hierfür werden die Tupel $B_{\overline{p_k},p_k}$ an Knoten k gesendet um lokales Schreiben von $C_{p_k,:}$ zu ermöglichen. Der Fall $C = A^T \star B$ wird analog behandelt. Es ergeben sich Kommunikationskosten von

$$\begin{aligned} {}^{z}_{A\star B^T}\kappa &= \underbrace{m}_{\text{pro Knoten}} \overbrace{(n-l)l}^{\text{Sende } B_{p_k,\overline{p}_k}} \\ &= n(n-l) = n^2\left(1-\frac{1}{m}\right) = {}^{z}_{A^T\star B}\kappa \end{aligned}$$

Elementen und die erwarteten lokalen Prozesskosten von

$${}^{z}_{A\star B^T}\zeta = nl = \frac{n^2}{m}$$

FLOPs.

Die Berechnung der Elemente $C = A^T \star B^T$ kann (analog zum Fall $A^T B^T$) jeweils lokal durchgeführt werden, benötigt jedoch die Repartitionierung der $(n-l)l$ Ergebnistupel pro Knoten gemäß $C_{:,\overline{p}_k}$ um eine konsistente Speicherung im Sinne der Verteilungsfunktion h zu erreichen.

Es ergeben sich hierbei die folgenden Kosten:

Operation	Kommunikation ${}^{z}_{*}\kappa$ in Tupel	Lokale FLOPs ${}^{z}_{*}\zeta$ nötig
$C = A \star B$	0	$\frac{n^2}{m}$
$C = A \star B^T$	$n^2(1-\frac{1}{m})$	$\frac{n^2}{m}$
$C = A^T \star B$	$n^2(1-\frac{1}{m})$	$\frac{n^2}{m}$
$C = A^T \star B^T$	$n^2(1-\frac{1}{m})$	$\frac{n^2}{m}$
$A^T \star B^T$	0	$\frac{n^2}{m}$

Tabelle 8.2: Übersicht Kostenfunktionen für elementweise Operationen $\star \in \{+,-,\cdot,/\}$ bezüglich zeilenweiser (und spaltenweiser) Partitionierung.

Blockweise Partitionierung

Als zweiter wesentlicher Ansatz wird im Folgenden eine blockweise Partitionierung (siehe Abbildung 8.4) im Kontext paralleler Datenbanken diskutiert, welche abschließend mit der zeilen- bzw. spaltenweise Partitionierung verglichen wird.

Hier konzentrieren wir uns der Einfachheit halber auf den Fall eines quadratische Gitters an Knoten $m = \mathfrak{m}^2$. Eine Matrix $A \in \mathbb{R}^{n\times n}$ mit $\mathfrak{m}\mathfrak{l} = n$ $(\mathfrak{l},\mathfrak{m} \in \mathbb{N}^{>0})$ kann dann in ein Gitter

$$\begin{bmatrix} 1 & 2 & \dots & \mathfrak{m} \\ \mathfrak{m}+1 & \mathfrak{m}+2 & \dots & 2\mathfrak{m} \\ \vdots & \ddots & \ddots & \vdots \\ (\mathfrak{m}-1)\mathfrak{m}+1 & \dots & & \mathfrak{m}^2 \end{bmatrix}$$

Abbildung 8.10: Verteilungsmuster für die Blockpartitionierung.

quadratischer Submatrizen der Form

$$A = \begin{bmatrix} A_{11} & \dots & A_{1\mathfrak{m}} \\ \vdots & \ddots & \vdots \\ A_{\mathfrak{m}1} & \dots & A_{\mathfrak{m}\mathfrak{m}} \end{bmatrix} \quad \text{mit} \quad A_{ij} = \begin{pmatrix} a_{(i-1)\mathfrak{l}+1,(j-1)\mathfrak{l}+1} & \dots & a_{(i-1)\mathfrak{l}+1,j\mathfrak{l}} \\ \vdots & \ddots & \vdots \\ a_{i\mathfrak{l},(j-1)\mathfrak{l}+1} & \dots & a_{i\mathfrak{l},j\mathfrak{l}} \end{pmatrix} \in \mathbb{R}^{\mathfrak{l}\times\mathfrak{l}} \tag{8.3}$$

aufgeteilt werden. Die Submatrix $A_{i,j}$ wird auf dem Knoten $(i-1)\cdot\mathfrak{m}+j$ gespeichert gemäß

$$\begin{aligned} &\xi_{\mathfrak{m}} : \{1,\dots,\mathfrak{m}\}^2 \mapsto \{1,\dots,\mathfrak{m}^2\} \\ &\xi_{\mathfrak{m}}(i,j) = (i-1)\cdot\mathfrak{m}+j \end{aligned} \tag{8.4}$$

repräsentiert wird (vgl. Abbildung 8.10).

Die Partitionierung kann im Coordinate-Schema mit den Attributen `i,j` und `v` etwa als zweidimensionale Bereichspartitionierung auf `i` und `j` interpretiert werden. Wie sich bei der Evaluation zeigen wird, ist eine mehrattributige Bereichspartitionierung nicht in jedem parallelen Datenbanksystem unterstützt. Beispielsweise wird in Postgres-XL (vergleiche Abschnitt 4.3.3) die Blockpartitionierung durch gezieltes Einfügen der Matrixelemente in Round-Robin-partitionierte Relationen herbeigeführt.

Im Folgenden werden analog zur zeilenweisen Partitionierung die fundamentalen Operationen auf deren Kosten untersucht. Das Skalarprodukt wird in diesem Fall aufgrund der Nicht-Existenz von Matrizen ausgelassen. Die verteilte Berechnung des Matrix-Vektor-Produktes $\mathbf{y} = A\mathbf{x}$ ist mit blockweiser Partitionierung etwa durch Vektorreplikation möglich, wobei im Vergleich zur zeilenweisen Partitionierung zusätzliche Kommunikationskosten entstehen. Demnach ist für die Berechnung von Matrix-Vektor-Produkten prinzipiell eine Zeilenpartitionierung zu bevorzugen. Eine mögliche replikationsfreie Strategie ist im Anhang B.4 beschrieben.

Es wird nun der Fokus auf verteilte Matrizenprodukte mittels Blockpartitionierung gelegt, da diese einen theoretischen Vorteil gegenüber zeilenweisen Partitionierungsansätzen bieten.

Kosten von Matrizenmultiplikationen: $C = A^{[T]}B^{[T]}$

In diesem Abschnitt wird zunächst das einfache Matrizenprodukt $C = AB$ betrachtet. Die Elemente von $C_{i,j}$, welche auf dem Knoten $\xi(i,j)$ gespeichert werden, können durch $\mathfrak{m}$ Submatrizen-Multiplikationen und $\mathfrak{m}-1$ Submatrizen-Additionen gemäß

$$C_{i,j} = \sum_{h=1}^{\mathfrak{m}} A_{i,h}B_{h,j}$$

berechnet werden.
Aufgrund der Blockstruktur enthält keiner der Knoten komplette Zeilen oder Spalten. Aus diesem Grund werden in blockweisen Strategien typischerweise alle oben aufgeführten 4 Phasen benötigt. Ein solcher Ansatz wird nun im Folgenden diskutiert. Dieser entspricht abermals einem distributiven Ansatz mit möglichst großer Ausnutzung von Datenlokalität.
Dies bedeutet etwa, dass die Kommunikation von Submatrizen möglichst gering gehalten werden soll. Hierfür sollen die Produkte

$$C_{i,h}^{(j)} = A_{i,j}B_{j,h} \quad h = 1,\ldots,\mathfrak{m} \tag{8.5}$$

auf dem Knoten $\xi(i,j)$ berechnet. Dafür werden die nötigen Submatrizen von B an die jeweiligen Knoten kommuniziert (Phase 1) und anschließend die beschriebenen Zwischenergebnisse (Phase 2) berechnet. Diese werden daraufhin gemäß der Verteilungsfunktion versendet (Phase 3) und anschließend lokal die finalisierende Blockaggregation

$$C_{i,j} = \sum_{h=1}^{\mathfrak{m}} C_{i,j}^{(h)} \tag{8.6}$$

umgesetzt. Es ergibt sich daraus die in Algorithmus 15 beschriebene Strategie.

Algorithmus 15 Strategie zur Berechnung von $C = AB$ mit blockweiser Partitionierung nach (8.3) aus Sicht von Knoten $\xi(i,j)$.

1: Sende $B_{i,j}$ an alle Knoten der Form $\xi(h,i)$
2: **async for** $B_{j,h}$ **do**
3: Berechne $C_{i,h}^{(j)} = A_{i,j}B_{j,h}$
4: Sende $C_{i,h}^{(j)}$ an Knoten $\xi(i,h)$
5: **async for async for**
6: **async for** $C_{i,j}^{(h)}$ **do**
7: Berechne $C_{i,j} += C_{i,j}^{(h)}$
8: **async for async for**

Aus (8.5) ist erkennbar, dass im Falle $i \neq j$ kein $h = 1,\ldots,\mathfrak{m}$ existiert, sodass die Submatrizen

$A_{i,j}$ und $B_{j,h}$ auf dem gleichen Knoten gespeichert werden. Im Diagonalfall $i = j$, kann die Berechnung von $C_{i,i}^{(i)} = A_{i,i}B_{i,i}$ in Phase 1 komplett lokal geschehen, sodass hier nur $\mathfrak{m} - 1$ Matrizen versendet werden müssen. Ferner ist ersichtlich aus (8.5) und (8.6), dass stets der Summand $C_{i,j}^{(j)}$ bereits lokal vorhanden ist, sodass in der finalen Phase jeder Knoten $\mathfrak{m} - 1$ Submatrizen anfragt. Daraus lassen sich die globalen Kommunikationskosten von

$$
\begin{aligned}
{}^{B}_{C=AB}\kappa &= \underbrace{\underbrace{(\mathfrak{m}^2 - \mathfrak{m})}_{\text{Nicht-Diagonal}} \overbrace{\mathfrak{m}}^{\text{Submat.}} \underbrace{\mathfrak{l}^2}_{\text{El.}} + \overbrace{\mathfrak{m}}^{\text{Diagonal}} \underbrace{(\mathfrak{m}-1)}_{\text{Submat.}} \overbrace{\mathfrak{l}^2}^{\text{El.}}}_{\text{Phase 1}} + \underbrace{\underbrace{\mathfrak{m}^2}_{\text{Knoten}} \overbrace{(\mathfrak{m}-1)}^{\text{Submat.}} \underbrace{\mathfrak{l}^2}_{\text{El.}}}_{\text{Phase 2}} \\
&= n^2(2\mathfrak{m} - 1) - n\mathfrak{l} \\
&= n^2\left(2\sqrt{m} - \frac{1}{\sqrt{m}} - 1\right)
\end{aligned}
$$

Elementen errechnen. Im Vergleich zu den Kommunikationskosten der zeilenweisen Partitionierung ergibt sich

$$
\frac{{}^{B}_{C=AB}\kappa}{{}^{z}_{AB1}\kappa} = \frac{n^2(2\sqrt{m} - \frac{1}{\sqrt{m}} - 1)}{n^2(m-1)} = \frac{2\sqrt{m} - \frac{1}{\sqrt{m}} - 1}{m-1}
$$

welches kleiner als 1 ist, insofern $(2\sqrt{m} - \frac{1}{m}) < m$ gilt. Da $m = \mathfrak{m}^2$ ist, zeigt sich schon bei sehr niedrigen m ein deutlicher Gewinn:

m	${}^{B}_{AB}\kappa / {}^{z}_{AB1}\kappa$
4	$0.8\overline{3}$
9	$0.58\overline{3}$
16	0.45
25	$0.3\overline{6}$

.

Zur Berechnung der lokalen Anzahl an Fließkommaoperationen werden die Phasen der Berechnung der lokalen Submatrizenprodukte und die der elementweisen Addition der Matrixproduktergebnisse unterschieden. Im ersten Schritt werden zunächst auf jedem Knoten $\mathfrak{m}$ Submatrizenprodukte und im zweiten Schritt $\mathfrak{m} - 1$ Submatrizenadditionen gebildet. Daraus folgen lokale Prozesskosten von

$$
\begin{aligned}
{}^{B}_{AB}\zeta &= \underbrace{\mathfrak{m}}_{\text{Produkte}} \overbrace{(2\mathfrak{l}^3 - \mathfrak{l}^2)}^{\text{Matrizenmult.}} + \underbrace{(\mathfrak{m}-1)}_{\text{Summanden}} \overbrace{\mathfrak{l}^2}^{\text{Additionen}} \\
&= \mathfrak{l}^2(\mathfrak{m}(2\mathfrak{l} - 1) + \mathfrak{m} - 1) \\
&= \frac{n^2}{m}(2n - 1)
\end{aligned}
$$

Operation	${}^{B}_{*}\kappa$	${}^{z}_{*}\kappa$	${}^{B}_{*}\zeta$
$C = AB$	$n^2\left(2\sqrt{m} - \frac{1}{\sqrt{m}} - 1\right)$	$n^2(m-1)$	$\frac{n^2(2n-1)}{m}$
$C = A^T B$	$n^2\left(2\sqrt{m} - 2\right)$	$n^2(m-1)$	$\frac{n^2(2n-1)}{m}$
$C = AB^T$	$n^2\left(2\sqrt{m} - 2\right)$	$n^2(m-1)$	$\frac{n^2(2n-1)}{m}$
$C = A^T B^T$	$n^2\left(2\sqrt{m} - \frac{2}{\sqrt{m}}\right)$	$n^2\left(1 + \frac{1}{m}\right)(m-1)$	$\frac{n^2(2n-1)}{m}$

Tabelle 8.3: Übersicht von Kostenfunktionen für Matrizenmultiplikation bezüglich zeilenweiser und Block-Partitionierung.

Fließkommaoperationen. Die lokalen Operationen sind damit identisch zu den lokalen Kosten der zeilenweisen Partitionierung und teilen die ursprünglichen Kosten gleichmäßig auf die Knoten auf. Die analoge Strategie, die Matrix A lokal zugänglich für jeden Knoten zu machen, führt hierbei auf die gleichen Kommunikations- und Fließkommaoperationskosten.
Im Anhang B.4 wird zudem gezeigt, dass die Kommunikationskosten für die Fälle $C = AB^T$, $C = A^T B$ und $C = A^T B^T$ mit denen von $C = AB$ übereinstimmen:

$$ {}^{B}_{C=AB}\kappa = {}^{B}_{C=AB^T}\kappa = {}^{B}_{C=A^T B}\kappa = {}^{B}_{C=A^T B^T}\kappa $$

Inwiefern Datenbanksysteme die finalisierende Aggregation oder die Erstellung der Zwischenergebnisse in den richtigen Knoten umsetzen, ist abermals nur zu mutmaßen. In den Auswertungen aus Abschnitt 9.2 konnte jedoch nachvollzogen werden, dass durch die gezielte Nutzung von Subanfragen nur die „korrekten“ Knoten für die Berechnung der jeweiligen Submatrizen genutzt worden sind. Dies lässt darauf schließen, dass Postgres-XL die Verteilung der Submatrizen im Falle einer einzelnen Anfrage nicht optimal wählt und dem etwa durch geschickte Anfragestrategien entgegengewirkt werden sollte. Diese theoretischen Ergebnisse motivieren trotzdem die theoretische Überlegenheit der Blockmatrixmultiplikation und die prinzipielle Durchführbarkeit durch parallele Datenbanksysteme.

Kosten elementweiser Operationen $C = A \star B$

Der Fall der elementweisen Operationen verhält sich ähnlich zu denen der zeilen- bzw. spaltenweise Partitionierung. Im Falle $C = A \star B$ können die Verbunde von A und B auf jedem Knoten lokal durchgeführt werden. Die jeweilige Operation kostet offenbar $\mathfrak{l}^2 = n^2/m$ Fließkommaoperationen. In den Fällen $C = A^T \star B$ und $C = A \star B^T$ müssen die kompletten Submatrizen der transponierten Matrix (im optimalen Fall) zu dem Knoten der nicht-transponierten Matrix gesendet werden um lokales Schreiben zu ermöglichen. Im Diagonalknoten-Fall ist eine Kommu-

nikation daher nicht notwendig. Daraus ergeben sich globale Kommunikationskosten von

$$\begin{aligned} {}^{B}_{A\star B^T}\kappa &= \underbrace{\mathfrak{m}(\mathfrak{m}-1)}_{\text{Nicht-Diag}} \overbrace{\mathfrak{l}^2}^{\text{Submat.}} \\ &= n^2\left(1-\frac{1}{m}\right) = {}^{z}_{A^T\star B}\kappa \end{aligned}$$

Tupeln. Für den Fall $C = A^T \star B^T$ werden zunächst die lokalen Matrizen $A^T_{i,j} \star B^T_{i,j}$ auf Knoten $\xi(i,j)$ berechnet und dann zum Knoten $\xi(j,i)$ gesendet. Die Kommunikation wird für den Diagonalfall abermals ausgespart. Es ergeben sich daraus dieselben Kosten wie bei den vorigen transponierten Fällen

$${}^{B}_{A^T\star B^T}\kappa = {}^{B}_{A\star B^T}\kappa = {}^{B}_{A\star B^T}\kappa.$$

Ohne Speicherung kann dieser Fall offensichtlich ohne Kommunikation berechnet werden. In der folgenden Tabelle sind die Ergebnisse für elementweise Operationen zusammengefasst. Es ist hierbei ersichtlich, dass diese in jedem Fall identisch sind mit denen der zeilen- und spaltenweisen Partitionierung.

Operation	Kommunikation ${}^{B}_{*}\kappa = {}^{z}_{*}\kappa$ in Tupel	Lokale FLOPs ${}^{B}_{*}\zeta = {}^{z}_{*}\zeta$ nötig
$C = A \star B$	0	$\frac{n^2}{m}$
$C = A \star B^T$	$n^2\left(1-\frac{1}{m}\right)$	$\frac{n^2}{m}$
$C = A^T \star B$	$n^2\left(1-\frac{1}{m}\right)$	$\frac{n^2}{m}$
$C = A^T \star B^T$	$n^2\left(1-\frac{1}{m}\right)$	$\frac{n^2}{m}$
$A^T \star B^T$	0	$\frac{n^2}{m}$

Tabelle 8.4: Übersicht der Kostenfunktionen für elementweise Operationen $\star \in \{+,-,\cdot,/\}$ bezüglich zeilenweiser, spaltenweiser und Block-Partitionierung.

Zusammenfassung

Aus den Tabellen 8.3 und 8.4 ist zu erkennen, dass alle Partitionierungsstrategien die Fließkommaoperationen generell gleichmäßig auf die Berechnungsknoten aufteilen. Für den Fall von Skalarprodukten und Matrix-Vektor-Multiplikationen sind effiziente Strategien für zeilenweise Partitionierungen etabliert worden. Ferner ist die Blockpartitionierung generell für alle Formen von dicht besetzten Matrizenmultiplikationen zu bevorzugen. Trotz der allgemein günstigeren Kommunikationskosten ist die Wahl einer optimalen Partitionierung offensichtlich von den zu berechnenden Algorithmen abhängig. Im Falle der Verfahren der Hidden-Markov-Modelle aus Kapitel 5.2 werden beispielsweise häufig konstruktionsbedingt Spalten der Beobachtungsmatrix selektiert und elementweise weiter verarbeitet. Dies würde etwa eine zeilenweise Partitionierung

im Sinne der Intraoperatorparallelisierung begünstigen.

8.1.3 Dünn besetzte Probleme

Nach der Betrachtung von Partitionierungsstrategien dicht besetzter Matrizen und Vektoren sollen in diesem Abschnitt jene für dünn besetzte quadratische Matrizen untersucht werden. Die quadratische Form (vergleiche etwa die Transitionsmatrix einer Markov-Kette oder Steifigkeitsmatrizen von Finite-Elemente-Methoden [137]) erhalten viele solcher Matrizen, da sie Korrelationseigenschaften von Objekten, Zuständen, Ereignissen oder anderen Charakteristika eines Modells repräsentieren. Zudem werden klassischerweise sehr große Dimensionen n mit nur wenigen Tupeln (Nicht-0-Elemente) pro Zeile genutzt. Im Beispiel 3 wird etwa eine quadratische Transitionsmatrix — die Hyperlink-Matrix — genutzt, um Informationen über per Weblinks miteinander verbundene Webseiten zu modellieren. Die Matrix besitzt mehr als 10 Milliarden Zeilen und durchschnittlich nur zwischen 10 und 15 Einträgen pro Zeile.
Die folgenden Betrachtungen konzentrieren sich vor allem auf den Fall der dünn besetzten Bandmatrizen. Dies sind Matrizen deren Nicht-0-Elemente nahe der Diagonalen geclustert sind. Es beschreibt hierbei

$$p(A) = \max_{i,j}(\{|i-j| \mid a_{ij} \neq 0\})$$

die Bandbreite einer Matrix A, für die in diesem Fall typischerweise $p(A) \ll n$ gilt. Die Bandform lässt sich etwa durch Nachbarschaftsbeziehungen der jeweiligen modellierten Charakteristika (etwa Zustände) interpretieren. Eine Bandform mit möglichst kleiner Bandbreite ist hierbei in vielen Fällen wünschenswert. So wird diese etwa bei der expliziten Lösung dünn besetzter linearer Gleichungssysteme benötigt, um das Einfügen von Nicht-0-Werten („Fill-Ins“) während des Lösungsprozesses zu vermeiden. In dem hier betrachteten Kontext kann die Struktur genutzt werden, um das Potenzial lokaler Berechnungen in verteilten Szenarien zu erhöhen.
Da es offensichtlich nicht in jedem Szenario möglich ist, den Aufbau von Matrizen so vorauszuplanen, dass die entstandene Matrix eine minimale Bandbreite besitzt, existieren viele heuristische Verfahren, die zu einer Reduktion dieser führen. Ein Stellvertreter dieser ist das *Cuthill-McKee-Verfahren* [138], welches im Anhang B.4.3 näher vorgestellt wird. Das Verfahren basiert maßgeblich auf Mengenoperationen und kann aus diesem Grund vergleichsweise einfach in SQL überführt werden. Andere Verfahren, wie die spektrale Partitionierung aus [139], beruhen auf Eigenvektoranalysen und können prinzipiell auch mit einer modifizierten Potenzmethode (beschrieben in Beispiel 3) in SQL umgesetzt werden.

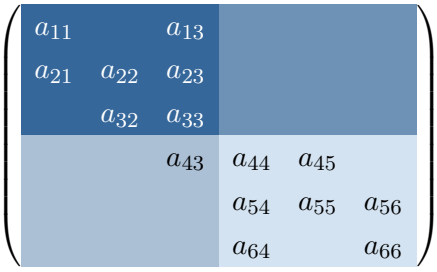
Partitionierung dünn besetzter Matrizen

Liegt nun eine vergleichsweise kleine Bandbreite vor, ist zunächst leicht ersichtlich, dass eine Blockpartitionierung im Allgemeinen zu einer stark unbalancierten Datenlast führt, da etwa auf

Nicht-Diagonalknoten nur wenig Tupel gespeichert werden. Als Beispiel soll etwa die Matrix

$$A = \begin{pmatrix} a_{11} & & a_{13} & & & \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & & & \\ & & a_{43} & a_{44} & a_{45} & \\ & & & a_{54} & a_{55} & a_{56} \\ & & & a_{64} & & a_{66} \end{pmatrix}$$

betrachtet werden. Eine Blockpartitionierung auf einem 2×2-Gitter könnte in diesem Fall wie folgt visualisiert werden:



führen, wobei etwa der zweite Knoten $\xi(1,2)$ kein Element beinhaltet und der dritte $\xi(2,1)$ nur das Tupel `(4,3,`a_{43}`)`. Dieser Effekt würde bei hohen Dimensionen offensichtlich extremer ausfallen. Die variable und strukturierte Zeilenbesetzung erschwert hierbei eine einfache Kostenanalyse über verschiedene Strategien, wie Sie etwa im vorigen Abschnitt für dicht besetzt Probleme umgesetzt worden ist. Aus diesem Grund wird in diesem Abschnitt, mit Fokus auf datenbankorientierte Strategien, nur die bereits im vorigen Abschnitt genutzte eindimensionale Bereichspartitionierung (über Zeilen) untersucht. Hierbei werden keine Kostenanalysen vorgenommen, sondern eine sinnvolle Verteilung der Daten diskutiert.

Der Fokus liegt hierbei vor allem auf der effizienten Umsetzung von Big-Data-Matrix-Vektor-Multiplikationen $A\mathbf{x}$. Dabei wird auch beachtet, dass durch die große Dimension n es, im Gegensatz zu den meisten dicht besetzten Problemen, sinnvoll oder nötig ist, den Vektor $\mathbf{x}$ zu partitionieren, da seine Größe auch Hauptspeicher- und Cachegrößen übersteigen kann. Ähnlich zum Fall dicht besetzter Matrix-Vektor-Multiplikationen ist es vorteilhaft, wenn dicht besetzte Vektoren nach der gleichen Verteilungsfunktion partitioniert werden. Dies hat sich zum einen für Skalarprodukte im Kontext dünn besetzter Probleme bewährt und führt zum anderen für Matrix-Vektor-Multiplikationen mit Bandmatrizen zu hohen lokalen Berechnungsmöglichkeiten.

Dies kann etwa mit der Beispielmatrix A wie folgt visualisiert werden:

$$\begin{pmatrix} a_{11} & & a_{13} & & & & \\ a_{21} & a_{22} & a_{23} & & & & \\ & a_{32} & a_{33} & & & & \\ & & a_{43} & a_{44} & a_{45} & & \\ & & & a_{54} & a_{55} & a_{56} & \\ & & & a_{64} & & a_{66} & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}x_1 + a_{13}x_3 & \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 & \\ a_{32}x_2 + a_{33}x_3 & \\ x_3 & a_{43} + a_{44}x_4 + a_{45}x_5 \\ & a_{54}x_4 + a_{55}x_5 + a_{56}x_6 \\ & a_{64}x_4 + a_{66}x_6 \end{pmatrix}$$

$$=: \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{pmatrix}.$$

Mit dieser Strategie wird ein hohes Maß an lokaler Berechnung und lokalem Schreiben von Ergebnissen ermöglicht. Es muss nur das Tupel `(3,`x_3`)` kommuniziert werden. In Praxisbeispielen mit wesentlich größeren Bereichen (Zeilen) pro Partitionierung werden hierbei (bei moderaten Bandbreiten) viele Elemente des Ergebnisvektors komplett lokal berechnet werden können. Im Fall nicht-reduzibler Matrizen wird jedoch im Allgemeinen vor allem durch „Randzeilen“ stets Kommunikation induziert. Ein essenzieller Aspekt ist hierbei die Bereichsstruktur im Vergleich zur beliebigen Zeilenanordnung (etwa Hash-Partitionierung) im obigen Abschnitt. Die Anzahl der Zeilen pro Knoten kann gleichmäßig gewählt werden, wenn die durchschnittliche Anzahl der Zeilenelemente nicht stark variiert. Dies impliziert damit auch eine ungefähre Gleichverteilung der lokalen Berechnungskosten. Bei starker Variation der Zeilenbesetzung sind Strategien, die das Aufteilen der Gesamtelemente pro Partitionierung fokussieren, nötig. Bei der Evaluation in Abschnitt 9.1 wird hier von einer gleichmäßigen Verteilung der Zeilen ausgegangen. Wie sich zeigen wird, ist diese Strategie aufgrund ihrer Einfachheit gut geeignet für die Umsetzung auf parallelen Datenbanksystemen in SQL. Da der Großteil der Daten im Allgemeinen lokal vorliegt, werden vergleichsweise wenig Zwischenergebnisse berechnet. Da die Bandstruktur in den meisten Fällen dafür sorgt, dass Knoten k nur Elemente von Knoten $k-1$ oder $k+1$

benötigt, ist es vergleichsweise einfach eine kommunikationsarme Strategie zu entwickeln. Wird vom System erkannt, dass in diesen Fällen nur eine Kommunikation der jeweiligen Einträge von $\mathbf{x}$ auf den Knoten k nötig ist, kann die Aggregation komplett lokal (ohne distributive Ansätze) geschehen. Inwiefern dies umgesetzt wird, ist abermals implementationsabhängig. Es kann jedoch zweifelsfrei in gängigen Implementationsmodellen umgesetzt werden.

8.2 Intraoperatorparallelität durch Anfragezerlegungen

Im Folgenden werden die, in den vorigem Abschnitt 8.1 etablierten, Partitionierungsstrategien als Basis genutzt, um einen einfachen aber effektiven Ansatz zur Intraoperator-Parallelisierung der fundamentalen Lineare-Algebra-Operatoren zu ermöglichen. Hierbei wird eine ursprüngliche Anfrage eines Lineare-Algebra-Operators Q in Unteranfragen $Q_1, \ldots, Q_m$ aufgeteilt, die von einer entsprechenden Schnittstelle (vgl. Abbildung 6.1) durch m unabhängige Prozesse über verschiedene Datenbanknutzerkonten an das Datenbanksystem gesendet werden. Hierdurch soll eine bessere Verteilung von Rechen- und Kommunikationslast ermöglicht werden. Prämisse hierfür ist, dass klassische SQL-Anfragepläne und deren Operationen im Allgemeinen nicht optimal beziehungsweise nicht ausgelegt für die Verarbeitung von Lineare-Algebra-Operationen ist. Die Optimierungsschnittstellen streben effiziente Umsetzungen von Gruppierungen, Verbunden, Selektionen und ähnlichem an. Eine Optimierung im Sinne der Struktur von Matrix/Vektor-Relationen und -Operationen ist (mit Einschränkungen), wie sich zeigen wird, unwahrscheinlicher. Zudem ist zu bedenken, dass (parallele) Datenbanksysteme für die simultane, unabhängige Anfrageverarbeitung mehrerer Nutzer ausgelegt ist. Das heißt, dass Strategien der Anfrageverarbeitung nicht zwingend auf die effizienteste Verarbeitung einer einzelnen Anfrage ausgelegt sein muss, sondern evtl. auch auf den allgemeinen Durchsatz mehrerer paralleler Anfragen optimiert ist. Hierbei ist die eigentliche Konfiguration des PRDBS und die Verteilung der Ressourcen der einzelnen Knoten auch entscheidend. So konnte bei der Evaluierung (vgl. Abschnitt 9.1) von Matrizenmultiplikationen in Postgres-XL beobachtet werden, dass im worst-case der Verbund `a` $\bowtie$ `b` und die darauffolgenden Aggregationen auf einem einzelnen Knoten umgesetzt wurden. Das System führt hierbei wahrscheinlich bedingt durch die „Sicht“ auf klassische Anfrageverarbeitungen, trotz der Nutzung von Statistiken, nicht optimal die Matrizenmultiplikation durch. Generell ist eines der kritischen Probleme, dass die Wahl von Berechnungsknoten für Teilanfragen und finaler Aggregation transparent für den Nutzer ist und nach den hier präsentierten Studien auch oftmals nicht optimal (in Postgres-XL) gewählt wird. Daher ist es eines der wesentlichen Ziele, durch eine gezielte Anfragezerlegung dem Datenbanksystem und deren Optimierern zu einer effizienten Wahl von Berechnungsknoten und besseren Anfragestrategie zu leiten. Hierfür werden aus den ursprünglichen SQL-Anfragen Unteranfragen erstellt, die nur Untermengen von Partitionen nutzen, sodass die Knotenwahl „offensichtlich“ ist. Zudem werden durch die simultane

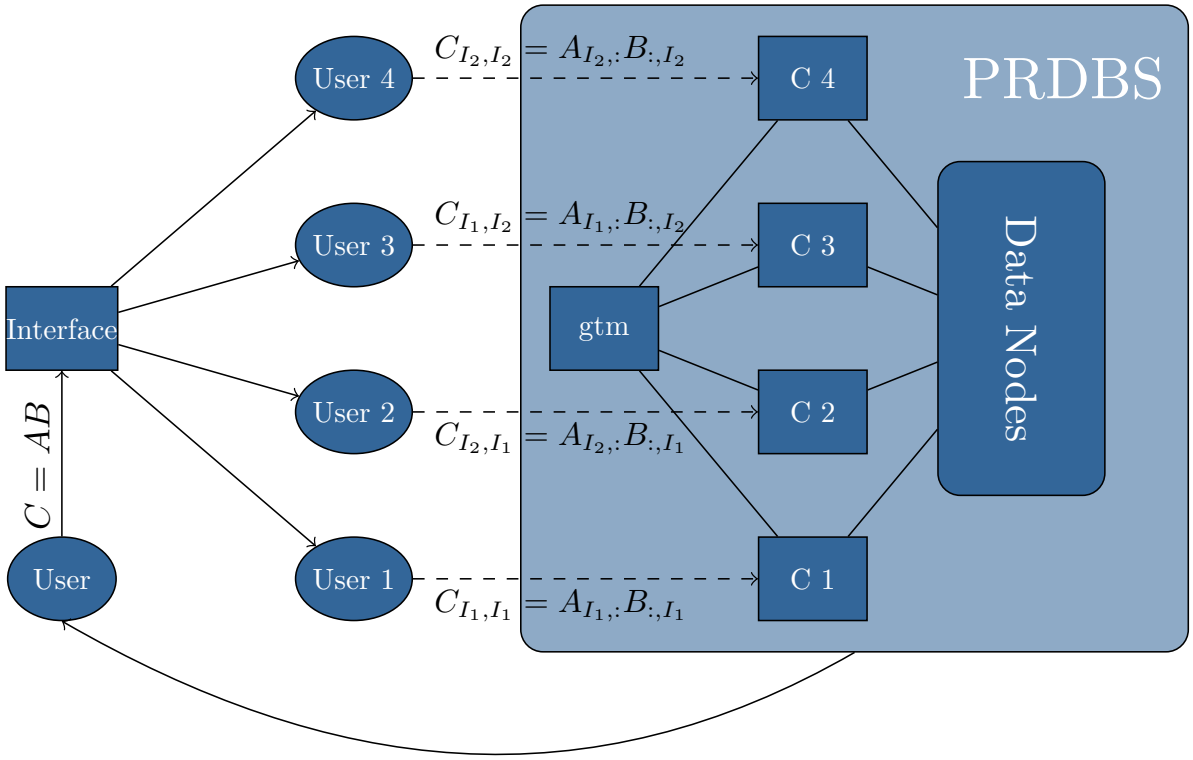


Abbildung 8.11: Intraoperator-Parallelisierung durch Anfragezerlegung mit 4 Prozessen auf einem PRDBS (Architektur angelehnt an Postgres-XL aus Abschnitt 4.3.3) und mindestens 4 Berechnungsknoten. Die etablierte Schnittstelle berechnet das Matrizenprodukt $C = AB$ der blockpartitionierten Matrizen A, B durch die parallel gestellten zweidimensionalen Bereichsanfragen für $C_{I_j,I_k} = A_{I_j,:}B_{:,I_k}$.

Anfrageverarbeitung mit jeweils günstiger Berechnungsknotenwahl eine Auslastung aller Knoten motiviert, welches wie beschrieben teilweise bei einfachen Anfragen nicht beobachtbar war.

Für die etablierten fundamentalen Operatoren aus Tabelle 5.2 ist die Aufteilung der SQL-Anfragen vergleichsweise einfach und universell (maßgeblich durch Bereichsanfragen) umsetzbar. Im Folgenden wird dies etwa am, in Abbildung 8.11 dargestellten, Beispiel einer Matrizenmultiplikation $C = AB$ mit 2×2-blockpartitionierten Matrizen A und B verdeutlicht.

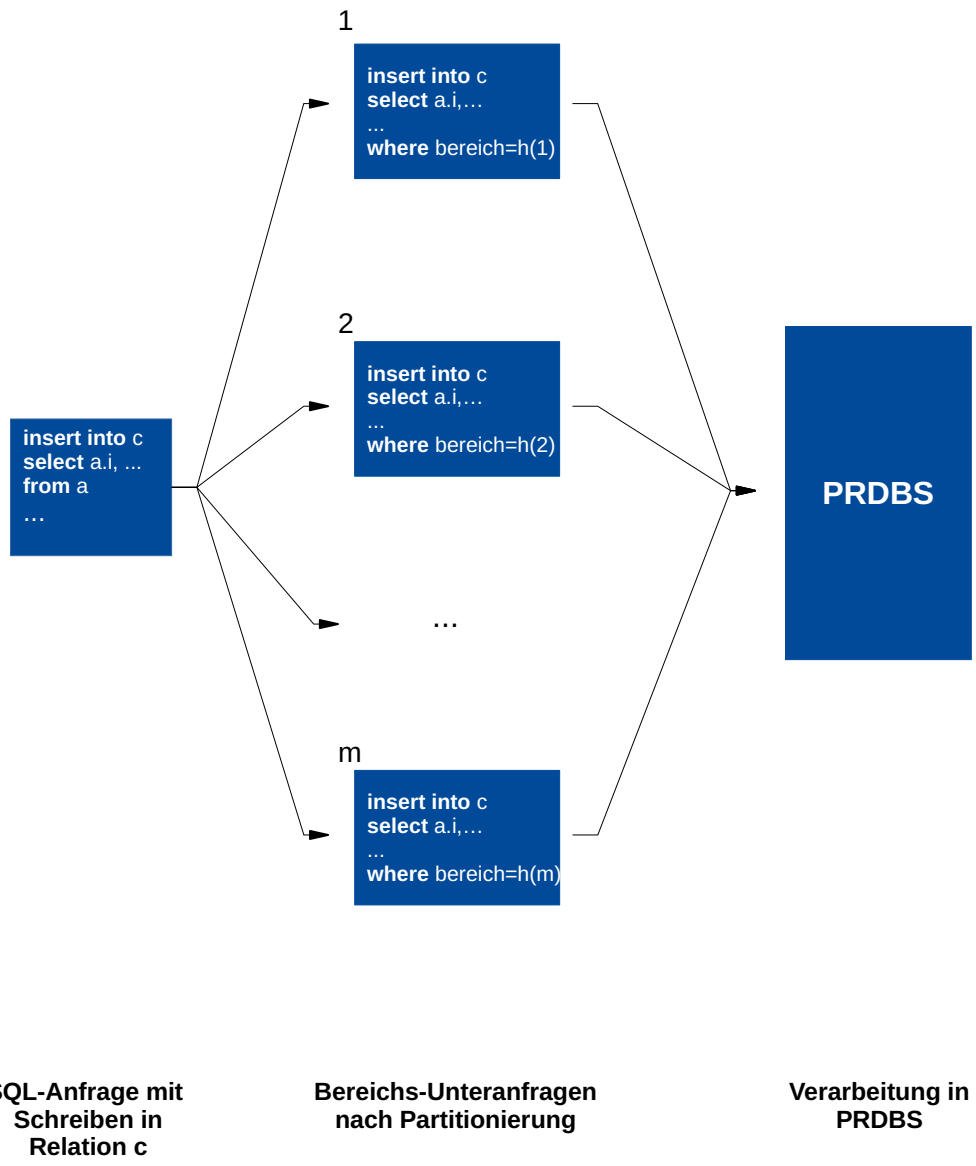


Abbildung 8.12: Konzept zur Anfragezerlegung: Aufteilen ursprünglicher SQL-Anfragen in Subanfragen (meist Bereichsanfragen) bezüglich der zugehörigen Partitionierungsfunktion $h(k)$.

Für die Ergebnis-Blockmatrizen $C_{i,j}$ ergibt sich hierbei

$$\begin{aligned}
&\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \\
&= \begin{pmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{pmatrix} \\
&= \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2}. \end{pmatrix}
\end{aligned}$$

In SQL kann dies beispielsweise für den ersten Knoten $\xi(1,1)$ durch die Teilanfrage

```
insert into c
select a.i, b.j, sum(a.v * b.v)
from a join b on a.j=b.i
where a.i between 1 and $n/2 and a.j between 1 and $n/2
group by a.i,b.j
```

umgesetzt werden. Es werden in diesem Fall alle Elemente der Knoten 1, 2 und 3 benötigt, wobei das Ergebnis auf Knoten 1 geschrieben wird. Eine Wahl des ersten Knotens durch das PRDBS zur Berechnung ist nicht garantiert, jedoch wahrscheinlicher. Wie sich in Abschnitt 9.1 zeigen wird, bringt diese Art der Anfrageformulierung deutliche Performance-Vorteile gegenüber hash-verteilten Ansätze ohne Anfragezerlegung in Postgres-XL. Eine Verallgemeinerung der Zerlegung von Anfragen in Bereichsanfragen bezüglich der genutzten Partitionierungsfunktion h ist in Abbildung 8.12 dargestellt. In diesem ist neben der Blockmatrixpartitionierung auch die Blockzeilen- und Blockspaltenpartitionierung einfach umsetzbar. Die Äquivalenz der Ergebnisse von Q und $\bigcup_{i=1}^{m} Q_i$ ist in diesem Fall unter Abdeckung aller Ergebniselemente trivialerweise identisch. Dieser Ansatz ist jedoch für Round-Robin und allgemeine Hash-Partitionerung über Zeilen- und Spaltenindizes problematisch. Hierbei könnten in künftigen Untersuchungen komplexere Anfragezerlegungsansätze über **where exists**-Klauseln getestet werden.
Die Aufteilung in Unteranfragen kann prinzipiell auch noch detaillierter geschehen. In diesem konkreten Fall könnte etwa die Selektion einzelner Blockmatrizen und den benötigten Submatrixprodukten durch Bereichsanfragen umgesetzt werden und so die eigentlich distributive verteilte Strategie der (Matrixelement-)Summen erzwungen werden. Die Umsetzung hierfür benötigt jedoch deutlich mehr simultane Unteranfragen und auch mehr Synchronisation zwischen den aktiven Berechnungsknoten. In den Untersuchungen mit Postgres-XL sind zudem teils lange Wartezeiten auf freie Verbindungsprozesse aus dem Connection-Pool aufgetreten. Dies konnte durch Konfigurierung der Garbage-Collection so eingedämmt werden, dass die obige Bereichsanfragezerlegung stabil ist, führt aber trotzdem bei großer Anzahl von Anfragen zu fragilem

Verhalten. Dieser Aspekt ist wahrscheinlich systemspezifisch, sodass fein granuliertere Anfragekonzepte auf anderen parallelen Datenbanksystemen in Zukunft getestet werden könnten.

Diskussion und Zusammenfassung

Die vorgestellte Methodik der Anfragezerlegung bezüglich bereichspartitionierter Vektoren und Matrizen ist ein simpler und effizienter Ansatz, der durch seine Umsetzung im standardisierten SQL nahezu universell in modernen parallelen relationalen Datenbanksystemen eingesetzt werden kann. Die in Abschnitt 9.1 dargestellten Experimente zeigen, dass durch die Nutzung effizienter Partitionierungsstrategien von Matrizen und Vektoren, parallele Datenbanksysteme (in diesem Fall Postgres-XL) in ihrer klassischen datenbankorientierten Anfrageverarbeitung deutlich beschleunigt werden können. Durch die einfache Natur der Umsetzung ist die Methodik vergleichsweise einfach auch auf inhomogene Cluster (Berechnungsknoten mit variablen Ressourcen) oder Multidatenbanksysteme erweiterbar (etwa durch Anpassung der Partitionsgrößen).
Trotz dieser Vorteile ist die Anfragezerlegung nicht unkritisch zu betrachten. Eine der wohl wesentlichen Einschränkung entsteht durch die große Variation von Architekturen, Funktionalitäten und SQL-Schnittstellen von parallelen Datenbanksystemen. Beispielsweise ist die Umsetzung der Partitionierung von Relationen in Postgres-XL problematisch, da hier nur Hash-Verteilungen und Modulo-Verteilungen bezüglich eines Attributs, Round-Robin und volle Replikation (Stand: 12/2022) unterstützt werden. In dem parallelen Datenbanksystem Actian VectorH hingegen wird ausschließlich Hash-Partitionierung unterstützt, wobei jedoch multiple Attribute genutzt werden können. Zusätzlich unterscheidet sich die Syntax für Verteilungsstrategien in den Systemen deutlich. Damit ist eine komplett systemunabhängige Entwicklung der Zerlegungsschnittstelle, im Vergleich zu den Übersetzungsprozessen in universell berechenbaren ANSI-SQL-Anfragen, nicht möglich. Das Fehlen entscheidender Partitionierungsstrategien fordert hierbei workarounds, wie die Modellierung von Blockmatrizen durch vererbte Relationen oder gezieltes Einfügen von Matrixelementen durch Round-Robin-Verteilungen. Dies zeigte hierbei jedoch entweder unerwünschtes Verhalten des PRDBS oder funktionelle Einschränkungen des Ansatzes. Beide Fälle werden in Abschnitt 9.1 näher diskutiert.
Selbst wenn die Partitionierungsstrategien unterstützt werden, ist durch die transparente Parallelisierung in PRDBS keine Garantie gegeben, dass die Systeme diejenigen Knoten zur Aggregierung nutzen, die die Kommunikationskosten minimieren. Zusätzlich kann die Aufteilung von ganzen Verfahren des wissenschaftlichen Rechnens in viele einzelne Anfragen, die zusätzlich in Unteranfragen aufgeteilt werden, kontraproduktiv im Sinne der physischen und logischen Optimierungsmöglichkeiten von Datenbanksystemen sein. Hierbei sei etwa auf die vorteilhafte Schachtelung von Verfahren aus Abschnitt 7.3 verwiesen, welche in diesem Ansatz in deutlich weniger Szenarien aufgrund der nötigen Synchronisierung eingesetzt werden kann.
Unter anderem aus diesen Gründen ist ein Push-Down der Anfragezerlegung in die Anfrage-

verarbeitung des Datenbanksystems im Allgemeinen effizienter. Die kommerzielle Umsetzung und Verbreitung ist jedoch unwahrscheinlich, aufgrund der aus Sicht relationaler Datenbanken unbekannten Matrix- und Vektorstrukturen und deren Operationen. Eine Fortführung der Entwicklung und Untersuchung von „special-purpose“-Datenbanken oder nicht-standardisierten Erweiterungen, wie sie in Abschnitt 4.4 diskutiert werden, ist hier wahrscheinlicher. In diesen Fällen sind Langlebigkeit, Verfügbarkeit (etwa aus Kostengründen) oder Portabilität eingeschränkt. Die Nutzung der Anfragezerlegung in geeigneten parallelen Datenbanksystemen bleibt demnach ein nützliches Werkzeug zur Performance-Verbesserung entsprechender Anfragen.

Kapitel 9

Evaluation

In diesem Kapitel werden zunächst die in Kapitel 8 entwickelten Partitionierungs- und Anfragestrategien für parallele relationale Datenbanksysteme experimentell ausgewertet. Die Ergebnisse werden daraufhin durch einen Vergleich mit dem Big-Data-System Apache Spark eingeordnet und diskutiert. Abgeschlossen wird das Kapitel durch die Beschreibung SQL-basierter Fourier-Transformationen, Zeitreihenanalysen im Automotive-Kontext und Vorhersagen für Routingszenarien im maritimen Bereich. Die dort vorgestellten positiven Ergebnisse bestätigen den Nutzen des in Kapitel 3 vorgestellten Frameworks und motivieren die Ausweitung dessen Anwendungsfeldes.

9.1 Fundamentale Operationen mit Anfragezerlegung

In diesem Abschnitt wird die Anfragezerlegung bezüglich Anfragen von Lineare-Algebra-Operationen aus Kapitel 8 evaluiert. Hierfür werden mehrere Experimente im parallelen relationalen Datenbanksystem Postgres-XL aus Abschnitt 4.3.3 ausgewertet. Trotz der vergleichsweise schwachen Performance des zugrundeliegenden row stores PostgreSQL, wurde Postgres-XL 9.5 R1.6 [86] aus mehreren Gründen als Stellvertreter gewählt. Hierzu zählen maßgeblich der Open-Source-Charakter, die vergleichsweise hohe Funktionalität von PostgreSQL und dessen Nähe am SQL-Standard. Im Folgenden werden nach einer Vorstellung der genutzten Cluster- und Systemkonfiguration drei Experimente ausgewertet, welche erstmalig in [10] präsentiert worden sind. Das erste Experiment untersucht den Einfluss der Partitionierung von Matrizen auf die Laufzeit von Matrizenmultiplikationen ohne Anfragezerlegung. Die darauf folgenden Experimente testen die etablierte Anfragezerlegung für dicht besetzte Matrizenmultiplikation und dünn besetzte Matrix-Vektor-Multiplikationen.

Cluster-Setup und Postgres-XL-Konfiguration

Für die folgenden Tests wurden 5 Berechnungsknoten und ein Gateway-Knoten genutzt. Die Hardware-Spezifikationen der Berechnungsknoten sind in Tabelle 9.1 aufgelistet. Wie in Abschnitt 4.3.3 beschrieben wurde, arbeitet Postgres-XL mit den drei wesentlichen Komponenten:

- Global Transaction Managers (GTMs),
- Coordinators und
- Datanodes.

Die Daten liegen in den Datanodes, die zudem den Großteil der Anfrageverarbeitung umsetzen und aus diesem Grund die meisten Berechnungsressourcen benötigen. Die Coordinators koordinieren, dem Namen entsprechenend, Anfragen auf dem Cluster im Sinne der Datenlokalität und Lastbalancierung. Zusätzlich werden knotenübergreifende parallele Aggregation teilweise in Coordinators umgesetzt. Der GTM und deren Proxys setzen die Multiversion Concurrency Control (MVCC) um und benötigen im Allgemeinen wenig Rechenressourcen.
Wie in Abbildung 9.1 dargestellt, wurden pro Berechnungsknoten (mit zwei Kernen) jeweils zwei Datanodes und einen Coordinator genutzt. Zusätzlich wurde jeweils ein GTM-Proxy genutzt, um die Last des GTMs zu reduzieren. Der GTM lief hierbei auf einem Extra-Knoten. Die Wahl des Setups ist aus Ausführungen der offiziellen Postgres-XL-Dokumentation [140] motiviert und soll eine gute Lastbalancierung und Datenlokalität bieten.
Da, analog zu PostgreSQL, die Standardkonfigurationen der Coordinator und Datanodes sehr konservativ sind und für leichtgewichtige Hardware ausgelegt worden sind (vgl. Ausführungen in [141]), wurden ausgewählte Parameterwerte geändert. Die Rekonfiguration ist in den Tabellen 9.2 und 9.3 dargestellt. Die Einstellungen sind hierbei nach Empfehlungen von Entwicklern und Nutzern getroffen (vgl. [141]) und wurden nicht anhand von Kriterien systematisch optimiert. Im Allgemeinen ist eine effizienzmaximierende Einstellung des Systems, aufgrund der großen Anzahl an teilweise einander beeinflussenden Parametern und möglicher verschiedenen Nutzungsarten (etwa Anfrage-Durchsatz gegen Latenzminimierung), nur sehr schwer erreichbar und könnte Bestandteil künftiger, darauf aufbauender Untersuchungen sein.
Mit der Etablierung des Cluster- und PRDBS-Setups werden nun die einzelnen Experimente näher beschrieben. Für jede der grafischen Auswertungen wurde der Durchschnitt aus drei Läufen genutzt. Im Anhang B.5 ist zudem der genutzte Code zur Erstellung von Matrizen und Vektoren aufgeführt.

Einfluss der Datenpartitionierung auf Matrizenmultiplikationen

Vor der eigentlichen Auswertung der Anfragezerlegung ist zunächst untersucht worden, inwiefern Postgres-XL einen Teil der in Abschnitt 8.1 diskutierten Strategien der Datenpartitionierung von

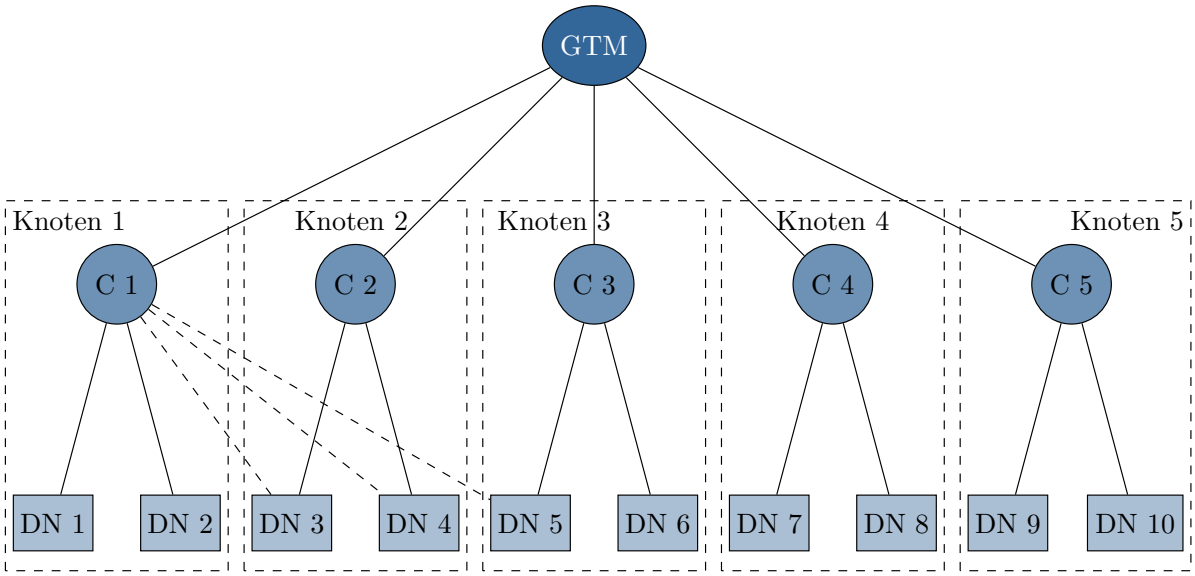


Abbildung 9.1: Genutztes Setup von Postgres-XL. Der Global Transaction Manager (GTM) arbeitet auf einem Gateway-Knoten mit einem Coordinator für externe Kommunikation (nicht in Abbildung dargestellt). Die Coordinators (C i) sind mit jedem Datanode (DN i) verbunden. Dies ist aus Gründen der Übersichtlichkeit nur für C 1 angedeutet. Auf jedem Knoten sind zusätzlich GTM-proxy-Knoten eingerichtet.

Parameter	Wert
Betriebssystem	CentOS 7
Prozessor	2 × 1.9GHz
Cache	2 × 4 MB
RAM	16 GB DDR4 (2133 Mhz)
Sekundärspeicher (Daten)	1 TB
Sekundärspeicher SSD (BS)	50 GB
Sekundärspeicher SSD (Verarbeitung Temporärer Tabellen)	20 GB

Tabelle 9.1: Hardware-Spezifikationen eines Berechnungsknoten mit jeweils zwei Datanodes und einem Coordinator.

Parameter	Value
shared_buffers	1GB
effective_cache_size	3GB
work_mem	52428kB
maintenance_work_mem	512MB
min_wal_size	4GB
max_wal_size	8GB
checkpoint_completion_target	0.9
wal_buffers	16MB
default_statistics_target	500
random_page_cost	4
max_pool_size	100
max_connections	400
autovacuum_vacuum_scale_factor	0.01
autovacuum_vacuum_cost_limit	1000

Tabelle 9.2: Veränderte Parameter der lokalen PostgreSQL-Coordinator-Instanzen.

Parameter	Value
shared_buffers	1536MB
effective_cache_size	4608MB
work_mem	78643kB
maintenance_work_mem	768MB
min_wal_size	4GB
max_wal_size	8GB
checkpoint_completion_target	0.9
wal_buffers	16MB
default_statistics_target	500
random_page_cost	4
max_pool_size	100
max_connections	400
autovacuum_vacuum_scale_factor	0.01
autovacuum_vacuum_cost_limit	1000

Tabelle 9.3: Veränderte Parameter der lokalen PostgreSQL-Datanode-Instanzen.

Matrizen und Vektoren transparent umsetzen konnte.
Hierfür wurden die zeilen- beziehungsweise spaltenweise und blockweise Partitionierung aus Abschnitt 8.1 bezüglich einer einfachen dicht besetzten Matrizenmultiplikation AB von Matrizen $A, B \in \mathbb{R}^{n \times n}$ untersucht. Es wurde für alle getesteten Ansätze entsprechend die etablierte SQL-Anfrage

```
select A.i,B.j,sum(A.v*B.v)
from A join B on A.j=B.i
group by A.i,B.j
```

genutzt.
Zur Umsetzung in Postgres-XL wurden zwei wesentliche Methoden untersucht. Die erste Möglichkeit ist die Nutzung der Postgres-XL-spezifischen **distribute by**-Erweiterung des **create table**-Statements [140], welches Tupel transparent auf den Datanodes verteilt. Hierbei kann die Verteilung wahlweise

- mittels hash über einem Attribut,
- mittels modulo über einem Zahlenattribut (Integer oder Datumstyp),
- per Round-Robin oder
- durch Replikation

geschehen. Diese Methode kann als primärer Ansatz zur balancierten Datenverteilung in Postgres-XL bezeichnet werden, da hier insbesondere die Anfrageverarbeitung und -optimierung am effizientesten funktionierte. Jedoch zeigt sich hierbei nachteilig, dass nicht die gewünschte ein- oder zweidimensionale Bereichsverteilung unterstützt wird. Um trotzdem die Blockpartitionierung nutzen zu können, wurde der **distribute by**-Ansatz mittels Round-Robin umgesetzt, wobei die zugehörigen Tupel in der entsprechenden Reihenfolge geordnet eingefügt wurden. Dies ermöglicht zwar die gewünschte Datenverteilung, führt im Allgemeinen jedoch zu einer nötigen Repartitionierung von Ergebnissen, insofern diese ebenfalls blockpartitioniert gespeichert werden sollen. Dieser Schritt wurde im Folgenden nicht berücksichtigt.

Als zweiter Ansatz zur Erstellung der blockweisen Matrixpartitionierung wurde die Vererbung von Relationen mittels `inherits`-Klausel aus Abschnitt 4.1.3 getestet. Hierfür wurde eine Master-Relation genutzt und erbende (dem Blockmatrix-Gitter entsprechend nummerierte) Kinder-Relationen erstellt (vergleiche Anhang B.5). Jede Kind-Relation wurde direkt einem Datanode zugeordnet und durch eine Reihe von Integritätsbedingungen sichergestellt, dass nur Matrixeinträge mit korrekt zugehörigen Indizes eingefügt werden können. Zusätzlich wurden Regeln etabliert, die das Einfügen in die Master-Relation auf die jeweilige Kinder-Relation umleiten. Durch diese Methodik ist es verhältnismäßig einfach, Daten einzufügen und zu selektieren, aber auch beliebige andere Partitionierungsstrategien zu etablieren. Zudem kann (wie im Folgeexperiment belegt) in diesem Fall die blockweise Matrizenmultiplikation direkt über das Ansprechen der nötigen Kind-Relationen umgesetzt und so die gewünschte Anfrageverarbeitung erzwungen werden. Details zu der Implementation sind im Anhang B.5 bereitgestellt.

In Abbildung 9.2 sind die Laufzeitergebnisse der beiden Ansätze grafisch dargestellt, wobei der Vererbungsansatz („Block Master Child“) nur für die Blockpartitionierung genutzt wurde. Die Relationen wurden auf 9 der 10 Datanodes gespeichert und die Blockansätze gemäß eines 3×3-Knotengitters verteilt. In der Grafik ist zu erkennen, dass die Laufzeit des Vererbungsansatzes deutlich langsamer als die der anderen `distribute by`-Strategien ist. Eine Analyse der Anfragepläne hat gezeigt, dass es Postgres-XL nicht möglich war, die Partitionierung, trotz expliziter Integritätsbedingungen und Statistiken, auszunutzen. Gegensätzlich hierzu materialisiert Postgres-XL sogar die gesamte Master-Relation (bzw. die Vereinigung der Kinder-Relationen) auf einem einzelnen Knoten und führt die Anfrage dann lokal aus. Dies entspricht etwa dem *worst case*. Inwiefern dieses Verhalten sich in künftigen Versionen des vergleichsweise jungen Systems Postgres-XL verbessert wird, ist offensichtlich spekulativ. Eine nachträgliche Evaluation in späteren Versionen könnte jedoch, aufgrund der guten Umsetzungsmöglichkeiten beliebiger Partitionierungsmuster, sinnvoll sein.

Neben dem Vererbungsansatz ist zu beobachten, dass die durch *distribute by* umgesetzte Blockpartitionierung geringfügig, aber konsistent, schneller ist, als die per Hash-verteilten Zeilen- und Spaltenpartitionierungen (bei $n = 1800$ in etwa 13 % schneller). Letztere Ansätze unterscheiden

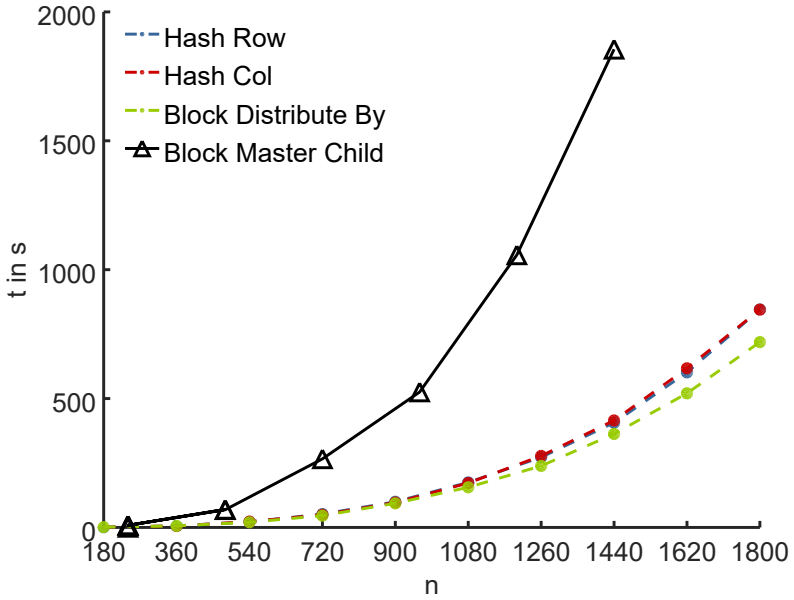


Abbildung 9.2: Ergebnisse einer dicht besetzten Matrizenmultiplikation AB mittels verschiedener Partitionierungsstrategien in Postgres-XL. Grafik erstmalig veröffentlicht in [10].

sich erwartungsgemäß nicht.
Aufgrund der wenigen Informationen bezüglich der parallelen Verarbeitung in Postgres-XLs einsehbaren Anfragepläne ist eine Analyse des vergleichsweise kleinen Unterschied der Block- und Hash-Partitionierungen schwierig. Es ist jedoch zu erkennen, dass Postgres-XL eine parallele Summierung gemäß der distributiven Definition 9 und den Ausarbeitungen in Abschnitt 8.1 in allen Fällen umsetzt.
Inwiefern die lokal optimale Blockmatrix-Multiplikation umgesetzt wurde, ist den Anfragepläne nicht zu entnehmen. Aufgrund der konsistenten, leichten Anfragebeschleunigung kann jedoch zumindest von einer Reduktion der Knoten-zu-Knoten-Kommunikation und deren Koordinierung ausgegangen werden. Eine Einordnung dieser Ergebnisse zu lokalen Berechnungen wird im nun folgenden Abschnitt vorgenommen.

Matrizenmultiplikation mittels Anfragezerlegung

Nachdem im ersten Experiment etabliert wurde, dass das ganze Potenzial der Blockmultiplikation vermutlich nicht genutzt werden konnte, wird im Folgenden eine Evaluation der Anfragezerlegung nach Kapitel 8 vorgestellt. Hierbei werden die vorherig etablierten Blockpartitionierungsansätze mittels Vererbung und **distribute by**-Verteilung genutzt, um durch das Aufteilen der Grundanfrage in zugehörige Teilanfragen und simultanem Senden dieser, eine Beschleunigung der Berechnung zu erlangen.

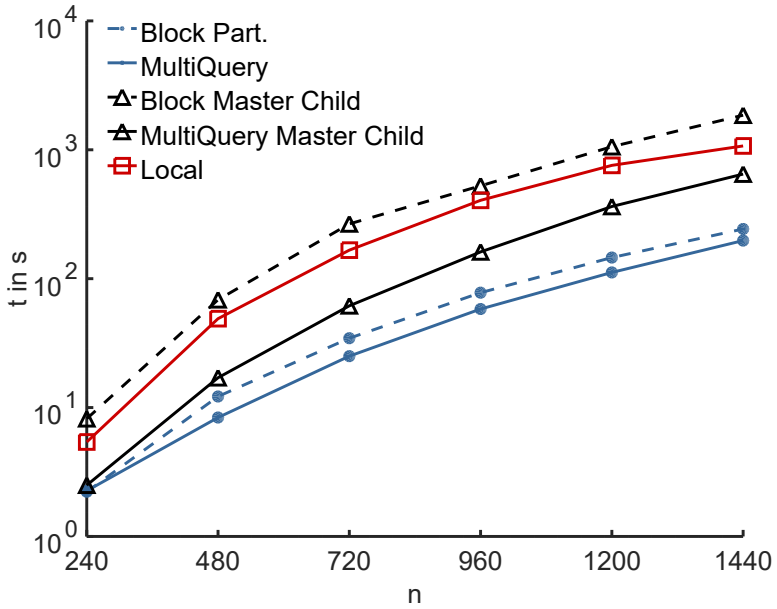


Abbildung 9.3: Ergebnisse von Anfragezerlegungsstrategien bezüglich dicht besetzter Matrizenmultiplikation in Postgres-XL. Grafik erstmalig veröffentlicht in [10].

Im Falle der **distribute by**-Verteilung wurden die jeweiligen SQL-Anfragen mit Bereichsbedingungen so erweitert, dass jede Anfrage

```
select A.i,B.j,sum(A.v*B.v)
from A join B on A.j=B.i
where A.i between min_ai and max_ai
   and B.j between min_bi and max_bi
group by A.i,B.j.
```

genau die Tupel bezüglich einer Ergebnisblockmatrix C_{ij} des ursprünglichen Produkts $C = AB$ berechnet. Im Falle des hier genutzten 3×3 Gitters wird im besten Fall auf dem zugehörigen Knoten (Datanode und lokal zugehörigem Coordinator) $\xi_3(i, j)$ (vgl. (8.4) aus Abschnitt 8.1.2), die Submatrix berechnet. Hierbei werden Daten aus jeweils vier weiteren Knoten/Datanodes benötigt. Prinzipiell ist eine noch detailliertere Auflösung der Blockmatrix-Produkte in deren Teilprodukte der Form $A_{ik}B_{kj}$ möglich und sinnvoll, um dem System die lokale Anfrageverarbeitung nahe zu legen. In der Praxis hat sich für Postgres-XL jedoch gezeigt, dass durch das parallele Senden zu vieler Anfragen die Verarbeitung deutlich verlangsamt wird. Dies ist maßgeblich auf eine Überbeanspruchung möglicher Verbindungen und interner Prozesse von Postgres-XL zurückzuführen (vgl. Beschreibung zum Connection-Pool in Abschnitt 4.3.3) und konnte auch nicht durch Rekonfiguration zugehöriger Parameter umgangen werden. Dieses Pro-

blem ist voraussichtlich systemspezifisch, so dass künftige Auswertungen auf anderen parallelen Datenbanksystemen sinnvoll sind.
Für den Ansatz mittels vererbter Relationen wurden ebenfalls 9 Anfragen gemäß den Ergebnisblockmatrizen C_{ij} formuliert. Hierbei wurden die jeweiligen Kind-Relationen/Blockmatrizen direkt verwendet. Eine Anfrage für C_{ij} besitzt dann die folgende Form:

```
select i,j, sum(v) from (
   select ai1.i as i, b1j.j as j, sum(ai1.v*b1j.v) as v
   from ai1 join b1j on ai1.j=b1j.i
   group by ai1.i,b1j.j
   union all
   select ai2.i as i, b2j.j as j, sum(ai2.v*b2j.v) as v
   from ai2 join b2j on ai2.j=b2j.i
   group by ai2.i,b2j.j
   union all
   select ai3.i as i, b3j.j as j, sum(ai3.v*b3j.v) as v
   from ai3 join b3j on ai3.j=b3j.i
   group by ai3.i,b3j.j
) temp
group by i,j
```

Die Ergebnisse der beiden Ansätze sind in Abbildung 9.3 dargestellt. Hierbei wurden neben den Ansätzen der Anfragezerlegung („Multi Query“) zu Vergleichszwecken auch die Ergebnisse mittels einfacher Anfrage aus dem ersten Experiment eingefügt. Zusätzlich wurden Laufzeitergebnisse der lokalen Berechnung in PostgreSQL 10.1 auf einem lokalen Notebook mit den Hardware-Spezifikationen 6.1 und der Rekonfiguration nach Tabelle 7.1 visualisiert. Der lokalen PostgreSQL-Version stehen hier zwar etwas mehr Rechenressourcen zur Verfügung als einem Datanode, die Setups ähneln sich jedoch in ihrer Größenordnung, sodass die Ergebnisse als ungefähre Abschätzung des Speed-Ups nutzbar sind.
In der zugehörigen Grafik ist erkennbar, dass der ursprüngliche Vererbungsansatz sogar deutlich langsamer als der lokale Ansatz ist, welches sich auf die 1-Knoten-Berechnung mit zusätzlicher Kommunikation zurückführen lässt. Die Anfragezerlegungsansätze sind konsistent schneller als deren Versionen, die nur eine Anfrage nutzen. Dies belegt entsprechend den Nutzen der Anfragezerlegung deutlich. Für den Vererbungsansatz konnte durch die Anfrageformulierung in etwa eine gesamte Ordnung gegenüber der ursprünglichen Version gewonnen werden, welches bezüglich der lokalen Versionen einen Speed-Up zwischen Faktor 3 und 4 bedeutet. Trotzdem ist diese Verarbeitungsversion langsamer als die ursprüngliche Blockmatrixmultiplikation mittels `distribute by`, welche bezüglich der lokalen Version zwischen 4 und 5.2 mal schneller ist. Durch die Anfrage-

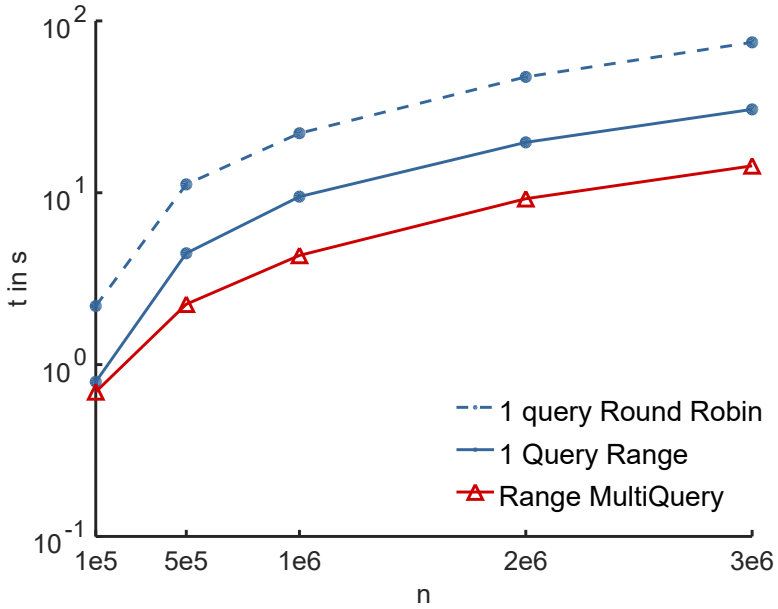


Abbildung 9.4: Ergebnisse der Anfragezerlegungsstrategien bezüglich dünn besetzter Matrix-Vektor-Multiplikation $A\mathbf{x}$ in Postgres-XL. Die Experimentalergebnisse wurden erstmalig in [10] vorgestellt[1].

zerlegung konnte der Faktor erhöht werden und liegt zwischen 5.5 und 7, welches vergleichsweise nahe an der theoretischen optimalen Obergrenze von 9 (mit 10 Datanodes) liegt (ohne Einberechnung der Kommunikationskosten). Inwiefern die gesamte Multiplikation optimal bezüglich der maximal erreichbaren Datenlokalität (vgl. Abschnitt 8.1) ist, ist aus den Anfrageplänen nicht zu erkennen. Jedoch kann aus den positiven Ergebnissen geschlossen werden, dass eine deutliche Reduktion der nötigen Interknotenkommunikation und deren Synchronisation benötigt wurde und eine relative Erhöhung der Berechnungslast der Clusterprozessoren erreicht werden konnte.

Dünn besetzte Matrix-Vektor-Multiplikation

Als letztes Experiment wurden dünn besetzte Matrix-Vektor-Produkte $\mathbf{y} = A\mathbf{x}$ mittels Anfragezerlegung in Postgres-XL ausgewertet. Hierbei wurde die Blockzeilen-Partitionierung, analog zur obigen Blockpartitionierung, per Round-Robin und der zugehörigen Anfragezerlegung nach Abschnitt 8.1.3 umgesetzt. Als Datengrundlage dienten dünn besetzte Bandmatrizen $A \in \mathbb{R}^{n \times n}$ mit jeweils 21 Einträgen je Zeile, wobei stets der Diagonaleintrag besetzt war. Die Bandbreite wurde hierbei zufällig auf $3n/40$ festgelegt.
In diesem Experiment konnte durch die Postgres-XL-spezifische Klausel **execute direct on** eine vollständige lokale Berechnung der Teilergebnisse von $\mathbf{y}$ erreicht werden. Die Klausel zwingt

[1] Die Dimensionsachse ist im Ursprungsartikel fehlerhaft und wurde hier korrigiert.

Postgres-XL zugehörige Anfragen komplett lokal zu berechnen, dies impliziert ebenfalls, dass nur die lokal vorhandenen Tupel verteilter Relationen genutzt werden können. Daher wurde der Vektor $\mathbf{x}$ in diesem Experiment auf alle Knoten repliziert. Damit konnte – bis auf die Replikation von $\mathbf{x}$ – die in Abschnitt 8.1.3 entwickelte Strategie für Matrix-Vektor-Multiplikationen direkt umgesetzt werden.
Getestet wurde neben der Blockzeilen-Verteilung mit und ohne Anfragezerlegung zusätzlich eine 1-Anfragen-Implementation mit gewöhnlicher Round-Robin-Verteilung der Matrix A. Dies soll insbesondere einen Eindruck darüber geben, wie sich naive Ansätze, ohne Vorbetrachtungen zur Partitionierung und operationsbedingter Anpassungen von Anfragen, verhalten.
Die Ergebnisse sind in Abbildung 9.4 dargestellt. Wie zu erkennen ist, verläuft der naive Round-Robin-Ansatz deutlich langsamer als die Blockzeilen-Ansätze. Dieser Verlauf bestätigt insbesondere die Vermutung, dass das Datenbanksystem lokal die vollständigen Teile des Ergebnisvektors $\mathbf{y}$ berechnen kann oder zumindest deutlich weniger Kommunikation zwischen den Knoten geschehen muss.
Trotzdem ist ein vergleichsweise deutlicher Speed-Up der Anfragezerlegung bezüglich des einfachen Ansatzes zu beobachten, welches erneut den praktischen Nutzen der Technik verdeutlicht. Es lässt sich in diesem Fall mittels der Anfragepläne mutmaßen, dass durch **execute direct on** die Teilvektoren von $\mathbf{y}$ komplett lokal berechnet werden, wohingegen die 1-Anfragen-Version zumindest Teile der gruppierten Summierung im distributiven Sinne auf andere Knoten (etwa dem zugehörigen lokalen Coordinator) auslagert.
Konstruktionsbedingt kann in Postgres-XL der Ansatz per **execute direct on** auch für die elementweisen Operationen aus Tabelle 5.2 genutzt werden, um parallele lokale Berechnungen zu gewährleisten. Allerdings erlaubt aktuell (Stand 12/2022) dieser Ansatz keine DDL- und DML-statements, welche den Nutzen für weiterführende Berechnungen noch einschränkt.
Zusammenfassend kann die Umsetzung der in Kapitel 8 etablierten Partitionierungsstrategien und der Anfragezerlegung für Lineare-Algebra-Operationen als sehr positiv gewertet werden. Die zugehörigen Anfragen konnten konsistent und deutlich beschleunigt werden. Die Umsetzung und Umsetzbarkeit variiert, aufgrund fehlender Standardisierung, hierbei in den Systemen. Die möglichen Partitionierungsstrategien, die Architektur des parallelen Datenbanksystems, sowie die Möglichkeit einzelne Rechenknoten direkt anzusprechen gehören hierbei zu den wesentlichen Aspekten, die eine effiziente Anfragezerlegung begünstigen.

9.2 Fundamentale Operationen in Big-Data-Umgebungen

In diesem Abschnitt werden die Experimentalergebnisse der Anfragezerlegungstechnik in Postgres-XL aus dem vorherigen Abschnitt mit Implementationen aus dem State-of-the-Art-Big-Data-System Apache Spark (vgl. Abschnitt 4.3.2) verglichen.

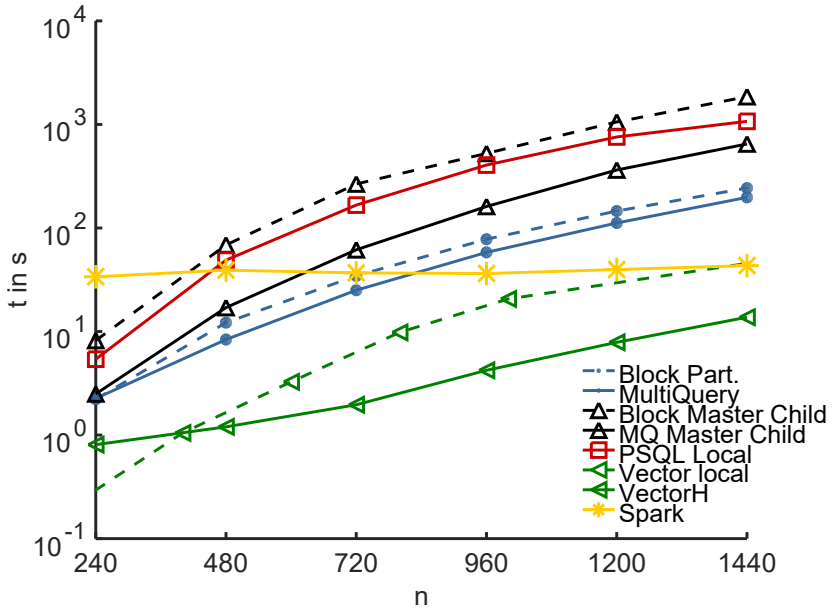


Abbildung 9.5: Ergebnisse von Anfragezerlegungsstrategien bezüglich dicht besetzter Matrizenmultiplikation in Postgres-XL.

Hierfür wurde Apache Spark auf dem gleichem Cluster wie Postgres-XL aufgesetzt. Auf jedem der fünf Berechnungsknoten wurde jeweils ein Spark-Worker genutzt. Der Spark-Master wurde auf dem Gateway-Knoten ausgeführt, auf dem ebenfalls Postgres-XLs GTM verwendet worden ist. Der Master führt insbesondere Sparks Driver-Programm aus, welche unter Anderem die Tasks der Worker organisiert, den auszuführenden Programmcode lokaler Anwendungen zu den einzelnen Worker versendet und mit dem Cluster-Manager kommuniziert. Als Cluster-Manager wurde Sparks interner Standalone-Modus genutzt (vgl. Abschnitt 4.3.2).

Die Implementationen und Testergebnisse sind hierbei Ausarbeitungen von *Lamster* [142] entnommen, welche im Zuge des hier vorgestellten Forschungsprojektes entstanden sind. Alle Implementationen sind im Anhang B.5.3 dokumentiert. Da die Anfragezerlegung in Postgres-XL stellenweise unter expliziten Restriktionen der Datanodes keine Schreiboperationen unterstützt, wurde in Spark ebenfalls auf das explizite Schreiben von Berechnungsergebnissen in das verteilte Dateisystem verzichtet. Stattdessen wurden Berechnungen mit der **count**-Funktion abgeschlossen, um die eigentliche Berechnung aufgrund der lazy evaluation auszulösen. Der Einfluss der Kosten von Count() ist in diesen Fällen, aufgrund der vergleichsweise kleinen Datengröße der Ergebnisse, gering und daher vernachlässigbar.

Dicht besetzte Matrizenmultiplikation

Für die verteilte Berechnung der dicht besetzten Matrizenmultiplikation wurden die native Blockmatrix-Blockmatrix-Multiplikationsimplementation **multiply** von Spark/MLlibs *linalg*-Paket (Apache Spark v. 2.4.5) genutzt. Wie in 4.3.2 beschrieben ist, nutzt *linalg* insbesondere lokal BLAS-Implementaton zur effizienten Berechnung von Basisoperatoren der Linearen Algebra. In Abbildung 9.5 sind die erhaltenen Ergebnisse zu denen des vorigen Abschnitts hinzugefügt worden. Hierbei wurde für jede Dimension der Durchschnittswert aus zehn Läufen abgebildet. Da bereits die Ergebnisse der lokalen Matrizenmultiplikation vergleichsweise schwach in PostgreSQL (vgl. Abbildung 7.3 in Abschnitt 7.2) waren, wurden hier zusätzlich Tests in dem lokalen System Actian Vector und deren parallelen Version Actian Vector in Hadoop („VectorH“) hinzugefügt. Dies soll ein bessere Einordnung des Potenzials geeigneterer paralleler relationaler Systeme ermöglichen. Actian Vector 5.0 wurde hier abermals auf dem in Tabelle 6.1 spezifizierten Notebook genutzt. Actian VectorH 5.0 wurde auf den fünf Berechnungsknoten des gleichen Clusters installiert. Dem Namen entsprechend baut letzteres System auf Hadoop auf. Wie etwa in [132] beschrieben, wird hierbei jedoch nur zur Speicherung von Daten das HDFS und nicht das eigentliche Programmiermodell von Hadoop genutzt. Die Kommunikation der einzelnen Knoten geschieht über MPI. Da in VectorH aktuell nur Hash-Verteilungen von Relationen [143] möglich sind (Stand 12/2022), wurde die Berechnung mittels einer einfachen SQL-Anfrage auf 10 Partitionen (entsprechend der Anzahl an Rechenkernen) umgesetzt.

Wie in Abbildung 9.5 dargestellt, ist Postgres-XL trotz Beschleunigung mittels Anfragezerlegung deutlich langsamer als die anderen Systeme. Ferner ist im betrachteten Dimensionsbereich Vector und VectorH schneller als Spark. Es ist jedoch zu erkennen, dass Spark deutlich besser skaliert und alleine der Overhead des Einlesens und Verteilens der Daten die Laufzeiten in niedrigen Dimensionen dominiert. Es ist demnach davon auszugehen, dass Spark für deutlich größere Dimensionen, durch seine interne Ausnutzung von Lineare-Algebra-Bibliotheken, auch die SQL-Implementationen in Vector und VectorH deutlich unterbieten wird.

Hierbei ist insbesondere zu beachten, dass der abgebildete Dimensionsbereich durch die Laufzeiten von Postgres-XL vergleichsweise niedrig gewählt ist und prinzipiell eine Parallelisierung für klassische Lineare-Algebra-Bibliotheken nicht zwingend nötig ist. Dies ist etwa einer der Gründe, warum der initiale Overhead deutlich größer ist, als die eigentlichen Laufzeitkosten der Berechnung, welches zu einer quasi konstanten Laufzeitkurve führt. Zudem wird der Einfluss des Dateneinlesens und -partitionierens in komplexeren Methoden, die verschiedene Operationen mehrfach und evtl. iterativ kombinieren, voraussichtlich deutlich niedriger sein. Damit ist die Verarbeitungsgeschwindigkeit dicht besetzter Matrizenmultiplikationen in Apache Spark voraussichtlich deutlich niedriger als SQL-basierte Ansätze in parallelen relationalen Systemen.

Gründe für die vergleichsweise langsamen SQL-Laufzeiten und deren Skalierung sind unter an-

derem die bereits in Abschnitt 7.2 diskutierte ineffiziente Cache-Verarbeitung der jeweiligen Anfragen, sowie die nur bedingt erzwingbare effiziente Umsetzung von Blockmatrix-Ansätzen.
Es lässt sich zusammenfassen, dass eine SQL-basierte-Verarbeitung hochdimensionaler Algorithmen, die auf dicht besetzten Matrizenmultiplikaionen aufbauen, im Sinne der Laufzeiten problematisch im Vergleich zu etablierten Big-Data-Systemen zu betrachten sind. Insofern keine reine SQL-Implementation zwingend erforderlich ist, ist eine datenbankinterne Umsetzung durch spezielle Push-Down-Ansätze gegebenfalls sinnvoll. Beispiele hierfür wurden in Abschnitt 4.4 beschrieben und mögliche zugehörige Architekturen in Abschnitt 6.1 diskutiert.
Neben der Laufzeit einfacher Multiplikationen bleibt in dicht besetzten Anwendungsfällen zu beachten, dass es aufgrund der hohen Anzahl an Summen und Multiplikationen numerische Probleme geben kann. Weiterführende Untersuchungen, etwa unter Berücksichtigung der Kahan-Summation (dessen SQL-Implementation mittels rekursiver Anfragen in Anhang B.1.1 zu finden ist), sind in diesem Kontext sinnvoll.

Dünn besetzte Matrix-Vektor-Multiplikation

Als zweites Szenario wurde die dünn besetzte Matrix-Vektor-Multiplikation in Apache Spark 2.4.0 getestet. In diesem konkreten Fall konnten keine nativen Implementationen des *linalg*-Pakets genutzt werden, da verteilte dünn besetzte Matrixmultiplikationen nach offizieller Dokumentation [144] (Stand 12/2022) für den hier genutzten Datentyp `CoordinateMatrix`, welcher für hochdimensionale verteilte dünn besetzte Matrizen dort empfohlen wird, nicht unterstützt werden. Der `CoordinateMatrix`-Typ nutzt analog zum etablierten Coordinate-Relationenschema aus Abschnitt 6.4 das Schema (`Long, Long, Double`).
Um die Nutzung nativer Implementationen für dicht besetzte Probleme zu meiden, welche aus offensichtlichen Gründen ungeeignet sind, wurden in [142] zwei Ansätze entwickelt und getestet:

- eine MapReduce-ähnliche Implementation, mittels der RDD-Funktionen **map**, **join** und **reduceByKey**
- eine Implementation in Spark SQL, die angelehnt an die hier mehrfach genutzte SQL-Implementation aus Abschnitt 7.1 ist

Die Implementationen sind im Anhang B.5.3 aufgeführt. In diesen Ansätzen ist im Vergleich zur Postgres-XL-Implementation kein direkter Einfluss auf die konkrete Partitionierung möglich.
Die Ergebnisse des Tests sind in Abbildung 9.6 dargestellt. Im Vergleich zur dicht besetzten Multiplikation sind die Laufzeitergebnisse in Spark vergleichsweise langsam. Sparks Lösungen skalieren ähnlich zu Postgres-XL (nach initialer Einschwingung), unterscheiden sich stellenweise jedoch um über eine Ordnung im Vergleich zur reinen Berechnungszeit in Postgres. Gründe hierfür sind offenbar die fehlende Implementation und Koordinierung mittels optimierter Lineare-Algebra-Bibliotheken.

Um den Overhead, den Spark durch das Einlesen und Verteilen der Daten aus dem verteilten Dateisystem benötigt, mit einzubeziehen, wurde in diesem Fall zusätzlich zu den Berechnungslaufzeiten in Postgres-XL die dort benötigte Zeit zum Einlesen der Matrix, zur Erstellung des Vektors und zur Berechnung von Indexstrukturen aufgeführt. Diese Zeiten wurden in einem zusätzlichen Test mit Matrizen gleicher Struktur und Größe durchgeführt und auf die ursprünglichen Ergebnisse der Anfragezerlegung in Postgres-XL aufsummiert. Durch diese Betrachtung lässt sich etwa der Vorteil der vorverarbeiteten Daten im Datenbanksystem, für möglicherweise nur einmalig genutzte Matrizen, besser einschätzen. Für den Fall getrennt mehrfach genutzter Matrizen, wie sie beispielsweise in hochdimensionalen dünn besetzten Hidden-Markov-Modellen auftreten können, ist diese Unterscheidung im Allgemeinen jedoch nicht sinnvoll. Durch die Verarbeitung der Daten im Datenbanksystem wird in etwa eine Ordnung benötigt, welches trotz Reduktion zu einer signifikant besseren Laufzeit des Anfragezerlegungsansatzes im Vergleich zu Spark führt.
Es bleibt hierbei zu betonen, dass beide Ansätze einfaches Optimierungspotenzial besitzen. So wurde etwa im Zuge des Forschungsprojekts in [145] im Kontext von Zeitreihen gezeigt, dass das Einlesen von Parquet-Dateien im Vergleich zu CSV-Dateien sich positiv auf die Laufzeiten in Spark auswirken kann. Ferner ist davon auszugehen, dass die für Big-Data-Szenarien wichtigen verteilten Operationen auf dünn besetzten Problemen künftig in Spark/MLlib optimiert eingepflegt werden, sodass eine Wiederholung der Laufzeituntersuchung in Apache Spark zu gegebener Zeit sinnvoll ist.
Auf der anderen Seite kann, wie im dicht besetzten Multiplikationsfall bereits demonstriert, die Wahl eines geeigneteren parallelen relationalen Datenbanksystems bereits Berechnungen um Ordnungen beschleunigen. Umsetzungen von Anfragezerlegungstechniken auf solchen Systemen vermögen (insofern unterstützt) möglicherweise diese Ergebnisse noch zu verbessern.
Damit lassen sich insgesamt die Ergebnisse zumindest als Indiz dafür werten, dass SQL-basierte datenbankinterne Verarbeitungen dünn besetzter Big-Data-Probleme vergleichsweise effizient in relationalen Systemen umgesetzt werden können. Diese Erkenntnisse verdeutlichen ein Potenzial, welches weiterführende Untersuchungen oder auch Anwendungsumsetzungen motiviert.

9.3 Fourier Transformation

In diesem Abschnitt wird eine SQL-basierte datenbankinterne Berechnung von Fourier-Transformationen präsentiert. Dieser Anwendungsfall ist insbesondere interessant, da er zum einen ebenfalls auf einfache Lineare-Algebra-Operationen zurückführbar ist, jedoch weiterführende Aspekte zur Beschleunigung (etwa rekursive Berechnungsansätze) existieren.
Fourier-Transformationen sind das wohl wesentlichste Mittel zur Analyse des Frequenzspektrums von Signalen (etwa Audio-Dateien). Die Fourier-Transformation teilt zu untersuchende

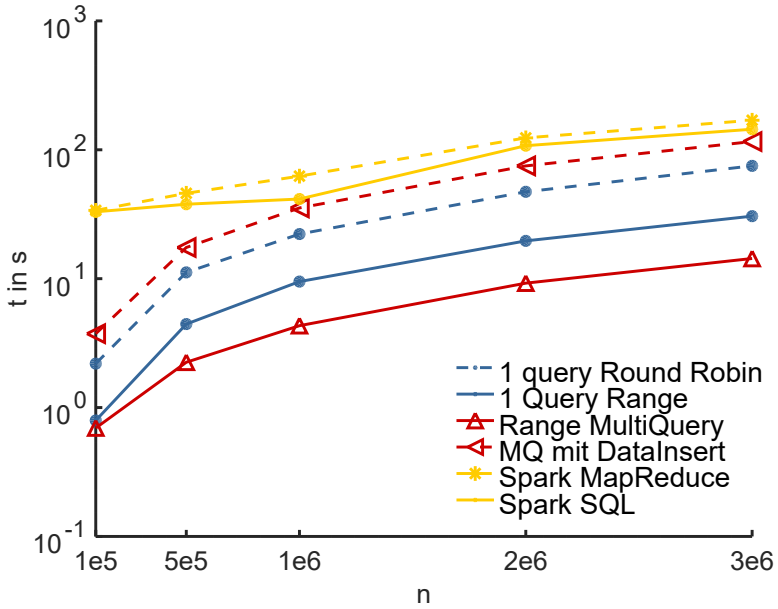


Abbildung 9.6: Ergebnisse der dünn besetzten Matrix-Vektor-Multiplikation in Postgres-XL mit und ohne Anfragezerlegung, mit Anfragezerlegung inklusive Datenimport und Indexerstellung ('MQ mit DataInsert') und Spark-Implementation mittels RDD und Spark SQL.

Signale eindeutig in eine Menge von Wellenformen des zugehörigen Frequenzspektrums so auf, dass deren Superposition dem ursprüngliche Signal entspricht. Diese Dekomposition ermöglicht eine vergleichsweise einfache Filterungen, Interpretationen und Weiterverarbeitungen komplexer Signale. Fourier-Transformationen besitzen eine sehr weitreichendes Spektrum von Anwendungsgebieten. Hierzu zählen beispielsweise:

- die Feature-Extraction im Kontext von Machine-Learning-Anwendungen (etwa im Beispiel der Aktivitätserkennungspipeline in [24])
- die Dimensionsreduktion (zum Beispiel zur Nutzung im PArADISE-Framework aus Abbildung 3.1),
- zur Transformation von Signalen (etwa Kompression durch Einschränkung des Frequenzspektrums),
- Ähnlichkeitsanfragen an Mediendaten (etwa Audiodateien oder Bilder) im Kontext von Multimedia-Datenbanken,
- und weitere.

All diese Punkte haben insbesondere große Relevanz für den hier betrachteten Kontext des PArADISE-Frameworks zur Unterstützung der Entwicklungs- und Nutzungsphase von Assis-

tenzsystemen. Ungeachtet der zahlreichen weiteren Anwendungsgebiete ist demnach eine Untersuchung zur Berechenbarkeit von Fourier-Transformationen wertvoll. Für diese wird im Folgenden eine kurze Erläuterung theoretischer Aspekte und Umsetzungsmöglichkeiten dieser gegeben, woraufhin die Umsetzung in SQL diskutiert wird. Hierbei wird insbesondere die Überführung von *Fast-Fourier-Transformationen* in eine Menge rekursiver Anfragen untersucht. Neben den eingeführten Implementationsarten der Transformation wird zusätzlich ein klassisches Anwendungsbeispiel der Audioanalyse evaluiert. Die Abhandlungen wurden erstmals in [11] veröffentlicht und sind hier zusammenfassend (mit teilweise leichten Erweiterungen) dargelegt.

9.3.1 Theoretische Aspekte

Im Folgenden wird eine kurze Beschreibung wesentlicher Aspekte von Fourier-Transformationen gegeben. Für detailliertere Ausführungen zur Transformierbarkeit, deren Eindeutigkeit und weiteren theoretischen Grundlagen sei etwa auf [146] verwiesen.

Die Fourier-Transformation einer Funktion $f \in L_1(\mathbb{R})$ (etwa kontinuierliche oder stetige Signale) aus dem Raum der integrierbaren Funktionen

$$L_1(\mathbb{R}) = \left\{ f : \mathbb{R} \mapsto \mathbb{R} \mid f \text{ messbar }, \int_{\mathbb{R}} |f|^1 \,\mathrm{d}\, x < \infty \right\},$$

ist eine Funktion $\hat{f} : \mathbb{R} \mapsto \mathbb{C}$ mit

$$\hat{f}(\omega) := \int_{\mathbb{R}} f(t)\, \mathrm{e}^{-2\pi i t \omega} \,\mathrm{d}\, t \tag{9.1}$$

Mehrdimensionale Funktionen $f \in L_1(\mathbb{R}^n)$ über dem Raum $\mathbb{R}^n$ werden hier vernachlässigt. Zusätzlich wurde hier und im späteren diskreten Fall, wie oftmals in der Literatur verbreitet, auf einen Vor- bzw. Normierungsfaktor aus Gründen der Einfachheit verzichtet. Diese können offensichtlich nachträglich eingeführt werden und haben für die folgenden Betrachtungen keinen weiteren Einfluss. Der Wert $|\hat{f}(\omega)|$ kann als Amplitude der zugehörigen sinusodialen Wellenformen zur Frequenz ω interpretiert werden. Der Zusammenhang ist beispielhaft in den Abbildung 9.7 und 9.8 dargestellt. Die Superposition aller Wellenformen (das Integral) entspricht der ursprüngliche Funktion (dem ursprünglichen Signal). Die Definition bezieht sich in diesem Fall auf Funktionen mit stetigem Definitionsbereich (Zeitraum). Im Falle von Sensordaten liegen typischerweise diskrete Werte vor, sodass eine Anpassung der ursprünglichen Formel (9.1) benötigt wird.

Hierfür soll eine diskrete Zeitreihe $\mathbf{x} = \begin{pmatrix} x_0 & \dots & x_{n-1} \end{pmatrix}^T \in \mathbb{R}^n$ betrachtet werden. Das ur-

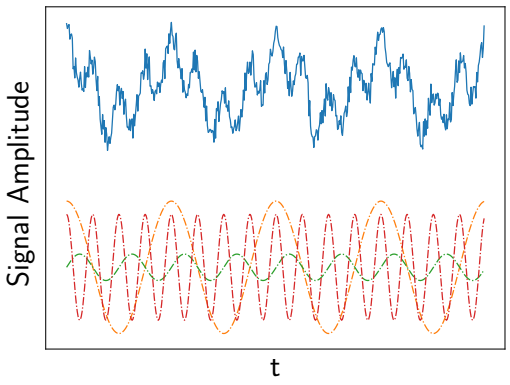


Abbildung 9.7: Beispiel eines einfachen verrauschten Signals (oben) und deren Zerlegung in seine wesentliche sinusodialen Wellenformen aus (9.4) (unten). Grafik erstmalig veröffentlicht in [11].

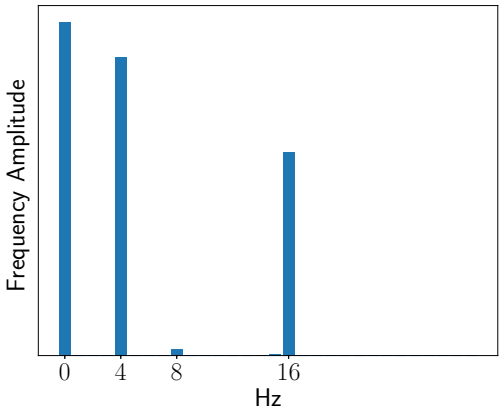


Abbildung 9.8: Ausschnitt des Frequenzspektrums von (9.4) aus Abbildung 9.7, berechnet mittels diskreter Fourier-Transformation, mit Werten bei 0 (Rauschen), 4, 8 and 16 Hz. Grafik erstmalig veröffentlicht in [11].

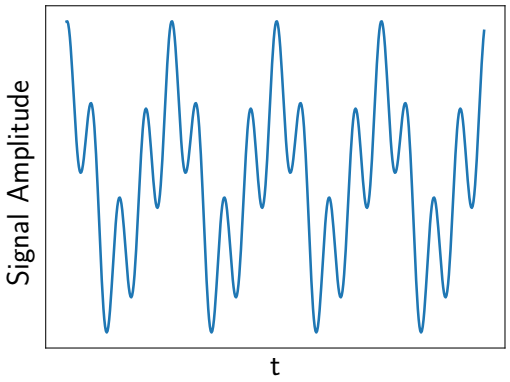


Abbildung 9.9: Beispiel einer Entrauschung beziehungsweise Kompression durch inverse Fourier-Transformation der Wellenformen der drei signifikantesten Frequenzen aus (9.4) und Abbildung 9.7. Grafik erstmalig veröffentlicht in [11].

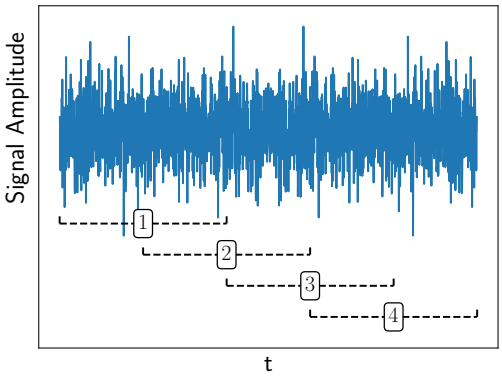


Abbildung 9.10: Beispielhafte Darstellung der short-time Fourier transform (STFT). Grafik erstmalig veröffentlicht in [11].

sprüngliche Integral wird durch die Diskretisierung in eine Summe der Form

$$\hat{\mathbf{x}} = (\hat{x}_j)_{j=0,\dots,n-1} := \left(\sum_{k=0}^{n-1} \mathrm{e}^{-2\pi i j k/n} \, x_k \right)_{j=0,\dots,n-1} \tag{9.2}$$

(unter Vernachlässigung des Vorfaktors) überführt. Hierbei bezeichnet $\hat{\mathbf{x}}$ die *diskrete Fourier-Transformation* (DFT) von $\mathbf{x}$. Analog zu den Methoden der HMM aus Abschnitt 5.2, kann auch (9.2) in ein einfaches Matrix-Vektor-Produkt

$$\hat{\mathbf{x}} = F\mathbf{x} \tag{9.3}$$

überführt werden, wobei

$$F = (f_{ij})_{ij} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w_n & w_n^2 & \dots & w_n^{n-1} \\ 1 & w_n^2 & w_n^4 & & w_n^{n-2} \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & w_n^{n-1} & w_n^{n-2} & \dots & w_n \end{pmatrix}$$

symmetrisch ist und als DFT-Matrix bezeichnet wird. Die Werte

$$w_n^j = \mathrm{e}^{-j2\pi i/n} \qquad j = 0, \dots, n-1$$

werden als Twiddle-Faktoren bezeichnet. Diese (und demnach auch die DFT-Matrix) sind offensichtlich nur abhängig von der Länge der Zeitreihen, sodass diese traditionell in Lookup-Tables gespeichert worden sind, um die Berechnungszeit der DFT zu verkürzen. Mit der hinterlegten DFT-Matrix benötigt die Berechnung der Transformation $n(2n-1)$ Fließkomma-Operationen.

Durch geschickte Ausnutzung der speziellen Struktur der DFT-Matrix beziehungsweise der eigentlichen DFT können die Kosten durch die schnelle Fourier-Transformation (englisch: *Fast-Fourier-Transform*, kurz: FFT) von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n \log n)$ gesenkt werden. Die FFT bezeichnet eine Familie von Algorithmen, die die DFT mittels Teile-und-Herrsche-Strategie berechnen. Im Folgenden wird hier der wohl verbreitetste Vertreter — der Radix-2-DIT-Algorithmus (vgl. etwa [147]) — vorgestellt. Für diesen muss die Länge der zu transformierenden Zeitreihe $\mathbf{x}$ eine Länge der Form $n = 2^p$ ($p \in \mathbb{N}^{\geq 1}$) besitzen oder mindestens durch 2 teilbar sein. Das Verfahren teilt die ursprüngliche Summe aus (9.2) in (dem Verfahrensnamen entsprechend) zwei Teil-DFTs:

$$\hat{x}_k = \underbrace{\sum_{m=0}^{n/2-1} x_{2m} e^{-\frac{2\pi i}{n/2}mk}}_{\text{DFT gerader Indizes}=:E_k} + e^{-\frac{2\pi i}{n}k} \underbrace{\sum_{m=0}^{n/2-1} x_{2m+1} e^{-\frac{2\pi i}{n/2}mk}}_{\text{DFT ungerader Indizes}=:O_k} = E_k + w_n^k O_k$$

Algorithmus 16 Der rekursive Radix-2-DIT-Algorithmus ($s = 1$, $n = 2^p$) zur Berechnung der diskreten Fourier-Transformation einer Zeitreihe **x**.

1: $(\hat{x}_k)_k = \mathrm{FFT}(x, n, s)$
2: **if** $n = 1$ **then**
3: $\quad \hat{x}_0 = x_0$
4: **else**
5: $\quad (\hat{x}_k)_{k=0,\ldots,n/2-1} = \mathrm{FFT}(x, n/2, 2s)$
6: $\quad (\hat{x}_k)_{k=n/2,\ldots,n-1} = \mathrm{FFT}(x_{\cdot+s}, n/2, 2s)$
7: $\quad$ **for** $k = 0, \ldots, n/2-1$ **do**
8: $\quad\quad t = \hat{x}_k$
9: $\quad\quad \hat{x}_k = t + w_n^k \hat{x}_{k+n/2}$
10: $\quad\quad \hat{x}_{k+n/2} = t - w_n^k \hat{x}_{k+n/2}$
11: $\quad$ **end for**
12: **end if**

Algorithmus 17 Der iterative Radix-2-DIT-Algorithmus ($n = 2^p$) zur Berechnung der diskreten Fourier-Transformation einer Zeitreihe **x**.

1: $h = 1$
2: $x = x[\mathrm{reverse\_bit\_order}(1:n)]$
3: **for** $s = 1, 2, \ldots, \log_2(n)$ **do**
4: $\quad$ **for** $j = 0, 2^s, 2 \cdot 2^s, 3 \cdot 2^s, \ldots, n-1$ **do**
5: $\quad\quad$ **for** $t = 0, 1, \ldots, h-1$ **do**
6: $\quad\quad\quad p = t + j + 1$
7: $\quad\quad\quad r = w_n^{t2^{\log_2(n)-s}}$
8: $\quad\quad\quad z = x_{p+h} r$
9: $\quad\quad\quad x_{p+h} = x_p - z$
10: $\quad\quad\quad x_p = x_p + z$
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: $\quad h = 2h$
14: **end for**

Aufgrund der Periodizität von $e^{2\pi i x}$ folgt zusätzlich

$$\hat{x}_{k+\frac{n}{2}} = E_k - w_n^k O_k.$$

Ist $n/2$ ebenfalls durch 2 teilbar, kann die Prozedur wiederholt werden. Im optimalen Fall von $n = 2^p$ kann der Prozess einfach rekursiv fortgeführt werden, bis $E_k, O_k \in \mathbb{C}$ sind. Das zugehörige rekursive Verfahren ist als Radix-2-DIT bekannt und ist in Algorithmus 16 dargestellt. Durch die $\log n$ Teilungsschritte benötigt das Verfahren, im Kontrast zur einfachen diskreten Fourier-Transformation, nur $\mathcal{O}(n \log n)$ Fließkommaoperationen.
Eine rekursive Implementation kann sich für große Probleme (tiefe Rekursionsstufen) aufgrund des hohen Speichermanagementaufwands negativ auf die Performance auswirken. Aus diesem Grund wird das Verfahren oft in eine iterative Version überführt. Der zugehörige Algorithmus ist in 17 dargestellt.

9.3.2 Fourier-Transformation in SQL

Nach der Etablierung der grundlegenden theoretischen Aspekte und Verfahren wird nun deren Überführung in SQL diskutiert. Hierfür werden drei Strategien untersucht, wobei zwei die klassische DFT mittels DFT-Matrix umsetzen und die dritte die iterative FFT in SQL überführt.
Für die Relationen der entsprechenden Zeitreihen/Vektoren und der DFT-Matrix wird das Coordinate-Schema genutzt, wobei zur Darstellung der komplexen Werte die Attribute `re` und

`im` für die Real- und Imaginärteile der Einträge genutzt werden. Zusätzlich wird zur Umsetzung der Berechnung der *bit reverse order* des iterativen Radix-2-Verfahrens aus Algorithmus 17 eine Relation

```
bro ( i int, v int not null, primary key (i))
```

genutzt. Diese speichert die initiale Restrukturierung der Zeitreiheneinträge durch Umkehrung der Bit-Darstellung ihrer Indizes (vgl. Abbildung 9.11).

DFT in SQL

Wie bereits beschrieben, können die Twiddle-Faktoren und die DFT-Matrix vorberechnet und hinterlegt werden, insofern eine feste Zeitreihenlänge genutzt wird. Aus diesem Grund wird die Berechnung dieser hier nicht näher betrachtet und davon ausgegangen, dass diese in den Relationen `w` und `fmat` vorliegen. Die DFT-Matrix kann auch einfach in einer Common-Table-Expression aus den Twiddle-Faktoren mittels kartesischem Produkt berechnet werden (vgl. [11]). Dies ist jedoch aufgrund der zusätzlich nötigen Operationen offensichtlich langsamer und wird hier vernachlässigt.
Für die eigentliche Berechnung der Transformation ist zunächst festzustellen, dass die DFT mittels der etablierten SQL-92-Anfrage der Matrix-Vektor-Multiplikation aus Abschnitt 7.1 umgesetzt werden kann. Hierbei muss jedoch das Produkt der elementweisen Multiplikationen

$$\begin{aligned} v_1 \cdot v_2 &= (\mathfrak{Re}(v_1) + i\mathfrak{Im}(v_1))(\mathfrak{Re}(v_2) + i\mathfrak{Im}(v_2)) \\ &= (\mathfrak{Re}(v_1) \cdot \mathfrak{Re}(v_2) - \mathfrak{Im}(v_1) \cdot \mathfrak{Im}(v_2)) + i(\mathfrak{Re}(v_1) \cdot \mathfrak{Im}(v_2) + \mathfrak{Im}(v_1) \cdot \mathfrak{Re}(v_2)) \end{aligned}$$

zweier komplexer Zahlen $v_1, v_2 \in \mathbb{C}$ erneut in deren Real- und Imaginärteil aufgeteilt werden. Die wohl etablierte Struktur der zugehörigen SQL-Anfrage erlaubt hierbei die Nutzung der Partitionierungs- und Anfragezerlegungsstrategien aus Kapitel 8 zur parallelen Berechnung.
Das Problem dieses Ansatzes ist neben den quadratischen Kosten der Speicherbedarf der DFT-Matrix. Dies gilt sowohl für die vorberechnete Version wie für die Online-Berechnung per CTE. So werden für n Zeitstempel („samples“) bereits

$$\underbrace{4+4+8+8}_{\text{2 integers (32bit) und 2 doubles}} \cdot \underbrace{n^2}_{\text{Elemente von } F} / \underbrace{2^{30}}_{\text{byte to GB}} \text{ GB}$$

an (umkomprimierten) Daten in `fmat` hinterlegt, welches beispielsweise für $32768 = 2^{15}$ in etwa 24 GB und für $65536 = 2^{16}$ bereits 96 GB darstellt. Dies ist trotz der moderaten Größe an samples zu groß, um effizient verarbeitet werden zu können. Es wurde hier kein effizienterer Weg im Coordinate-Relationenschema gefunden, die Twiddle-Faktoren direkt mit der Zeitreihe `x` zu verbinden, ohne ein kartesisches Produkt von `w` zu nutzen.

Aus diesem Grund wurde ein weiterer Ansatz getestet, der aufgrund der Symmetrie von F etwa nur die Hälfte der nötigen Elemente nutzt. Hierfür wurde nur die obere Dreiecksmatrix ($i \leq j$) von F aus (9.3.1) in `fmat` hinterlegt und die ursprüngliche Anfrage in eine Aggregation zweier Teilanfragen umgeformt. Damit werden in etwa die Hälfte der Daten eingespart (genauer: $\sum_{k=1}^{n-1} k = (n^2 - n)/2$ Matrixeinträge).
Durch diese Einsparung wird die ursprüngliche Summe durch zwei Teilsummen

$$(\hat{x}_i)_i = \left(\sum_{k=0}^{i-1} f_{ki} x_k + \sum_{k=i}^{n-1} f_{ik} x_k \right)_i$$

implementiert. Dies kann in SQL durch eine Aggregation über zwei Teilanfragen umgesetzt werden. Hier verarbeitet die erste Teilanfrage alle Tupel der Relation `fmat` ($(n^2 + n)/2$ Tupel) und die zweite die gleiche Relation ohne Diagonalelemente (`i<>j`) ($(n^2 - n)/2$ Tupel) unter geänderter Verbundbedingung `fmat.i=x.i`. Die zugehörige Anfrage lautet:

```
insert into f
select i, sum(re), sum(im)
from (
   select fmat.i as i, x.re*fmat.re-x.im*fmat.im as re,
      x.re*fmat.im+ x.im*fmat.re as im
   from fmat join x on fmat.j=x.i
   union all
   select fmat.j as i, x.re*fmat.re-x.im*sfmat.im as re,
      x.re*fmat.im+x.im*fmat.re as im
   from fmat join x on fmat.i=x.i
   where fmat.j <> fmat.i
) t
group by i
```

Wie aus den Ausführungen aus Abschnitt 7.4 entnommen werden kann, ist die Nutzung von Indexstrukturen aufgrund der nicht-selektiven Natur der Anfragen nicht von Vorteil. Beide Anfrage-Ansätze der DFT werden nach der Diskussion der FFT in PostgreSQL und Actian Vector ausgewertet.

FFT in SQL

Da beide DFT-Ansätze, neben der quadratischen Anzahl an Fließkomma-Operationen, einen großen Speicherbedarf haben, ist eine Umsetzung der FFT in SQL für große Dimensionen wünschenswert. Für die folgenden Betrachtungen seien hierfür die Twiddle-Faktoren in der Relation `w` und die bit reverse order in der Relation `bro` hinterlegt.

Wie in Abschnitt 7.3 beschrieben, ist die Verarbeitung hoch-iterativer Verfahren in SQL-92 schwierig effizient umzusetzen. Aus diesem Grund werden hier rekursive Anfragen aus SQL:1999 genutzt, welche in einigen DBS nicht unterstützt werden. Ironischerweise wird mit diesem im Folgenden die iterative FFT-Version aus Algorithmus 17 in SQL überführt. Rekursive Anfragen können hierbei gewinnbringend genutzt werden, da jeder Eintrag der Zwischenergebnisse von $\hat{\mathbf{x}}$ in jeder „Rekursionsstufe“ (bzw. äußeren Iterationsstufe) nur genau einmal genutzt wird. Dies ist beispielhaft für $n = 2^3$ in Abbildung 9.11 dargestellt. Aufgrund der Struktur wird der dort abgebildetete Graph oftmals als Schmetterlingsgraph bezeichnet. Die initiale Umordnung von $\mathbf{x}$ entspricht hier der bit reverse order für $n = 8$. Durch das einmalige Vorkommen jedes Vektorindexes in jeder Rekursionsstufe, kann in SQL jede dieser durch eine eigene rekursive Anfrage der Form

```
with recursive xrec (
    with z = g1 ( (x ⋈ xrec) ⋈ w ), xjz = g2 ( x ⋈ z )
    select x_p + z
    from xjz
    union all
    select x_{p+h} − z
    from xjz
    where p < h
)
```

umgesetzt werden. Wie hierbei zu erkennen ist, werden die zwei inneren Schleifen ($j = 0, 2^s, 2 \cdot 2^s, 3 \cdot 2^s, \ldots, n-1$ und $t = 0, 1, \ldots, h-1$) aus Algorithmus 17 in der Anfrage modelliert. Die äußere Schleife ($s = 1, 2, \ldots, \log_2(n)$) wird durch sukzessive Wiederholung der Anfrage unter Anpassung von h umgesetzt. Der hierbei entstehende anfrageübergreifende Update-Prozess wurde in 3 Strategien implementiert und getestet:

1. Jedes Zwischenergebnis einer Stufe wurde in die ursprüngliche Signal-Relation `x` eingefügt, durch das Hinzufügen eines zusätzlichen Attributes `level` **int** (mit den Werten $s = 1, 2, \ldots, \log_2(n)$) im zugehörigen Relationenschema.

2. Alle Zwischenergebnisse wurden in sequenziell angeordneten CTEs gespeichert und nur das finale Ergebnis in die Relation `f` eingefügt.

3. Die Tupel aus `x` wurden in jedem äußeren Iterationsschritt per `update`-Klausel angepasst.

Hierbei hat sich in Stichproben ergeben, dass die Berechnung per **update**-Klausel mit Abstand am effizientesten ist, insofern bezüglich `i` eine Indexstruktur (B-Baum) erstellt wurde. Damit

werden insgesamt $\log_2(n)$ rekursive Anfragen benötigt um die FFT in SQL zu berechnen. Da dies im Allgemeinen vergleichsweise wenig Anfragen sind, ist der Overhead durch sequenzielle Anfragepläne (vgl. Abschnitt 7.3) vernachlässigbar. Der Anfrageplan der gesamten Methode ist vergleichsweise groß und unübersichtlich und ist daher im Anhang B.5 exemplarisch für $n = 2^3$ dargestellt.
Im Folgenden werden nun Laufzeitergebnisse der drei vorgestellten Berechnungen präsentiert und ausgewertet.

Experiment: Fourier-Transformation in SQL

Zur Auswertung der Implementation wurden zwei Experimente umgesetzt. Im ersten wurde die Transformation eines Signals der Form

$$S(t) = \cos(2\pi 4t) + 0.2\sin(2\pi 8t) + 0.8\cos(2\pi 16t) + \varepsilon(t) \tag{9.4}$$

berechnet. Das Signal ist in Abbildung 9.7 dargestellt und besteht aus überlappenden Wellenformen bezüglich der Frequenzen 4 Hz, 8 Hz und 16 Hz. Zusätzlich wurde dem Signal aperiodisches gleich-verteiltes Rauschen hinzugefügt, um ein realistischeres Sensordatenszenario nachzuempfinden. Für das Experiment wurde die Schrittweite der äquidistanten Diskretisierung variiert, um das skalierende Verhalten der Implementationen zu testen. Zur Auswertung wurde das Notebook mit den Spezifikationen aus Tabelle 6.1 genutzt. Als Vertreter relationaler Datenbanksysteme wurde der row store PostgreSQL 11 und der column store Actian Vector 5.1. getestet. Da in letzterem keine rekursiven Anfragen unterstützt werden, wurde nur in PostgreSQL die FFT ausgewertet. Zu Vergleichszwecken wurde zusätzlich eine eigene direkte Überführung des Algorithmus 16 in Python3 getestet. Hierbei wurden keine existierenden Bibliotheksfunktionen (etwa aus NumPy) genutzt, da diese im Allgemeinen komplexere Versionen der FFT umsetzen und so ein Vergleich identischer Algorithmen garantiert werden konnte. Wie sich im Folgeabschnitt zeigen wird, ist in praktischen Anwendungen oftmals das Datenmanagement ein entscheidenderer Faktor für die Performance, sodass in diesen Fällen schnellere Transformationen nur geringen Einfluss auf die Gesamtzeit haben würden. Der zugehörige Quellcode ist im Anhang B.5 hinterlegt. Zusätzlich wurde hierbei zwischen der reinen Berechnungszeit in Python („Py Pure Calc“; Daten bereits im Speicher) und der Verarbeitung inklusive des Datenimports und dem Export von Ergebnissen aus beziehungsweise in PostgreSQL („Py IO PSQL“) unterschieden. Letzterer Ansatz wurde betrachtet, um ein realitätsnäheres Szenario zu simulieren und durch das Schreiben der Ergebnisse einen faireren Vergleich zur reinen Datenbanklösung zu ermöglichen. Die Performance von PostgreSQL und Actian Vector haben sich für die reinen IO-Operationen der Zeitreihen nicht maßgeblich unterschieden, so dass hier nur eines der Systeme für diesen Zweck getestet wurde.

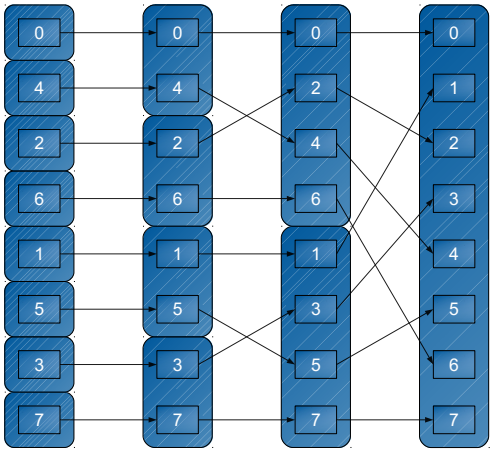


Abbildung 9.11: Abhängigkeitsgraph („Schmetterlingsgraph“) der Zwischenergebnisindizes von $\hat{\mathbf{x}}$ der einzelnen Rekursionsstufen aus Algorithmus 16.

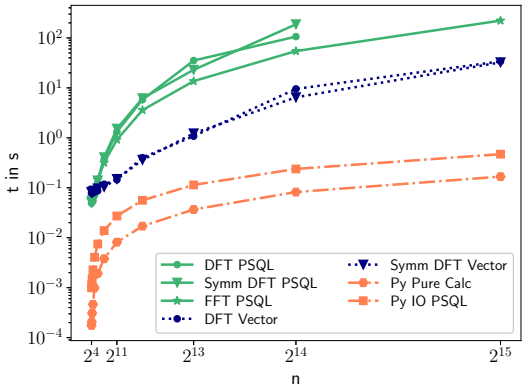


Abbildung 9.12: Laufzeitvergleich zur Berechnung der Fourier-Transformation in PostgreSQL 11, Actian Vector 5.1 und Python3.

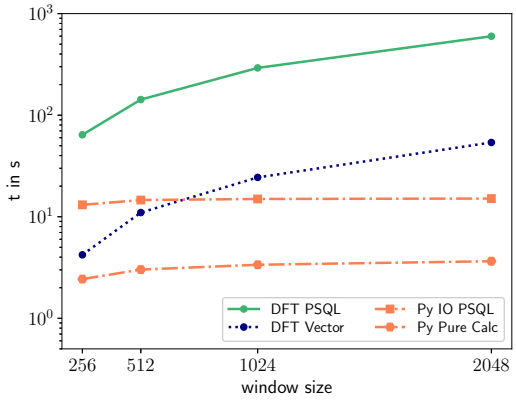


Abbildung 9.13: Laufzeitergebnisse der STFT mit variabler Fenstergröße auf einem 10 Sekunden langen Signal in PostgreSQL, Actian Vector und Python3.

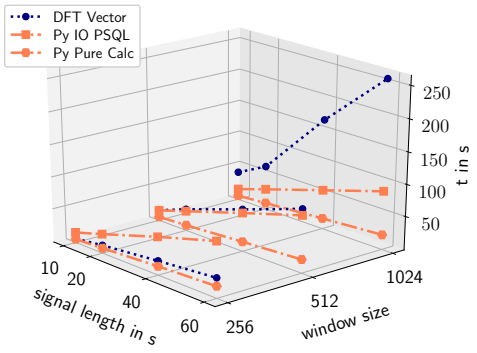


Abbildung 9.14: Laufzeitergebnisse der STFT mit variabler Fenstergröße und Signallänge in Actian Vector 5.1 und Python3.

Die Laufzeitergebnisse der Berechnung der Fourier-Transformationen sind in Abbildung 9.12 dargestellt. Hierbei lässt sich zunächst erkennen, dass alle drei verwendeten Systeme voneinander getrennt im Rahmen von 1-2 Ordnungen agieren. Die Python-Implementationen (FFT) sind erwartungsgemäß am schnellsten und skalieren für hohe Dimensionen am besten. Actian Vector ist bis zu 8192 samples weniger als eine Ordnung langsamer als Python, wobei sich der Abstand aufgrund der quadratischen Komplexität der klassischen DFT vergrößert. Trotzdem kann der vergleichsweise geringe Abstand für die hier betrachtete Auswahl klassischer Sample-Größen als Motivation für die datenbankinterne DFT gesehen werden. Für PostgreSQL zeigt sich abermals eine vergleichsweise langsame Laufzeit, mutmaßlich aufgrund der internen tupelweisen Speicherung (vgl. Abschnitt 6.2). Es zeigt sich jedoch, dass die entwickelte FFT-Implementation mittels rekursiver Anfragen konsistent schneller um einen Faktor von etwa 2-3 ist als die Anfragen zur klassischen DFT. Dies lässt insbesondere erhoffen, dass modernere Systeme, wie Actian Vector, bei der Einführung rekursiver Anfragen den Abstand zu Software des wissenschaftlichen Rechnens weiter verringern können. In den Ergebnissen ist nur bedingt erkennbar inwiefern die SQL-FFT-Implementation in $\mathcal{O}(n \log n)$ liegt. Hierfür sind weitere Tests in PostgreSQL und anderen Datenbanksystemen nötig. Zusätzlich ist zu beobachten, dass sich die Umsetzung der einfachen Matrix-Vektor-Multiplikation und dem darauf aufbauenden symmetrischen Ansatz in ihrer Reihenfolge systemabhängig abwechseln. Im Allgemeinen ist der Unterschied vergleichsweise klein, sodass prinzipiell der symmetrische Ansatz im Sinne des Speicherbedarfs bevorzugt werden kann.

Experiment: Kurzzeit-Fourier-Transformation in SQL

Im zweiten Experiment wurde die Laufzeit eines klassischen Anwendungsszenarios der Audioanalyse untersucht: die Kurzzeit-Fourier-Transformation (in Englisch: short-time Fourier transform; kurz: STFT). In dieser werden auf kleinen überlappenden Fenstern eines Signals (vergleiche Abbildung 9.10) Fourier-Transformationen berechnet. Hierdurch kann beispielsweise der Wechsel dominanter Frequenzen im Laufe der Zeit untersucht werden. Für nähere Erläuterung zur STFT sei auf [148] verwiesen. Klassisch für solche realitätsnahen Szenarien ist hierbei, dass ein deutlich größeres Maß an Datenmanagement (Selektion und Gruppierungen) nötig ist. Dies war initialer Grund für die Untersuchung der STFT und lässt einen besseren Aufschluss zu, inwiefern Frequenzanalysen und darauf aufbauende Methoden in Datenbanksystemen per SQL berechnet werden können.
Für die Tests wurde ein Signal ähnlich dem von Audiodateien erstellt. Hierfür wurde bezüglich einer Abtastrate von 44.1 kHz, der meist genutzten Rate in Audioanwendungen, eine Folge von zufälligen Integer-Werten erstellt. Benachbarte Fenster überlappen sich hier gleichmäßig über 50% der samples (vgl. Abbildung 9.10) und beinhalten, angelehnt an klassische reale Anwendungen, bis zu 2048 Zeitstempel [148]. Die Signallänge variiert hier zwischen 10 und 60 Sekunden.

Für dieses Setup wurden abermals PostgreSQL, Actian Vector und Python getestet. Die fensterweise FFT wurde hierbei für PostgreSQL ausgespart, da diese bei den genutzten jeweiligen Sample-Größen nur unwesentlich schneller war und die klassische DFT vergleichsweise einfach in den Fensteransatz überführt werden kann. Aus gleichem Grund wurde eine zusätzliche Auswertung des symmetrischen-DFT-Ansatzes vernachlässigt.
Die ursprüngliche Matrix-Vektor-Multiplikation-Anfrage kann mittels zweier Teilanfragen (für Fenster mit gerader und ungerader Nummer) und geeigneter Modulo-Verbundbedingungen umgesetzt werden. Dabei wird in beiden Anfragen die DFT-Matrix mit allen nicht überlappenden aufeinanderfolgenden Fenstern verbunden. Der Startpunkt der Fenster wird hierbei konstruktionsbedingt für die zweite Anfrage um 50 % der Fenstergröße `$ws` verschoben. Es ergibt sich damit die Anfrage:

```
insert into stft
select i, window_even, sum(re), sum(im)
from (
   select fmat.i as i, 2*( x.i / $ws ) as window_even,
      x.re*fmat.re as Re, x.re * fmat.Im as im
   from x join fmat on mod(x.i,$ws)=fmat.j
   where x.i < $n-mod($n,$ws)
) ttt
group by i, window_even
union all
select i, window_odd, sum(re), sum(im)
from (
select fmat.i as i, 1+2*( (x.i-$shift) / $ws ) as window_odd,
   x.Re*fmat.re as re, x.re * fmat.im as im
from x join fmat on mod(x.i-$shift,$ws)=fmat.j
where x.i>=$shift and x.i < ($n - mod($n-$shift,$ws) )
) ttt
group by i, window_odd
```

Hierbei bezeichnet zudem `$n` die Länge des ganzen Signals und `$shift`(=0.5`$ws`) den Offset der zweiten Fensterfolge. Neben der erstaunlich kompakten Darstellung des Verfahrens lässt sich der Ansatz einfach per Anfragezerlegung (vgl. Kapitel 8) parallelisieren. Dies kann speziell für lange oder große Mengen an Signalen von Sensoren im Big-Data-Kontext von Bedeutung sein.
Die STFT wurde in zwei Tests ausgewertet. Die Ergebnisse des ersten Teilexperiments sind in Abbildung 9.13 dargestellt. Hier wurden verschiedene Fenstergrößen auf einem 10-Sekunden-

Signal getestet. Es ist hierbei zu erkennen, dass die Laufzeiten dieser Lösungen deutlich näher an der Python-Lösung sind. Insbesondere ist für niedrigere Fenstergrößen Actian Vector performanter als Python mit In- und Export aus beziehungsweise in PostgreSQL. Zusätzlich ist zu erkennen, dass die schlechtere Skalierung der DFT sich auf die STFT mit steigender Fenstergröße negativ auswirkt.

Dieses Verhalten ist noch deutlicher im zweiten Test zu erkennen. In diesem wurden neben der Fenstergröße ebenfalls die Signallängen zwischen 10 und 60 Sekunden variiert. Die Ergebnisse in Abbildung 9.14 zeigen, dass alle Lösungen linear mit der Signallänge skalieren. Im Falle der Fenstrgröße 1024 ist zu erkennen, dass jedoch mutmaßlich Auslagerungseffekte ab 20-Sekunden-Signalen auftreten, sodass die Skalierung von Actian Vector deutlich langsamer (wenn auch linear) ist. Dieser Effekt kann wahrscheinlich durch Aufteilen der Anfragen in Teilsignale umgangen werden und ist demnach nicht kritisch. Es zeigt sich hier abermals, dass die STFT in SQL sehr effizient für kleine Fenster genutzt werden kann. Für größere Fensterlängen ist jedoch mit Performance-Einschränkungen zu rechnen. Dieser ansteigende Abstand entspringt dem unterschiedlichen Skalierungsverhalten der fensterweisen DFT in SQL und der fensterweisen FFT in Python.

Es lässt sich zusammenfassen, dass die Verarbeitung diskreter Fourier-Transformationen in Standard-SQL möglich ist und unter Einbeziehung von SQL:1999 ebenfalls die Berechnung der FFT in SQL umgesetzt werden kann. Die Ergebnisse haben gezeigt, dass niedrig-dimensionale Transformationen effizient in SQL umgesetzt werden können. Dies bezieht insbesondere Anwendungen wie die STFT zur Frequenzanalyse ein.

9.4 Automotive Analysis

Im Folgenden wird ein Beispiel diskutiert, welches wissenschaftliche Methoden nutzt, die nicht effizient oder direkt durch die etablierten Lineare-Algebra-Operatoren umgesetzt werden können. Hierbei handelt es sich um ein Szenario der Zeitreihenverwaltung und -analyse von Messungen aus dem Automotive-Bereich. Wie sich zeigen wird, kann diese datenintensive Verwaltung und Nutzung von Sensordaten — mit ähnlichem Anforderungsprofil zur Berechnung von Features oder dem Lernen von Machine-Learning-Modellen — von der datenbankinternen Verarbeitung profitieren. Dies soll insbesondere die Erweiterbarkeit der Anwendungsbereiche und die Vorteile des hier diskutierten Frameworks aus Kapitel 3 bzw. von SQL-Implementationen verdeutlichen. Die folgende Abhandlung ist eine Zusammenfassung des Konferenzbeitrags [12] und deren Langfassung [13] (Technischer Bericht), in denen das zugehörige Industrieprojekt, deren Ziele und Ergebnisse beschrieben worden sind. Für detailliertere Ausführungen sei auf diese verwiesen.

Szenario

In der Entwicklung und Evaluation von Automobilen spielt die Analyse von Messreihen eine zentrale Rolle. Solche Messreihen werden in Testfahrten mit Messgeräten aufgenommen und im späteren Verlauf — mitunter mehrfach durch verschiedene Analysten/Ingenieure — ausgewertet. Grundlage für die Untersuchungen in dem hier beschriebenen Projekt ist hierbei das MDF-Dateiformat (Measurement Data Format). Das Binärdateiformat ist de-facto Industriestandard und hat seine Ursprünge in den 1990er Jahren. Das Dateiformat wurde von der *Association for Standardization of Automation and Measuring Systems* [149] im Jahre 2009 mit Version 4.0 offiziell standardisiert. Die genutzten Messgeräte schreiben hierbei direkt in die MDF-Datei die jeweiligen Zeitreihen und Zeitachsen, sowie andere Metainformationen, wie beispielsweise Kanal-/Sensornamen, Messgeräteinformationen (etwa Abtastrate und Hardwarespezifikationen), Konvertierungsregeln für Rohdaten und weitere. Für diese Anforderungen ist das Dateiformat (vgl. beispielsweise [150]) als komplexe Struktur von durch Zeiger verbundener Datenblöcke konstruiert. Ein Beispielausschnitt ist in Abbildung 9.15 dargestellt. Die Zeitreihen selbst sind, bedingt durch hohe Abtastraten, oftmals stark komprimierbar, welches beispielsweise durch das MDF4-Format transparent unterstützt wird.
Da ein deutlicher Anstieg der Anzahl der MDF-Dateien und deren Größen in naher Zukunft erwartet wird, ist — auch im Kontext künftiger messreihenübergreifender Analysen — eine Suche nach Alternativen zur Arbeit mit Dateien nötig. Im Projekt wurde daher untersucht, inwiefern MDF-Dateien effizient in (parallelen) Datenbanksystemen verwaltet und genutzt werden können. Dafür wurden klassische Methoden von Analysen evaluiert und in SQL überführt. Es wurden hierfür drei Grundanwendungen untersucht:

1. Selektion von n Kanälen aus m Dateien; Interpolation der Zeitreihen auf eine Zeitachse pro Messung („Time-Merge“) und anschließender Visualisierung der Werte.

2. Anwendung 1 mit zusätzlicher Suche von Zeitreihen-Intervallen bezüglich einfacher logischer Bedingungen.

3. Metadatenanfrage: Suche nach Messungen, die zu spezifizierende Bedingungen erfüllen (Datum, Messlänge, enthaltene Kanäle).

Anwendung 2 wurde aufgrund seiner Ähnlichkeit zu Anwendung 1 in [12, 13] nicht explizit ausgewertet. Die Interpolation ausgewählter Zeitreihen einer Messdatei bezüglich einer gemeinsamen Zeitachse ist hierbei essenzieller Bestandteil der ersten beiden Aufgaben, um die Zeitreihen punktuell vergleichbar zu machen. Hierfür wurde der *zero-order hold* als ein klassischer Vertreter von Interpolationsmethoden genutzt, welcher den Zeitstempel einer Zeitreihe auf den nächstfolgenden Zeitstempel der Basis-Zeitreihe verschiebt, ohne den zugehörigen Zeitreihenwert

zu verändern. Das Verfahren ist konzeptuell in Grafik 9.16 dargestellt. Die Auswahl der Methoden ist hierbei an den aktuellen Stand der Technik angepasst und bezieht sich, gegensätzlich zu Machine-Learning-Anwendungen, nur auf einen kleinen Teil der Sensordaten. Dies ist prinzipiell durch die Verwaltung mittels einzelner Dateien pro Messung vergleichsweise effizient handhabbar. Da jedoch in naher Zukunft die Suche von Mustern und ähnliches über alle oder große Mengen an Messungen und Zeitreihen geschehen soll, ist eine einfache Selektion und Gruppierung dieser nötig. Der Nutzen von Datenbanksystemen solcher messungsübergreifenden Methoden wird etwa in der Auswertung der dritten Anwendung deutlich.
Für die Untersuchungen wurde unterschieden zwischen lokaler (lokale Datenbanksysteme) und Cluster-basierter Verarbeitung (parallele Datenbanksysteme). Zusätzlich wurde eine Verarbeitung in Apache Spark mit Datenspeicherung in HDFS (CSV und Parquet-Dateien) getestet. Da der Testdatenbestand (40 GB MDF3-Dateien) nicht groß genug für eine gewinnbringende parallele Verarbeitung in Spark war, wurde die weitere Einbeziehung des Systems jedoch vernachlässigt.

Umsetzung

Für die Überführung der MDF-basierten Zeitreihenanwendung in relationale Datenbanksysteme wurden verschiedene klassisch (objekt-)relationale und parallele (objekt-)relationale SQL-Systeme untersucht. Die genutzten Architekturen entsprechen maßgeblich den hier vorgestellten Architekturversionen aus Abbildung 6.1 (volle SQL-Umsetzung) und 6.2 (Auslagerung einzelner Methoden in externe Scientific-Computing-Software) aus Abschnitt 6.1. Hierbei wurde jedoch im parallelen Fall auf die Anfragezerlegung verzichtet und sich auf adäquate Partitionierungsstrategien beschränkt. Die Visualisierung der interpolierten Zeitreihen, sowie mögliche zusätzliche Auslagerung von wissenschaftlichen Methoden, wurden hierbei in Python3 lokal auf den Rechenknoten des Anwenders umgesetzt.
Für den Pushdown in das Datenbanksystem wurden mehrere Teiloperationen ausgewertet und getestet. Hierzu zählen neben der Selektion von Daten (etwa Zeitreihen) und spezifizierten Bedingungen auch der Time-Merge (Interpolation), sowie arithmetische Operationen zur Konvertierung der rohen Sensordaten in interpretierbare Werte (beispielsweise SI-Einheiten). Letztere konnten vergleichsweise einfach in SQL-92 umgesetzt werden, welches etwa die Online-Konvertierung von Rohdaten ermöglicht hat. Konträr hierzu konnte die Interpolation nur mit vergleichsweise langsamen Laufzeiten und der Zunahme von SQL-Window-Funktion umgesetzt werden. Zusätzlich wurden diese nicht von allen Systemen unterstützt. Aus diesem Grund wurde in PostgreSQL/-XL die Nutzung von (imperativ formulierbaren) User-Defined-Functions (UDFs) in Postgres' eigener Sprache PL/pgsql, welche syntaktisch dem Quasi-Standard PL/SQL ähnelt, umgesetzt.
Die Aufweichung des strikt relationalen Ansatzes hat sich insbesondere im Fall der eigentlichen

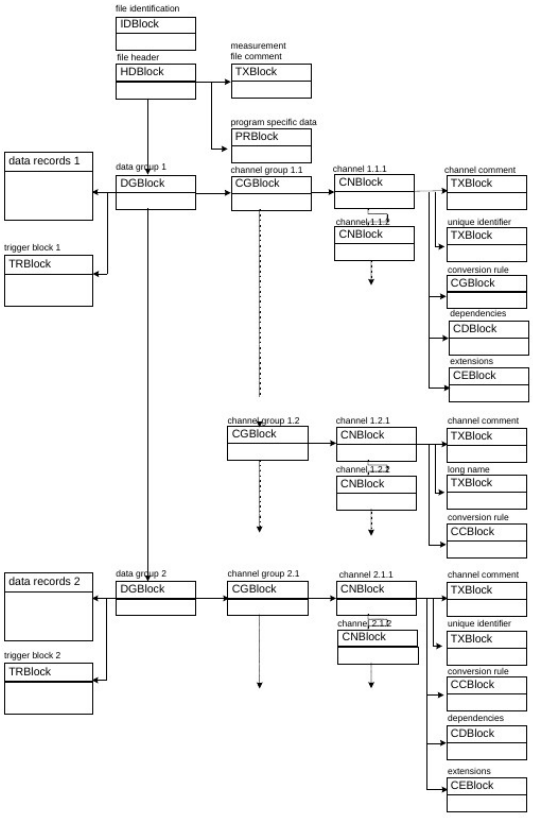


Abbildung 9.15: Ausschnitt einer Beispiel-MDF3-Datei. Grafik entnommen aus [150].

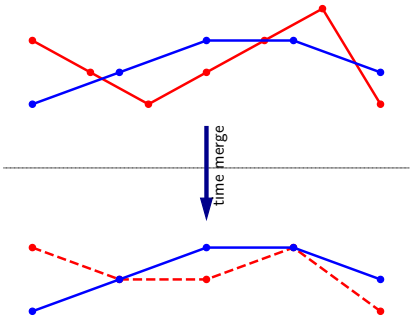


Abbildung 9.16: Beispiel eines Time-Merges mittels zero-order hold. Die in rot dargestellte Zeitreihe wird bezüglich der Zeitachse des blauen Kanals interpoliert.

Speicherung und Verwaltung von Zeitreihen positiv dargestellt. So wurde zunächst für die Speicherung der MDF-Schemata ein Datenbankschema entwickelt, welches die Zeitreihen in einem leicht erweiterten Coordinate-Relationenschema (vgl. Abbildung 9.17 und Abschnitt 6.4) mit zusätzlichem Messungsidentifizierungsattribut hinterlegen.

Hierbei hat sich gezeigt, dass in row stores (etwa PostgreSQL/-XL) dieses Schema nicht nutzbar ist, da aufgrund fehlender transparenter Lauflängenkompression (vgl. Ausführungen in Abschnitt 6.4), die resultierende Datenbankgröße nach dem Import um ein Vielfaches größer ist, als die der eigentlichen MDF-Dateien. So wurde in PostgreSQL 11.3 beobachtet, dass eine 800 MB Datei im erweiterten Coordinate-Schema etwa 25 GB groß ist. In column stores wurden deutlich bessere Kompressionsraten, aufgrund der sequenziellen Speicherung der Zeitreihenelemente, erreicht. Trotzdem waren die Datenbanken größer als die eigentlichen MDF-Dateien. Dies ist vermutlich durch nicht komprimierbare zusätzliche Identifizierungsattribute für die je-

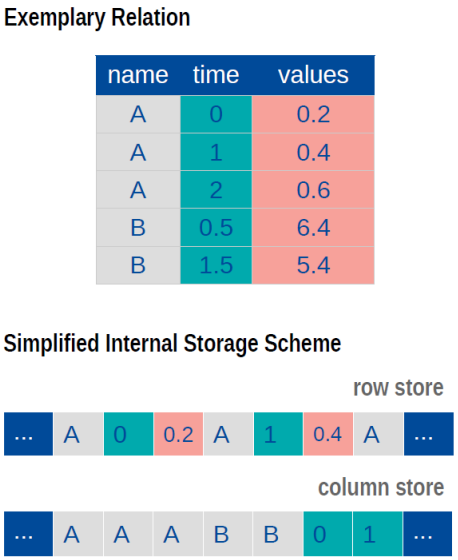


Abbildung 9.17: Exemplarische Darstellung zur Speicherung von Zeitreihen in einem erweiterten Coordinate-Schema. Die Grafik ist aus [13] entnommen.

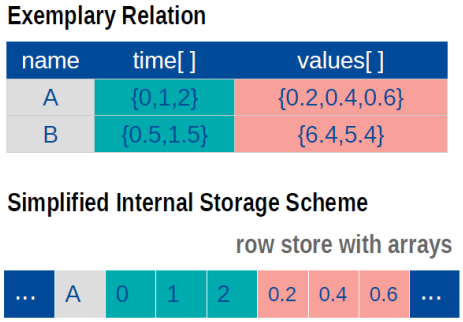


Abbildung 9.18: Exemplarische Darstellung zur Speicherung von Zeitreihen im entwickelten Array-Schema. Die Grafik ist aus [13] entnommen.

weiligen Zeitreihenindices zu erklären. Um eine Verwaltung und Speicherung in row stores zu ermöglichen, wurden in PostgreSQL und -XL Schemata getestet, in denen Zeitreihen und Zeitstempel in Array-Datentypen hinterlegt worden sind (vgl. Abbildung 9.18) . Eines der Array-Datenbankschemata, für die konvertierte Sensordaten gespeichert worden sind, ist hierbei in Abbildung 9.19 dargestellt. Die Zeitreihen sind hier in der `timeseries`-Relation hinterlegt, dessen Schema dem Array-Relationenschema aus Abschnitt 6.4.1 ähnelt. Da durch die Nutzung von Arrays keine Identifizierungsattribute einzelner Zeitreihenelemente benötigt werden, konnte in Postgres für die obige 800 MB MDF-Datei eine Reduktion auf eine 140 MB Datenbank verzeichnet werden. Zusätzlich können durch die deutliche einhergehende Tupelreduktion eine Selektions- und Verbundbeschleunigung für Anfragen auf der `timeseries`-Relation erreicht werden. Wie im folgenden Abschnitt beschrieben, ist dieser Ansatz in Postgres der performanteste für die im Kontext der drei definierten Grundanwendungen. Abseits von diesen birgt die Nutzung von Arrays jedoch auch Nachteile. So müssen die Zeitreihen, analog zu den Betrachtungen über Array-Schemata in Abschnitt 6.4, zur Weiterverarbeitung entschachtelt oder per UDF verarbeitet werden. Dies führt insbesondere dazu, dass Kurvenverläufe oder Selektionen mit Bedingungen (etwa Intervalle über Schwellenwerte o.ä.), die zeitreihen- und messungsübergreifend sind, nicht effizient auf großen Datenbeständen umgesetzt werden können. Insbesondere entfällt hierbei die Möglichkeit der Nutzung von Indexstrukturen zur Erkennung von Mustern (etwa Extrema oder rapide Anstiege durch zusätzliche Speicherung numerischer Ableitungen). Speziell

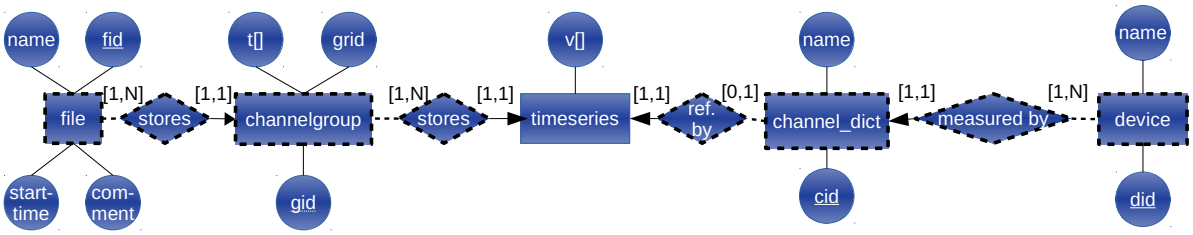


Abbildung 9.19: Entity-Relationship-Modell für die Speicherung von Fließkomma-Zeitreihen in Array-Datentypen. Gestrichelte Kanten und Pfeile repräsentieren schwache Entitäten. Grafik erstmalig veröffentlicht in [12, 13].

bei großen Datenbeständen über tausende Messungen und Millionen von Zeitreihen können solche Ansätze jedoch essenziell für die Umsetzbarkeit von Analysen sein. Neben der Handhabung in SQL-Systemen ist die Unterstützung von Array-Datentypen selbst zusätzlich nur bedingt in Systemen unterstützt. Beispielsweise ist die Nutzung von Arrays in keinem der beiden untersuchten column stores möglich. Dies beeinträchtigt die Entwicklung eines universellen Frameworks bezüglich eines festen Datenbankschemas, welches die Austauschbarkeit bis hin zur Nutzbarkeit einzelner SQL-Systemen begrenzt. So wurden Untersuchungen bezüglich eines row stores ohne Array-Funktionalität vorzeitig eingestellt, da die Performance und der Speicherbedarf nicht effizient genug waren. Im Kontext der Funktionalität musste zusätzlich die Betrachtung eines parallelen column stores vernachlässigt werden, da das System keine B-Baum-Indexstrukturen unterstützte. Dies führt insbesondere in Kombination mit dem vergleichsweise tupelintensiven erweiterten Coordinate-Schema zu schwachen Laufzeiten für Selektionen und Verbunden.
Mit dieser Kurzdiskussion der Schemata und erweiterter SQL-Funktionalitäten werden im folgenden Abschnitt Experimentalergebnisse des Array-Ansatzes in PostgreSQL und -XL mit den Stand-der-Technik Python-MDF-Lösungen verglichen. Postgres-Systeme wurden hierbei aufgrund der beschriebenen guten Zeitreihenkompression und effizienten Selektion ganzer Reihen genutzt.

9.4.1 Auswertung

Zur Auswertung der Array-Ansätze im Kontext der Grundanwendungen 1 und 3 wurden zwei wesentliche Szenarien getestet. Im ersten wurde die Verarbeitung fünf lokal vorhandener MDF3-Dateien mit einer Gesamtgröße von 3.7 GB in Python gegen den etablierten Array-Datenbankansatz in PostgreSQL 11.3 verglichen. Hierfür wurde das Notebook mit den Hardware-Spezifikationen aus Tabelle 6.1 genutzt. PostgreSQL wurde hierbei nach Tabelle 6.2 rekonfiguriert. Im zweiten Setup wurde eine Cluster-Berechnung mit insgesamt 50 GB Daten aus 43 MDF-Dateien

evaluiert. Hierfür wurden die MDF-Dateien auf einem Cluster gespeichert und per Netzlaufwerk angesprochen. Auf dem gleichen Cluster wurde Postgres-XL genutzt. Hierbei wurde die gleiche Hardware und Aufteilung von Postgres-XL-Komponenten wie in Abschnitt 9.1 verwendet. Zusätzlich wurde für die Anwendungen unterschieden, inwiefern bereits in Fließkomma-Darstellungen konvertierte Zeitreihen gespeichert werden („`conv`“) oder Rohdaten in deren Festkomma-Darstellung mit zugehörigen Konvertierungsinformationen („`raw`“) hinterlegt sind. Im letzteren Fall werden Zeitreihen online in der Datenbank konvertiert, welches in den meisten Fällen durch arithmetische Operationen einfach in SQL umgesetzt werden kann.
Die Ergebnisse der ersten Grundanwendung (Selektion von Kanälen aus mehreren Messungen; Time-Merge; Visualisierung) sind in Abbildung 9.20 dargestellt.
Hierbei wurden die Kanäle aus 5 Messungen selektiert und in den Grafiken jeweils der Durchschnitt aus 10 Läufen dargestellt. Zusätzlich ist hierbei unterschieden worden, inwiefern der Time-Merge mittels UDF in Postgres oder nach der Selektion und der Ergebniskommunikation in Python umgesetzt wurde. Wie zu erkennen ist, skalieren lokal alle Ansätze linear bezüglich der Anzahl der Kanäle, wohingegen die UDF-Ansätze in Postgres-XL mit steigender Kanalanzahl zunehmend langsamer skalieren. Letzterer Umstand beruht maßgeblich auf einem unerwartet langsamen Zugriff auf Array-Elemente des vergleichsweise jungen Systems Postgres-XL. Da die Selektion einzelner Elemente aus kleinen Arrays bereits mehrere Sekunden benötigte, kann hier von einem Fehlverhalten des Systems ausgegangen werden. Zusätzliche Tests auf anderen parallelen Systemen mit ähnlicher Funktionalität sind daher für diesen Ansatz nötig. Neben diesem Umstand sind die Datenbank-Implementationen, bei unterschiedlich ansteigender linearer Skalierung, im Allgemeinen schneller als die reinen Python-Implementationen. Zusätzlich zeigt sich, dass der Push-Down des Time-Merges in PostgreSQL sehr gewinnbringend ist. Die Online-Konvertierung von Rohdaten ist, durch die zusätzlich nötigen Operationen, wie zu erwarten geringfügig langsamer. In Postgres-XL ist in diesem Fall aus ungeklärten Gründen die Rohdaten-Speicherung schneller. Dieses Verhalten ist abermals mutmaßlich unoptimiert und sollte in künftigen Versionen oder anderen Systemen erneut getestet werden. Die Ergebnisse der Suche nach Intervallen in Zeitreihen, die spezifizierte Bedingungen erfüllen, (Grundanwendung 2) sind hierbei strukturell ähnlich und wurden aus diesem Grund nicht weiter ausgeführt.
Für die Auswertung der dritten Grundanwendung wurde eine Metadatenanfrage zur Selektion von Messungen unter spezifizierten Bedingungen getestet. Zu letzteren zählten hierbei:

- die Existenz einer Menge von Kanälen,
- die Kanäle sind mit spezifizierter Abtastfrequenz aufgenommen,
- die Messungen besitzen eine spezifizierte Mindestlaufzeit und
- die Messungen sind innerhalb einer spezifizierten Zeitspanne (Datum) aufgenommen.

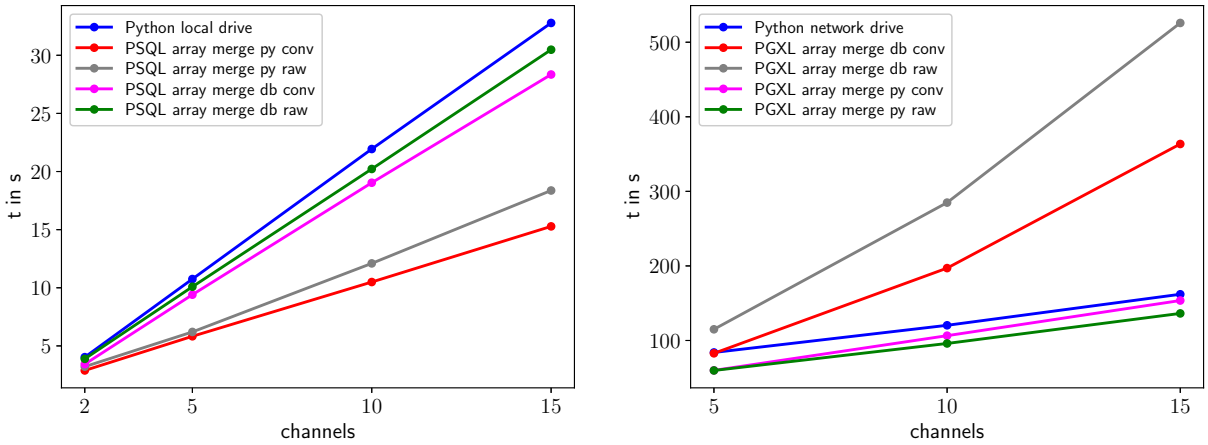


Abbildung 9.20: Grafische Darstellung der Laufzeiten der Selektion von Kanälen aus 5 Messungen mit Time-Merge und Visualisierung je Messdatei (Anwendung 1). Unterschieden wird die Auswertung im lokalen Setup (links) und Cluster-Setup (rechts). Grafik erstmalig veröffentlicht in [13].

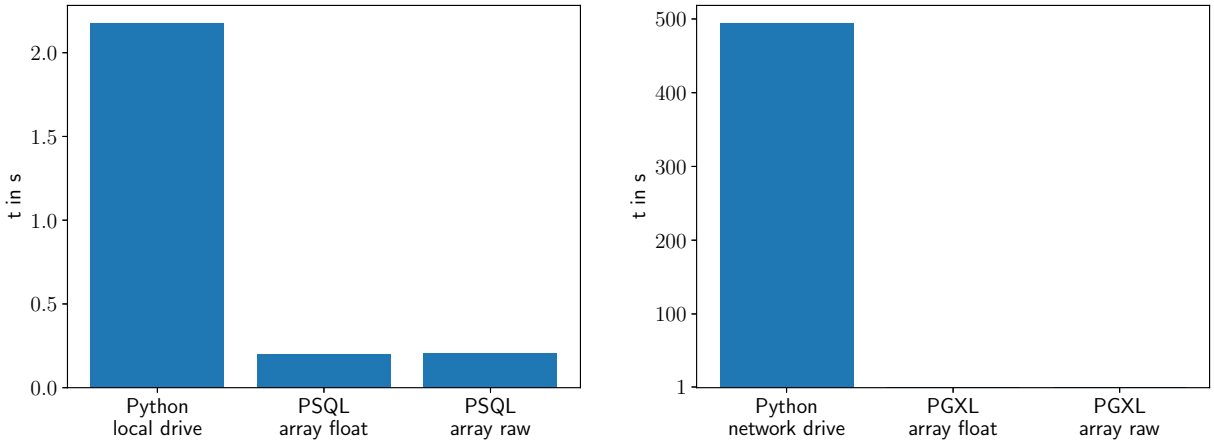


Abbildung 9.21: Grafische Darstellung der Laufzeiten der Metadatenanfragen (Anwendung 3). Unterschieden wird die Auswertung im lokalen Setup (links) und Cluster-Setup (rechts). Grafik erstmalig veröffentlicht in [13].

Die Struktur der Anfragen ist hierbei, aufgrund der einfachen Selektion über mehrere Relationen unter logisch verknüpften Bedingungen, vergleichsweise „natürlich“ in SQL umsetzbar. Diese Art der Anfrage zeichnet insbesondere eine große Selektivität und eine geringe anschließende Datenkommunikation aus. Dementsprechend gut ist die Effizienz der Datenbanklösungen in dieser Anwendung. Die zugehörigen Ergebnisse aus Abbildung 9.21 zeigen hierbei den deutlichen Gewinn, welcher in der parallelen Version noch ausgeprägter ist. Der maßgebliche Grund für letzteres ist die einfache, per Indexstrukturen optimierte, messungsübergreifende Selektion im Datenbanksystem. Diese steht im direkten Kontrast zur iterativen Verarbeitung einzelner MDF-Dateien, in welchen die nötigen Informationen durch eine vergleichsweise aufwändige Navigation durch deren komplexe Zeigerstruktur gesammelt werden müssen.
Insgesamt zeigen die Ergebnisse der SQL-basierten Verarbeitung wissenschaftlicher Methoden ein großes Potenzial für das vorgestellte Automotive-Szenario. Mit der Etablierung der guten Performance für existierende Grundanwendungen können eine Vielzahl weiterer Methoden und Szenarien untersucht werden. Hierzu zählt etwa der Mehrnutzerbetrieb, wie beispielsweise messungsübergreifende Analysen auf großen Datenbeständen oder die bereits diskutierte Indizierung von Kurvenverläufen, um Analysen von Zeitreihen in großen Datenbeständen zu erleichtern oder zu ermöglichen. Für das allgemeine Framework motiviert der Laufzeitgewinn der Zeitreihenanalysen künftige Untersuchungen zur Feature-Extraction oder dem Lernen von Machine-Learning-Modellen aus annotierten Zeitreihen (vgl. Abschnitte 10.2.1 und 10.2.2) im Kontext von Assistenzsystemen.

9.5 Vorhersagen von Schiffsrouten

Als praktische industrielle Anwendung von SQL-basiertem In-Database Machine Learning — basierend auf den Ergebnissen des vorliegenden Projektes — wird im letzten Abschnitt des Kapitels ein Projekt aus dem maritimen Bereich vorgestellt. Das Ziel der im Folgenden vorgestellten Anwendung ist es, für individuelle Schiffe vorauszusagen, welcher Hafen oder welche Häfen als nächstes angesteuert werden. Die Vorhersage der Häfen ist integraler Bestandteil eines in [14] propagierten globalen Routing-Netzwerkes, welches unter Einfluss von Wetterbedingungen Prognosen über den globalen Schiffsverkehr in Echtzeit treffen soll. Voraussetzung für das globale Routing ist hierbei, dass bis zu mehrmals täglich die Zielhäfen hunderttausender Schiffen bestimmt werden. Der Prozess zur Bestimmung des nächsten Zielhafens muss hierbei sehr leichtgewichtig sein, da das eigentliche Routing und die Visualisierung in Echtzeit umzusetzen ist.
Die Datenbasis für dieses Projekt wird durch das AIS gewonnen: das *Automatic Identification System*. Dieses wird in Kürze eingeführt, um den Kontext des zugrundeliegenden Szenarios und die Struktur der vorliegenden Daten zu beschreiben. Darauf aufbauend wird eine Diskussion

zur Modellierung von Hafenrouten mittels Markov-Ketten geführt. Die Ergebnisse der untersuchten Modelle, sowie die In-Datenbank-Implementation werden abschließend ausgewertet und diskutiert.
Alle Ausführungen in diesem Abschnitt stellen eine zusammenfassende Abhandlung von [14] dar.

9.5.1 AIS (Automatic Identification System)

Das AIS ist verbindliche Standardausrüstung für international fahrende Schiffe, welche eine Bruttoraumzahl[2] über 300 besitzen. Die Menge dieser Schiffen umfasst den Großteil des kommerziellen maritimen Verkehrs. AIS wurde ursprünglich entwickelt, um die Kollisionsgefahr mit umliegenden Schiffen in kritischen Navigationslagen zu verringern. Schiffe mit AIS broadcasten AIS-Nachrichten mittels Ultrakurzwellen und empfangen jene von naheliegenden Schiffen über ihre eigenen AIS-Receiver. Neben Schiffen werden AIS-Nachrichten heutzutage zudem über terrestrische AIS-Antennen oder AIS-Satelliten empfangen. Dadurch ist eine umfangreiche Analyse des aktuellen und historischen maritimen Verkehrs möglich, sowie eine Vorhersage künftiger Bewegungen ableitbar. AIS-Nachrichten können in drei Klassen unterteilt werden:

1. Statische Daten (Schiffsmaße, Identifikationsnummern, ...)
2. Dynamische Schiffsdaten (Position, Navigationsstatus, Geschwindigkeit, ...)
3. Reisedaten (Zielhafen, Erwartete Ankunft, maximaler Tiefgang, ...)

Aus dieser Klassifizierung ist erkennbar, dass die AIS-Reisedaten bereits einen Eintrag für den Zielhafen der aktuellen Reise enthalten. Wie in [14] beschrieben ist, wird dieser Eintrag typischerweise von Menschen per Hand gesetzt, welches in der Praxis zu einer hohen Rate verschiedenster Fehler führt. Um dieser Problematik zu entgegnen, wurde ein hybrider Ansatz propagiert: Ist der AIS-Zielhafeneintrag gültig beziehungsweise deterministisch interpretierbar, wird das zugehörige Ergebnis genutzt für aufbauende Routing-Verfahren. Ist der Eintrag nicht interpretierbar, wird der nächste Hafen mittels Markov-Kette höherer Ordnung (vergleiche Ausführungen in Abschnitt 5.1.1) vorausgesagt.

9.5.2 Markov-Ketten zur Schätzung von Zielhäfen

Die grundlegende Hypothese des Projektes ist die, dass die Sequenz der Häfen, die ein Schiff anreist, durch eine Markov-Kette höherer Ordnung mit Zustandsraum $S = \{S_1, \ldots, S_n\}$ modelliert werden kann. Solch eine Markov-Kette der Ordnung k nutzt demnach die Sequenz der letzten

[2] Die Bruttoraumzahl $\mathrm{GT} = V \cdot (0.2 + 0.02 \cdot \log_{10}(V))$ (englisch: Gross Tonnage) ist eine verbreitete Kennzahl zur Klassifizierung der Schiffsgröße, wobei V das Gesamtvolumen aller geschlossenen Räume eines Schiffes beschreibt [151].

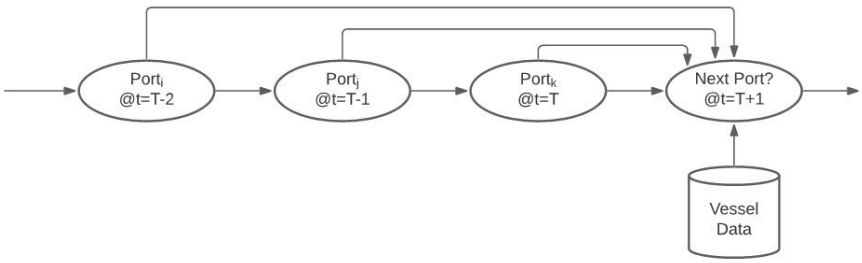


Abbildung 9.22: Beispielhafte Darstellung des Inferenzprozesses zur Schätzung des nächsten Hafens mittels Markov-Kette der Ordnung 3 und zusätzlicher Schiffsdaten.

k Hafenanfahrten, um auf den nächsten Hafen zu schließen. In Abbildung 9.22 ist dies beispielhaft für einen Prozess der Ordnung 3 dargestellt, wobei hier zudem angedeutet wird, dass die Nutzung zusätzlicher Schiffsdaten für den Inferenzprozess wertvoll ist. Wie in Abschnitt 5.1.1 beschrieben wurde, ist es möglich, Ketten höherer Ordnungen logisch in Prozesse erster Ordnung zu überführen. Hierfür werden die Zustände der letzten k diskreten Zeitschritte in einen Zustand zusammengefasst. Daraus ergibt sich der neue Zustandsraum

$$\tilde{S} = \{\ (S_{i_1}, S_{i_2}, \ldots, S_{i_k}) \mid \forall t \in \{1, \ldots, k\} :\ i_t \in \{1, \ldots, n\}\}$$

Mit diesen Zuständen ist es möglich, effektiv mit einer Zustandsübergangsmatrix $A \in \mathbb{R}^{n^k \times n^k}$ und Wahrscheinlichkeitsvektoren im Datenbanksystem, wie es detailliert in der vorliegenden Arbeit diskutiert worden ist, Vorhersagen zu treffen. Eine Vorhersage der nächsten m Häfen kann dann etwa mittels

$$\mathbf{v}_i = A^T \mathbf{v}_{i-1} \qquad i = 1, \ldots, m$$

getroffen werden, wobei der Vektor $v_i \in \mathbb{R}^{n^k}$ die Distribution der wahrscheinlichsten k-Sequenz von Hafenanläufen im i-ten Schritt beschreibt. Ist die Anfangsdistribution $\pi = \mathbf{v}_0$ der j-te kanonische Einheitsvektor — wie es im primären Anwendungsfall zur Schätzung des nächsten Hafen ist— vereinfacht sich die Multiplikation zur Selektion der j-ten Zeile von A. Die Zustandsübergangsmatrix ist in dem hier vorgestellten maritimen Kontext sehr dünn besetzt, da gemäß der etablierten See- bzw. Handelsrouten nicht jeder Hafen auf der Welt von jedem beliebigen Hafen aus angefahren wird. Mit steigender Ordnung der Kette wird das Verhältnis zwischen Nicht-Nullen und Nullen hierbei zudem vergrößert.

Basierend auf den Ergebnissen der vorliegenden Arbeit wurde in diesem Projekt daraus geschlossen, dass das vorliegende Szenario mit

- seiner dünn besetzten Struktur,
- der vergleichsweise einfache Anwendungslogik,
- der massiv parallelen Verabeitung von read-only Anfragen und
- dem sehr selektiven Charakter dieser Vorhersage-Anfragen

sehr geeignet für eine datenbankbasierte SQL-Implementation in (verteilten) relationalen Datenbanksystemen ist. Neben der guten Latenz der Anfrageverarbeitung mithilfe von Indexstrukturen wurde im industriellen Kontext

- die einfache Wartbarkeit, Portabilität und Langlebigkeit der Anwendung,
- das Einsparen von Netzwerkkosten aufgrund der Anwendungsauswertung nahe der Daten und
- das Einhalten von Datensicherheitsaspekten durch Datenbankfunktionalität (etwa Rechteverwaltung oder Verschlüsselung)

als positive Argumente für die In-Datenbank-Verarbeitung genannt.

9.5.3 Ansätze und Implementation

Da sich nach initialen Tests gezeigt hat, dass das ausschließliche Nutzen der letzten k Hafenanfahrten nur mäßige Vorhersagegenauigkeiten ermöglicht, wurden Ansätze evaluiert, die die aus den AIS-Daten gewonnenen Reisedaten bezüglich statischer Schiffsinformationen (vergleiche Abbildung 9.22) clustern. Auf jedem Cluster C wird hierzu eine eigene Zustandsübergangsmatrix $A_C \in \mathbb{R}^{n \times n}$ gelernt. Da die Clusterung bezüglich statischer Informationen geschieht und für den Vorhersageprozess pro Schiff stets genau eine Matrix benötigt wird, wäre eine physisch getrennte Speicherung der Matrizen prinzipiell möglich. Für die Implementation in relationalen Datenbanksystemen wurden jedoch alle Matrizen zusammen in einer Relation mit tensor-ähnlichem Relationenschema der Form

```
A (
   CA1 T_1,
        ⋮
   CAn_C T_n_C,
   lastport char(5),
```

```
        ⋮
    k_th_lastport char(5),
    nextport char(5),
    p double precision,
    primary key (CA1,...,CA$n_C$,lastport,...,k_th_lastport,nextport)
)
```

hinterlegt, wobei CAi das i-te Clusterattribut und $\mathbf{T}_i$ den zugehörigen Datentypen bezeichnet. Die Häfen wurden mittels ihres LOCODEs — ein aus 5 Buchstaben bestehender international standardisierter Identifikator— bestimmt.

Ausgewertet wurden in [14] zwei konkrete Fälle. Im ersten Ansatz wurden die Daten bezüglich der Schiffstypen kategorisiert. Die Klassifizierung ist eine in der maritimen Industrie etablierte Klassifizierung und unterscheidet zehn Klassen (zusätzlich eine Klasse für nicht klassifizierbare Schiffe). Das Nutzen von Schiffstypen gründet auf der Tatsache, dass verschiedene Schiffe verschiedener Typen sich oftmals erheblich in ihrem Reiseverhalten unterscheiden. So fahren Schlepper typischerweise in den gleichen Hafen ein und aus, während große Containerschiffe im Allgemeinen deutliche komplexere, internationale Routen verfolgen. Als zweiter Ansatz wurde eine Clusterung mittels der quasi-schiffsidentifizierenden *Maritime Mobile Service Identity* (MMSI) realisiert. Die MMSI besteht aus 9 Ziffern und ist quasi-eindeutig in dem Sinne, dass sie zu jedem festen Zeitpunkt eindeutig ein Schiff identifiziert. Dies gilt jedoch nicht über die Zeit, da MMSI-Nummern neu verteilt werden können.

Durch die schiffsspezifische Modellierung können individuelle Schiffsrouten nachempfunden werden. Demnach müssen jedoch ebenfalls hunderttausende Übergangsmatrizen berechnet werden, welche zwar im Allgemeinen ein höheres Verhältnis zwischen Null- und Nicht-Null-Werten besitzen als die alternativen Markov-Ansätze, aber insgesamt trotzdem deutlich mehr Speicherbedarf benötigen. Für die hier evaluierten Ordnungen und Schiffsmengen hat sich dies aufgrund der günstigen Skalierung von B-Baum-Indexstrukturen nur geringfügig auf die Latenz der Anfragen ausgewirkt. Für größere Datensätze oder höhere Ordnungen ist dies jedoch ein kritischer Punkt. Zusätzlich ist zu beachten, dass aufgrund der separierten Betrachtung das Modell eine längere Einschwingphase hat. Für Schiffe, die erst wenige Häfen angefahren haben, ist eine solide Voraussage nicht möglich. Im Gegensatz hierzu ermöglicht die Kategorisierung nach Schiffstyp eine Kandidatengenerierung von Zielhäfen, die ein Schiff auch noch nicht besucht haben muss. Zeitgleich können existente spezifische Handelsrouten mitunter nicht unter den wahrscheinlichsten Kandidaten liegen.

9.5.4 Evaluation

Wie im vorigen Unterabschnitt beschrieben, wurden drei Markov-Ketten-Ansätze untersucht. Diese bestimmen den nächsten Zielhafen entweder mittels

1. der letzten k Hafenanfahrten,
2. der letzten k Hafenanfahrten und dem Schiffstyp oder
3. der letzten k Hafenanfahrten und der MMSI.

Die Zustandsübergangsmatrix wurde mittels vorprozessierter AIS-Daten unter Berücksichtigung von Geodaten (etwa Hafengebiete) gelernt. Insgesamt wurden 50 Millionen Hafen-zu-Hafen-Reisen (Tupel) aus einem Zeitraum von 2 Jahren genutzt. Jedes Tupel entspricht einer beobachteten Schiffsreise (von Hafenabfahrt zur Hafenankunft). Neben dem Starthafen und dem Zielhafen sind zusätzlich der Schiffstyp, die MMSI, der prozessierte AIS-Eintrag zur Deklarierung des Zielhafens, sowie die letzten vorherigen 4 Hafenanfahrten (`NULL` falls nicht vorhanden) in den Trainingsdaten tupelweise zusammengetragen worden. Die prozessierten Daten beinhalteten mehr als 4000 verschiedene Häfen und über 600 000 Schiffe.
Die Zustandsübergangsmatrix wurde klassisch gemäß

$$a_{ij} = \frac{\text{Erwartete Anzahl von Übergängen von } S_i \text{ zu } S_j}{\text{Erwartete Anzahl von Übergängen ausgehend von } S_i}$$

bestimmt, wobei S_i und S_j die Zustände (Hafensequenzen und Clusterparameter) beschrieben. Eine In-Datenbank-Evaluation dieses Trainingsansatzes für Zustandsübergangsmatrizen wird in Abschnitt 10.2.2 ausführlicher diskutiert.
Die drei Ansätze wurden anhand der 20 Millionen Reisetupel evaluiert, die AIS-Zielhafen-Einträge besitzen, die nicht erfolgreich interpretiert werden konnten. Jeder der 3 Ansätze wurde mit zugrundeliegenden Markov-Ketten erster, dritter und fünfter Ordnung trainiert (d.h. unter Nutzung des letzten Hafens oder der letzten drei beziehungsweise fünf bereisten Häfen). Die Ergebnisse der Tests zur Vorhersagegenauigkeit der Ansätze sind in Abbildung 9.23 dargestellt. Hierbei wurde zudem unterschieden, ob der tatsächlich beobachtete Zielhafen exakt vom Modell vorhergesagt wurde (der Hafen mit der höchsten Wahrscheinlichkeit[3]) oder ob der Zielhafen Teil der Top 3 wahrscheinlichsten Häfen war. Letzter Ansatz umschließt offensichtlich den Exact-Match-Fall und kann beispielsweise unter Zuhilfenahme des beobachteten Kurses eines Schiffes nach der Abfahrt genutzt werden, um präzisere Vorhersagen mithilfe weiterer Beobachtungen zu treffen.

[3]Bei Nicht-Eindeutigkeit wurde der Eintrag maximaler Wahrscheinlichkeit gewählt, der den niedrigsten Index besaß.

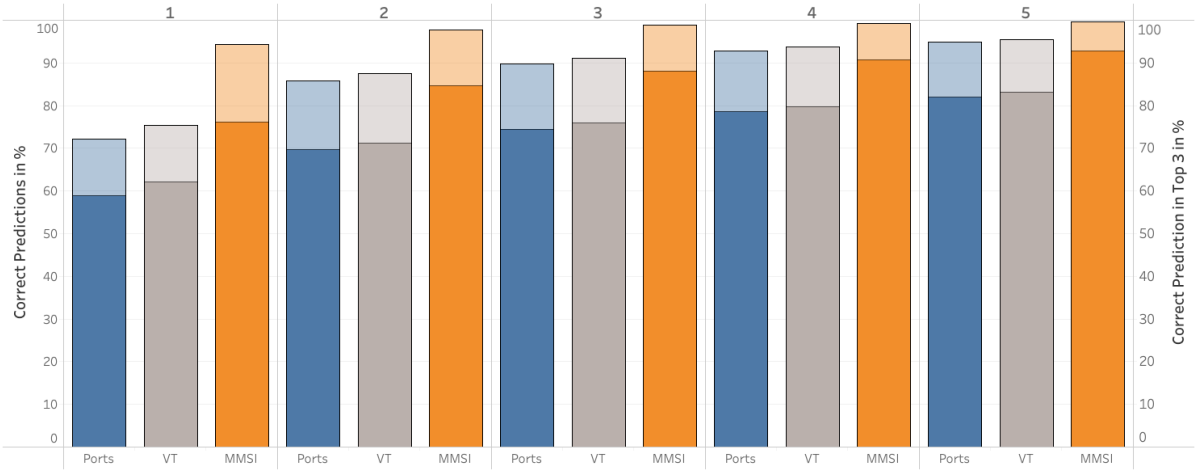


Abbildung 9.23: Darstellung der Vorhersagegenauigkeit gelernter Markov-Modelle für jene Reisen, dessen AIS-Eintrag nicht deterministisch erfolgreich interpretiert werden können. Unterschieden wird hierbei zwischen der Ordnung der zugrundeliegenden Markov-Ketten, der Clusterung und zwischen Exact-Match-Vorhersagen (undurchsichtig) und korrekten Vorhersagen innerhalb der 3 wahrscheinlichsten Häfen (durchsichtig). Die Clusterung unterscheidet zwischen reinen Hafenrouten (Blau), der Vorclusterung bezüglich des Schiffstyps (Grau) und schiffsspezifischen Markov-Modellen (Orange).

Wie in der Abbildung erkennbar ist, liegen die Vorhersagewahrscheinlichkeiten der Exact-Match-Ansätze zwischen 58% und 92 % und die Top-3-Ansätze zwischen 71 % und 99 %. Jeder der Ansätze verbessert sich mit steigender Ordnung. Insgesamt zeigen die schiffsspezifischen MMSI-Ansätze ein deutlich besseres Verhalten als die ungeclusterten und Schiffstyp-Ansätze. Die Hinzunahme von Schiffstypinformationen haben nur eine geringere Verbesserung im Vergleich zu den klassischen Markov-Ketten aufgezeigt. Aufgrund des vergleichweise großen Abstands zum MMSI-Ansatz lässt sich aus diesen Ergebnissen schließen, dass die Schiffstypenkategorisierung allein in ihrer Form ungenügend oder ungeeignet für eine Separierung des individuellen Schiffsverkehrverhaltens ist. Aus diesem Grund ist eine Untersuchung einer weiterführenden Clusterung, wie beispielsweise durch die Hinzunahme von Schiffslängen, sinnvoll. Solch ein Modell ist vor allem daher wünschenswert, da in diesem Schiffe, von denen nur begrenzt historische Routeninformationen zur Verfügung stehen, mittels der Daten ähnlicher Schiffe gerechnet werden können. Dies steht im direkten Kontrast zum MMSI-Ansatz, welcher sehr gute Vorhersagen ermöglicht, doch in der beschriebenen Weise keine noch nicht besuchten Häfen vorhersagen kann und somit mitunter lange Einschwingphasen benötigt, um akkurat etablierte Handelsrouten abbilden zu können.

Insgesamt wurden die Ergebnisse der Evaluation als äußerst positiv gedeutet. Die Modelle zeigten gute bis sehr gute Vorhersageraten und sind vergleichsweise einfach zu implementieren, zu aktualisieren und bieten aufgrund ihrer Umsetzung in relationalen Datenbanksystemen zahlreiche Vorteile, die mit diesen Systemen einhergehen. Eine Laufzeitanalyse wurde zwar in diesem Projekt nicht vorgenommen, jedoch zeigten die präsentierten Ansätze allesamt Latenzen im Millisekundenbereich für Zielhafenvorhersagen und können dementsprechend für aufbauende Echtzeitanwendungen genutzt werden. Aufbauend auf diesen Ergebnissen sollen Untersuchungen zur Verfeinerung der Schiffstypenkategorisierung vorgenommen werden. Zudem sollen mehr Trainings- und Evaluationsdaten genutzt werden. Zusätzlich ist es nötig — vor allem im schiffsspezifischen Ansatz — einen Mechanismus zu entwickeln, der alte verworfene Handelsrouten (etwa nach Wechseln des Schiffsbesitzers) aus den Zustandsübergangsmatrizen entfernt, um das Vorhersagemodell nicht zu verfälschen.

Kapitel 10

Zusammenfassung und Ausblick

Abschließend wird in diesem Kapitel eine kurze Zusammenfassung und Einordnung der Ergebnisse der vorliegenden Arbeit vorgenommen. Da hier viele initiale und grundlegende Tests und Analysen bezüglich Basisoperatoren in Datenbanksystemen umgesetzt worden sind, existieren eine Vielzahl möglicher Ansatzpunkte zur Weiterführung der Forschung. Aus diesem Grund werden nach der Zusammenfassung einige naheliegende Punkte und erste Erkenntnisse zu diesen in Kürze diskutiert. Hierbei konzentrieren wir uns maßgeblich auf die Erweiterung der Funktionalität des präsentierten Frameworks für datenbankinterne Analysen in SQL.

10.1 Zusammenfassung und Einordnung der Ergebnisse

Basis der vorliegenden Arbeit ist das grundlegende Ziel, Entwickler von Assistenzsystemen bei dem Prozess der Erstellung und Nutzung entsprechender Systeme im Big-Data-Bereich zu unterstützen. Hierfür wurde nach der Etablierung grundlegender Begriffe zur Problemeinordnung in Kapitel 2 das PArADISE-Projekt (vgl. etwa [9]) vorgestellt. Dieses ist ein mehrstufiges Framework, welches datenbankgestützte Analysen in SQL für Assistenzsysteme propagiert. Hierbei wird den Anforderungen entsprechend unterschieden zwischen der Verarbeitung von Analysen auf großen Beständen von Testdaten in parallelen relationalen Systemen und der Verarbeitung von kontinuierlich eingehenden Stromdaten in einer vertikal angeordneten Folge von SQL-Datenbanken abnehmender Funktionalität und zunehmender physischer Nähe zu den jeweiligen Sensoren. Ziel ist im letzteren Fall die Funktionalität des Systems zu sichern bei gleichzeitiger Wahrung der Datenprivatheit durch den Pushdown von Selektionen und gezielten Anfragemanipulationen. In der Entwicklungsphase hingegen werden Verfahren zur Aktivitäts- und Situationserkennung (maßgeblich durch Machine-Learning-Modelle) und allgemeine Zeitreihenanalysen in parallelen relationalen Datenbanksystemen auf großen Mengen von Sensordaten aus Testläufen durchgeführt. Diese Phase wurde in dieser Arbeit behandelt und ein Konzept zur In-

traoperatorparallelisierung in Datenbanksystemen für zugehörige Basisoperationen entwickelt. Die verbindende Zwischenstufe dieser beiden Fälle ist der Schritt der Dimensionsreduktion. In diesem soll durch die Analyse der akkumulierten Sensordaten aus der Entwicklungsphase eine Auswahl der wichtigsten Sensoren für die darauf aufbauende Nutzungsphase getroffen werden. Dieser Schritt wird im Folgeabschnitt 10.2.1 für künftige Betrachtungen andiskutiert.

Die übergreifende Prämisse des Frameworks ist die datenbankinterne Verarbeitung von Machine-Learning-Verfahren und Methoden des wissenschaftlichen Rechnens in SQL. Hierbei werden durch die weite Etablierung und jahrzehntelange Standardisierung des SQL-Kerns zahlreiche Vorteile der datenbankgestützten Verarbeitung ausgenutzt. Beispiele hierfür sind etwa die Systemunabhängigkeit, die Implementationslanglebigkeit und die einfache Anbindung zusätzlicher Schnittstellen zur Manipulation der SQL-Anfragen (etwa zur Wahrung der Datenprivatheit). Eine der wesentlichen zu untersuchenden Forschungsfragestellungen an diesem Punkt ist die Überführung der wissenschaftlichen Methoden, die klassischerweise in imperativen Programmiersprachen umgesetzt werden, in den maßgeblich deklarativen Kern von SQL.

Da die Verarbeitung solcher Methoden mittels Anfragen des SQL-Kerns vergleichsweise unerforscht ist, liegt der Fokus dieser Arbeit, nach einem Überblick über relevante Forschungs- und Industrieprojekte, zunächst nur auf der Etablierung eines einzigen Machine-Learning-Modells. Als geeigneter Vertreter für den Kontext von Assistenzsystemen wurde hierbei in Kapitel 5 das Hidden-Markov-Modell (HMM) motiviert. Dieses ist zum einen geeignet temporäre Beziehungen gut modellieren zu können und zum anderen, da sie einen Bezug zwischen internen und (durch Sensoren) beobachtbaren Zuständen erstellen. Von Vorteil ist hierbei zusätzlich, dass eine etablierte Menge von Verfahren zur Lösung von Grundproblemen existiert. Es konnte hierbei gezeigt werden, dass diese Verfahren fast ausschließlich aus einfachen Lineare-Algebra-Operatoren bestehen. Insbesondere hat sich gezeigt, dass diese Operatoren eine Untermenge der weit verbreiteten Basic Linear Algebra Subprograms (BLAS) nutzen, dessen optimierte Implementationen Basis in zahlreichen Umgebungen des wissenschaftlichen Rechnens sind. Durch eine leichte Erweiterung der aus den HMM gewonnenen Operatormenge durch punktuelle arithmetische Operatoren und allgemeine Matrizenmultiplikationen konnte der wesentliche Kern der BLAS-Bibliothek abgedeckt werden. Mit einer effizienten Überführung dieser Menge in SQL konnte dadurch ein großes Potenzial zur Verallgemeinerung des diskutierten Frameworks erreicht werden. Dies wurde und wird im weiteren Verlauf durch die Evaluation von Fourier-Transformationen in Abschnitt 9.3 oder die noch folgende Diskussion zur Feature-Extraction mittels Hauptkomponentenanalyse bekräftigt.

Mit der Spezifikation zu untersuchender Operationen wurden in Kapitel 6 mögliche Architekturen zur datenbankbasierten Unterstützung für Entwickler von Assistenzsystemen diskutiert. Hierbei wurden mögliche Abstufungen zur Überführung der wissenschaftlichen Methoden in SQL, die für den Fall ungenügender Performance reiner SQL-Implementationen genutzt werden

können, vorgestellt. Mögliche Ansatzpunkte hierbei sind etwa das Einbinden externer Software oder die Nutzung erweiterter spezifischer Datenbankfunktionalitäten. Da es wenig aussagekräftige, publizierte und systematisch durchgeführte Laufzeitvergleiche zwischen reinen SQL- und anderen datenbankgestützten Implementationen für den hier betrachteten Kontext gibt, wurde für den Verlauf der Arbeit die reine Betrachtung voller SQL-Analysen gewählt. Damit soll es insbesondere in künftigen Anwendungen möglich sein, eine geeignete Wahl der Architektur im Vorhinein zu treffen. Als ersten Schritt zur Effizienzuntersuchung von Analysen wurde darauf folgend in Kurzform eine Charakterisierung geeigneter Datenbanksysteme vorgenommen, wobei insbesondere die Stärken von column stores im gegebenen Kontext erläutert worden sind. Mit der Etablierung potenzieller Kriterien zur Wahl geeigneter Datenbanksysteme wurde im Laufe des Kapitels eine Diskussion zur relationalen Repräsentation von Matrizen und Vektoren (und bedingt Zeitreihen; vgl. etwa Abschnitt 9.4) geführt. Hierfür wurden zunächst gängige Schemata zur Datendarstellung in den bereits eingeführten BLAS-Bibliotheken präsentiert, wobei hier und im weiteren Verlauf unterschieden wurde zwischen dicht und dünn besetzen Matrizen und Vektoren. Da diese Darstellungen für imperative Programmiersprachen entwickelt wurden, sind Probleme der Überführung und mögliche Anpassungen für Relationenschemata erläutert und experimentell in verschiedenen Systemen ausgewertet worden. Hierbei sind systemabhängige Unterschiede in den Laufzeiten und der Umsetzbarkeit der Schemata (etwa aufgrund der Nutzung von Array-Datentypen) aufgetreten. Als universellstes und vielversprechendstes Schema hat sich das *Coordinate-Relationenschema* erwiesen, welches daher als Basis für die weiteren Untersuchungen gewählt wurde.

Im folgenden Kapitel 7 wurden wesentliche Aspekte zur Übersetzung von Methoden, die auf den etablierten Lineare-Algebra-Operatoren basieren, behandelt. Als erstes wurde hierfür gezeigt, dass die Operatoren in einer vergleichsweise gering erweiterten Relationenalgebra und SQL-92 dargestellt werden können. Dies motiviert zusätzlich eine Berechenbarkeit in allgemeineren relationalen Systemen, welche insbesondere bedeutungsvoll für die vertikal verteilte Datenbankhierarchie in der Nutzungsphase des PArADISE-Frameworks sein kann, da in dieser leichtgewichtige relationale Systeme mit abgeschwächter SQL-Funktionalität genutzt werden. Die Performance der etablierten Übersetzungen wurde daraufhin verglichen mit der Nutzung von Software für wissenschaftliches Rechnen (hier: R). Es konnte hierbei gezeigt werden, dass die Datenbankansätze im wesentlichen in der gleichen Komplexitätsklasse liegen, wie die der etablierten imperativen Bibliotheken. Abhängig vom System unterschieden sich dicht besetzte Probleme in den besten Fällen zwischen einer und zwei Ordnungen. Dieser Performance-Abstand konnte hierbei durch die vergleichsweise aufwändigere Speicherung in Relationen und der darauf aufbauenden Verarbeitung mittels hierfür ineffizienterer Datenbank- bzw. Relationenalgebra-Operatoren begründet werden. Ein Vergleichstest von dünn besetzten Methoden in (Sparse-)BLAS-Implementationen wurde nicht durchgeführt und sollte in folgenden Untersuchungen vorgenommen werden. Auf-

grund des geringeren Cache-Hit-Optimierungspotenzials und der größeren Ähnlichkeit genutzter Matrizenschemata und deren Verarbeitung ist ein geringerer Abstand der Datenbanklösung zu erwarten. Da dünn besetzte Probleme häufig im Kontext großer Dimensionen und Datenmengen auftreten, ist dies insbesondere für Big-Data-Anwendungen von Bedeutung.

Im weiteren Verlauf des Kapitels wurde die Komposition der einzelnen Operatoren diskutiert, wobei sich gezeigt hat, dass die Anfrageschachtelung in vielen Fällen eine deutliche Beschleunigung von Anfrageplänen erreichen kann. Zusätzlich wurde gezeigt, dass trotz logischer Optimierung wiederverwendbare Anfrageteilbäume teilweise mehrmalig ausgewertet werden. In solchen Fällen hat sich die Verwendung von *Common Table Expressions* als vorteilhaft erwiesen. Eine Umsetzung hoch-iterativer Verfahren, die nicht kompakt in SQL umgesetzt werden können (wie beispielsweise Matrix-Faktorisierungen) wurde an dieser Stelle nicht vorgenommen und sollte Bestandteil künftiger Forschungsprojekte sein. Abschließend wurden Untersuchungen zur Anfragebeschleunigung durch die Nutzung von Indexstrukturen präsentiert. Hierbei wurde gezeigt, dass diese nur begrenzt von Vorteil für die vollständige Berechnung der fundamentalen Lineare-Algebra-Operatoren sind. Im Kontext von Anwendungsszenarien, die Submatrixselektionen wie beispielsweise die Selektion von Spalten oder Zeilen beinhalten, haben sich Indexstrukturen jedoch als essenziell für die effiziente Anfrageverarbeitung gezeigt.

Nachdem ein Grundgerüst zur Umsetzung der spezifizierten Operatoren in SQL-Datenbanken etabliert wurde, ist darauf aufbauend eine effiziente Berechnung dieser auf parallelen relationalen Datenbanken in Kapitel 8 diskutiert worden. Hierfür wurden klassische Verteilungsstrategien von Matrizen und Vektoren vorgestellt und deren Einfluss auf die parallele Berechnung der Lineare-Algebra-Operatoren in Datenbanksystemen aufgezeigt. Hierbei haben wir uns auf zeilen- und spaltenweise Verteilungen sowie Blockmatrizendarstellungen konzentriert. Aufgrund der Einfachheit paralleler elementweiser Operatoren wurde hier die Umsetzung von Produkten bevorzugt. Die Blockmatrixdarstellung, welche in diesem Fall über Bereichsverteilungen auf den Zeilen- und Spaltenattributen umgesetzt werden kann, hat sich hierbei für dicht besetzte Probleme als am theoretisch effizientesten gezeigt. Im Fall dünn besetzter Probleme ist die Blockzeilendarstellung, welche mit einer einfachen Bereichsverteilung über dem Zeilenidentifizierungsattribut erreicht werden kann, ein geeigneter Ansatz. Für die Forcierung der gewünschten möglichen Intraoperatorparallelisierung wurde das in Kapitel 3 vorgestellte Konzept zur Anfragezerlegung angewandt. Es konnte hierbei gezeigt werden, dass dieses im Lineare-Algebra-Kontext prinzipiell durch vergleichsweise einfache Erweiterungen von Bereichsbedingungen in den ursprünglichen Anfragen umgesetzt werden kann.

Die vorgestellte Technik wurde daraufhin im initialen Abschnitt von Kapitel 9 experimentell ausgewertet. Hierfür wurden im parallelen row store Postgres-XL mehrere Tests vorgestellt und deren Ergebnisse und Anfrageverarbeitung analysiert. Hierbei hat sich gezeigt, dass die Nutzung der Blockpartitionierung bereits eine Beschleunigung der Matrizenmultiplikation im

Vergleich zu Zeilen- oder Spaltenpartitionierungen erreicht. Zusätzlich wurde diese Beschleunigung durch die Anwendung der Anfragezerlegungstechnik deutlich verstärkt. Beschrieben wurde hierbei zusätzlich das Auftreten mehrerer praktischer Probleme. So wurde etwa keine Bereichspartitionierung in Postgres-XL unterstützt. Dies wurde durch gezielte Einfügereihenfolgen per Round-Robin umgangen. Zusätzlich wurde in Postgres-XL eine Überlastung des Connection-Pools durch die Anwendung der Anfragezerlegung und eine generell ineffizientes Verhalten bei der Verarbeitung vererbter Relationen beobachtet. Diese Umstände schränken die Nützlichkeit des Systems und das Ergebnispotenzial, vor allem für weiterführende Betrachtungen ganzer Methoden ein und motiviert das Testen anderer parallele Systeme in der Zukunft. Im Folgeabschnitt wurde etwa gezeigt, dass das State-of-the-Art Big-Data-System Apache Spark im Falle dicht besetzter Multiplikationen deutlich performanter als Postgres-XL (mit Anfragezerlegung) ist, jedoch der parallele column store Actian VectorH (zumindest in vergleichsweise niedrigeren Dimensionen) noch schneller ist. Im Falle dünn besetzer Matrix-Vektor-Multiplikationen hat sich Postgres-XL als performanter bei ähnlicher Skalierung wie Apache Spark gezeigt. Damit können die Ergebnisse der Intraoperatorparallelisierung als weitgehend positiv bewertet und dünn besetzte Big-Data-Anwendungen als potenzieller Datenbankanwendungsfall motiviert werden.

Neben diesen Auswertungen zur Einordnung der Anfragezerlegungstechnik wurde der zweite Teil des Evaluationskapitel genutzt, um die mögliche Ausweitung und Verallgemeinerung des Konzepts für In-Datenbank-Implementationen aufzuzeigen. Im ersten von drei unabhängigen Szenarien wurde hierfür die SQL-basierte Berechnung von Fourier-Transformationen — einer viel genutzten Methode zur Frequenzanalyse von Signalen — erläutert. Hierbei wurden neben der Nutzung einfacher Ansätze für Matrix-Vektor-Multiplikationen auch erweiterte Umsetzungsmöglichkeiten, wie die komprimierte Speicherung symmetrischer Matrizen oder die Umsetzung von Fast-Fourier-Transformationen mittels rekursiver SQL-Anfragen, ausgewertet. Letztere konnte hierbei in PostgreSQL einen deutliche Performance-Gewinn gegenüber der klassischen Transformation erreichen. Da die nötigen rekursiven Anfragen im performanteren Vergleichssystem Actian Vector (column store) zu diesem Zeitpunkt nicht unterstützt worden sind, ist nach einer etwaigen Unterstützung dieser Anfrageart eine zukünftige Nachempfindung der Untersuchung sinnvoll. In diesem Fall lässt sich der Abstand zu klassischen Umgebungen des wissenschaftlichen Rechnens wohl möglich signifikant verkleinern, sodass eine SQL-basierte Umsetzung noch profitabler wäre. Abschließend wurde die Short-Time Fourier-Transformation ausgewertet, die auf kleinen Zeitfenstern Transformationen eines Signals berechnet. Dies kann genutzt werden, um zeitliche Wechsel im Frequenzspektrum zu erkennen. In diesem Fall konnte gezeigt werden, dass für ein nachempfundenes Audio-Szenario die SQL-Implementation effizienter ist als klassische Client-Datenbanken-Umsetzungen. Als zweites Anwendungsszenario des Datenbankkonzeptes wurde ein Projekt zur Zeitreihenverwaltung und -analyse von Messreihen im Automotive-Kontext vorgestellt. Im Gegensatz zu den bisherigen Untersuchungen bestand in diesem Fall

ein Großteil der nötigen Verarbeitungskosten aus Selektionen und Gruppierungen von Messreihen. Weiterführende Methoden haben sich hier vor allem auf die Homogenisierung der Daten (Interpolation auf gemeinsames Zeitraster oder Konvertierung in interpretierbare Daten) bezogen, welches thematisch tiefer liegenden sensornahen Schichten der Assistenzsystempyramide aus Abbildung 2.1 und Abschnitt 2.1 zugeordnet werden kann. Auch in diesem Szenario konnten Laufzeitgewinne durch reine Datenbanklösungen erreicht werden, wobei die zugehörigen Ansätze objekt-relationale Funktionalitäten und User Defined Functions nutzten. Die beiden letzteren Funktionalitäten wurden, aufgrund ihrer begrenzten Verfügbarkeit in Datenbanksystemen und den dadurch einschränkenden Einfluss auf die Systemunabhängigkeit und -austauschbarkeit, im eigentlich Framework vermieden. In diesem Szenario konnte jedoch gezeigt werden, dass der einhergehende potenzielle Laufzeitgewinn in manchen Szenarien essenziell für die Nutzbarkeit reiner Datenbanklösungen sein kann. Im dritten Szenario wurden die positiven Erkenntnisse vorangegangener Tests zur Berechnung dünn besetzter Matrix-Vektor-Multiplikationen in relationalen Datenbanksystemen im industriellen Kontext einer Anwendung der maritimen Logistik genutzt. In diesem Fall wurden verschiedene Ansätze von Markov-Ketten implementiert, die eine Vorhersage wahrscheinlicher Zielhäfen von Schiffen ermöglicht. Der Datenbankansatz wurde in diesem Fall präferiert, da dieser viele simultane Echtzeit-Vorhersagen ermöglicht, Nahe an den Daten agiert, einfach und transparent parallelisierbar (Interanfrage-Parallelisierung) ist und direkt in die vorhandene Infrastruktur einzubinden ist.
Abschließend lässt sich zusammenfassen, dass durch die Ausführungen in dieser Arbeit ein Rahmen für die SQL-basierte Verarbeitung von wissenschaftlichen Methoden erstellt worden ist. Da viele der Untersuchungen initialer Natur waren, existieren eine Vielzahl potenzieller aufbauender Betrachtungen. Nach einer Übersicht wesentlicher Eigenanteile der Arbeit wird im darauffolgenden Abschnitt auf einige naheliegende Vertreter eingegangen.

Übersicht wesentlicher Eigenanteile

Die Zusammenfassung wird mit einer grobgranularen Übersicht wesentlicher, eigener Forschungsanteile der vorliegenden Arbeit beendet.

- Es wurde eine konzeptuelle Überführung grundlegender Lineare-Algebra-Operatoren in relationale Sprachen — mit besonderem Fokus auf SQL — vorgestellt.
- Es wurden wesentliche Implementationsaspekte, wie geeignete Relationenschemata, die Komposition von Anfragen oder die geeignete Nutzung von Indexstrukturen untersucht und evaluiert.
- Es wurde mittels der entwickelten Überführungsparadigmen eine SQL-Übersetzung von Lösungsverfahren wesentlicher Grundanwendungen von Hidden-Markov-Modellen — ein

Machine-Learning-Modell, welches beispielsweise Anwendung im Bereich von Assistenzsystemen findet — hergeleitet und in Teilen evaluiert.

- Es wurde eine Anfragetechnik zur Beschleunigung der Verarbeitung der grundlegenden Lineare-Algebra-Operatoren in parallelen relationalen Datenbanksystemen vorgestellt und evaluiert.

- Es wurde die industrielle/praktische Anwendbarkeit des hier etablierten In-Datenbank-Ansatzes anhand von Szenarien der Automobilindustrie, der maritimen Logistik und der allgemeinen Signalverarbeitung belegt.

10.2 Ausblick

Im Folgenden werden mögliche Ansatzpunkte vorgestellt, die Potenzial für zukünftige Untersuchungen besitzen. Hierfür werden mögliche weitere Evaluationen diskutiert, die teilweise bereits im Verlauf der Arbeit angedeutet wurden.
Zunächst werden jedoch mögliche Erweiterungen der Framework-Funktionalität und erste zugehörige Erkenntnisse zu diesen vorgestellt. Die in der Arbeit etablierten Lineare-Algebra-Operatoren decken mit der Hinzunahme der Selektionen von Submatrizen und einfachen Aggregationen bereits eine große Menge an (Teil-)Methoden des wissenschaftlichen Rechnens ab. Trotzdem sind offenbar viele Operationen nicht durch diese Operatormenge effizient darstellbar. Als Beispiel wurde hierfür bereits das Automotive-Szenario in Abschnitt 9.4 vorgestellt. Bei diesem lag der Schwerpunkt der Berechnungskosten auf der gezielten Datenselektion und Interpolation von Zeitreihen. Ein weiteres bereits diskutiertes Beispiel ist die Berechnung der Fast-Fourier-Transformation mittels rekursiver Anfragen in Abschnitt 9.3.
Von diesem Punkt an sind verschiedene Erweiterungen der Operatormenge beziehungsweise Framework-Funktionalität denkbar. So könnte etwa eine vollständige Abdeckung der BLAS-Funktionalität durch vergleichsweise wenige zusätzliche Betrachtungen erreicht werden. Mit diesen könnten erweiterte Verfahren aus dem darauf aufbauenden und viel genutzten Linear Algebra Package (LAPACK), wie beispielsweise Eigenwertverfahren (vgl. einzelne SQL-Implementationen im Anhang B.6) oder Lösungsverfahren von linearen Gleichungssystemen, umgesetzt werden. Abseits vom Lineare-Algebra-Kontext ist eine Erweiterung der Funktionalität im Kontext der Aktivitätserkennung und -vorhersage und des PArADISE-Frameworks aus Kapitel 3 durch die Einbeziehung der Feature-Berechnung, dem Lernenprozess von ML-Modellen, der Evaluation der Klassifizierung oder der Erkennung signifikanter Sensoren, denkbar. Zusätzlich könnten etwa datenbankinterne Preprocessing-Schritte, wie die Erkennung von Ausreißern oder allgemeine Ansätze zur Untersuchung und Analyse der Sensordaten, gewinnbringende Erweiterungen für Entwickler sein.

Im Folgenden wird hierfür die Dimensionsreduktion und deren Verbindung zum Provenance Management im Kontext von Datenbanktechnologie erörtert. Abschließend wird das Lernen von HMM-Parametern aus annotierten Sensordaten diskutiert und erste Ergebnisse in Kürze vorgestellt. Anschließend werden mögliche zusätzliche Evaluationen diskutiert. Hierzu zählen beispielsweise die Untersuchung der Anfragezerlegung in anderen parallelen relationalen Datenbanksystemen, aber auch mögliche Anpassungen der Framework-Architektur.

10.2.1 Provenance Management

Die Dimensionsreduktion ist ein wichtiger Bestandteil der Entwicklung von Assistenzsystemen oder anderen Anwendungen im Kontext der Aktivitätserkennung. Mit ihnen wird aus der Menge der ursprünglichen Sensoren der Entwicklungsphase auf die statistisch signifikantesten Kombinationen dieser geschlossen. Dies ermöglicht eine Selektion der wesentlichen Sensoren und reduziert ultimativ die zu verarbeitende Datenmenge in der Nutzungsphase, welches vor allem für Echtzeit-Analysen essenziell ist. Die Reduktion verbindet auf diese Weise die Entwicklungs- und Nutzungsphase von Assistenzsystemen und ist aus diesem Grund im PArADISE-Framework durch seine eigene Teilphase repräsentiert. Im Kontext der Aktivitätserkennungspipeline werden vorrangig Verfahren zur Feature-Extraction und Feature-Selection genutzt (vgl. etwa [24]). Verfahren zur Feature-Extraction erstellen aus den segmentierten Signale handhabbare und bedeutsame Einheiten im Kontext der Anwendung. Ein Beispiel hierfür ist die (Short-Time) Fourier-Transformation, dessen Umsetzbarkeit bereits im Evaluationskapitel 9.3 präsentiert wurde.
Im Selection-Schritt wird eine signifikante Untermenge der berechneten Features ausgewählt, die im optimalen Fall minimal bezüglich des erfolgreichen Vorhersageprozesses ist. Ein weitverbreitetes Verfahren hierfür ist die Hauptkomponentenanalyse, dessen Ziel es ist, aus korrelierten Features eine Untermenge unkorrelierter Linearkombinationen dieser zu berechnen. Das Verfahren existiert in verschiedenen Varianten und wurde (als eines von wenigen Forschungsprojekten) bereits im Jahr 2009 in [88] in SQL diskutiert. Eine im Laufe des vorliegenden Projektes entwickelte vollständige Standard-SQL-Implementation ist im Anhang B.6.2 dargestellt. In diesem wurde das *QR-Verfahren* mit Shift-Strategie nach [107] zur Berechnung von Eigenwerten und Eigenvektoren genutzt. Das QR-Verfahren ist insbesondere Teil der bereits diskutierten *Linear Algebra Packages*.
Ein alternativer Ansatz für die Erkennung signifikanter Sensoren zur Feature-Extraction und -Selection ist das im Kontext von Datenbanksystemen genannte Provenance Management. In diesem werden die Ursprungsinformationen von Daten im Laufe oder nach deren Verarbeitung ermittelt. Klassische Anwendungsgebiete sind etwa Data Warehouses, bei denen durch den Extraktionsprozess von Daten aus verschiedenen Datenquellen und deren Transformationen eine Verifizierung oder Nachempfindung späterer Ergebnissen nötig sein kann. Im Kontext der Entwicklung von Assistenzsystemen ist eine Untersuchung des Provenance Managements interessant,

da etwa die Homogenisierung, die Feature-Extraction und die Anwendung gelernter ML-Modelle ursprüngliche Sensordaten miteinander verbindet und transformiert. Diese Prozesse erschweren daher die Auswahl der signifikantesten Sensoren. Für die konkrete Untersuchung geeigneter Provenance-Techniken gilt es zunächst eine Studie zur Anwendbarkeit über die Vielzahl der etablierten Ansätze zu führen. Als Startpunkt bietet sich beispielsweise die in [152] vorgestellte und verbreitete Why-, How- und Where-Provenance an. Die Techniken nutzen eindeutig identifizierende Annotationen von Tupeln und Attributwerten von Basisrelationen, um die Tupel zu berechnen, die für die Entstehung von Tupeln aus einer zu untersuchenden Anfrage benötigt wurden. Die Ansätze variieren hierbei in folgender Form:

- In der Why-Provenance werden Tupelidentifizierer genutzt, um für die Tupel eines Anfrageergebnis zu bestimmen, welche Tupel oder Tupelkombinationen (nach Verbünden) der Basisrelationen für das Zustandekommen des Ergebnistupels genutzt worden sind.

- Die How-Provenance berechnet mit Tupelidentifizierern nicht nur die Menge der genutzten Tupel, sondern gibt auch Aufschlüsse darüber, wie das Ergebnis entstanden ist. Hierfür wird ein Polynom aus den Identifizieren gebildet, wobei beispielsweise Verbunde durch Multiplikationen oder die Vereinigung durch Additionen der jeweiligen Tupelidentifizierer genutzt werden.

- In der Where-Provenance werden im Gegensatz zur Why-Provenance nicht die beeinflussenden Tupel berechnet, sondern die konkreten Positionen (etwa in der Form:(Relation, Tupel, Attributname)), aus der das entstehende Tupel kopiert wurde.

Für mehr Details zu den einzelnen Provenance-Techniken, sei erneut auf [152] verwiesen.
Bei künftigen Betrachtungen lässt sich voraussichtlich die How-Provenance als potenzieller Kandidat vernachlässigen, da die Art und Weise der Ergebniserstellung für die Selektion signifikanter Sensoren nur von sekundärer Bedeutung ist. Der Nutzen der Why- oder Where-Provenance ist maßgeblich abhängig von den genutzten Datenbankschemata. Liegen beispielsweise Zeitreihen im erweiterten Coordinate-Schema (vgl. das Automotive Beispiel aus Abschnitt 9.4) der vereinfachten Form `Data(Sensorname,timestamp,value,Metainformation...)` vor, ist die Nutzung der Where-Provenance ineffizient, da die Position des Zeitstempels oder des Zeitreihenwert keine hilfreichen Information für den Selektionsprozess besitzen. In diesem Fall ist die Tupelidentifizierung (Why-Provenance) ausreichend. Liegen beispielsweise die Sensordaten hingegen im (erweiterten) Attributspalten-Schemas der Form `(t,SensorA,SensorB,...)` aus Abschnitt 6.4.1 vor, ist die Lokalität bezüglich des Attributs entscheidend um die nötigen Sensoren zu unterscheiden.
Problematisch für potenzielle Untersuchungen zur Dimensionsreduktion mittels Provenance Management ist, dass die vorgestellten Techniken nicht invariant bezüglich der Formulierung der

zu untersuchenden Anfragen sind. Zusätzlich ist die Natur der Lineare-Algebra-Operatoren kritisch für die direkte Umsetzung der Provenance-Methoden. Dies kann leicht am Beispiel eines Matrix-Vektor-Produktes $\mathbf{v} = A\mathbf{w}$ mit den Einträgen

$$v_i = \sum_{k=1}^{n} a_{ik} w_k$$

gezeigt werden. Die präsentierten Provenance-Techniken würden in diesem Fall für das Tupel zu v_i die Tupel (oder Attribute) der i-ten Zeile von A, sowie den gesamten Vektor $\mathbf{w}$ errechnen. Dabei wird nicht der eigentliche Wert und deren Anteil in der entsprechenden Summe berücksichtigt, sodass in diesem Fall keine Reduktion der Dimension erreicht werden kann. Im Kontext von Matrizen mit Wahrscheinlichkeitseinträgen (etwa die Matrixparameter von HMMs) ist hierbei eine Untersuchung denkbar, in der die Teilprodukte $a_{ij}w_k$ sukzessiv bezüglich steigender Schwellwerte abgeschnitten werden. Demnach könnte

$$v_i = \sum_{k=1}^{n} f_\varepsilon(a_{ik} w_k)$$

mit

$$f_\varepsilon : [0,1] \mapsto [0,1]$$
$$f_\varepsilon(x) = \begin{cases} x & \text{falls } x > \varepsilon \\ 0 & \text{sonst} \end{cases}$$

genutzt werden. Alternativ könnten die Top-k Einzelprodukte genutzt werden bezüglich des relativen Anteils der einzelnen Summanden:

$$\frac{a_{ij} w_j}{\sum_{k=1}^{n} a_{ik} w_k}$$

Dieser Ansatz könnte ebenfalls vergleichsweise einfach auf allgemeine Matrizen übertragen werden, durch Betrachtung absoluter Produktwerte

$$\frac{|a_{ij} w_j|}{\sum_{k=1}^{n} |a_{ik} w_k|}.$$

Mittels Sortierung der einzelnen Summanden wäre in diesem Fall auch eine Auswahl der k-größten Absolutwerte, die aufaddiert einen Schwellwert des relativen absoluten Anteils erreichen (etwa 99% der absoluten Summe $\sum_{k=1}^{n} |a_{ik} w_k|$), möglich.
Ob diese Anpassungsstrategien bezüglich der klassischen Provenance-Techniken zielführend sind und inwiefern negative Seiteneffekte hierbei auftreten, gilt es an konkreten Anwendungen zu

untersuchen.
Ein Abschneiden der eigentlichen Matrixeinträge hingegen ist im Allgemeinen ungenügend, da hohe Einträge von **w** durch Multiplikation mit kleinen Werten von A auch signifikanten Einfluss besitzen können. Dies würde etwa im Kontext von Transitionsmatrizen in HMM zu fehlerhaftem Verhalten führen, da etwa alle möglichen Zustandsübergänge in einem Schritt abgeschnitten werden könnten.
Die Multiplikation ist hierbei offensichtlich nur eines von vielen Beispielen, die untersucht werden müssen. Aufgrund des hohes Anwendungspotenzials sind weiterführende Betrachtung jedoch sinnvoll.

10.2.2 Lernen von Modellen

Mit der Etablierung der Linearen-Algebra-Operationen in SQL wurde die Berechnung der drei Grundprobleme von Hidden-Markov-Modellen aus Abschnitt 5.2 grundlegend ermöglicht. Eine Auswahl verschiedener SQL-Implementationen ist für diese im Anhang B.7 hinterlegt. Dort wird ebenfalls eine Anfrageplanerstellung für den Baum-Welch-Algorithmus vorgestellt, welcher ein Lösungsverfahren zur Schätzung optimaler HMM-Parameter ist. In diesem Fall werden die Transitions- und Beobachtungsmatrix sowie die Startverteilung anhand einer oder mehrerer Beobachtungsfolgen so angenähert, dass die Wahrscheinlichkeit der Beobachtungsfolge lokal maximiert wird.
Ein weiteres wichtiges Szenario zum Lernen der HMM-Parameter ist das der annotierten Testdaten. In diesem Fall werden den Sensordaten zu jedem Zeitstempel die aktuellen Systemzustände automatisch oder manuell zugeordnet. Mit diesem Wissen können die Parameter ebenfalls geschätzt werden. Die resultierende Güte dieser ist hierbei offensichtlich abhängig von der Szenarienabdeckung der genutzten Testdaten, sodass im Allgemeinen für moderate Zustandsräume bereits enorme Datenmengen benötigt werden.
Initiale Tests für einen solchen Lernprozess wurden hierfür im Zuge des präsentierten Forschungsprojektes in [9] vorgenommen. In diesem wurde die Berechnung der Transitionsmatrix gemäß der einfachen Schätzungsformel

$$a_{ij} = \frac{\text{Erwartete Anzahl von Übergängen von } S_i \text{ zu } S_j}{\text{Erwartete Anzahl von Übergängen ausgehend von } S_i}$$

aus Abschnitt 5.2 in SQL beschrieben und experimentell ausgewertet. Die Formel wurde in diesem Fall mittels der, in dieser Arbeit nicht näher diskutierten, Sichten mittels

```
create view tmp as
    select a1.v as pre, a2.v as suc, count(a1.v) as v
    from anno a1 join anno a2 on a1.i = a2.i-1
```

```
    group by a1.v, a2.v

create view denom as
    select pre, sum(v) as v
    from tmp
    group by pre

select t1.pre, t1.suc, cast( t1.v as double ) / t2.v as v
from tmp t1 join denom t2 on t1.pre = t2.pre
```

konzeptuell geschachtelt umgesetzt. Hierbei wurde das stark vereinfachte Schema `anno(i int, v character varying)` mit der Zeitstempelnummer `i` und den annotierten Zuständen `v` genutzt. Eine Umsetzung kann aber auch mittels Anfrageschachtelung überführt werden, welche in diesem Fall voraussichtlich zu den gleichen Anfrageplänen führt. Diese erste Implementation wurde in MonetDB 11.19.9 umgesetzt und mit einem klassisch imperativen — BLAS nutzenden — Ansatz in R 3.3.1 verglichen. Die R-Umsetzung wurde dem in Beispiel 5 beschriebenen Forschungsprojekt [109, 153, 154] entnommen und ist in Anhang B.6.1 dargestellt. Beide Implementationen liefen auf dem Notebook mit den Hardwarespezifikationen aus Tabelle 6.1. In dem Test wurden die Transitionsmatrix mittels Testdaten variierender Größe (`n`) angelernt. Hierbei wurden die Daten so konstruiert, dass die Übergangswahrscheinlichkeiten eines 26-elementigen Zustandsraum gleichverteilt sind. Die zugehörigen Ergebnisse sind in Abbildung 10.1 dargestellt. In diesen ist die vergleichsweise gute Performance von MonetDB erkennbar. Der deutliche Abstand wird in Anwendungen zusätzlich verstärkt, da für die R-Implementation nur die reine Berechnungszeit nach dem Laden der Daten in den Hauptspeicher dargestellt ist. Die Ladezeit benötigte in diesem Fall, abhängig von der Datengröße, die 2- bis 16-fache Zeit der eigentlichen Berechnungszeit. Neben der guten Performance der Datenbankimplementation zeigt die Kernstruktur annotierter Daten und der daraus resultierenden Anfrage ebenso ein großes Potenzial zur parallelen Verarbeitung. Da die SQL-Implementation maßgeblich auf gruppierten, distributiven Aggregationen basiert, ist eine transparente Ausnutzung der Intraoperatorverarbeitung durch parallele Datenbanksysteme auch ohne Anfragezerlegung wahrscheinlich. Hierfür wird lediglich eine Bereichspartitionierung der Sensordaten bezüglich der Zeitstempel benötigt, da die Folgebeziehung der Zustände lokal benötigt wird.
Das Lernen der Beobachtungsmatrix B kann gemäß der Formel

$$b_{ij} = \frac{\text{Erwartete Anzahl von Zustand } S_i \text{ und gleichzeitiger Beobachtung } V_j}{\text{Erwartete Anzahl von Zuständen in } S_i}$$

aus Abschnitt 5.2 ähnlich zur Transitionsmatrix in SQL übersetzt werden und ist in [9] beschrieben. Hierbei wird in den Anwendungsszenarien eine Schlussfolgerung der Beobachtungs-

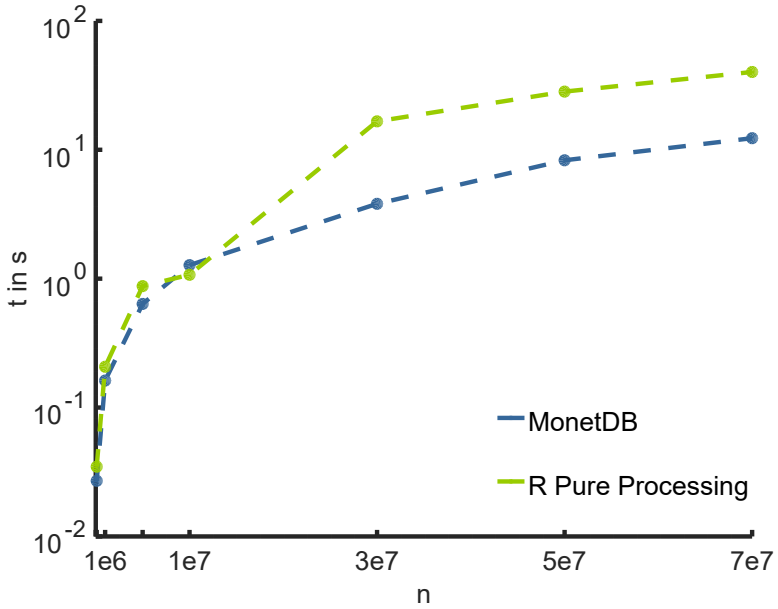


Abbildung 10.1: Berechnung einer Transitionsmatrix eines HMM aus annotierten Testdaten. Die Ergebnisse wurden in [9] erstmalig veröffentlicht.

symbole benötigt, die typischerweise aus Sensordaten berechnet werden müssen. Inwiefern dies effizient in SQL-Systemen umgesetzt werden kann, ist abhängig vom jeweiligen Szenario. In [9] wurde beispielsweise beschrieben, inwiefern es in SQL möglich ist, durch die Berechnung von Abstandsnormen Symbole aus ortsbestimmenden Sensordaten zu ermitteln. Hierbei kann insbesondere auch die Behandlung zeitlich asynchroner Sensormessungen von Bedeutung sein, so wie es beispielsweise durch die Interpolation von Zeitreihen im Automotive-Beispiel aus Abschnitt 9.4 diskutiert wurde. Demnach ist eine weiterführende Betrachtung individueller Anwendungsszenarien und Analysen von (asynchronen aufgenommen) Zeitreihen im Kontext von HMMs oder anderen Machine-Learning-Modellen zur Unterstützung von Assistenzsystem-Entwicklern sinnvoll.

10.2.3 Weiterführende Auswertungen

Neben der Erweiterung zusätzlicher Funktionalitäten wurden in der Arbeit viele initiale Tests durchgeführt. Auf den Ergebnissen dieser können aufbauende Untersuchungen angesetzt werden, die weiteren Aufschluss über das Potenzial SQL-basierter Datenbankimplementationen in der Praxis bringen. Im Folgenden werden mehrere solcher Ansatzpunkte in Kurzform aufgegriffen.

Untersuchung anderer paralleler relationaler Datenbanksysteme

Wie im Abschnitt 9.1 beschrieben wurde, sind während der Auswertung paralleler Ansätze in Postgres-XL mehrere einschränkende Umstände des Systems aufgekommen. Neben dem Aspekt der ungünstigeren tupelweisen Speicherstruktur wurden mehrere essenzielle Funktionalitäten nur bedingt unterstützt. Hierzu zählt vornehmlich das Fehlen einer (mehrattributigen) Bereichspartitionierung, die dem System ermöglichen soll, konsistent Zwischenergebnisse weiter zu verarbeiten. Zudem haben Probleme bei der Freigabe des Connection-Pools zur Einschränkung der simultanen Verarbeitung größerer Mengen von Anfragen geführt, welches die Umsetzung der präsentierten Anfragezerlegungsstratiegen aus Kapitel 8 deutlich einschränkte.
Die Evaluation weiterer paralleler Systeme kann demnach Aufschluss darüber geben, inwiefern sich die vorgestellte Strategie der Anfragezerlegung systemunabhängig positiv auswirkt. Hierbei sind Untersuchungen feinerer aufgeteilter Anfragen möglich und Analysen, die überprüfen, inwiefern die jeweiligen Anfrageoptimierungsschnittstellen selbst effiziente Verarbeitungsstrategien entwickeln. Dafür könnten etwa Anfrageschachtelungen in Kombination mit Bereichspartitionierungen ausgewertet und untersucht werden. Zusätzlich gilt es hier zu untersuchen, inwiefern in den jeweiligen Systemen die Nutzung von CTEs nötig ist, um eine effiziente Wiederverwertung von Zwischenergebnissen (im parallelen Kontext) zu erreichen (vgl. Abschnitt 7.3).
Eine detailliertere Untersuchung paralleler column stores würde zusätzlich, aufgrund der voraussichtlich besseren Performance gegenüber row stores, einen wichtigen Einblick für das Potenzial zur parallelen Verarbeitung von linearer Algebra in Datenbanken geben und zudem einen besseren Vergleich zu etablierten Big-Data-Umgebungen (siehe unten) ermöglichen.

Untersuchung weiterer spezieller Datenbanksysteme

Neben weiterführenden Untersuchungen bezüglich klassischer paralleler relationaler Systeme ist ein Vergleich von Special-Purpose-Datenbanksystemen oder Systemen mit speziellen Erweiterungen sinnvoll. Hierunter zählen etwa die Untersuchung von Datenbanksystemen mit transparenter GPU-Unterstützung (vgl. Abschnitt 6.2), welche durch ihre uneingeschränkte Standard-SQL-Unterstützung großes Potenzial für das vorgestellte Framework besitzen.
Trotz der Einschränkungen, die durch das Nutzen nicht-standardisierter SQL-Dialekte oder UDFs (vgl. Diskussion aus Abschnitt 3.1) entstehen, ist auch die Untersuchung etablierter Datenbanksysteme sinnvoll, die zusätzliche Spezialfunktionalitäten bieten. Hierzu zählen vor allem Zeitreihen-Datenbanken, wie beispielsweise der auf PostgreSQL basierten Open-Source-Datenbank TimeScaleDB [155], und etablierte Systeme, die Lineare-Algebra- oder Machine-Learning-Funktionalitäten in der Anfrageschnittstelle ermöglichen. Zu letzteren zählt beispielsweise das parallele relationale System SciDB [102], welches etwa Matrix-Datentypen und zugehörige Funktionen (wie die Matrizenmultiplikation) in das relationale System und deren An-

frageverarbeitung eingebunden hat. Hierbei werden insbesondere bereits parallele Berechnungen der Matrixoperationen (in einer Shared-Nothing-Architektur) unterstützt.
Die Untersuchung solcher Systeme sollte einen Aufschluss darüber geben, inwiefern und wie hoch ein Performance-Gewinn durch die Nutzung spezieller Funktionalitäten im Vergleich zu Standard-SQL-Verarbeitung auf geeigneten modernen Datenbanksystemen ist. Hierbei sollten insbesondere auch Anwendungsszenarien berücksichtigt werden, wie die Short-Time Fourier-Transformation aus Abschnitt 9.3, dessen Aufwand nicht nur auf der reinen Berechnung wissenschaftlicher Methoden beruht, sondern auch ein signifikantes Maß an Datenverwaltung benötigt. Dies ist vor allem daher wichtig, da die reine Berechnung wissenschaftlicher Methoden im Datenbanksystem nur einen Teil des Aufgabenfeldes des Datenbanksystems darstellt. Eine reine Beschleunigung von Linearen-Algebra-Operationen kann etwa nicht immer fehlende Funktionalitäten oder schwächere Performance im Datenmanagement und -zugriff ausgleichen.

Vergleich zu etablierten Big-Data-Umgebungen

Zusätzlich zum Vergleich spezieller Datenbanksysteme ist eine deutlich ausgeweitete Evaluation von Big-Data-Umgebungen, wie Tensorflow, Apache Spark, Apache Flink oder Apache Hadoop nötig. Hiermit soll eine bessere Einschätzung der Laufzeiten des Datenbankansatzes gegenüber klassischen State-of-the-Art-Workflows ermöglicht werden.
Im Zuge der Entwicklung des PArADISE-Frameworks wurde hierfür beispielsweise in einem Studentenprojekt in [4] der in der Einführung präsentierte Vergleichstest von *Stonebraker et al.* [3] von parallelen Datenbanksystemen und Hadoop/MapReduce im modernen Kontext nachempfunden. In dem Projekt wurden die Laufzeiten von Postgres-XL mit denen von Apache Spark (vgl. Abschnitt 4.3.2) und Apache Flink bezüglich derselben drei Operationen (Grep-Task, Web Log, Verbund) aus dem ursprünglichen Test verglichen. Hierbei war Postgres-XL im Web-Log-Szenario und der Verbundberechnung deutlich schneller als die beiden Big-Data-Umgebungen, welches als Verifizierung der ursprünglichen Prämisse der *Stonebraker*-Gruppe interpretiert werden kann. In zusätzlichen Auswertungen in [145] wurde hingegen gezeigt, dass — abhängig vom Szenario — Implementationen in Apache Spark mitunter stark beschleunigt werden können, wenn Parquet- anstatt CSV-Dateien genutzt werden. Weitere Untersuchungen bezüglich des Einflusses von Datenquellen auf Apache-Spark-Laufzeiten oder auf denen anderer Big-Data-Umgebungen sind demnach nötig.
Von diesem Punkt an ist etwa eine weitreichende Vergleichsstudie zu den Methoden von Hidden-Markov-Modellen mit den in dieser Arbeit etablierten Datenbankansätzen sinnvoll. Hierbei kann durch die Betrachtung des Lernens von Parametern und den Methoden zur Aktivitätsvorhersage und -erkennung eine gute Balance zwischen datenintensiven, klassischen SQL-Methoden und komplexeren Verfahren des wissenschaftlichen Rechnens erreicht werden. Wie in den hier beschriebenen Tests mehrfach experimentell gezeigt worden ist, ist hierfür eine Auswertung auf

einem parallelen column store mit besseren Partitionierungsmöglichkeiten vorzuziehen, da diese potenziell um ein Vielfaches schneller als Postgres-XL sein könnten.
Von besonderem Interesse ist hierbei auch die Untersuchung hoch-iterativer wissenschaftlicher Verfahren, die eine große Menge komplexer von einander unabhängiger Operationen durchführt. In diesem Fall ist, aufgrund der begrenzten standardisierten Iterationsmöglichkeiten im SQL-Kern und der lokalen Nutzung von Lineare-Algebra-Bibliotheken in Big-Data-Systemen wie Apache Spark (vgl. [84]), ein Nachteil von parallelen Datenbanksystemen zu vermuten. Sollte sich dies bestätigen, sind weiterführende Kombinationen von parallelen Datenbanksystemen mit dem (teilweise SQL-unterstützenden) Apache Spark denkbar. Initiale Tests in [145] haben im Fall von Postgres-XL jedoch gezeigt, dass von Spark angefragte Daten in Postgres-XL auf einem einzelnen Knoten gesammelt und daraufhin verschickt worden sind. Dies ist speziell für daten-intensive Fälle im Vergleich zu verteilt gespeicherten Dateien im HDFS ineffizient. Inwiefern dieses Verhalten umgangen werden kann und wie beziehungsweise wann die Kombination durch Vorfilterungen von Datenbanksystemen gewinnbringend ist, könnte in künftigen Betrachtungen ergründet werden.

Anhang A

Literaturverzeichnis

[1] S. Arndt, "Presenting a Reference Integration Data Model for Sensors in the context of Industrial Internet of Things," Master's thesis, TU Dresden, 02 2017.

[2] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks*, vol. 30, no. 1-7, pp. 107–117, 1998. [Online]. Available: https://doi.org/10.1016/S0169-7552(98)00110-X

[3] M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "MapReduce and parallel DBMSs: friends or foes?" *Commun. ACM*, vol. 53, no. 1, pp. 64–71, 2010. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629197

[4] D. Dietrich, O. Fenske, S. Schomacker, P. Schweers, and A. Heuer, "Stonebraker versus Google: 2-0 Scores in Rostock - A Comparison of Big Data Analytics Environments (Stonebraker gegen Google: Das 2: 0 fällt in Rostock)," in *Proceedings of the 30th GI-Workshop Grundlagen von Datenbanken, Wuppertal, Germany, May 22-25, 2018*, 2018, pp. 101–107. [Online]. Available: http://ceur-ws.org/Vol-2126/paper16.pdf

[5] M. Winslett and V. Braganholo, "Dan Suciu Speaks Out on Research, Shyness and Being a Scientist," *SIGMOD Rec.*, vol. 46, no. 4, pp. 28–34, Feb. 2018. [Online]. Available: http://doi.acm.org/10.1145/3186549.3186557

[6] A. Heuer, "METIS in PArADISE Provenance Management bei der Auswertung von Sensordatenmengen für die Entwicklung von Assistenzsystemen," in *Datenbanksysteme für Business, Technologie und Web (BTW 2015) - Workshopband, 2.-3. März 2015, Hamburg, Germany*, 2015, pp. 131–136. [Online]. Available: https://dl.gi.de/20.500.12116/2375

[7] D. Marten and A. Heuer, "A framework for self-managing database support and parallel computing for assistive systems," in *Proceedings of the 8th ACM International Conference*

on PErvasive Technologies Related to Assistive Environments, PETRA 2015, Corfu, Greece, July 1-3, 2015, F. Makedon, Ed. ACM, 2015, pp. 25:1–25:4. [Online]. Available: http://doi.acm.org/10.1145/2769493.2769526

[8] ——, "Transparente Datenbankunterstützung für Analysen auf Big Data," in *Proceedings of the 27th GI-Workshop Grundlagen von Datenbanken, Gommern, Germany, May 26-29, 2015.*, 2015, pp. 36–41. [Online]. Available: http://ceur-ws.org/Vol-1366/paper8.pdf

[9] ——, "Machine Learning on Large Databases: Transforming Hidden Markov Models to SQL Statements," *Open Journal of Databases (OJDB)*, vol. 4, no. 1, pp. 22–42, 2017. [Online]. Available: https://www.ronpub.com/ojdb/OJDB_2017v4i1n02_Marten.html

[10] D. Marten, H. Meyer, D. Dietrich, and A. Heuer, "Sparse and Dense Linear Algebra for Machine Learning on Parallel-RDBMS Using SQL," *OJBD*, vol. 5, no. 1, pp. 1–34, 2019. [Online]. Available: https://www.ronpub.com/ojbd/OJBD_2019v5i1n01_Marten.html

[11] D. Marten, H. Meyer, and A. Heuer, "Calculating Fourier transforms in SQL," in *Advances in Databases and Information Systems - 23nd European Conference, ADBIS 2019, Bled, Slovenia, September 8-11, 2019, Proceedings*, 2019.

[12] ——, "Database support for automotive analysis," in *Proceedings of the Conference "Lernen, Wissen, Daten, Analysen", LWDA 2019, Berlin Germany, 2019.*, 2019.

[13] ——, "Database support for automotive analysis," Chair of Database and Information Systems, Rostock University, Rostock, Germany, technical report, September 2019. [Online]. Available: http://eprints.dbis.informatik.uni-rostock.de/995/

[14] D. Marten, C. Hilgenfeld, and A. Heuer, "Scalable In-Database Machine Learning for the Prediction of Port-to-Port Routes," *Journal für Mobilität und Verkehr*, vol. 6, pp. 3–10, 2020. [Online]. Available: https://journals.qucosa.de/jmv/article/view/42

[15] K. Yordanova, S. Lüdtke, S. Whitehouse, F. Krüger, A. Paiement, M. Mirmehdi, I. Craddock, and T. Kirste, "Analysing Cooking Behaviour in Home Settings: Towards Health Monitoring," *Sensors*, vol. 19, no. 3, p. 646, 2019. [Online]. Available: https://doi.org/10.3390/s19030646

[16] S. Teipel, C. Heine, A. Hein, F. Krüger, A. Kutschke, S. Kernebeck, M. Halek, S. Bader, and T. Kirste, "Feasibility of a multidimensional assessment of challenging behavior in advanced stages of dementia in the naturalistic environment of nursing homes - the insideDEM framework," *Alzheimer's & Dementia, Diagnosis, Assessment & Disease Monitoring*, vol. 8, pp. 36–44, 2017.

[17] A. Heuer, "METIS in PArADISE Provenance Management bei der Auswertung von Sensordatenmengen für die Entwicklung von Assistenzsystemen," Universität Rostock, Tech. Rep., 2015. [Online]. Available: http://eprints.dbis.informatik.uni-rostock.de/544/1/dbis-tr-cs-03-15.pdf

[18] A. Heuer, T. Kirste, W. Hoffmann, and D. Timmermann, "Coast: Concept for proactive assistive systems and technologies," Fakultät für Informatik und Elektrotechnik der Universität Rostock, Bericht, 2006.

[19] H. Grunert and A. Heuer, "Privacy Protection through Query Rewriting in Smart Environments," in *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, E. Pitoura, S. Maabout, G. Koutrika, A. Marian, L. Tanca, I. Manolescu, and K. Stefanidis, Eds. OpenProceedings.org, 2016, pp. 708–709. [Online]. Available: https://doi.org/10.5441/002/edbt.2016.98

[20] ——, "Rewriting Complex Queries from Cloud to Fog under Capability Constraints to Protect the Users' Privacy," *Open J. Internet Things*, vol. 3, no. 1, pp. 31–45, 2017. [Online]. Available: https://nbn-resolving.org/urn:nbn:de:101:1-2017080613421

[21] ——, "Query Rewriting by Contract under Privacy Constraints," *Open J. Internet Things*, vol. 4, no. 1, pp. 54–69, 2018. [Online]. Available: https://www.ronpub.com/ojiot/OJIOT_2018v4i1n05_Grunert.html

[22] T. Umblia, A. Hein, I. Bruder, and T. Karopka, "MARIKA: A Mobile Assistance System for Supporting Home Care," in *Proceedings of the 1st International Workshop on Mobilizing Health Information to Support Healthcare-related Knowledge Work - Volume 1: Workshop MobiHealthInf, (BIOSTEC 2009)*, INSTICC. SciTePress, 2009, pp. 69–77.

[23] A. Hein, A. Hoffmeyer, and T. Kirste, "Utilizing an Accelerometric Bracelet for Ubiquitous Gesture-Based Interaction," in *Universal Access in Human-Computer Interaction. Intelligent and Ubiquitous Interaction Environments, 5th International Conference, UAHCI 2009, Held as Part of HCI International 2009, San Diego, CA, USA, July 19-24, 2009. Proceedings, Part II*, 2009, pp. 519–527. [Online]. Available: https://doi.org/10.1007/978-3-642-02710-9_57

[24] A. Bulling, U. Blanke, and B. Schiele, "A Tutorial on Human Activity Recognition Using Body-worn Inertial Sensors," *ACM Comput. Surv.*, vol. 46, no. 3, pp. 33:1–33:33, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2499621

[25] H. T. Anh and L. M. Pereira, "State-of-the-art of intention recognition and its use in decision making," *AI Commun.*, vol. 26, no. 2, pp. 237–246, 2013. [Online]. Available: https://doi.org/10.3233/AIC-130559

[26] S. Bader and T. Kirste, "A Tutorial Introduction to Automated Activity and Intention Recognition," in *Lecture Notes for the Interdisciplinary Colleg, Rostock University*, 2011 - enthalten auf digitalen Anhang.

[27] F. Krüger, K. Yordanova, C. Burghardt, and T. Kirste, "Towards Creating Assistive Software by Employing Human Behavior Models," *Journal of Ambient Intelligence and Smart Environments*, vol. 4, pp. 209–226, 05 2012.

[28] F. Krüger, K. Yordanova, A. Hein, and T. Kirste, "Plan Synthesis for Probabilistic Activity Recognition," in *ICAART 2013 - Proceedings of the 5th International Conference on Agents and Artificial Intelligence*, vol. 2, 02 2013.

[29] K. Yordanova, S. Whitehouse, A. Paiement, M. Mirmehdi, T. Kirste, and I. Craddock, "What's cooking and why? Behaviour recognition during unscripted cooking tasks for health monitoring," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, March 2017, pp. 18–21.

[30] F. Krüger, M. Nyolt, K. Yordanova, A. Hein, and T. Kirste, "Computational State Space Models for Activity and Intention Recognition. A Feasibility Study," *PLOS ONE*, vol. 9, no. 11, pp. 1–24, 11 2014. [Online]. Available: https://doi.org/10.1371/journal.pone.0109381

[31] K. Yordanova, S. Bader, F. Krüger, A. Hein, and T. Kirste, "Automatic Detection of Challenging Behaviour of People with Dementia and Identification of Intervention Strategies - Requirements and Challenges," in *Proceedings of the European Project Space on Intelligent Technologies, Software engineering, Computer Vision, Graphics, Optics and Photonics*, 02 2016, pp. 166–180.

[32] A. Hein, T. Low, M. Hensch, T. Kirste, and A. Nürnberger, "Gesture spotting for controlling a mobile assistance system for service and maintenance," in *INFORMATIK 2012*, U. Goltz, M. Magnor, H.-J. Appelrath, H. K. Matthies, W.-T. Balke, and L. Wolf, Eds. Bonn: Gesellschaft für Informatik e.V., 2012, pp. 549–560.

[33] G. Saake, K.-U. Sattler, and A. Heuer, *Datenbanken - Konzepte und Sprachen, 4. Auflage.* MITP, 2010.

[34] A. Heuer, G. Saake, and K. Sattler, *Datenbanken - Konzepte und Sprachen, 6. Auflage.* MITP, 2018. [Online]. Available: https://mitp.de/IT-WEB/Datenbanken/Datenbanken-Konzepte-und-Sprachen-oxid.html

[35] G. Saake, A. Heuer, and K.-U. Sattler, *Datenbanken: Implementierungstechniken (mitp Professional)*, 3rd ed. mitp, 2011. [Online]. Available: http://www.amazon.de/Datenbanken-Implementierungstechniken-Professional-Gunter-Saake/dp/3826614380

[36] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Third Edition.* Springer, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-8834-8

[37] International Organization for Standardization, "Information technology database languages – SQL – Part 15: Multi-dimensional arrays (SQL/MDA)," International Organization for Standardization, Geneva, CH, Standard, 6 2019, Key: ISO/IEC 9075-15:2019.

[38] V. Köppen, G. Saake, and K. Sattler, *Data Warehouse Technologien, 2. Auflage.* MITP, 2014.

[39] M. Stonebraker, "The Case for Shared Nothing," *IEEE Database Eng. Bull.*, vol. 9, pp. 4–9, 1985.

[40] J. Han, J. Pei, and M. Kamber, *Data Mining, Southeast Asia Edition*, ser. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2006. [Online]. Available: https://books.google.de/books?id=AfL0t-YzOrEC

[41] S. King, *Big Data: Potential und Barrieren der Nutzung im Unternehmenskontext.* Springer Fachmedien Wiesbaden, 2014. [Online]. Available: https://books.google.de/books?id=-aAuBAAAQBAJ

[42] F. X. Diebold, X. Cheng, S. Diebold, D. Foster, M. Halperin, S. Lohr, J. Mashey, T. Nicholas, M. Pai, M. Pospiech, F. Schorfheide, and M. Shin, "A Personal Perspective on the Origin(s) and Development of "Big Data": The Phenomenon, the Term, and the Discipline," 2019.

[43] D. Klein, P. Tran-Gia, and M. Hartmann, "Big Data," *Informatik-Spektrum*, vol. 36, no. 3, pp. 319–323, Jun 2013. [Online]. Available: https://doi.org/10.1007/s00287-013-0702-3

[44] D. Fasel and A. Meier, Eds., *Big Data: Grundlagen, Systeme und Nutzungspotenziale*, ser. Edition HMD. Wiesbaden: Springer Vieweg, 2016.

[45] H. Grunert and A. Heuer, "Datenschutz im PArADISE," *Datenbank-Spektrum*, vol. 16, no. 2, pp. 107–117, 2016. [Online]. Available: https://doi.org/10.1007/s13222-016-0216-7

[46] Auge, Tanja and Heuer, Andreas , "ProSA - Using the CHASE for Provenance Management," in *Advances in Databases and Information Systems - 23rd European Conference, ADBIS 2019, Bled, Slovenia, September 8-11, 2019, Proceedings*, ser. Lecture Notes in Computer Science, T. Welzer, J. Eder, V. Podgorelec, and A. K. Latific, Eds., vol. 11695. Springer, 2019, pp. 357–372. [Online]. Available: https://doi.org/10.1007/978-3-030-28730-6_22

[47] PostgreSQL Web Team, "Cyclic Tag System," Accessed: 19 August 2019 - im digitalen Anhang enthalten. [Online]. Available: https://wiki.postgresql.org/wiki/Cyclic_Tag_System

[48] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008.

[49] O. Bartunov and S. Teodor, "Generalized Inverted Index," July 2006, Presentation at the PostgreSQL Summit Toronto - im digitalen Anhang enthalten.

[50] PostgreSQL Global Development Group, "PostgreSQL 9.6 Documentation, GIN index," 2017. [Online]. Available: https://www.postgresql.org/docs/9.6/static/gin-implementation.html

[51] A. Heuer, G. Saake, and K. Sattler, *Datenbanken - Implementierungstechniken, 4. Auflage*. MITP, 2019. [Online]. Available: https://www.mitp.de/IT-WEB/Datenbanken/Datenbanken-Implementierungstechniken.html

[52] The PostgreSQL Global Development Group, "Official PostgreSQL10 Documentation: Inheritance," Accessed: 14 December 2020. [Online]. Available: https://www.postgresql.org/docs/10/tutorial-inheritance.html

[53] The Postgres-XL Global Development Group, "Official Postgres-XL 10 Documentation: Inheritance," Accessed: 14 December 2020. [Online]. Available: https://www.postgres-xl.org/documentation/ddl-inherit.html

[54] J. Dongarra, K. Jakub, J. Langou, J. Langou, H. Ltaief, P. Luszczek, A. YarKhan, W. Alvaro, M. Faverge, A. Haidar, J. Hoffman, E. Agullo, A. Buttari, and B. Hadri, *PLASMA Users' Guide*, 2nd ed., September 2010. [Online]. Available: http://icl.cs.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf

[55] Intel Corporation, *Intel® Math Kernel Library Developer Reference C*, 023rd ed., 2019.

[56] ——, *Intel® Math Kernel Library Developer Reference FORTRAN*, 023rd ed., 2019.

[57] R Core Team, *R Internals*, October 2019, Accessed: 28th October 2019. [Online]. Available: https://cran.r-project.org/doc/manuals/r-release/R-ints.pdf

[58] C. Moler, "MATLAB Incorporates LAPACK," 2000, Accessed: 28th October 2019 - im digitalen Anhang enthalten. [Online]. Available: https://de.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html

[59] J. W. Eaton, *GNU Octave - Official Documentation*, Accessed: 28th October 2019. [Online]. Available: https://octave.org/doc/interpreter/

[60] The SciPy community, *SciPy Reference Guide - Linear algebra (scipy.linalg)*, Accessed: 27th September, 2019 - im digitalen Anhang enthalten. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html

[61] ——, *NumPy Reference Guide - Linear algebra (numpy.linalg)*, Accessed: 26th July, 2019 - im digitalen Anhang enthalten. [Online]. Available: https://numpy.org/devdocs/reference/routines.linalg.html

[62] C. L. Lawson, R. Hanson, D. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for FORTRAN usage," *ACM Trans. Math. Softw.*, vol. 5, pp. 308–323, 09 1979.

[63] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 14, no. 1, pp. 1–17, 1988.

[64] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 3 1990. [Online]. Available: http://doi.acm.org/10.1145/77626.79170

[65] S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[66] S. Corliss, G. Demmel, J. Dongarra, J. Duff, I. Hammarling, S. Henry, G. Heroux, M. Hu, C. Kahan, W. Kaufmann, L. Kearfott, B. Krogh, F. Li, X. Maany, Z. Petitet, A. Pozo, R. Remington, K. Walster, W. Whaley, C. Wolff, J. v. Gudenberg, and A. Lumsdaine, *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*, University of Tennessee. [Online]. Available: www.netlib.org/blas/blast-forum

[67] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, "A proposal for a set of parallel basic linear algebra subprograms," in *Applied Parallel Computing*

Computations in Physics, Chemistry and Engineering Science, J. Dongarra, K. Madsen, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 107–114.

[68] J. Leskovec, A. Rajaraman, and J. Ullman, *Mining of Massive Datasets*. Cambridge University Press, 2020. [Online]. Available: https://books.google.de/books?id=OefRhZyYOb0C

[69] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, 2004, pp. 137–150.

[70] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major technical advancements in apache hive," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 1235–1246. [Online]. Available: https://doi.org/10.1145/2588555.2595630

[71] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine, "Simulation of Database-Valued Markov Chains Using SimSQL," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 637–648. [Online]. Available: https://doi.org/10.1145/2463676.2465283

[72] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine, "Scalable Linear Algebra on a Relational Database System," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, April 2017, pp. 523–534.

[73] Y. Tian, S. Tatikonda, and B. Reinwald, "Scalable and Numerically Stable Descriptive Statistics in SystemML," in *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, 2012, pp. 1351–1359. [Online]. Available: https://doi.org/10.1109/ICDE.2012.12

[74] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on MapReduce," in *2011 IEEE 27th International Conference on Data Engineering*, April 2011, pp. 231–242.

[75] S. Günnemann, "Machine Learning Meets Databases," *Datenbank-Spektrum*, vol. 17, 01 2017.

[76] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 135–146. [Online]. Available: https://doi.org/10.1145/1807167.1807184

[77] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 439–455. [Online]. Available: https://doi.org/10.1145/2517349.2522738

[78] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The Stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec 2014. [Online]. Available: https://doi.org/10.1007/s00778-014-0357-y

[79] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds. USENIX Association, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[80] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on Apache Spark," *I. J. Data Science and Analytics*, vol. 1, no. 3-4, pp. 145–164, 2016. [Online]. Available: https://doi.org/10.1007/s41060-016-0027-9

[81] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742797

[82] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-tolerant Streaming Computation at Scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP

'13. New York, NY, USA: ACM, 2013, pp. 423–438. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737

[83] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A Resilient Distributed Graph System on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13. New York, NY, USA: ACM, 2013, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2484425.2484427

[84] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine Learning in Apache Spark," *CoRR*, vol. abs/1505.06807, 2015. [Online]. Available: http://arxiv.org/abs/1505.06807

[85] R. Bosagh Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia, "Matrix computations and optimization in apache spark," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 31–38.

[86] 2ndQuadrant, "Postgres-XL official website - Overview," 2020, Accessed: 22th March 2020 - im digitalen Anhang enthalten. [Online]. Available: https://www.postgres-xl.org

[87] The PostgreSQL Global Development Group, *PostgreSQL 12.2 Documentation*, 2020.

[88] M. Navas and C. Ordonez, "Efficient computation of PCA with SVD in SQL," in *Proceedings of the 2nd ACM SIGKDD Workshop on Data Mining using Matrices and Tensors, Paris, France, June 28, 2009*, 2009.

[89] Y. Zhang, H. Herodotou, and J. Yang, "RIOT: I/O-Efficient Numerical Computing without SQL," *CoRR*, vol. abs/0909.1766, 2009.

[90] E. Meijer, B. Beckman, and G. M. Bierman, "LINQ: reconciling object, relations and XML in the .NET framework," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, S. Chaudhuri, V. Hristidis, and N. Polyzotis, Eds. ACM, 2006, p. 706. [Online]. Available: https://doi.org/10.1145/1142473.1142552

[91] G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers, "Haskell Boards the Ferry - Database-Supported Program Execution for Haskell," in *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. Hage and M. T. Morazán, Eds., vol. 6647. Springer, 2010, pp. 1–18. [Online]. Available: https://doi.org/10.1007/978-3-642-24276-2_1

[92] Y. Wang, Y. Yang, W. Zhu, Y. Wu, X. Yan, Y. Liu, Y. Wang, L. Xie, Z. Gao, W. Zhu, X. Chen, W. Yan, M. Tang, and Y. Tang, "SQLFlow: A Bridge between SQL and Machine Learning," *CoRR*, vol. abs/2001.06846, 2020. [Online]. Available: https://arxiv.org/abs/2001.06846

[93] C. Duta, D. Hirn, and T. Grust, "Compiling PL/SQL Away," in *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p1-duta-cidr20.pdf

[94] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005, pp. 225–237. [Online]. Available: http://cidrdb.org/cidr2005/papers/P19.pdf

[95] J. Lajus and H. Mühleisen, "Efficient Data Management and Statistics with Zero-copy Integration," in *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, ser. SSDBM '14. New York, NY, USA: ACM, 2014, pp. 12:1–12:10.

[96] P. Große, W. Lehner, T. Weichert, F. Färber, and W.-S. Li, "Bridging Two Worlds with RICE Integrating R into the SAP In-Memory Computing Engine," *PVLDB*, vol. 4, pp. 1307–1317, 2011.

[97] C. Ordonez, N. Mohanam, and C. Garcia-Alvarado, "PCA for large data sets with parallel data summarization," *Distributed and Parallel Databases*, vol. 32, pp. 377–403, 09 2013.

[98] J. M. Hellerstein, C. Ré, F. Schoppmann, Z. D. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The MADlib Analytics Library or MAD Skills, the SQL," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-38, Apr 2012.

[99] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann, "SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases," in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß, Eds. OpenProceedings.org, 2017, pp. 84–95. [Online]. Available: https://doi.org/10.5441/002/edbt.2017.09

[100] M. Schüle, F. Simonis, T. Heyenbrock, A. Kemper, S. Günnemann, and T. Neumann, "In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory

Database Systems," in *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs ,,Datenbanken und Informationssysteme"(DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*, 2019, pp. 247–266. [Online]. Available: https://doi.org/10.18420/btw2019-16

[101] M. Schüle, M. Bungeroth, D. Vorona, A. Kemper, S. Günnemann, and T. Neumann, "ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python," in *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, 2019, pp. 562–565. [Online]. Available: https://doi.org/10.5441/002/edbt.2019.56

[102] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The Architecture of SciDB," in *Scientific and Statistical Database Management - 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011. Proceedings*, ser. Lecture Notes in Computer Science, J. B. Cushing, J. C. French, and S. Bowers, Eds., vol. 6809. Springer, 2011, pp. 1–16. [Online]. Available: https://doi.org/10.1007/978-3-642-22351-8_1

[103] M. Stonebraker, "SciDB: An Open-Source DBMS for Scientific Data," *ERCIM News*, vol. 2012, no. 89, 2012. [Online]. Available: http://ercim-news.ercim.eu/en89/special/scidb-an-open-source-dbms-for-scientific-data

[104] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, Feb 1989.

[105] S. J. Russell and P. Norvig, *Künstliche Intelligenz - ein moderner Ansatz, 3. Auflage.* Pearson Studium, 2012. [Online]. Available: https://www.pearson-studium.de/kunstliche-intelligenz.html

[106] S. Dörn, *Markov-Modelle.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 219–290. [Online]. Available: https://doi.org/10.1007/978-3-662-54304-7_6

[107] J. Stoer and R. Bulirsch, *Numerische Mathematik 2: Eine Einführung - unter Berücksichtigung von Vorlesungen von F.L.Bauer*, ser. Springer-Lehrbuch. Springer Berlin Heidelberg, 2005. [Online]. Available: https://books.google.de/books?id=_TPRZ9pabGcC

[108] A. N. Langville and C. D. Meyer, *Google's PageRank and beyond - the science of search engine rankings.* Princeton University Press, 2006.

[109] M. Giersich, T. Heider, and T. Kirste, "AI Methods for Smart Environments - A Case Study on Team Assistance in Smart Meeting Rooms," in *Constructing Ambient Intelligence - AmI 2007 Workshops Darmstadt, Germany, November 7-10, 2007 Revised Papers*, 2007, pp. 4–13. [Online]. Available: https://doi.org/10.1007/978-3-540-85379-4_2

[110] B. Pfister and T. Kaufmann, *Hidden-Markov-Modelle*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 107–136. [Online]. Available: https://doi.org/10.1007/978-3-662-52838-9_6

[111] P. F. Felzenszwalb, D. P. Huttenlocher, and J. M. Kleinberg, "Fast algorithms for large-state-space HMMs with applications to web usage analysis," in *Advances in neural information processing systems*, 2004, pp. 409–416.

[112] S. M. Siddiqi and A. W. Moore, "Fast Inference and Learning in Large-state-space HMMs," in *Proceedings of the 22Nd International Conference on Machine Learning*, ser. ICML '05. New York, NY, USA: ACM, 2005, pp. 800–807. [Online]. Available: http://doi.acm.org/10.1145/1102351.1102452

[113] A. Jones, R. Stephens, R. Plew, R. Garrett, and A. Kriegel, *SQL Functions Programmer's Reference*, ser. Programmer to Programmer. Wiley, 2005. [Online]. Available: https://books.google.de/books?id=rWJY1bwC7vUC

[114] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for declarative large-scale machine learning," *Commun. ACM*, vol. 62, no. 5, pp. 83–91, 2019. [Online]. Available: https://doi.org/10.1145/3318221

[115] R Development Staff, "The R Project," 2016. [Online]. Available: https://www.r-project.org

[116] K. Sattler, A. Kemper, T. Neumann, and J. Teubner, "DFG Priority Program SPP 2037: Scalable Data Management for Future Hardware," in *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs ,,Datenbanken und Informationssysteme"(DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*, 2019, pp. 273–276. [Online]. Available: https://doi.org/10.18420/btw2019-ws-29

[117] T. Gubner, D. Tomé, H. Lang, and P. Boncz, "Fluid Co-processing: GPU Bloom-filters for CPU Joins," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, ser. DaMoN'19. New York, NY, USA: ACM, 2019, pp. 9:1–9:10. [Online]. Available: http://doi.acm.org/10.1145/3329785.3329934

[118] V. Rosenfeld, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Performance Analysis and Automatic Tuning of Hash Aggregation on GPUs," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, ser. DaMoN'19. New York, NY, USA: ACM, 2019, pp. 8:1–8:11. [Online]. Available: http://doi.acm.org/10.1145/3329785.3329922

[119] PG-Strom Development Team, "PG-Strom official homepage," Accessed: 16th December 2019. [Online]. Available: https://heterodb.github.io/pg-strom/

[120] A. Eisenberg and J. Melton, "SQL: 1999, Formerly Known As SQL3," *SIGMOD Rec.*, vol. 28, no. 1, pp. 131–138, Mar. 1999. [Online]. Available: http://doi.acm.org/10.1145/309844.310075

[121] N. Kamat and A. Nandi, "A Closer Look at Variance Implementations In Modern Database Systems," *SIGMOD Rec.*, vol. 45, no. 4, pp. 28–33, May 2017. [Online]. Available: http://doi.acm.org/10.1145/3092931.3092936

[122] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in MonetDB," *Commun. ACM*, vol. 51, no. 12, pp. 77–85, 2008. [Online]. Available: http://doi.acm.org/10.1145/1409360.1409380

[123] D. J. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, 2008, pp. 967–980. [Online]. Available: https://doi.org/10.1145/1376616.1376712

[124] Netlib Repository, "BLAS - Basic Linear Algebra Subprograms," 2019. [Online]. Available: http://www.netlib.org/blas/

[125] Y. Saad, "SPARSKIT: a basic tool kit for sparse matrix computations - Version 2," Research Inst. for Advanced Computer Science; Moffett Field, CA, United States, Tech. Rep., 1994.

[126] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1994. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611971538

[127] C. Tuerker, *SQL: 1999 & SQL: 2003 – Objektrelationales SQL, SQLJ & SQL/XML*. Heidelberg: dpunkt.verlag, 2003.

[128] The PostgreSQL Global Development Group, *PostgreSQL 11.1 Documentation*, 2018.

[129] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The Case for Learned Index Structures," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 489–504. [Online]. Available: http://doi.acm.org/10.1145/3183713.3196909

[130] L. Sidirourgos and M. Kersten, "Column Imprints: A Secondary Index Structure," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 893–904. [Online]. Available: https://doi.org/10.1145/2463676.2465306

[131] Actian Corporation, *Vector 5.1 Documentation*, Accessed: 6th January 2020. [Online]. Available: https://docs.actian.com/vector/5.1/index.html#page/Welcome%2FVector_5.1_Guides.htm%23

[132] C. von Kutzleben, "The Actian Vector Database Kernel," Actian Corporation, Sponsor Tutorial in *18. Fachtagung für „Datenbanksysteme für Business, Technologie und Web"*, Rostock, 2019 - im digitalen Anhang enthalten.

[133] ——, "Mitteilung der Actian Corporation zur Performance von Primärindexstrukturen in Actian Vector," Erhalten am: 12 Mai 2020 - im digitalen Anhang enthalten.

[134] Apache Software Foundation, "Apache Spark MLlib - Data Types - RDD-based API - im digitalen Anhang enthalten," Accessed: 6th November 2019. [Online]. Available: https://spark.apache.org/docs/2.3.0/mllib-data-types.html

[135] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK User's Guide*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[136] J. Dongarra, R. van de Geijn, and D. Walker, "A look at scalable dense linear algebra libraries," in *Proceedings Scalable High Performance Computing Conference SHPCC-92.*, April 1992, pp. 372–379.

[137] O. Zienkiewicz, R. Taylor, and D. Fox, "The Finite Element Method for Solid and Structural Mechanics," in *The Finite Element Method for Solid and Structural Mechanics*, 7th ed. Oxford: Butterworth-Heinemann, 2014.

[138] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. New York, NY, USA: ACM, 1969, pp. 157–172. [Online]. Available: http://doi.acm.org/10.1145/800195.805928

[139] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM journal on matrix analysis and applications*, vol. 11, no. 3, pp. 430–452, 1990.

[140] The Postgres-XL Global Development Group, *Postgres-XL 10r1.1 Documentation*, 2018.

[141] D. Dietrich, "Vergleich zeilen- und spaltenorientierter DBMS als Basis für die Parallelisierung von Vektorraumoperationen auf einem Cluster-Rechner," Bachelor's Thesis, Universität Rostock, December 2018. [Online]. Available: http://eprints.dbis.informatik.uni-rostock.de/970/

[142] M. Lamster, "Vergleich paralleler Datenbanksysteme und Big-Data-Umgebungen für Hidden-Markov-Modelle," Bachelor's Thesis, Universität Rostock, April 2020.

[143] Actian Corporation, *Vector in Hadoop 5.0 Documentation*, 2018.

[144] Apache Software Foundation, "Machine Learning Library (MLlib) Programming Guide - Data Types - RDD-based API - Spark 2.4.5 Documentation," Accessed: 24th March 2020 - im digitalen Anhang enthalten. [Online]. Available: https://spark.apache.org/docs/2.3.0/mllib-data-types.html

[145] A. Lutsch, "Effiziente Datenvorbereitung für Analysen in Automotive-Anwendungen," Bachelor's Thesis, Universität Rostock, April 2019. [Online]. Available: http://eprints.dbis.informatik.uni-rostock.de/989/

[146] M. Wong, *Discrete Fourier Analysis*, ser. Pseudo-Differential Operators. Springer Basel, 2011.

[147] K. R. Rao, D. N. Kim, and J.-J. Hwang, *Fast Fourier Transform - Algorithms and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[148] C. Weihs, U. Ligges, F. Mörchen, and D. Müllensiefen, "Classification in music research," *Advances in Data Analysis and Classification*, vol. 1, no. 3, pp. 255–291, Dec 2007.

[149] Association for Standardization of Automation and Measuring Systems, "ASAM MDF," 2019. [Online]. Available: https://www.asam.net/standards/detail/mdf

[150] Vector Informatik GmbH, *Format Specification MDF Format Version 3.3.1*, 2014.

[151] I. M. Organization, *International Conference on Tonnage Measurement of Ships, 1969: Final Act of the Conference, with Attachments, Including the International Convention on Tonnage Measurement of Ships, 1969*, ser. IMCO publication. IMO, 1983. [Online]. Available: https://books.google.de/books?id=uGfgAp58gZ4C

[152] J. Cheney, L. Chiticariu, and W.-c. Tan, "Provenance in Databases: Why, How, and Where," *Foundations and Trends in Databases*, vol. 1, pp. 379–474, 01 2009.

[153] F. Krüger, K. Yordanova, A. Hein, and T. Kirste, "Plan Synthesis for Probabilistic Activity Recognition," in *ICAART (2)*, 2013, pp. 283–288.

[154] F. Krüger, K. Yordanova, C. Burghardt, and T. Kirste, "Towards creating assistive software by employing human behavior models," *Journal of Ambient Intelligence and Smart Environments*, vol. 4, no. 3, pp. 209–226, 2012.

[155] Timescale Corporation, "TimeScaleDB official homepage," Accessed: 3th February 2020. [Online]. Available: https://www.timescale.com/

[156] W. Kahan, "Pracniques: Further Remarks on Reducing Truncation Errors," *Commun. ACM*, vol. 8, no. 1, p. 40, Jan. 1965. [Online]. Available: https://doi.org/10.1145/363707.363723

[157] J. Shlens, "A Tutorial on Principal Component Analysis," *CoRR*, vol. abs/1404.1100, 2014. [Online]. Available: http://arxiv.org/abs/1404.1100

[158] M. Journée, Y. Nesterov, P. Richtárik, and R. Sepulchre, "Generalized Power Method for Sparse Principal Component Analysis," *J. Mach. Learn. Res.*, vol. 11, p. 517–553, Mar. 2010.

[159] J. Stoer and F. Bauer, *Numerische Mathematik: eine Einführung - unter Berücksichtigung von Vorlesungen von F.L. Bauer*, ser. Springer-Lehrbuch. Springer-Verlag, 1994. [Online]. Available: https://books.google.de/books?id=hh7vAAAAMAAJ

[160] J. Wilkinson, *The Algebraic Eigenvalue Problem*, ser. Monographs on numerical analysis. Clarendon Press, 1988. [Online]. Available: https://books.google.de/books?id=5wsK1OP7UFgC

Anhang B

Programmcodes und zusätzliche Betrachtungen

In diesem Abschnitt werden Zusatzinformationen, geordnet nach der Kapitelstruktur der Kernarbeit, präsentiert. Abschließend werden Skripte zur Erstellung von SQL-Anfragen für die Verarbeitung von Hidden-Markov-Modellen vorgestellt.

B.1 Hidden-Markov-Modelle

Im folgenden Beispiel wird anhand des erweiterten Wettermodells aus Abschnitt 5.2 der Baum-Welch-Algorithmus dargestellt.

Beispiel 8 (Erweitertes Wettermodell)**.** Es wird nun das erweiterte Wetterszenario aus Beispiel 4 erneut betrachtet. In diesem Fall soll die Beobachtung (V_1 =`Luftdruck hoch`, V_2=`Luftdruck niedrig`) als Grundlage genommen werden, jedoch diesmal zur Schätzung der HMM-Parameter A, B, π. Als Startwerte werden die im Anfangsbeispiel etablierten Matrizen

$$A = \begin{pmatrix} 0.6 & 0.3 & 0.1 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.4 & 0.4 \end{pmatrix} \qquad B = \begin{pmatrix} 0.9 & 0.1 \\ 0.3 & 0.7 \\ 0.1 & 0.9 \end{pmatrix}$$

und der Startvektor

$$\pi = \begin{pmatrix} 0.3 \\ 0.3 \\ 0.4 \end{pmatrix} \tag{B.1}$$

genutzt.

Es wird nun ein Iterationsschritt des Baum-Welch-Verfahrens im Detail berechnet. Die Forward- und Backwardvariablen sind hierbei

$$\begin{aligned}\alpha_1 = \pi \odot B_{:,o_1} &= \begin{pmatrix}0.3\\0.3\\0.4\end{pmatrix} \odot \begin{pmatrix}0.9\\0.3\\0.1\end{pmatrix}\\ &= \begin{pmatrix}0.27\\0.09\\0.04\end{pmatrix}\end{aligned}$$

$$\begin{aligned}\alpha_2 &= (\alpha_1^T A) \odot B_{:,o_2}\\ &= \left(\begin{pmatrix}0.27 & 0.09 & 0.04\end{pmatrix}\begin{pmatrix}0.6 & 0.3 & 0.1\\0.3 & 0.5 & 0.2\\0.2 & 0.4 & 0.4\end{pmatrix}\right) \odot \begin{pmatrix}0.1\\0.7\\0.9\end{pmatrix}\\ &= \begin{pmatrix}0.0197\\0.0994\\0.0549\end{pmatrix}\end{aligned}$$

$$\beta_2 = \begin{pmatrix}1\\1\\1\end{pmatrix}$$

$$\begin{aligned}\beta_1 = A(B_{:,o_2} \odot \beta_2) &= \begin{pmatrix}0.6 & 0.3 & 0.1\\0.3 & 0.5 & 0.2\\0.2 & 0.4 & 0.4\end{pmatrix}\left(\begin{pmatrix}0.1\\0.7\\0.9\end{pmatrix} \odot \begin{pmatrix}1\\1\\1\end{pmatrix}\right)\\ &= \begin{pmatrix}0.36\\0.56\\0.66\end{pmatrix}.\end{aligned}$$

Mit den Variablen können nun die γ_t-Variablen berechnet werden:

$$\gamma_1 = \frac{\alpha_1 \odot \beta_1}{\langle \alpha_1, \beta_1 \rangle} = \frac{\begin{pmatrix} 0.27 \\ 0.09 \\ 0.04 \end{pmatrix} \odot \begin{pmatrix} 0.64 \\ 0.44 \\ 0.34 \end{pmatrix}}{\begin{pmatrix} 0.27 & 0.09 & 0.04 \end{pmatrix} \begin{pmatrix} 0.64 \\ 0.44 \\ 0.34 \end{pmatrix}}$$

$$\approx \begin{pmatrix} 0.55862 \\ 0.28966 \\ 0.15172 \end{pmatrix}$$

$$\gamma_2 = \frac{\alpha_2 \odot \beta_2}{\langle \alpha_2, \beta_2 \rangle} = \frac{\begin{pmatrix} 0.0197 \\ 0.0994 \\ 0.0549 \end{pmatrix} \odot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}}{\begin{pmatrix} 0.0197 & 0.0994 & 0.0549 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}}$$

$$\approx \begin{pmatrix} 0.11322 \\ 0.57127 \\ 0.31552 \end{pmatrix}$$

Die abschließende Hilfsvariable ξ_1 ergibt sich aus

$$\begin{aligned} \xi_1 = \xi_t &= A \odot (\alpha_t \cdot (\beta_{t+1} \odot B_{:,o_{t+1}})^T) / (\alpha_t^T A (B_{:,o_{t+1}} \odot \beta_{t+1})) \\ &= \frac{\begin{pmatrix} 0.6 & 0.3 & 0.1 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.4 & 0.4 \end{pmatrix} \odot \left(\begin{pmatrix} 0.27 \\ 0.09 \\ 0.04 \end{pmatrix} \cdot \begin{pmatrix} 1 \cdot 0.1 & 1 \cdot 0.7 & 1 \cdot 0.9 \end{pmatrix} \right)}{\begin{pmatrix} 0.27 & 0.09 & 0.04 \end{pmatrix} \begin{pmatrix} 0.6 & 0.3 & 0.1 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.4 & 0.4 \end{pmatrix} \begin{pmatrix} 1 \cdot 0.1 \\ 1 \cdot 0.7 \\ 1 \cdot 0.9 \end{pmatrix}} \\ &\approx \begin{pmatrix} 0.09310 & 0.32586 & 0.13966 \\ 0.01552 & 0.18103 & 0.09310 \\ 0.00460 & 0.06437 & 0.08276 \end{pmatrix} \end{aligned}$$

Mit den berechneten Hilfsvariablen können nun die neuen Schätzungen bestimmt werden:

$$\overline{\pi} = \gamma_1 \approx \begin{pmatrix} 0.55862 \\ 0.28966 \\ 0.15172 \end{pmatrix}$$

$$\begin{aligned} \overline{A} &= (\sum_{t=1}^{T-1} \xi_t[i,j])/((\sum_{t=1}^{T-1} \gamma_t) \cdot \mathbf{1}^T) \\ &= \begin{pmatrix} 0.09310 & 0.32586 & 0.13966 \\ 0.01552 & 0.18103 & 0.09310 \\ 0.00460 & 0.06437 & 0.08276 \end{pmatrix} / \begin{pmatrix} 0.55862 & 0.55862 & 0.55862 \\ 0.28966 & 0.28966 & 0.28966 \\ 0.15172 & 0.15172 & 0.15172 \end{pmatrix} \\ &\approx \begin{pmatrix} 0.16667 & 0.58333 & 0.25000 \\ 0.05357 & 0.62500 & 0.32143 \\ 0.03030 & 0.42424 & 0.54546 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} (\overline{B})_{:,1} &= \left(\sum_{t=1}^{T} \delta_{o_t,V_1} \gamma_t\right) / \left(\sum_{t=1}^{T} \gamma_t\right) \\ &= \begin{pmatrix} 0.55862 \\ 0.28966 \\ 0.15172 \end{pmatrix} / \begin{pmatrix} 0.55862 + 0.11322 \\ 0.2896 + 0.57127 \\ 0.15172 + 0.31552 \end{pmatrix} \approx \begin{pmatrix} 0.83148 \\ 0.33645 \\ 0.32472 \end{pmatrix} \end{aligned}$$

$$(\overline{B})_{:,2} = \begin{pmatrix} 0.11322 \\ 0.57127 \\ 0.31552 \end{pmatrix} / \begin{pmatrix} 0.55862 + 0.11322 \\ 0.2896 + 0.57127 \\ 0.15172 + 0.31552 \end{pmatrix} \approx \begin{pmatrix} 0.16852 \\ 0.66355 \\ 0.67528 \end{pmatrix}$$

Mit der Abbruchbedingung

$$\|\overline{A} - A\|_F^2 + \|\overline{B} - B\|_F^2 + \|\overline{\pi} - \pi\|_2^2 < 1e-3,$$

wobei $\|.\|_F$ der Frobenius-Norm $\|B\|_F = \sqrt{\sum_{i=1}^{N} \sum_{j=1}^{M} |b_{ij}|^2}$ und $\|.\|_2$ der 2-Vektornorm $\|\pi\|_2 = \sqrt{\sum_{i=1} |\pi_i|^2}$ entspricht, konvergiert das Verfahren nach 5 Iterationen mit der (gerundeten) monoton fallenden Residuenfolge

$$(0.67387, 0.16055, 0.14039, 0.024751, 0.0009726).$$

Die finalen Parameter sind

$$A \approx \begin{pmatrix} 0 & 0.7 & 0.3 \\ 0 & 0.66 & 0.34 \\ 0 & 0.438 & 0.562 \end{pmatrix}$$

$$B \approx \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}$$

$$\pi \approx \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Aufgrund der kurzen Beobachtungsfolge nehmen die Einträge der Parameter extreme Werte an, erfüllen jedoch dabei genau die Rahmenbedingungen. Ein initialer hoher Luftdruck erhöht die Wahrscheinlichkeit sonnigen Wetters, wobei der darauf folgende Abfall des Luftdrucks Bewölkung oder Regen bedeuten kann. Da keine weiteren Beobachtungsschritte existieren, muss das Modell eine Rückkehr in sonniges Wetter nicht berücksichtigen.

B.1.1 Kahan-Summation

Die Kahan-Summation ist ein Verfahren zur numerisch stabilen Berechnung von vielelementigen Summen bzw. von Summen mit stark variierenden Größenordnungen. Die Anwendung dieses Summationsverfahren ist insbesondere im Kontext von Big Data und der Wahrscheinlichkeitstheorie von Bedeutung. So wurde beispielsweise in [73] beschrieben, wie die Kahan-Summation für eine Vielzahl von statistischen Aggregationen (Durchschnitt, Varianz, Kovarianz,...) in SystemML umgesetzt wurde. Der Algorithmus wurde in [156] vorgestellt und ist in Algorithmus 18 dargestellt.

Algorithmus 18 Kahan-Summation zur numerisch stabilen Summierung der Elemente eines Arrays x.

Input-Array: x
$s = 0.0$
$c = 0.0$
for $i = 1, \ldots, \text{len}(\mathbf{x})$ **do**
    $y = x[i] - c$
    $t = s + y$
    $c = (t - s) - y$
    $s = t$
end for

Als Beispiel für numerische Einschränkungen durch einfache Summierung sei im Folgenden eine einfache SQL-Implementation (PostgreSQL-Dialekt) dargestellt. Ziel des Anfrageplans ist die Berechnung der Summe

$$1\mathbf{e}9 + \sum_{i=1}^{1000} 0.001 = 1\mathbf{e}9 + 1.$$

Der Anfrageplan mittels einfacher Aggregation

```
create table Kahan (i int, v double precision);

insert into Kahan values (0,1e9);

with recursive test (i,v) as (
   select 1,0.001
   union all
   select i+1,0.001
   from test
   where i < 1000
)
insert into Kahan
select i,v
from test;

select sum(v)
from Kahan;
```

liefert in diesem Fall das Ergebnis 1000000001.00005. Die Kahan-Summation kann in SQL gemäß des Algorithmus 18 mittels rekursiver Anfrage der Form

```
with recursive sumrec(i,x,y) as (
   select 1,t2,t2-t1
   from (
      select v, v
      from Kahan
      where i=0
   ) temp0 (t1.t2)
   union all
   select i+1, t2, (t2-x) - t1
   from (
      select i,t1,x+t1,x
```

```
        from (
            select sumrec.i, Kahan.v - sumrec.y, sumrec.x
            from sumrec, Kahan
            where Kahan.i=sumrec.i
        ) temp1 (i,t1,x)
    ) temp2 (i,t1,t2,x)
    where temp2.i <= (select max(i) from Kahan)
)
select x
from sumrec
where i = (select max(i) from sumrec);
```

umgesetzt werden. In diesem Fall ergibt sich das korrekte Ergebnis $1e9 + 1$. Demnach ist die Umsetzung des Verfahrens in SQL vergleichsweise einfach möglich, benötigt jedoch rekursive SQL-Anfragen, welche in manchen etablierten Datenbanksystemen nicht unterstützt werden.

B.2 Datenrepräsentation

Im Folgenden wird die Erstellung einer User-Defined-Function in PostgreSQL dargestellt, die im Falle des Array-Schemas dicht besetzter Matrizen aus Abschnitt 6.4.1 für die Berechnung von Matrizenprodukten genutzt wurde. In Postgres' plpgsql wird die UDF genutzt, um ein Skalarprodukt aus zwei Arrays direkt zu berechnen, um das Nutzen des sonst nötigen **unnest** zu umgehen. Die Funktion wird wie folgt definiert:

```
create or replace function array_mat_mult_element(v double precision[],
      w double precision[])
returns double precision
as $$
declare
   erg double precision;
begin
   erg:=0.0;

   for i in 1..array_length(v,1) loop
   erg:=erg+v[i]*w[i];
   end loop;

   return erg;
```

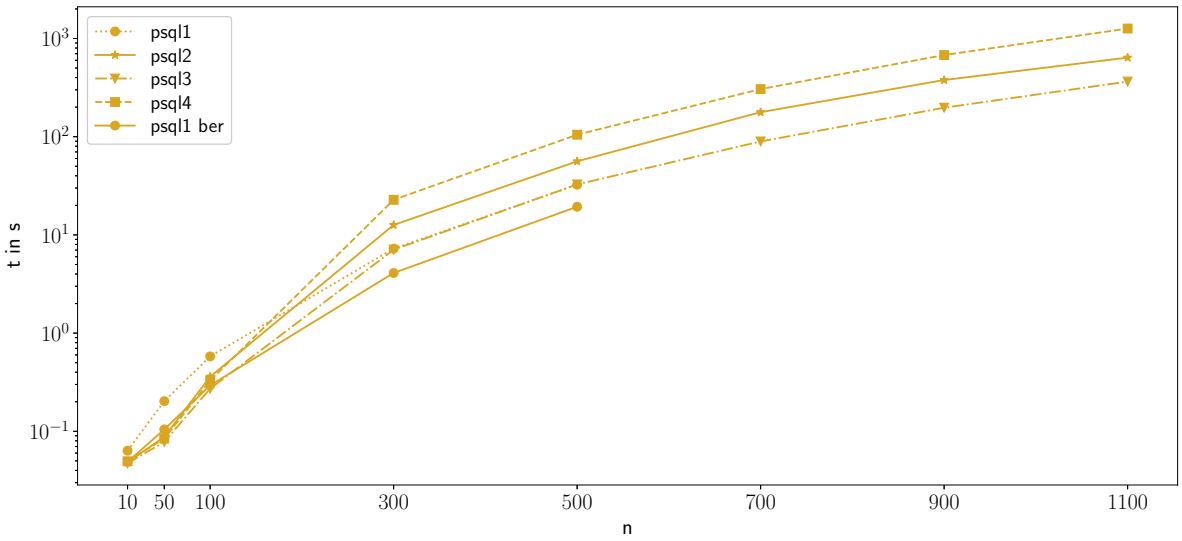


Abbildung B.1: Experimentelle Auswertung von Matrizenmultiplikationen $C = AB$ mit $A, B, C \in \mathbb{R}^{n \times n}$ und den verschiedenen Matrizenschemata (1,2,3,4) aus Abschnitt 6.4 in PostgreSQL.

```
end;
$$ language plpgsql
```

Die zugehörige Matrizenmultiplikation AB kann mit dieser UDF dann durch das SQL-Statement

```
select i, array_agg(v)
from (
    select at.i as i,b.i as j, array_mat_mult_element(at.col,b.col) as v
    from at, b
    order by at.i,b.i
) t
group by i
```

dargestellt werden. Dieser Ansatz wird in den Abbbildungen 6.13 und B.1 als „psql2agg“ bezeichnet.

Die Abbildungen B.1 bis B.3 zeigen die Performance-Ergebnisse der SQL-Matrizenmultiplikation der verschiedene Varianten aus Abbildung 6.13, aufgeteilt auf die einzelnen Systeme, um den visuellen datenbankinternen Vergleich zu erleichtern.

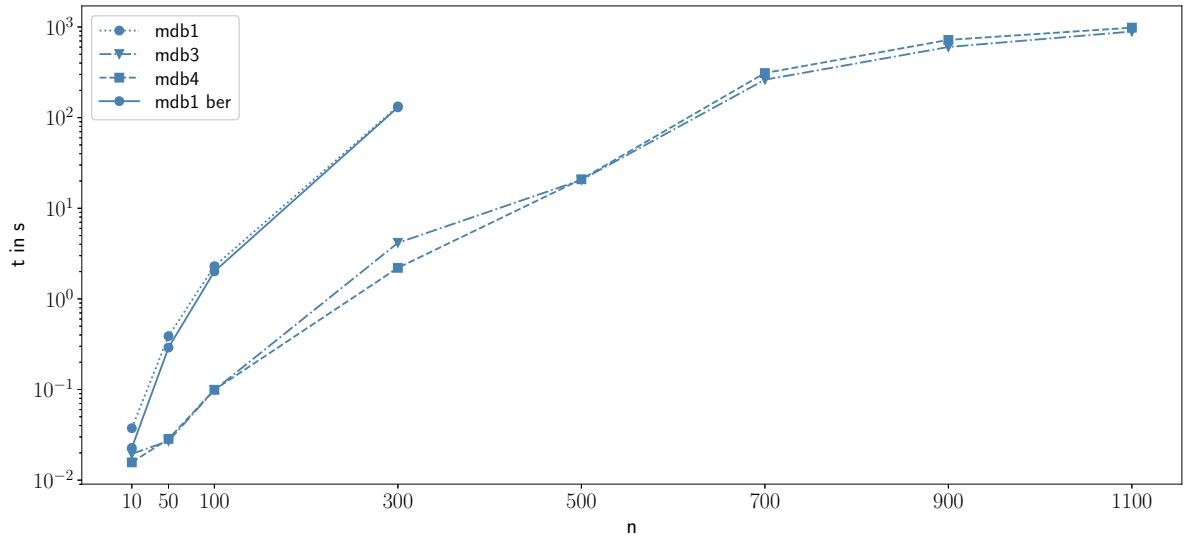


Abbildung B.2: Experimentelle Auswertung von Matrizenmultiplikationen $C = AB$ mit $A, B, C \in \mathbb{R}^{n \times n}$ und den verschiedenen Matrizenschemata (1,3,4) aus Abschnitt 6.4 in MonetDB.

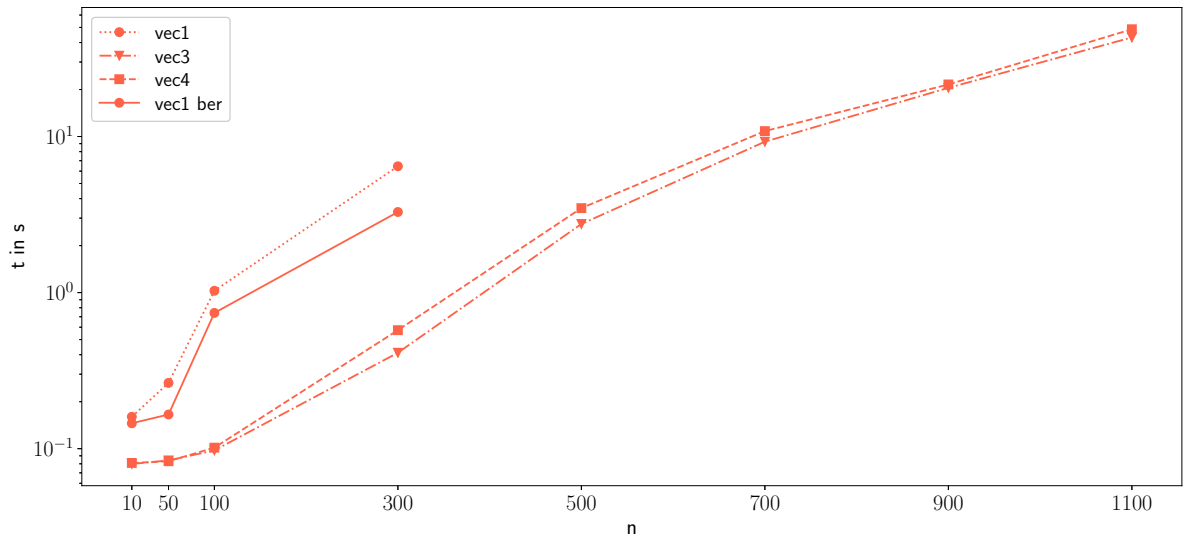


Abbildung B.3: Experimentelle Auswertung von Matrizenmultiplikationen $C = AB$ mit $A, B, C \in \mathbb{R}^{n \times n}$ und den verschiedenen Matrizenschemata (1,3,4) aus Abschnitt 6.4 in Actian's Vector.

Anfragen für Matrix-Vektor-Multiplikation

Im Folgenden werden die genutzten Anfragen für die dünn besetzte Matrix-Vektor-Multiplikation $y = Bw$ bezüglich des *Compressed-Sparse-Row*-Schemas (CSR-Schema) und dem Coordinate-Schemas dargelegt. Letztere Anfrage entspricht

```
insert into y
select b.i, sum(b.v*w.v)
from b join w on b.j=w.i
group by b.i,
```

wobei `w` und `y` das Coordinate-Schema

```
w/y(
   i int,
   v double precision not null,
   primary key (i)
)
```

für Vektoren nutzen. Die Anfrage für das CSR-Schema ist wie folgt:

```
insert into y
select bb.i, sum(bb.v*w.v)
from (
   select i, unnest(j) as jj, unnest(col) as v
   from b
) bb join w on bb.jj=w.i
group by bb.i
```

Die Vektoren **w** und **y** nutzen hier ebenfalls im Vektor-Coordinate-Schema. In der Anfrage ist zu erkennen, dass die temporäre Relation `b` genau die Coordinate-Darstellung der Matrix B enthält und die weitere Verarbeitung identisch mit dem der *Coordinate*-Anfrage ist.

B.3 Transformation in relationale Operatoren

In diesem Abschnitt werden Programmcodes für Vergleichsrechnungen aus Kapitel 7 aufgeführt.

B.3.1 R-Code für Vergleichsrechnungen in Abschnitt 7.2

Im Folgenden ist der genutzte Quellcode zur Berechnung der untersuchten Lineare-Algebra-Operationen für die Experimente aus Abschnitt 7.2 aufgelistet. Prämisse war hierbei, dass die eigentlichen double-Einträge keinen Einfluss auf die Berechnungsergebnisse haben.

Skalarpodukt

Für das Skalarprodukt wurde der folgende Code genutzt:

```
dim <- c(1,3,5,10,15,20)*1000*1000;
repitions <-3

t <- rep(0,3*length(dim));

for (i in 1:length(dim)){
    for (j in 1:repitions){

        a <- runif(dim[i]);
        b <- runif(dim[i]);

        t1 <- Sys.time();
        d <- a%*%b;
        t[repitions*(i-1)+j] <- Sys.time()-t1;
    }
}

write.table(cbind(kronecker(dim,rep(1,repitions)),t),file="erg_vec_r.csv",
    row.names=FALSE,col.names=FALSE,sep=",");
```

Matrix-Vektor-Produkt

Das Matrix-Vektor-Produkt wurde in R wie folgt umgesetzt:

```
dim <- c(1,2,3,4,5,6,7)*1000;
repitions <-3

t <- rep(0,3*length(dim));
tins <- rep(0,3*length(dim));

for (i in 1:length(dim)){
    for (j in 1:repitions){

        a <- matrix(runif(dim[i]*dim[i]) , ncol=dim[i] );
        b <- runif(dim[i]);

        t1 <- Sys.time();
        d <- a%*%b;
        write.table(d,file="d.csv",row.names=FALSE,col.names=FALSE,sep=",");
        tins[repitions*(i-1)+j] <- Sys.time()-t1;

        t2 <- Sys.time();
        d <- a%*%b;
```

```
        t[repitions*(i-1)+j] <- Sys.time()-t2;

    }
}

write.table(cbind(kronecker(dim,rep(1,repitions)),t),file="erg_r.csv",
    row.names=FALSE,col.names=FALSE,sep=",");
write.table(cbind(kronecker(dim,rep(1,repitions)),tins),
    file="erg_vec_r.csv",row.names=FALSE,col.names=FALSE,sep=",");
```

Matrizen-Multiplikation

Die Matrizenmultiplikation wurde analog wie folgt berechnet:

```
dim <- c(50,100,200,400,600,800,1000,1500,2000,3000)
repitions <-3

t <- rep(0,3*length(dim));
tvec <- t;

for (i in 1:length(dim)){
    for (j in 1:repitions){

        a <- matrix(runif(dim[i]*dim[i]) , ncol=dim[i] );
        b <- matrix(runif(dim[i]*dim[i]) , ncol=dim[i] );

        avec<-runif(dim[i]*dim[i]);
        bvec<-runif(dim[i]*dim[i]);

        t1 <- Sys.time();
        d <- a%*%b;
        write.table(d,file="d.csv",row.names=FALSE,col.names=FALSE,sep=",");
        t[repitions*(i-1)+j] <- Sys.time()-t1;

        t1 <- Sys.time();
        dvec <- avec%*%bvec;
        write.table(dvec,file="d.csv",row.names=FALSE,col.names=FALSE,sep=",");
        tvec[repitions*(i-1)+j] <- Sys.time()-t1;
    }
}

write.table(cbind(kronecker(dim,rep(1,repitions)),t),file="erg_r.csv",
    row.names=FALSE,col.names=FALSE,sep=",");
write.table(cbind(kronecker(dim*dim,rep(1,repitions)),tvec),
    file="erg_vec_r.csv",row.names=FALSE,col.names=FALSE,sep=",");
```

Update-basierte SQL-Implementation des Forward-Verfahrens im Abschnitt 7.3

Im Folgenden wird der sequenzielle Anfrageplan des Forward-Verfahrens dargestellt, welcher in Abschnitt 7.3 mit einer geschachtelten Version ausgewertet und verglichen wurde.

```
insert into alpha
select pi.i, pi.v * b.v
from pi join b
   on pi.i = b.i
where b.j = (select v from o where i=0);


for t = 1,...,T    /* Externe Schleife */
   update alpha set v = (
   select alpT_a.v*b.v
   from (
      select a.j as i, sum(aa.v*a.v) as v
      from {alpha| Q_{t-1}} aa join a on aa.i=a.i
      group by a.j
   ) alpT_a join b on alpT_a.i=b.i
   where b.i=alpT_a.i and b.j=(
      select v
      from o
      where i=t
   )
   ) where exists (
      select 1
      from (
         select distinct j as i
         from a
      ) ttt
      where ttt.i=alpha.i
   );
endfor

select sum(v) from {alpha| Q_T};
```

B.4 Parallelisierung durch Anfragezerlegung

Im Folgenden werden die Kostenfunktion verschiedener in Kapitel 8.1.2 ausgesparter Partitionierungsfälle im Detail bestimmt.

B.4.1 Zeilenweise/Spaltenweise Partitionierung dicht besetzter Probleme

Zunächst wird eine Kostenanalyse bezüglich der in Abschnitt 8.1.2 nicht vorgestellten Szenarien der Zeilen- bzw. Spaltenpartitionierung vorgenommen.

Kosten von $C = AB$ bei Kommunikation von A

In dieser Strategie wird auf jedem Knoten k eine Teilsumme für jedes Element von $C = AB$, nachdem die zugehörigen Elemente $A_{\overline{p_k},p_k}$ erhalten wurden, berechnet:

$$C^{(p_k)} = A_{:,p_k} B_{p_k,:}.$$

Anschließend, werden die Elemente $C^{(p_k)}_{\overline{p_k},:}$ versendet und die empfangenen Zwischenergebnisse aggregiert

$$C_{p_k,:} = \sum_{i=1}^{m} C^{(\overline{p_k})}_{p_k,:}.$$

Daraus ergeben sich Kommunikationskosten von

$$\begin{aligned}
{}^{z}_{AB}\kappa_2 &= \underbrace{m}_{\text{pro Knoten}} \Big(\overbrace{(n-l)l}^{\text{Senden von } A_{p_k,\overline{p_k}}} + \overbrace{(n-l)n}^{\text{Senden von } C^{(p_k)}_{\overline{p_k},:}} \Big) \\
&= n(n-l)(m+1) \\
&= nl(m-1)(m+1) \\
&= n^2\left(m - \frac{1}{m}\right)
\end{aligned}$$

Tupel. Für die lokalen Berechnungskosten lassen sich

$${}^{z}_{AB}\zeta_2 = \underbrace{\overbrace{n^2}^{\text{Elemente}} \Big(\overbrace{l}^{\text{Multiplikationen}} + \overbrace{l-1}^{\text{Additionen}} \Big)}_{A_{:,p_k} B_{p_k,:}} + \underbrace{\overbrace{nl}^{\text{Elemente}} \overbrace{m-1}^{\text{Additionen}}}_{\sum_{i=1}^{m} C^{(\overline{p_k})}_{p_k,:}}$$

$$= nl(2n-1) = \frac{n^2(2n-1)}{m}$$

FLOPs folgern. Durch die zwei Kommunikationsphasen entsteht hierbei zusätzlich ein erhöhter

Synchronisationsaufwand, welcher in dieser Analyse nicht dargestellt wird. Ein Vergleich der Kostenabschätzungen zeigt, dass der relative Unterschied der Kommunikationskosten wie folgt ist:

$$\frac{{}^{z}_{AB}\kappa - {}^{z}_{AB}\kappa_2}{{}^{z}_{AB}\kappa} = -\frac{1}{m}$$

Demnach ist die in Abschnitt 8.1.2 vorgestellte Strategie stets günstiger im Sinne der Kommunikation. Ferner ist der relative Unterschied umgekehrt proportional zur Knotenanzahl. Dies bedeutet, dass für kleinere Clustergrößen ein deutlicher Unterschied in den Kommunikationskosten entstehen kann. Ein Verbund von 16 Knoten führt beispielsweise zu Mehrkosten von 6.25% für Strategie 2. Auf der anderen Seite sind die lokalen Prozesskosten der beiden Strategien identisch. Die ursprüngliche Strategie ist aus diesem Grund zu bevorzugen.

Kosten von $C = A^T B$

Im Fall von $C = A^T B$ kann auf dem Knoten k bereits die Teilsumme

$$C^{(p_k)} = (A_{p_k,:})^T B_{p_k,:} = (A^T)_{:,p_k} B_{p_k,:}$$

berechnet werden, wobei ein Element $c_{ij}^{(p_k)}$ von $C^{(p_k)}$ die Form

$$c_{ij}^{(p_k)} = \sum_{k \in p_k} a_{ki} b_{kj}$$

besitzt. Das finale Ergebnis wird nach Kommunikation der Teilsummen durch

$$C = \sum_{k=1}^{m} C^{(p_k)}$$

bestimmt, wobei jeder Knoten k die Teilsummen $C_{\overline{p}_k,:}$ versendet und die erhaltenen Teilsummen aggregieren muss.

Daraus ergeben sich hierfür Kommunikationskosten von

$$\begin{aligned} {}^{z}_{A^T B}\kappa &= \underbrace{m}_{\text{pro Knoten}} \overbrace{(m-1)}^{\text{Teilergebnisse}} \underbrace{nl}_{C^{(p_{\tilde{k}})}} \\ &= m(m-1)nl \\ &= n^2(m-1) = {}^{z}_{AB}\kappa \end{aligned}$$

Elementen und lokale Prozesskosten von

$$ {}_{A^T B}^{z}\zeta = nl(2n-1) = {}_{AB}^{z}\zeta $$

FLOPS.

Kosten von $C = AB^T$

Im Fall AB^T liegen die Daten bereits im nötigen Zeilen-Spalten-Schema vor, sodass die Submatrizen C_{p_k,p_k} bereits komplett lokal berechnet werden können. Für die Elemente $C_{p_k,\overline{p}_k}$ sind jedoch die nötigen Spalten von B erforderlich.

$$ C_{p_k,p_k} = A_{p_k,:}\left(B^T\right)_{:,p_k} = \underbrace{A_{p_k,:}\left(B_{p_k,:}\right)^T}_{\text{lokal vorhanden}} $$

$$ C_{p_k,\overline{p_k}} = A_{p_k,:}\left(B^T\right)_{:,\overline{p_k}} = A_{p_k,:} \quad \underbrace{\left(B_{\overline{p_k},:}\right)^T}_{\text{nicht vorhanden}} $$

Daraus ergeben sich die Kostenfunktionen

$$ \begin{aligned} {}_{AB^T}^{z}\kappa &= \underbrace{m}_{\text{pro Knoten}} \overbrace{n}^{\text{Zeilen } B} \underbrace{(n-l)}_{\text{Spalten} B} \\ &= n^2(m-1) \\ &= {}_{A^T B}^{z}\kappa = {}_{AB}^{z}\kappa \end{aligned} $$

und

$$ \begin{aligned} {}_{AB^T}^{z}\zeta &= nl(2n-1) \\ &= {}_{A^T B}^{z}\zeta = {}_{AB}^{z}\zeta. \end{aligned} $$

Kosten $C = A^T B^T$

Der Fall $C = A^T B^T$ ist im Vergleich zu den vorigen speziell. Die Berechnung des Produktes $A^T B^T$ kann analog zu den vorigen Fällen mit der Kommunikation von $n^2(m-1)$ Tupeln berechnet werden. Dies ist ersichtlich, wenn bedenkt wird, dass $A^T B^T = (BA)^T$ ist, sodass BA analog zu der Strategie für AB berechnet werden kann. Problematisch hierbei ist, dass in diesem Fall Spalten von C lokal berechnet werden, jedoch Zeilen gespeichert werden sollten. Demnach wird eine anschließende Repartitionierung benötigt.

Analog zu den vorigen Strategien, in denen die Elemente des Produktes komplett lokal berechnet werden, ergeben sich die Kostenfunktionen

$$
\begin{aligned}
{}_{A^T}{}_{B^T}{}^{z}\kappa &= n^2(m-1) \\
{}_{A^T}{}_{B^T}{}^{z}\zeta &= nl(2n-1),
\end{aligned}
$$

wobei jedoch

$$
\begin{aligned}
{}_{C=A^T}{}_{B^T}{}^{z}\kappa &= \underbrace{n^2(m-1)}_{\text{Berechnung}BA} + \overbrace{m}^{\text{pro Knoten}} \underbrace{(n-l)l}_{\text{Repartitionierung } C_{\overline{p_k},p_k}} \\
&= (m-1)(n^2+nl)
\end{aligned}
$$

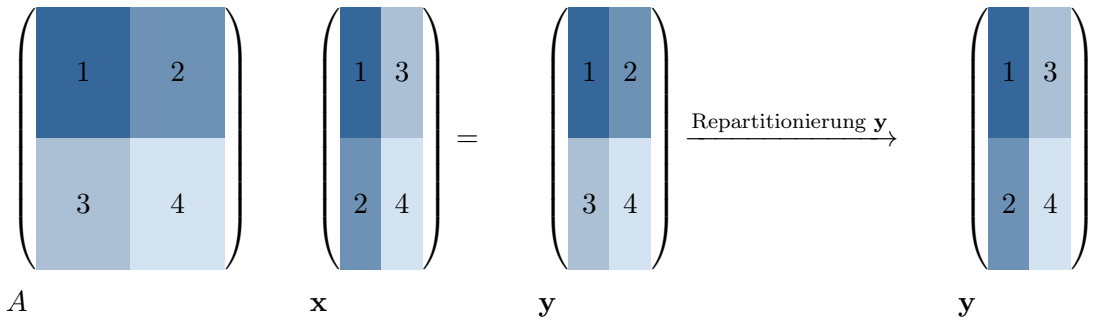
aufgrund der nötigen Repartitionierung ist.

B.4.2 Blockweise Partitionierung

Im Folgenden werden die Kommunikations- und Berechnungskosten bezüglich der Blockpartitionierung dicht besetzter Matrizen vorgestellt.

Matrix-Vektor-Multiplikation $y = A\mathbf{x}$

Im Fall einer Blockpartitionierung von A wird eine nicht triviale Replikation nötig, die beispielsweise in Systemen wie Postgres-XL nicht nativ möglich ist (Stand 12/2022). In diesem Fall ist es sinnvoll, eine teilweise Replikation einiger Elemente auf verschiedenen Knoten umzusetzen. Beispielsweise kann dies für ein 2×2-Gitter wie folgt visualisiert werden:



Hierbei zeigen die zweispaltigen Vektoren die Knotennummern an, auf denen die jeweiligen Replikate und Zwischenergebnisse hinterlegt sind. Offenbar kann dies (ineffizienter) auch durch komplette Replikation der Vektoren ermöglicht werden.

Die Vektoren $\mathbf{y}$ und $\mathbf{x}$ werden mit 1-attributiger Bereichspartitionierung in $\mathfrak{m}$ Teile $\mathbf{x}_i, \mathbf{y}_i$ verteilt. Jeder Bereich ist $\mathfrak{m}$-mal repliziert um die lokale Berechnungsmöglichkeiten zu maximieren. Dies ist aus der Vorschrift

$$\mathbf{y}_k = \sum_{i=1}^{\mathfrak{m}} \underbrace{A_{k,i}\mathbf{x}_i}_{\mathbf{y}_k^{(i)}}$$

ersichtlich. Die Bereiche $\mathbf{x}_i$ werden im Produkt $A\mathbf{x}_i$ mit den Submatrizen der Form $A_{k,i}$, welche auf den Knoten $\xi(k,i)$ liegen, kombiniert. Daher müssen in diesem Fall die i-ten Bereichs-Vektoren auf den Knoten $\xi(k,i)$ $(k = 1, \ldots, \mathfrak{m})$ repliziert hinterlegt werden. Da die Zwischenergebnisse $\mathbf{y}_k^{(i)}$ auf Knoten des i-ten Bereichs für die Berechnung des $k-$ten Bereichs berechnet werden, ist eine finale Aggregation auf dem Diagonalknoten $\xi(i,i)$ sinnvoll, da dort bereits das Ergebnis $\mathbf{y}_i^{(i)}$ liegt. Eine verteilte Berechnung mit anschließender Repartitionierung ist prinzipiell ebenfalls möglich. Im Sinne der konsistenten Speicherung für weitere Operationen muss dann nach der Aggregation jeder Diagonalknoten $\xi(i,i)$ die Ergebnisse $\tilde{\mathbf{y}}_\mathbf{i}$ auf die $\mathfrak{m}-1$ Knoten der Form $\xi(k,i)$ senden.
Zusammengefasst ergibt sich für den Knoten $\xi(i,j)$ die Strategie:

1. Berechne $\mathbf{y}_i^{(j)}$
2. Sende an Diagonalknoten $\xi(i,i)$
3. Finale Aggregation $\mathbf{y}_i = \mathbf{y}_i^{(j)}$ auf Knoten $\xi(i,i)$
4. Sende von $\xi(i,i)$ zur Replikation $\mathbf{y}_i$ an alle Knoten $\xi(k,i)$ $k = 1, \ldots, \mathfrak{m}$

Es ergeben sich daraus Kommunikationskosten von

$$\begin{aligned} {}_{A^T\mathbf{x}}{}^{b}\kappa &= \underbrace{(m-\mathfrak{m})}_{\text{Nicht-Diag}} \overbrace{\mathfrak{l}}^{\text{Sende } y_i^{(j)}} + \underbrace{(m-\mathfrak{m})}_{\text{Nicht-Diag}} \overbrace{\mathfrak{l}}^{\text{Repartitionierung } y_i} \\ &= 2n(\sqrt{m}-1) \end{aligned}$$

Tupeln und maximale lokale Prozesskosten (Diagonalknoten) von

$$\begin{aligned} {}_{A^T\mathbf{x}}{}^{b}\zeta &= \underbrace{\mathfrak{l}^2(2\mathfrak{l}-1)}_{y_i^{(i)}} + \underbrace{(\mathfrak{m}-1)\mathfrak{l}}_{\mathbf{y}_i=\sum \mathbf{y}_i^{(k)}} \\ &= \frac{n^2}{m}\left(\frac{2n}{\sqrt{m}}-1\right) + n\left(1-\frac{1}{\sqrt{m}}\right) \end{aligned}$$

FLOPs.

Kosten von $C = A^T B$

In diesem Kontext kann gezeigt werden, dass konträr zum Fall $C = AB$ es vorteilhafter ist, die Submatrizen A_{ij} in Phase 1 zu versenden. Da

$$A^T = \begin{bmatrix} A_{1,1}^T & \dots & A_{\mathfrak{m},1}^T \\ \vdots & \ddots & \vdots \\ A_{1,\mathfrak{m}}^T & \dots & A_{\mathfrak{m},\mathfrak{m}}^T \end{bmatrix}$$

ist, gilt

$$C_{i,j} = \sum_{h=1}^{\mathfrak{m}} A_{h,i}^T B_{h,j}.$$

Auf dem Knoten $\xi(i,j)$ können dann lokal die Submatrizen

$$C_{h,j}^{(i)} = A_{h,i}^T B_{i,j} \quad h = 1, \dots, \mathfrak{m}$$

in Phase 1 berechnet werden. Hierbei gilt offenbar, dass für $h = j$ das Produkt $C_{j,j}^{(i)} = A_{i,j}^T B_{i,j}$ auf dem Knoten $\xi(i,j)$ vor der ersten Sendephase bereits lokal vorhanden ist. Analog ist auf dem Knoten $\xi(i,j)$ nach der ersten Phase bereits das Teilergebnis $C_{i,j}^{(i)}$ auf dem korrekten Knoten. Aus diesem Gründen müssen

$$\begin{aligned} {}^{B}_{C=A^T B}\kappa &= \underbrace{\underbrace{\mathfrak{m}^2}_{\text{Knoten}} \overbrace{(\mathfrak{m}-1)}^{\text{Submat.}} \underbrace{\mathfrak{l}^2}_{\text{El.}}}_{\text{Phase 1}} + \underbrace{\overbrace{\mathfrak{m}^2}^{\text{Knoten}} \underbrace{(\mathfrak{m}-1)}_{\text{Submat.}} \overbrace{\mathfrak{l}^2}^{\text{El.}}}_{\text{Phase 2}} \\ &= 2n^2(\mathfrak{m}-1) \\ &= n^2\left(2\sqrt{m}-2\right) \end{aligned}$$

Tupel kommuniziert werden. Analog zu $C = AB$ werden lokal auf jedem Knoten $\mathfrak{m}$ Matrizenprodukte und $\mathfrak{m} - 1$ Matrizenadditionen durchgeführt. Damit ist

$${}^{B}_{C=A^T B}\zeta = {}^{B}_{C=AB}\zeta.$$

Kosten von $C = AB^T$

Analog zu den Betrachtungen des Falles $C = A^T B$ lassen sich die lokalen Teilmatrizen $C_{i,j}^{(h)}$ durch

$$C_{i,j} = \sum_{h=1}^{\mathfrak{m}} A_{i,h} B_{j,h}^T$$

berechnen. In diesem Fall können auf dem Knoten $\xi(i,j)$ die Teilergebnisse

$$C_{i,h}^{(j)} = A_{i,j}B_{h,j}^T \quad h = 1,\ldots,\mathfrak{m}$$

berechnet werden. Das Produkt $C_{i,i}^{(j)} = A_{i,j}B_{i,j}^T$ kann hier auf dem Knoten $\xi(i,j)$ in der ersten Phase komplett lokal berechnet werden. Für die zweite Phase ist hierbei das Teilergebnis $C_{i,j}^{(j)}$ bereits vorhanden. Daraus folgt

$$\begin{aligned}
{}^{B}_{C=A^TB}\kappa &= \underbrace{\underbrace{\mathfrak{m}^2}_{\text{Knoten}}\overbrace{(\mathfrak{m}-1)}^{\text{Submat.}}\underbrace{\mathfrak{l}^2}_{\text{El.}}}_{\text{Phase 1}} + \underbrace{\overbrace{\mathfrak{m}^2}^{\text{Knoten}}\underbrace{(\mathfrak{m}-1)}_{\text{Submat.}}\overbrace{\mathfrak{l}^2}^{\text{El.}}}_{\text{Phase 2}} \\
&= 2n^2(\mathfrak{m}-1) \\
&= n^2\left(2\sqrt{m}-2\right)
\end{aligned}$$

und

$${}^{B}_{C=AB^T}\zeta = {}^{B}_{C=AB}\zeta = {}^{B}_{C=A^TB}\zeta.$$

Kosten von $C = A^TB^T$

Im Fall $C = A^TB^T$ gilt

$$C_{i,j} = \sum_{h=1}^{\mathfrak{m}} A_{h,i}^T B_{j,h}^T.$$

Hier können auf dem Knoten $\xi(i,j)$ die Teilergebnisse

$$C_{j,h}^{(i)} = A_{i,j}^T B_{h,i}^T \quad h = 1,\ldots,\mathfrak{m}$$

berechnet werden. In diesem Fall sind für Phase 1 und 2 nur auf den Diagonalknoten $\xi(i,i)$ bereits Submatrizen lokal vorhanden. Daher sind die Kommunikationskosten höher.

$$\begin{aligned}
{}^{B}_{C=A^TB}\kappa &= \underbrace{\underbrace{(\underbrace{(\mathfrak{m}^2-\mathfrak{m})}_{\text{Nicht-Diag}}\overbrace{\mathfrak{m}}^{\text{Submat.}} + \underbrace{\mathfrak{m}}_{\text{Diag}}\overbrace{(\mathfrak{m}-1))}^{\text{Submat.}}\underbrace{\mathfrak{l}^2}_{\text{El.}}}_{\text{Phase 1}} + \underbrace{\overbrace{(\underbrace{(\mathfrak{m}^2-\mathfrak{m})}_{\text{Submat.}}\underbrace{\mathfrak{m}}_{\text{Submat.}} + \overbrace{\mathfrak{m}}^{\text{Diag}}\underbrace{(\mathfrak{m}-1))}_{\text{Submat.}}}^{\text{Nicht-Diag}}\overbrace{\mathfrak{l}^2}^{\text{El.}}}_{\text{Phase 2}} \\
&= 2\mathfrak{l}^2(\mathfrak{m}^2(\mathfrak{m}-\frac{1}{\mathfrak{m}})) \\
&= n^2(2\sqrt{m}-\frac{2}{\sqrt{m}})
\end{aligned}$$

Das Versenden von Submatrizen von A führt zu dem gleichen Ergebnis. Die Berechnungskosten sind aufgrund der Struktur gleich zu den anderen Fällen.

B.4.3 Bandbreitenreduktion symmetrischer dünn besetzter Matrizen: Das Cuthill-McKee-Verfahren

Im Folgenden wird eine Überführung des Cuthill-McKee-Verfahren, einer heuristischer Methode zur Bandbreitenreduktion symmetrischer dünn besetzter Verfahren, in SQL vorgestellt. Der Algorithmus wurde in [138] detailliert präsentiert und in Algorithmus 19 in Pseudocode dargestellt. Der dort genutzte Nachbarschaftsoperator $N(I)$ beschreibt hierbei alle nicht-0-Einträge der zugehörigen Zeilen bzw. Spalten einer symmetrischen dünn besetzten irreduziblen Matrix $A \in \mathbb{R}^{n \times n}$:

$$N(I) = \{j \in \{1, \ldots, n\} \mid a_{ij}! = 0 \wedge i \in I\}.$$

Ziel ist es eine Reduktion der Bandbreite der Matrix zu erreichen, um Fill-Ins (Einfügen von nicht-0-Werten) bei der Weiterverarbeitung zu vermeiden. Wie in Abschnitt 8.1.3 beschrieben, kann sich eine niedrige Bandbreite zudem positiv auf die parallele Verarbeitung von dünn besetzter Lineare-Algebra-Operationen auswirken.

Algorithmus 19 Bandbreitenreduktion mittels Cuthill-McKee Verfahrens. Der Operator N bezeichnet hierbei die Nachbarn eines Input-Knotens.

1: **Input**: Matrix $A \in \mathbb{R}^{m \times n}$
2: $A = AA^T$
3: Setzte $I = (i_0)$ mit Knoten minimalen Grades $i_0 = \min_{i \in 1,\ldots,n} |N(i)|$
4: **for** `j=1,...,n` **do**
5: Setzte $N_j = N(I[j]) \backslash I$
6: Sortiere Knoten $i_1, \ldots, i_k$ aus N_j nach aufsteigendem Grad $|N(i_{o_1})| \leq |N(i_{o_2})| \leq \ldots \leq |N(i_{o_k})|$
7: Füge sortierte Knoten an I an: $I = (I, i_{o_1}, \ldots, i_{o_k})$
8: **if** $|I| == n$ **then**
9: **break**
10: **end if**
11: **end for**
12: $A = A[I, I]$

Das Verfahren bietet sich für die Überführung in SQL, aufgrund seines Ursprungs als Graphenproblem, an. Im Folgenden wird eine Überführung mit externer Schleife vorgestellt. Das Verfahren ist aufgrund seiner Struktur jedoch voraussichtlich effizienter mittels rekursiver Anfragen umsetzbar. Für die Überführung in SQL wird das Coordinate-Relationenschema für die Matrix A genutzt, sowie das Schema `aneu (i serial, v int)` mit auto-increment `i`. Initial kann durch die folgende Anfrage ein Knoten (Zeile bzw. Spalte) mit minimalen Grad bestimmt werden.

```
insert into aneu
select min(i)
from ( select i, count(j) as v from a group by i ) temp
where temp.v = ( select min(v) from (
      select i, count(j) as v from a group by i ) temp1
);
```

Daraufhin wird die folgende Iterationen solange wiederholt, bis alle Knoten in die Liste aufgenommen worden sind.

```
for i = 1,...,n do

   insert into  aneu (v)
   select i
   from (
         select j as i, count(i) as v
         from a
         where j in ( select j from a where i in (
               select v from aneu where i=$i )
         except select v as j from aneu
         )
         group by j
         order by v asc
   ) temp

end for
```

B.5 Evaluation

Im Folgenden wird zunächst ein Anfageplan von Postgres-XL vorgestellt. Daraufhin werden verschiedene Implementationen dargestellt, die in Experimenten in Kapitel 9 genutzt wurden.

Anfrageplan Matrizenmultiplikation Postgres-XL

Es folgt ein Anfrageplan zur Matrizenmultiplikation AB von Postgres-XL. Der Plan ist in seiner Struktur stabil bezüglich der in dieser Arbeit getesteten Matrixdimensionen. Aus dem Plan ist ersichtlich, dass Postgres-XL die Aggregation **sum** distributiv verarbeitet. Der Plan ermöglicht jedoch keine genauen Einblicke über die Zwischenverteilung beziehungsweise der Lokalität der Daten.

```
********* QUERY **********
explain verbose select A.i,B.j,sum(A.v*B.v)
from A join B on A.j=B.i
group by A.i,B.j;
**************************

QUERY PLAN
------------------------------------------------------------------------------------
HashAggregate  (cost=694.84..865.96 rows=17112 width=24)
Output: a.i, b.j, sum((sum((a.v * b.v))))
Group Key: a.i, b.j
->  Remote Subquery Scan on all (dn_1,dn_2,dn_3,dn_4,dn_5,dn_6,dn_7,dn_8,dn_9)
 (cost=257.79..523.72 rows=17112 width=24)
Output: a.i, b.j, sum((a.v * b.v))
    ->  HashAggregate  (cost=257.79..523.72 rows=17112 width=24)
    Output: a.i, b.j, sum((a.v * b.v))
    Group Key: a.i, b.j
       ->  Merge Join  (cost=257.79..523.72 rows=17112 width=24)
       Output: a.i, a.v, b.j, b.v
       Merge Cond: (b.i = a.j)
          ->  Remote Subquery Scan on all (dn_1,dn_2,dn_3,dn_4,dn_5,dn_6,dn_7,dn_8,dn_9)
                  (cost=100.00..167.35 rows=1850 width=16)
          Output: b.j, b.v, b.i
          Distribute results by H: i
              ->  Sort  (cost=267.74..272.37 rows=1850 width=16)
              Output: b.j, b.v, b.i
              Sort Key: b.i
                  ->  Seq Scan on public.b (cost=0.00..28.50 rows=1850 width=16)
                  Output: b.j, b.v, b.i
         ->  Sort  (cost=128.89..133.52 rows=1850 width=16)
          Output: a.i, a.v, a.j
          Sort Key: a.j
              ->  Seq Scan on public.a (cost=0.00..28.50 rows=1850 width=16)
              Output: a.i, a.v, a.j
(24 rows)
```

B.5.1 Blockpartitionierung und Anfragezerlegung in Postgres-XL

Die Umsetzung der Auswertungen der Blockpartitionierung mittels Vererbung und darauf basierender Anfragezerlegungsansätze in Postgres-XL aus Abschnitt 9.1 wurde durch das folgende Shell-Skript umgesetzt.

```
dim=( 240 480 720 960 1200 1440 )
```

```
grid=3
nodes=$(( grid * grid ))
A="mA"
B="mB"
C="mC"
cur_dir=‘pwd‘
db_call1="psql -U USER -d DATENBANKNAME -c"

for n in ${dim[*]}
do

    echo "---------------- starting with dimension $n"
    echo "------- starting data creation "

    psql -c "truncate $A , $B, $C"

    sql=‘./master_child_update.sh $n $A‘
    psql -c "$sql"
    sql=‘./master_child_update.sh $n $B‘
    psql -c "$sql"

    rm *.csv
    ######## data creation
    ./create_data_block $n data 800

    echo "------- starting data insertion"
    ######## data insertion
    for file in $(ls -a | grep .csv); do
        psql -c "copy $C from ’${cur_dir}/$file’ CSV;" -q
    done
    psql -c "insert into $B select * from $C;
insert into $A select * from $B;"

    echo "------- starting calculation"
    ## Create Grid Queries

    sql=""
    for j1 in ‘seq 1 $grid‘
    do
        for j2 in ‘seq 1 $grid‘
        do
            cur_node=$(( ( $j1 -1 )*$grid + $j2 ))

            sqlmc[$cur_node]="select i,j,sum(v) from ( "
            for k in ‘seq 1 $grid‘
            do
```

```
            sqlmc[$cur_node]="${sqlmc[$cur_node]}
select a1.i as i, a2.j as j, sum(a1.v*a2.v) as v
from ${A}child$(( ( $j1 - 1 )*$grid + $k )) a1
join ${B}child$(( ( $k -1 )*$grid + $j2 )) a2 on a1.j=a2.i
group by a1.i,a2.j"

            if [ $k -lt $grid ]
            then
            sqlmc[$cur_node]="${sqlmc[$cur_node]}
union all"
            fi
            done
            sqlmc[$cur_node]="${sqlmc[$cur_node]} ) temp
group by i,j"
        done
    done

    echo "---- start normal approach "

    psql -t -c "select min(i),max(i),min(j),max(j) from $A;"
    psql -t -c "select min(i),max(i),min(j),max(j) from ${A}child1;"

    ts=$(($(date +%s%N)/1000000))
    $db_call1 "select $A.i,$B.j,sum($A.v*$B.v)
from $A join $B on $A.j=$B.i
group by $A.i,$B.j;" >/dev/null 2>&1
    te=$(($(date +%s%N)/1000000))
    echo "$n, $(( $te - $ts )) , normal" >> erg_mc.txt

    echo "--- start vacuuming --- "
    psql -c "vacuum analyze ;"
    echo "---- start master child matrix ---- "

    echo "${sqlmc[$1]}"

    ts=$(($(date +%s%N)/1000000))
    for k in `seq 1 $nodes`
    do
        ssh BerechnungsknotenNr$(( ( $k - 1 ) / 2 + 1 ))
"psql -U dd$k -d postgres -c '${sqlmc[${k}]};' >/dev/null 2>&1 ; echo Node $k done" &
    done
    wait
    te=$(($(date +%s%N)/1000000))
    echo "$n, $(( $te - $ts )), Multi Query 9" >> erg_mc.txt
done
```

```
echo "${sqlmc[1]}"
```

Die Matrizen für den Vererbungsansatz wurden mittels folgendem Shell-Skript erstellt.

```
n="$1"
grid=3
nodes=$(( $grid * $grid ))
A="$2"
suf="child"

sql="CREATE TABLE if not exists ${A} (i int not null , j int not null , v double
     precision not null );"

for i in `seq 1 $grid`
do
    for j in `seq 1 $grid`
    do
        nr=$(( ( $i - 1 )*$grid + $j ))
        L=$(( n / grid ))
        row_start=$(( ( i - 1 ) * L  + 1 ))
        row_end=$(( i * L ))
        col_start=$(( ( j - 1 ) * L  + 1 ))
        col_end=$(( j * L ))

        sql="$sql
        create table $A$suf$nr (check ( i >= $row_start and i <= $row_end and j
            >= $col_start and j <= $col_end )) inherits($A) to node(dn_$nr);
        create or replace rule rule$A$nr as on insert to $A where i>=$row_start
            and i <=$row_end and j>=$col_start and j<=$col_end do instead insert
             into $A$suf$nr values (NEW.*);"
    done
done

echo "$sql"
```

Das Skript `master_child_update.sh` updatet die Integritätsbedingungen der Matrizen bezüglich der variierenden Dimensionen. Das Skript ist dabei wie folgt implementiert:

```
n="$1"
grid=3
nodes=$(( $grid * $grid ))
A="$2"
suf="child"

for i in `seq 1 $grid`
do
   for j in `seq 1 $grid`
```

```
    do
        nr=$(( ( $i - 1 )*$grid + $j ))
        L=$(( n / grid ))
        row_start=$(( ( i - 1 ) * L + 1 ))
        row_end=$(( i * L ))
        col_start=$(( ( j - 1 ) * L + 1 ))
        col_end=$(( j * L ))

        sql="$sql
create or replace rule rule$A$nr as on insert to $A
where i>=$row_start and i <=$row_end and j>=$col_start and
j<=$col_end do instead insert into $A$suf$nr values (NEW.*);"
    done
done

echo "$sql"
```

Die *executable* `create_data_block` aus dem Basis-Skript erstellt Daten derart, dass sie beim Einfügen per RoundRobin auf 9 Knoten exakt die Blockdarstellung erreicht. Diese Form ist durch die erstellten Einfügeregeln für den Vererbungsansatz prinzipiell nicht nötig, wird aber für den in Abschnitt 9.1 beschriebenen *distribute by*-Ansatz genutzt. Das zugehörige kompilierte C++-Programm ist im Folgenden dargestellt.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <string>
#include <sys/time.h>
#include <sstream>

using namespace std;


template<typename T>
string to_string (const T& t)
{
    std::stringstream ss;
    ss << t;
    return ss.str();
}

int main(int argc, char *argv[])
{
    if (argc < 4){
```

```
    cout  << "Not enough input arguments" << endl
    << "Input needed:" << endl
    << "1. matrix dimension" << endl
    << "2. file_name" << endl
    << "3. elements per file " << endl;
    return(-1);
    }

    double lv = -1.0, mv = 1.0;
    long n = atol(argv[1]),
    k = 3,        // grid_size_one_dim m = k x k
    l= n / k,       // entries per dim per node
    el_p_f= atol(argv[3]),   // elements per file parameter, 800 leads to 1.1GB files,
    el_file= k * k * el_p_f,  // elements per file
    nofiles= n/el_file;

    if (n%k != 0) {
        cout << "Dimension is not compatible with grid size (3x3).
Programm aborted." << endl;
        return (-2);
    }

    if (n%el_file != 0) nofiles++;


    srand((unsigned)time(0));
    for (unsigned long nof = 0; nof < nofiles; nof++){
        cout<<"\r --- File "<<nof+1<<" from "<<nofiles<<" ---"<<flush;

        ofstream fileDB((string(argv[2])+"_"+to_string<int>(nof)+".csv").c_str());
        long start_row = nof*k*el_p_f;


        for (unsigned long i=start_row;i<min((start_row+k*el_p_f),l); i++){
            for (unsigned long j = 0; j < l; j++){
                for (int ig = 0; ig < k; ig++){
                    for (int jg = 0; jg < k; jg++) {
                        double val=lv+(mv-lv)*
                            ((double) rand()/RAND_MAX);
                        fileDB << 1 + i + ig*l
                        << ", " << 1 + j + jg*l
                        << ", " << val << endl;
                    }
                }
            }
        }
```

```
        fileDB.close();
    }
    cout << endl;
    return(0);
}
```

B.5.2 Erstellung von Testmatrizen und - vektoren

Der folgende C++-Code wurde genutzt zur Erstellung von Testdaten für die Experimente aus Abschnitt 9.1.

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <sstream>
#include <algorithm>
#include <random>

using namespace std;

template<typename T>
string to_string(const T& t)
{
std::stringstream ss;
ss << t;
return ss.str();
}

int create_dense_matrix(int argc, char *argv[])
{

    if (argc < 3) {
        cout << "Not enough input arguments" << endl
        << "Input needed:" << endl
        << "1. matrix dimension" << endl
        << "2. file_name" << endl;
        return(-1);
    }

    double lv = -1.0, mv = 1.0;
    long n = atol(argv[1]);

    srand((unsigned)time(0));
    ofstream fileDB((string(argv[2]) + ".csv").c_str());
```

```
    for (int ig = 0; ig < n; ig++) {
        for (int jg = 0; jg < n; jg++) {
            double val = lv + (mv - lv) * ((double)rand()
            / RAND_MAX);
            fileDB << 1 + ig << ", "
            << 1 + jg << ", "
            << val << endl;
        }
    }
    fileDB.close();

    return(0);
}

int create_random_sparse_bandmatrix(int argc, char *argv[])
{

    if (argc < 5) {
        cout << "Not enough input arguments" << endl
        << "Input needed:" << endl
        << "1. matrix dimension" << endl
        << "2. file_name" << endl
        << "3. rows per file" << endl
        << "4. elements per row" << endl;
        return(-1);
    }

    // -------------------------
    // ---- Variables
    // -------------------------

    double lower_value_bound = -1.0,
    upper_value_bound = 1.0;

    string file_prefix = string(argv[2]) + "_";

    long n = atol(argv[1]), // matrix dimension
    rows_per_file = min(n, atol(argv[3])), // elements per file
    l = (3 * rows_per_file) / 4, // bandwwidth
    elements_per_row = atol(argv[4]), // elements per row
    nofiles = n / rows_per_file; // number of files

    if ((2 * l) <= elements_per_row) {
        cout << "bandwidth too low";
        return -2;
    }
```

```
if (n % elements_per_row != 0) nofiles++;
srand((unsigned)time(0)); // random seed

// -----------------------
// ---- Calculation / Writing
// -----------------------

for (long number_of_file=0;number_of_file<nofiles;number_of_file++) {
    ofstream fileDB((file_prefix+to_string<int>(number_of_file)+".csv").c_str());

    long start_row = number_of_file * rows_per_file;

    for (long i=1+start_row;i<= min(start_row+rows_per_file,n);i++){
        vector<int> columns;
        columns.push_back(i);

        fileDB << i << ", " << i << ", "
        << lower_value_bound + (upper_value_bound -
        lower_value_bound) * ((double)rand() / RAND_MAX)
        << endl;

        for (long j = 0; j < elements_per_row; j++) {
            long new_column;

            while (true) {
                new_column = static_cast<long>(i + pow(-1,
                rand() % 2) * (rand() % 1));
                new_column = min(n, max((long)1, new_column));

                if (find(columns.begin(), columns.end(),
                    new_column) == columns.end()) {
                    columns.push_back(new_column);
                    break;
                }
            }

            fileDB << i << ", " << new_column << ", "
            << lower_value_bound + (upper_value_bound -
            lower_value_bound) * ((double)rand() / RAND_MAX)
            << endl;
        }
    }
    fileDB.close();
}
```

```
    cout << endl;
    return(0);
}
```

B.5.3 Implementation in Apache Spark

Im Folgenden werden die Implementation für Apache Spark aufgeführt, die in Abschnitt 9.2 genutzt wurden. Alle Implementationen stammen aus der Arbeit [142], welche im Rahmen des präsentierten Forschungsprojektes entstanden sind.

Dicht besetzte Blockmatrixmultiplikation

Es folgt zunächst die Berechnung dicht besetzter Matrixmultiplikationen mittels blockpartitionierter Matrizen in mllib/linalg.

```
import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix, MatrixEntry}
import org.apache.spark.sql.SparkSession

object MatrixMult {
    val mst = $MASTER_URL
    val name = "MatrixMultiplication"
    val sparkSession = SparkSession.builder().appName(name).master(mst).
        getOrCreate()

    def main(args: Array[String]): Unit = {
        val path1 = "/m1.csv"
        val path2 = "/m2.csv"

        val cooMatP1 = csvToCooMatrix(path1)
        val cooMatP2 = csvToCooMatrix(path2)

        val bMatP1 = cooMatP1.toBlockMatrix(rowsPerBlock,colsPerBlock)
        val bMatP2 = cooMatP2.toBlockMatrix(rowsPerBlock,colsPerBlock)

        val newMat = bMatP1.multiply(bMatP2)
        newMat.validate()
    }

    def csvToCooMatrix(path: String): CoordinateMatrix = {
        val df = sparkSession.read.format("csv").option("header", "true").load(
            path)
        val tempDF = df.withColumn("i", df("i").cast("long")).withColumn("j", df
            ("j").cast("long")).withColumn("v", df("v").cast("double"))
        val corMat = new CoordinateMatrix(tempDF.rdd.map(m => MatrixEntry(m.
            getLong(0), m.getLong(1), m.getDouble(2))))
```

```
        corMat
    }
}
```

Dünn besetzte Matrix-Vektor-Multiplikation mittels map und reduce

Der folgende Code wurde genutzt, um eine dünn besetzte Matrix-Vektor-Multiplikation $A\mathbf{x}$ zu berechnen. Die Daten liegen hier im HDFS im HOME-Ordner als CSV-Dateien im Coordinate-Schema vor. Diese Implementation arbeitet im wesentlichen mit klassischen *map*- und *reduce(ByKey)*-Operationen.

```
import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix,MatrixEntry}
import org.apache.spark.sql
import org.apache.spark.sql.SparkSession

object MatrixMult {
    val mst = $MASTER_URL
    val name = "MatrixMultiplication"
    val sparkSession = SparkSession.builder().appName(name).master(mst).
        getOrCreate()

    def main(args: Array[String]): Unit = {
        val path1 = "/m1.csv"
        val path2 = "/m2.csv"
        val cooMatP1 = csvToCooMatrix(path1)
        val cooMatP2 = csvToCooMatrix(path2)
        val newMat = coordinateMatrixMultiply(cooMatP1, cooMatP2)
        newMat.entries.count()
    }

    def coordinateMatrixMultiply(leftMatrix: CoordinateMatrix, rightMatrix:
        CoordinateMatrix): CoordinateMatrix = {
        val M_ = leftMatrix.entries.map({ case MatrixEntry(i, j, v) => (j, (i, v
            )) })
        val N_ = rightMatrix.entries.map({ case MatrixEntry(j, k, w) => (j, (k,
            w)) })
        val productEntries = M_.join(N_).map({ case (_, ((i, v), (k, w))) => ((i
            , k), (v * w)) }).reduceByKey(_ + _).map({ case ((i, k), sum) =>
            MatrixEntry(i, k, sum) })
        new CoordinateMatrix(productEntries)
    }

    def csvToCooMatrix(path: String): CoordinateMatrix = {
        val df = sparkSession.read.format("csv").option("header", "true").load(
            path)
```

```
        val tempDF = df.withColumn("i", df("i").cast("long")).withColumn("j", df
            ("j").cast("long")).withColumn("v", df("v").cast("double"))
        val corMat = new CoordinateMatrix(tempDF.rdd.map(m => MatrixEntry(m.
            getLong(0), m.getLong(1), m.getDouble(2))))
        corMat
    }

    def cooMatToDf(cooMat: CoordinateMatrix) : sql.DataFrame = {
        val ivRDD = cooMat.entries.map({ case MatrixEntry(i, j, v) => (i, v) })
        val newDf = sparkSession.createDataFrame(ivRDD).toDF("i","v")
        newDf
    }
}
```

Dünn besetzte Matrix-Vektor-Multiplikation in Spark SQL

Der folgende Code berechnet ebenfalls ein dünn besetztes Matrix-Vektor-Produkt $A\mathbf{x}$, nutzt jedoch die relationalen Operationen aus Spark SQL, welche der SQL-Implementation aus Abschnitt 7.1 nachempfunden wurde.

```
import org.apache.spark.sql.{DataFrame, SparkSession}

object MatrixMult {

    val mst = $MASTER_URL
    val name = "MatrixMultiplication"
    val sparkSession = SparkSession.builder().appName(name).master(mst).
        getOrCreate()

    def main(args: Array[String]): Unit = {

        val path1 = "/m1.csv"
        val path2 = "/m2.csv"

        val df1 = csvToDf(path1)
        val df2 = csvToDf(path2)

        val newMat = coordinateMatrixMultiply(df1, df2)
        newMat.count()
    }

    def coordinateMatrixMultiply(preLdf: DataFrame, preRdf: DataFrame):
        DataFrame = {
        import org.apache.spark.sql.functions._
        import sparkSession.implicits._
```

```
        val ldf = preLdf.withColumn("t",preLdf("v")).select("j","i","t")

        val rdf = preRdf.withColumnRenamed("i","k").withColumnRenamed("j","l").
            withColumnRenamed("v","m")

        val joinedDf = ldf.join(rdf).where(ldf("j") === rdf("k"))
        val preFinDf = joinedDf.withColumn("v",joinedDf("t") * joinedDf("m")).
            select("i","l", "v")

        val finDf = preFinDf.withColumnRenamed("l","j")

        finDf.groupBy("i","j").agg(sum($"v")).withColumnRenamed("sum(v)","v")
    }

    def csvToDf(path: String): DataFrame = {
        val df = sparkSession.read.format("csv").option("header", "true").load(
            path)
        df
    }
}
```

B.5.4 FFT in SQL und Python

Im Folgenden ist der vollständige Code einer Überführung des iterativen Radix-2-DIT-Algorithmus (Algorithmus 17) aus Abschnitt 9.3 in SQL:1999 präsentiert. Die Schemata entsprechen jeweils den Coordinate-Relationenschema mit komplexen Werten (dargestellt mittels zwei Attributen). Aus Gründen der Platzersparnis wird hier nur der Fall $n = 2^3 = 8$ dargestellt, da für die Berechnung $\log_2(n)$ rekursive Anfragen benötigt werden. Größere Dimensionen der Form $n = 2^p$ können analog zu den dargestellten `update`-Schritten umgesetzt werden. Nach initialem Einfügen der Zeitreihenwerte, die per *bit reverse order* umstrukturiert worden sind, werden die dreifach geschachtelten Schleifen umgesetzt. Die inneren beiden Schleifen werden durch eine rekursive Anfrage zusammengefasst.

```
insert into f
select bro.v,x.re,x.im
from x join bro
on x.i=bro.i;

with recursive xrec(i,re,im,j) as (
   with z(i,re,im) as (
      select xph.i, xph.re*w.re-xph.im*w.im, xph.re*w.im+xph.im*w.re
```

```
        from ( select * from f where i>=1 and i<2 ) xph join w
           on 2^(13)*(xph.i-1)=w.i
    ), xtemp(i,re,im) as (
        with xjz(i,xRe,xIm,zRe,zIm) as (
           select f.i, f.re, f.im, z.re, z.im
           from f join z on f.i=z.i-1
        )
        select i, xRe+zRe, xIm+zIm from xjz
        union all
        select i+1, xRe-zRe, xIm-zIm from xjz
    )
    select i, re, im, 2 from xtemp
    union all
    select i, re, im, cast(j+2^1 as integer)
    from (
        with z(i,re,im,j) as (
           select xph.i, xph.re*w.re-xph.im*w.im, xph.re*w.im+xph.im*w.re,
              xph.j
           from (
              select f.i as i, f.re as re, f.im as im, xrect.j as j
              from f, ( select distinct j from xrec ) xrect
              where f.i>=1+xrect.j and f.i<2+xrect.j
           ) xph join w on 2^(13)*(xph.i-1-xph.j)=w.i
        ), xjz(i,xRe,xIm,zRe,zIm,j) as (
           select f.i, f.re, f.im, z.re, z.im, z.j
           from f join z on f.i=z.i-1
        )
        select i as i, xRe+zRe as re, xIm+zIm as im, xjz.j as j from xjz
        union all
        select i+1, xRe-zRe, xIm-zIm, xjz.j from xjz
    ) xtemp
    where j<8
)
update f set re=xrec.re , im=xrec.im
from xrec
where xrec.i=f.i;
```

```
with recursive xrec(i,re,im,j) as (
   with z(i,re,im) as (
      select xph.i, xph.re*w.re-xph.im*w.im, xph.re*w.im+xph.im*w.re
      from ( select * from f where i>=2 and i<4 ) xph join w
         on 2^(12)*(xph.i-2)=w.i
   ), xtemp(i,re,im) as (
      with xjz(i,xRe,xIm,zRe,zIm) as (
         select f.i, f.re, f.im, z.re, z.im from f join z on f.i=z.i-2
      )
      select i, xRe+zRe, xIm+zIm from xjz
      union all
      select i+2, xRe-zRe, xIm-zIm from xjz
   )
   select i, re, im, 4 from xtemp
   union all
   select i, re, im, cast(j+2^2 as integer)
   from (
      with z(i,re,im,j) as (
         select xph.i,xph.re*w.re-xph.im*w.im,xph.re*w.im+xph.im*w.re,xph.j
         from (
            select f.i as i, f.re as re, f.im as im, xrect.j as j
            from f, (select distinct j from xrec) xrect
            where f.i>=2+xrect.j and f.i<4+xrect.j
         ) xph join w on 2^(12)*(xph.i-2-xph.j)=w.i
      ), xjz(i,xRe,xIm,zRe,zIm,j) as (
         select f.i, f.re, f.im, z.re, z.im, z.j
         from f join z on f.i=z.i-2
      )
      select i as i, xRe+zRe as re, xIm+zIm as im, xjz.j as j from xjz
      union all
      select i+2, xRe-zRe, xIm-zIm, xjz.j from xjz
   ) xtemp
   where j<8
)
update f set re=xrec.re , im=xrec.im from xrec where xrec.i=f.i;

with recursive xrec(i,re,im,j) as (
```

```
    with z(i,re,im) as (
       select xph.i, xph.re*w.re-xph.im*w.im, xph.re*w.im+xph.im*w.re
       from ( select * from f where i>=4 and i<8 ) xph join w
          on 2^(11)*(xph.i-4)=w.i
    ), xtemp(i,re,im) as (
       with xjz(i,xRe,xIm,zRe,zIm) as (
          select f.i, f.re, f.im, z.re, z.im
          from f join z on f.i=z.i-4
       )
       select i, xRe+zRe, xIm+zIm from xjz
       union all
       select i+4, xRe-zRe, xIm-zIm from xjz
    )
    select i, re, im, 8 from xtemp
    union all
    select i, re, im, cast(j+2^3 as integer)
    from (
       with z(i,re,im,j) as (
          select xph.i, xph.re*w.re-xph.im*w.im, xph.re*w.im+xph.im*w.re, xph.j
          from (
             select f.i as i, f.re as re, f.im as im, xrect.j as j
             from f, ( select distinct j from xrec ) xrect
             where f.i>=4+xrect.j and f.i<8+xrect.j
          ) xph join w on 2^(11)*(xph.i-4-xph.j)=w.i
       ), xjz(i,xRe,xIm,zRe,zIm,j) as (
          select f.i, f.re, f.im, z.re, z.im, z.j
          from f join z on f.i=z.i-4
       )
       select i as i, xRe+zRe as re, xIm+zIm as im, xjz.j as j
       from xjz
       union all
       select i+4, xRe-zRe, xIm-zIm, xjz.j
       from xjz
    ) xtemp
    where j<8
)
update f set re=xrec.re, im=xrec.im from xrec where xrec.i=f.i;
```

FFT und STFT in Python

Die Berechnung des Radix-2-DIT-Verfahrens in Python wurde mittels

```
''def FFT_py(f):
    import cmath
    def _reverse(x, bits):
        y = 0
        for _ in range(bits):
            y = (y << 1) | (x & 1)
            x >>= 1
        return y

    # Initialization

    exptable = [cmath.exp(-2j * cmath.pi * i / n) for i in range( int (n / 2) )]
    f = [f[_reverse(i, log2n)] for i in range(n)]  # Copy with bit-reversed
        permutation

    # Radix-2 decimation-in-time FFT
    size = 2
    while size <= n:
        h = int(size / 2)
        tablestep = int(n / size)
        for i in range(0, n, size):
            k = 0
            for j in range(i, i + h):
                temp = f[j + h] * exptable[k]
                f[j + h] = f[j] - temp
                f[j] += temp
                k += tablestep
        size *= 2
    return f
```

umgesetzt.
Für die STFT wurde der folgenden Code genutzt:

```
''def stft_paper_experiment (sa_hz=44100, song_duration=180, window_size=512,
    res_file = 'stft_erg.csv', dbs='psql' ):

    import numpy as np
    from time import time

    def FFT_py(f):
        import cmath
        def _reverse(x, bits):
            y = 0
```

```
            for _ in range(bits):
                y = (y << 1) | (x & 1)
                x >>= 1
            return y

    # Initialization
    _n = len(f)
    log2n = int(np.log2(_n))

    exptable = [cmath.exp(-2j * cmath.pi * i / _n) for i in range( int (_n / 2)
        )]
    f = [f[_reverse(i, log2n)] for i in range(_n)]  # Copy with bit-reversed
        permutation

    # Radix-2 decimation-in-time FFT

    size = 2
    while size <= _n:
        h = int(size / 2)
        tablestep = int(_n / size)
        for i in range(0, _n, size):
            k = 0
            for j in range(i, i + h):
                temp = f[j + h] * exptable[k]
                f[j + h] = f[j] - temp
                f[j] += temp
                k += tablestep
        size *= 2
    return f

    transforms = []
    n = sa_hz*song_duration
    t0=time()
    db = db_connection("$db","localhost",5432,"$user","$pw").connect_to_db(dbs)
    cur = db.cursor()
    cur.execute('select Re from x order by i')
    S = list(zip(*cur.fetchall()))[0]


    step_size = int(window_size/2)

    # Starting
    t1 = time()
    akt_ind = 0
    while akt_ind+window_size < n:
        transforms.append(FFT_py(S[akt_ind:(akt_ind+window_size)]))
```

```
        akt_ind += step_size

    t_py = time() - t1


    for j,t in enumerate(transforms):
        cur.execute('insert into stft_py values ' +     ','.join(cur.mogrify("(%
            s,%s,%s,%s)",(j,ii,ss.real,ss.imag)).decode('utf-8') for ii,ss in
            enumerate(t) ))

    db.commit()
    cur.close()
    db.close()
    t_py_db = time() - t0

    print(f"-> STFT in Python without csv took {t_py} s")
    with open(res_file,'a') as f:
        f.write(f'{n},{window_size}, Py Pure Calculation, {t_py} \n')

    print(f"-> STFT in Python with {dbs} took {t_py_db} s")
    with open(res_file,'a') as f:
        f.write(f'{n},{window_size}, Py IO PSQL, {t_py_db} \n')
```

B.6 Zusammenfassung und Ausblick

Im Folgenden wird die Hauptkomponentenanalyse vorgestellt und deren Überführung in SQL diskutiert. Hierbei wird insbesondere auf die Berechnung von Eigenwerten und -vektoren, sowie die Orthogonalisierung von Matrizen in SQL eingegangen. Die Untersuchung bezieht sich auf die Ausführungen zum Provenance Management in Abschnitt 10.2.1.

B.6.1 R-Implementation: Lernen der HMM-Transitionsmatrix

Für den in Abschnitt 10.2.2 beschriebenen Vergleich zur Berechnung der Transitionsmatrix in R und MonetDB wurde die anschließend dargestellte R-Funktion genutzt.

```
trainModel <- function(anno){
    tab <- table(anno[-1],anno[-length(anno)]);
    result <- tab %*% diag (1/colSums(tab));
    dimnames(result) <- dimnames(tab);
    result
}
```

B.6.2 Die Hauptkomponentenanalyse in SQL

Die Hauptkomponentenanalyse (im englischen: *Principal Component Analysis*, kurz: PCA) ist ein weit genutztes Verfahren der multivariaten Statistik zur Datenanalyse und Dimensionsreduktion. Ziel des Verfahrens ist das Finden einer Basis eines Unterraums, die den aufgespannten Vektorraum einer betrachteten Datenmenge möglichst optimal approximiert. Im Folgenden wird die prinzipielle Struktur der PCA erklärt und anschließend eine Überführung in SQL vorgestellt und diskutiert. Die mathematischen Beschreibungen basieren hierbei auf der Einführung aus [157], auf welche für weiterführende Informationen verwiesen wird.

Verfahrensbeschreibung

Es sei ein Datensatz in Form einer Matrix $X \in \mathbb{R}^{m \times n}$ betrachtet. In der Praxis entspricht hierbei oftmals m der Anzahl verschiedener Zeitreihen der Länge n. Ziel des Verfahren ist die Suche einer optimalen Basisdarstellung bestehend aus Linearkombinationen der ursprünglichen Zeilen von X. Aus diesen soll eine Unterraumbasis bestimmt werden, welche den ursprünglichen Datenbestand weitestgehend verlustfrei mit weniger Basisvektoren (Dimensionen) abbilden kann.
Wie in [157] motiviert wird, kann dies erreicht werden durch die Berechnung der Eigenvektoren[1] der symmetrischen positiv definiten Kovarianzmatrix $C = \frac{1}{n} X X^T \in \mathbb{R}^{m \times m}$ von X. Hierbei sollten, gemäß der Definition der Kovarianz, die Zeilen von X im Vorhinein bezüglich deren Mittelwerte korrigiert werden: $\mathbf{x}_i = \mathbf{x}_i - \frac{1}{n}\mathbf{sum}(\mathbf{x}_i)$. Eine Eigenwertzerlegung von C liefert in diesem Fall das Produkt

$$D = Q^T C Q,$$

wobei D die Diagonalmatrix ist, die absteigend sortiert die Eigenwerte von C enthält, und Q die orthogonale Matrix der Eigenvektoren (in diesem Kontext Hauptkomponenten genannt) von C bezüglich der Eigenwerte aus D ist. Die Eigenwerte sind in diesem Fall so konstruiert, dass diese der Varianz der zugehörigen Hauptkomponenten entspricht. Aus diesem Grund können nach der Eigenwertanalyse die Hauptkomponenten zu Eigenwerten um den Wert 0 ausgespart werden, da diese nur wenig Information bzw. lediglich Rauschen beinhalten.
Die Berechnung der PCA kann ebenfalls mittels Singulärwertzerlegung umgesetzt werden und wurde beispielsweise in [88] in SQL auf dem row store Microsoft SQL Server mit vergleichsweise niedrig dimensionalen Daten umgesetzt. Hierbei wurde eine Variation der im Folgenden genutzten QR-Verfahren mittels Householder-Transformationen umgesetzt. Die dort genutzten Relationenschemata und die ausgesparte Schachtelung von Anfragen haben, nach den Analysen aus Abschnitt 6.4 und 7.3, vermutlich einen negativen Einfluss (speziell bei dort nicht betrachteten großen Mengen an Zeitreihen m) auf die Performance des propagierten Ansatzes.

[1] Das Tupel (λ,$\mathbf{v}$) heißt Eigenpaar (Eigenwert,Eigenvektor) einer Matrix $A \in \mathbb{R}^{n \times n}$, wenn $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ gilt ($\mathbb{C}^n \ni \mathbf{v} \neq \mathbf{0}, \lambda \in \mathbb{C}$).

Umsetzung in SQL: Mittelwert-Korrektur und Kovarianzmatrix

Wie im vorigen Abschnitt beschrieben besteht die hier diskutierte Variante der PCA im wesentlichen aus den folgenden Schritten

1. Mittelwertkorrektur von X: $X = X - \begin{pmatrix}1 & 1 & \dots & 1\end{pmatrix}^T \mu^T$
2. Berechnung der Kovarianzmatrix aus X: $C = \frac{1}{n} X X^T$
3. Berechnung der Eigenwertzerlegung von C:

$$D = Q^T C Q$$

Die Struktur lässt mehrere Schlüsse zu. Im Allgemeinen ist die Anzahl der Zeitstempel n wesentlich größer als die der verschiedenen Zeitreihen m ($n \gg m$). Demnach ist vor allem die Berechnung der Kovarianzmatrix datenintensiv, wo hingegen die Berechnung einer Eigenwertzerlegung hoch iterativ und intensiv in der Anzahl der Fließkomma-Operationen ist. Aus diesem Grund lässt sich im Allgemeinen vermuten, dass vor allem die Berechnung der Kovarianzmatrix (inklusive Mittelwertkorrektur) effizient im Datenbanksystem umgesetzt werden kann. Durch die einhergehende Reduktion der Datenmenge ist eine externe Umsetzung der Eigenwertverfahren (vgl. die Architekturdiskussion aus Abschnitt 6.1) möglicherweise sinnvoll. Im Folgenden wird jedoch eine reine Standard-SQL-Übersetzung verschiedener Verfahren vorgestellt.
Zunächst sei jedoch für die Berechnung der Kovarianzmatrix angenommen, dass die eigentliche Matrix $X \in \mathbb{R}^{m \times n}$ in der Relation `x` mit dem Coordinate-Relationenschema `x(i int, j int, v double precision NOT NULL, primary key (i,j) )` hinterlegt ist. Die Mittelwertkorrektur und die Berechnung der Matrix können dann über die geschachtelte Anfrage

```
insert into c
with co(i,j,v) as (
   with updx (i,j,v) as (
      select x.i, x.j, x.v-mu.v
      from x join (
         select i, avg(v)
         from x
         group by i
      ) mu(i,v) on x.i=mu.i
   )
   select x1.i, x2.i, sum(x1.v * x2.v) / (select max(j) from x)
   from updx x1 join updx x2 on x1.j=x2.j
   where x1.i <= x2.i
```

```
    group by x1.i, x2.i
)
select * from co
union all
select j,i,v from co where i<>j
```

umgesetzt werden. Hierbei wurde zunächst, aufgrund der Symmetrie der Kovarianzmatrix, nur die obere Dreiecksmatrix berechnet, um unnötige Aggregationen zu meiden.

Adaptive Eigenwertverfahren: Die Potenzmethode

Mit der Erstellung der Kovarianzmatrix in der Relation `c` verbleibt die Berechnung einer Eigenwertzerlegung. Die Wahl eines geeigneten Eigenwertverfahrens ist abhängig von dem ausgehenden Szenario. Ist die Kovarianzmatrix etwa hochdimensional und dünn besetzt (etwa durch Abschneiden niedriger Werte) ist es sinnvoll adaptive Verfahren für dünn besetzte Probleme zu wählen. Durch diese kann nach einer angemessenen Approximation der Datenmenge die Berechnung weiterer Vektoren eingestellt werden. In [158] wurde etwa die adaptive Nutzung der Potenzmethode, welche etwa integraler Teil des PageRanks aus Beispiel 3 ist, mittels Shifts propapgiert, um iterativ in absteigender Reihenfolge, die signifikanten Eigenpaare adaptiv zu berechnen. Eine Iteration zur Berechnung des maximalen Eigenwerts — in seiner einfachsten Variante — besitzt die Form

$$\mathbf{q}^{(i+1)} = C\mathbf{q}^{(i+1)} + \omega I_{m\times m},$$
$$\mathbf{q}^{(i+1)} = \frac{\mathbf{q}^{(i+1)}}{\sqrt{\langle \mathbf{q}^{(i+1)}, \mathbf{q}^{(i+1)} \rangle}}$$

wobei $I_{m\times m}$ die m-dimensionale Einheitsmatrix, ω der Shift-Parameter und $\mathbf{q}^{(i)}$ die Schätzung des Eigenvektors zum maximalen Eigenwert bezeichnet. Das Verfahren ist insbesondere direkt überführbar in SQL mittels den Basisoperatoren aus Abschnitt 7.1 und ist iterationsübergreifend (unter Nutzung von `with`-Klauseln für die Skalierung) schachtelbar. Mittels Deflation (vgl. etwa [107]) kann das gleiche Verfahren für die Berechnung des Eigenpaars bezüglich des nächst größeren Eigenwerts bestimmt werden. Hierfür wird dem Raum, der durch C aufgespannt wird, der Unterraum des berechneten Eigenpaars entfernt:

$$C = C - \lambda_1 \mathbf{q}_1 \mathbf{q}_1^T.$$

Der Ansatz ist analog zur eigentlichen Iteration mittels der etablierten Übersetzung und Schachtelung der Basisoperatoren umsetzbar und wird daher nicht näher aufgeführt.
Auch wenn das Verfahren vergleichsweise langsam konvergiert, ist es durch seinen adaptiven Cha-

rakter besonders gut geeignet zur Bestimmung der k einflussreichsten Sensoren aus hochdimensionalen Räumen. Im Gegenzug hierzu wird im Folgenden eine Überführung des QR-Verfahrens diskutiert, welches eine vollständige Eigenwertzerlegung einer Matrix berechnet.

Vollständige Eigenwertzerlegung: Das QR-Verfahren

In dicht besetzten Szenarien moderater Dimensionen ist eine vollständige spektrale Untersuchung oftmals sinnvoll. In diesem Fall werden durch iterative Eigenwertverfahren alle Eigenvektoren und Eigenwerte simultan berechnet.

Eines der wohl verbreitetsten und numerisch stabilsten Verfahren für diesen Zweck ist das QR-Verfahren, welches — seinen Namen entsprechend — auf der QR-Zerlegung beruht. Letzteres beschreibt die Faktorisierung einer Matrix $A \in \mathbb{R}^{n\times n}$ durch das Produkt $A = QR$ mit einer orthogonalen Matrix $Q \in \mathbb{R}^{n\times n}$ und einer Dreiecksmatrix $R \in \mathbb{R}^{n\times n}$. Weitere Details, wie Bedingungen zur Existenz und Eindeutigkeit solch einer Zerlegung, sind etwa in [107] beschrieben.

Grundlegend besteht eine Iteration des QR-Verfahrens aus den zwei Schritten

$$\begin{aligned} A_i &= Q_i R_i \\ A_{i+1} &= R_i Q_i = Q_i^T A_i Q_i, \end{aligned}$$

wobei $Q_i R_i$ der QR-Faktorisierung von A_i entspricht. Die Berechnung der Faktorisierung kann mittels verschiedener Methoden[2] berechnet werden und ist im allgemeinen Fall $\mathcal{O}(n^3)$ Fließkommaoperationen kostenintensiv. Zur Beschleunigung der Methode existieren verschiedene Ansätze. Hier werden zwei klassische Ansätze beschrieben und übersetzt: die Vorkonditionierung durch Tridiagonalisierung und das Einführen von Shift-Strategien.

Tridiagonalisierung und QR-Faktorisierung der Kovarianzmatrix

Die Überführung in Tridiagonalgestalt ist neben der Kostenreduktion der Iterationen interessant, da ein Iterationsschritt des QR-Verfahrens invariant bezüglich der symmetrischen Tridiagonalform ist (mit sukzessiver kleiner werdenden Nebendiagonalelementen). Die Tridiagonalisierung wird über eine Ähnlichkeitstransformation umgesetzt, dass heißt es wird eine orthogonale Matrix V[3] konstruiert, sodass

$$T = VCV^T$$

[2] Klassischerweise werden das Gram-Schmidt-Verfahren, die Householder-Transformation oder Givens-Rotationen zur Berechnung der QR-Faktorisierung genutzt. Für nähere Details sei etwa auf [107] verwiesen.

[3] Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt orthogonal, wenn $A^T A = AA^T = I$ gilt.

einer symmetrischen Tridiagonalmatrix entspricht. Ist dann $(\lambda, \mathbf{w})$ ein Eigenpaar von T, ist $(\lambda, V^T\mathbf{w})$ ein Eigenpaar von C, da

$$\lambda\mathbf{w} = T\mathbf{w} = VCV^T\mathbf{w} = \lambda VV^T\mathbf{w} = \lambda\mathbf{w}$$

gilt. Demnach können die Eigenvektoren der Kovarianzmatrix durch eine einfache lineare Transformation aus den Eigenvektoren von T gewonnen werden. Als Verfahren zur Tridiagonalisierung wird hier das Verfahren von *Householder* aus Algorithmus 20 genutzt, welches etwa in [107] näher beschrieben wird. Die Kosten der QR-Zerlegung können so von $\mathcal{O}(n^3)$ auf $\mathcal{O}(n)$ reduziert werden.

Die QR-Zerlegung wurde in diesem Fall mittels *Givens-Rotationen* (vgl. [159]) umgesetzt. Eine Givens-Matrix (auch Givens-Rotation) $G_{kl}(\theta)$
$(k, l \in \{1, \ldots, n\}, k \neq l)$ ist eine orthogonale Matrix der Form

$$G_{kl}(\cos(\theta), \sin(\theta)) = \begin{cases} \cos(\theta) & \text{für } i = j = k \vee i = j = l \\ -\sin(\theta) & \text{für } (i = k \wedge j = l) \\ \sin(\theta) & \text{für } (i = l \wedge j = k) \\ 1 & \text{für } (i = j \wedge i \neq l \wedge i \neq l) \\ 0 & \text{sonst} \end{cases}$$

die eindeutig durch die drei Parameter k, l und θ bestimmt wird. Durch die Multiplikation mit einer Matrix von links $G_{kl}(\cos(\theta), \sin(\theta))A$ werden hier nur die k-te und l-te Zeile von A geändert. Zur Berechnung einer QR-Zerlegung einer Tridiagonalmatrizen kann daher das Produkt

$$\underbrace{\left(\prod_{i=1}^{n-1} G_{i,i+1}(\cos(\theta_i), \sin(\theta_i))\right)}_{=:Q^T} T = R$$

sukzessive bestimmt werden, wobei in jedem Schritt das Subdiagonalement der aktualisierten Matrix gezielt eliminiert wird. Dies kann etwa kostengünstig umgesetzt werden, durch das Nutzen der Werte

$$\cos(\theta_k) \approx \frac{a_{kk}}{\sqrt{a_{kk}^2 + a_{k+1,k}^2}} =: c$$
$$\sin(\theta_k) \approx \frac{a_{k+1,k}}{\sqrt{a_{kk}^2 + a_{k+1,k}^2}} =: s$$

für die k-te Givens-Rotation $G_{k,k+1}(c, s)$. In diesem Fall existieren numerische stabilere Berechnungen der Givens-Rotationen, die im Folgenden jedoch nicht umgesetzt worden sind. Es

Algorithmus 20 Tridiagonalisierung einer symmetrischen Matrix C mittels Householder-Transformationen.

for $i = 1, \ldots, n-2$ **do**
  $\alpha = -\mathbf{sign}(c_{i+1,i}) \cdot \sqrt{\sum_{k=i+1}^{n} c_{k,i}^2}$
  $r = \sqrt{0.5(\alpha^2 - c_{k+1,k}\alpha)}$
  $\mathbf{v} = [\mathbf{zeros}(k,1); c_{k+1,k} - \alpha; c_{k+2:n,k}]/(2 * r)$
  $P = I - 2\mathbf{v}\mathbf{v}^T$
  $C = PCP$
end for

sei hierfür etwa auf [159] verwiesen. Durch das sukzessive Aktualisieren der Matrix T wird diese in $n-1$ Schritten (mit jeweils $\mathcal{O}(1)$ Fließkomma-Operationen) in eine obere Dreiecksgestalt R überführt. Die Matrix Q kann, durch die Orthogonalität der Givens-Rotationen, mittels $Q = \prod_{i=1} G_{i,i+1}(\theta_i)^T$ bestimmt werden. Die QR-Zerlegung mittels Givens-Rotationen ist für den Kontext von Tridiagonalmatrizen in Algorithmus 21 dargestellt.

Beschleunigung des QR-Verfahrens durch Shift-Strategien

Als zweite maßgebliche Strategie zur Beschleunigung des Verfahrens wird eine Shift-Strategie eingeführt.
In dieser werden die Kovarianzmatrizen C_i durch Einfügen von Shifts $C_i - \kappa I$ vor der Berechnung der QR-Zerlegung manipuliert. Für eine Faktorisierung $C_i - \kappa I = Q_i R_i$ folgt

$$Q_i^T(C_i - \kappa I)Q_i = Q_i^T C_i Q_i - \kappa I,$$

sodass für die Iterierte C_{i+1} durch nachträglicher Addition des Shifts

$$C_{i+1} = Q_i^T(C_i - kI)Q_i + kI$$

die Ähnlichkeit der Iterierten zur Ausgangsmatrix C sichergestellt wird. Durch eine Wahl von κ nahe an einen Eigenwert λ können die Konvergenzraten deutlich verbessert werden. In der folgenden SQL-Implementation wird die einfache Strategie $\kappa = c_{nn}$ aus [107] genutzt. Als mögliche Verbesserung dieser, sei auf Arbeiten von *Wilkinson* in [160] verwiesen, in welcher gezeigt wurde, dass durch die Wahl des Eigenwertes λ von

$$\begin{pmatrix} c_{n-1,n-1}^{(i)} & c_{n-1,n}^{(i)} \\ c_{n-1,n}^{(i)} & c_{n,n}^{(i)} \end{pmatrix}$$

für den $|c_{n,n}^{(i)} - \lambda|$ kleiner ist, die Konvergenzgeschwindigkeit mindestens quadratisch, aber oft kubisch ist.

Algorithmus 21 Die Berechnung einer QR-Zerlegung einer symmetrischen Tridiagonalmatrix T mittels Givens-Rotation.

1: $Q = I, R = T$
2: **for** $i = 1, \ldots, n-1$ **do**
3: $\quad c = r_{kk}/\sqrt{r_{kk}^2 + r_{k+1,k}^2}$
4: $\quad s = r_{k+1,k}/\sqrt{r_{kk}^2 + r_{k+1,k}^2}$
5: $\quad Q = Q \cdot G_{k,k+1}(c,s)^T$
6: $\quad R = G_{k,k+1}(c,s)^T R$
7: **end for**

Algorithmus 22 Das QR-Verfahren zur Berechnung der vollständigen Eigenwertzerlegung der Kovarianzmatrix C.

1: Input: $C = \textbf{Kovarianzmatrix}(X)$ ($X \in \mathbb{R}^{m \times n}$-Datensatz)
2: $[Q_C, R]$ =Tridiagonalisierung von C nach Algorithmus 20 ▷ Tridiagonalform in R gespeichert
3: **for** $i = 1, \ldots,$`max_iteration` **do**
4: $\quad$ Bestimmung des Shift-Parameters $\kappa = c_{nn}$
5: $\quad [Q, R] = QR$-Faktorisierung von $R - \kappa I$ mittels Givens-Rotation nach Algorithmus 21
6: $\quad R = RQ + \kappa I$
7: $\quad$ **if** Abbruchbedingung **then**
8: $\quad\quad$ **break**
9: $\quad$ **end if**
10: **end for**
11: Return: (R,$Q_C^T Q$) ▷ Transformation der EV fuer C

Ferner können die Werte der unteren Nebendiagonale genutzt werden, um eine sinnvolle Abbruchbedingung zu erstellen. Sind diese etwa im Vergleich zu den Diagonalwerten sehr klein, kann von einer akzeptablen numerischen Genauigkeit der Eigenwerte ausgegangen werden. Beispielsweise ist

$$\min(|c_{n-1,n}^{(i)}|, |c_{n-1,n-2}^{(i)}|) \leq \varepsilon(|c_{n,n}^{(i)}| + |c_{n-1,n-1}^{(i)}|)$$

mit der Maschinengenauigkeit ε eine geeignetes Abbruchkriterium.
Mit diesen finalen Betrachtungen kann die PCA mittels QR-Verfahren, wie in Algorithmus 22 beschrieben, in SQL überführt werden.

Erstellung des SQL-Anfrageplans

Im Folgenden wird ein Shell-Skript zur Erstellung des Anfrageplans zur Tridiagonalisierung der Kovarianzmatrix und des QR-Verfahrens vorgestellt. Dieses erstellt lokal die Dateien *Kovarianz.sql*, *HHTridiag.sql*, *QR_EV.sql*, welche die jeweiligen SQL-Anfragepläne enthalten und

hintereinander vom Datenbanksystem verarbeitet werden müssen. In `QR_EV.sql` werden hier 10 (veränderbar) Iterationsschritte des QR-Verfahrens durchgeführt. Dieser Anfrageplan muss demnach wiederholt an die Datenbank gesendet werden, bis eine zu verfassende Abbruchbedingung erfüllt ist. Die Implementation benutzt die vergleichsweise neue `truncate`-Klausel zum Löschen aller Tupel einer Relation. Wird dies nicht unterstützt, ist eine Umsetzung mittels des meist langsameren `delete from`-statements möglich. Die Effizienz der Implementation ist optimierbar, da zum einen Potenzial für weitere Anfrage-Schachtelungen vorliegt und zum anderen eine bessere Nutzung von `update`-Klauseln, anstatt dem physischen Schreiben ganzer aktualisierter Relationen, möglich ist. Zudem ist in der Implementation der QR-Zerlegung mittels Givens-Rotation eine Anpassung für hochdimensionale Probleme nötig, da der aktuelle Ansatz zu Overflow-Problemen führen kann. Für etwaige stabilere Strategien sei auf [159] verwiesen. Im Allgemeinen werden in solchen Verfahren Parameter aus dem Vergleich von Elementen der zu faktorisierenden Matrix gewählt. Dieser dynamische Vergleich ist, etwa über externe Schnittstellen, nur ungünstig in Standard-SQL umsetzbar. Da im Laufe des Verfahrens vergleichsweise viele Rotationen durchgeführt werden, ist ein signifikanter negativer Effekt durch die frequente Kommunikation zwischen Datenbanksystem und externer Schnittstelle zu erwarten (vgl. etwa Ausführungen in Abschnitt 7.3). Für eine vergleichsweise effizientere Umsetzung ist zudem systematisch zu testen, inwiefern die Relationen der Kovarianzmatrix `$C` und `$Ct` durch Indexstrukturen profitieren könnte. Auf diesen werden mehrfach sehr selektive Anfragen für die Umsetzung der Multiplikation und Erstellung der Givens-Rotationen gestellt. Da zudem auch mehrfach große Datenmengen eingefügt werden, sind die Zusatzkosten der Indexverwaltung zu beobachten.

Es folgt nun ein Basis-Shell-Skript zur Erstellung von SQL-Anfrageplänen für die Berechnung der Hauptkomponentenzerlegung mittels Mittelwertkorrektur, Tridiagonalisierung und QR-Zerlegung mit Shift-Strategie.

```
X="x" # in mxn, Read-Only
C="c" # in nxn
Ct="${C}temp"
Q="q"
v="v"
P="p"
shift="shift"   # 1x1 shift value for qr method
ex_step=20
n=$n
max\_ite=10
ev_step=30
Array_k=1

clear
rm Kovarianz.sql HHTridiag.sql QR_EV.sql
```

```
echo "Erstelle Query Plan"
echo "Starte Mittelwertkorrektur und Kovarianzmatrixplaene"

sql="drop table $C;
drop table $Ct;
drop table $v;
drop table $P;
drop table $Q;
create table $C ( i int, j int, v double precision );
create table $Ct (i int, j int, v double precision );
create table $v ( i int, v double precision );
create table $P ( i int, j int, v double precision );
create table $Q ( i int, j int, v double precision );"

# Mittelwertkorrektur
# X = X - mu I_nxm
sql="$sql
update $X
set v = ${X}.v - mu.v
from (
    select j, avg(v) as v
    from $X
    group by j
) mu
where $X.j=mu.j;"

# Kovarianzmatrix
# C = X\^T X
sql="$sql
insert into $C
select x1.j,x2.j, sum(x1.v*x2.v)
from $X x1 join $X x2 on x1.i=x2.i
group by x1.j,x2.j;"

alt="${C}"
neu="${Ct}"

echo "$sql
" >> Kovarianz.sql
sql=""

echo "Starte Tridiagonalisierungsplan"

for k in `seq 1 $((${n}-2))`
do
```

```
    alpha="(
select sqrt(sum( v*v )) * (select -sign(v) from $alt where i=${k}+1 and j=${k})
   from $alt where i>$k and j=${k}
)"
    r="(
select sqrt(0.5 * $alpha * ( $alpha - v ) ) from $alt where i=${k}+1 and j=$k
)"
    vstr="select i, (v - ${alpha})/(2*${r}) as v from $alt where i=${k}+1 and j=
        $k
union all
select i,v/(2*${r}) as v from $alt where i>${k}+1 and j=${k}"

    Pstr="with recursive identity (i,j,v) as (
select 1,1,1.0
union all
select i+1,j+1,1.0
from identity
where i<$k
)
select * from identity
union all
select vdiag.i as i, vdiag.i as i, 1 - 2*vdiag.v\^2 as v from $v vdiag
union all
select vrest1.i as i, vrest2.i as j, -2*vrest1.v*vrest2.v as v
from $v vrest1, $v vrest2
where vrest1.i<>vrest2.i"

# T_neu = P T_alt P
    PCP="insert into $neu
select PC.i, Pt.j, sum(PC.v*Pt.v)
from (
    select Pt.i as i, ${alt}.j as j, sum(Pt.v*${alt}.v) as v
    from $P Pt join $alt on Pt.j=${alt}.i
    group by Pt.i, ${alt}.j
) PC join $P Pt on PC.j=Pt.i
where not ((PC.i=$k and Pt.j>${k}+1 ) or ( Pt.j=$k and PC.i>${k}+1) )
group by PC.i, Pt.j;
"

    sql="${sql}
truncate $neu;
truncate $v;
truncate $P;
insert into $v $vstr;
insert into $P $Pstr;
$PCP"
```

```
    if [ $(( $k % $ex_step == 0 )) -gt 0 ];
    then
        echo "$sql
        " >> HHTridiag.sql
        sql=""
    fi
    alt2="$alt"
    alt="$neu"
    neu="$alt2"
done

if [ "$alt" != "$C" ]
then
    sql="${sql}
truncate $C;
insert into $C select * from $Ct;"
fi

if [ $(( ( ${n}-1 ) % $ex_step )) -gt 0 ]
then
    echo "$sql" >> HHTridiag.sql
    sql=""
fi

echo "Starte QR-Verfahrensplan"

sql="drop table $shift ;
drop table $grot ;
create unlogged table $shift (v double precision);
create unlogged table $grot (i int, j int, v double precision);"

alt="$C"
neu="$Ct"

# zweite matrix ist basis fuer daten und wird manipuliert
# erste matrix nur fuer zwischenergebnisse genutzt
qr\_C\_Ct=`./qr\_giv.sh $n $C $Ct $Q $grot`
qr\_Ct\_C=`./qr\_giv.sh $n $Ct $C $Q $grot`


for i in `seq 1 $max_ite`
do
    sql="${sql}
truncate $shift ;
insert into $shift select v from $alt where i=$n and j=$n ;
```

```
update $alt set v = v - ( select v from $shift )
where i=j;"

    if [ $(( $i % 2 )) -eq 0  ]
    then
        sql="${sql}
${qr\_C\_Ct}"
    else
        sql="${sql}
${qr\_Ct\_C}"
    fi

    # alt enthaelt R, $Q enthaelt Q
    # RQ erstellt obere Dreiecks matrix ist aber tridiag form
    sql="${sql}
truncate $neu ;
insert into $neu
select ${alt}.i,${Q}.j, sum(${alt}.v*${Q}.v)
from $alt join $Q on ${alt}.j=${Q}.i
group by ${alt}.i,${Q}.j;
update $neu set v = v + (select v from $shift)
where i=j;
delete from $neu where (abs(v)<1e-10 and i-j not in (0,1) );
"

    alt2="$alt"
    alt="$neu"
    neu="$alt2"

    sql="$sql

    -- Ende Iteration $i ----------------------------

    "
    if [ $(( $i % $ev_step )) -eq 0 ]
    then
        echo "$sql
        " >> QR_EV.sql
        sql=""
    fi
done
echo "$sql
" >> QR\_EV.sql
```

Die Berechnung der QR-Zerlegung mittels Givens-Rotationen kann hierbei durch das folgende lokal abzulegende Shell-Skript `qr_giv.sh` umgesetzt:

```
# QR Zerlegung mittels Givens Rotation fuer Tridiagonalmatrizen
# hier noch numerisch instabile Berechnung (Overflow) von Cosinus- und Sinus-
    Werten in Givens-Rotationen moeglich
# Dynamische Anpassung (vgl. Literatur, etwa Stoer und Bulirsch- Numerische
    Mathematik 1) fuer groessere Dimensionen noetig; fuehrt jedoch zu
    Performance-Einbussen

n=$1        # Dim
temp=$2     # Zwischenergebnisspeicher
r=$3        # Dreiecksmatrix R
q=$4        # Matrix Q

sql="truncate $q;
insert into $q
with recursive identity (i,j,v) as (
    select 1,1,1.0
    union all
    select i+1,j+1,1.0
    from identity
    where i<$n
)
select * from identity;
"
for k in `seq 1 $(($n-1))`
do
    kp1=$(( $k + 1 ))

    Gstr="(
with givc(v) as ( select v / (select sqrt(sum(v*v)) from $r where i in (${k},${
    kp1}) and j=${k}) from $r where i=$k and j=$k ),
givs(v) as ( select v / (select sqrt(sum(v*v)) from $r where i in (${k},${kp1})
    and j=${k}) from $r where i=$kp1 and j=$k )
select $k,$k, (select v from givc)
union all
select $k,$kp1, (select v from givs)
union all
select $kp1,$k, (select -v from givs)
union all
select $kp1,$kp1, (select v from givc)
)"
    sql="${sql}
truncate $temp;
insert into $temp
select ${q}.i,G.i,sum(${q}.v*G.v)
from $q join ${Gstr} G(i,j,v) on ${q}.j=G.j
where ${q}.j between $k and $kp1
```

```
group by ${q}.i,G.i;
delete from $q where j between $k and $kp1 ;
insert into $q select * from $temp;
truncate $temp ;
insert into $temp
select G.i, ${r}.j, sum(G.v*${r}.v)
from $r join ${Gstr} G(i,j,v) on G.j=${r}.i
where ${r}.i between $k and $kp1
group by G.i, ${r}.j;
delete from $r where i in (${k},${kp1});
insert into $r select * from $temp where not (i=$kp1 and j=$k);
"
done
echo "$sql"
```

Diskussion zur Überführung in SQL

Die effiziente Überführung der PCA auf SQL-Systeme ist offensichtlich stark abhängig von der Wahl der genutzten inneliegenden Eigenwert- oder Singulärwertverfahren und der vorliegenden Dimension der Kovarianzmatrix. Die Berechnung der Kovarianzmatrix (nach Mittelwertkorrektur) ist aufgrund seiner SQL-affinen Struktur mit starkem Fokus auf Datenmanagement (Gruppierung) voraussichtlich sehr effizient und einfach in SQL-Systemen umsetzbar und parallelisierbar. Die Effizienz der SQL-Umsetzung zur Eigenanalyse hingegen ist abhängig von der Besetzungsstruktur der Kovarianzmatrix und der Unterscheidung zwischen Top-k-Komponentenanfragen und vollständiger Spektralanalyse. So ist die Nutzung der Potenzmethode für dünn besetzte Methoden für eine geringe Anzahl zu berechnender signifikanter Hauptkomponenten, ähnlich zu seiner Nutzung im PageRank aus Beispiel 3, mutmaßlich effizient umsetzbar und aufgrund seiner Matrix-Vektor-Form gut parallelisierbar (vgl. Kapitel 8). Die Berechnung aller Hauptkomponenten mittels QR-Verfahren ist im Sinne der Performance nach stichprobenartigen Tests vergleichsweise problematisch. Maßgeblich ist dies durch den hoch-iterativen und in sich aktualisierenden Charakter, sowie dem vergleichsweise hohen Verhältnis zwischen Fließkommaoperationen und Datenmanagement des QR-Verfahrens zu begründen. Wie etwa in Abschnitt 7.3 beschrieben, sind vor allem nicht-schachtelbare Iterationen in Standard-SQL nur ineffizient über ausgerollte Schleifen von leichtgewichtigen Anfragen umsetzbar. Als Lösung dieses Problems könnten systemabhängige nicht-standardisierte Schleifen-Funktionalitäten oder, wie in Abschnitt 6.1 diskutiert, eingeführte spezielle Lineare-Algebra-Funktionalitäten genutzt werden. Dies führt mutmaßlich zu einer deutlichen Beschleunigung des Verfahrens, verletzt jedoch die Systemunabhängig des hier diskutierten Konzepts aus Kapitel 3.
Für eine bessere Einschätzung der Effizienz und des Potenzials der Hauptkomponentenanalyse sind von diesem Punkt aus verschiedene künftige Untersuchungen nötig. Zum einen sind An-

passungen zur Wahrung der numerischen Stabilität und Schachtelung der Implementation des QR-Verfahrens nötig. Zum anderen sollten systematisch Vergleichstests in verschiedenen Datenbanksystemen und Tests zur Effizienz von Indexstrukturen durchgeführt werden. Mit den in dieser Arbeit gewonnen Erkenntnissen liegt es jedoch nahe, dass vor allem Szenarien mit dünn besetzten hochdimensionalen Kovarianzmatrizen am geeignetsten für Hauptkomponentenanalysen in relationalen Systemen sind.

B.7 Übersetzung von HMM-Methoden

Abschließend werden SQL-Implementation der HMM-Grundprobleme aus Abschnitt 5.2 vorgestellt. Hierbei werden neben den klassischen Ansätzen ebenfalls die vorgestellten Erweiterungen zur Wahrung der numerischen Stabilität aus Abschnitt 5.3 umgesetzt.

B.7.1 Forward-Variablen

Im Folgenden wird eine Python3-Funktion aufgelistet, welche die Standard-SQL-Anfragepläne zur Berechnung von Forward-Variablen erstellt. Als Argument wird die Beobachtungsliste `o` als Python-Liste benötigt, dessen Einträge die Indizes der beobachteten Symbole entspricht. Für die Wahl des Anfrageplans werden hierbei mehrere Varianten (durch `mode`) unterschieden. Für die Werte -1 und 0 werden alle α_t in einer Relation mit dem für Matrizen etablierten Coordinate-Relationenschema `alpha(i int, t int, v double precision)` gespeichert, wobei anstatt dem etablierten Spaltenbezeichner `j`, der Attributname `t` im Sinne des Kontexts genutzt wurde. Für `mode=-1` werden hierbei die α_t in einer einzelnen rekursiven Anfrage berechnet, wobei jedoch die Ablage der Beobachtungsfolge in einer zugehörigen Vektor-Relation `o ( i int, v int)` benötigt wird. In `mode=0` werden sukzessive die Forward-Variablen (pro Iteration eine Anfrage) berechnet. Dies ist der einzige Modus, für den die numerisch stabilere Skalierung implementiert wurde. In diesem Fall werden die Skalierungsfaktoren in der Relationen `skal_alp` gespeichert. Der Modus 1 berechnet ebenfalls sukzessive die Forward-Variablen, wobei in diesem Fall Schritt für Schritt nur der aktuelle Vektor berechnet wird und per **update** aktualisiert wird. Abschließend wird die Wahrscheinlichkeit $P(\mathbf{o} \mid \lambda)$ durch Summierung von α_T berechnet. In diesem Fall wird für α das Schema (`i int,v double precision`) benötigt. Im Modus 2 wird die Wahrscheinlichkeit in einer großen geschachtelten Anfrage berechnet. Es folgt nun die beschriebene Funktion.

```
def db_forward_verfahren(o,A='a',B='b',alp='alpha',pi='pi',mode=1,stable=False,
    skal_alp='salpha'):

    if mode == -1:
        o='o'
        sql =f'''with recursive recalpha (i,t,v) as (
    select {pi}.i,0,{B}.v*{pi}.v
```

```
    from {pi}, {B}, {o}
    where {pi}.i={B}.i and {B}.j={o}.v and {o}.i = 0
    union all
    select talpa.i,talpa.t+1,talpa.v*{B}.v
    from (
        select {A}.j as i, alpha2.t as t, sum({A}.v*alpha2.v) as v
        from {A} join ( select i,t,v from recalpha ) alpha2 on {A}.i=alpha2.i
        group by {A}.j, alpha2.t
    ) talpa, {B}, {o}
    where talpa.i={B}.i and {o}.i = talpa.t+1 and {B}.j={o}.v and talpa.t <= (
        select max(i) from {o})
)
insert into {alp}
select * from recalpha;'''

        return sql
    elif mode == 0:
        sql =[f'''insert into {alp}
    select {pi}.i,0,{B}.v*{pi}.v
    from {pi} join {B} on {pi}.i={B}.i
    where {B}.j={o[0]};
    ''',]

        if stable:
            sql.append(f'''insert into {skal_alp}
select t, sum(v)
from {alp}
where t = 0
group by t;
update {alp} set v = v / (select v from {skal_alp} where t=0)
where t = 0;
''')

        for j,ok in enumerate(o[1:]):
            j=j+1
            sql.append(f'''insert into {alp}
select talpa.i,{j},talpa.v*{B}.v as v
from (
    select {A}.j as i, sum({A}.v*{alp}.v) as v
    from {A} join {alp} on {A}.i={alp}.i
    where {alp}.t={j-1}
    group by {A}.j
) talpa join {B} on talpa.i={B}.i
where {B}.j={ok} and talpa.i={B}.i;
''')
            if stable:
```

```
                sql.append(f'''insert into {skal_alp}
select t, sum(v)
from {alp}
where t = {j}
group by t;
update {alp} set v = v / (select v from {skal_alp} where t={j})
where t = {j};
''')
    elif mode == 1:
    # update-based calculation of P(o)
        sql =[f'''insert into {alp}
select {pi}.i,{B}.v*{pi}.v
from {pi} join {B} on {pi}.i={B}.i
where {B}.j={o[0]};
''',]

        for ok in o[1:]:
            sql.append(f'''update {alp}
set v = (
select talpa.v*{B}.v as v
from (
    select {A}.j as i, sum({A}.v*{alp}.v) as v
    from {A} join {alp} on {A}.i={alp}.i
    group by {A}.j
    ) talpa join {B} on talpa.i={B}.i
    where {B}.j={ok} and {alp}.i={B}.i
)
where exists
(
    select 1
    from (
        select distinct j as i
        from {A}
    ) ttt
    where ttt.i={alp}.i
);''')

        sql.append(f'''select sum(v) from {alp};''')

    elif mode ==2:
    # calculating P(o) via nested query

        sql=['''select sum(v) from
(''',]
        for k in range(len(o)-1):
            sql.append(f'''    select alpta.i as i, alpta.v * {B}.v as v
```

```
    from (
    select {A}.j as i, sum(aa.v*{A}.v) as v
    from {A} join (''')

        sql.append(f'''    select {pi}.i as i, {pi}.v*{B}.v as v
    from {pi} join {B} on {pi}.i={B}.i
    where {B}.j={o[0]}''')

        for k in range(1,len(o)):
            sql.append(f'''    ) aa
    on aa.i = {A}.i
    group by {A}.j) alpta
    join {B} on alpta.i=B.i
    where {B}.j={o[k]}''')

        sql.append(f''') aaa'''
    )

    return '\n'.join(sql)
```

B.7.2 Backward-Variablen

Im Folgenden wird eine Python3-Funktion präsentiert, die verschiedene Standard-SQL-Anfragepläne zur Berechnung der Backward-Variablen vorstellt. Hierbei existieren 3 verschiedene Modi. In Modus 0 werden sukzessive die Backward-Variablen β_t berechnet und in der zugehörigen Relation mit dem Matrixschema `beta (i int, j int, v double precision)` gespeichert. Im Mode 1 wird die Wahrscheinlichkeit $P(\mathbf{o} \mid \lambda)$ durch sukzessives aktualisieren des β_t-Vektors berechnet. In Modus 2 wird ebenfalls eine einzelne geschachtelte Anfrage erstellt.

```
def db_backward_verfahren(o,A='a',B='b',bet='beta',pi='pi',mode=1):

    if mode == 0:
        sql =[f'''insert into {bet}
select i,{len(o)-1},1.0 from {pi};''',]

        for k in range(len(o)-1,0,-1):
            sql.append(f'''insert into {bet}
select {A}.i, {k-1}, sum({A}.v*betBk.v)
from (
    select {bet}.i, {bet}.v*{B}.v
    from {B} join {bet} on {B}.i={bet}.i
    where {B}.j={o[k]} and {bet}.t = {k}
) betBk (i,v) join {A} on betBk.i={A}.j
group by {A}.i;''')
```

```
    elif mode == 1:
    # update-based calculation of P(o)

        sql =[f'''insert into {bet}
select i,1.0 from {pi};''',]

        for k in range(len(o)-1,0,-1):
            sql.append(f'''update {bet}
set v = (
    select v
    from (
        select {A}.i as i, sum({A}.v*betBk.v) as v
        from (
            select {bet}.i , {bet}.v*{B}.v
            from {B} join {bet} on {B}.i={bet}.i
            where {B}.j={o[k]}
        ) betBk (i,v) join {A} on betBk.i={A}.j
        group by {A}.i
    ) ttt
    where ttt.i={bet}.i
)
where exists
(
    select i
    from (
        select distinct i as i
        from {A}
    ) ttt
    where ttt.i={bet}.i
);''')

        sql.append(f'''select sum({bet}.v*{pi}.v*{B}.v)
from {bet},{pi},{B}
where {bet}.i={pi}.i and {bet}.i={B}.i and {B}.j={o[0]};''')

    elif mode == 2:
    # calculation of P(o) nested

        sql=[f'''select sum(bbb.v*{pi}.v*{B}.v)
from
(''',]
        for k in range(len(o)-1):
            sql.append(f'''    select {A}.i as i, sum({A}.v*betBk.v) as v
from (
    select bbb.i , bbb.v*{B}.v
    from {B} join (''')
```

```
        sql.append(f'''             select {pi}.i, 1.0
        from {pi}''')

        for k in range(len(o)-1,0,-1):
            sql.append(f'''         ) bbb(i,v) on bbb.i={B}.i
                where {B}.j={o[k]}
            ) betBk (i,v) join {A} on betBk.i={A}.j
            group by {A}.i''')

        sql.append(f'''
) bbb (i,v) ,{pi},{B}
where bbb.i={pi}.i and bbb.i={B}.i and {B}.j={o[0]}''')
    return '\n'.join(sql)
```

B.7.3 Viterbi-Algorithmus

Es folgt eine Python3-Funktion, welche einen SQL-Anfrageplan des Viterbi-Verfahrens erstellt. Hierbei wird die Anfrageschachtelung nur für einzelne Variablen genutzt. Eine zeilenübergreifende Komposition, etwa durch rekursive Anfragen, sollte Zustand künftiger Betrachtungen sein. Die Option `stable` ermöglicht hierbei die Umsetzung des numerisch stabileren Verfahrens aus Algorithmus 13. Aufgrund der Logarithmierung dürfen in diesem Fall die HMM-Parameter A, B und π keine 0-Werte besitzen.

```
def viterbi_sql(A='a',B='b',o=[],pi='pi',delta='delta',psi='psi',stable=False,
    logA='loga',logB='logb'):

    m = len(o)

    # sequential approach
    if stable:
        sql=[f'''insert into {delta}
    select {pi}.i,0,log({pi}.v)+{logB}.v
    from {pi} join {logB} on {pi}.i={logB}.i
    where {logB}.j={o[0]};''']
    else:
        sql=[f'''insert into {delta}
    select {pi}.i,0,{pi}.v*{B}.v
    from {pi} join {B} on {pi}.i={B}.i
    where {B}.j={o[0]};''',]

    sql.append(f'''insert into {psi}
    select i,0,0
    from {pi};''')
```

```
    for jm1,oj in enumerate(o[1:]):
        j = jm1 + 1
        if stable:
            sql.append(f'''insert into {delta}
    select dela.j,{j},delA.v+{logB}.v
    from (
        select {logA}.j,max({logA}.v+{delta}.v)
        from {logA} join {delta} on {logA}.i = {delta}.i
        where {delta}.j={jm1}
        group by {logA}.j
    ) delA (j,v) join {logB} on delA.j={logB}.i
    where {logB}.j={oj};
insert into {psi}
    select i,{j},v
    from (
        select maxad.i,min(delA.i)
        from
        (
            select {logA}.i,{logA}.j,{logA}.v+{delta}.v
            from {logA} join {delta} on {logA}.i={delta}.i
            where {delta}.j={jm1}
        ) delA (i,j,v) join(
            select ad.j,max(ad.v)
            from (
                select {logA}.i,{logA}.j,{logA}.v+{delta}.v
                from {logA} join {delta} on {logA}.i={delta}.i
                where {delta}.j={jm1}
            ) ad (i,j,v)
            group by ad.j
        ) maxad (i,v)
        on delA.v=maxad.v and maxad.i=delA.j
        group by maxad.i
    ) temp (i,v);''')
        else:
            sql.append(f'''insert into {delta}
    select dela.j,{j},delA.v*{B}.v
    from (
        select {A}.j,max({A}.v*{delta}.v)
        from {A} join {delta} on {A}.i = {delta}.i
        where {delta}.j={jm1}
        group by {A}.j
    ) delA (j,v) join {B} on delA.j={B}.i
    where {B}.j={oj};
insert into {psi}
    select i,{j},v
    from (
```

```
        select maxad.i,min(delA.i)
        from (
            select {A}.i,{A}.j,{A}.v*{delta}.v
            from {A} join {delta} on {A}.i={delta}.i
            where {delta}.j={jm1}
        ) delA (i,j,v) join (
            select ad.j,max(ad.v)
            from (
                select {A}.i,{A}.j,{A}.v*{delta}.v
                from {A} join {delta} on {A}.i={delta}.i
                where {delta}.j={jm1}
            ) ad (i,j,v)
            group by ad.j
        ) maxad (i,v)
        on delA.v=maxad.v and maxad.i=delA.j
        group by maxad.i
    ) temp (i,v);''')


    sql.append(f'''with recursive q (v,i) as (
    select min(i),(select max(j) from delta)
    from {delta}
    where {delta}.j={m-1} and v= ( select max(v) from {delta} where j=(select
        max(j) from delta) )
    union all
    select min(v),i
    from (
        select {psi}.v,q.i-1
        from {psi},q
        where q.i>0 and {psi}.i= q.v and {psi}.j = q.i
    ) ttt (v,i)
    group by i
)
select v
from q
order by i;''')

    return '\n'.join(sql)
```

B.7.4 Baum-Welch-Verfahren

Abschließend wird ein Python3-Code aufgeführt, welche einen SQL-Anfrageplan für einen Iterationsschritt des Baum-Welch-Verfahrens erstellt. Das verwendete Datenbankschema ist der darauf folgenden Funktion entnehmbar. Zusätzlich ist eine Beispielanwendung umgesetzt, in der

das vollständige Baum-Welch-Verfahren aus dem Wetterbeispiel aus Anhang B.1 realisiert ist. Als Beispieldatenbanksystem wurde PostgreSQL genutzt, welches mittels des `psycopg2`-Pakets angesprochen wird. Hier wird die Datenbank „hmm“ lokal betrieben und die jeweiligen Anfragen als `postgres`-Nutzer gesendet.

```
def db_baum_welch_schritt(o,orel='o',A='a',B='b',pi='pi', Aneu='aneu',Bneu='bneu
    ', pineu='pineu',   alp='alpha',bet='beta',gam='gamma',
    skal_alp='salp',xi='xi', res='bwres', stable=False):


    sql = [f'''delete from {Aneu}; delete from {Bneu}; delete from {pineu};
    delete from {gam}; delete from {xi}; delete from {alp}; delete from {bet};
    delete from {skal_alp};''']

    sql.append(db_forward_verfahren(o, A, B, alp, pi, mode=0, skal_alp=skal_alp,
         stable=stable))
    sql.append(db_backward_verfahren(o, A, B, bet, pi, mode=0))


    if stable:
        sql.append(f'''update {bet}
    set v = {bet}.v / {skal_alp}.v
    from {skal_alp}
    where {skal_alp}.t = {bet}.t;''')


    # gamma
    sql.append(f'''insert into {gam}
    select {alp}.i,{alp}.t,{alp}.v*{bet}.v/denom.v
    from {alp},{bet},(
        select {alp}.t,sum({alp}.v*{bet}.v)
        from {alp} join {bet} on {alp}.i={bet}.i and {alp}.t={bet}.t
        group by {alp}.t
        ) denom(t,v)
    where {alp}.i={bet}.i and {alp}.t={bet}.t and {alp}.t=denom.t;''')

    for t,ot in enumerate(o[:-1]):
        sql.append(f'''insert into xi
        select {t},{A}.i,{A}.j,{A}.v*nona.v
        from {A} join (
            select {alp}.i,bbet.i,{alp}.v*bbet.v / (
                select sum({alp}.v*abbet.v)
                from {alp} join (
                    select {A}.i,sum({A}.v*bbet.v)
                    from {A} join (
                        select {B}.i, {B}.v*{bet}.v
```

```
                    from {B} join {bet} on {B}.i = {bet}.i
                    where {B}.j = {o[t+1]} and {bet}.t = {t+1}
                ) bbet(i,v) on {A}.j = bbet.i
                group by {A}.i
            ) abbet (i,v) on {alp}.i= abbet.i
            where {alp}.t={t}
        )
        from {alp}, (
            select {B}.i, {B}.v*{bet}.v
            from {B} join {bet} on {B}.i = {bet}.i
            where {B}.j = {o[t+1]} and {bet}.t = {t+1}
        ) bbet(i,v)
        where {alp}.t={t}
    ) nona (i,j,v) on {A}.i=nona.i and {A}.j=nona.j; ''')


# update
#pi
sql.append(f'''insert into {pineu}
select i,v from {gam} where t=0;''')

# A
sql.append(f'''insert into {Aneu}
select sumxi.i, sumxi.j, sumxi.v / sumgam.v
from (
    select i,j,sum(v)
    from {xi}
    group by i,j
) sumxi (i,j,v) join (
    select i,sum(v)
    from {gam}
    where t <> (select max(t) from {gam})
    group by i
) sumgam (i,v) on sumxi.i = sumgam.i;''')

# B
sql.append(f'''insert into {Bneu}
select dgam.i,dgam.j,dgam.v/sumgam.v
from (
    select {gam}.i,{orel}.v,sum({gam}.v)
    from {gam}, {orel}
    where {gam}.t = {orel}.t
    group by {gam}.i,{orel}.v
    ) dgam (i,j,v) join
    (
        select i,sum(v)
```

```
            from {gam}
            group by i
        ) sumgam (i,v) on dgam.i = sumgam.i;''')

    # res = | A_neu-A |_F^2 + |B_neu-B|_F^2 + | pineu - pi |_2^2
    sql.append(f'''insert into {res}
    select (select max(i)+1 from {res}) ,(
        select sum(abs({Aneu}.v-{A}.v)^2)
        from {Aneu} join {A}
            on {Aneu}.i={A}.i and {Aneu}.j={A}.j
        ) + (
        select sum(abs({Bneu}.v-{B}.v)^2)
        from {Bneu} join {B}
            on {Bneu}.i={B}.i and {Bneu}.j={B}.j
        ) + (
        select sum(abs({pineu}.v-{pi}.v)^2)
        from {pineu} join {pi} on {pi}.i={pineu}.i
        );''')

    return '\n'.join(sql)

def db_baum_welch_test():

    relnames = {
        "orel" : "o",
        "A":"a",
        "B":"b",
        "pi":"pi",
        "Aneu":"aneu",
        "Bneu":"bneu",
        "pineu":"pineu",
        "alp":"alpha",
        "bet":"beta",
        "gam":"gamma",
        "skal_alp":"salp",
        "xi":"xi",
        "res":"bwres"
    }

    with psycopg2.connect(dbname='hmm',host='localhost', port=5432, user='
        postgres', password=None) as db:
        with db.cursor() as cur:
            sql = []
            for relname in relnames.values():
                sql.append(f'drop table if exists {relname};')
            cur.execute('\n'.join(sql))
```

```
    sql = []
    for relname in ['A','B','Aneu','Bneu']:
        sql.append(f'create table {relnames[relname]} ( i int, j int, v
            double precision );')
    for relname in ['pi','pineu','res']:
    sql.append(f'create table {relnames[relname]} ( i int, v double
        precision );')
        for relname in ['alp','bet','gam']:
    sql.append(f'create table {relnames[relname]} ( i int, t int, v
        double precision );')
    sql.append(f'create table {relnames["orel"]} (t int, v int );')
    sql.append(f'create table {relnames["skal_alp"]} (t int, v double
        precision );')
    sql.append(f'create table {relnames["xi"]} (t int, i int, j int, v
        double precision);')
    cur.execute('\n'.join(sql))
    db.commit()

with db.cursor() as cur:
    sql = [f'''insert into {relnames["A"]} values (1,1,0.6),(1,2,0.3)
        ,(1,3,0.1),(2,1,0.3),(2,2,0.5),(2,3,0.2),
    (3,1,0.2),(3,2,0.4),(3,3,0.4);''']
    sql.append(f'''insert into {relnames["B"]} values (1,1,0.9)
        ,(1,2,0.1),
    (2,1,0.3),(2,2,0.7),(3,1,0.1),(3,2,0.9);''')
    sql.append(f'''insert into {relnames["pi"]} values (1,0.3),(2,0.3)
        ,(3,0.4); ''')
    sql.append(f'''insert into {relnames["orel"]} values (0,1),(1,2);'''
        )
    sql.append(f'''insert into {relnames["res"]} values (0,NULL);''')
    cur.execute('\n'.join(sql))
    db.commit()


# Anfrageerstellung gerader Iterationslauf
sql_gerade = db_baum_welch_schritt(o, stable=False,**relnames)
relnames["Aneu"]="a"
relnames["Bneu"]="b"
relnames["pineu"]="pi"
relnames["A"]="aneu"
relnames["B"]="bneu"
relnames["pi"]="pineu"

# Anfrageerstellung ungerader Iterationslauf
sql_ungerade = db_baum_welch_schritt([1,2], stable=False,**relnames)
```

```
max_rep = 100
eps = 0.001
with db.cursor() as cur:
for i in range(max_rep):
print(f'starte run {i}')

if i%2==1:
    cur.execute(sql_ungerade)
else:
    cur.execute(sql_gerade)
db.commit()

cur.execute(f'select v from {relnames["res"]} where i=(select max(i)
   from {relnames["res"]});')
res = float(cur.fetchall()[0][0])

print(f'Residuum: {res}')
if res < eps:
    break
```

Zusammenfassung

Die kontinuierlichen technischen Fortschritte der vergangenen Jahre haben das enorme Potenzial datengetriebener Anwendungen, wie etwa Assistenzsysteme, aufgezeigt. Aufgrund enormer Datenmengen ist zur Analyse und Verarbeitung dieser oftmals eine massive Parallelisierung nötig. Entgegen dem bestehenden Trend spezifischer Neuentwicklungen werden in parallelen Datenbanksystemen bereits seit mehreren Jahrzehnten transparente parallele Verarbeitungen großer Datenmengen unterstützt und optimiert. Daraus motiviert wird in dieser Arbeit untersucht, inwiefern parallele relationale Datenbanksysteme für Methoden der Aktivitätserkennung und -vorhersage in Assistenzsystemen gewinnbringend eingesetzt werden können. Der Fokus liegt hierbei auf der effizienten und skalierbaren Umsetzung und Komposition von Basisoperatoren der linearen Algebra. Dies ermöglicht neben der Umsetzung zugehöriger Machine-Learning-Verfahren die Einbeziehung zahlreicher weiterer Methoden des wissenschaftlichen Rechnens. Für die potenzielle Umsetzung solcher werden daher zahlreiche Aspekte diskutiert und experimentell ausgewertet. Hierzu zählen unter anderem die Wahl geeigneter Datenbanksysteme, Datenrepräsentationen und zugehörige Übersetzungen in SQL und der Einfluss von Indexstrukturen. Für die effiziente Intraoperatorparallelisierung wird zudem eine Anfragetechnik vorgestellt, die die Verarbeitung einzelner Lineare-Algebra-Operatoren deutlich beschleunigen kann. Zur Einordnung der Güte des datenbankinternen Ansatzes werden Laufzeitvergleiche einzelner Operatoren und ganzer Methoden mit etablierten Lineare-Algebra-Bibliotheken und Big-Data-Ansätzen präsentiert und analysiert.

Continuous technological advances in recent years have highlighted the potential of data-driven applications such as assistance systems. Due to enormous amounts of data, massive parallization is often required in order to process such applications in time. Contrary to the existing trend of specific new developments, transparent parallel processing of large amounts of data has already been supported and optimized in parallel database systems for several decades. Motivated by this, the present thesis investigates to what extent parallel relational database systems can be profitably used for activity recognition and prediction methods in assistance systems. The focus is on efficient and scalable implementations of basic linear algebra operators and their composition. In addition to the realization of associated machine learning methods, this enables the inclusion of numerous other methods of scientific computing. For the potential implementation of such, numerous aspects are therefore discussed and experimentally evaluated. These include the choice of suitable database systems, data representations and associated translations in SQL, as well as the influence of index structures. For efficient intraoperator parallelization, a query technique is presented that can significantly speed up the processing of single linear algebra operators. To prove the usability of the intra-database approach, runtime comparisons of individual operators and entire methods with established linear algebra libraries and big data environments are presented and analyzed.

Lebenslauf

Persönliche Daten

Name:	Dennis Marten
Geburtsdatum:	04.05.1988
Geburtsort:	Berlin

Beruflicher Werdegang

2007	**Abitur** *Erasmus-Gymnasium, Rostock, Deutschland*
10/2007-03/2010	**Bachelor of Science Mathematik** *Universität Rostock Deutschland*
04/2010-09/2012	**Master of Science Mathematik** *Universität Rostock, Deutschland*
10/2012-08/2014	**Berechnungsingenieur** *Windrad Engineering GmbH, Bad Doberan, Deutschland*
09/2014-09/2017	**Stipendiat im Graduiertenkolleg MuSAMA** *Lehrstuhl für Datenbank- und Informationssysteme, Universität Rostock, Deutschland*
09/2017-05/2018	**Stipendiat (Überbrückungs- und Abschlussstipendien für Nachwuchswissenschaftler*innen der Universität Rostock)** *Universität Rostock, Deutschland*
07/2018-11/2019	**Projektmitarbeiter** *Lehrstuhl für Datenbank- und Informationssysteme, Universität Rostock, Deutschland*
seit 04/2020	**Projektassistent und Entwickler** *JAKOTA Cruise Systems GmbH — FleetMon, Rostock, Deutschland*