Entwurf und Implementierung einer relationalen Algebramaschine

Diplomarbeit
Universität Rostock, Fachbereich Informatik

vorgelegt von
Kruse, Hans-Henning
geboren am 06.05.1969 in Rostock

Betreuer: Prof. Dr. P. Forbrig
Dr. H. Meyer, Dipl. Inf. U. Langer

Rostock, den 01.04.1994

Danksagung

Herrn Prof. Dr. P. Forbrig, Dr. H. Meyer, Dipl. Inf. U. Langer danke ich für die aktiven Diskussionen und die sehr gute Unterstützung und Förderung bei der Durchführung dieser Diplomarbeit.

Rostock, 01.04.1994 Hans-Henning Kruse

Zusammenfassung

Die **d**atenflußgesteuerte **R**elationen**a**lgebra-**M**aschine (DRAM) stellt im HE$_A$D Prototypen, einem heterogenen, verteilten Datenbankverwaltungssystem, die Operationen einer erweiterten Relationenalgebra zur Verfügung. Durch die Verwendung geeigneter Algorithmen bei der Realisierung der DRAM-Operationen wird horizontale und vertikale Parallelität unterstützt. Die parallel ausgeführten Prozesse kooperieren miteinander. Die Kommunikation zwischen Prozessen auf unterschiedlichen Rechnern erfolgt durch Message passing. Zur Verringerung der Kosten für die Kommunikation zwischen Prozessen auf einem Rechner wird sogenannter Shared memory verwendet. Ein Tupelprotokoll realisiert die Kommunikation zwischen den DRAM-Prozessen, wobei es den Nutzer von den Unterschieden der Kommunikation zwischen den DRAM-Prozessen auf einem Rechner bzw. auf unterschiedlichen Rechnern befreit. Einen wesentlichen Anteil an den Kosten für die Anfragebearbeitung haben Join-Operationen, deren Kosten durch Parallelisierung verringert werden. Die Parallelisierung des Joins erfolgt durch Partitionierung der zu verbindenden Relationen und durch paralleles Ausführen der Joins über die Partitionen.

Abstract

The **D**ata Driven **R**elational **A**lgebra **M**achine (DRAM) implements the extended relational algebra operations of the heterogeneous distributed database management system HE$_A$D. Horizontal and vertical parallelism is supported by suitable algorithms. The cooperation of concurrent processes plays an important role in query execution. The communication of processes placed on different sites is based on message passing. On the other hand shared memory is used to decrease the communication costs between processes on a local site. Differences of communication between local and remote processes are hidden by a special tuple protocol. Since the cost of a query execution is influenced considerably by join operations, parallel join algorithms are used to decrease the cost. The parallel join is based on partitioning both input relations and joining the partions concurrently.

CR-Klassifikation

C.2.1	Netzkommunikation	Network communications
C.2.2	Netzprotokolle	Network protocols
C.2.4	verteilte Datenbanken (H.2.4)	Distributed databases
D.1.3	Programmierung paralleler Prozesse	Concurrent programming
D.4.7	verteilte Systeme	Distributed systems
H.2.4	Verarbeitung von Anfragen	Query processing
H.2.5	Heterogene Datenbanken	Heterogeneous databases

Key Words: Distributed database system, shared nothing architecture, interprocess communication, threads, distributed query processing, relational algebra, hash join, virtual shared memory

Abkürzungen

ASN.1	Abstract Syntax Notation One
BSP	Basic Stream Protocol (HEAD)
CAD	Computer Aided Design
CPU	Central Processing Unit
DCE	Distributed computing environment
DRAM	Data Driven Relational Algebra Machine
EDV	Elektronische Datenverarbeitung
EM	Execution Monitor
FIFO	Named pipe
FLIP	Fast Local Internet Protocol (AMOEBA)
GQM	Global Query Manager
HEAD	Heterogeneous Extensible and Distributed Database Management System
IDP	Internet Datagram Protocol (XNS)
IP	Internet Protocol (ARPANET)
IPC, ipc	Interprocess communication
ISO	International Standards Organization
LDI	Local Data Interface
LQM	Local Query Manager
LWP	Light weighted process
Mmap, mmap	Memory mapped file
MTU	Maximum Transmission Unit
SPP	Sequenced Packet Protocol (XNS)
SunOS MT	Sun-OS Multi thread architecture
OSF	Open Software Foundation
OSI	Open Systems Interconnection
PC	Personalcomputer
PDU	Protocol Data Unit
PEX	Packet Exchange Protocol (XNS)
QEP	Query Evaluation Plan
RDP	Reliable Data Protocol (ARPANET)

RPC	Remote Procrdure Call
SVR4	UNIX System V Release 4
TB	Tuple Block
TBP	Tuple Block Protocol (HEAD)
TBPP	Tuple Block Presentation Protocol (HEAD)
TCP	Transmission Control Protocol (ARPANET)
TP1-TP4	OSI Transportprotokolle
UDP	User Datagram Protocol (ARPANET)
WS	Workstation
XNS	Xerox Network System

1. Einleitung

HE$_A$D (Heterogeneous Extensible and Distributed Database Management System) [Wiso90, Flac93] ist der Prototyp eines heterogenen verteilten Datenbanksystems. Bei dem gleichnamigen Projekt werden Untersuchungen zur parallelen Anfragebearbeitung in heterogenen verteilten Datenbanksystemen durchgeführt.

Der Vorteil verteilter Systeme [Slom89] gegenüber herkömmlichen Konzepten liegt in:

- der Flexibilität und Erweiterbarkeit
- der hohen Verfügbarkeit und Vollständigkeit
- der Anwendungsnähe und lokalen Autonomie
- der hohen Zuverlässigkeit
- der Anpaßbarkeit der EDV-Struktur an die Organisationsstruktur des Unternehmens.

Eine verteilte Datenbank [Ceri85, Ozsu91] besteht aus mehreren miteinander in Wechselwirkung stehenden Datenbanken, die über ein lokales Computernetzwerk miteinander verbunden sind. Sie werden durch ein verteiltes Datenbankverwaltungssystem organisiert, wobei die Verteilung für den Benutzer transparent ist. Die Transparenz wird durch das konzeptuelle Datenbankschema der verteilten Datenbank realisiert. Es beschreibt eine einzige logische Datenbank, die auf mehrere Rechner verteilt ist.

Basis des HE$_A$D-Prototypen ist eine Shared nothing-Architektur [DeWi92a] (siehe Abb. 1), bei der gewöhnliche Rechnerknoten (mit eigenem Prozessor, flüchtigem und nichtflüchtigem Speicher) über ein lokales Netz miteinander kommunizieren. Die Prozessoren beeinflussen einander nicht. Sie sind lose miteinander gekoppelt. Kommunikation über das Netz erfolgt während der Kooperation zwischen den Prozessoren. Gegenüber eng gekoppelten Architekturen, wie Shared disk- oder Shared memory-Architekturen, belasten lose gekoppelte Architekturen das Computer-Netz beim Extern-Speicherzugriff bzw. Hauptspeicherzugriff nicht. Die Shared nothing-Architektur erlaubt eine große Erweiterbarkeit der Prozessoranzahl.

Die Heterogenität in HE$_A$D bezieht sich auf die Hardware (Sun SPARC 10, Sun SPARC 2, Sun IPX, IBM-kompatible PC), auf die Betriebssysteme (Sun-OS, BSD-UNIX,

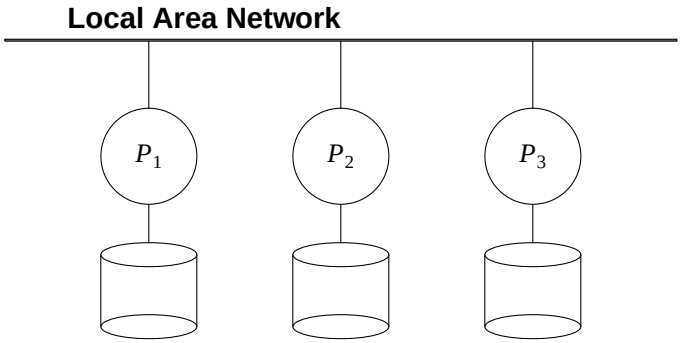


Abb. 1: Die Shared nothing-Architektur

SVR4) und auf die lokalen Datenbankverwaltungssysteme (Ingres, Oracle, Postgres, Sybase).

HEAD wird in relativ eigenständige Funktionseinheiten zerlegt. Aufgrund der unterschiedlichen Leistungsfähigkeit der Rechner in einer heterogenen Umgebung können die Rechner entsprechend ihres Leistungsvermögens in die Anfragebearbeitung einbezogen werden [Flac93]. Infolge der funktionsorientierten Zerlegung des Systems entstehen zwei relativ eigenständige Systemkomponenten:

- zur Anfrageübersetzung und -optimierung und
- zur Abarbeitung der Ausführungspläne.

Die Komponente zur Anfrageübersetzung und -optimierung kann auf allen Rechnerknoten (auch WS und PC) realisiert werden.

Die Komponenten zur Abarbeitung der Ausführungspläne unterteilen sich wiederum in Komponenten:

- zur Abbildung physischer Datenstrukturen auf Mengen von Tupeln (Schnittstelle zur jeweiligen lokalen Datenbank) und
- zur mengenorientierten, datenflußgesteuerten Verarbeitung von Tupeln.

Auf Rechnerknoten, auf denen eine lokale Datenbank gespeichert ist, muß eine lokale Datenbankschnittstelle bereitgestellt werden. Die Anfragebearbeitung ist zu einem wesentlichen Teil nicht an die Rechnerknoten gebunden, auf denen die zu verarbeitenden Daten gespeichert sind. Die Komponente zur mengenorientierten, daten-

flußgesteuerten Verarbeitung wird auf Prozessoren mit den notwendigen Ressourcen zur Verfügung gestellt.

Durch die Parallelisierung der Anfragebearbeitung kommt es zur Leistungssteigerung in verteilten Systemen. Die Parallelisierung erfolgt durch Inter- und Intra-Operatorparallelität. Damit müssen Konzepte für die effiziente Verteilung der Anfrageoperationen auf die verschiedenen Rechnerknoten im Computernetz sowie zur Partitionierung von Relationen bereitgestellt werden. Um eine möglichst kurze Antwortzeit für die Anfragen zu erreichen, werden grundlegende Techniken wie Lastverteilung [Link93] und Pipelining [Mikk88] verwendet.

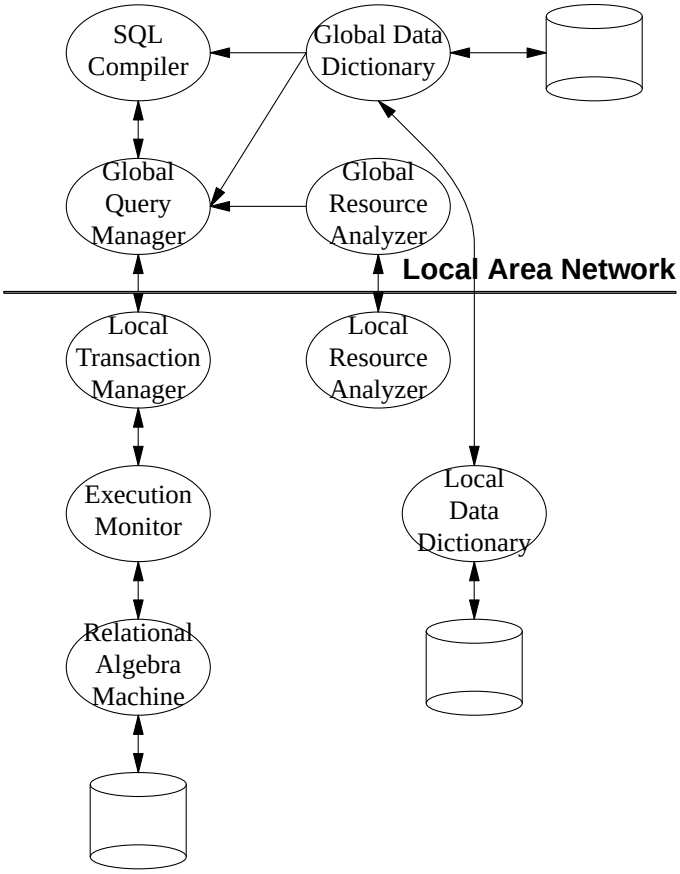


Abb. 2: Struktur des HEAD-Sytems

Die Funktionen zur Anfrageübersetzung werden durch den SQL-Compiler, den Anfrageoptimierer und den Global Query Manager realisiert. Die Funktionen zur Abarbeitung der Ausführungspläne sind der Local Query Manager, der Execution Monitor und

die Data Driven Relational Algebra Machine (datenflußgesteuerte Relationenalgebra-Maschine, DRAM). Local und Global Resource Analyzer realisieren die notwendigen Lastmessungen. Das Global Data Dictionary übernimmt die Metadatenverwaltung auf globaler, das Local Data Dictionary auf lokaler Ebene (siehe Abb. 2).

Der HEAD-Prototyp wird mit der Programmiersprache C++ [Stro91a] realisiert. Die Quelltexte der DRAM-Implementierung sind in [Krus94] dokumentiert.

2. Einordnung der DRAM in den HEAD-Prototypen

Durch die datenflußgesteuerte Relationenalgebra-Maschine werden die Operationen der relationalen Algebra unter Verwendung von Pipelining- und Prozeßinteraktionsmechanismen verwirklicht. Zwischen benachbarten Prozessen werden Nachrichten durch Interprozeßkommunikation (ipc) ausgetauscht. Werden benachbarte Prozesse auf einem Rechnerknoten ausgeführt, erfolgt die Ergebnisübermittlung über gemeinsame Speicherbereiche (Shared memory) [Stev92], ansonsten wird die Nachrichtenübermittlung (Message passing) [Stev92] verwendet.

Relationale Datenbanksysteme basieren auf dem von Codd geschaffenen Relationenmodell [Codd90, Saue91, Lock87]. Eine Relation wird als eine Tabelle dargestellt. Der Tabellenkopf besteht aus beschreibenden Attributen (zeitinvariant). Die folgenden Zeilen bestehen aus Tupeln der Relation (zeitvariante Attributwerte). Für mathematische Klarheit im Datenbereich sorgt die Methode der Normalisierungstechnik, mit deren Umformungsregeln eine normalisierte, redundanzfreie Speicherung der Relationen möglich ist.

Relationale Algebraoperationen können nach der Anzahl der gelesenen und geschriebenen Datenströme (Rang) [Flac93] in einwertige, zweiwertige und mehrwertige Operationen eingeteilt werden. Bei einer Klassifikation entsprechend des Datenflusses gibt es Filter (einwertige Operationen), Multiplexer (ein gelesener und mehrere geschriebene Datenströme) und Demultiplexer (mehrere gelesene Datenströme, ein geschriebener Datenstrom). Die Operationen der erweiterten relationalen Algebra in HEAD werden in der Tabelle 1 klassifiziert. Die Kardinalitäten der Ergebnisrelationen der relationalen Algebraoperationen [Ceri85] sind für die Bewertung der Operationen von Bedeutung, deshalb werden sie in der Tabelle mit angegeben. Die Projektion im HEAD-Prototypen unterscheidet sich von der relationalen Algebraoperation Projektion dadurch, daß sie keine Duplikateleminierung vornimmt. Die Entfernung von Duplikaten kann, wenn sie erforderlich ist, explizit vorgenommen werden. Die Kardinalität der Duplikateleminierung wird anhand der Anzahl unterschiedlicher Werte vom Attribut A_i der Relation R ($val(A[R])$) bestimmt. Die Kardinalitäten der Selektion und der Join-Operation werden durch ihre Selektivität p beeinflußt.

Operation	Parameter	Rang	Datenfluß	Anmerkungen
Scan †	RelName OutStream		Filter	Daten aus dem lokalen Datenbanksystem entnehmen.
Selection	InStream R OutStream T Qual	ein-wertig	Filter	Tabelle horizontal entsprechend der Qualifikation Qual zerschneiden. $card(T) = card(R) * p$
Projection	InStream R OutStream T AttrList	ein-wertig	Filter	Tabelle vertikal entsprechend der Zielattributliste AttrList zerschneiden. in HEAD: card(T) = card(R); sonst: wie Unique
Unique †	InStream R OutStream T	ein-wertig	Filter	explizite Duplikateleminierung $card(T) = val(A_1[R]) * \cdots * val(A_n[R])$
Tee †	InStream R OutStream T	ein-wertig	Filter	Anlegen einer Kopie des Tupelstromes an einem festgelegten Sicherungspunkt. Verdoppelung des Datenbestandes
Sort †	InStream R OutStream T AttrList	ein-wertig	Filter	Sortieren der Tupel nach den Werten der in der Attributliste AttrList angegebenen Attributen.$card(T) = card(R)$
Join	InStream1 R InStream2 S OutStream T Qual	zwei-wertig	Demulti-plexer	Fügt zwei Tabellen entsprechend der Qualifikation Qual zusammen (Verbund). Je nach Selektivität des Joins: $card(T) = card(R) * card(S) * p$
Union	InStream1 R InStream2 S OutStream T	zwei-wertig	Demulti-plexer	Mengenoperation Vereinigung. $card(T) \leq card(R) + card(S)$
Intersection	InStream1 R InStream2 S OutStream T	zwei-wertig	Demulti-plexer	Mengenoperation Durchschnitt. $max(0, card(R) - card(S)) \leq card(T) \leq card(R)$
Minus	InStream1 R InStream2 S OutStream T	zwei-wertig	Demulti-plexer	Mengenoperation Differenz. $card(T) = card(R) - card(S)$
CartProd	InStream1 R InStream2 S OutStream T	zwei-wertig	Demulti-plexer	Mengenoperation kartesisches Produkt. $card(T) = card(R) * card(S)$
Division	InStream1 R InStream2 S OutStream T	zwei-wertig	Demulti-plexer	Division. $card(T) \leq card(R)$
Split † (Hash)	InStream OutStream1 ... OutStreamN	mehr-wertig	Multi-plexer	Partitionierungsoperator.
Merge †	InStream1 ... InStreamN OutStream	mehr-wertig	Demulti-plexer	Faßt mehrere Datenströme zu einem zusammen.

Tab. 1: Operationen der erweiterten Relationenalgebra

† erweiterte Relationenalgebra-Operation

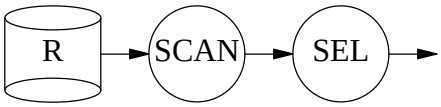
Durch Kombination dieser Operationen ist es möglich, beliebige Auswertungen zu erstellen.

Der Anfragebearbeitungsplan (Query Evaluation Plan, QEP) [Flac93, Haas89] ist die zentrale Ausführungsstruktur zur Steuerung der Anfragen in HE$_A$D. Ein Anfragebearbeitungsplan kann als ein zusammenhängender, zyklenfreier, gerichteter Graph [Läuc91] $G = (K, V)$ dargestellt werden, wobei die Knotenmenge K die Operationen der Relationenalgebra und die Kantenmenge V den Datenfluß repräsentieren. Zwei durch eine Kante verbundene Knoten eines Graphen G heißen benachbart (adjazent).

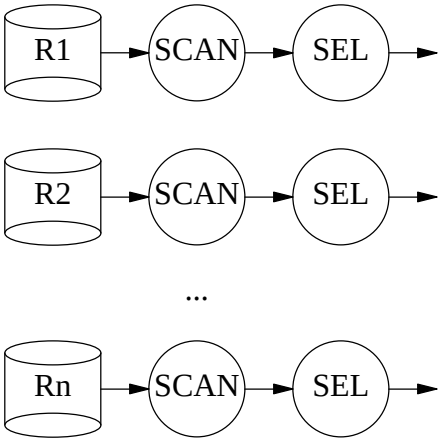
Die Anfragebearbeitungspläne sind ideal für die parallele Ausführung geeignet [DeWi92a]. Einerseits können mehrere Transaktionen unabhängig voneinander und gleichzeitig bearbeitet werden (Inter-Transaktionsparallelität), andererseits können die einzelnen Operationen einer Transaktion (Knoten des Anfragebaums) parallelisiert werden (Intra-Transaktionsparallelität) [Kilg89]. In dieser Arbeit ist besonders die Intra-Transaktionsparallelität von Bedeutung, für die es mehrere Ansatzpunkte gibt:

- Die gleichzeitige Abarbeitung voneinander unabhängiger Teilbäume des Anfragebaums. Falls Ergebnisse eines solchen Teilbaumes mehrmals benötigt werden, wird dieser Teilbaum nur einmal abgearbeitet. Die Ergebnisse werden an die entsprechenden Folgeknoten geschickt [Mikk88].
- Die Partitionierung (siehe Abb. 3 (b) und 4 (b)) der Eingaberelation über mehrere Prozessoren und Speicher. Diese Partitionen können durch voneinander unabhängige Operatoren, von denen jeweils einer auf eine der Partitionen zugreift, gleichzeitig bearbeitet werden (Partitioned parallelism) [DeWi92a].
- Als Eingabe für eine relationale Operation dienen Relationen, d.h. eine uniforme Menge von Tupeln. Als Ausgabe wird wiederum eine Relation erzeugt, die die Eingabe des nachfolgenden Operators bildet (siehe Abb. 3 (a) und 4 (a)). Dadurch können die Berechnungen des nachfolgenden Operators parallel zu denen der vorhergehenden relationalen Operation ausgeführt werden (Pipelined parallelism, Pipelining) [DeWi92a].

Durch Nutzung von Pipelining (Pipelined parallelism) und Partitionierung (Partitioned parallelism) wird die horizontale und vertikale Parallelität - und somit die Leistungsfähigkeit des Datenbankverwaltungssystems - erhöht [Mikk88]. Der Vorteil des



(a) pipelined parallelism



(b) partitioned parallelism

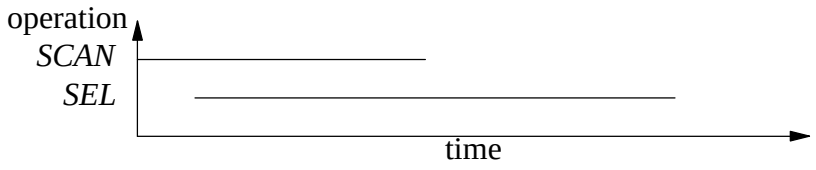
Abb. 3: Parallelität in relationalen Anfragen

Pipelining geht bei Operationen, die ihre Ergebnisse erst berechnen können, nachdem alle Eingaben eingetroffen sind, verloren. Zu diesen Operationen zählen die SQL-Aggregatoperationen wie MIN und MAX und der Sortieroperator SORT.

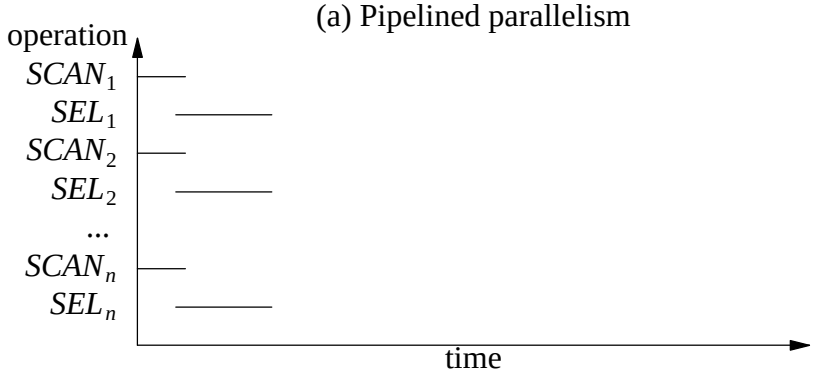
Bei unären Operationen, wie Scan, Projektion und Selection, wird ein Tupel des Eingabestroms bearbeitet, und gegebenenfalls als Ergebnistupel in den Ausgabestrom eingefügt. Hierbei kommt der Vorteil des Pipelinings besonders zum Tragen.

Durch Partitionierung wird ein paralleles Ausführen der Operation durch Operationen über Teile der Gesamtrelation möglich.

Die Vereinigung (Union), der Durchschnitt (Intersection) und die Differenz (Difference), die Division und der Verbund (Join) haben die Aufgabe, zwei Relationen zu vergleichen oder zu kombinieren. Im Folgenden wird vorwiegend der Equi-Join betrachtet, bei dem aus zwei Eingaberelationen eine Ausgaberelation erzeugt wird. Die Tupel werden entsprechend einer Join-Bedingung - beim Equi-Join die Gleichheit zweier Attributwerte jeweils einer Eingaberelation (z.B. R.r1 = S.s2) - zusammengefügt.



(a) Pipelined parallelism



(b) Partitined parallelism

Abb. 4: Verringerung der Verarbeitungszeit durch Partitionierung und Pipelining

2.1. Pipelining (Pipelined parallelism)

Anfragebearbeitungspläne werden durch Datenflußgraphen dargestellt. Die Tupel der Eingabeströme werden bearbeitet und anfallende Ergebnistupel in den Ausgabestrom eingefügt. Der Ausgabestrom dient als Eingabestrom des im Graphen nachfolgenden Operatorknoten. Die Überlappung der Operationen ermöglicht die Verringerung der Antwortzeit (siehe Abb. 4 (a)) gegenüber der sequentiellen Ausführung der Operationen. Für eine optimale Realisierung des Datenflusses werden die Tupel nicht einzeln an die Operatoren übergeben. Für jeden Ausgabestrom wird ein Puffer zur Verfügung gestellt, der eine bestimmte Anzahl von Tupeln aufnehmen kann. Ist dieser Puffer voll, werden die Tupel an den nachfolgenden Operator geschickt und dort verarbeitet.

2.2. Partitionierung (Partitioned parallelism)

Eine Eingaberelation wird in mehrere (disjunkte) Teilrelationen (Partitionen) gespalten. Diese einzelnen Teilrelationen können unabhängig voneinander bearbeitet werden. Somit ist es möglich, eine Operation über Tupel einer Relation als parallele Operationen über Tupel der Teilrelationen auszuführen. Die gleichzeitige Bearbeitung kleiner Teilmengen verringert die Antwortzeit (siehe Abb. 4 (b)). Dafür werden die einzelnen

Partitionen Rechnerknoten im Netz zugewiesen, auf denen sie verarbeitet werden. Durch

- Round robin-Partitionierung,
- Hash-Partitionierung und
- Bereichspartitionierung (Range partitioning)

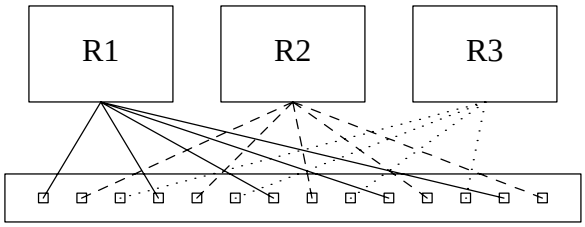
kann die Aufteilung der Gesamtrelation erfolgen (siehe Abb. 5). Die einfachste Partitionierungsstrategie ist die Round robin-Partitionierung, bei der ein ankommendes Tupel immer in die nächstfolgende Partition eingefügt wird. Ist das vorangegangene Tupel in die letzte Partition eingetragen worden, so ist die erste Partition die nächstfolgende.

Bei der Hash-Partitionierung werden die Tupel aufgrund einer Hash-Funktion, die auf jedes Tupel der Relation angewendet wird, den Teilrelationen zugeteilt. Ein Problem der Hash-Partitionierung ist, daß die Teilrelationen bei ungleichmäßiger Verteilung der Attributwerte in ihrer Größe variieren.

Die Bereichspartitionierung, bei der Tupel entsprechend ihrer Attributwerte geclustert werden, geht auf die ungleichmäßige Verteilung (Data skew) der Attributwerte ein. Dazu müssen die Bereichsgrenzen für die einzelnen Cluster so gewählt werden, daß entsprechend der Verteilung der Attributwerte gleichgroße Teilrelationen entstehen [DeWi92b]. Ein Partitionierungsvektor gibt die Bereichsgrenzen für die einzelnen Cluster an. Er wird auf der Grundlage statistischer Informationen über die Verteilung der Attributwerte ermittelt. Ein optimaler Partitionierungsvektor führt zu gleichgroßen Teilrelationen.

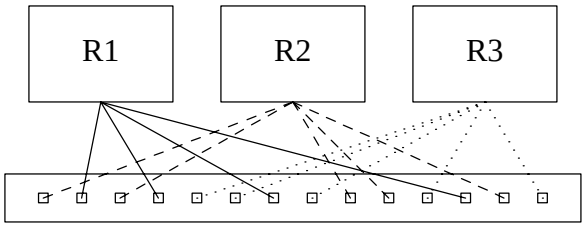
Das Ziel der Partitionierung sind möglichst gleichgroße Partitionen, um ähnlichen Antwortzeiten für jede Partition zu erhalten. Bei Join-Operationen werden die Tupel beider Relationen entsprechend einem Attributwert miteinander verknüpft. Miteinander zu verbindende Tupel der Eingangsrelationen müssen demselben Prozessor zugewiesen werden. Deshalb müssen die Tupel auf der Grundlage einer bestimmten Qualifikation den Partitionen zugeordnet werden.

Die Partitionierung mittels Round robin-Verfahren ist nur für unäre Operationen von Bedeutung. Für den Join-Operator werden die Eingaberelationen nach einer bestimmten Qualifikation in Teilrelationen aufgeteilt. Dadurch kann sichergestellt



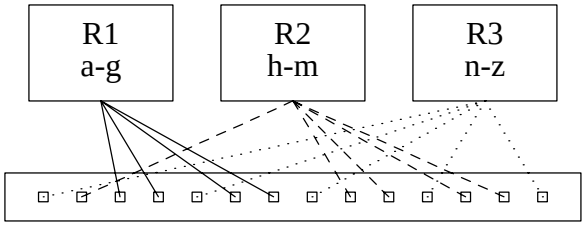
(a) Round robin partitioning

n sei die Anzahl der Partitionen; das *i*-te Tupel wird an die Partition ($i \bmod n$) übergeben.



(b) Hash partitioning

Ein Tupel wird der sich aus der Hash-Funktion ergebenden Partition zugeteilt.



(c) Range partitioning

Die Tupel werden entsprechend dem Bereich, in dem der Wert ihres Attributes liegt, ihrer Partition zugeordnet.

Abb. 5: Partitionierungsstrategien

werden, daß in den korrespondierenden Teilrelationen der beiden Eingaberelationen alle jeweils zu verbindenden Tupel enthalten sind. Die Hash-Partitionierung oder die Bereichspartitionierung nehmen eine derartige Einteilung vor.

Das Round robin-Verfahren erzeugt gleichgroße Teilrelationen. Bei einer gleichmäßigen Verteilung der Attributwerte erreicht man durch die Hash-Partitionierung ebenfalls gleichgroße Partitionen. Sind die Attributwerte jedoch ungünstig verteilt, können gegebenenfalls extrem unterschiedlich große Teilrelationen entstehen.

Wird die Bereichspartitionierung verwendet, kann der ungleichmäßigen Attributwertverteilung mit einem optimalen Partitionierungsvektor entgegengewirkt werden. Bei einer binären Operation (z.B. Join), stellt sich die Frage, von welcher Eingaberelation die Verteilung der Attributwerte betrachtet werden sollte, um den Partitionierungsvektor zu ermitteln. Es bietet sich an, die innere (inner, building) Relation dafür zu verwenden [DeWi92b]. Sie wird beim Hash join† verwendet, um eine Hash-Tabelle aufzubauen. Die Tupel der äußeren (outer, probing) Relation können in der Hash-Tabelle erst nach Tupeln, mit denen sie verbunden werden, suchen, wenn alle Tupel der inneren Relation in der Hash-Tabelle eingetragen sind. Ergebnistupel können erst nachdem vollständigen Empfang der inneren Relation erzeugt werden. Dadurch wird der kontinuierliche Datenfluß unterbrochen. Verarbeiten die parallelen Join-Operationen gleichgroße Partitionen der inneren Relation, wird der kontinuierliche Datenfluß bei allen Join-Operationen für dieselbe Zeit unterbrochen. Eine unbalanciert partitionierte äußere Relation hat keine Auswirkung auf den Datenfluß. Ihre Tupel werden sofort nach ihrem Eintreffen zur Erzeugung von Ergebnistupeln verwendet.

Bei einem besonders häufigen Auftreten eines Attributwertes kann es notwendig werden, Tupel mit diesem Attributwert auf mehrere Partitionen zu verteilen, um gleichgroße Partitionen zu erhalten. Alle Tupel der äußeren Relation, die mit den Tupeln mit gleichem Attributwert korrespondieren, die auf mehrere Partitionen verteilt sind, werden auf alle Rechnerknoten kopiert, auf denen diese Partitionen verbunden werden.

Beispiel: Join der Relationen R und S mit $R.B = S.B$.

† Auf die Join-Verfahren wird weiter unten ausführlich eingegangen.

R(A,B)	S(A,B)
(1,5)	(1,1)
(2,5)	(2,2)
(3,5)	(3,3)
(4,5)	(4,4)
(5,5)	(5,5)
(6,5)	(6,6)
	(7,7)
	(8,8)
	(9,8)

Es soll eine Partitionierung auf zwei Prozessoren $p0$ und $p1$ erfolgen; der Partitionierungsvektor lautet (5).

p0	p1
R1: (1,5) (2,5) (3,5)	R2: (4,5) (5,5) (6,5)
S1: (1,1) (2,2) (3,3) (4,4) (5,5)	S2: (5,5) (6,6) (7,7) (8,8) (9,8)

Die balancierte Aufteilung der Tupel mit mehrfach auftretendem Attributwert auf die Partitionen erfolgt durch die Wichtung (Weighting) [DeWi92b].

> Beispiel: R = { 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 5 } Der Partitionierungsvektor bei einer Partitionierung auf drei Prozessoren wäre (3,3). Um gleichgroße Partitionen zu erhalten, muß eine entsprechende Aufteilung der 8 Tupel mit dem Attributwert 3 auf die verschiedenen Partitionen erfolgen. Das Ergebnis der Partitionierung ist:

p0	p1	p2
R1 = { 1, 2, 3, 3 }	R2 = { 3, 3, 3, 3 }	R3 = { 3, 3, 4, 5 }
25% der Tupel mit dem	50% der Tupel mit dem	25% der Tupel mit dem
Join-Attribut 3	Join-Attribut 3	Join-Attribut 3

Tritt ein Attributwert in beiden Relationen sehr häufig auf, jeder andere Attributwert aber jeweils nur einmal, dann ist es möglich, daß alle Tupel mit diesem Attributwert bei der Bereichspartitionierung einem Prozessor zugeteilt werden. Das führt zu einer starken Belastung dieses Prozessors. Alle anderen Prozessoren werden durch die Ausführung der Operation über ihre Partitionen niedrig belastet. Die virtuelle Ausführungspartitionierung (Virtual process partitioning) [DeWi92b] umgeht dieses Problem, indem eine Partitionierung in bedeutend mehr Partitionen erfolgt, als Prozessoren zur Verfügung stehen. Die einfachste Methode, diese Partitionen den Prozessoren zuzuweisen, wäre die Round robin-Methode, bei der bei k Prozessoren die i-te virtuelle Ausführungspartition zum aktuellen Prozessor ($i \ mod \ k$) zugewiesen wird.

> Beispiel: Jede Relation besteht aus 10000 Tupeln. Der Attributwert 1 tritt in jeder Eingaberelation 1000 mal auf. Der globale Join wird auf zehn Prozessoren ausgeführt. Auf dem Prozessor, der den Join mit dem häufig auftretenden Attributwert ausführt, entstehen 1000000 Ergebnistupel. Erfolgt eine Partitionierung der Eingangsrelationen in 1000 Partitionen, werden je Prozessor 100 Partitionen bearbeitet. Jedes Partition enthält 10 Tupel. Die Tupel mit dem häufig auftretenden Attributwert werden auf 100 Partitionen aufgeteilt, die auf allen Prozessoren bearbeitet werden.

2.3. Parallele relationale Operatoren

Die Parallelität durch Partitionierung wird durch Datenpartitionierung erreicht. Dadurch können parallele Datenströme bei der Ausführung der Anfragebearbeitungspläne verwendet werden. Die Anfragebearbeitungspläne werden gleichzeitig über die Partitionen der Gesamtrelation ausgeführt. Die parallelen Datenströme kommen dadurch zustande, daß die Relationen der verteilten Datenbank über verschiedene externe Speicher im Rechnernetz verteilt sind, so daß der Datenflußgraph auf den entsprechenden Prozessoren, zu denen die externen Speicher gehören, über die entsprechenden Partitionen ausgeführt werden können. Weiterhin können parallele Datenströme durch den Partitionierungsoperator (Split) [DeWi92a] erzeugt werden. Er verwirklicht eine Aufteilung der Tupel eines Datenstromes auf mehrere Datenströme. Diese Herangehensweise erlaubt es, existierende sequentielle relationale Operatoren parallel auszuführen.

Für die Ausführung eines Joins werden zur Verkürzung der Antwortzeit mehrere Join-Operationen über Teilrelationen ausgeführt. Die Partitionen der Eingangsrelation eines Joins werden parallel verarbeitet (siehe Abb. 7). Durch die Partitionierung (Split) werden die Tupel von jeder Partition auf die entsprechenden Join-Ausführungsknoten qualifiziert zugeteilt. Dadurch entstehen für jede Partition Eingaben zu jedem Join-Operator. Der Mischoperator (Merge) faßt diese Teilströme zu einem Eingabestrom zusammen (siehe Abb. 6).

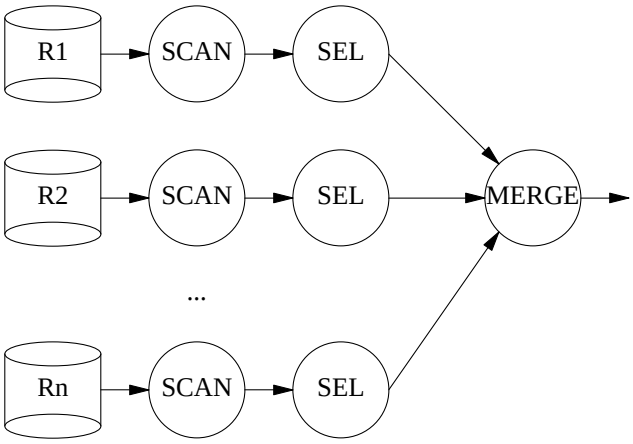


Abb. 6: Der Operator Merge

Der Operator Merge unterstützt Pipelining. Der Operator Split unterstützt Pipelining nicht, wenn statistische Betrachtungen zur Verteilung der Attribute für die Aufteilung der Tupel auf die Teilrelationen erfolgen (z.B. Ermittlung eines Partitionierungsvektors).

2.3.1. Die unären Operationen Selektion und Projektion

Durch die Selektion erfolgt eine Auswahl von Tupeln einer Eingaberelation entsprechend einer Qualifikation für eine Ausgaberelation. Das sequentielle Bearbeiten eines Eingabestroms unterstützt Pipelining. Für ankommende Tupel kann entsprechend der Qualifikation entschieden werden, ob sie in den Ausgabestrom eingefügt werden. Mit der Ankunft des ersten Tupels kann die Erzeugung von Ergebnistupeln beginnen. Ebenso ist die Projektion gut mittels Pipelining realisierbar. Ankommende Tupel werden sofort nach ihrer Bearbeitung in den Ausgabestrom eingefügt.

2.3.2. Die binäre Operation Join

Der Join verbindet zwei Relationen, indem er jedes Tupel der inneren Relation mit jedem Tupel der äußeren Relation verbindet, wenn ihre Join-Attribute in einer gewissen Beziehung zueinander stehen. Beim Equi-Join muß die Gleichheit der Join-Attributwerte erfüllt sein. Parallele Join-Algorithmen [Schn89] beruhen auf:

- Partitionierung, in der Regel Hash-Partitionierung
- unabhängig voneinander über die einzelnen Teilrelationen ausgeführte Join-Berechnungen (lokale Joins).

Der Sort merge join sortiert die beiden Eingangsrelationen nach dem Join-Attribut und verbindet die Tupel, wenn sie die Join-Bedingung erfüllen. Durch paralleles Sortieren der Eingangsrelationen kann eine Optimierung des Sort merge joins erfolgen. Trotzdem bestimmt das kostenintensive Sortieren die Ausführungskosten für den Sort merge join [DeWi92a]. Für das Pipelining bringt der Sort merge join keine Vorteile [Mikk88], da die beiden Eingaberelationen erst komplett und sortiert vorliegen müssen, ehe sie auf Ergebnistupel hin untersucht werden können. Liegen jedoch sortierte Eingaberelationen vor, ist der Sort merge join sehr effektiv, da dann das Sortieren entfällt.

Der Nested block join - bei dem ein ankommender Block einer Relation mit den akkumulierten Blöcken der anderen verbunden werden - unterstützt das Pipelining. Ergebnistupel werden erzeugt, wenn Tupel, die die Join-Bedingung erfüllen, eingetroffen sind. Die Ausführungskosten sind aber viel höher als beim Sort merge join [Mikk88], weil die Join-Berechnung größeren Einfluß auf die Kosten hat als der durch das Pipelining gewonnene Vorteil.

Die beste Leistungsfähigkeit bietet der Hash join [Schn89, DeWi92a, Mikk88]. Gegenüber dem Sort merge join verschlechtert sich die Leistungsfähigkeit beim Hash join mit Anwachsen der Relationen nicht so schnell wie beim Sort merge join [Mikk88].

2.3.3. Parallele Hash join-Verfahren

Hash join-Verfahren können gut parallelisiert werden, da sie auf Partitionierung der Eingangsrelationen beruhen. Dabei werden die Partitionen unabhängig voneinander bearbeitet, was die Voraussetzung für die Bearbeitung der einzelnen Partitionen auf unter-

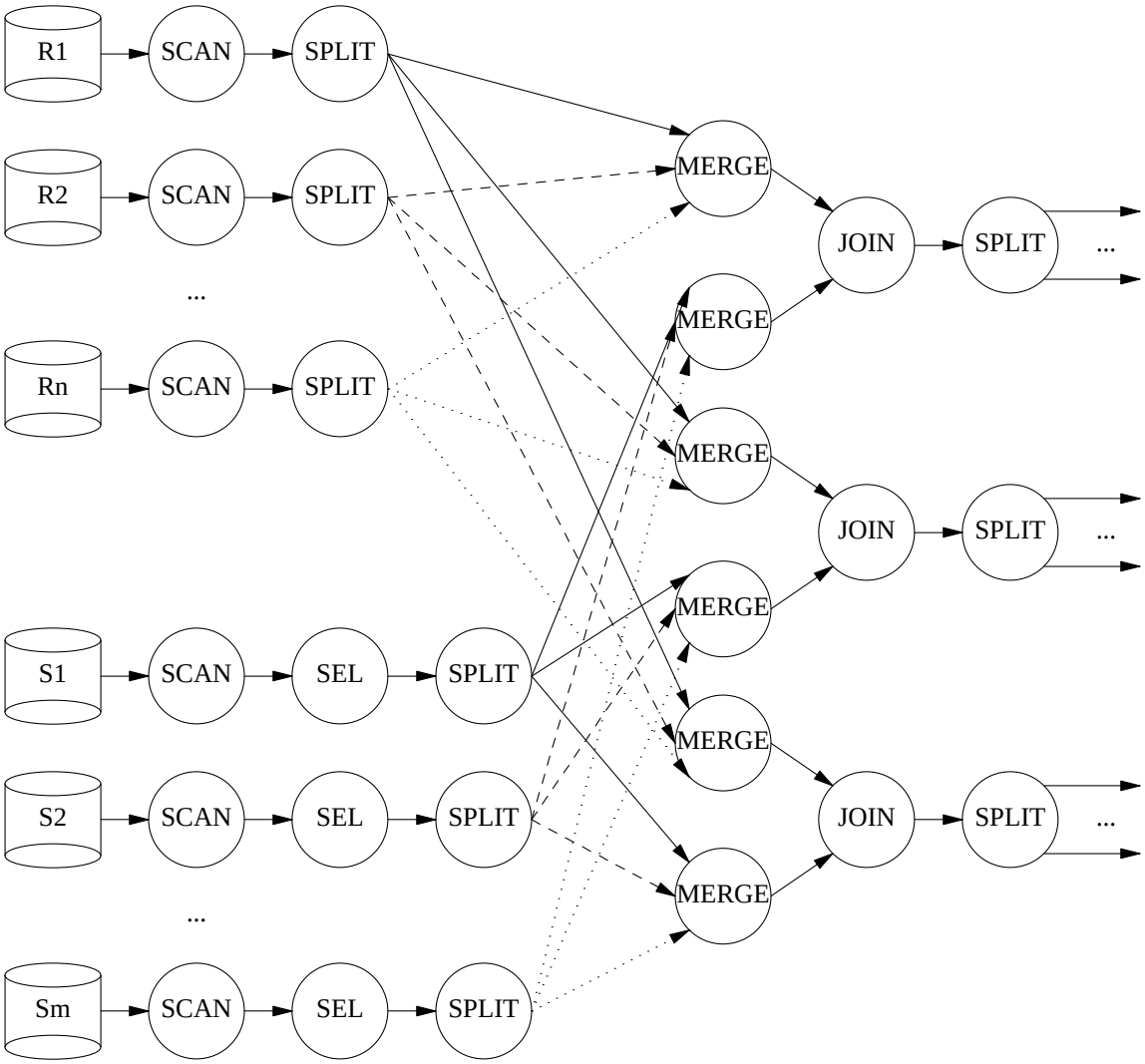


Abb. 7: Join Berechnung durch mehrere Joins über Partitionen der Eingangsrelationen (Partitionierung), die Pipelining unterstützen

schiedlichen Rechnerknoten zur gleichen Zeit ist.

Es gibt verschiedene parallelen Hash join-Verfahren. Sie unterscheiden sich darin, inwieweit Partitionierung und lokaler Join miteinander kombiniert werden [Kilg89]. Der Grund-Algorithmus besteht im wesentlichen aus:

- der Partitionierungsphase:

  Beide Relationen werden nach der gleichen Hash-Funktion h_{part} über die Join-Attribute partitioniert. So entstehen Paare von korrespondierenden Partitionen, die jeweils unabhängig voneinander verbunden werden, um alle Ergebnistupel zu

erzeugen. Diese Partitionen werden an die Rechner übertragen, die den Join ausführen.

- der Aufbau-Phase (Building-Phase):

  Auf jedem Rechner, der einen lokalen Join ausführt, wird aus den Tupeln der Partition der inneren Relation eine Hash-Tabelle im Hauptspeicher aufgebaut. Das ermöglicht einen schnellen Zugriff auf die Tupel der inneren Relation.

- der Test-Phase (Probing-Phase):

  Für die Tupel des korrespondierenden Partition der äußeren Relation werden mittels der Hash-Tabelle aus der Building-Phase die Treffertupel† ermittelt. Die Treffertupel werden miteinander verbunden und in den Ausgabestrom eingefügt.

Der Gesamt-Join wird somit durch mehrere Joins über Partitionen der Eingangsrelationen berechnet (siehe Abb. 7).

2.4. Die Anfrageausführung in HEAD

Der SQL-Compiler transformiert die SQL-Anfrage in einen globalen Anfrageausführungsplan (QEP). Aus dem QEP erzeugt der Globale Query Manager (GQM) parallel ausführbare lokale QEPs. Die lokalen Anfrageausführungspläne werden aufgrund der Lastsituation im gesamten System den Rechnerknoten zugeteilt. Der GQM startet auf diesen Rechnerknoten den Local Query Manager (LQM). Der LQM nimmt die lokalen QEPs vom GQM entgegen und startet den Execution Monitor (EM). Der Execution Monitor aktiviert für jede Operation des lokalen QEP einen eigenen DRAM- oder LDI- (Local Data Interface-) Prozeß. Das LDI dient zum Einlesen von Daten aus den lokalen Datenbanksystemen. Die Daten werden aus dem lokalen Datenformat in das globale Datenformat umgewandelt. Die DRAM stellt die Operationen einer erweiterten relationalen Algebra zur Verfügung. Der Datenfluß zwischen den Operationen wird durch ein Tupel-Protokoll realisiert.

---

† Die Tupel, die die Join-Bedingung erfüllen werden Treffertupel genannt.

Das Starten eines Prozesses für jede Algebraoperation des QEP gestattet:

- die gleichzeitige Ausführung der voneinander unabhängigen Operationen über verschiedene Relationen oder unterschiedliche Partitionen einer Relation (horizontale Parallelität) und
- die Bearbeitung von Ergebnistupeln der Vorgängeroperation, ohne daß deren Ergebnisse komplett vorliegen (vertikale Parallelität).

Durch geeignete Algorithmen, die die Überlappung der DRAM-Operationen erlauben, wird die vertikale Parallelität bei der Bearbeitung der Relationen erreicht. Zu einem Zeitpunkt t_i werden unabhängige Operationen parallel ausgeführt, benachbarte Operationen überlappen einander (siehe Abb. 8).

Jeder Prozeß empfängt vom vorhergehenden Prozeß die Tupelblöcke über Interprozeßkommunikationsmechanismen. Nachdem die Tupel eines Tupelblocks bearbeitet wurden, wird der Ergebnis-Tupelblock an den nachfolgenden Prozeß gesendet [Mikk88].

Für die Tupelübertragung in HEAD ergibt sich dann:

- Es müssen bei der Nachrichtenübertragung Pakete mit einer festen Struktur (Tupelblöcke) übertragen werden.
- Die Einhaltung der Reihenfolge der Tupelblöcke hat keine Bedeutung bei der Berechnung der Ergebnisse, falls die Tupel beim Einlesen eines Tupelstroms nicht sortiert vorliegen müssen.
- Es dürfen keine Tupel verlorengehen oder verfälscht werden.
- Das Transportprotokoll muß wegen der Heterogenität des Systems auf möglichst vielen Plattformen verfügbar sein.

Teilanfragen (lokale QEPs) werden aufgrund der Lastsituation im gesamten System durch den GQM auf verschiedene Rechner aufgeteilt, um eine ausgeglichene Last zu erzeugen. Die Prozesse der DRAM kooperieren miteinander. Der Nachrichtenaustausch zwischen den Prozessen erfolgt durch die Kommunikation. Die Kommunikation über das Rechnernetz führt beim Senden zu Kopiervorgängen in den Sendepuffer und beim

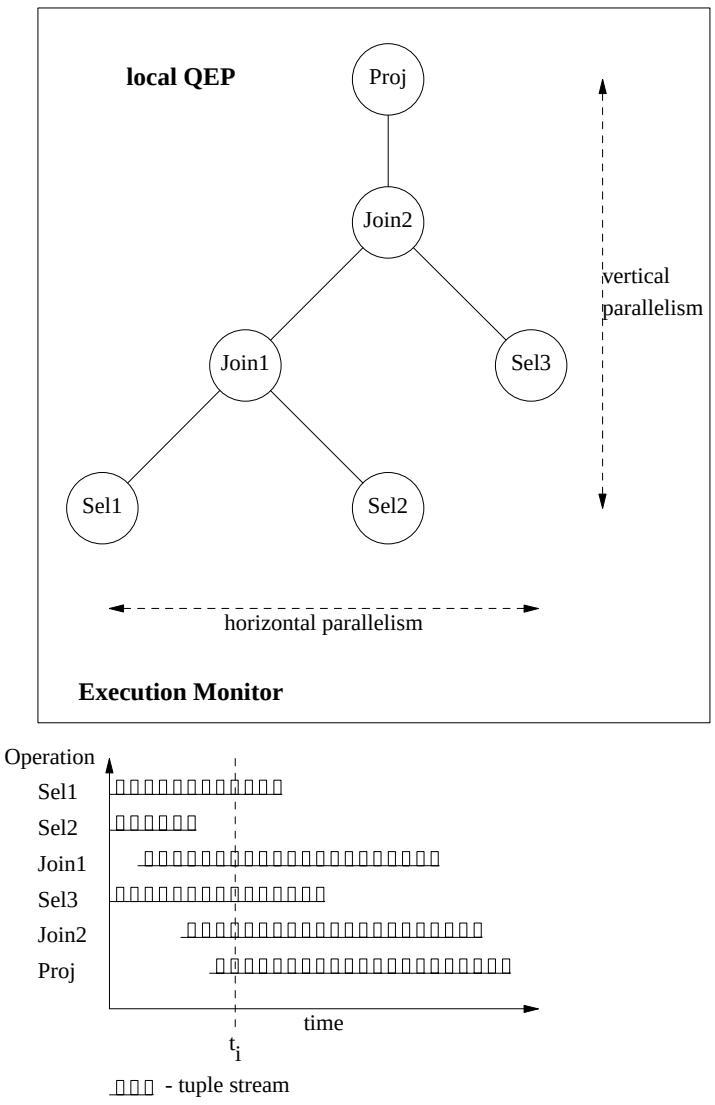


Abb. 8: Tupelstrom zwischen den vom EM parallel gestarteten Prozessen

Empfangen zu Kopiervorgängen aus dem Empfangspuffer. Die Daten werden über das Rechnernetz übertragen. Das Ausführen weniger komplexer Operationen auf einem Rechnerknoten vermeidet diese Kopieroperationen. Werden solche Operationen mit geringer Komplexität durch einen einzigen Prozeß realisiert, können ihre Ausführungskosten zusätzlich gesenkt werden.

Für die interne Pufferung von Zwischenergebnissen wird im Rahmen dieser Arbeit ein virtueller Speicher verwendet, dessen Verwaltung eine Freispeicherverwaltung erfordert.

3.1. Prozeßkonzepte

Wesentliche Merkmale eines konventionellen UNIX-Prozesses [Tane92] sind eine sequentielle Befehlsfolge (Thread), seine eigenen Daten und seine Zustände (bereit, aktiv und blockiert). Diese Prozesse heißen Schwergewichtsprozesse (Heavy weighted processes). Besitzt ein Prozeß mehrere voneinander unabhängig auszuführende Threads, die sich den Adreßraum des Prozesses teilen, handelt es sich bei diesen Threads um Leichtgewichtsprozesse (Light weighted processes).

In vielen verteilten Systemen sind mehrere Threads (Multiple threads of control) [Grau94, Tane92, Powe91] innerhalb eines Prozesses erlaubt. Sie können wie separate Prozesse parallel ausgeführt werden, nutzen aber einen gemeinsamen Adreßraum.

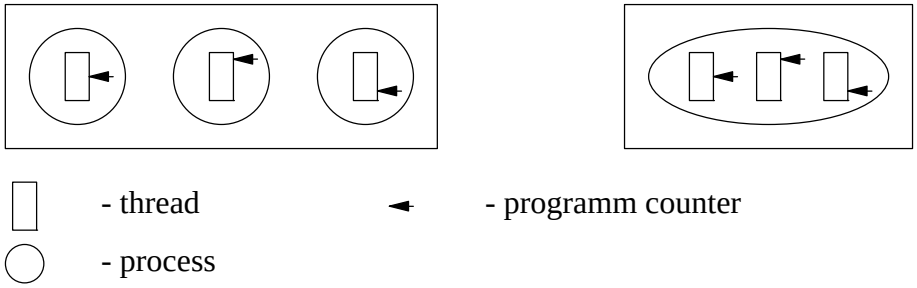


Abb. 9: (a) Drei Prozesse mit einem Thread (b) Ein Prozeß mit drei Threads

In der Abbildung 9 werden Schwergewichtsprozesse Leichtgewichtsprozessen gegenübergestellt. Bei den Prozessen mit jeweils einem Thread (traditioneller UNIX-Prozeß) (siehe Abb. 9 (a)) hat jeder Prozeß seinen eigenen Programmzähler, Stack, Registersatz und Adreßraum. Die einzelnen Prozesse sind voneinander unabhängig, mit der Ausnahme, daß sie in der Lage sind, mit Hilfe der Interprozeßkommunikation zu kooperieren. Gegenüber dem traditionellen UNIX-Prozeß, der nur eine einzelne Befehlsfolge (Thread) besitzt, die von einem Programm abgearbeitet wird, schließt der Multiple thread-Prozeß (siehe Abb. 9 (b)) mehrere Threads ein. Diese Threads können wie Miniprozesse betrachtet werden, die unabhängig voneinander ausgeführt werden. Dabei benutzen sie einen gemeinsamen Adreßraum. Jeder Thread läuft sequentiell ab und besitzt seinen eigenen Programmzähler und Stack. Threads teilen sich die CPU und den Prozeß. Im Gegensatz zu Single-Prozessormaschinen laufen sie in Multiprozessormaschinen wirklich parallel ab. Threads können die gleichen Zustände (aktiv, bereit,

blockiert) wie Prozesse annehmen und vollführen dieselben Zustandsübergänge.

Obwohl Threads in vielen Merkmalen mit Prozessen vergleichbar sind, sind verschiedene Threads in einem Prozeß nicht so unabhängig voneinander wie separate Prozesse. Da alle Threads denselben Adreßraum haben, teilen sie sich auch die globalen Variablen. Es gibt also keinen Schutz zwischen den Threads, der den gleichzeitigen schreibenden und lesenden Zugriff auf jede virtuelle Adresse durch die Threads vermeidet. Der Entwurf von Threads sollte demzufolge möglichst so erfolgen, daß die Threads friedlich miteinander kooperieren und nicht feindlich konkurrieren.

Die Anwendung von verschiedenen Prozessen erfolgt, wie in der Abb. 9 (a), wenn die drei Prozesse unabhängig voneinander sind. Mehrere Threads in einem Prozeß, wie in Abbildung 9 (b) dargestellt, werden genutzt, wenn Threads Teil derselben Aufgabe sind sowie aktiv und eng miteinander kooperieren.

Es existieren verschiedene Anwendungsmodelle für Threads:

- Beim Dispatcher-Modell [Tane92] liest ein Thread, der Dispatcher, ankommende Anforderungen von einer Mailbox und übergibt sie nach ihrer Überprüfung einem unbeschäftigtem Arbeits-Thread (Worker thread) zur Bearbeitung.
- Beim Team-Modell [Tane92] ist jeder Thread gleichberechtigt und erhält seine eigenen Anforderungen direkt vom Auftragserzeuger oder aus einer Auftragsschlange.
- Beim Pipeline-Modell [Tane92] erzeugt der erste Thread aufgrund einer Anforderung Daten, die er an den nächsten Thread übergibt, der sie bearbeitet. Dieser bearbeitet die Daten und gibt sie an den ihn folgenden Thread, der genauso verfährt usw. (siehe Abb. 10).

Das Dispatcher-Modell findet vorrangig bei Servern Anwendung, bei denen die Anforderungen von Clients parallel durch die einzelnen (gleichen) Worker Threads erledigt werden.

Das Team-Modell ist in Verbindung mit einer Auftragsschlange (Job queue) ebenfalls für die Realisierung von Servern geeignet, wobei die unbeschäftigten Threads die Job queue auf Einträge untersuchen und gegebenenfalls die Anforderungen erfüllen.

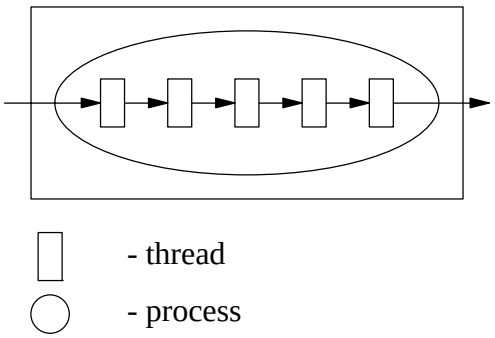


Abb. 10: Das Thread-Pipeline-Modell

Das Pipeline-Modell entspricht durch die nacheinander ausgeführte Manipulation von Daten den Anforderungen für die Ausführung von datenflußgesteuerten relationalen Algebraoperationen. DRAM-Operationen mit geringer Komplexität (Projektion, Selektion oder Scan) können auf einem Rechnerknoten durch einen Prozeß mit mehreren Threads ausgeführt werden. Der Vorteil durch die Verwendung des Pipeline-Modells ist die Unterstützung des Pipelining während der Anfragebearbeitung.

Bei der Kommunikation zwischen den Threads finden keine Kopieroperationen statt, weil die Tupel in dem gemeinsam genutzten Adreßraum untergebracht werden.

Es existieren verschiedene Thread-Implementierungen, sie unterscheiden sich in der Verwaltung der Threads [Tane92]:

- Verwaltung im Nutzerbereich (user space), bei der ein zusätzliches Laufzeitsystem das Umschalten zwischen den einzelnen Threads ausführt.
- Verwaltung im Betriebssystemkern (Kernel).
- Verwaltung von Threads im Nutzerbereich durch ein Laufzeitsystem, das Teilaufgaben der Thread-Verwaltung ohne Einbeziehung des Kernels erledigt. Kernel-Routinen werden nur bei Bedarf ausgeführt (Mischform).

Im Vergleich zu Kernel-Routinen liegt der Vorteil bei der Implementierung von Thread-Paketen im Nutzerraum in der Bereitstellung von Threads in Systemen, die keine Threads unterstützen (z.B. UNIX). Die Realisierung der Zustandsübergänge durch Thread-Pakete im Nutzerraum erfolgt schneller als durch Kernel-Routinen [Powe91]. Nachteilig ist jedoch, daß beim Aufruf einer blockierenden System-Routine, die

Blockierung des gesamten Prozesses erfolgt.

Bei einer Verwaltung von Threads im Kernel bezieht sich das Blockieren nur auf den Thread, der die blockierende Systemroutine aufruft. Ein weiterer Vorteil einer Thread-Verwaltung im Kernel ist die CPU-Zuteilung (Scheduling) durch das System.

Bei einem Thread-Laufzeitsystem ist die CPU-Zuteilung von der freiwilligen Freigabe durch einen aktiven Thread abhängig. Die Nutzung eines Laufzeitsystems, das nur bei Bedarf auf die vom Systemkern zur Verfügung gestellten Funktionen zugreift, vereinigt die Vorteile der ersten beiden Varianten.

Die Tabelle 2 vergleicht die Thread-Implementierungen im Betriebssystem AMOEBA [Tane92], die C-Threads von Mach [Tane92], Sun-OS Multi thread-Architektur (MT) [Powe91], Sun's Light weighted processes (LWP) [Powe91] und die Threads aus dem OSF/DCE (Distributed computing environment) [Ster92]. Der Posix-Standard P1003.4a beschreibt die Thread-Schnittstellen und orientiert sich im wesentlichen an den C-Threads.

	Thread unterstützung durch Kernel erforderlich?	**Verwaltung von Threads durch**	**Threads auf verschiedenen CPU's lauffähig?**	**Verfügbarkeit**
AMOEBA	ja	Kernel	nein	AMOEBA
Mach C-Threads Posix P1003.4a	nein	Laufzeitsystem oder Mischform	ja	viele verschiedene Systeme
OSF/DCE	nein	Laufzeitsystem	nein	verschiedene Systeme
SunOS MT	ja	Mischform	ja	SunOS
Sun LWP	nein	Laufzeitsystem	nein	SunOS

Tab. 2: Einordnung der verschiedenen Thread-Implementierungen

Für die relationalen Algebraoperationen sind vor allem effektive Thread-Konzepte von Bedeutung, bei denen Kernel-Routinen nur bei Bedarf von einem Laufzeitsystem verwendet werden, das die Threads verwaltet.

Ein Vorteil der C-Threads ist, daß sie auf vielen Systemen verfügbar sind. C-Threads stehen in Systemen ohne Thread-Unterstützung durch den Kernel (z.B. UNIX) zur Verfügung, wobei sie durch ein Laufzeitsystem verwaltet werden. Existieren Kernel-Routinen zur Unterstützung von Threads, verwendet das Laufzeitsystem diese bei Bedarf. Damit sind C-Threads für die Systementwicklung in einer heterogenen Umgebung am besten geeignet.

3.2. Das Tupelprotokoll

3.2.1. Prozeßinteraktion

In einem verteilten System werden die einzelnen Prozesse eines komplexen Programms gleichzeitig ausgeführt. Damit diese Komponenten (Prozesse) miteinander kooperieren können, müssen sie untereinander kommunizieren [Slom89, Zeda90]. Die Kommunikation wird durch Synchronisation koordiniert. Sie bewirkt die Einflußnahme eines Prozesses auf einen anderen.

Die Kommunikation kooperierender Prozesse

Die Kommunikation zwischen kooperierenden Prozessen gewährleistet den Austausch von Informationen, der nicht unbedingt synchronisiert erfolgen muß. Die Interprozeßkommunikation basiert auf der Nutzung gemeinsamer Variablen (Shared variables) oder auf Nachrichtenübermittlung (Message passing).

Es werden unidirektionale und bidirektionale Nachrichtenübermittlungen unterschieden.

Bei der unidirektionalen Nachrichtenübermittlung besteht eine Nachrichtentransaktion aus der Übertragung einer Nachricht von der Quelle zum Ziel (bei 1:n-Transaktion zu mehreren Zielen). Die Ausgabe des einen Prozesses wird als Eingabe des anderen Prozesses verwendet. Das Pipe-Konzept stellt eine solche unidirektionale Nachrichtenübermittlung dar.

Eine bidirektionale Nachrichtenübermittlung ist das Client-Server-Konzept. Der Auftragserzeuger (Client) wendet sich mit einem Auftrag an den Diensterbringer (Server), der ihn abarbeitet und das Ergebnis an den Client zurückschickt. Eine solche bidirektionale Nachrichtenübermittlung ist der Remote Procedure Call (RPC). Beim RPC gibt

es einen Serverprozeß, der von einem Clientprozeß Nachrichten erwartet, die ihn zur Ausführung seiner Aufgabe veranlassen. Die Ergebnisse des Serverprozesses werden an den wartenden Client-Prozeß zurückgeschickt. Server- und Clientprozeß können dabei auf demselben oder auf unterschiedlichen Rechnerknoten ausgeführt werden.

Die Synchronisation der Kommunikation

Die Synchronisation stellt eine bestimmte Reihenfolge für die Aktionen mit gemeinsam genutzten Ressourcen wie gemeinsamen Variablen (Shared variables) oder Sende-/Empfangspuffern sicher. Sie bewirkt die Unterbrechung eines Prozesses, bis die Regeln für die Einhaltung der Reihenfolge der Aktionen erfüllt sind.

Bei gemeinsamen Variablen (Shared variables) als Kommunikationsmittel kommen gegenseitiger Ausschluß und Zustandssynchronisation in Betracht:

Gegenseitiger Ausschluß:

> Eine Befehlsfolge, die eine unteilbare Operation ausführt, heißt kritischer Abschnitt. Das exklusive Ausführen eines kritischen Abschnitts durch einen Prozeß bedeutet gegenseitiger Ausschluß.

Zustandssynchronisation:

> Beabsichtigt ein Prozeß den Zugriff auf ein gemeinsames Datenobjekt, daß zur selben Zeit von einem anderen Prozeß manipuliert wird, sollte der entsprechende Prozeß solange unterbrochen werden, bis sich der Zustand dieses Datenobjektes infolge des Ergebnisses des anderen Prozesses geändert hat (Zustandssynchronisation). Eine derartige Synchronisation kann beispielsweise mit Semaphoren implementiert werden.

Die Nachrichtenübermittlung [Slom89] kann als Kommunikations- und Synchronisationsmechanismus betrachtet werden. Die Kommunikation besteht im Empfang von Nachrichten, die ein anderer Prozeß gesendet hat. Die Synchronisation wird dadurch erreicht, daß die Reihenfolge für das Senden und Empfangen einer Nachricht festgelegt ist, indem Nachrichten nur empfangen werden können, wenn sie zuvor gesendet wurden.

Die Nachrichtenübertragung erfolgt asynchron oder synchron.

Beim asynchronen (nicht blockierenden) Senden [Slom89, Zeda90, Tane81] setzt nach dem Abschicken der Nachricht die Komponente ihre Ausführung fort. Damit wird die ausführende Komponente nicht verzögert, wodurch der Grad der Parallelität erhöht wird. Das asynchrone Senden (siehe Abb. 11) kann einen Pufferüberlauf bewirken, was das Blockieren des Senders oder eine Meldung "Puffer voll" an den Sender erfordert, damit die Nachrichten verlustfrei übertragen werden können. Bei einem leeren Puffer ist das Warten des Empfängers oder eine Meldung "Puffer leer" an den Empfänger notwendig. Beim No wait send-Konzept sendet der Sender unblockiert, solange der Puffer nicht voll ist. Der Empfänger wird blockiert, solange der Puffer leer ist.

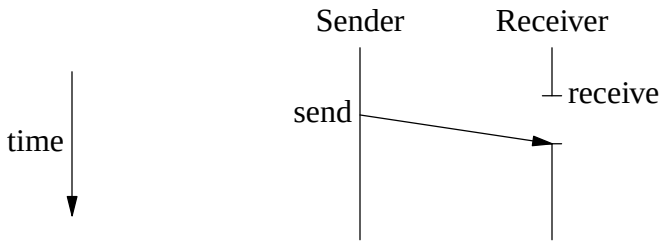


Abb. 11: Das No wait send-Konzept

Beim synchronen (blockierenden) Senden [Slom89, Zeda90, Tane81] (siehe Abb. 12 und 13) wartet der Quellprozeß, bis der Zielprozeß die Nachricht entgegennimmt. Der Zielprozeß wird unterbrochen, bis der Quellprozeß die Nachricht gesendet hat. Damit sind der Sender und der Empfänger synchronisiert.

Vom Rendezvous-Konzept spricht man, wenn keine Antwortnachricht des Senders erwartet wird, sondern eine Empfangsbestätigung (Acknowledgement).

Wird vom Sender auf eine Rückantwort gewartet, handelt es sich um einen Fernaufruf (RPC).

3.2.2. IPC zwischen Prozessen auf unterschiedlichen Rechnerknoten

Im OSI 7-Schichtenmodell [Kern92, Tane81, Slom89], dem ISO-Referenzmodell für Open Systems Interconnection, bewirken die Kommunikationsprotokolle einen virtuellen Datenfluß zwischen den Partnerinstanzen, also den Instanzen ein und derselben

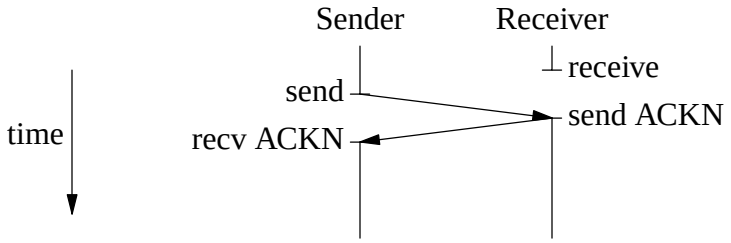


Abb. 12: Das Rendezvous-Konzept

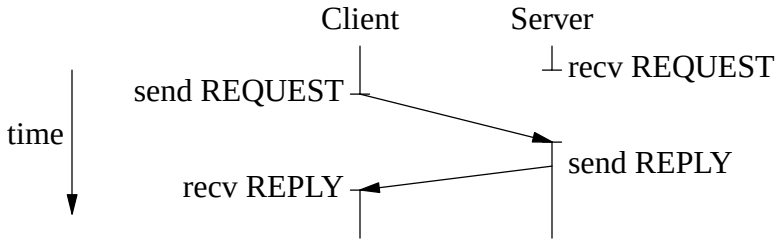


Abb. 13: Das RPC-Konzept

Schicht. Das Modell nimmt eine Trennung in Transportdienstleistender (Schichten 1 - 4) und Transportdienstnutzer (Schichten 5 - 7) vor. Die Transportschicht bildet den Kern der gesamten Protokollhierarchie. Ihr Ziel ist der zuverlässige und kosteneffektive Datentransport vom Quellrechner zum Zielrechner. Dabei wird von tatsächlich vorhandenen Netzwerken abstrahiert. Die Transportschicht schirmt die Transportdienstnutzer von den technischen Gegebenheiten der Datenübertragung ab, so daß die oberen Schichten (5. - 7. Schicht) von den Schwierigkeiten der verschiedenen Subnet-Schnittstellen und der unzuverlässigen Übertragung befreit sind [Tane81].

Die Klassifikation von Übertragungsprotokollen [Stev92] erfolgt nach:

- Verbindungsart des Netzwerkdienstes - in verbindungsunabhängig oder verbindungsorientiert,
- Überwachung der Reihenfolge der empfangenen Nachrichten (Sequencing) - in sequentiell oder nicht sequentiell,
- Vorhandensein einer Fehlerüberwachung - Verhinderung von Datenverfälschung und Datenverlust,

- Sicherstellung, daß der Sender Daten nicht zu schnell an den Empfänger schickt,
- Übertragung in Form eines Byte-Stromes (Stream) oder nachrichtenorientiert (Datagram service).

Durch die Transportschicht des OSI-7-Schichten-Modells werden fünf verschiedene verbindungsorientierte Transportprotokolle (TP0 bis TP4) [Kern92] bereitgestellt, die unterschiedliche Zuverlässigkeit der Netzwerke voraussetzen. Es wird von drei verschiedenen Diensttypen, die die Vermittlungsschicht bereitstellt, ausgegangen:

- Typ A: zuverlässiger fehlerfreier Dienst,
- Typ B: zuverlässiger Dienst, bei dem kaum Fehler auftreten, die jedoch erkannt werden und
- Typ C: unzuverlässiger Dienst.

Weiterhin wird durch die Transportschicht ein verbindungsunabhängiges Transportprotokoll bereitgestellt, bei dem Datenverlust auftreten kann, und die Reihenfolge der Datenpakete beim Empfang mit der Reihenfolge beim Versenden nicht übereinstimmen muß. Die meisten OSI-Protokolle werden durch das ISODE-Software-Paket [Stev92] zur Verfügung gestellt.

Neben dem OSI-Modell haben sich in kommerziellen Systemen die Transportprotokolle des ARPANETs, das verbindungsorientierte TCP (Transmission Control Protocol) [Post81a] und das verbindungsunabhängige UDP (User Datagram Protocol) [Post80] durchgesetzt.

TCP und UDP basieren auf dem verbindungsunabhängigen Vermittlungsprotokoll IP (Internet Protocol) [Post81b].

TCP ist zwar zuverlässig, aber wegen der Übertragung von sequentiellen Byte-Strömen sehr langsam und kostenintensiv. UDP stellt einen schnellen Datagram-Dienst zur Verfügung. Dadurch wird gegenüber der Nachrichtenübertragung auf der Basis von TCP die nachrichtenorientierte Übertragung auf der Basis von UDP effektiver. Der Nachteil von UDP ist seine Unzuverlässigkeit. Beim ARPANET werden für spezifische Anforderungen unterschiedliche Transportprotokolle bereitgestellt. Das RDP (Reliable Data Protocol) [Velt84] ist ein solches Protokoll. Es wurde für die zuverlässige, verbindungsorientierte, nachrichtenorientierte Übertragung entwickelt. RDP ist weniger

komplex als TCP, dafür aber effizienter.

Das XNS (Xerox Network System) [Slom89] stellt das unzuverlässige verbindungsunabhängige nachrichtenorientierte Transportprotokoll PEX (Packet Exchange Protocol) und das zuverlässige verbindungsorientierte Transportprotokoll SPP (Sequenced Packet Protocol) bereit. Das SPP verfügt über hierarchisch angeordnete Schnittstellen für:

- byteorientierte Datenströme,
- paketorientierte Datenströme und
- zuverlässige Datagramm-Übertragung ohne Garantie für die Reihenfolgeerhaltung.

Das Vermittlungsprotokoll im XNS heißt IDP (Internet Datagram Protocol).

Das Vermittlungsprotokoll in AMOEBA [Tane92] heißt FLIP (Fast Local Internet Protocol) [Kaas9?]. Es ist verbindungsunabhängig, nachrichtenorientiert, unsicher und nicht sequentiell. FLIP wurde für die AMOEBA Sitzungsdienste RPC und Gruppenkommunikation entworfen, um auch Prozesse in einem verteilten System übertragen zu können. Die OSI-Protokolle und TCP/IP sind wegen der Prozeßidentifikation über Maschinenadressen für solch eine Anwendung sehr ungeeignet.

Die Tabelle 3 vergleicht die unterschiedlichen Vermittlungs- und Transportprotokolle hinsichtlich ihrer Erfüllung der von HEAD gestellten Anforderungen.

Die Anforderungen eines sicheren, nachrichtenorientierten Protokolls erfüllt nur RDP. SPP bietet diese Möglichkeit optional. Da RDP gegenwärtig in der HEAD-Umgebung nicht verfügbar ist, fällt die Wahl von den zur Verfügung stehenden Transportprotokollen für die Implementierung des HEAD-Tupelprotokolls auf UDP. UDP erfüllt die Ansprüche von den verfügbaren Transportprotokollen am ehesten. Es ist weitestgehend verfügbar und unterstützt die Übertragung von Datagrammen. Es muß aber gewährleistet werden, daß eine zuverlässige Tupelübertragung erfolgt, denn UDP stellt keine verlorengegangenen Pakete fest und gewährleistet nicht, daß der Sender Daten schneller verschickt, als sie vom Empfänger entgegengenommen werden können [Stev92]. TCP erfüllt im Gegensatz zum UDP die Zuverlässigkeit der Übertragung, unterstützt aber keinen Datagramm-Dienst. Durch den zuverlässigen, verbin-

Protokoll	*OSI-Netzwerktyp*	*zuverlässig*	*sequentiell*	*nicht sequentiell*	*Bytestrom*	*Datagramm*	*Verfügbarkeit*
IP	C	nein	nein	ja	nein	ja	Standard in
UDP	C	nein	nein	ja	nein	ja	SVR4, 4.3BSD
TCP	C	ja	ja	nein	ja	nein	u.a.
RDP	C	ja	optional	ja	ja	ja	optional
IDP	C	nein	nein	ja	nein	ja	optional in
PEX	C	nein	nein	ja	nein	ja	SVR4, 4.3BSD
SPP	C	ja	ja	ja	ja	ja	u.a.
TP0	A	ja	ja	nein	ja	nein	optional
TP1	B	ja	ja	nein	ja	nein	durch
TP2	A	ja	ja	nein	ja	nein	ISODE-
TP3	B	ja	ja	nein	ja	nein	Paket
TP4	C	ja	ja	nein	ja	nein	
∗	C	nein	nein	ja	nein	ja	
FLIP	C	nein	nein	ja	nein	ja	AMOEBA optional in SVR4, 4.3BSD

Tab. 3: Gegenüberstellung der Übertragungsprotokolle

dungsorientierten, sequentiellen Byte-Strom entstehen für die Übertragung von Nachrichten mit TCP im Vergleich zur Übertragung mit UDP höhere Kosten, die sich vor allem aus der Reihenfolgeeinhaltung der zu übertragenden Bytes ergeben [Stev92].

3.2.3. IPC zwischen Prozessen auf einem Rechnerknoten

Bei der lokalen Interprozeßkommunikation erfolgt der Datenfluß nicht über das Rechnernetz, sondern über den Betriebssystemkern (Kernel). Deshalb ist der Datenaustausch zwischen Prozessen auf einem Rechnerknoten nicht anfällig gegenüber Reihenfolgeveränderungen oder Datenverlust. Lokale IPC-Mechanismen werden klassifiziert nach:

- der Richtung der Kommunikation - unidirektional oder bidirektional,
- Übertragung von Byte-Strömen oder nachrichtenorientierter Übertragung,

∗ verbindungsunabhängiges OSI-Transportprotokoll

- der Notwendigkeit des Kopierens in einen Kernel-Puffer durch einen Sendeprozeß und aus einem Kernel-Puffer heraus durch einen Empfangsprozeß,
- Kommunikation zwischen beliebigen Prozessen auf einem Rechner oder zwischen Vater- und Sohnprozessen und
- ob Sender und Empfänger während der gesamten Kommunikation existieren müssen.

Die Tabelle 4 vergleicht die lokalen IPC-Mechanismen Pipe, FIFO (Named pipe), Nachrichtenwarteschlange (Message queue) und von mehreren Prozessen gemeinsam nutzbarer Speicher (Shared memory) [Roch88, Stev92].

Kriterium	*Pipe*	*FIFO*	*Message Queue*	*Shared Memory*
Richtung	unidi-rektional	unidi-rektional	bidi-rektional	bidi-rektional
Byte-Strom	ja	ja	nein	nein
nachrichten-orientiert	nein	nein	ja	ja
Kopiervor-gänge	ja	ja	ja	nein
Verwandtschaft der Prozesse	Vater-Sohn	keine	keine	keine
Existenz der IPC-Partner während gesamter IPC	ja	ja	nein	ja
Verfügbarkeit	4.3BSD SVR4	SVR4	SVR4	SVR4 4.3BSD †

Tab. 4: Gegenüberstellung der lokalen Kommunikationsmechanismen

Zwar unterstützen auch unidirektionale Verbindungen das von HEAD geforderte Pipelining, aber Pipe und FIFO übertragen Byte-Ströme und bewirken Kopiervorgänge. Die nachrichtenorientierte Übertragung der Message queue benötigt ebenfalls Kopiervorgänge, Shared memory als Kommunikationsmechanismus dagegen vermeidet

† virtueller Shared memory-Bereich: Memory mapped file (Mmap)

sie, indem sich die kooperierenden Prozesse Speichersegmente teilen, in denen die gemeinsam genutzten Daten untergebracht werden. Der Zugriff auf diese Speichersegmente muß vom Anwender selbst verwaltet werden. Diese Verwaltung kann zum einen durch gegenseitigen Ausschluß oder durch Zustandssynchronisation erfolgen.

Als gemeinsam nutzbarer Speicherbereich stehen entweder Shared memory des System V oder Berkeley UNIX Memory mapped file (mmap) zur Verfügung. Ein wesentliches Argument für die Verwendung von Memory mapped file ist, daß es sich um virtuellen Shared memory handelt. Damit kann es auch für die Pufferung von Zwischenergebnissen verwendet werden. Ein weiterer Grund für die Verwendung von Memory mapped files ist ihre Verfügbarkeit sowohl in BSD-basierten UNIX-Versionen als auch in modernen System V Versionen.

Für die Kommunikation zwischen Leichtgewichtsprozessen ist kein spezieller IPC-Mechanismus notwendig, weil die Threads ihren Adreßraum und somit auch ihre Daten teilen.

3.2.4. Möglichkeiten der Prozeßinteraktionen

Die möglichen Prozeßinteraktionen in einem verteilten System zeigt die Abbildung 14. Für die Kommunikation zwischen Threads innerhalb eines Prozesses kommt als Kommunikationsmedium der lokale Adreßraum zur Anwendung. Der Zugriff auf die im lokalen Speicher des Prozesses befindlichen Daten wird durch gegenseitigen Ausschluß oder mit Hilfe von Zustandsvariablen synchronisiert. Bei der Kooperation zwischen Threads in unterschiedlichen Prozessen auf einem Rechnerknoten wird Shared memory als Kommunikationsmedium angewendet, wobei die Koordinierung des Zugriffs auf die im Shared memory befindlichen Daten durch gegenseitigen Ausschluß bzw. Zustandssynchronisation erfolgt. Weitere Kommunikationsmöglichkeiten zwischen Prozessen auf einem Rechnerknoten sind Pipes, FIFO’s oder Message queues, bei denen der Nachrichtenaustausch über den Kernel erfolgt. Werden die Prozesse auf unterschiedlichen Rechnerknoten ausgeführt, erfolgt die Interprozeßkommunikation via Nachrichtenübermittlung (Message passing).

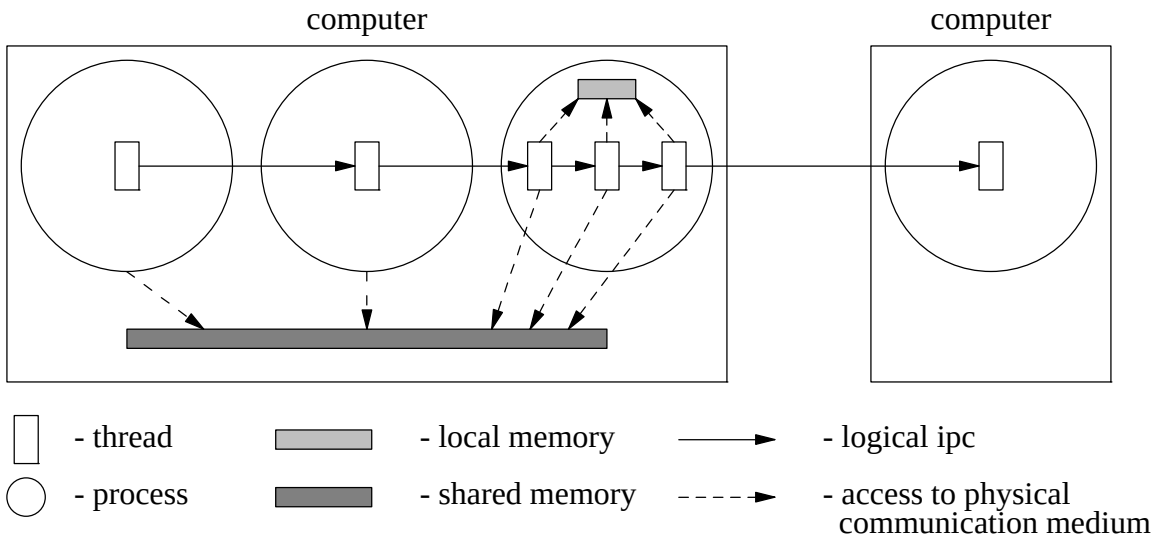


Abb. 14: Prozeßinteraktionen zwischen Threads

3.3. Die Freispeicherverwaltung im virtuellen Shared memory

Die Verwendung eines virtuellen, von mehreren Prozessen nutzbaren Speicherbereiches für die Speicherung von Zwischenergebnissen und für die Interprozeßkommunikation macht die Realisierung einer Freispeicherverwaltung für diesen Bereich erforderlich. Das Berkeley UNIX Memory mapped file stellt einen virtuellen Shared memory bereit.

Das virtuelle Speicherkonzept [Schi78, Tane90] wird charakterisiert durch:

- Adressierung in einem logischen - vom physikalischen Adreßraum unabhängigen - geordneten Adreßraum und
- eine Adreßtransformation zwischen virtueller und Hauptspeicheradresse.

Relationale Algebraoperationen, wie zum Beispiel der Join, können hohe Speicheranforderungen zur Folge haben. Durch die Verwendung eines virtuellen Speichermediums ist die Realisierung expliziter Pufferungsmechanismen in der DRAM nicht notwendig, da ein ausreichend großer virtueller Speicher zur Verfügung steht, bei dem das System den Seitenwechsel [Tane90] vornimmt.

Die Speicherverwaltung stellt bei Bedarf einen freien Speicherbereich zur Verfügung. Dieser Bereich wird so freigegeben, daß möglichst große, zusammenhängende Freispeicherbereiche entstehen. Da virtueller Speicher verwaltet wird, spielt das Einlagern bzw. Auslagern von Seiten bei der Wahl der Speicherbelegungsstrategie keine

Rolle, weil dies bereits vom System organisiert wird.

Für die Speicherverwaltung gibt es die Belegungsstrategien [Tane92, Lang91]:

- First-Fit-Strategie
- Next-Fit-Strategie und
- Best-Fit-Strategie.

Bei der First-Fit-Strategie wird der Speicherbereich zur Verfügung gestellt, der als erstes größer oder gleich dem angeforderten Bereich ist. Die Suche beginnt immer am Anfang des zu verwaltenden Bereiches. Das führt zu einer starken Zerteilung großer, zusammenhängender Bereiche und einer aufwendigeren Suche bei fortschreitender Belegung. Das Entstehen großer zusammenhängender Speicherbereiche bei der Wiedereingliederung freigegebener Bereiche ist mit großem Aufwand verbunden.

Bei der Best-Fit-Strategie wird der Nachteil der First-Fit-Strategie, daß die Suche nach bereitzustellendem Speicher immer vom Anfang des zu verwaltenden Bereiches aus erfolgt, dadurch beseitigt, daß der passende Bereich gleich dort gesucht wird, wo der letzte Suchvorgang endete. Damit werden Bereiche durchsucht, die mit großer Wahrscheinlichkeit noch nicht belegt sind. Dies bewirkt einen erheblichen Zeitgewinn, führt aber zu einer noch größeren Zergliederung als bei der First-Fit-Strategie.

Die Best-Fit-Strategie stellt den kleinsten Bereich bereit, der der Anforderung genügt. Das führt zu einer guten Auslastung des Speichers. Demgegenüber ist das Durchsuchen des gesamten Speichers nach dem optimalen Bereich sehr aufwendig.

Die Verfahren für die Speicherverwaltung [Tane92, Lang91] sind:

- das Vektorverfahren (Bit maps)
- das Tabellenverfahren (Linked lists) und
- das Halbierungsverfahren (Buddy System). [Kalf88].

Beim Vektorverfahren wird das Vielfache einer festen Einheit von Speicherzellen vergeben. Mit einem Bitvektor kann nun die Belegungsinformation für alle Speichereinheiten zusammengefaßt werden. Freier Speicherbereich mit der Größe von k $(k \in \{ 0, 1, 2, ...\})$ Speichereinheiten wird ermittelt, indem im Bitvektor nach k aufeinanderfolgenden, als frei markierten Blöcken gesucht wird. Das Auffinden k solcher zusam-

menhängender freier Speichereinheiten wird mit größerer Belegung sowohl unter Verwendung der First-Fit-Stratgie als auch unter der Verwendung der Next-Fit-Strategie immer aufwendiger, wobei die Next-Fit-Strategie wesentlich effizienter ist. Bei der Freigabe eines Speicherbereiches verschmelzen benachbarte freie Bereiche von selbst. Damit ist der Aufwand für die Freigabeoperation gering.

Beim Tabellenverfahren werden Speicherbereiche beliebiger Länge vergeben. Der Freispeicher wird doppelt verkettet verwaltet. Die Verwaltung erfolgt in absteigender Reihenfolge der Größe der freien Blöcke. Die Verwaltungsinformationen werden an den Rändern in den freien Speicherbereichen untergebracht. Dabei handelt es sich um die Art der Belegung und die Größe des entsprechenden Bereiches. Bei der Freispeicherliste kommen die Adressen von Vorgänger und Nachfolger hinzu. Bei Anwendung der First-Fit-Strategie würde immer der größte freie Speicherblock geteilt werden und der freie Bereich in die Freispeicherliste einsortiert werden. Bei dieser Zerteilung von großen, zusammenhängenden Freispeicherbereichen ist es daher eher sinnvoll, die Freispeicherliste bis zum kleinstmöglichen Bereich zu durchlaufen und diesen für die Vergabe von Speicher zu nutzen. Vor der Freigabe von Speicher sollten benachbarte Blöcke auf ihre Belegungsart überprüft werden. Für den Fall, daß Nachbarblöcke nicht belegt sind, führt das Verschmelzen benachbarter Blöcke zu größeren freien Speicherblöcken, die dann in die Freispeicherliste eingefügt werden. Diese Speicherfreigabe ist sehr aufwendig.

Beim Halbierungsverfahren werden ebenfalls die Verwaltungsinformationen im Speicherbereich untergebracht. Es wird bei jeder Speicheranforderung der kleinste 2^k $(k \in \{ 0, 1, 2, ... \})$ große Bereich, in den der angeforderte Speicher mit seiner Verwaltungsinformationen paßt, zur Verfügung gestellt. Für jede Bereichsgröße gibt es eine doppelt verkettete Freispeicherliste (siehe Abb. 15). Diese Freispeicherlisten sind wiederum untereinander nach absteigender Ordnung doppelt verkettet. Kommt es zur Anforderung von Speicher, wird die Freispeicherliste für die erforderliche Größe gesucht. Das bedeutet, daß die Liste gesucht wird, die den kleinstmöglichen 2^k großen Block für die Erfüllung der Anforderung enthält. Enthält diese Liste L_k kein Element, wird die nächste belegte Freispeicherliste $L_{k'}$ ($k' > k$) ermittelt und deren erstes Element ausgezeigert, halbiert und in die vorhergehende Freispeicherliste $L_{k'-1}$ eingehängt. Das wird wiederholt, bis in der Freispeicherliste L_k für die Blöcke der geforderten Größe

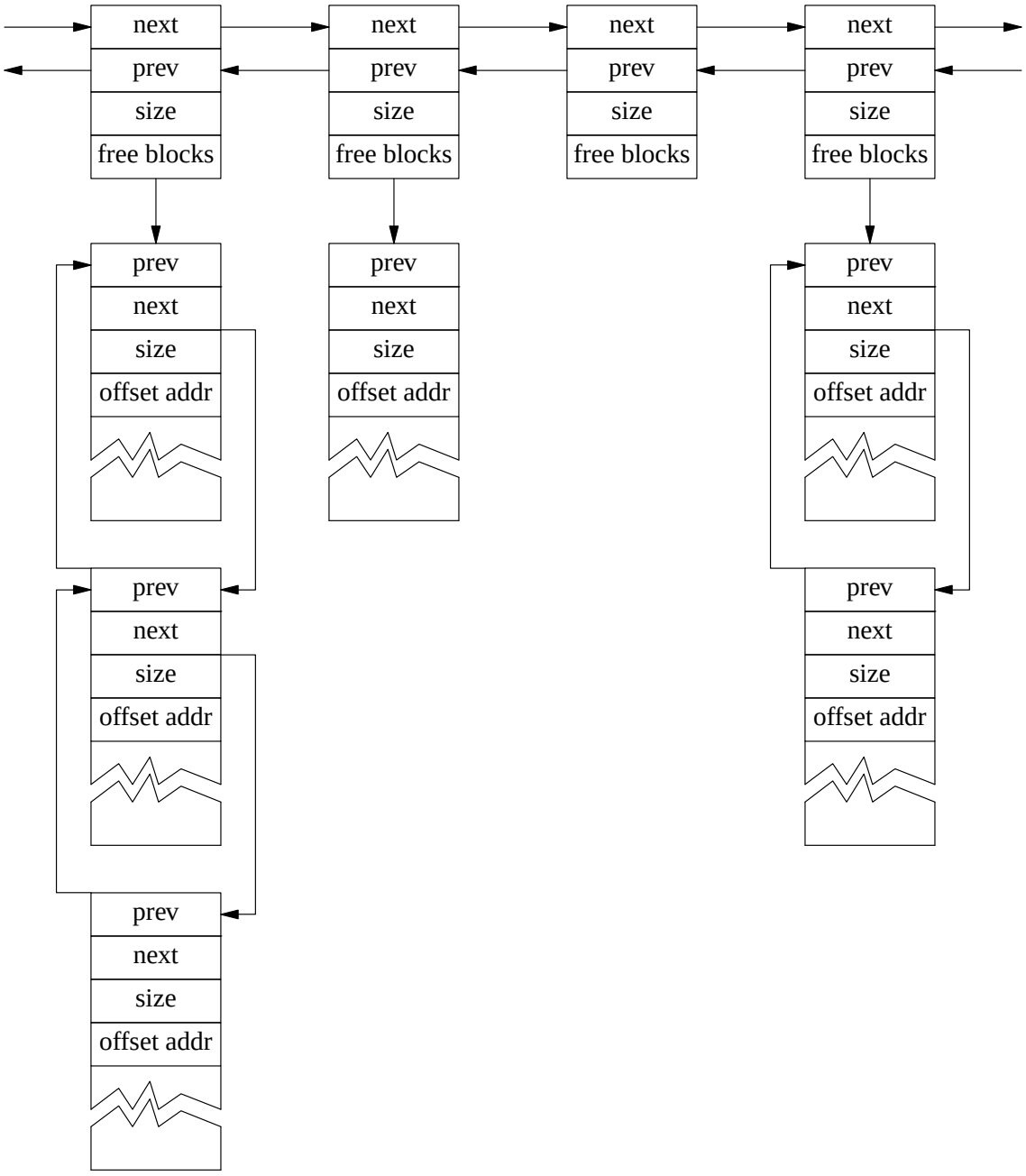


Abb. 15: Die Freispeicherliste

Elemente enthalten sind. Aus der Freispeicherliste L_k für Blöcke der angeforderten Größe wird der erste Block zur Verfügung gestellt. Bei diesem Verfahren braucht keine Belegungsstrategie ausgewählt werden, da in einer Freispeicherliste nur Blöcke gleicher Größe enthalten sind. Beim Freigeben werden benachbarte Blöcke solange verschmolzen, bis der größtmögliche sinnvoll zusammenhängende freie Speicherblock

entstanden ist. Durch die Initialisierung eines Freispeicherbereiches wird ein Block der gewünschten Größe bereitgestellt. Infolge der Speicheranforderung aus diesem Bereich kommt es zur Teilung dieses gesamten Blocks. Die bei der Teilung entstehenden Speicherblöcke werden nachfolgend weiter geteilt, wenn Speicher angefordert wird. Beim Freigeben von Speicher wird die Freispeicherliste nach Blöcken, die von der gleichen Größe wie der freizugebende Block sind, daraufhin untersucht, ob ein physischer Nachbarblock in der Liste existiert. Es ist nicht sinnvoll, den Block beliebig mit einem seiner beiden Nachbarblöcke zu einem größeren Block zu verschmelzen. In Hinsicht auf weitere Verknüpfungen zu möglichst großen freien Speicherblöcken ist es günstig, nur Blöcke miteinander zu verknüpfen, die vorher bereits einen gemeinsamen Block gebildet haben, also durch Teilung voneinander getrennt wurden. Das Verschmelzen benachbarter Speicherblöcke erfordert nur das Durchsuchen gleichgroßer Blöcke innerhalb einer Liste 2^k großer freier Blöcke. Es ist bedeutend effizienter als beim Tabellenverfahren. Der Nachteil des Halbierungsverfahrens liegt in der ungünstigen Speicherauslastung durch die interne Fragmentierung, da nur 2^k große Blöcke bei der Anforderung von Speicher zugewiesen werden.

Das Laufzeitverhalten für die Anforderung von Speicher beim Halbierungsverfahren ist im Vergleich zu den anderen oben genannten Verfahren am günstigsten, weil hier das erste Element einer Freispeicherliste genutzt wird, das zudem die kleinste mögliche Blockgröße besitzt. Dadurch erfolgt auch keine übermäßig starke Zergliederung des Speichers. Für die Freigabe von Speicher ist das Laufzeitverhalten des Vektorverfahrens am günstigsten. Es steht aber in keinem Verhältnis zu dessen starken Zerstückelung des Speichers durch die Speicheranforderung. Das Tabellenverfahren hat die günstigste Speicherauslastung, weil nur soviel Speicher belegt ist, wie auch angefordert wurde. Die Zergliederung des Speichers ist gering, wenn der kleinstmögliche Block in der absteigend geordneten Freispeicherliste gesucht wird, was wiederum mit einem höheren Aufwand verbunden ist. Der hohe Aufwand für das Verschmelzen benachbarter Blöcke beim Freigeben, um möglichst große Blöcke zu erhalten, ist bei diesem Verfahren von großem Nachteil.

Als eines der geeigneten Verfahren kommt für die Freispeicherverwaltung des virtuellen Shared memory das Halbierungsverfahren zur Anwendung. Für jede Speicherblockgröße existiert eine doppelt verkettete Freispeicherliste. Die im freien

Speicherblock untergebrachten Verwaltungsinformationen bestehen aus den Zeigern auf den Vorgänger und auf den Nachfolger in der Freispeicherliste, der Größe des Bereiches und der Offset-Adresse innerhalb des zu verwaltenden Speicherbereiches. Diese Listen sind wiederum in geordneter Reihenfolge doppelt verkettet. Mit dieser Verkettung ist es möglich, in sortierter Reihenfolge auf die jeweils ersten Elemente der Freispeicherliste zuzugreifen (siehe Abb. 15).

4. Realisierung der Freispeicherverwaltung

Der virtuelle Shared memory wird durch die Klassen *mmap*, *man_mem* und *mmem* realisiert:

```
class memmap {
      public:
      caddr_t    get_mmap( int len);
      caddr_t    get_mmap( const char *fn, int prot, int flags, int len);
      void       del_mmap( caddr_t p_arena, int len);
};



struct chain_of_free_mem {
      chain_of_free_mem     *prev;
      chain_of_free_mem     *next;
      size_t                size;
      int                   addr;
};

struct free_mem_list {
      free_mem_list         *prev;
      free_mem_list         *next;
      size_t                size;
      chain_of_free_mem     *first_free_block;
};

struct mmap_head {
      free_mem_list    *first;
      free_mem_list    *last;
      size_t           size;
};



class man_mem:public memmap, free_mem {
      protected:
                  man_mem( long size, statist_t s = NO_STATISTICS);
                  ~man_mem();
```

```
        free_mem_list          *split_mem( free_mem_list* fml);
        chain_of_free_mem      *conn_mem( free_mem_list* fml,
                                          chain_of_free_mem* block1,
                                          chain_of_free_mem* block2);
};



class mmem:public man_mem {
        public:
                mmem(long sz, statist_t st = NO_STATISTICS);
        void  *alloc( long len);
        int   free( void *arena);
};
```

Die Klasse *memmap* stellt eine Schnittstelle zu den Systemfunktionen für die Nutzung von Memory mapped files (*mmap*) [Tane92] zur Verfügung.

Die Methode *memmap::get_mmap(len)* legt ein Memory mapped file der Länge *len* an. Zum Auslagern von Speicherseiten auf ein externes Speichermedium wird *"/dev/zero"* verwendet, es sind Schreib- und Lesezugriffe erlaubt. Der virtuelle Speicherbereich kann als Shared memory benutzt werden. Eine allgemeine Methode für das Anfordern eines *mmap*-Bereiches ist *memmap::get_mmap(fn,prot,flags,len)*, bei der virtueller Shared memory mit den gewünschten Rechten, der gewünschten Länge und dem übergebenen Swap file angelegt wird. Freigegeben wird der *mmap*-Bereich mit *memmap*: : *del_mmap*(*p_arena*, *len*), wobei *arena* die Adresse des Memory mapped-Bereichsanfangs ist und *len* die Länge des freizugebenden Bereiches.

Die Klasse *man_mem* stellt Member-Funktionen für die Verwaltung des virtuellen Shared memory bereit. Von der Klasse *free_mem* erbt sie Funktionen für die Verwaltung der Freispeicherketten und deren Verkettung untereinander. Die Klasse *memmap* stellt die Funktionen für das Anfordern und Freigeben des virtuellen Shared memory bereit. Mit Hilfe des Konstruktors *man_mem::man_mem(size,s)* wird der virtuelle Shared memory angelegt, so daß sowohl die gewünschte Speichergröße zur Verfügung steht als auch die Verwaltungsinformationen in diesem Bereich untergebracht werden können. Die Verwaltungsinformationen (siehe Abb. 16) bestehen aus:

- dem *mmap_head*, der Angaben über die Größe des gesamten virtuellen Shared

memory und Zeiger auf das erste und das letzte Element der Freispeicherliste enthält und

- aus der Freispeicherliste.

Im zu verwaltenden Speicherbereich selbst befinden sich freie und belegte Speicherblöcke. Der Konstruktor initialisiert die Freispeicherliste und hängt den gesamten zu verwaltenden Bereich als freien Block in das letzte Element der Freispeicherliste. Die Klasse nutzt des weiteren eine Instanz der Klasse *statistics*. Die Klasse *statistics* ermöglicht Testmessungen zur Speicheranforderung im *mmap* Bereich und zur Dauer der Belegung dieses Speicherbereiches. Der Destruktor *man_mem::˜man_mem()* sorgt für die Freigabe des virtuellen Shared memory-Bereiches.

mmap_head	**free_mem_list**	**memory to manage**

Abb. 16: Die Struktur des Memory mapped Bereiches

Das Auffinden des kleinstmöglichen freien Speicherblocks für einen angeforderten Bereich erfolgt durch die Methode *man_mem::split_mem(fml)*. Sie teilt den ersten Block der angegebenen Liste freier Speicherblöcke in zwei gleichgroße Blöcke und hängt sie in die vorherige Freispeicherliste.

Für das Verschmelzen benachbarter Speicherblöcke, deren Verknüpfung sinnvoll ist, enthält ein Speicherblock zwei wesentliche Informationen: die Größe des Blockes und seine Offset-Adresse innerhalb des zu verwaltenden Bereiches. Die Größe ist eine notwendige Information für das Einsortieren des Blocks in die Liste des entsprechenden Freispeicherlistenelementes. Die Offset-Adresse wird für das Zusammenfügen benachbarter Bereiche benötigt. Mit ihrer Hilfe ist erkennbar, ob die beiden Blöcke physisch benachbart sind, und ob sie durch Teilung auseinander hervorgegangen sind. Die Member-Funktion *man_mem:conn_mem(fml,block1,block2)* überprüft, ob die beiden Speicherblöcke sinnvoll zusammenfügbare physische Nachbarblöcke sind. Ist das der Fall, so löst die Member-Funktion sie aus ihrer Liste freier Blöcke heraus und verknüpft

sie. Als Ergebnis liefert sie den Zeiger auf den neuen Block, der noch nicht in die Freispeicherliste eingefügt wurde.

Die Klasse *mmem* (Mapped memory) stellt folgende Methoden zur Verfügung:

- zum Bereitstellen und Initialisieren des virtuellen Speicherbereiches *mmem*: : *mmem*(*sz*)
- zum Anfordern von Speicher *mmem::alloc(len)* und
- zum Freigeben von Speicher *mmem::free(arena)*.

Speicher aus dem so verwalteten Shared memory-Bereich kann von verschiedenen Prozessen angefordert werden. Der Zugriff auf diese gemeinsam genutzte Ressource wird durch gegenseitigen Ausschluß synchronisiert, der mittels Semaphoren realisiert wird.

5. Realisierung des HEAD-Tupelprotokolls

Für die Übertragung der Relationen in der DRAM wird ein Tupelstrom erzeugt. Zur Verringerung des Kommunikationsoverheads werden die Tupel nicht einzeln, sondern in kleinen Paketen zusammengefaßt, als Tupelblöcke verschickt. Die Instanzen zur Realisierung des Übertragungsprotokolls werden durch C++-Klassen realisiert [Stro91a, Krus94]. Die Bibliothek *libipc* [Meye92] enthält eine Klassenhierarchie zur Interprozeßkommunikation mit verschiedenen Transportprotokollen. Die Basisklasse *ipc* ist protokollunabhängig. Zur Realisierung der Instanzen der darüberliegenden Schichten werden ihre Methoden für das Senden und Empfangen von Daten verwendet, so daß auch deren Protokolle von einem konkreten Transportprotokoll unabhängig implementiert werden können. Damit ist sowohl die Nutzung einer verbindungslosen als auch einer verbindungsorientierten Übertragung durch die darüberliegenden Schichten durchführbar.

Die Abbildung 17 zeigt die implementierte Klassenhierarchie für das Protokoll zur Übertragung von Tupelblöcken in HEAD. Die durchgezogenen Linien zeigen Vererbung zwischen Klassen an, die gestrichelten Linien die Nutzung von Instanzen der einen Klasse in der anderen Klasse. Bei der dunkel unterlegten Klasse *ipc* handelt es sich um die Basisklasse der *ipc*-Klassenhierarchie aus *libipc*. Die hell unterlegten Klassen sind bei der weiteren Implementierung der Operationen der Relationenalgebra von Bedeu-

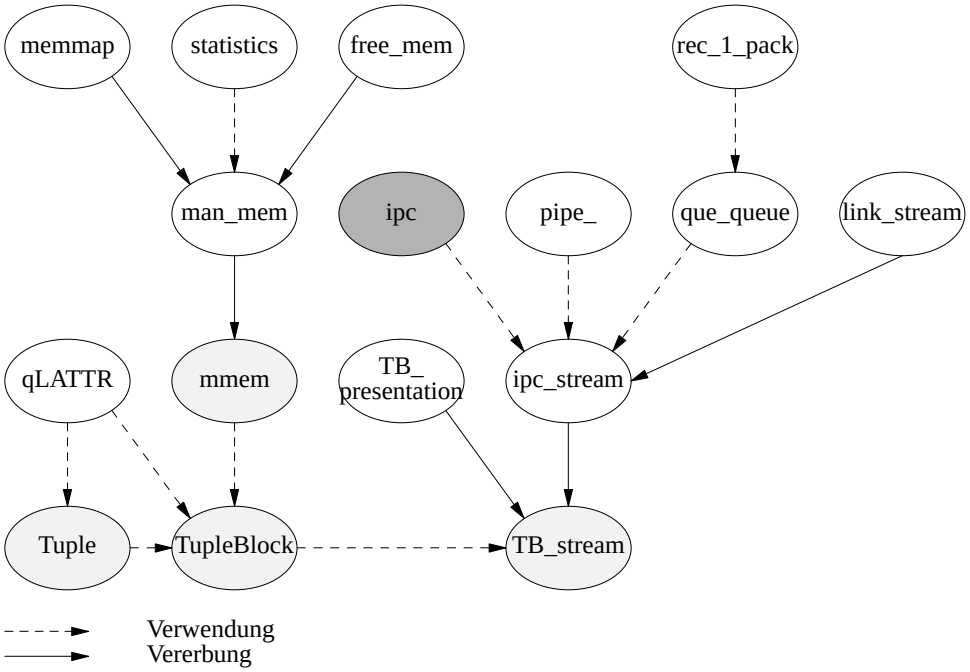


Abb. 17: Klassenhierarchie des Datenflußprotokolls und der Freispeicherverwaltung

tung.

Wegen der Verwendung von UDP als Transportprotokoll müssen, wie bereits erwähnt, zusätzliche Maßnahmen für die Sicherung der Übertragung getroffen werden.

5.1. Das HEAD-Tupelprotokoll und das OSI-7-Schichtenmodell

Die transportdienstleistenden Schichten (Schicht 1 bis Schicht 4) ermöglichen den darüberliegenden Schichten, netzwerkunabhängige Transportdienste zu nutzen. Die Hauptaufgaben der Sitzungsschicht (Schicht 5) bestehen im Ermöglichen des Aufbaus einer Verbindung (=Sitzung) und der geordneten Datenübertragung. Im HEAD-Tupelprotokoll (siehe Abb. 18) stellt die Übertragung eines Tupelblocks eine Sitzung dar. Die Funktionen der Klassen aus der *libipc*-Klassenbibliothek realisieren die Aufgaben des Verbindungsaufbaus, der Aufrechterhaltung der Verbindung und des geordneten Verbindungsabbaus. Die Funktionen der Klasse *link_stream* zur Unterstützung eines Schiebefensterprotokolls [Tane81] dienen im Fall von unzuverlässigen Transportprotokollen wie UDP der sicheren Übertragung von Tupelblöcken. Die Klasse *ipc_stream* gewährleistet mit Hilfe der Sende- und Empfangsprimitive aus der *ipc-Klasse* die ge-

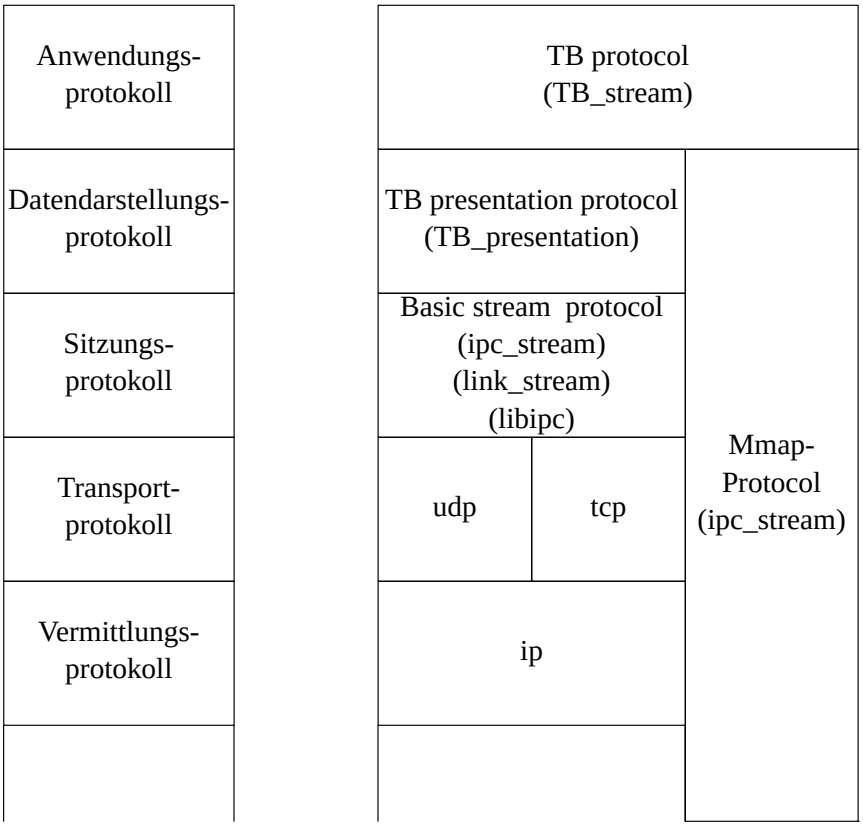


Abb. 18: Gegenüberstellung des OSI-Protokolls und des DRAM-Datenflußprotokolls

ordnete Übertragung eines Tupelblocks. Ihre Funktionen übernehmen gegebenenfalls die Zerlegung der Tupelblöcke in MTU-große Pakete †. Die Funktionen der Klassen aus der *libipc*-Klassenbibliothek sowie der Klassen *link_stream* und *ipc_stream* erfüllen die Aufgaben einer OSI-Sitzungsschicht [Tane81].

Die Klasse *TB_presentation* erfüllt die Aufgaben der in der OSI-Datendarstellungsschicht geforderten Funktionen zur Konvertierung von system- und maschineninternen Datenformaten in einheitliche Standardformate und deren Rückkonvertierung (Unterstützung heterogener, offener Systeme). Gegenwärtig erfolgt die Konvertierung auf Basis der durch das UNIX-System zur Verfügung stehenden Funktionen zur rechnerunabhängigen Darstellung von Basistypen. Werkzeuge zur Umwandlung komplexer Datenstrukturen in ASN.1 (Abstract Syntax Notation One) [Gora90], dem ISO-Standard für die Beschreibung von Daten und Datenstrukturen, werden durch das

† Maximum Transmission Unit (MTU), die maximale Übertragungseinheit im Rechnernetz

ISODE-Paket oder durch spezielle ASN.1 Compiler zur Verfügung gestellt.

Die Kommunikation zwischen Anwendungsprozessen, die der siebenten OSI-Schicht (Anwendungsschicht) zuzuordnen sind, wird durch das Tupel Block Protokoll (TB_Protocol) realisiert. Es verwirklicht die gesamte Realisierung eines Tupelstroms.

Die oben beschriebene Protokollhierarchie wird bei der Kommunikation zwischen Prozessen, die auf einem Rechnerknoten ablaufen, aus Effiziensgründen geändert. In diesem Fall kommt der virtuelle Shared memory (*mmap*) als Kommunikationsmedium zur Anwendung. Die Übertragung der Strukturinformationen und des Zeigers auf die Tupelwerte im Shared memory erfolgt mittels Nachrichtenübertragung entsprechend der angegebenen Protokollhierarchie. Damit kann auf Kopiervorgänge, die andere IPC-Mechanismen hervorrufen, auf die Umwandlung in eine abstrakte Datenpräsentation sowie auf die Zerlegung der Tupelblöcke in MTU-große Datenpakete verzichtet werden. Des weiteren muß auf der Empfängerseite keine Rekonstruktion des Tupelblocks erfolgen.

Die Funktionen der obersten Schicht des Übertragungsprotokolls befreien den Benutzer von diesen Details bei der Kommunikation zwischen den Prozessen der DRAM.

5.2. Die Nachricht des Tupelprotokolls - der Tupelblock

Die Nachrichten des Tupelprotokolls sind Tupelblöcke. Ein Tupelblock besteht aus mehreren Tupeln einer Relation, die von den Operationen der Relationenalgebra verarbeitet werden. Die C++-Klassendefinitionen für die Klasse *Tuple* und für die Klasse *TupleBlock* sind:

```
class Tuple {
      public:
                  Tuple( qLATTR *qla, char *T = NULL);
                  ~Tuple();

            // copy constructor
                  Tuple( Tupel &T, char *T = NULL);
      void        operator = (Tuple &T);

      qLATTR      *goto_ATTR( char *v, char *a);
      ptr_to      value();
      void        value( ptr_to va);
      attrtype    getatype();
      count_t     getasize();
      Tuple      *BuiltNewTuple( Tuple *T1, Tuple *T2, qLATTR *dst_qla);
};
```

```
class TupleBlock {
      public:
                  TupleBlock( long count, long size, mmem *mm);
                  TupleBlock( long count, long size, char *tb);
                  TupleBlock( long count, qLATTR *qla, mmem *mm);
                  TupleBlock( long count, qLATTR *qla, char *tb);
                  ~TupleBlock();

      void     ins( Tuple *T, mmem *mm);
      Tuple    *del( qLATTR *qla);
      long     TupCnt();
      long     TupSize();

      char  *first();

      // iterator stuff
      Tuple *first( qLATTR *qla);
      Tuple *next( qLATTR *qla);
      Tuple *prev( qLATTR *qla);
      Tuple *last( qLATTR *qla);
};
```

Die Aufteilung einer Relation in Tupelblöcke fördert die vertikale Parallelität. Die Tupel eines Tupelblocks werden kompakt in einem zusammenhängenden Speicherbereich untergebracht. Zur Unterstützung der effizienten Kommunikation zwischen Prozessen auf einem Rechnerknoten werden die Tupelblöcke grundsätzlich im Shared memory gespeichert.

Der Konstruktor *Tuple::Tuple(qla,T)* erzeugt eine Instanz des Tupels. Der Standardwert für *T* ist *NULL* und bedeutet, daß ein Tupel mit leeren Tupelwerten erzeugt wird. Wird für *T* der Zeiger auf die Tupelwerte mit übergeben, wird das Tupel mit diesen initialisiert. Damit entfällt das Kopieren von Tupelwerten. Der Copy-Konstruktor [Stro91b] ermöglicht die Zuweisung eines Tupels an ein anderes Tupel. Die Member-Funktion *Tuple::goto_ATTR(v,a)* legt fest, auf welches Attribut *a* der Variablen einer Relation *v* sich das Ermitteln eines Attributwertes *Tuple::value()*, das Ändern eines Attributwertes *Tuple::value(va)* und das Ermitteln eines Attributtyps *Tuple::getatype()* bezieht. Um einen ermittelten Attributwert verarbeiten zu können,

muß der Attributtyp bestimmt werden. Ist der Attributtyp eine Zeichenkette, also vom Typ *A_CHAR*, spielt auch die Länge der Zeichenkette eine Rolle, die mit *Tuple::getasize()* ermittelt wird. Mit der Funktion *Tuple::BuiltNewTuple(T1,T2,dst_qla)* wird ein neues Tupel entsprechend der Attributliste *dst_qla* aus den Tupeln *T1* und *T2* erzeugt.

Der Konstruktor der Klasse *TupleBlock* ist mehrmals überladen [Stro91a]:

- *TupleBlock(count,size,mm)* erzeugt eine Instanz für einen leeren Tupelblock für *count* Tupel der Tupellänge *size* in einem Mmap-Bereich *mm*.
- *TupleBlock(count,size,tb)* erzeugt eine Instanz für einen Tupelblock für *count* Tupel der Tupellänge *size*. Das Feld *tb* liegt im Mmap-Bereich und enthält die Tupelwerte des Tupelblocks.
- *TupleBlock(count,qla,mm)* erzeugt eine Instanz für einen leeren Tupelblock für *count* Tupel mit der Attributliste *qla* in einem Mmap-Bereich *mm*.
- *TupleBlock(count,qla,tb)* erzeugt eine Instanz für einen Tupelblock für *count* Tupel mit der Attributliste *qla*. Das Feld *tb* liegt im Mmap-Bereich und enthält die Tupelwerte des Tupelblocks.

Um ein Tupel *T* in den Tupelblock einzufügen, wird die Methode *TupleBlock::ins(T,mm)* benutzt. Soll das Tupel in einen bereits vollen Tupelblock geschrieben werden, wird der Tupelblock vergrößert. Das ist mit einem Umkopieren des alten Tupelblocks verbunden. Deshalb empfiehlt es sich, die zu erwartende Größe des Tupelblocks für die Initialisierung zu verwenden. Das letzte Tupel eines Tupelblocks wird mit *TupleBlock::del(qla)* aus dem Tupelblock entfernt. Als Ergebnis wird das Tupel geliefert. Mit *TupleBlock::TupSize()* wird die Größe eines Tupels und mit *TupleBlock::TupCnt()* die Anzahl der Tupel in dem Tupelblock geliefert. *TupleBlock::first()* liefert den Zeiger auf den Anfang des Tupelblocks, was bei Operationen von Nutzen ist, die nicht die einzelnen Tupel betreffen, sondern den gesamten Block (z.B. das Senden und das Empfangen).

Die tupelweise Bearbeitung eines Tupelblocks wird durch den Iterator [Stro91b] unterstützt. Die Iterator-Operationen liefern einen Zeiger auf ein Tupel zurück. Um den Nachfolger des gerade bearbeiteten Tupels zu erhalten, benötigt man

TupleBlock::next(qla). Das vorhergehende Tupel erhält man mit *TupleBlock::prev(qla)*. Mit *TupleBlock::first(qla)* erhält man das erste Tupel und mit *TupleBlock::last(qla)* das letzte Tupel im Tupelblock.

5.3. Das Basic Stream Protocol (BSP)

Das **B**asic **S**tream **P**rotocol (BSP) ermöglicht mit Hilfe der in der Klasse *ipc* zur Verfügung gestellten Funktionen, Verbindungen auf- und abzubauen und anhand der Funktionen der Klassen *link_stream* und *ipc_stream*, die Daten einer Relation in Form von Tupelblöcken zu übertragen. Es gewährleistet eine sichere nachrichtenorientierte Übertragung, bei der die Reihenfolge der ankommenden Tupelblöcke keine wesentliche Rolle spielt. Da unsichere Transportprotokolle zur Anwendung kommen können, wird bei solchen Protokollen ein Schiebefensterprotokoll (Sliding window protocol) durch Nutzung von Funktionen der Klasse *link_stream* realisiert.

5.3.1. Das Schiebefensterprotokoll

Um ein sicheres Protokoll auf der Basis des UDP zu implementieren, wird ein Schiebefensterprotokoll [Tane81] verwendet. Da UDP nicht gewährleistet, daß abgeschickte Datenpakete auch ankommen, erwartet der Sender für jedes Datenpaket eine Bestätigung (Acknowledgement) für den Empfang. Um zu verhindern, daß Daten beim Empfänger überhaupt nicht ankommen, verschickt der Sender ein Datenpaket erneut, wenn er für das Datenpaket in einem gewissen Zeitintervall keine Empfangsbestätigung erhalten hat. Für ein Datenpaket, das beim Empfänger eingetroffen ist, kann die Empfangsbestätigung verlorengehen. Dadurch kann der Empfänger das daraufhin erneut abgeschickte Datenpaket mehrfach erhalten. Da der Empfänger die eingetroffenen Datenpakete registriert, kann er entscheiden, ob die Daten bereits entgegengenommen wurden. In diesem Fall kann er sie verwerfen. Ansonsten nimmt er sie entgegen und bestätigt den Empfang.

Eine Lösung für ein Schiebefensterprotokoll ist z.B. das Stop and wait-Protokoll [Tane81], das:

- ein Datenpaket abschickt und

- auf die Empfangsbestätigung wartet, ehe das nächste Datenpaket gesendet wird.

Bei der Implementierung des Schiebefensterprotokolls in HEAD muß die Reihenfolge der abgeschickten Protokolldateneinheiten (PDU, Protocol Data Unit) [Tane81] nicht mit der Reihenfolge der angekommenen PDUs übereinstimmen. Jede PDU erhält für die Übertragung im Rechnernetz bei der Anwendung des Schiebefensterprotokolls eine Rahmennummer (*FrameNo*).

Das Sendefenster besitzt eine festgelegte Größe, die angibt, wieviele PDUs maximal aufgenommen werden können. Jede PDU, die abgeschickt wird, wird in dieses Fenster eingetragen. Sie verbleibt solange im Fenster, bis ihre Bestätigung eingetroffen ist. Nach dem Eintreffen der Bestätigung wird die PDU aus dem Fenster entfernt. Damit enthält das Fenster nur abgeschickte, unbestätigte PDUs. Ist das Schiebefenster voll, kann erst wieder eine PDU übernommen und damit auch abgeschickt werden, wenn eine PDU aus dem Schiebefenster freigegeben wurde.

Das Schiebefenster des Empfängers nimmt die empfangenen PDUs auf. Jede PDU, die nicht angenommen wird - in der Regel die Duplikate bereits empfangener PDUs, kann kommentarlos gelöscht werden. Für jede empfangene PDU wird eine Empfangsbestätigung an den Sender geschickt.

Für die Realisierung des Schiebefensterprotokolls in HEAD läuft ein separater Prozeß. Dieser und der Prozeß, der die Verarbeitung der empfangenen Daten ausführt, können als Leichtgewichtsprozesse realisiert werden. Der Prozeß, der das Schiebefenster-Protokoll realisiert, übernimmt die zu verschickenden PDUs und fügt sie in eine Warteschlange ein. Wenn das Schiebefenster PDUs aufnehmen kann, wird das erste Listenelement in das Schiebefenster übernommen und abgeschickt. Dabei wird der PDU eine Rahmennummer (*FrameNo*) zugewiesen. Sie ergibt sich immer aus dem Inkrement der vorherigen Rahmennummer und entspricht somit immer der Anzahl der bereits verschickten PDUs. Trifft die Empfangsbestätigung für eine PDU ein, wird sie aus dem Fenster gelöscht. Ist das Schiebefenster voll und nach einer bestimmten Zeit keine Bestätigung für eine PDU eingetroffen, werden alle älteren PDUs, die sich im Schiebefenster befinden, erneut abgeschickt. Der Empfänger führt eine Liste mit allen Rahmennummern der bereits empfangenen PDUs. Anhand dieser Liste entscheidet der Empfänger, ob er eine PDU entgegennimmt und bestätigt oder kommentarlos verwirft.

5.3.2. Die Operationen des BSP mit ihren Schnittstellen

Neben den Funktionen zum Auf- und Abbau der Verbindungen, die von der Klasse *ipc* zur Verfügung gestellt werden, stellt das BSP Sende- und Empfangsoperationen bereit. Die C++-Klassendefinitionen für die das BSP realisierenden Klassen *link_stream* und *ipc_stream* sind:

```
class link_stream {
      public:
            link_stream( ipc *ip1);
            ~link_stream();

      // for the sender
      long  send( char *buf, size_t size);
      void sendEnd();
      void sender();

      // for the receive process
      bool  IsFrameNew( long FrameNo);
      bool  InsertRecvFrame( long FrameNo);
      void  SendAckn( long FrameNo);
};
```

```
class ipc_stream:link_stream      {
      public:
            ipc_stream( ipc *ip);
            ~ipc_stream();
      long send( long PackNo, char *Msg);
      long sendDATA( long PackNo, long TupCnt, long TupSize);
      long sendEOT( long sendedPacks);
      long sendMMAP( long TupCnt, long TupSize, char *TupleBlockAddr);
      long sendEOT_MMAP();
      void receivePACK( pipe_ *P, mmem *mm);
};
```

Bei den Sendeoperationen werden die PDUs bei der Verwendung eines sicheren Transportprotokolls mit dessen Sende- und Empfangsdiensten verschickt. Ein unzuver-

lässiges Transportprotokoll macht die Sicherung der Übertragung durch Operationen der Klasse *link_stream* notwendig.

Der Sender übergibt mit *link_stream::send(buf,size)* die PDUs an den Prozeß, der das Schiebefensterprotokoll realisiert. Die Funktion *link_stream::sender()* verwirklicht das Schiebefensterprotokoll, indem es PDUs entgegennimmt und in eine Warteschlange einträgt. Ist im Schiebefenster ein Platz frei, wird die erste PDU der Warteschlange dort eingetragen. Sie erhält ihre Rahmennummer. Durch die Sendedienste des unzuverlässigen Transportprotokolls erfolgt dann die Übertragung der PDU. Sie verbleibt solange im Schiebefenster, bis eine Bestätigung (Acknowledgement) für den Empfang dieser PDU eingetroffen ist. Mit Hilfe eines Round trip time-Handlers (rtt-Handler) wird nach Überschreitung der durchschnittlichen Übertragungszeit ohne Ankunftsbestätigung der PDU, d.h. einem Timeout, die PDU aus dem Schiebefenster erneut verschickt.

Durch die PDUs können verschiedene Nachrichten übertragen werden - Datenfelder und Steuerfelder. An jede Nachricht wird eine Nummer *PackNo* vergeben, durch die eine Zuordnung der Nachricht zu einem bestimmten Tupelblock erfolgen kann.

Die Daten eines Tupelblocks werden mit *ipc_stream::send(PackNo,Msg)* verschickt. Gegebenenfalls erfolgt eine Zerlegung der Nachricht *Msg* in MTU-große PDUs. Die Übertragung der Strukturinformationen für die Wiederherstellung des Tupelblocks durch den Empfänger (Tupelblocknummer, Tupelanzahl und Tupelgröße) übernimmt *ipc_stream::sendDATA(PackNo,TupCnt,TupSize)*. Das Ende einer Übertragung teilt der Sender durch *ipc_stream::sendEOT(sendedPacks)* mit. Noch nicht entgegengenommene PDUs können von der Ende-Nachricht bei der Verwendung nicht sequentieller Transportdienste überholt werden. Damit die von der Ende-Nachricht überholten PDUs ordnungsgemäß empfangen werden, bevor die Verbindung abgebaut wird, wird dem Empfänger die Anzahl der verschickten Tupelblöcke (*sendedPacks*) mitgeteilt.

Für den Empfang der PDUs werden die Dienste der Transportschicht verwendet. Bei der Entgegennahme einer PDU wird sofort die Empfangsbestätigung abgeschickt. Wird ein unsicheres Transportprotokoll verwendet, wird durch *link_stream::IsFrameNew(FrameNo)* überprüft, ob eine PDU mit der Rahmennummer *FrameNo* zum wiederholten Mal entgegengenommen wurde. Ist das der Fall, wird die Nachricht kommentarlos verworfen. Ansonsten erfolgt ein Vermerk über den Empfang

in einer Liste, die die eingetroffenen PDUs registriert. Der Empfangsprozeß, der durch *ipc_stream::receivePACK(P,mm)* realisiert wird, empfängt die PDUs und stellt aus ihnen die Nachrichten wieder her. Dafür werden die von der Klasse *que_queue* zur Verfügung gestellten Funktionen verwendet, um die empfangenen Daten- und Steuerfelder der Tupelblöcke in separaten Listen zu verwalten. Sind alle PDUs eines Tupelblockes korrekt eingetroffen, wird im Mmap-Bereich *mm* das Datenfeld des Tupelblocks zusammengebaut. Über die Pipe *P* wird die Strukturinformation - Anzahl der Tupel und Größe eines Tupels - den Diensten der nächsthöheren Protokollschichten zur Verfügung gestellt.

Es bietet sich an, Empfangsprozeß, Sendeprozeß und den tupelverarbeitenden Prozeß als Leichtgewichtsprozesse zu realisieren.

5.3.3. Die Protokolldateneinheit (PDU) des BSP

Die Protokolleinheit besteht aus einem Nachrichtenkopf (Header) und dem Datenfeld (siehe Abb. 19 und Tab. 5). Das Typfeld im Header gibt bei Steuerfeldern an, welche Steuerinformation verschickt wird. Datenfelder sind vom Typ *PACKAGE*. Jede PDU enthält ihre Rahmennummer. Für den Nachrichtenkopf sind drei Argumente möglich (siehe Tab. 5).

Type	**Argument 1**	**Argument 2**	**Argument 3**	**Function**
PACKAGE	Tuple Block Number	Sequence	Bytes	Message
DATA	Tuple Block Number	Tuple Count	Tuple Size	Tuple Block structure
EOT	Count of sended Tuple Blocks			End Of Trans-mission
ACKN				Acknowledge-ment

Tab. 5: Aufbau der Steuerblöcke

Tuple Block Number ist die Nummer des Tupelblocks, dem die PDU zugeordnet wird. *Tupel Count* ist die Anzahl der Tupel, die der entsprechende Tupelblock besitzt, *Tupel*

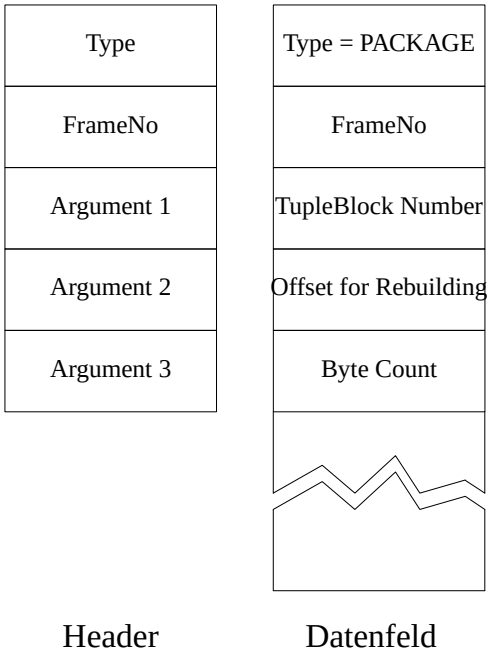


Abb. 19: Die Paketarten im Datenflußprotokoll in HEAD

Size die Größe eines Tupels. Damit kann der Empfänger feststellen, ob ein Tupelblock komplett entgegengenommen wurde. Die Sequenz gibt bei Datenfeldern an, an welcher Position sie beim Zusammensetzen der Nachrichten, die bei *ipc::stream_send()* zerlegt wurden, gehören. *Bytes* (oder *Byte Count*) gibt die Größe eines Datenfeldes an. *Count Of Sended Tuple Blocks* ist die Mitteilung des Senders über die Anzahl der abgeschickten Tupelblöcke. Damit kann der Empfänger alle Daten ordnungsgemäß entgegennehmen, bevor er den Verbindungsabbau veranlaßt.

5.3.4. Die Nutzung des BSP

Der Konstruktor der Klasse *ipc_stream*, die die Funktionen für die Nachrichtenübertragung bereitstellt, benötigt für seine Initialisierung den Zeiger auf eine IPC-Instanz [Meye92]. Über diese IPC-Instanz erfolgt der Aufbau, der Abbau und gegebenenfalls die Aufrechterhaltung der Verbindung. Der Beginn einer Tupelblockübertragung wird durch das Steuerfeld *DATA* signalisiert. Es enthält die für die Herstellung des Tupelblocks notwendigen Informationen. Die Datenfelder (PDU-Typ: *PACKAGE*) enthalten die Tupelblöcke. Die ordnungsgemäße Entgegennahme aller PDUs vor dem Ver-

bindungsabbau gewährleistet der Empfänger, indem er nach dem Erhalt des EOT-Steuerfeldes die Entgegennahme aller PDUs feststellt. Ist das nicht der Fall, wartet der Empfänger die Ankunft der PDUs ab, die vom EOT-Steuerfeld überholt wurden. Bei der Verwendung eines verbindungsunabhängigen, unsicheren Transportprotokolls wird ein Schiebefensterprotokoll realisiert. Das Senden von Tupelblöcken wird nachfolgend dargestellt†:

```
// Create a field of TupleBlocks TB[10]
// Put Tuples into the TupleBlocks
TupleBlock TB[10];

// Create an ipc-instance, establish and create the connection
ipc *conn;

// Send the TupleBlocks
ipc_stream *stream = new ipc_stream( conn);

for ( int i=0; i < 10; i++) {
      stream->sendDATA( i, TB[i]->TupCnt(), TB[i]->TupSize());
      stream->send( i, TB[i]->first());
}

stream->sendEOT( i);

// Delete TupleBlocks, stream and conn
```

Beim Empfang wertet ein Empfangsprozeß die ankommenden PDUs aus und stellt die ursprüngliche Nachricht† in einem Memory mapped-Bereich wieder her. Anschließend gibt er die Strukturinformation an den Nutzer über eine Pipe weiter. Der Empfang von Nachrichtenpaketen wird nachfolgend dargestellt:

---

† Die Darstellung erfolgt in einem an C++ orientierten Pseudo-Code.

† Ist die Nachricht größer als die MTU, muß sie für die Kommunikation über das Netz zergliedert werden.

```
// Use the Pipe P for Communication between receive process
// and operating process

// Create the memory mapped arena
mmem *mm = new mmem( MMAP_SIZE);

// Create an ipc-instance, establish and create the connection
ipc *conn;

// Receive the messages
que_queue      *ReceiveList;
package_t      *package;
message_t      *msg:
ipc_stream     *stream = new ipc_stream( conn);

while ( not( End Of Transmission) && not( All Tupel Blocks Are Received)) {

      // receive a package
      package = stream->receivePACK();

      // put the package entry ( message) in it's receive list
      ReceiveList->Insert( TupleBlockNumber( package), MSG( package));

      if ( ReceiveList
            ->IsTupleBlockCompleteReceived( TupleBlockNumber( package))) {

            msg = ReceiveList
                  ->BuiltCompleteMsgInMMAP( TupleBlockNumber( package), mm);

            // send via Pipe the address of the message in Mmmap and the
            // structure of the TupleBlock to the operating ( user) process
            P->write( &msg, ReceiveList->TupCnt(), ReceiveList->TupSize());
      }
}
```

Der tupelverarbeitende Prozeß nimmt die Strukturinformationen aus der Pipe entgegen und erzeugt ein Objekt der Klasse TupleBlock:

```
// Use the Pipe P for Communication between receive process
// and operating process

// Use the Mmap-Bereich mm as Shared memory arena

// Receive the TupleBlocks
TupleBlock  *TB;
count_t     TupCnt;
count_t     TupSize;
message_t   *msg;

while ( TRUE) {

      // receive vi Pipe the address of the message in Mmap and the
      // structure of the Tupleblock from the receive process
      P->read( &msg, &TupCnt, &TupSize);

      // create an object of the class TupleBlock
      TB = new TupleBlock( TupCnt, TupSize, msg);

      // Now, you can manipulate this TupleBlock TB
      ...
}
```

Die Abbildung 20 stellt die Vorgänge beim Senden und Empfangen einer einwertigen Operation dar. Ein Empfangsprozeß nimmt die PDUs entgegen und erzeugt aus ihnen die ursprüngliche Nachricht. Über eine Pipe (Klasse *pipe_*) (siehe Abb. 17) wird die Strukturinformation der Nachricht und ihre Adresse im gemeinsamen Speicherbereich an den Prozeß, der die Tupelblöcke bearbeitet, geschickt. Nach der Bearbeitung der Tupel wird der Ergebnistupelblock an den nächsten Prozeß versendet. Die Abbildung berücksichtigt nicht die Realisierung des Sliding window-Protokolls bei der Verwendung unsicherer Transportprotokolle.

5.4. Das Tuple Block Presentation Protocol (TBPP)

Das **T**uple **B**lock **P**resentation **P**rotocol (TBPP) stellt die Funktionen für eine rechnerunabhängige Datendarstellung der Tupelblöcke zur Verfügung. Die C++-Klasse

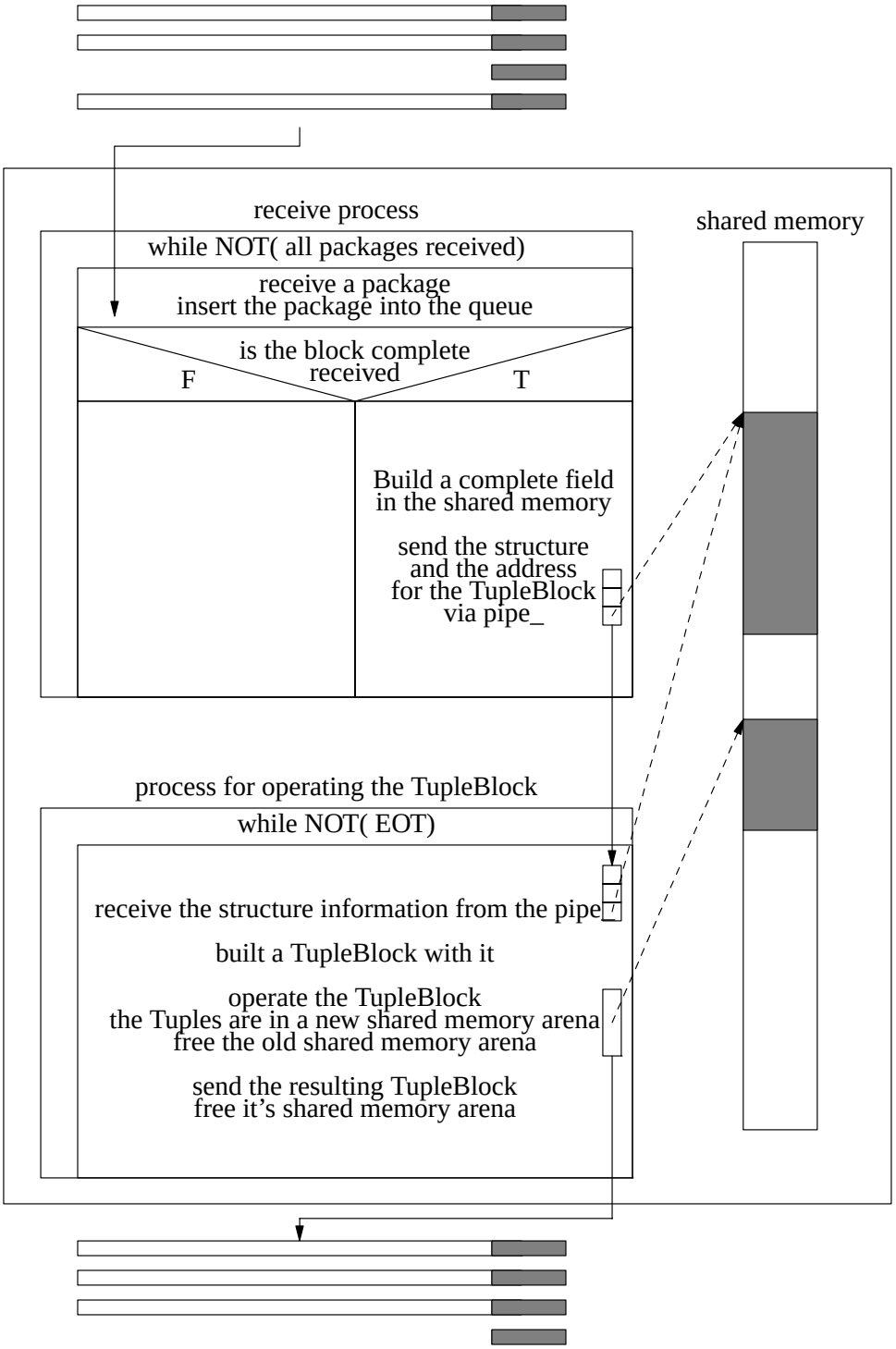


Abb. 20: Die Übertragung von Tupelblöcken

TB_presentation enthält die Methoden für die Transformation von Tupelblöcken zwischen rechnerinterner und rechnerunabhängiger Datendarstellung:

```
class TB_presentation {
      protected:
      TupleBlock *htonTB( TupleBlock *TB, qLATTR *qla);
      TupleBlock *ntohTB( TupleBlock *TB, qLATTR *qla);
};
```

Die Aufgaben der Methoden sind:

- *TB_presentation::htonTB(TB,qla)* wandelt die Tupelwerte eines Tupelblocks von einer rechnerinternen Darstellung in eine rechnerunabhängige um.
- *TB_presentation::ntohTB(TB,qla)* wandelt die Tupelwerte eines Tupelblocks von einer rechnerunabhängigen Darstellung in eine rechnerinterne um.

5.5. Das Mmap-Protokoll

Aus Effizienzgründen wird bei der Tupelblockübertragung zwischen Prozessen auf einem Rechnerknoten nicht die gesamte Protokollhierarchie durchlaufen. Als Kommunikationsmedium wird ein virtueller Shared memory-Bereich verwendet. Die Methoden für die Kommunikation über Memory mapped files enthält die Klasse *link_stream*:

```
class ipc_stream:link_stream      {
      public:
            ipc_stream( ipc *ip);
            ~ipc_stream();
      long  sendMMAP( long TupCnt, long TupSize, char *TupleBlockAddr);
      long sendEOT_MMAP();
      void receivePACK( pipe_ *P, mmem *mm);
};
```

Aufgrund der Verwendung von Memory mapped files werden während der Kommunikation Tupelblöcke weder in einen Kommunikationspuffer noch aus einem Kommunikationspuffer kopiert. Es werden also keine Datenfelder übertragen. Damit sind eine rechnerunabhängige Datendarstellung und ein Schiebefensterprotokoll unnötig.

Zwischen den Operationen werden durch ein Steuerfeld *MMAP* (siehe Abb. 19 und Tab. 6) die Adresse auf die Tupel des Tupelblocks (*TB* Address) und seine Strukturinformationen übertragen. Das Ende der Übertragung wird durch das Steuerfeld *EOT_MMAP* mitgeteilt.

Type	**Argument 1**	**Argument 2**	**Argument 3**
MMAP	Tuple Count	Tuple Size	TB Address
MMAP_EOT			

Tab. 6: Aufbau der Steuerblöcke des Mmap-Protokolls

Die folgenden Funktionen der Klasse *ipc_stream* erledigen diese Dienste:

- *ipc_stream::sendMMAP(TupCnt,TupSize,TupleBlockAddr)* versendet ein Steuerfeld *MMAP*.
- *ipc_stream::sendEOT_MMAP()* versendet ein Steuerfeld *MMAP_EOT* und beendet die Kommunikation.
- *ipc_stream::receivePACK(P,mm)* nimmt die PDUs entgegen. Durch den Empfang eines Steuerpaketes *MMAP* wird dem Empfänger der Zugriff auf den Tupelblock gestattet. Wie bei der Beschreibung für das BSP (Basic Stream Protocol) wird dem tupelmanipulierenden Prozeß die Adresse der Tupel des Tupelblocks und die Strukturinformation zur Erzeugung eines Objektes der Klasse *TupleBlock* übergeben.

5.6. Das Tuple Block Protocol (TBP)

Das **T**uple **B**lock **P**rotocol (TBP) stellt die Funktionen zur Übertragung von Tupelblöcken zwischen Operationen der Relationenalgebra bereit, die auf unterschiedlichen Rechnern in einem Rechnernetz oder auf demselben Rechner ausgeführt werden. Die Methoden für die Tupelblockübertragung enthält die Klasse *TB_stream*:

```
class TB_stream::public ipc_stream, public TB_presentation {
      public:
      TupleBlock  *receive( qLATTR *qla, pipe_ *P);
      void        send( TupleBlock *TB, qLATTR *qla,
                        where sendto=global, mmem *mm=NULL);
      void        sendEOT();
};
```

Die Klasse *TB_stream* realisiert die Dienste zum Empfang von Tupelblöcken *TB_stream::receive()*, zum Verschicken von Tupelblöcken *TB_stream::send()* und für das Mitteilen des Übertragungsendes.

Der Tupelblockaustausch erfolgt beim TBP:

- auf der Grundlage von BSP und TBPP, wenn die kooperierenden Operationen auf unterschiedlichen Rechnern (global) ausgeführt werden oder
- auf der Grundlage des Mmap-Protokoll, wenn die kooperierenden Operationen auf demselben Rechner (lokal) ausgeführt werden.

Der Nutzer wird von den Unterschieden zwischen der lokalen und der globalen Kommunikation befreit. Eine einwertige Operation der DRAM führt ihre Kommunikation, den Empfang und das Senden von Tupelblöcken folgendermaßen durch:

```
// Create the MMAP arena
mmem *mm = new mmem( MMAP_SIZE);

// Create, establish and accept (if connection oriented ipc)
// the incomming ipc
ipc *conn_in;
conn_in = ipc::establish( "sockudp");
if ( conn_in->create( PORT_IN) < 0) {
      /* ERROR: Can't create PORT_IN */
}
if ( conn_in->isconn() && ( conn_in = conn_in->accept())
      ;

// receive TupleBlocks
TB_stream *in_TBs = new TB_stream( conn_in, mm);

if ( conn_in->fork() == 0) {

      // start the receiver process
      in_TB->receivePACK();
      exit(0);
}
```

```
// process for manipulating/operating the TupleBlocks

// Establish and attach (connect) the outgoing ipc
ipc *conn_out;
conn_out = ipc::establish( "sockudp");
while ( conn_out->attach( PORT2, DESTINATIN_HOST) < 0)
      ;

// if the outgoing ipc uses a connectionless transport protocol, then
// start the Sliding Window Service
if ( NOT( conn_out->isconn())) {
      if ( conn_out->fork() == 0) {
            conn_out->sender();
            exit(0);
      }
}

TupleBlock *TB;

// Get the TupleBlock from the receive process
while ( (TB = conn_in->receive( AttributeList)) != EndOfTransmission) {

      // Manipulation of the TupleBlock
      ...

      // Send the TupleBlock
      if ( TupleBlock TB is Full) {
            conn_out->send( TB, AttributeList, global);
      }
}
// If there is an uncomplete not sended TupelBlock TB after
// leaving the loop:
// send the TB
if ( TupleBlock TB exists) {
      conn_out->send( TB, AttributeList, global);
}

// Send the End Of Transmission information
conn_out->sendEOT( global);

// Wait for the end of the sender process
```

```
// Close the connections
```

6. Realisierung der relationalen Algebraoperationen am Beispiel des Joins

6.1. Realisierungen der Join-Operationen in verteilten Systemen

In verteilten Systemen kann durch die parallelisierenden Operatoren Split und Merge (siehe Abb. 6 und 7) ein Join durch mehrere Teil-Joins berechnet werden. Während des lokalen Joins wird zu jedem Tupel der äußeren Relation ein Tupel der inneren Relation gesucht, so daß beide Tupel die Join-Bedingung erfüllen. Tupel der Relation *R* und der Relation *S* müssen demselben lokalen Join zugeteilt werden, wenn sie zusammen die Join-Bedingung erfüllen. Zur Berechnung des lokalen Joins kann ein Join-Verfahren wie Nested loop join, Sort merge join oder Hash join verwendet werden. Von diesen Join-Verfahren sind die Hash join-Verfahren am leistungsfähigsten [DeWi92a].

6.1.1. Der Simple hash join

Nachfolgend wird die zentralisierte Form des Simple hash joins [DeWi84] betrachtet:

- (1) Bucket-Erzeugungs-Phase (Bucket forming phase): Die Relation *R* wird mit Hilfe einer Hash-Funktion h_{part} partitioniert. Erfüllt das *i*-te Tupel r_i die Bedingung, um dem zur Zeit erzeugten Bucket† B_k anzugehören, wird es dort eingetragen. Aus den Tupeln des Buckets B_k wird im Hauptspeicher eine Hash-Tabelle HT_{join} erzeugt. Gehört r_i nicht in das Bucket B_k, so wird es in eine Überlaufrelation R' eingetragen. Auf diese Art und Weise werden Teilrelationen erzeugt, die im Hauptspeicher bearbeitet werden können.
- (2) Bucket-Join-Phase (Bucket joining phase): Die Hash-Funktion h_{part} wird ebenfalls auf die Tupel der Relation *S* angewendet. Mit der Hash-Funktion h_{join} wird überprüft, ob ein Tupel s_j mit Tupeln aus *R*, die sich im Bucket B_k befinden, Treffertupel bildet. Korrespondiert s_j nicht mit Tupeln im Bucket B_k, wird das Tupel in eine temporäre Relation *S'* eingetragen.
- Solange noch Überlaufpartitionen angelegt werden, wird mit *R'* und *S'* genauso verfahren.

† Ein Bucket [Schn91] ist ein zusammenhängender Adreßraum, der alle zu speichernden Tupel aufnimmt deren Hash-Attribut den selben Hash-Wert ergibt. Für jeden möglichen Hash-Wert existiert ein Bucket.

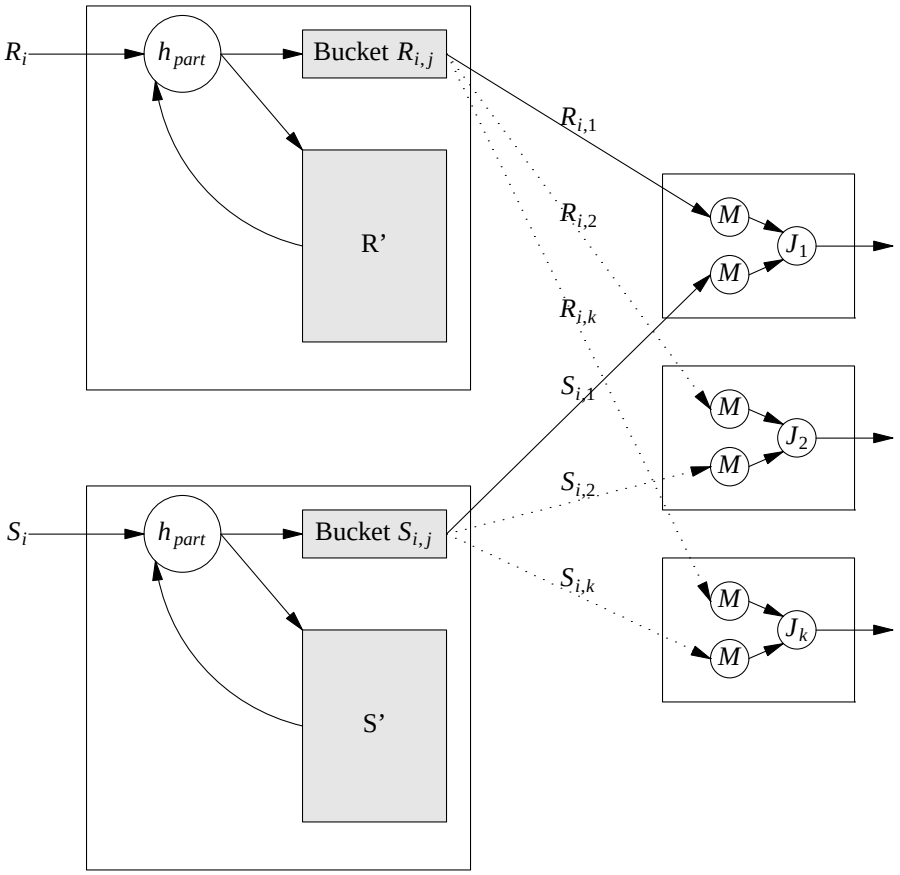


Abb. 21: Der parallelisierte Simple hash join

Beim Simple hash join erfolgt ein ständiger Wechsel zwischen Partitionierungs- und Join-Phase. Eine Parallelisierung kann dadurch erreicht werden, daß einerseits in der ersten Phase mittels Hashing die Tupel zu verschiedenen Ausführungsknoten geschickt werden, wo dann die zentralisierte Form des Simple hash joins die ankommenden Tupel verarbeitet (siehe Abb. 21). Andererseits kann sie durch die Überlappung von Partitionierung und Bucket-Erzeugungsphase des lokalen Joins (Pipelining) erreicht werden [DeWi92a].

Das Entstehen von Überlaufpartitionen bei der Ausführung des lokalen Joins kann dadurch vermieden werden, daß die Partition der Relation *R* kleiner als der zur Verfügung stehende Hauptspeicher auf dem Ausführungsknoten ist.

Mit der Partitionierung der Überlaufpartition *R'* kann erst begonnen werden, nachdem die Relation *R* komplett empfangen wurde und für die erste Partition alle Tupel ermittelt

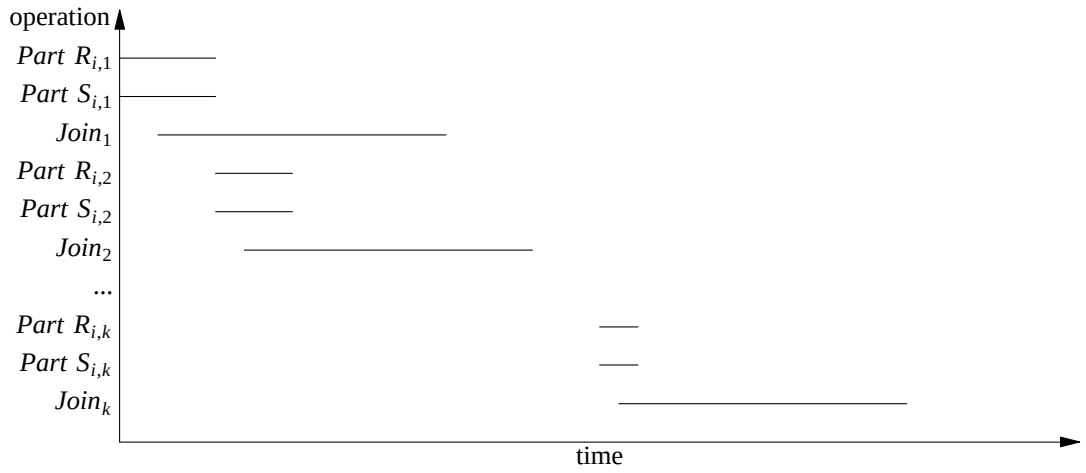


Abb. 22: Der parallelisierte Simple hash join

wurden. Der kontinuierliche Datenfluß wird dadurch unterbrochen. Die entstandenen Partitionen R_i und S_i können nun auf einem anderen Knoten verbunden werden, so daß die Teil-Joins untereinander parallel und parallel zur weiteren Partitionierung ausgeführt werden können. Dabei kann, bedingt durch das Pipelining, der Join zweier Teilrelationen mit deren Partitionierung überlappt ausgeführt werden (siehe Abb. 22).

6.1.2. Der Grace hash join

Die zentrale Form des Grace hash joins [Kits83] wird in drei Phasen ausgeführt:

- 1. Phase: Partitionierung von *R* durch Hashing über die Join-Attribute.
- 2. Phase: Partitionierung von *S* unter Nutzung derselben Hash-Funktion.
- 3. Phase: Verbinden der entsprechend zusammengehörenden Buckets.

Der Grace hash join unterscheidet sich vom Simple hash join im wesentlichen darin, daß die Partitionierungsphase völlig getrennt von der Join-Phase abläuft. Wird die innere Relation *R* in eine große Anzahl von Buckets eingeteilt, vermindert sich die Gefahr, daß ein Bucket die Speicherkapazität eines Prozessors überschreitet. In der dritten Phase werden die kleinen Buckets miteinander verknüpft (Bucket tunining), um eine optimale Bucket-Größe für die Join-Ausführung zu erhalten.

Die Parallelisierung des Grace hash joins [Schn89] wird durch eine Aufteilung der Buckets beider Eingangsrelationen auf unterschiedliche Rechnerknoten mit Hilfe einer

Split-Tabelle, die auf Hashing beruht, erreicht. Diese Buckets werden anschließend parallel verarbeitet. Die Tupel der Teilrelation R_i erzeugen eine Hash-Tabelle HT_{join}, die von den Tupeln der Teilrelation S_i genutzt wird, um Treffertupel zu finden (siehe Abb. 23).

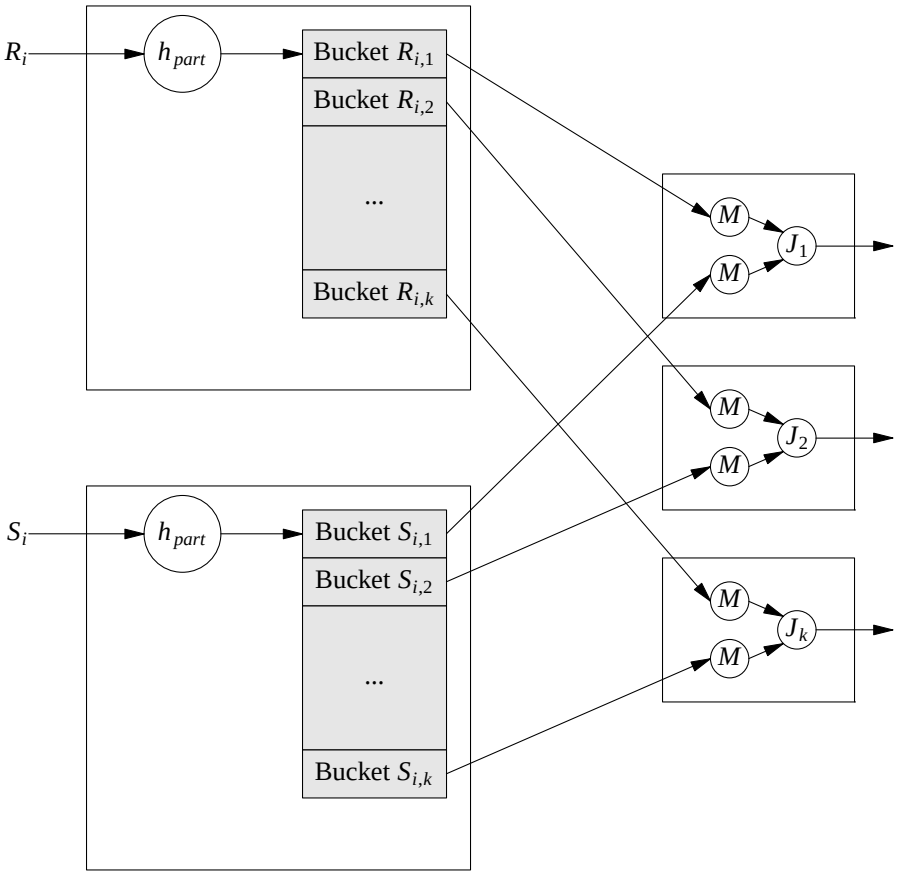


Abb. 23: Der parallelisierte Grace hash join

Das Bucket tuning der parallelen Form des Grace hash joins wirkt Pipelining und der partitionierten Ausführung entgegen. Zum einen muß die gesamte Eingangsrelation partitioniert werden, bevor die Tupel zu den Ausführungsknoten geschickt werden können. Dadurch ist keine Überlappung von Partitionierung und Hash-Tabellen-Aufbau der lokalen Joins möglich. Zum anderen kann keine gleichzeitige Partitionierung von Teilrelationen erfolgen, da das Zusammenfassen von Partitionen auf allen Partitionierungs-Ausführungsknoten gleichermaßen geschehen muß. Dafür wäre ein reger Informationsaustausch zwischen den Partitionierungsoperationen einer Relation notwendig, um überhaupt zu erfassen, ob eine Bucket-Zusammenlegung sinnvoll ist.

Darüber müßte eine zentrale Stelle entscheiden. Dieses Dilemma kann nur durch die Ausführung einer Partitionierungsoperation für jeweils eine Gesamtrelation beseitigt werden. Damit verhindert das Bucket tuning nicht nur ein kontinuierliches Pipelining, sondern auch partitionierte Parallelität. Die Überlappung von Partitionierung und Hash-Tabellenaufbau des lokalen Joins sowie die Möglichkeit der Partitionierung über Fragmente der Eingangsrelationen wird durch Verzicht auf das Bucket tuning erreicht (siehe Abb. 24).

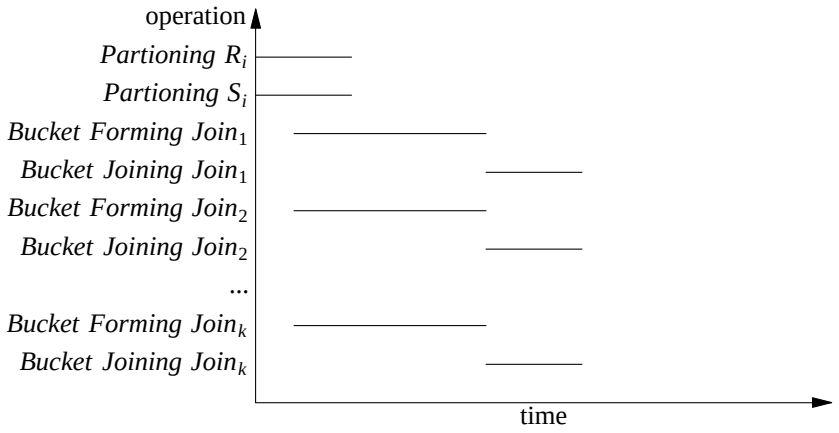


Abb. 24: Der parallelisierte Grace hash join

6.1.3. Der Hybrid hash join

Der zentrale Hybrid hash join-Algorithmus [DeWi84] wird ebenfalls in drei Phasen eingeteilt:

- 1. Phase: Partitionierung der inneren Relation *R*, wobei aus den Tupeln des ersten Buckets im Hauptspeicher eine Hash-Tabelle aufgebaut wird, während die restlichen Buckets in temporären Relationen auf dem Externspeichergerät gespeichert werden. Die Hash-Funktion h_{part} soll die Relation so partitionieren, daß jedes Bucket kleiner als der Hauptspeicher ist.
- 2. Phase: Die Relation *S* wird nach derselben Hash-Funktion h_{part}, mit der *R* aufgeteilt wurde, partitioniert. Korrespondieren Tupel von *S* mit Tupeln des Buckets von *R*, die sich in der Hash-Tabelle im Hauptspeicher befinden, dann werden mit Hilfe der Hash-Funktion Treffertupel aus der Hash-Tabelle ermittelt und Ergebnistupel gebildet. Andernfalls werden die Tupel in ihre Buckets auf dem externen

Speicher geschrieben.

- 3. Phase: Die auf dem externen Speicher untergebrachten Buckets werden entsprechend miteinander verbunden, indem mit den Tupeln des Buckets R_i eine Hash-Tabelle HT_{join} aufgebaut wird. Die Tupel des Buckets S_i ermitteln mit Hilfe der Hash-Funktion Treffertupel und erzeugen Ergebnistupel.

Jeder Join wird somit in mehrere Joins über Partitionen untergliedert, von denen jeder ohne Überlauf berechnet werden kann.

Die Parallelisierung wird dadurch erreicht, daß die Buckets nicht auf den externen Speicher geschrieben, sondern zu Ausführungsknoten gesendet werden, die die lokalen Joins berechnen. Für ein Bucket wird die Hash-Tabelle im Hauptspeicher des Ausführungsknotens aufgebaut, der die Partitionierung vornimmt. Der Vorteil des Hybrid hash joins, der Aufbau der Hash-Tabelle für ein Bucket während der Partitionierung von *R* und das Erzeugen von Ergebnistupeln während der Partitionierung von *S*, spielt keine Rolle, weil die Relationen *R* und *S* gleichzeitig partitioniert werden. Durch die parallele Partitionierung der Eingangsrelationen müssen die Tupel von S_i zu dem Ausführungsknoten geschickt werden, der die Partitionierung der Relation *R* vornimmt. Die Probing-Phase, bei der die Tupel aus S_i auf die Hash-Tabelle zugreifen, kann erst dann beginnen, wenn *R* komplett partitioniert wurde und die Hash-Tabelle somit komplett aufgebaut ist.

Nachfolgend werden die Fälle betrachtet, die bei der Ausführung des Hybrid hash joins auftreten können.

- (1) Wenn eine Fragmentierung beider Eingangsrelationen vorliegt, werden *R* und *S* auf mehreren Rechnern partitioniert. Auf den Rechnern, die die Partitionierung von *R* ausführen, werden die Tupel jeweils einer Partition in eine Hash-Tabelle im Hauptspeicher eingetragen. Die zu den Tupeln in den Hash-Tabellen korrespondierenden Tupel von *S* werden an alle Rechner, die Fragmente von *R* partitionieren, geschickt. Es werden demzufolge Duplikate jedes dieser Tupel erzeugt, versendet und verarbeitet (siehe Abb. 25).

- (2) Liegt im Gegensatz dazu die gesamte innere Relation für den Join vor, weil *R* entweder nicht fragmentiert ist oder die Fragmente zusammengefaßt wurden, wird die Partitionierung der Relation *R* nur auf einem Rechner vorgenommen. Damit

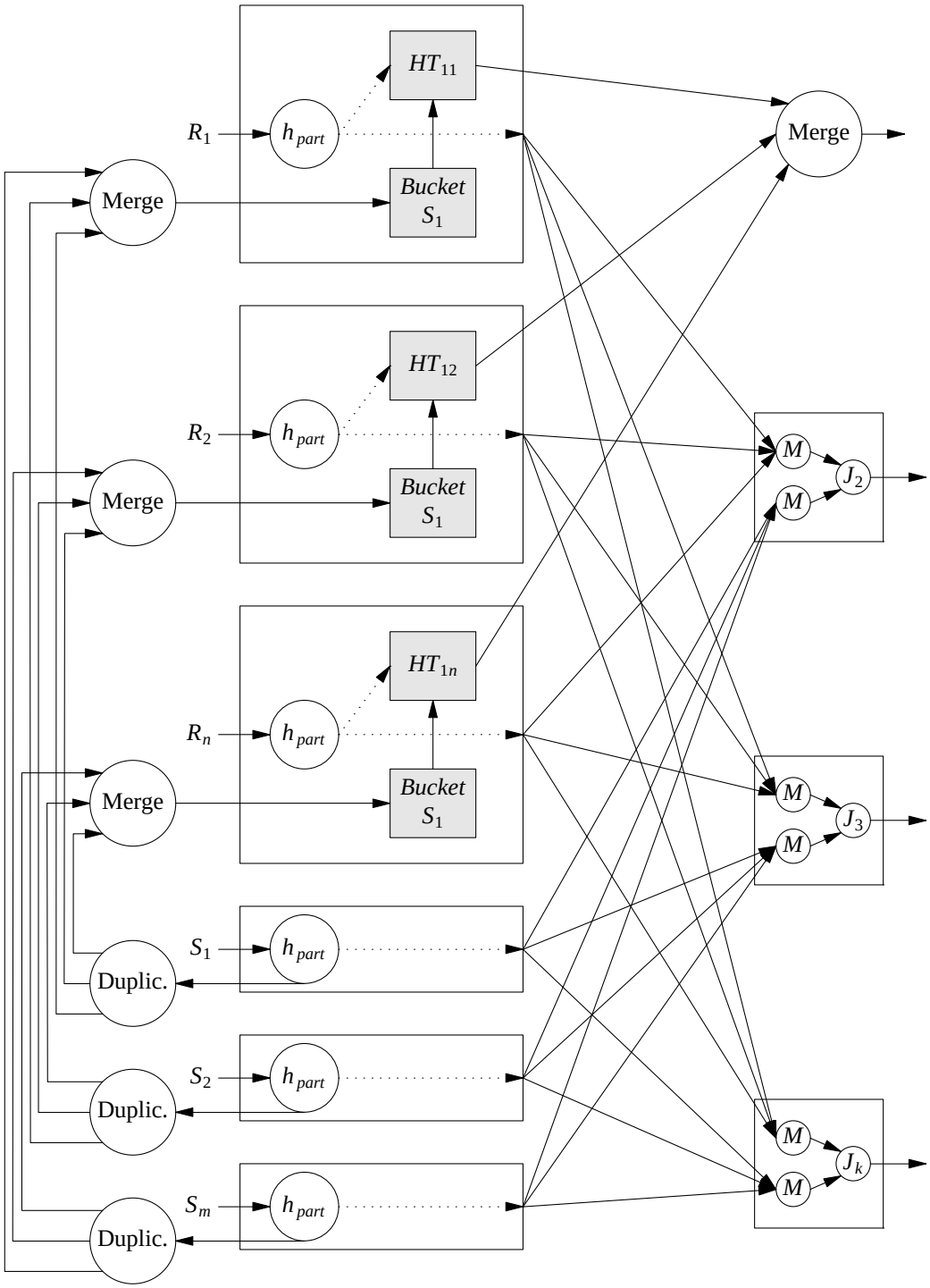


Abb. 25: Der parallelisierte Hybrid hash join, mit fragmentierten Eingangsrelationen

wird nur eine Hash-Tabelle mit den Tupeln einer Partition von *R* auf dem *R*-partitionierenden Rechner angelegt. Die Tupel von *S*, die zu den Tupeln in der Hash-Tabelle korrespondieren, werden nur an einen Rechner geschickt, wodurch

die Duplikatbildung und die mehrfache Verarbeitung ein und desselben Tupels entfällt. Muß die Zusammenfassung zur Gesamtrelation *R* erst vorgenommen werden, ist zu überprüfen, ob die Leistungsfähigkeit des Grace hash joins übertroffen wird. Wenn *S* ebenfalls fragmentiert vorliegt, kann die Partitionierung von *S* parallel über die Fragmente erfolgen (siehe Abb. 26).

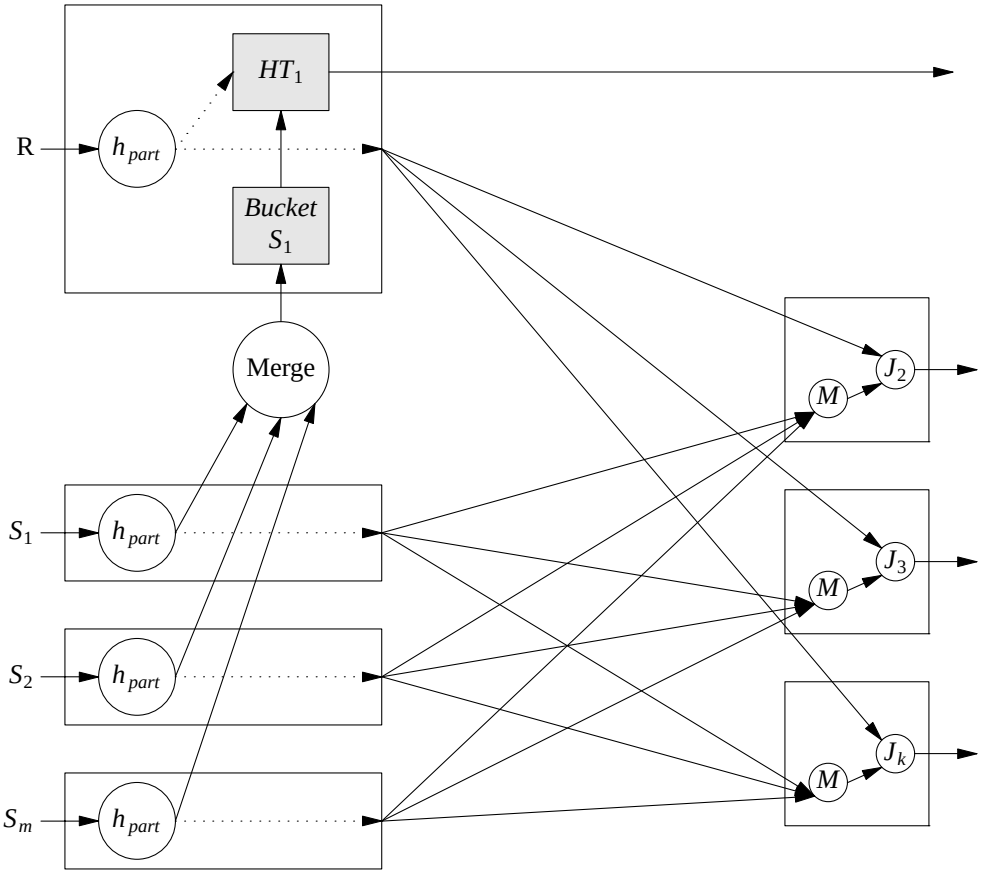


Abb. 26: Der parallelisierte Hybrid hash join mit Partitionierung der Gesamtrelation *R*

- (3) Wird nicht nur die gesamte Relation *R*, sondern auch die gesamte Relation *S* durch jeweils eine Partitionierungsoperation auf die einzelnen Rechnerknoten aufgeteilt, werden weniger Kommunikationsverbindungen erforderlich.
- (4) Erfolgt die Partitionierung von *R* und *S* nacheinander durch einen einzigen Partitionierungsoperator, kann *S* erst partitioniert werden, nachdem *R* vollständig partitioniert ist. Dadurch ist ein Überlappen der Partitionierung von *R* und *S* nicht möglich.

In allen genannten Fällen kann das erste Ergebnistupel erst nach der vollständigen Partitionierung von *R* erzeugt werden. In den ersten drei Fällen kann davon ausgegangen werden, daß ein Teil der zu den entsprechenden Buckets von *R* korrespondierenden Tupel der Relation *S* bereits vorliegt, wenn die Aufteilung von R abgeschlossen ist. Somit können sofort nach Beendigung der Partitionierung von *R* Ergebnistupel erzeugt werden. Im letzten Fall wird nach dem Aufbau der Hash-Tabellen auf den Join-Ausführungsknoten und im Hauptspeicher des Rechnerknoten, der die Partitionierung von *R* vorgenommen hat, mit der Partitionierung von *S* begonnen. Dadurch wird die Erzeugung der ersten Ergebnistupel verzögert. Die zu bearbeitenden Tupel von *S* werden nur nach und nach den entsprechenden Ausführungsknoten zugeordnet. Bei den ersten drei Fällen kann davon ausgegangen werden, daß mehrere Tupel auf allen Ausführungsknoten nach dem Erzeugen der Hash-Tabelle bereits zur Bearbeitung vorliegen. Damit liefert die Variante (4) auch das letzte Ergebnistupel später als die anderen Varianten, weil der Datendurchsatz im Vergleich zu den anderen Varianten bedeutend geringer ist. Der hohe Kommunikationsaufwand der ersten Version, der durch die vielen Kommunikationsverbindungen und das mehrfache Verschicken ein und desselben Tupels des Buckets S_i an die verschiedenen Partitionierungsoperatoren von *R* verursacht wird, sowie die zusätzlichen Kosten, die aus dem mehrfachen Verarbeiten dieser Duplikate anfallen, lassen auch die Variante (1) ungünstig erscheinen. Das Verlagern der Teil-Joins, die durch die Partitionierungsoperatoren berechnet werden, auf eine separate Operation führt zur Vermeidung der Übertragung und Verarbeitung von Duplikaten. Diese Vorgehensweise entspricht dem Grace hash join (vergleiche Abb. 23 und Abb. 25).

Die Fälle (2) und (3), bei denen die Gesamtrelation *R* durch einen einzigen Partitionierungsoperator auf die Buckets aufgeteilt wird, vermeidet das Aufblähen der Anzahl der zu verschickenden und zu verarbeitenden Tupel der Relation *S* durch Duplikatbildung (im Vergleich zu (1)). Die Fälle (2) und (3) erzeugen eher als der Fall (4) ihre Ergebnistupel, weil die miteinander zu verbindenden Tupel beider Eingangsrelationen zu einem früheren Zeitpunkt vorliegen. Die Partitionierung von *S* und somit in den meisten Fällen der gesamte Join ist durch das gleichzeitige Partitionieren von *R* und *S* zu einem früheren Zeitpunkt abgeschlossen als bei der Version (4).

6.1.4. Vergleich der drei Hash join-Verfahren

Bei der Betrachtung der Leistungsfähigkeit der Join-Verfahren wird im folgenden davon ausgegangen, daß die beiden Eingangsrelationen *R* und *S* gleichzeitig partitioniert werden.

Die gleichzeitige Partitionierung von Teilrelationen der jeweiligen Eingangsrelationen wird vom Simple hash join und Grace hash join unterstützt. Für den Hybrid hash join müssen die Eingangsströme zumindest von der inneren Relation vorher zusammengefaßt werden, falls *R* fragmentiert vorliegt.

Beim Simple hash join findet zwar eine Überlappung der Partitionierungsphase mit der Bucket building-Phase der lokalen Joins statt, aber die Partitionierung in die einzelnen Buckets erfolgt sequentiell (siehe Abb. 22).

Durch die gleichzeitige Ausführung der Partitionierung von *R* und *S* entfällt der Vorteil des Hybrid hash joins - die Ausführung der Join-Phase für ein Bucket zur Zeit der Partitionierung von S, die nach abgeschlossener Partitionierung von *R* erfolgt - in der Datenflußumgebung von HEAD. Dadurch ähnelt der Hybrid hash join-Algorithmus dem des Grace hash joins. Der einzige Unterschied besteht darin, daß auf dem Ausführungsknoten, auf dem *R* partitioniert wird, auch ein Teil-Join stattfindet.

6.1.5. Weitere parallele Hash join-Verfahren

Für die Ausführung von Join-Operationen werden die Tupel zur Ausführung zu verschiedenen Rechnerknoten geschickt, um kleinere Teil-Joins gleichzeitig ausführen zu können. Von den Teilrelationen der inneren Relation *R* wird auf den Rechnerknoten eine Hash-Tabelle aufgebaut. Sind diese Teilrelationen größer als der zur Verfügung stehende Hauptspeicher, muß eine Überlaufbehandlung erfolgen. Durch eine geeignete Partitionierung werden Teilrelationen der inneren Relation erzeugt, die vom Hauptspeicher aufgenommen werden können.

Wie weiter oben bereits gezeigt, versagt die Hash-Partitionierung beim gehäuften Auftreten eines Attributwertes, weil bei jeder Hash-Funktion immer eine Zuordnung zum selben Bucket erfolgt. In diesem Fall kann die Bereichspartitionierung zur Aufteilung der Relation *R* in gleichgroße Teilrelationen herangezogen werden. Dafür sind statistische Informationen über die Verteilung der Attributwerte erforderlich, nach denen der

Partitionierungsvektor zur Aufteilung der Tupel auf die Buckets ermittelt wird. Die Scan-Operation kann beim Einlesen der Relation vom Externspeichergerät diese Informationen für die erforderlichen Attribute sammeln. Aufgrund der zu erwartenden Selektivität vor der Partitionierung, wird die Verteilung des Attributwertes zum Zeitpunkt der Partitionierung bestimmt. Erfolgt eine Partitionierung über Fragmente, ermittelt ein Koordinator den Partitionierungsvektor entsprechend der Verteilung des Attributwertes in jedem Fragment.

Eine andere Methode ist die Ermittlung statistischer Informationen für die Erzeugung des Partitionierungsvektors während der Partitionierung [DeWi92b]. Bevor das erste Tupel an einen Join-Ausführungsknoten zugeteilt werden kann, muß die gesamte Relation vorliegen. Das hält den kontinuierlichen Datenfluß während der Anfragebearbeitung auf. Eine Überlappung von Partitionierungsphase und Hash-Tabellenaufbau auf den lokalen Join-Ausführungsknoten ist auf diese Art und Weise nicht möglich.

Eine weitere Verbesserung der Leistungsfähigkeit von Joins bei Data skew bieten die adaptiven, lastbalancierenden, parallelen Hash joins [Zell90, Hua91], bei denen eine Balancierung der Last auf alle Ausführungsknoten entsprechend ihrer Belastung eine hohe Systemleistungsfähigkeit gewährleistet. Der Ansatz zu diesen Verfahren besteht in einer derartigen Partitionierung der Relation *R*, daß bedeutend mehr Buckets als Ausführungsknoten zur Verfügung stehen. Die unterschiedlich großen Buckets werden zu annähernd gleichgroßen Buckets zusammengefügt.

Das geschieht beim Adaptive load balancing parallel hash join [Hua91] folgendermaßen:

(1) Split-Phase: Partitionierung der Eingangsrelationen in bedeutend mehr Buckets als Ausführungsknoten existieren. *R* und *S* können parallel partitioniert werden. Die Zuweisung der Buckets zu den Ausführungsknoten erfolgt über Round robin-Verfahren.

(2) Partitioning tuning Phase: Diese Phase läuft auf jedem lokalen Ausführungsknoten ab und ist untergliedert in:

> (a) Bucket partitioning: Die Buckets, die kleiner als der Hauptspeicher sind und damit zum Überlauf führen, sowie der noch freie Speicher werden ermittelt.

(b) Bucket relocation: Die in (2.a) ermittelten Informationen werden an einen Koordinator gesendet, der die übergelaufenen Buckets an weniger belastete Ausführungsknoten zuweist. Die übergelaufenen Buckets von *R* werden mit ihren korrespondierenden Buckets von *S* zu den ermittelten Knoten geschickt.

(3) Bucket tuning-Phase: Die kleinen Buckets auf den Ausführungsknoten werden zu optimalen Buckets zusammengefaßt.

(4) Join-Phase: Ausführung der lokalen Joins.

Um die Kommunikationskosten zu reduzieren, sollten große Buckets auf einem Knoten gehalten werden und kleine Buckets neu verteilt werden. Ziel sind Buckets, die der Rechnerlast und dem Grad des Data skew angepaßt sind.

Bei extremen Data skew versagt dieser Parallele Hash join für adaptive Lastbalancierung. Der Erweiterte Parallele Hash join für adaptive Lastbalancierung [Hua91] umgeht einerseits den einseitigen Kommunikationsaufwand, der bei extremen Data skew auftritt, andererseits verteilt er die Buckets ausgeglichen auf die Ausführungsknoten. Dafür wird das Senden von Tupeln auf die Partitioning tuning-Phase verlagert. Dort werden die Buckets in absteigender Reihenfolge ihrer Größe nach sortiert. In sortierter Reihenfolge werden die Buckets anschließend den Ausführungsknoten zugewiesen. Bei einer ausreichend hohen Belastung eines Ausführungsknotens werden ihm keine Buckets mehr zugeteilt. Die Bucket tuning-Phase faßt die Buckets auf den Ausführungsknoten zu optimalen Buckets zusammen, die in der Join-Phase mit ihren korrespondierenden Buckets der Eingangsrelation *S* verbunden werden. Bei diesem Join werden die Tupel nur noch zu den Ausführungsknoten geschickt, es erfolgt kein Austausch mehr zwischen den Join-Ausführungsknoten.

Partitionierung und lokaler Join können nicht überlappt ausgeführt werden. Das beeinträchtigt das kontinuierliche Pipelining. Wenn die Buckets aufgrund von Data skew unterschiedlich groß sind, kommt beim Parallelen Hash join für adaptive Lastbalancierung ein beträchtlicher Kommunikationsaufwand durch den Austausch von Buckets zwischen den Join-Ausführungsknoten hinzu. Zwar erlaubt der Parallel Hash join für adaptive Lastbalancierung das fragmentierte Partitionieren der Eingangsrelationen, jedoch benötigt er einen zusätzlichen Koordinator. Beim Erweiterten Parallelen Hash join für adaptive Lastbalancierung kann keine fragmentierte Partitionierung

erfolgen, weil bereits in der Split-Phase die gesamten Eingangsrelationen für die Aufteilung betrachtet wird.

Die Berücksichtigung verfügbarer Ressourcen der Ausführungsknoten ermöglicht auf der einen Seite eine ausgeglichene Belastung der Ausführungsknoten, auf der anderen Seite kann auf die Ausführung anderer Anwendungen auf einem Rechnerknoten wie etwa CAD-Systeme reagiert werden. In der HEAD-Umgebung sind die beiden Parallelen Hash join für adaptive Lastbalancierung uninteressant. Beim Erweiterten Parallelen Hash join-Verfahren für adaptive Lastbalancierung ist keine parallele Partitionierung möglich†. Der Parallele Hash join für adaptive Lastbalancierung bewirkt einen hohen zusätzlichen Kommunikationsaufwand. Beide beeinträchtigen den kontinuierlichen Datenfluß und somit die vertikale Parallelität.

Ein anderer Ansatz für die Behandlung von Data skew ist in [Wolf93] zu finden. Während der Partitionierungsphase erfolgt ein Hashing in zwei Stufen, einem groben und einem feinen Hashing. Das Ergebnis sind zusammengesetzte Hash-Klassen. Dabei werden Informationen gesammelt, die für die hinzugefügte Scheduling-Phase notwendig sind. Sie wird durch einen Koordinator ausgeführt, der von allen Partitionierungsphasen die Daten über die Bucket-Einteilung entgegennimmt und durch einen heuristischen Optimierungsalgorithmus die Arbeit der Join-Phase in unabhängige Aufgaben einteilt. Jede dieser unabhängigen Aufgaben setzt sich aus mehreren Teilaufgaben zusammen, die auf den zusammengesetzten Hash-Klassen beruhen. Der Koordinator teilt die Teilaufgaben den Prozessoren so zu, daß die Last während der Join-Phase annähernd balanciert ist. Diese Zuteilung wird den Prozessen mitgeteilt, die die Partitionierung durchführen. Damit kann die Transfer-Phase mit der Übertragung der Tupel an die entsprechenden Join-Ausführungsknoten beginnen. Ein Bucket, das durch grobes Hashing entstanden ist, wird mehreren Prozessoren zugewiesen, wenn es für die Ausführung auf einem Prozessor zu groß ist. In diesem Fall werden die Buckets, die sich aus dem feinen Hashing ergeben den Prozessoren zugeteilt. An die Transfer-Phase schließt sich die Join-Phase an.

Bei diesem Verfahren wird der kontinuierliche Datenfluß während der Hash- und Scheduling-Phase aufgehalten. Der zusätzliche Kommunikationsaufwand ist gering.

† Gilt nur für fragmentierte Eingangsrelationen!

Fragmentierte Partitionierung ist möglich, die innere und die äußere Relation können gleichzeitig partitioniert werden. Bei extremen Data skew erfolgt durch die Hash-Klassen eine Einteilung in gleichgroße Buckets, wodurch die Last der Join-Phasen auf die Rechner balanciert verteilt wird.

6.2. Implementierung paralleler Join-Operationen

Die den Datenfluß in HEAD unterstützenden relationalen Algebraoperationen nutzen für die Übertragung von Tupeln zwischen den Operationen des Anfrageausführungsplans die von der Klasse *TB_stream* zum Empfangen und Senden von Tupelblöcken bereitgestellten Operationen. Für die Verarbeitung eines Tupels in einem Tupelblock und für das Erzeugen von Ergebnistupelblöcken werden die Funktionen der Klassen *Tuple* und *TupleBlock* genutzt. Die Abbildung 27 stellt die implementierte Klassenhierarchie zur Realisierung der datenflußgesteuerten relationalen Algebraoperationen in HEAD [Krus94] dar. In diesem Abschnitt wird jedoch nur auf die für die Ausführung einer verteilten Join-Operation relevanten Klassen eingegangen.

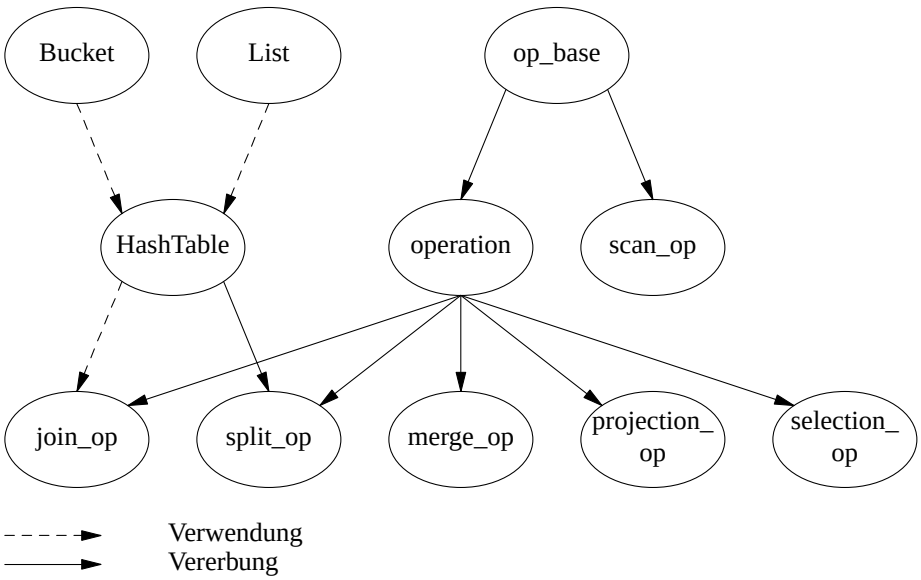


Abb. 27: Klassenhierarchie der erweiterten relationalen Algebraoperationen

Für die Speicherung von Tupelblöcken wird der virtuelle Shared memory genutzt, so daß das System die Seitenauslagerung vornimmt. Die Bereitstellung des Mmap-Bereiches und dessen Freispeicherverwaltung wird durch die Klasse *mmem* realisiert.

6.2.1. Die Basisklassen

Ein einheitliches Schema der DRAM-Operationen wird durch die Klassen zur Initialisierung der Tupelströme *op_base* und *operation* erreicht:

```
typedef TupelBlock  *TB_ptr;
typedef Tupel       *T_ptr;
typedef TB_stream   *TB_stream_ptr;
typedef qLATTR      *qLATTR_ptr;
typedef enum where  *where_ptr;
```

```
class op_base {
      public:
            op_base(   mmem *m, qLATTR *qla,
                       TB_stream_ptr *TBs, where_ptr s,
                       int streams = 1);
            ~op_base();
};
```

```
class operation:public op_base {
      public:
            operation(  mmem *m, qLATTR_ptr *qla, qLATTR *qla_out,
                       TB_stream_ptr *TBs, TB_stream_ptr *TBs_out,
                       where_ptr s_out, int streams = 1, int streams_out = 1);
            ~operation();
};
```

Die Basisklassenkonstruktoren realisieren das Anlegen der Puffer für den Empfang (*operation::operation()*) und für das Senden (*op_base::op_base()*) sowie notwendige Initialisierungen.

Dem Konstruktor *op_base::op_base(m,qla,TBs,s,streams)* wird für die Initialisierung der herausgehenden Tupelströme der Operation folgendes übergeben:

- der Zeiger *m* auf den zuvor angelegten Mmap-Bereich, der durch den Prozeß genutzt wird,

- der Zeiger *qla* auf ein Feld von Attributlisten für herausgehende Tupelströme,
- der Zeiger *TBs* auf ein Feld von Zeigern, das die herausgehenden Tupelströme (*TB_stream*) enthält,
- der Zeiger auf ein Feld, das für jeden herausgehenden Tupelstrom angibt, ob eine lokale Kommunikation über den Mmap-Bereich oder eine globale Kommunikation durch Message passing erfolgt und
- die Anzahl der herausgehenden Tupelströme *streams* (Default-Wert: 1).

Neben den Initialisierungsparametern für die herausgehenden Tupelströme (*qla_out*, *TBs_out*, *s_out*, *streams_out*), sowie den Zeiger auf den Mmap-Bereich (*m*), die der Konstruktor *operation::operation(m, qla, qla_out, TBs, TBs_out, s_out, streams, streams_out)* an den Konstruktor seiner Basisklasse *op_base* weitergibt, verwendet der Konstruktor *operation::operation()* für die Initialisierung der eingehenden Tupelströme:

- den Zeiger *qla* auf ein Feld von Attributlisten der eingehenden Tupelströme,
- den Zeiger *TBs* auf ein Feld von Zeigern, das die eingehenden Tupelströme enthält und
- die Anzahl der eingehenden Tupelströme *streams* (Default-Wert: 1).

6.2.2. Die Klassen zur Realisierung der parallelisierenden Operationen

6.2.2.1. Der Mischoperator Merge

Die Klasse *merge_op* realisiert den Mischoperator Merge:

```
class merge_op:public operation {
      public:
            merge_op( mmem *m, qLATTR_ptr *qla_in, qLATTR *qla_out,
                        TB_stream_ptr *TBs_in, TB_stream_ptr *TBs_out,
                        where_ptr s_out, int streams_in,
                        int streams_out = 1);
      void  merge();
};
```

An den Konstruktor *merge_op::merge_op()* wird eine Eingabe-Attributliste

und eine damit übereinstimmende Ausgabe-Attributliste, ein Feld von Zeigern auf eingehende Tupelströme sowie ein Zeiger auf einen herausgehenden Tupelstrom übergeben. Weiterhin erhält der Konstruktor die Information darüber, ob der nachfolgende Prozeß auf demselben oder auf einem anderen Rechner abgearbeitet wird, und wieviele eingehende Tupelströme durch *merge_op::merge()* zu einem herausgehenden Tupelstrom zusammengefaßt werden.

6.2.2.2. Der Partitionierungsoperator Split

Der Partitionierungsoperator wird durch die Klasse *split_op* realisiert:

```
typedef void * vector_t;

class split_op:public operation, public HashTable {
      public:
            split_op( mmem *m, qLATTR_ptr *qla_in, qLATTR *qla_out,
                        TB_stream_ptr *TBs_in, TB_stream_ptr *TBs_out,
                        where_ptr s_out, int streams_in, int streams_out);
      void  split11( qATTR *SplitAttr);
      void  split12( qATTR *SplitAttr, vector_t SplitVector, int VectorSize);
};
```

An den Konstruktor der Klasse zur Realisierung des Partitionierungsoperators

split_op::split_op(m,qla_in,qla_out,TBs_in,TBs_out,s_out,streams_in,streams_out)

wird eine eingehende und eine damit übereinstimmende herausgehende Attributliste, ein Zeiger auf einen eingehenden Tupelstrom und ein Feld von Zeigern auf herausgehende Tupelströme übergeben. Weiterhin erhält der Konstruktor ein Feld mit den Angaben darüber, ob die entsprechenden

nachfolgenden Operationen auf demselben oder auf einem anderen Rechnerknoten abgearbeitet werden, die Anzahl der eingehenden Tupelströme und die Anzahl der herausgehenden Tupelströme.

Der Partitionierungsoperator nimmt die Tupelblöcke des eingehenden Tupelblockstroms entgegen und berechnet den herausgehenden Tupelstrom für die Tupel:

- bei der Hash-Partitionierung *split_op::split11(SplitAttr)* durch zweistufiges Hashing [Wood93], bezogen auf den Wert des Attributes SplitAttr und
- bei der Bereichspartitionierung *split_op::split12(SplitAttr,SplitVector)* durch die Einordnung des Attributwertes entsprechend dem Split-Vektor.

Bei der Hash-Partitionierung muß bei einem nicht-kardinalen Typ des zu verbindenden Attributes der Attributwert auf sinnvolle Weise in einen kardinalen Wert umgewandelt werden. Hierfür wird das weiter unten beschriebene *HashTable::StrToKey()* verwendet.

6.2.3. Der lokale Join

Die Join-Operation ist im Vergleich zu den anderen Operationen der DRAM am komplexesten. Die Klasse *join_op* stellt die Funktionen zur Ausführung hashbasierter Joins bereit, und wird daher nur für Equi-Joins verwendet.

6.2.3.1. Die Realisierung des Hashing

Das schnelle Auffinden von Treffertupeln basiert beim Hash join auf der Verwendung von Hash-Tabellen. Die Klassen *List*, *Bucket* und *HashTable* realisieren das Hashing:

```
struct List_T {
      Tupel *T;
      List_T *next;
};

class List {
      public:
            List();
            ~List();
      void  insert( Tupel *Tup);
      void  kill( Tupel *Tup);
      Tupel *First();
      Tupel *Next();
};



class Bucket {
      protected:
            unsigned short   LocalDeep;
            unsigned short   EntryCount;
            unsigned long    Key[BucketSize];
            List     *Entry[BucketSize];

      public:
            Bucket( unsigned short LocalDeepInit);
            ~Bucket();
      unsigned short   GetLocalDeep();
      short unsigned   GetEntryCount();
      unsigned long    GetKey( unsigned short index);
      List     *GetEntry( unsigned short index);
      bool     IsFull();
      void     InsertEntry( unsigned long key, Tupel *entry);
      void     InsertEntry( unsigned long key, List *entry);
};
```

```
class HashTable {
      private:
            unsigned short GlobalDeep;
            typedef Bucket *Bucket_ptr;
            Bucket_ptr *Table;
      protected:
            unsigned long KeyToTableIndex( unsigned long key);
      public:
            HashTable();
            ~HashTable();
      unsigned long StrToKey( char *, size_t);
      bool       Insert( unsigned long key, Tupel *Entry);
      List       *Get( unsigned long key);
      bool       Kill( unsigned long key, List *Entry);
};
```

Zum schnellen Auffinden von Tupeln während der Probing-Phase des lokalen Joins werden die eintreffenden Tupel der inneren Relation in eine Hash-Tabelle eingetragen. Über den Hash-Schlüssel, sinnvollerweise das Join-Attribut, kann auf diese Weise ein bestimmtes Tupel mit geringem Suchaufwand ermittelt werden. Dafür wird ein Hash-Verfahren [Wood93, Mehl88, Brat84] benötigt, das unabhängig von der Verteilung des Hash-Attributes zu guten Zugriffszeiten führt und den zur Verfügung stehenden Speicher effektiv nutzt.

Hash-Verfahren werden entsprechend der Zuweisung des Hash-Bereiches in statisches und dynamisches Hashing [Lock87] eingeteilt.

Beim statischen Hashing wird nach Verfahren ohne oder mit Kollisionsbehandlung unterschieden. Verfahren ohne Kollisionsbehandlung setzen eine dichte Hash-Schlüsselmenge voraus, wie sie zum Beispiel durch die Vergabe von fortlaufenden Nummern bei Primärschlüsseln entstehen. Die Attributwerte sind im allgemeinen jedoch nicht gleichverteilt und bilden nur eine kleine Teilmenge der möglichen Schlüsselmenge. In solch einem Fall führt das statische Hashing ohne Kollisionsbehandlung zu einer geringen Auslastung des zur Verfügung stehenden Speicherbereiches und zum Überlauf bei gleichen Schlüsselwerten. Durch ein Hash-Verfahren mit Kollisionsbehandlung werden mehrere Tupel mit dem gleichen Hash-Wert in einem Bucket zusammengefaßt,

wodurch der bereitgestellte Speicherbereich besser ausgelastet wird und Kollisionen vermieden werden.

Die statische Bereitstellung des Hash-Bereiches führt bei zu großen Hash-Tabellen - bezogen auf die einzutragenden Tupel - zu ineffizientem Umgang mit Speicher. Bei einer zu kleinen Dimensionierung der Hash-Tabelle werden Überlaufbereiche angelegt, die zu ungünstigen Zugriffszeiten auf die einzelnen Tupel führen. Bei Verfahren mit dynamischen Hash-Bereichen schrumpft und wächst der Hash-Bereich entsprechend der Tupelanzahl, so daß keine Überlaufbereiche entstehen. Die Belegung des bereitgestellten Bereiches ist unabhängig von der Dichte und des Wachstums der Schlüsselmenge immer konstant.

Damit erfüllen dynamische Hash-Verfahren die Anforderungen für die Realisierung des lokalen Joins am besten. Für Hash-Verfahren mit dynamischem Hash-Bereich gibt es verschiedene Ansätze: Erweiterbares Hashing, virtuelles Hashing und dynamisches Hashing [Lock87].

Das Erweiterbare Hashing [Mehl88] ist von diesen Verfahren am einfachsten zu implementieren. Gegenüber einer ungleichen Schlüsselverteilung besitzt es die geringste Störanfälligkeit [Lock87]. Die Hash-Tabelle besteht aus einem Feld von 2^d-Bucket-Adressen. Anhand der ersten d Bits eines Schlüssels, der globalen Tiefe, wird der Hash-Wert bestimmt. Der Tabelleneintrag für den Hash-Wert ist die Adresse des zugehörigen Buckets.

Zu jedem Bucket gehört die Information über die lokale Tiefe *d'*, welche angibt, auf wieviel Bits des Hash-Wertes die Einträge im Bucket basieren. Die lokale Tiefe kann maximal genauso groß wie die globale Tiefe sein ($d' \leq d$). Soll ein Tupel in ein bereits volles Bucket eingetragen werden, wird das Bucket aufgespalten. Die lokale Tiefe des neuentstandenen Buckets wird inkrementiert. Die Tupel werden entsprechend des Hash-Wertes ihres Schlüssels den Buckets neu zugeteilt. Ist die lokale Tiefe des zu spaltenden Buckets gleich der globalen Tiefe, muß die globale Tiefe vor dem Aufspalten des Buckets inkrementiert werden, wodurch die Hash-Tabelle verdoppelt wird.

Die Klassen *List*, *Bucket* und *HashTable* realisieren das Erweiterbare Hashing in der DRAM. Beim mehrfachen Auftreten eines Attributwertes korrespondieren alle Tupel mit diesem Attributwert mit dem zugehörigen gleichen Tupel der anderen Relation. Die

Treffertupel können mit einem Hash-Tabellenzugriff ermittelt werden, wenn der Bucket-Eintrag eine Liste aller Tupel mit gleichem Schlüsselwert enthält. Die Tupel können durch einen Iterator sequentiell abgearbeitet werden. Durch diese Liste wird weiterhin ein Bucket-Überlauf bei einem extrem häufigen Auftreten einzelner Attributwerte vermieden. Die Klasse *List* realisiert diese Liste. Aus Effektivitätsgründen besteht ein Listeneintrag aus der Adresse einer Instanz der Klasse *Tuple*. Ein Tupel wird durch *List::Insert(Tup)* in eine Liste eingetragen. Ein Listeneintrag kann durch *List::Kill(Tup)* gelöscht werden. Die Operationen *List::First()* und *List::Next()* realisieren den Iterator.

Ein Bucket enthält neben der Information über seine lokale Tiefe (*LocalDeep*) die aktuelle Listenanzahl (*EntryCount*), die Listen mit Tupeln (*Entry[]*) und für jede Liste von Tupeln den Schlüsselwert der Tupel (*Key[]*). Die Funktion *Bucket::IsFull()* liefert *TRUE* als Ergebnis, wenn keine Liste mehr in das Bucket aufgenommen werden kann. Ein Tupel oder eine Liste von Tupeln kann durch die Funktion *Bucket::InsertEntry(Key,entry)* in das Bucket eingetragen werden.

Die Klasse *HashTable* realisiert die Hash-Tabelle. Sie stellt folgende Funktionen bereit:

- Eintragen eines Tupels in die Hash-Tabelle *HashTable::Insert(key,entry)*,
- Ermitteln einer Liste, die alle Tupel mit einem bestimmten Schlüssel enthält *HashTable::Get(key)* und
- Löschen eines Eintrages *HashTable::Kill(key,Entry)*.

Der Hash-Wert kann nur aus einem kardinalen Schlüsselwert ermittelt werden. Um nicht-kardinale Schlüssel in einen kardinalen Wert umzuwandeln, wird *HashTable::StrToKey(addr,size)* verwendet.

Sollen Tupel mit kardinalen Schlüsselwerten ohne mehrfach auftretende Schlüsselwerte in einer Hash-Tabelle verwaltet werden, bestehen die Bucket-Einträge aus je einem Tupel; die Verwaltung einer Liste ist nicht erforderlich. Dafür werden zusätzlich die Klassen *PrimaryKeyBucket* und *PrimaryKeyHashTable* zur Verfügung gestellt, die den Klassen *Bucket* und *HashTable* sehr ähneln, auf die an dieser Stelle jedoch nicht weiter eingegangen wird.

6.2.3.2. Die Implementierung des lokalen Joins

Der lokale Join, die Join-Phase des parallelen Joins, wird durch die Klasse *join_op* realisiert:

```
class join_op:public operation {
      public:
      // ................... HJoinV1 ..............................

            join_op(  Hash_T HashType_R,
                      mmem *m, qLATTR_ptr *qla_in, qLATTR *qla_out,
                      TB_stream_ptr *TBs_in, TB_stream_ptr *TBs_out,
                      where_ptr s_out, int streams_in = 2,
                      int streams_out = 1);
      void  join1( qnode *qual, qATTR *HashAttr_R, qATTR *HashAttr_S);

      // ................... HJoinV2 ..............................

            join_op(  Hash_T HashType_R, Hash_T HashType_S,
                      mmem *m, qLATTR_ptr *qla_in, qLATTR *qla_out,
                      TB_stream_ptr *TBs_in, TB_stream_ptr *TBs_out,
                      where_ptr s_out, int streams_in = 2,
                      int streams_out = 1);
      void  join2( qnode *qual, qATTR *HashAttr_R, qATTR *HashAttr_S);
};
```

Die Join-Konstruktoren haben neben den Argumenten für ihre Basisklasse zusätzlich ein Argument, das für jede Hash-Tabelle angibt, ob es sich um kardinale Schlüsselwerte mit einfachem Auftreten (*PrimaryKey*) handelt oder nicht (*MultipleKey*). Bisher wurde der lokale Join als Hemmnis für den Datenfluß betrachtet, weil die Tupel der gesamten inneren Relation *R* in eine Hash-Tabelle eingetragen werden müssen, bevor die Tupel der äußeren Relation *S* den Test nach Treffertupeln durchführen können (siehe Abb.). Dieser Join wird durch *join_op::join1(qual,JoinAttr_R,JoinAttr_S)* mit der Qualifikation *qual* und den Join-Attributen der inneren Relation *JoinAttr_R* und der äußeren Relation *JoinAttr_S* realisiert. Er wird im folgenden mit *HJoinV*1 bezeichnet. Die empfangenen Tupelblöcke der inneren Relation werden in eine Liste eingetragen. Für jedes Tupel aus den Tupelblöcken erfolgt ein Eintrag in die Hash-Tabelle entsprechend des

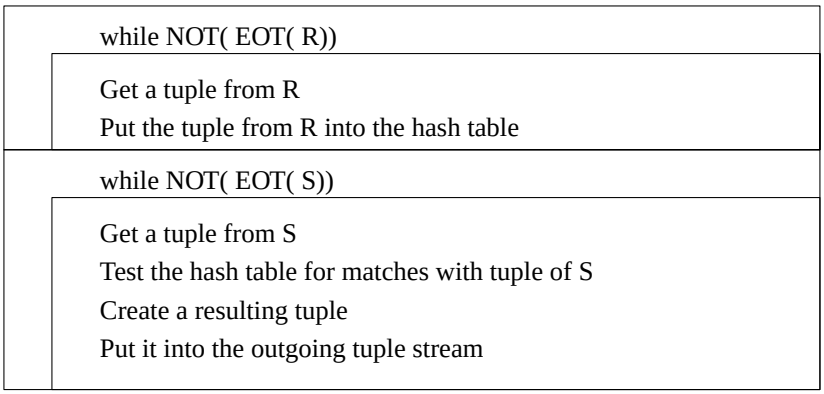


Abb. 28: Der lokale Join *HJoinV*1

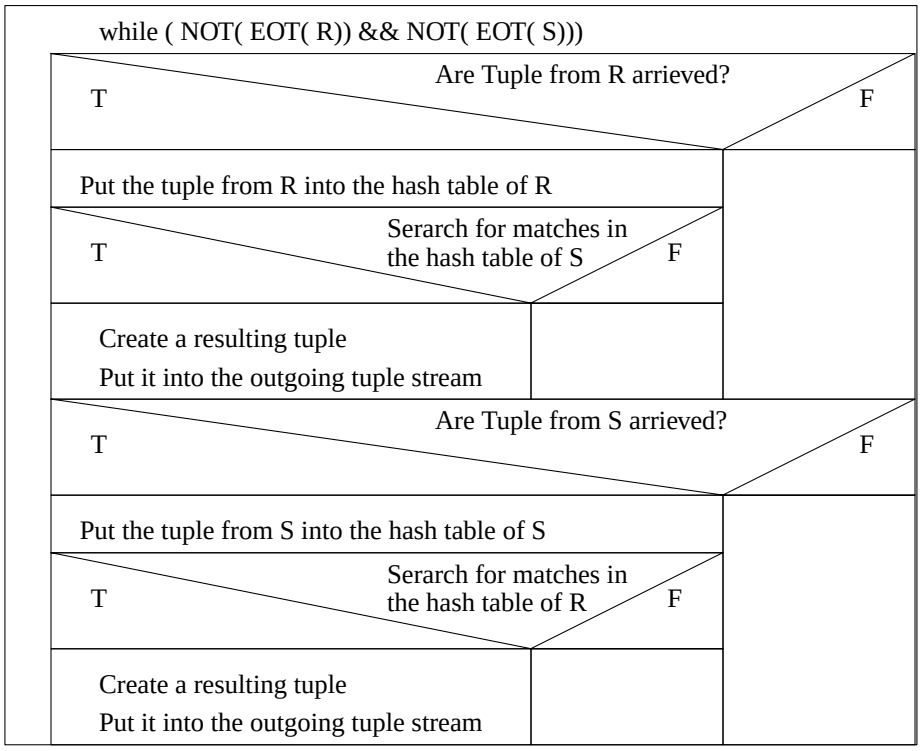


Abb. 29: Der lokale *HJoinV*2

Attributwertes. Der herausgehende Tupelstrom versendet Tupelblöcke einer festen Tupelanzahl. Enthält ein Tupelblock diese Anzahl von Tupeln, wird er an den nächsten Operator weitergegeben.

Bauen sowohl die innere als auch die äußere Relation gleichzeitig eine Hash-Tabelle auf, können für jedes ankommende Tupel nach dem Eintragen in ihre entsprechende Hash-Tabelle Treffertupel in der Hash-Tabelle für die Tupel der korrespondierenden Relation ermittelt werden (siehe Abb. 29). Durch die Überlappung von Building- und Probing-Phase des lokalen Joins muß nicht erst die Hash-Tabelle für alle Tupel der inneren Relation aufgebaut werden, ehe Ergebnistupel an die nächste Operation geschickt werden können. Dadurch wird der kontinuierliche Datenfluß besser unterstützt als beim *HJoinV*1, weil das erste Ergebnistupel früher zur Verfügung steht. Für die Organisation der zweiten Hash-Tabelle entstehen zusätzliche Kosten. Die Verwendung von Hash-Tabellen für beide eingehende Relationen erzeugt einen größeren Bedarf an Hauptspeicher. Dieser Join wird durch *join_op::join2(qual,JoinAttr_R,JoinAttr_S)* mit der Qualifikation *qual* und den Join-Attributen der inneren Relation *JoinAttr_R* und der äußeren Relation *JoinAttr_S* realisiert. Er wird im folgenden mit *HJoinV*2 bezeichnet. Für beide eingehende Tupelströme werden die Tupelblöcke in einer Liste aufbewahrt, auf deren Tupel die Einträge der jeweiligen Hash-Tabelle zeigen.

7. Leistungsbetrachtungen

Zur Leistungsbetrachtung der DRAM-Operationen wurden sechs Relationen unterschiedlicher Größe verwendet (siehe Tab. 7). Die Tupel der Relationen bestehen aus sechs Attributen mit unterschiedlicher Attributwert-Verteilung (siehe Tab. 8). Die Messungen der CPU-Kosten wurden ausschließlich auf einer SPARCstation 10/40 vorgenommen, um vergleichbare Ergebnisse zu erhalten.

Die UNIX-Werkzeuge profile [Bent88] und time [Hero92] wurden verwendet, um die Meßergebnisse zu erhalten.

Relation	**Tupelanzahl**
R1	17
R2	57
R3	507
R4	1007
R5	5007
R6	10007

Tab. 7: Tupelanzahl der Relationen† und ihre Größe in Bytes

Als Übertragungskosten werden alle anfallenden CPU-Kosten betrachtet, die auf der Realisierung der Tupelübertragung beruhen. Sie ergeben sich aus den CPU-Kosten für den Empfang und für das Senden.

Jeweils für die gleiche Anzahl von ankommenden Tupeln sind die Empfangskosten gleich. Die Kosten bei der gleichen Anzahl von zu verschickenden Tupeln sind für das Senden ebenfalls gleich. Bei einer geringen Anzahl von Ergebnistupeln ist ihr Einfluß auf die Gesamtkosten unbedeutend. Dagegen beeinflußt eine große Anzahl von Ergebnistupeln die Kosten einer Operation deutlich. Die Realisierung des Schiebefensters erzeugt den wesentlichen Teil (über 80%) dieser Kosten. Die Verwendung von RDP als Transportprotokoll anstelle von UDP verringert den Einfluß des Schiebefenster-Protokolls auf die Ausführungskosten, weil es im Betriebssystemkern implementiert wird.

In der Tabelle 9 wird der prozentuale Anteil der benötigten Zeit bei der Ausführung des Join-Prozesses für das Senden, das Empfangen und den eigentlichen Join aufgeführt.

† Notation bei einem Self-Join: Join über R_i und R_i'

Attribut	**Verteilung**	**Bereich**
PK	diskret gleichverteilt	0 - $card(R) - 1$
A1	negativ exponential	0 - ∞ Mittelwert: 100
A2	Poisson	0 - ∞ Mittelwert: 100
A3	diskret gleichverteilt	10 - 1000
A4	normal	4090-5010
A5	normal	3000-7000

Tab. 8: Die Attribute und ihre Verteilung

Innere Relation	**Äußere Relation**	**Join-Bedingung**	**Ergebnis-tupelanz.**	**Senden**	**Emp-fang**	**Join**
R3	*R3'*	*R3.PK = R3'.PK*	507	1%	15%	84%
R4	*R4'*	*R4.PK = R4'.PK*	1007	1%	16%	83%
R1	*R6*	*R1.PK = R6.PK*	17	1%	22%	77%
R6	*R6'*	*R6.PK = R6'.PK*	10007	46%	8%	54%
R4	*R4'*	*R4.A5 = R4'.PK*	0	1%	10%	89%
R4	*R4'*	*R4.A4 = R4'.A4*	89171	59%	4%	37%

Tab. 9: Kostenanteile von Senden, Empfangen und Join

Eigene Untersuchungen über den Einfluß der Schiebefenstergröße beim Sender ergaben, daß sowohl ein zu kleines als auch ein zu großes Schiebefenster Nachteile mit sich bringt. Bei einem zu kleinen Schiebefenster ist der Datendurchsatz gering, weil der Puffer des Schiebefensters schnell voll ist und dann auf die Empfangsbestätigungen gewartet werden muß, bis der nächste Rahmen abgeschickt werden kann. Das Stop and wait-Protokoll [Tane81] ist ein Extremfall, bei dem genau eine PDU im Schiebefenster aufgenommen werden kann. Der Sender kann erst dann wieder eine PDU abschicken, wenn die Empfangsbestätigung für die zuvor abgeschickte PDU eingetroffen ist. Der

Vorteil eines großen Schiebefensters ist sein größerer Datendurchsatz. Kommt es jedoch zu Kollisionen, trifft für die betroffenen PDUs keine Empfangsbestätigung ein. Dadurch halten sich diese PDUs vergleichsweise lange im Schiebefenster auf. Das wiederholte Abschicken einer unbestätigten PDU erfolgt, wenn das Schiebefenster keine PDUs mehr aufnehmen kann oder wenn die Kommunikation beendet werden soll und das Schiebefenster noch unbestätigte Rahmen enthält. Im Vergleich zu kleinen Schiebefenstern dauert es bei großen Schiebefenstern lange, bis keine PDUs mehr aufgenommen werden können. Untersuchungen ergaben eine angemessene Übertragungszeit sowohl bei geringen, als auch bei großen Datenmengen, wenn das Schiebefenster zwischen 16 und 64 Einträge aufnehmen kann.

Durch das Tupelprotokoll werden neben den Daten zusätzliche Steuerinformationen übertragen, die den Transport-Overhead bilden. Dieser besteht für das Senden eines Tupels aus einem Steuerfeld (Typ *DATA*) und dem Nachrichtenkopf jedes Datenfeldes. In der DRAM ist ein Steuerfeld genauso groß wie der Nachrichtenkopf, und zwar 18 Bytes.

$$Overhead_{sendTB} = 18Byte * (1 + ceil(\frac{sizeof(TupleBlock)}{MTU})) \quad (1)$$

Die Beendigung der Tupelübertragung erfordert das Abschicken des Steuerfeldes (Typ *EOT*):

$$Overhead_{sendEOT} = 18Byte \quad (2)$$

Zusätzlich wird der Empfang jedes Nachrichtenpaketes bestätigt:

$$Overhead_{Ackn} = Overhead_{sendTB} + Overhead_{sendEOT} \quad (3)$$

Werden die Nachrichtenpakete in einem Rechnernetz ohne Verlust übertragen, entsteht für eine Relation *R* folgender Overhead:

$$Overhead = ceil(\frac{card(R)}{n\dagger_{sizeTB}}) * (Overhead_{sendTB} + Overhead_{Ackn}) \quad (4)$$

$$+Overhead_{sendEOT}$$

† Anzahl der Tupel in einem Tupelblock (siehe Tab. 12).

(5)

Für die Tupelübertragung ergeben sich für die Beispielrelationen für Tupelblöcke mit 25 Tupeln (kleiner als die MTU‡) und für Tupelblöcke mit 50 Tupeln (größer als MTU‡) die in der Tabelle 10 dargestellten Overheads.

Relation	**Daten**	**Overhead bei 25 Tupeln in TB**	**Overhead bei 50 Tupeln in TB**
R1	408 Byte	54 Byte (13,2%)	72 Byte (17,6%)
R2	1368 Byte	126 Byte (9,2%)	72 Byte (5,3%)
R3	12168 Byte	774 Byte (6,4%)	612 Byte (5,0%)
R4	24168 Byte	1494 Byte (6,2%)	1152 Byte (4,8%)
R5	240168 Byte	7254 Byte (6,0%)	5472 Byte (4,5%)
R6	240168 Byte	14454 Byte (6,0%)	10872 Byte (4,5%)

Tab. 10: Daten-Overhead beim Tupelprotokoll für die Übertragung von Relationen

Verglichen mit der Nutzung großer Tupelblöcke bewirkt die Verwendung von kleinen Tupelblöcken einen großeren Overhead bei der Übertragung einer Relation. Eine Ausnahme bildet eine kleinere Tupelanzahl der zu versendenden Relation gegenüber der Tupelblockgröße. Ein Tupelblock der größer als die Anzahl der zu versendenden Tupel ist, führt immer zu einem größeren Overhead gegenüber Tupelblöcken, die kleiner als die Tupelanzahl sind.

Eine Empfangsbestätigung wird für jedes empfangene Nachrichtenpaket gesendet. Das Sammeln mehrerer Empfangsbestätigungen und ihr gemeinsames Versenden verringern den Aufwand für die Kommunikation.

Die Kommunikation über Shared memory bei Prozessen, die auf einem Rechner ausgeführt werden, bewirkt die Übertragung eines Steuerfeldes je Tupelblock und die Übertragung eines Steuerfeldes für die Kommunikationsbeendigung:

$$Overhead_{shm} = 18Byte * (ceil(\frac{card(R)}{n_{sizeTB}}) + 1) \quad (6)$$

Bei der Kommunikation über Shared memory wird nur der $Overhead_{shm}$ übertragen.

‡ MTU-Größe: 1024 Byte

Damit werden die Kommunikationskosten durch die Verwendung der lokalen Kommunikation für die Tupelübertragung bei der Ausführung von Operationen geringer Komplexität auf einem Rechner gesenkt.

Ein weiterer Punkt für die Bewertung des Tupelprotokolls ist der Einfluß der Netzlast und damit die Anzahl der Kollisionen. Die Kollisionsrate sollte auf keinen Fall vernachlässigt werden.

7.1. Die Bewertung der Join-Operation

Zur Bewertung der Join-Phase, dem lokalen Join, wird eine Kostenfunktion für die CPU-Kosten hergeleitet. Ihre Parameter sind in den Tabellen 11 und 12 aufgeführt.

Wert	**Bedeutung**
$card(R)$†	Tupelanzahl einer Relation (Kardinalität)
$deg(R)$‡	Attributanzahl der Tupel der Relation R (Rang)
$n_{DoubleHT}(R)$	Anzahl der notwendigen Hash-Tabellenvergrößerungen
$n_{MatchTup}(R)$	Anzahl der gefundenen Treffertupel
$n_{SizeBucket}$	Bucketgröße (Konstante)
$n_{SizeHT}(R)$	Anzahl der Hash-Tabelleneinträge
n_{SizeTB}	Anzahl der Tupel im Tupelblock (Konstante)
$n_{Split}(R)$	Anzahl der notwendigen Bucket-Aufspaltungen
R	innere Relation
S	äußere Relation
T	Ergebnisrelation

Tab. 12: Konstanten und Werte für die Kostenfunktionen

Die Join-Phase ist die kostenintensivste Komponente eines verteilten Hash join-Algorithmus' wenn man von den lastbalancierten Hash join-Verfahren [DeWi92b, Hua91, Wolf93, Zell90] absieht. Mit Hilfe der Bewertung der lokalen Joins wird durch die Anfrageoptimierung eine günstige Aufteilung der Join-Phasen auf die verschiedenen

† Vereinfachend für die Kardinalität der Ergebnisrelation: $card(T) = p * card(S) * card(R)$

‡ Vereinfachend für den Rang der Ergebnisrelation: $deg(T) = deg(R) + deg(S)$

Parameter	Kosten für
c_{alloc}	das Anfordern von Hauptspeicher
c_{cmp}	eine Vergleichs-Operation
c_{cpy}	eine Kopieroperation
c_{match}	die Überprüfung der Erfüllung der Join-Bedingung
$c_{BuildNewTup}(R)$	das Erzeugen eines Ergebnistupels aus zwei Tupeln
$c_{DoubleHT}(R)$	die Vergrößerung einer Hash-Tabelle
c_{Entry}	das Eintragen eines Tupels in ein Bucket
$c_{HashRel}(R)$	das Hashing der gesamten Relation R
$c_{HashTup}(R)$	das Eintragen eines Tupels in eine Hash-Tabelle
c_{Join}	die Ausführung des Joins über die Tupel
c_{JoinOp}	die Join-Phase eines verteilten Joins
c_{JoinV1}	$HJoinV1$
c_{JoinV2}	$HJoinV2$
$c_{PosAttr}(R)$	die Positionierung auf ein Attribut
$c_{ProbeRel}(R)$	das Probing der gesamten Relation R
$c_{ProbeTup}(R)$	das Ermitteln der Treffertupel für ein Tupel
c_{RecvI}	den Empfang von Tupeln der inneren Relation
c_{RecvO}	den Empfang von Tupeln der äußeren Relation
$c_{Recv}(R)$	den Empfang einer Relation
$c_{Recv\_TB}(R)$	den Empfang eines Tupelblocks
$c_{ResRel}(R)$	die Ergebnistupelerzeugung einer Relation
$c_{ResTup}(R)$	die Ergebnistupelerzeugung aus den "Treffern" eines Tupels
c_{Send}	das Senden von Tupeln der Ergebnisrelation
$c_{Send}(R)$	das Senden einer Relation
$c_{Send\_TB}(R)$	das Senden eines Tupelblocks
$c_{SplitBucket}(R)$	das Aufspalten eines Buckets
$c_{Tup}(R)$	die Verarbeitung eines Tupels einer Eingangsrelation
$c_{TupInner}(R)$	die Verarbeitung eines Tupels der inneren Relation
$c_{TupOuter}(R)$	die Verarbeitung eines Tupels der äußeren Relation
$k_{EntryPtr}$	die Zuweisung eines Zeigers (Konstante)

Tab. 11: Parameter für die Kostenfunktionen

Ausführungsknoten erreicht. Die Kosten für einen lokalen Join bestehen aus den

Übertragungskosten und den Kosten für die Ausführung des Joins über die Eingangsrelationen:

$$c_{JoinOp} = c_{RecvI} + c_{RecvO} + c_{Join} + c_{Send} \quad (7)$$

Die Kosten für den Empfang von Tupeln sind von der Tupelanzahl der Eingangsrelationen abhängig. Die Selektivität des Joins hat entscheidenden Einfluß auf den Anteil der Versandkosten c_{Send} an den Gesamtkosten c_{JoinOp} (siehe Tab. 9).

Nachfolgend wird für die beiden Join Algorithmen *HJoinV*1 und *HJoinV*2 eine Kostenfunktion vorgestellt. Über die Relevanz der einzelnen Kostenkomponenten für die Gesamtkosten der Join-Operation wird auf Grundlage der umfangreichen Meßergebnisse, die mit dem UNIX-Profiler *prof* ermittelt wurden, entschieden.

7.1.1. Die Kosten je Tupel für die einzelnen Phasen des Joins

Die Tupel einer Relation werden aufgrund eines Attributwertes verbunden. Das erfordert eine Positionierung auf das Join-Attribut. Sie erfordert zwei Vergleichsoperationen je Attribut, das in der Attributliste vor dem gesuchten Attribut steht. Jeweils zwei Vergleiche sind notwendig, weil die Attributidentifizierung über den Relationsnamen und über den Attributnamen erfolgt. Die Vergleiche werden maximal für alle Attribute in der Attributliste ausgeführt.

$$c_{PosAttr}(R) = deg(R) * 2 * c_{cmp} \quad (8)$$

Die Tupel werden in eine Hash-Tabelle eingetragen, um einen schnellen Zugriff auf ein gesuchtes Tupel zu ermöglichen. Für den Eintrag eines Tupels in ein Bucket der Hash-Tabelle muß der Hash-Wert bestimmt werden. Die daraus resultierenden Kosten sind vernachlässigbar. Für den ermittelten Hash-Wert kann über die Hash-Tabelle das betreffende Bucket bestimmt werden. Ist das Bucket voll, muß es aufgespalten werden, bevor das Tupel eingetragen werden kann. Dieser Vorgang wird solange wiederholt, bis das Tupel in das ermittelte Bucket eingetragen werden kann. Wird die lokale Tiefe des Buckets aufgrund der Spaltungen größer als die globale Tiefe, wird die Hash-Tabelle vergrößert. Das Vergrößern der Hash-Tabelle besteht im Anfordern von Speicher und der Umorganisation der Tabelle.

$$c_{DoubleHT}(R) = c_{alloc} * 2 * n_{SizeHT}(R) * k_{EntryPtr} \quad (9)$$

Durch das Aufspalten eines Buckets werden zwei Buckets angelegt und die Tupel auf die beiden neuen Buckets aufgeteilt.

$$c_{SplitBucket}(R) = 2 * c_{alloc} + n_{SizeBucket} * c_{Entry} \quad (10)$$

Die Kosten für das Eintragen eines Tupels hängen vom Hash-Attributtyp ab. Handelt es sich beim Hash-Attribut um einen kardinalen, einfach auftretenden Attributwert, wird nur der Zeiger auf das Tupel in das Bucket eingetragen. Ein nicht-kardinales Hash-Attribut muß in einen kardinalen Hash-Schlüssel transformiert werden. Die Transformation in einen Hash-Schlüssel verursacht vernachlässigbare Kosten. Nach der Transformation eines nicht-kardinalen, atomaren Attributwertes kann nicht gewährleistet werden, daß der erzeugte Hash-Schlüssel ebenfalls atomar ist. Falls Attributwerte mehrfach auftreten, sind die Bucketeinträge Listen. wenn Attributwerte mehrfach auftreten. Das Eintragen eines Tupels verlangt eine Überprüfung, ob bereits Tupel mit demselben Attributwert in dem Bucket eingetragen wurden. Diese Überprüfung erfordert eine Vergleichsoperation je Liste (Bucketeintrag). Es erfolgen maximal $n_{SizeBucket}$ Vergleichsoperationen.

$$c_{Entry} = \begin{cases} k_{EntryPtr} \\ n_{SizeBucket} * c_{cmp} + k_{EntryPtr} \end{cases} \quad (11)$$

Die Kosten für das Eintragen eines Tupels in eine Hash-Tabelle sind:

$$c_{HashTup}(R) = n_{DoubleHT}(R) * c_{DoubleHT} \quad (12)$$

$$+ n_{Split}(R) * c_{SplitBucket} + c_{Entry}$$

Das Ermitteln von Treffertupeln erfolgt durch den Zugriff auf die Tupel der korrespondierenden Relation über die Hash-Tabelle. Dieser Zugriff erfordert gegebenenfalls die Transformation eines nicht-kardinalen Attributwertes in einen kardinalen Hash-Schlüssel. Die Hash-Schlüssel werden in den Hash-Wert umgewandelt. Anschließend wird das dem Hash-Wert zugeordnete Bucket auf Treffer-Tupel untersucht, indem die Bucketeinträge mit dem gesuchten Hash-Schlüssel verglichen werden. Für die gefundenen Tupel erfolgt eine Überprüfung auf Erfüllung der

Join-Bedingung (match). Die Attributwert-Transformation und die Hash-Wertermittlung können wegen ihrer Geringfügigkeit ebenfalls vernachlässigt werden.

$$c_{ProbeTup}(R) = n_{MatchTup}(R) * c_{match} + n_{SizeBucket} * c_{cmp} \quad (13)$$

Die Kostenfunktionen für Verarbeitung jeweils eines Tupels durch $HJoinV1$ werden für die innere Relation durch $c_{TupelInner}$ und für die äußere Relation durch $c_{TupelOuter}$ dargestellt:

$$c_{TupInner}(R) = c_{PosAttr}(R) + c_{HashTup}(R) \quad (14.a)$$

$$c_{TupOuter}(S) = c_{PosAttr}(S) + c_{ProbeTup}(S) \quad (14.b)$$

Die Kostenfunktionen für Verarbeitung jeweils eines Tupels durch $HJoinV2$ sind für die innere und äußere Relation gleich (c_{Tup}).

$$c_{Tup}(R) = c_{PosAttr}(R) + c_{HashTup}(R) + c_{ProbeTup}(R) \quad (15)$$

$HJoinV2$ führt je Tupel der Eingangsrelationen sowohl Hashing als auch Probing aus, dagegen führt $HJoinV1$ je Tupel der Eingangsrelationen entweder Hashing oder Probing aus. Durch den höheren Aufwand je Tupel beim $HJoinV2$ sind die verursachten Kosten für die Verarbeitung eines Tupels einer Eingangsrelation höher als für den $HJoinV1$.

Die Kosten für das Erstellen eines Ergebnistupels und dessen Eintrag in einen Ergebnis-Tupelblock aus den Treffertupeln sind für beide Join-Algorithmen gleich.

Um ein Ergebnis-Tupel aus zwei Tupeln zu erzeugen, wird Speicher angefordert. Für jedes Attribut des Tupels der Ergebnisrelation wird der Wert aus einem Tupel einer der Eingangsrelationen kopiert. Dafür muß die Positionierung auf das Attribut im Ergebnistupel und auf das Attribut in einem der beiden Treffertupel erfolgen. Die Kosten für die Positionierung auf das Attribut in einem der Treffertupel werden den Kosten für die Positionierung auf das Attribut im Ergebnistupel gleichgesetzt, weil ihre Attributanzahl zusammengenommen gleich der Attributanzahl des Ergebnistupels ist ($deg(T) = deg(R) + deg(S)$).

$$c_{BuilNewTup}(T) = deg(T) * (c_{PosAttr}(R) * 2 + c_{cpy}) + c_{alloc} \quad (16)$$

Das Eintragen eines Tupels in einen Ergebnis-Tupelblock wird durch eine Kopieroperation ausgeführt. Die Kosten für ein Ergebnistupel sind:

$$c_{ResTup}(T) = n_{MatchTup}(T) * (c_{BuildNewTup}(T) + c_{cpy}) \quad (17)$$

7.1.2. Die Join-Ausführungskosten, Antwort- und Reaktionszeit

Auf die Kosten für den Join einer Relation (bzw. Teilrelation) wird im folgenden eingegangen.

Die Kosten für das Hashing der Tupel einer Relation bestehen aus den Kosten für das Vergrößern der Hash-Tabelle, das Aufspalten der Buckets und für das Eintragen der Tupel in das Bucket.

$$c_{HashRel}(R) = n_{DoubleHT}(R) * c_{DoubleHT}(R) + n_{Split}(R) * c_{SplitBucket} \quad (18)$$

$$+card(R) * (c_{Entry} + c_{PosAttr}(R))$$

Die Kosten für das Probing einer Relation ergeben sich aus dem Probing der einzelnen Tupel.

$$c_{ProbeRel}(R) = card(R) * c_{ProbeTup}(R) \quad (19)$$

Die Kosten für das Erstellen aller Ergebnistupel eines Joins ergeben sich aus den Kosten für das Erzeugen der Ergebnistupel und das Eintragen in den Ergebnistupelblock.

$$c_{ResRel}(R) = card(R) * (c_{BuildNewTup}(R) + c_{cpy}) \quad (20)$$

Die Tupel einer Relation werden in Tupelblöcken verschickt. Die Empfangskosten einer Relation setzen sich aus den Empfangskosten je Tupelblock zusammen.

$$c_{Recv}(R) = ceil(\frac{card(R)}{n_{SizeTB}}) * c_{Recv\_TB}(R) \quad (21)$$

Die Sendekosten einer Relation setzen sich aus den Sendekosten je Tupelblock zusammen.

$$c_{Send}(R) = ceil(\frac{card(R)}{n_{SizeTB}}) * c_{Send\_TB}(R) \quad (22)$$

Die Gesamtkosten für einen $HJoinV1$ betragen:

$$c_{JoinV1}(R,S,T) = c_{HashRel}(R) + c_{ProbeRel}(S) + c_{ResRel}(T) \tag{23}$$

$$+c_{Recv}(R) + c_{Recv}(S) + c_{Send}(T)$$

Die Gesamtkosten für einen $HJoinV2$ betragen:

$$c_{JoinV2}(R,S,T) = c_{HashRel}(R) + c_{ProbeRel}(R) \tag{24}$$

$$+c_{HashRel}(S) + c_{ProbeRel}(S) + c_{ResRel}(T)$$

$$+c_{Recv}(R) + c_{Recv}(S) + c_{Send}(T)$$

Die Kosten für die Berechnung des lokalen Joins bei einer Berechnung durch den $HJoinV1$ sind niedriger als beim $HJoinV2$:

$$c_{JoinV2}(R,S,T) = c_{JoinV1}(R,S,T) + c_{ProbeRel}(R) \tag{25}$$

$$+c_{HashRel}(S)$$

Die Anzahl der Hash-Tabellenvergrößerungen ist gering. Die Kosten für die Vergrößerung sind relativ gering. Sie können vernachlässigt werden.

Die detaillierte Kostenfunktion für $HJoinV1$ lautet:

$$c_{JoinV1}(R,S,T) = c_{alloc} * (2 * n_{Split}(R) + card(T)) \tag{26}$$

$$+c_{Entry} * (n_{Split}(R) * n_{SizeBucket} + card(R))$$

$$+c_{cmp} * (card(R) * deg(R) * 2$$

$$+ card(S) * n_{SizeBucket}$$

$$+ card(T) * (deg(T))^2 * 4)$$

$$+c_{cpy} * card(T) * deg(T)$$

$$+c_{match}(S) * card(S)$$

$$+ceil(\frac{card(R) + card(S)}{n_{SizeTB}}) * c_{RecvTB}$$

$$+ceil(\frac{card(T)}{n_{SizeTB}}) * c_{SendTB}$$

Die Kostenfunktion für *HJoinV2* lautet:

$$c_{JoinV2}(R,S,T) = c_{alloc} * (2 * n_{Split}(R) + 2 * n_{Split}(S) \tag{27}$$

$$+ card(T))$$

$$+c_{Entry} * ((n_{Split}(R) + n_{Split}(S)) * n_{SizeBucket}$$

$$+ card(R))$$

$$+c_{cmp} * (card(R) * deg(R) * 2$$

$$+ card(S) * deg(S) * 2$$

$$+ (card(R) + card(S)) * n_{SizeBucket}$$

$$+ card(T) * (deg(T))^2 * 4)$$

$$+c_{cpy} * card(T) * deg(T)$$

$$+c_{match}(R) * card(R)$$

$$+c_{match}(S) * card(S)$$

$$+ceil(\frac{card(R) + card(S)}{n_{SizeTB}}) * c_{RecvTB}$$

$$+ceil(\frac{card(T)}{n_{SizeTB}}) * c_{SendTB}$$

Die Kostenfunktionen bestätigen die weiter oben getroffene Feststellung, daß die Anzahl der Ergebnistupel einen wesentlichen Einfluß auf die Kosten des Joins haben. Die Abbildung 30 stellt die CPU-Kosten von Join-Operationen über die selben Eingangsrelationen (Join über R4 und R4') bei unterschiedlicher Selektivität der Joins dar.

Der Vergleich der Kostenfunktionen beider Join-Algorithmen je Tupel ($c_{TupInner}$ und $c_{TupOuter}$ sowie c_{Tup}) und für eine Relation (c_{JoinV1} und c_{JoinV2}) macht deutlich, daß die

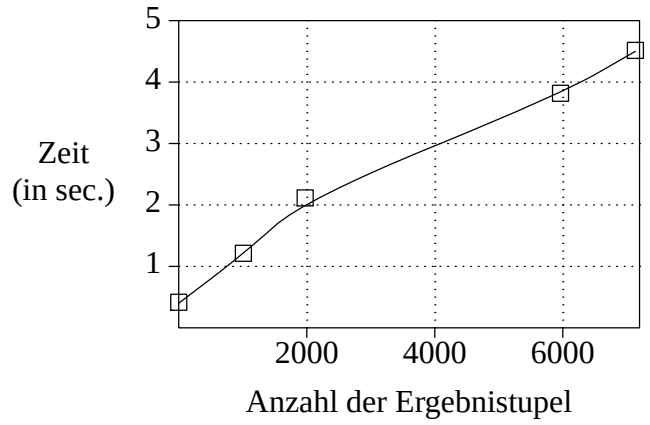


Abb. 30: Der Einfluß der Ergebnistupelanzahl auf die Join-Kosten

Kosten für *HJoinV*2 höher sind als für *HJoinV*1. Das wird durch die in Abbildung 31 dargestellten Meßergebnisse bestätigt (Join-Bedingung: $R_i.PK = R_i'.PK$).

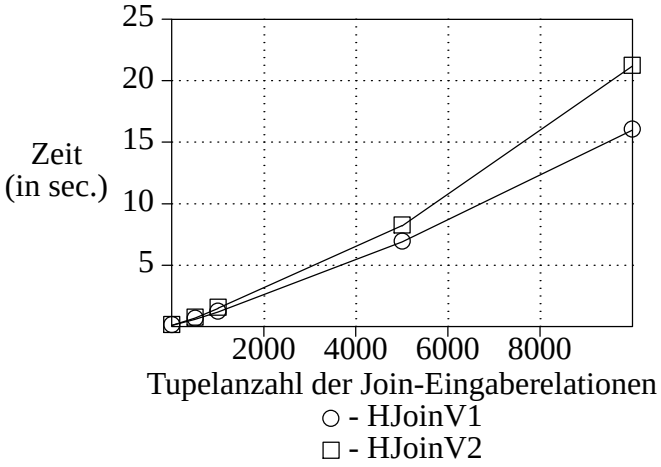


Abb. 31: Die unterschiedlichen Kosten der beiden Join-Algorithmen

Der Vorteil des *HJoinV*2 liegt in seiner Unterstützung des Datenflusses. Ergebnistupel werden früher als bei *HJoinV*1 erzeugt. Die nachfolgenden Operationen können ihre Eingabetupel eher entgegennehmen und verarbeiten, als wenn ihre vorhergehende Operation *HJoinV*1 ist.

Die Zeit, die bis zur Erzeugung des letzten Ergebnistupels benötigt wird, ist die Antwortzeit. Die Reaktionszeit gibt an, wann das erste Ergebnistupel einer Operation erzeugt wird. Eine Operation mit einer hohen Reaktionszeit unterbricht den kontinuierlichen Datenfluß.

O_i sei die Operation, deren Ergebnistupel die innere Relation der Join-Operation O_{join} bilden. O_o sei die Operation, deren Ergebnistupel die äußere Relation der Join-Operation bilden. Die Antwortzeit ist A_k und die Reaktionszeit ist R_k mit $k \in (i, o, join)$.

Die Antwortzeit hängt von der Antwortzeit der vorhergehenden Operationen im Anfragebearbeitungsplan und von den CPU-Kosten der Operation ab. Sie beträgt für $HJoinV1$:

$$A_{joinV1} = max(A_{i,} A_o) + c_{JoinV1} \tag{28}$$

Die Antwortzeit des $HJoinV2$ ist:

$$A_{joinV2} = max(A_{i,} A_o) + c_{JoinV2} \tag{29}$$

Die Antwortzeit des $HJoinV2$ ist höher als die Antwortzeit des $HJoinV1$, weil $HJoinV2$ kostenintensiver als $HJoinV1$ ist.

Das erste Ergebnistupel kann beim $HJoinV1$ erzeugt werden, nachdem alle Tupel der inneren Relation empfangen wurden und das erste Tupel der äußeren Relation eingetroffen ist.

$$R_{joinV1} = max(A_{i,} R_o) + c_{TupOuter}(S) + c_{ResTup}(T) \tag{30}$$

Das erste Ergebnistupel beim $HJoinV2$ kann bereits nach dem Eintreffen des jeweils ersten Tupels beider Ergebnisrelationen erzeugt werden, spätestens jedoch, nachdem alle Tupel einer Eingangsrelation eingetroffen sind.

$$max(R_{i,} R_o) + c_{Tup}(R) + c_{ResTup}(T) \leq R_{joinV2} \tag{31}$$

$$\leq max(A_{i,} R_o) + c_{Tup}(R) + c_{ResTup}(T)$$

Die Reaktionszeit ist beim $HJoinV2$ günstiger.

Es wurden vier Join-Operationen in einer Kaskade hintereinander ausgeführt, um den Einfluß der Antwortzeit der Vorgänger-Operationen auf eine Operation zu ermitteln. Die Eingaberelation ist jeweils die Relation $R4$, die Join-Bedingung ist beim Join auf dem ersten Rechner $R_4'.PK = R_4.PK$, bei allen weiteren Joins $R_{temp}.PK = R_4.PK$. Die Anzahl der Ergebnistupel stimmt mit der Anzahl der Tupel einer Eingaberelation überein. Es wird auf jedem Rechnerknoten ein Join über Relationen gleicher Tupelanzahl ausgeführt, der gleichgroße Ergebnisrelationen zum Ergebnis hat ($p = \frac{1}{card(R_4)}$). Somit sind die Join-Operationen auf den einzelnen Knoten vergleichbar.

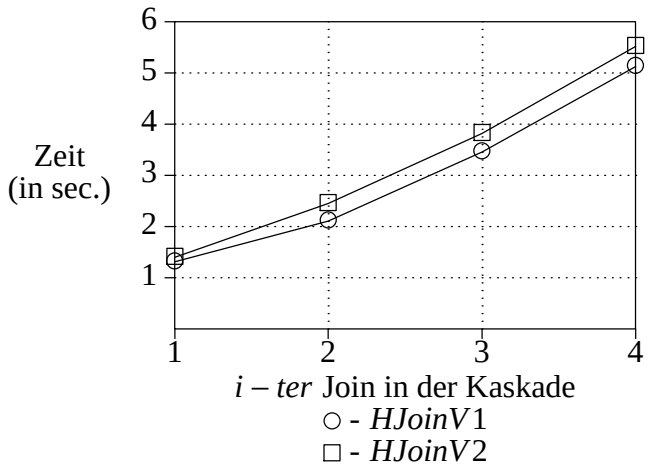


Abb. 32: Kosten auf den einzelnen Rechnerknoten bei der Join-Kaskade

Je weiter der Join sich in der Kaskade befindet, um so höher ist seine Antwortzeit (siehe Abb. 32). Die höheren Antwortzeiten der am Ende der Kaskade ausgeführten Join-Operationen beruhen auf dem Einfluß der Antwortzeiten der vorhergehenden Join-Operationen.

Wenn eine kurze Antwortzeit für die Berechnung der Anfrageergebnisse gefordert ist, muß $HJoinV1$ verwendet werden. In den weiteren Betrachtungen wird, wenn nicht gesondert darauf hingewiesen wird, nur noch dieser Join betrachtet.

Die bisherigen Leistungsbetrachtungen ergeben, daß die innere Relation immer die kleinere der beiden Eingangsrelationen sein sollte, da hierbei die Hash-Tabelle kleiner

ist. Dadurch kann der Aufwand für die Verwaltung der Hash-Tabelle vermindert werden. Gegenüber einer großen, inneren Relation ist die Anzahl der Bucket-Aufspaltungen geringer. Es werden weniger Tupel im Hauptspeicher gehalten, so daß weniger Hauptspeicher für den Join erforderlich ist. Die Meßergebnisse bestätigen diese Erkenntnis. Die Abbildung 33 stellt den Join der Relation R4 mit den anderen Relationen dar (Join-Bedingung: $R_4.PK = R_i.PK$). Die innere Relation ist bei der einen Kurve jeweils die kleinere Relation, bei der anderen Kurve die äußere Relation.

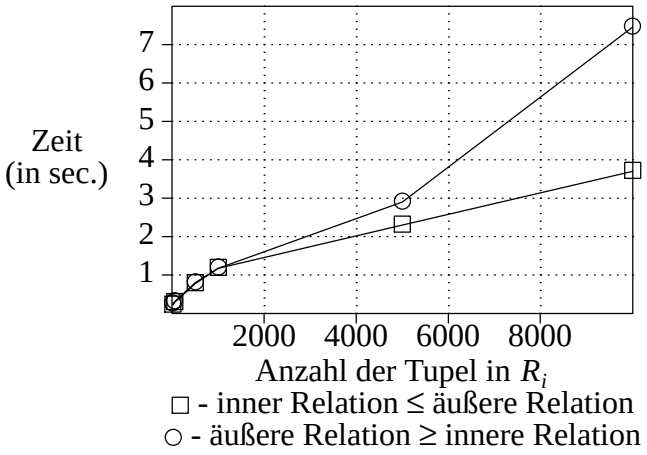


Abb. 33: Join der Relation $R4$ mit der Relation R_i

Die Kosten der Join-Operation werden desweiteren durch die Größe eines Tupelblocks beeinflußt. Die Größe des zu übertragenden Overheads sowie die Übertragungskosten für die Relationen werden durch die Tupelblockgröße beeinflußt. Die Abbildung 34 zeigt die Ausführungszeit für den Join über die Relationen R_i und R_i' (i = 3, 4), die Abbildung 35 für den Join über die Relationen R_5 und R_5', mit unterschiedlich großen Tupelblöcken (siehe Tab. 13). Die Join-Bedingung lautet $R_i.PK = R_i'.PK$.

Eine Tupelblockgröße, die bedeutend kleiner als die MTU (1024 Byte) ist, bewirkt zwar einen besseren Tupelstrom, führt aber zu hohen Ausführungskosten. Die Meßergebnisse führen bei den Beispielrelationen zu einer idealen Tupelblockgröße von 75 Tupeln (1875 Byte).

Die bisherigen Betrachtungen bezogen sich auf die Ausführung der Join-Phase des verteilten Joins, dem lokalen Join. Kleinere Eingaberelationen führen bei einem Join zu

Tupelanzahl	Bytes
10	240
25	625
50	1250
75	1875
100	2400

Tab. 13: Die verschiedenen Tupelblockgrößen

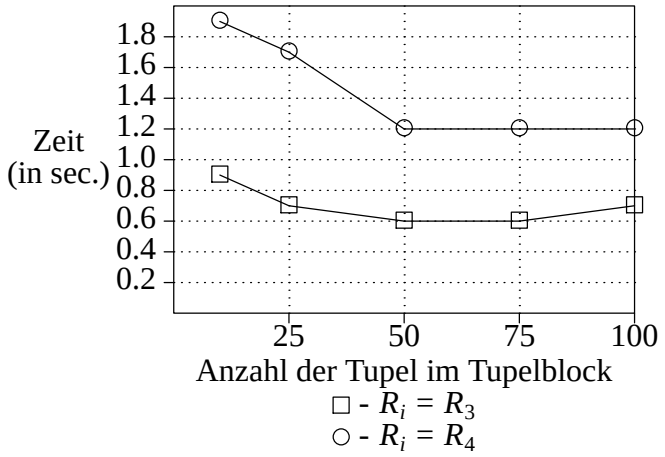


Abb. 34: Join über die Relationen R_i mit unterschiedlicher Tupelblockgröße

niedrigeren Ausführungskosten (siehe Abb. 31). Ein verteilter Join erreicht eine günstigere Antwortzeit durch die Ausführung mehrerer Joins über Partitionen der Eingangsrelationen (siehe Abb. 7). Die Abbildung 36 stellt die Kosten für die Ausführung der Join-Operation † über die Beispielrelationenm dar. Der Join wurde über die Gesamtrelationen und über partitionierte Relationen ausgeführt. Von den Kosten der Join-Phasen wurde ein Durchschnittswert gebildet.

Aus den Kurvenverläufen der Abbildung 36 wird deutlich, daß sich die Antwortzeit für einen verteilten Join durch eine Aufteilung der Gesamtrelation auf mehrere Partitionen, die parallel verbunden werden, verringert.

† Join-Bedingung $R_i.PK = R_i'.PK$

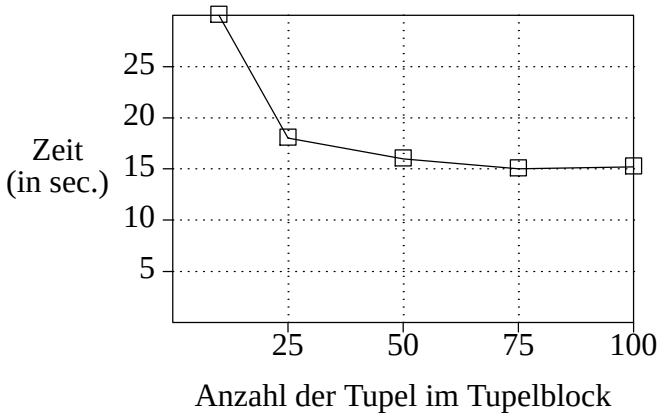


Abb. 35: Join über die Relationen R_5 mit unterschiedlicher Tupelblockgröße

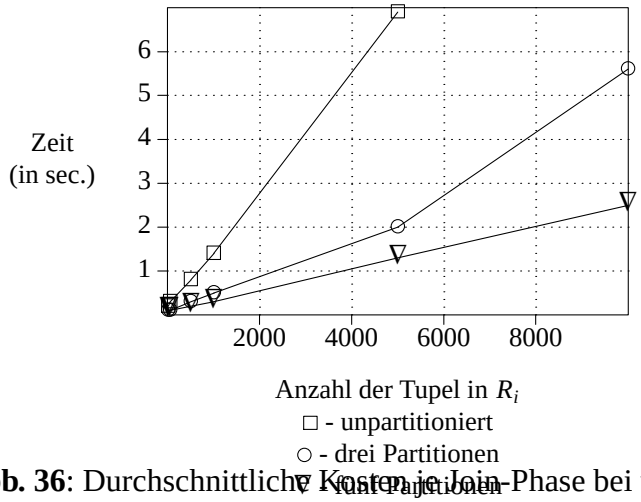


Abb. 36: Durchschnittliche Kosten je Join-Phase bei verteilten Joins

Die Kosten der Partitionierungsphase sind mit denen der Selektion vergleichbar. Wenn zusätzlich während der Partitionierung statistische Informationen über die Attributwertverteilung ‡ ermittelt werden müssen, sind die Partitionierungskosten höher als die Selektionskosten.

‡ siehe "2.2. Partitionierung" und "6.1. Mögliche Realisierung der Join-Operationen in verteilten Systemen"

8. Ausblick

Im Rahmen dieser Arbeit konnten nicht alle Probleme behandelt werden, die mit der Realisierung der DRAM in Zusammenhang stehen. Das liegt einerseits daran, daß erforderliche Komponenten in der HEAD-Umgebung gegenwärtig nicht zur Verfügung stehen. Andererseits konnte wegen des geringen Bezugs zur Aufgabenstellung innerhalb des zeitliche Rahmens dieser Arbeit nicht darauf eingegangen werden.

In HEAD werden die DRAM-Prozesse parallel ausgeführt. Die Kooperation zwischen den Prozessen erfolgt durch Kommunikation. Ein Tupelprotokoll realisiert die notwendigen Nachrichtenübertragungen. Die Betrachtung der Transportprotokolle für den Entwurf einer Protokollhierarchie zur Tupelübertragung ergab, daß RDP das am besten geeignete Transportprotokoll ist. Zur Zeit steht RDP in der HEAD-Umgebung nicht zur Verfügung. Aus diesem Grund wird UDP als Transportprotokoll verwendet. UDP führt jedoch keine sichere Nachrichtenübertragung durch. Ein Schiebefensterprotokoll gewährleistet die sichere Tupelübertragung in der DRAM. Das Schiebefensterprotokoll hat wesentlichen Einfluß auf die Kosten der Tupelübertragung. Es ist wegen der Realisierung durch einen separaten Prozeß im Nutzerbereich ineffizienter als das Schiebefensterprotokoll des RDP, das im Betriebssystemkern realisiert wird. Durch die Verwendung von RDP anstelle von UDP als Transportprotokoll sinken die Kosten für die Tupelübertragung.

Das **T**uple **B**lock **P**resentation **P**rotocol (TBPP) führt die Transformation der rechnerinternen Datendarstellung in eine systemunabhängige Datendarstellung und umgekehrt durch. Dafür werden die UNIX-Systemfunktionen für die Umwandlung von Basistypen verwendet. Für die systemunabhängige Beschreibung komplexer Daten und Datenstrukturen empfiehlt sich die Verwendung von ASN.1. Das ISODE-Paket oder spezielle ASN.1 Compiler stellen Funktionen für die erforderlichen Transformationen zur Verfügung.

Zur Speicherung von Zwischenergebnissen wird der vom UNIX-System zur Verfügung gestellte virtuelle Speicherbereich (Memory mapped file) verwendet. Dadurch entfällt bei der Realisierung der DRAM des HEAD-Prototypen die Berücksichtigung der Seitenauslagerungsstrategie, eine Aufgabe, die eigentlich durch das Datenbankverwaltungssystem übernommen wird. In zukünftigen Untersuchungen sollte ein Effizienzvergleich

der Eigenimplementierung der Seitenauslagerung mit der Strategie des Mmap vorgenommen werden.

Die kostenintensivste Algebra-Operation ist der Join. Die Betrachtung der Join-Verfahren in HEAD bezieht sich auf Equi-Joins. Ihre Berechnung erfolgt am effizientesten durch Hash join-Verfahren. Die Verwendung von Hash join-Verfahren für die Berechnung von Non-Equi-Joins ist nicht sinnvoll. Die Non-Equi-Joins werden am günstigsten durch Sort merge join-Verfahren [DeWi91] berechnet.

Die parallele Ausführung von Hash join-Verfahren bewirkt geringere Antwortzeiten. Der Partitionierungs-Operator zerlegt eine Relation in mehrere Teilrelationen. Durch geeignete Algorithmen können diese Teilrelationen parallel und unabhängig voneinander verbunden werden. Ziel der Partitionierung ist eine ausgeglichene Balancierung der Last auf die Rechner. Die Partitionierung bei den parallelen Hash join-Verfahren in HEAD erfolgt durch Hash-Partitionierung. Bei einer ungleichen Attributwertverteilung wird eine ausgeglichene Verteilung der Tupel auf die Partitionen durch Bereichspartitionierung erreicht. Dafür wird ein spezieller Split-Operator zur Verfügung gestellt, der anhand eines Partitionierungsvektors eine balancierte Aufteilung der Tupel auf die Teilrelationen vornimmt. Für die Ermittlung eines optimalen Partitionierungsvektors sind statistische Betrachtungen über die Attributwertverteilung erforderlich, die während der Scan-Operation oder zur Zeit der Partitionierung erfolgen. Dieses Problem konnte in dieser Arbeit nur ansatzweise betrachtet werden.

Die Verwendung von Leichtgewichtsprozessen zur Realisierung von erweiterten relationalen Algebraoperationen wurden ausführlich erläutert. Leichtgewichtsprozesse konnten für die Implementierung der DRAM nicht berücksichtigt werden, da eine Thread-Bibliothek mit Unterstützung des Thread-Pipeline-Modells in der HEAD-Umgebung nicht zur Verfügung stand.

Aus diesen Betrachtungen ergeben sich folgende weiterführenden Aufgabenstellungen:

(1) Realisierung von Non-Equi-Join-Verfahren, die vertikale und horizontale Parallelität zulassen.

(2) Ermittlung optimaler Partitionierungsvektoren aufgrund statistischer Betrachtungen über die Verteilung der Join-Attribute unter Berücksichtigung der parallelen Partitionierung von Fragmenten einer Relation.

(3) Entwurf einer C++-Klassenhierarchie zur Realisierung von Threads mit Unterstützung der Ausführung von relationalen Algebraoperationen nach dem Thread-Pipeline-Modell.

(4) Entwurf und Realisierung von Seitenauslagerungsstrategien zur effizienten Unterstützung der DRAM-Operationen.

(5) Entwurf eines Datendarstellungsprotokolls auf der Basis von ASN.1.

Läuc91.

Bent88.

Bentley, J., *More Programming Pearls - Confessions of a Coder,* Addison-Wesley, Reading, M. A. (1988).

Brat84.

Bratbergsengen, K., “Hashing Methods and Relational Algebra Operations” in *Proceedings of the 10th VLDB Conference,* pp. 323-333, Singapore (Aug. 1984).

Ceri85.

Ceri, S. and Pelagatti, G., “Distributed Databases: Principles and Systems,” *McGraw-Hill,* New York (1985).

Codd90.

Codd, E. F., *The relational model for database management: version 2,* Addison-Wesley, Reading, M. A. (1990).

DeWi91.

DeWitt, D. J., Naughton, J. F., and Schneider, A. S., “An Evaluation of Non-Equijoin Algorithms,” TR 1011, Computer Sciences Department, University of Wisconsin-Madison (February 1991).

DeWi92a.

DeWitt, D. J. and Gray, J., “Parallel Database Systems: The Future of High Performance Database Processing,” TR 1079, Computer Sciences Department, University of Wisconsin-Madison (February 1992).

DeWi92b.

DeWitt, D. J., Naughton, J. F., Schneider, D. A., and Seshadri, S., “Practical Skew Handling in Parallel Joins,” TR 1098, Computer Sciences Department, University of Wisconsin-Madison (July 1992).

DeWi84.

DeWitt, D. J., et. al., *Implementation Techniques for Main Memory Database Systems,* Proceedings of 1984 SIGMOD Conference, Boston, MA (June 1984).

Flac93.

Flach, G., Langer, U., and Meyer, H., “Das Projekt HEaD,” *Forschungsbericht (DFG),* Universität Rostock, Fachbereich Informatik, Arbeitsgruppe Datenbanken, Rostock (1993).

Gora90.

Gora, B. and Speyerer, R., *ASN.1 - Abstract Syntax Notation One,* DATACOM - Verlag, Bergheim (1990).

Grau94.

Graupner, S., “Coroutinen und preemptive Threads in C,” *Technische Universitaẗ Chemnitz-Zwickau, Fakultaẗ fur̈ Informatik, Lehrstuhl Systemprogrammierung und Betriebssysteme,* Chemnitz (1994).

Haas89.

Haas, L. M., Freytag, J. C., Lohman, G. M., and Pirahesh, H., “Extensible Query Processing in Starburst,” *ACM ToDS,* pp. 377-388 (Apr. 1989).

Zeda90.

Harget, A. J. and Johnson, J. D., “Load Balancing Algorithms in loosely-coupled Distributed Systems: a Survey” in *Zedan, H. S. M. "Distributed Computer Systems", Butterworths,* pp. 85-108, London (1990).

Hero92.

Herold, H., *UNIX-Grundlagen: UNIX und seine Werkzeuge; Kommandos und Konzepte,* Addison-Wesley, Reading, M. A. (1992).

Hua91.

Hua, K. A. and Lee, C., *Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning,* Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona (September 1991).

Kaas9?.

Kaashoek, M. F., van Renesse, R., van Staveren, H., and Tanenbaum, A. S.,

"FLIP: an Internetwork Protocol for Supporting Distributed Systems," *Vrije Universiteit,* Amsterdam (accepted for publication).

Kalf88.

Kalfa, W., *Betriebssysteme,* Akademie-Verlag, Berlin (1988).

Kern92.

Kerner, H., "Rechnernetze nach OSI," *Addison-Wesley,* Reading, M. A. (1992).

Kilg89.

Kilger, C., Hinrichs, C., and Fritz, T., *Alternative Realisierungsstrategien fuer Parallelisierung von Join-Algorithmen,* Studienarbeit, Universitaet Karlsruhe (1989).

Kits83.

Kitsuregawa, M., Tanaka, H., and Moto-oka, T., *Application of Hash to Data Base Machine and Its Architecture,* New Generation Computing, Vol. 1, No. 1 (1983).

Krus94.

Kruse, H.-H., "DRAM Quelltext-Dokumentation," *Universitaet Rostock, Fachbereich Informatik, Arbeitsgruppe Datenbanken,* Rostock (1994).

Lang91.

Langer, U., "Implementierung eines Execution Monitors," *Studienarbeit,* Universitaet Rostock, FB Informatik (1991).

Link93.

Linke, A., "Kriterien der Lastmessung und der Lastbalancierung," *Studienarbeit,* Universitaet Rostock, Fachbereich Informatik, Arbeitsgruppe Datenbanken (Rostock 1993).

Lock87.

Lockemann, P. C. and Schmidt, J. W., *Datenbank-Handbuch,* Springer Verlag, Tokyo (1987).

Mehl88.

Mehlhorn, K., *Datenstrukturen und effiziente Algorithmen - Band 1: Sortieren und Suchen,* B. G. Teubner, Stuttgart (1988).

Meye92.

Meyer, H., "`libipc` − A C++-class library for process interaction and communication," *Technical Report*, 2-92, University of Rostock, Dept. of Computer Science, Database Research Group (1992).

Mikk88.

Mikkilineni, K. P. and Su, S. Y. W., "An Evaluation of Relational Join Algorithms in a Pipelined Query Processing Environment," *IEEE Trans. on Software Eng.*, 14, 6nd, pp. 838-848 (Jun. 1988).

Ozsu91.

Ozsu, M.T. and Valduriez, P., "Principles of distributed database systems" in *Prentice-Hall, Inc., Englewood Cliffs* (1991).

Post80.

Postel, J., "User Datagram Protocol," *RFC 768* (Aug. 1980).

Post81a.

Postel, J., "Transmission Control Protocol," *RFC 793* (Sept. 1981).

Post81b.

Postel, J., "Internet Protocol," *RFC 791* (Sept. 1981).

Powe91.

Powell, M. L., Kleimann, S. R., Barton, S., Shah, D., Stein, D., and Weeks, M., "SunOS Multi-thread Architecture," *USENIX Winter 1991*, Sun Microsystems Inc., Dallas, TX (1991).

Roch88.

Rochkind, M. J., *UNIX Programmierung fuẗ Fortgeschrittene*, Carl Hanser Verlag, Muñchen (1988).

Saue91.

Sauer, H., *Relationale Datenbanken, Theorie und Praxis*, Addison Wesley (Deutschland) GmbH, Muñchen (1991).

Schi78.

Schiemangk, H., *in Schriftenreihe Informationsverarbeitung: Virtuelle Speicher - Dynamische Hauptspeicherverwaltung durch Betriebssysteme*, BSB B.G Teubner

Verlagsgesellschaft, Leipzig (1978).

Schn89.

Schneider, D. A. and DeWitt, D. J., “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment,” TR 836, Computer Sciences Department, University of Wisconsin-Madison (April 1989).

Schn91.

Schneider, H.-J., *Lexikon der Informatik und Datenverarbeitung,* R. Oldenburg Verlag, Wien (1991).

Slom89.

Sloman, M. and Kramer, J., “Verteilte Systeme und Rechnernetze,” *Carl Hanser Verlag,* Wien, Muñchen (1989).

Ster92.

Sterken, A. and Pawlita, P., “Referenzportierung von OSF/DCE auf Unix SVR4 - Grundstein gelegt,” *UNIX-Magazin Heft 12/92,* pp. 55-59 (Dez. 1992).

Stev92.

Stevens, W. R., “Programmieren von UNIX-Netzen - Grundlagen, Programmierung, Anwendungen,” *Verlag Carl Hanser,* Muñchen (1992).

Stro91a.

Stroustrup, B., *The C++ Programming Language (Second Edition),* Addison-Wesley, Reading, Massachusetts (1991).

Stro91b.

Stroustrup, B., “What is ‘‘Object-Oriented Programming’’?,” *Computing Science Technical Report,* 116, AT&T Bell Laboratories, Murray Hill, New Jersey (May 1991).

Tane81.

Tanenbaum, A. S., “Computer Networks,” *Prentice Hall,* Eaglewood Cliffs, NJ (1981).

Tane90.

Tanenbaum, A. S., *Betriebssysteme - Entwurf und Realisierung,* Prentice-Hall International Inc., London (1990).

Tane92.

Tanenbaum, A. S., *Modern Operating Systems,* Prentice-Hall International Editions, Englewood Cliff, New Jersey (1992).

Velt84.

Velten, D., Hinden, R., and Sae, J., "Reliable Data Protocol," *RFC 908* (July 1984).

Wiso90.

Wisotzki, A., "Parallele Anfrageverarbeitung in HEaD," *Forschungsbericht,* 1, Universiat Rostock, Fachbereich Informatik (1990).

Wolf93.

Wolf, J. L., Yu, P. S., Turek, J., and Dias, D. M., "A Parallel Hash Join Algorithm for Managing Data Skew," *IEEE Transactions On Parallel and Distributed Systems Vol. 4,* pp. 1355-1371 (Dec. 1993).

Wood93.

Wood, A., *Data Structures, Algorithms and Performance,* Addison Wesley, Reading, Massachustts (1993).

Zell90.

Zeller, H. and Gray, J., "An Adaptive Hash Join Algorithm for Multiuser Environments" in *Proceedings of the 16th VLDB Conference*, Brisban, Australia (1990).

Abbildungsverzeichnis

Tabellenverzeichnis

Erklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 01.04.1994

Inhaltsverzeichnis