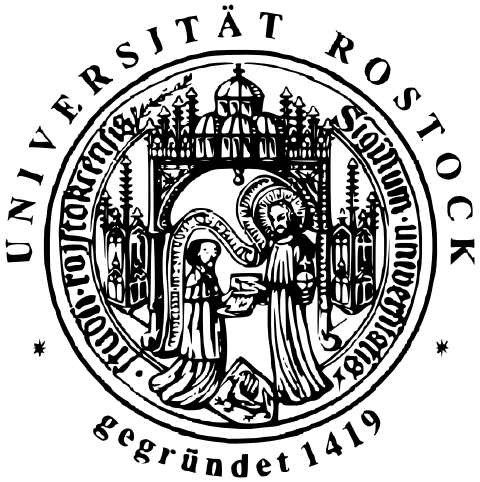

Anfrageoptimierung in Data Warehouses

durch Verwendung voraggregierter Views

Diplomarbeit

Fakultät für Informatik und Elektrotechnik
Lehrstuhl Datenbanken und Informationssysteme
Universität Rostock

Vorgelegt von: Nösinger, Thomas
Geboren am: 8. Juli 1982 in Schwerin
Matr.-Nr.: 3201812

Betreuer:

- Dr.-Ing. habil. Meike Klettke

Gutachter:

- Prof. Dr. rer. nat. habil. Andreas Heuer
- Prof. Dr.-Ing. habil. Peter Forbrig

Abgabedatum: 9. Januar 2009

Zusammenfassung

In der vorliegenden Diplomarbeit wird ein Verfahren vorgestellt, mit dem Aggregatanfragen an ein relational verwaltetes Data Warehouse optimiert werden. Mittels voraggregierter Relationen, welche das Ergebnis von Aggregatanfragen sind und materialisierte Views genannt werden, werden Anfragen an das Data Warehouse analysiert und umgeschrieben. Dafür müssen Metadaten über die Views vorliegen, die in einem umgesetzten Relationenschema verwaltet werden. Darüber hinaus wird eine Kostenfunktion definiert, mit der entschieden wird, welche materialisierte View zur Optimierung verwendet werden soll. Weiterhin wird ein Verfahren erläutert, mit dem materialisierte Views bei Änderungen der Daten des Data Warehouses aktualisiert werden.

Abstract

The diploma thesis presents a technique for optimizing aggregation-queries to relational data warehouses. With the use of preaggregated relations, which are the results of former aggregation-queries and named as materialized views, queries to data warehouses are analyzed and reformulated. Therefore metadata are needed, which are stored in an implemented relation-schema. Furthermore a cost function is defined, that determines the materialized view used for optimization. A maintenance technique for updating materialized views after changing data of the data warehouse is also presented.

Einordnung

ACM Computing Classification System (1998): H.3.3, H.3.4
Keywords: materialisierte Views, Anfrageoptimierung, Aggregatanfragen, Kostenfunktion, Data-Warehouse, Star-Schema, Metadaten-Verwaltung

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Ein *Data Warehouse* bzw. Datenlager enthält multidimensionale Daten aus unterschiedlichen Quellen, welche zentralisiert zur Analyse und Entscheidungsunterstützung gesammelt und ausgewertet werden.

Im Mittelpunkt eines Data Warehouses stehen *Kennzahlen*, welche als verdichtete Messgrößen betriebswirtschaftliche Zusammenhänge beschreiben. Typische Kennzahlen in Bezug auf ein Unternehmen sind: Umsatz, Kosten, Ausgaben, ... Diese Kennzahlen können, je nach Anwendung und Bedürfnissen eines Unternehmens, aus unterschiedlichen Perspektiven betrachtet werden. Perspektiven sind in diesem Zusammenhang *Dimensionen*, welche eine Sicht auf die zugeordneten Kennzahlen beschreiben. Das heißt, dass zum Beispiel der Umsatz einer bestimmten Region, die Kosten in einem angegebenen Jahr oder auch die Ausgaben von Rentnern betrachtet werden sollen. Die Dimensionen sind im vorangegangenen Beispiel: Ort (Region), Zeit (Jahr) und Kunde (Rentner).

Die Kennzahlen und Dimensionen können auf unterschiedliche Art und Weise verwaltet und gespeichert werden. Es existieren diesbezüglich Formalismen und Datenmodelle, die eine effektive Speicherung mit gleichzeitigem effizienten Zugriff ermöglichen sollen. Im Umfeld relationaler Datenbanken wäre dies zum Beispiel das *Star-Schema*, mit dem multidimensionale Daten relational gespeichert werden. Das Star-Schema enthält eine zentrale *Faktentabelle*, in welcher die Kennzahlen gespeichert werden, und für jede Dimension eine *Dimensionstabelle*. Die Faktentabelle und Dimensionstabellen werden *Basisrelationen* genannt und mittels Schlüssel-/Fremdschlüsselbeziehungen verknüpft.

Typische Anfrage an ein solches Star-Schema sind multidimensionale *Aggregatanfragen*. Das heißt, dass eine Anfrage aggregierte Kennzahlen aus der Faktentabelle und unterschiedliche Dimensionstabellen selektiert. Dabei ist zu beachten, dass aufgrund der Struktur des Star-Schemas die Faktentabelle aus einer riesigen Anzahl von Tupeln besteht, welche potentiell nicht zur Beantwortung einer Anfrage benötigt werden. Zum Beispiel könnte ein Data Warehouse eines Unternehmens Datensätze über deren Umsatz der letzten 10 Jahre gesammelt und relational gespeichert haben. Dieses Data Warehouse wird mit einer Aggregatanfrage abgefragt, die den Umsatz eines angegebenen Jahres selektiert. Dass heißt, dass, vereinfacht gerechnet, 90% der vorhandenen Datenbestände nicht zur Beantwortung benötigt werden, aber dennoch müsste die komplette Faktentabelle geladen und analysiert werden.

Angenommen, obige Aggregatanfrage würde häufiger an das Data Warehouse gestellt werden. Dann wäre es in diesem Szenario sinnvoll, das Ergebnis der Anfrage relational zu speichern, und beim Aufruf der entsprechenden zugehörigen Aggregatanfrage

das voraggregierte Ergebnis zurückzugeben. Solche voraggregierten Relationen werden *materialisierte Views* genannt. Diese materialisierten Views enthalten weit weniger Tupel als die Basisrelationen, können aber die zur Beantwortung einer Anfrage benötigten Informationen beinhalten. Somit könnte auf den kostenintensiven Zugriff auf die Faktentabelle verzichtet und das gleiche Ergebnis in einer geringeren Antwortzeit zurückgeliefert werden.

Es ergibt sich diesbezüglich die allgemeine Frage: „Wie können materialisierte Views zur Optimierung von Anfragen verwendet werden?“. In der vorliegenden Diplomarbeit wird diese Problemstellung thematisiert. Das heißt, es wird erläutert, wie Anfragen an ein Data Warehouse durch Verwendung materialisierter Views optimiert werden können.

1.2. Problemstellung

Bei der Optimierung von Aggregatanfragen unter Verwendung materialisierter Views sind unterschiedliche Problemstellungen zu betrachten. Dazu zählen:

- Analyse der gestellten Aggregatanfragen,
- Verwaltung der Metadaten materialisierter Views,
- Vergleich materialisierter Views,
- Restrukturierung der gestellten Anfrage,
- Wartung materialisierter Views.

Die *Analyse der gestellten Aggregatanfragen* ist der Einstiegspunkt in eine mögliche Optimierung. Es müssen, vorausgesetzt materialisierte Views sind vorhanden, die unterschiedlichen Bestandteile einer Aggregatanfrage erkannt, zugeordnet und verglichen werden. Das heißt, dass unter anderem untersucht werden muss, welche Dimensionen, Kennzahlen etc. in einer Aggregatanfrage verwendet werden, und wie diese im Verhältnis zu einer materialisierten View stehen. Zum Beispiel kann eine View nur zur Optimierung genutzt werden, wenn passende Dimensionen selektiert wurden und die notwendigen Kennzahlen vorhanden sind.

Somit wird eine *Verwaltung der Metadaten materialisierter Views* benötigt. Es müssen Informationen über die materialisierten Views gespeichert werden, sodass diese mit den vorhandenen Bestandteilen einer gestellten Aggregatanfrage verglichen werden können. Dabei kann die Situation eintreten, dass mehr als eine View zur Beantwortung verwendet werden kann.

In diesem Fall ist der *Vergleich materialisierter Views* erforderlich. Es muss eine Kostenfunktion entwickelt werden, die jeder View einen Wert zuordnet, sodass die Views untereinander verglichen werden können. Mit Hilfe einer solchen Funktion könnte somit die kostengünstigste View ermittelt werden, mit der eine Anfrage an ein Data Warehouse optimiert wird. Dabei ist es allerdings auch denkbar, dass die Datensätze unterschiedlicher Views verwendet werden könnten. Das heißt, dass zum Beispiel die

aggregierten Kennzahlen unterschiedlicher Views gemeinsam die selektierten Kenngrößen der Anfrage liefern oder auch nachträglich berechnen.

Nachdem eine oder mehrere materialisierte Views ausgewählt wurden, muss die gestellte Anfrage umgeschrieben werden, es kommt somit zur *Restrukturierung der gestellten Anfrage*. Eine restrukturierte Anfrage ist dabei eine Aggregatanfrage, die materialisierte Views verwendet, anstatt auf die Basisrelationen, speziell auf die Faktentabelle, zuzugreifen.

Eine materialisierte View ist das Ergebnis einer Aggregatanfrage, welche an das gleiche Star-Schema gestellt wurde wie die zu optimierende Anfrage. Falls die entsprechenden Basisrelationen geändert werden, dann müssen die materialisierten Views ebenfalls angepasst bzw. aktualisiert werden. Die *Wartung materialisierter Views* ist daher ebenfalls notwendig, damit die Views keine veralteten Datensätze enthalten.

1.3. Aufbau der Diplomarbeit

Die unterschiedlichen Problemstellungen aus Abschnitt 1.2 werden in der vorliegenden Diplomarbeit behandelt. Diesbezüglich werden zuerst Grundlagen in **Kapitel 2** geschaffen. Es wird geklärt, was ein Data Warehouse ist und wie in einem solchen Daten gespeichert und analysiert werden können. Im Anschluss daran wird auf die Eigenschaften und die Struktur von multidimensionalen Anfragen an relational verwaltete Datenbestände eingegangen.

Im **Kapitel 3** werden die Arbeiten anderer Autoren vorgestellt, geprüft und bewertet, sodass der aktuelle Stand der Forschung veranschaulicht wird. Des Weiteren werden aus den Veröffentlichungen der Autoren Rückschlüsse für die vorliegende Arbeit gezogen.

Im darauf folgenden **Kapitel 4** wird unter Verwendung der eingeführten Terminologie ein Konzept entwickelt, mit dem Aggregatanfragen an ein Star-Schema mit Hilfe materialisierter Views optimiert werden können. Es wird detailliert beschrieben, wie entschieden werden kann, welche materialisierten Views zur Optimierung verwendet werden können, wie diese mit Hilfe einer entwickelten Kostenfunktion ausgewählt werden und wie die Restrukturierung einer Anfrage abläuft.

Im **Kapitel 5** wird danach anhand eines Beispiels der Optimierungsprozess einer Anfrage vorgestellt und erläutert werden. Dabei wird ebenfalls dargestellt, wie die Metadaten von Views in einer relationalen Datenbank gespeichert werden und wie materialisierte Views genau aktualisiert werden können.

Im Anschluss wird im **Kapitel 6** der Inhalt der vorliegenden Arbeit zusammengefasst. Des Weiteren wird in einem Ausblick dargestellt, welche offenen Probleme und weiterführenden Fragestellungen existieren.

2. Technische Grundlagen

2.1. Data Warehouse

Ein Data Warehouse ist eine themenorientierte, integrierte, persistente und zeit-variante Sammlung von Daten zum Zweck der Analyse und Entscheidungsunterstützung. Eine Definition des Begriffes des Data Warehouses, auch Datenlager genannt, stammt von Inmon:

> *„A data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions [IWIP02].“*

Die genannten Eigenschaften des Data Warehouses sollen nachfolgend kurz beschrieben werden. *Themenorientiert* bedeutet, dass der Zweck der Datenbasis nicht mehr auf der Erfüllung einer Aufgabe wie einer Personaldatenverwaltung liegt, sondern auf der Modellierung eines spezifischen Anwendungsziels [BG04]. Es werden demnach Daten gesammelt, die für Analysen und Entscheidungsprozesse wichtig sind und nicht für den operativen Betrieb. Die Datensammlung soll *zeit-variant* sein, damit Analysen über zeitliche Veränderungen stattfinden können. Um solche Analysen zu ermöglichen, ist die persistente Speicherung über lange Zeiträume notwendig. *Persistent* bedeutet, dass die ins Data Warehouse eingebrachten Daten „nicht mehr entfernt oder geändert [BG04]“ werden. Die letzte Eigenschaft ist, dass die Daten *integriert* werden. Das heißt, die Daten werden aus verschiedenen Quellen im Data Warehouse zusammengefügt. Diese Quellen können zum Beispiel einzelne operative Systeme eines Unternehmens sein, deren heterogene Daten per ETL-Prozess im Data Warehouse zusammengeführt und dauerhaft gespeichert werden.

2.1.1. ETL-Prozess

ETL ist ein Akronym für Extraktion, Transformation und Laden, und soll die „kontinuierliche Datenversorgung des Data Warehouses [Kle07]“ gewährleisten. Des Weiteren sichert der ETL-Prozess die Konsistenz des Data Warehouses gegenüber der verschiedenen Datenquellen [Kle07].

Extraktion

Die Extraktion wählt die Daten einer Quelle aus, die im Data Warehouse gespeichert werden sollen. In diesem Zusammenhang muss sowohl entschieden werden zu welchem Zeitpunkt eine Extraktion erfolgen soll als auch welche Daten extrahiert werden sollen.

Die Qualität der extrahieren Daten ist von entscheidender Bedeutung [HGZS02]. Dabei wird die Qualität daran gemessen, ob die Daten der Quellen komplett, eindeutig, aussagekräftig und korrekt sind [HH02].

Transformation

Die extrahierten Daten werden transformiert, das heißt sie werden unter anderem bereinigt und korrigiert. Die Bereinigung und Korrektur sind notwendig, damit eine einheitliche und strukturierte Informationsmenge im Data Warehouse entstehen kann. Die Transformation verbessert die Qualität der Daten und beseitigt die unterschiedlichen Konflikte, welche bei der Zusammenführung von Daten heterogener Quellen entstehen können. Zum Beispiel könnte ein internationales Unternehmen verschiedene Datenbanken in unterschiedlichen Ländern betreiben, die den Umsatz der lokalen Filialen speichern. Mögliche Konflikte sind in diesem Szenario, dass unterschiedliche Währungen verwendet werden. Das heißt bei der Transformation sollte eine Umrechnung in eine einheitliche Währung erfolgen, damit Daten untereinander verglichen werden können. Weitere Konflikte resultieren aus der Autonomie der einzelnen Datenbanken. Zum Beispiel könnte in einer Datenbank eine Relation Kunde enthalten sein, während die semantisch gleiche Relation in einer anderen Datenbank Käufer heißt. Solche Namenskonflikte werden im Transformationsprozess zum Beispiel durch die Verwendung von Lexika behoben, die Käufer als Synonym von Kunde ausweisen. Ein weiterer Konflikt kann durch die unterschiedliche Speicherung der Daten entstehen. Gesetzt den Fall das Unternehmen speichert Informationen über das Geschlecht der Kunden. Dieses kann durch ein zusätzliches Attribut Geschlecht in der Relation Kunde erfolgen, durch mehrere Attribute wie Frau und Mann oder auch durch eine oder mehrere zusätzliche Relationen. Solche Schemakonflikte werden zum Beispiel durch das Schema Matching behandelt, das heißt das Finden gleicher oder ähnlicher Bestandteile in unterschiedlichen Quellen.

Laden

Der letzte Schritt des ETL-Prozesses ist das Laden der aufbereiteten Daten in das Data Warehouse. Das heißt die durch die Transformation bereinigten und korrigierten Daten werden im Data Warehouse zusammengeführt.

2.1.2. Datenmodell

Ein Data Warehouse besitzt ein multidimensionales Datenmodell, das die Analyse unterstützen soll. In diesem Zusammenhang werden die Begriffe der Kennzahlen, Dimensionen und des Datenwürfels eingeführt, welche nachfolgend skizziert werden sollen.

Kennzahl

Eine Kennzahl ist eine verdichtete Messgröße, die betriebswirtschaftliche Zusammenhänge beschreibt. Solche Kennzahlen können zum Beispiel die Kosten, Verkäufe, der Umsatz etc. einer Firma sein.

Dimension

„Eine Dimension ist innerhalb des multidimensionalen Datenmodells eine ausgewählte Entität, mit der eine Analysesicht eines Anwendungsbereichs definiert wird [BG04]." Eine Dimension ist somit die Beschreibung einer Sicht auf die zugeordneten Kennzahlen und dient der orthogonalen Strukturierung des Datenraums [Kle07].

Dimensionen bilden Klassifikationshierarchien, die mit Hilfe eines Klassifikationsschemas beschrieben werden. Die Einträge des Schemas werden Klassifikationsstufen bzw. Hierarchiestufen genannt, während die Elemente der Klassifikationshierarchie Klassifikationsknoten heißen. Die Elemente der niedrigsten Hierarchiestufe werden auch als Dimensionselemente bezeichnet. Des Weiteren sind auf der niedrigsten Hierarchiestufe die atomaren Werte der Kennzahlen hinterlegt [BG04]. Es existiert eine höchste, ausgezeichnete Klassifikationsstufe, welche die Aggregation aller niederen Stufen ist. Dieser Klassifikationsknoten kann zum Beispiel „Top" oder „All" heißen.

Die Abbildung 2.1 visualisiert eine Klassifikationshierarchie und deren Klassifikationsschema am Beispiel der Dimension Zeit.

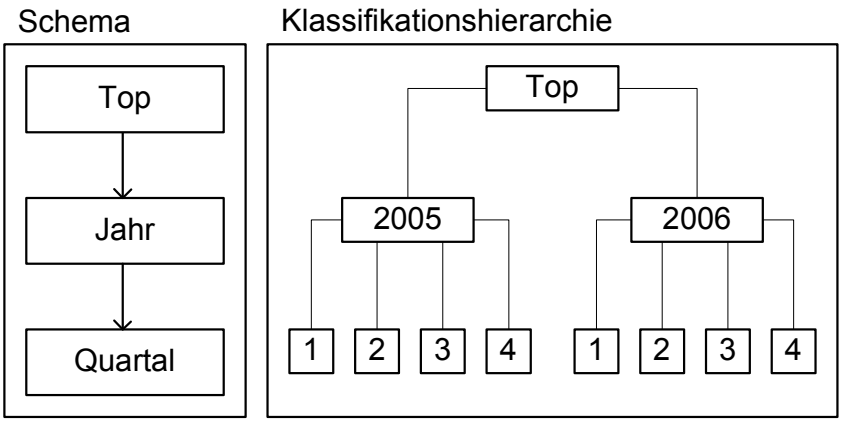

Abbildung 2.1.: Klassifikationsschema mit zugeordneter Klassifikationshierarchie

In der Abbildung ist ein Schema der Dimension Zeit dargestellt. Es enthält die Klassifikationsstufen Quartal, Jahr und die höchste, ausgezeichnete Stufe „Top". Eine mögliche Klassifikationshierarchie, die das Klassifikationsschema beschreibt, ist ebenfalls dargestellt. Die Klassifikationsknoten sind in dem Beispiel die Knoten „Top", 2006, 2007 und jeweils die Knoten 1, 2, 3 und 4. Die Knoten 1, 2, 3 und 4 werden auch als Dimensionselemente bezeichnet, da sie die Elemente der niedrigsten Hierarchiestufe

sind. Der Klassifikationsknoten „Top“ ist der höchste, ausgezeichnete Klassifikationsknoten.

Es wird unterschieden zwischen einfachen und parallelen Hierarchien [Kle07]. Während bei einfachen Hierarchien eine Klassifikationsstufe die Aggregation genau einer niedrigeren Klassifikationsstufe ist, treten bei parallelen Hierarchien unterschiedliche, nicht abhängige Hierarchiestufen auf. In parallelen Hierarchien können somit höhere Klassifikationsstufen durch die Aggregation verschiedener niedrigerer Klassifikationsstufen erhalten werden. Es entstehen verschiedene Pfade im Klassifikationsschema, welche Konsolidierungspfade genannt werden. In der Abbildung 2.2 wird ein Beispiel für eine einfache und parallele Hierarchie im Klassifikationsschema Zeit illustriert.

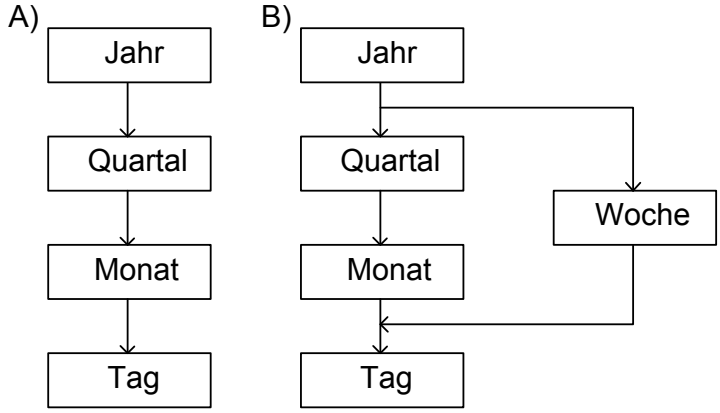

Abbildung 2.2.: Einfache und parallele Hierarchien im Klassifikationsschema Zeit

Beispiel A) zeigt eine einfache Hierarchie mit verschiedenen Klassifikationsstufen. Die Klassifikationsstufen können durch die Aggregation der jeweils niedrigeren Hierarchiestufe erhalten werden. Das heißt beim Zusammenfassen aller Tage wird ein Monat erhalten, durch Aggregation bestimmter Monate das jeweilige Quartal etc.. Beispiel B) zeigt eine parallele Hierarchie. Es existiert einerseits die einfache Hierarchie aus Beispiel A), allerdings enthält Beispiel B) die parallele Klassifikationsstufe Woche. Das heißt ein Jahr kann entweder durch die Aggregation der Wochen oder der Quartale erhalten werden. Dabei sind weder Wochen und Quartalen, noch Monate und Quartale voneinander abhängig. Durch die Aggregation von Monaten oder Quartalen kann somit in dem Beispiel aus Abbildung 2.2 nicht die Klassifikationsstufe Woche erhalten werden. Es existieren somit zwei Konsolidierungspfade. Einerseits ist es der Pfad, welcher die Hierarchiestufen Monat und Quartal aggregiert, andererseits der Pfad, der die Hierarchiestufe Woche aggregiert. In beiden Beispielen wurde auf die Darstellung der höchsten, ausgezeichneten Klassifikationsstufe verzichtet.

Datenwürfel

Der letzte Begriff, der eingeführt werden soll, ist der des Datenwürfels. Der Datenwürfel ist die „Grundlage für das mehrdimensionale Datenmodell [Kle07]“. Der Würfel

ist eine mehrdimensionale Matrix, deren Achsen durch die Dimensionen aufgespannt werden. Die einzelnen Zellen des Würfels repräsentieren eine oder mehrere Kennzahlen und können durch die Angabe der Dimensionselemente adressiert werden. Dabei gilt Orthogonalität zwischen den Dimensionen [Kle07]. Das heißt, es existieren keine funktionalen Abhängigkeiten zwischen den Klassifikationsknoten unterschiedlicher Dimensionen. Des Weiteren wird die Anzahl der Dimensionen, die die mehrdimensionale Matrix aufspannen, Dimensionalität genannt.

2.1.3. Speicherung

Das multidimensionale Datenmodell kann auf unterschiedliche Art in Datenbanksystemen umgesetzt bzw. intern verwaltet werden. Es wird diesbezüglich unterschieden zwischen relationaler und direkter multidimensionaler Speicherung der Daten [BG04]. In der vorliegenden Diplomarbeit wird die relationale Speicherung verwendet. Aus diesem Grund wird im Folgenden erläutert, wie das multidimensionale Datenmodell (siehe Abschnitt 2.1.2) auf das relationale Datenmodell abgebildet wird.

Anforderungen an die relationale Speicherung

Es werden unterschiedliche Anforderungen an die relationale Speicherung des multidimensionalen Datenmodells gestellt, welche Bauer et al. in [BG04] aufzeigen. Dazu zählen, dass

- eine „möglichst [BG04]“ verlustfreie Abbildung des multidimensionalen Datenmodells erfolgt,
- eine effiziente Übersetzung und Verarbeitung multidimensionaler Anfragen erfolgen kann und
- die Wartung der Relationen, das heißt zum Beispiel das Laden neuer Daten, einfach und schnell möglich ist.

Des Weiteren wird darauf hingewiesen, dass „klassische Techniken des Schemaentwurfs [...] nicht notwendigerweise das gewünschte Ergebnis liefern [BG04]“. Das heißt, auf Techniken wie zum Beispiel die Normalisierung wird bewusst verzichtet, um „Einsparungen von Verbundoperationen zu erlauben [BG04]“.

Abbildung der Strukturen des multidimensionalen Datenmodells

Die Abbildung 2.3 zeigt eine mögliche Abbildungsvorschrift des multidimensionalen Datenmodells auf relationale Strukturen.

Mit Hilfe dieser Abbildungsvorschrift können die verschiedenen Strukturen des multidimensionalen Datenmodells (siehe Abschnitt 2.1.2) umgesetzt werden. Kennzahlen und Dimensionen können demzufolge als Attribute von Relationen betrachtet werden, während die Zellen des Datenwürfels durch Tupel repräsentiert werden. Im Folgenden werden kurz Möglichkeiten skizziert, wie die noch nicht berücksichtigten Klassifikationshierarchien abgebildet werden können.

Multidimensionales Datenmodell	Relationale Abbildung
Kennzahl	Spalte
Dimension	Spalte
Zelle des Datenwürfels	Tupel

Abbildung 2.3.: Abbildung des Datenmodells auf relationale Strukturen

Fakten- und Dimensionstabellen

Es existieren verschiedene Schemata, mit denen die Klassifikationshierarchien in relationalen Datenbanken umgesetzt werden können. Diese Schemata unterscheiden sich untereinander durch die unterschiedliche Anzahl und Verwendung von Fakten- und Dimensionstabellen. Eine Faktentabelle ist eine Relation in der die Kennzahlen gespeichert werden. In den Dimensionstabellen werden hingegen die Informationen der Dimensionen gespeichert. Fakten- und Dimensionstabellen werden dabei mit Hilfe von Schlüssel/Fremdschlüsselbeziehungen verknüpft und Basisrelationen genannt. Ein konkretes Schemata ist das Star-Schema, welches nachfolgend beschrieben wird.

Star-Schema

Das Star- oder auch Sternschema hat „eine einfache und verständliche Struktur [BG04]“. Es enthält eine Faktentabelle und für jede Dimension eine Dimensionstabelle. Das Schema zeichnet sich durch die „einfache und flexible Darstellung von Klassifikationshierarchien [BG04]“ aus. „Die Klassifikationshierarchien können einfach innerhalb der Dimensionstabellen als Spalten angegeben werden [BG04]“. Die Abbildung 2.4 stellt ein Sternschema am Beispiel dar.

Das Schema enthält die Faktentabelle Umsatz, in der die Kennzahlen Kosten und Verkauf enthalten sind. Die anderen Attribute der Relation sind Fremdschlüssel der Dimensionstabellen, die zusammen den Primärschlüssel der Faktentabelle bilden. In der Abbildung werden drei Dimensionen dargestellt, welche sternförmig um die Faktentabelle angeordnet werden. Dies sind die Dimensionen Zeit, Kunde und Produkt. Jede Dimensionstabelle enthält dabei einen Primärschlüssel, welcher gleichzeitig als Fremdschlüssel in der zugeordneten Faktentabelle verwendet wird. Des Weiteren sind als Attribute die Klassifikationsstufen gespeichert. Für die Dimension Zeit sind das zum Beispiel die in Abbildung 2.2 gezeigten Stufen Tag, Monat, Quartal, Jahr und Woche.

Im Sternschema treten Redundanzen auf, da jedes Tupel für alle Klassifikationsstufen der Klassifikationshierarchie Einträge enthält. Es wird allerdings auf eine Normalisierung, zum Beispiel durch das Auslagern der einzelnen Klassifikationsstufen in eigene Relationen, verzichtet. Das Snowflake-Schema nimmt eine solche Normalisierung vor. Dieses Schema enthält eine Faktentabelle und für jede Klassifikationsstufe eine eigene Relation, die durch Fremdschlüsselbeziehungen untereinander verknüpft werden. Das Sternschema hat aber neben der einfachen Struktur und der flexiblen Darstellung

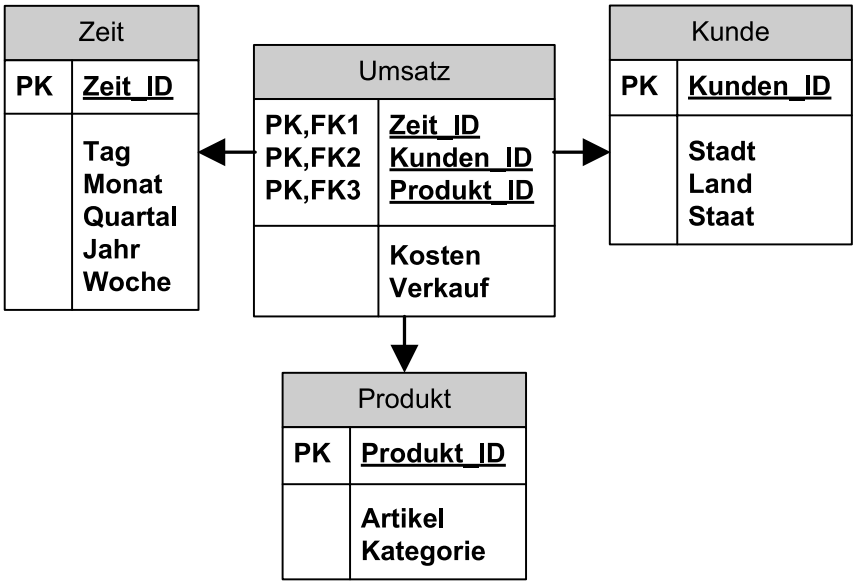

Abbildung 2.4.: Sternschema am Beispiel

der Klassifikationshierarchie den Vorteil, dass bei multidimensionalen Anfragen weniger Verbundoperationen zwischen den einzelnen Dimensionstabellen ausgeführt werden müssen. „Man weicht also an dieser Stelle bewusst von dem Prinzip der Normalisierung ab, um eine schnellere Anfragebearbeitung durch Einsparung von Verbundoperationen zu erlauben [BG04]." Ein anderer Kritikpunkt ist der Verlust der funktionalen Abhängigkeiten zwischen den einzelnen Klassifikationsstufen. Das heißt auf Ebene des Star-Schemas sind die Beziehungen der Klassifikationsstufen nicht erkennbar, was erneut zu Gunsten einer schnelleren Anfragebearbeitung geduldet wird.

2.1.4. Analyse und Entscheidungsunterstützung

Die im Data Warehouse zusammengeführten und gespeicherten Daten bilden eine zentrale, einheitliche und strukturierte Informationsmenge, die ausgewertet bzw. analysiert werden kann. Das Data Mining und das Online Analytical Processing (OLAP) sind zum Beispiel Methoden, mit denen die gespeicherten Daten analysiert werden können.

Das Data Mining ist die „Extraktion von interessanten (nicht-trivialen, impliziten, vorher unbekannten und potentiell nützlichen) Informationen oder Mustern aus Daten in großen Datenbanken [Kle07]". Es ist demnach die „Suche nach unbekannten Mustern oder Beziehungen in Daten [BG04]". Ein Unternehmen könnte zum Beispiel mit Hilfe des Data Minings versuchen, das Kaufverhalten von deren Kunden zu analysieren, um mit Hilfe der gewonnenen Erkenntnisse über die Einführung eines neuen Produkts zu entscheiden.

Online Analytical Processing (OLAP)

„OLAP ist die explorative, interaktive Analyse auf Grundlage des konzeptuellen, multidimensionalen Datenmodells [BG04]“. Der Ansatz des OLAP steht für eine Gattung von Anfragen, die nicht den Zugriff auf einzelne Werte sondern auf eine Vielzahl von Einträgen abzielt [BG04]. Das multidimensionale Datenmodell spielt im OLAP-Themengebiet eine zentrale Rolle [BG04]. Abhängig von der Speicherung des Datenmodells wird unterschieden zwischen verschiedenen Arten des OLAP. ROLAP bezeichnet zum Beispiel das relationale OLAP, das heißt das Datenmodell wird relational verwaltet. Weitere Arten des OLAP existieren, sollen aber nicht weiter betrachtet werden, da in der vorliegenden Diplomarbeit die relationale Speicherung verwendet wird.

Neben dem multidimensionalen Datenmodell sind auch spezifische Operationen fürs OLAP charakteristisch, mit denen der Anwender durch die multidimensionale Datenstruktur navigieren kann [BG04]. Diese Operationen beziehen sich auf den Datenwürfel (siehe 2.1.2) und sollen nachfolgend kurz erläutert werden. Der Datenwürfel der verwendeten Beispiele soll die Kennzahlen Verkauf und Kosten enthalten und durch die Dimensionen Zeit, Kunde und Produkt aufgespannt werden. Die Abbildung 2.4 illustriert dieses Beispiel.

- *Slice* schneidet einzelne Scheiben aus dem Datenwürfel, „indem eine Aggregation der Kenngrößen über einen Klassifikationsknoten einer Dimension stattfindet [Kle07]“. Zum Beispiel könnte ein Unternehmen an den Verkauf eines bestimmten Artikels in Abhängigkeit der Zeit und der Kunden interessiert sein.

- *Dice* betrachtet einen Teilwürfel des multidimensionalen Datenwürfels. Ein Beispiel ist der Verkauf eines Artikels in einem Jahr und in einer bestimmten Stadt.

- *Pivotierung* bzw. Rotation ist das Vertauschen der Achsen des Datenwürfels, wobei je nach Anwendung unterschiedliche Dimensionen im Vordergrund stehen.

- *Roll-up* aggregiert Daten entlang eines Konsolidierungspfades um neue Informationen ausgeben zu können. Zum Beispiel könnte über die Klassifikationsstufen der Dimension Zeit navigiert werden, damit statt Informationen über einzelne Monate die Informationen über einzelne Quartale ausgegeben werden können.

- *Drill-down* ist die komplementäre Operation zu Roll-up. Das heißt statt Informationen über einzelne Quartale auszugeben, könnten Informationen über Monate ausgegeben werden.

- *Drill-across* bezeichnet das Wechseln zwischen den Würfeln, die durch die Operationen Roll-up und Drill-down entstanden sind.

2.2. Multidimensionale Anfragen

In der vorliegenden Diplomarbeit werden multidimensionale Anfragen an relational verwaltete Datenbestände eines Data Warehouses gestellt. Das heißt, es wird das relationale OLAP bzw. ROLAP angewendet. Diese Anfragen haben bestimmte Charakteristika, welche im Folgenden näher erläutert werden sollen. Im Anschluss daran wird auf die relationale Umsetzung multidimensionaler Anfragen eingegangen.

2.2.1. Charakteristika von Anfragen

„Anfragen an das Data Warehouse umfassen in der Regel Aggregationen von bestimmten Datenbereichen [BG04]“. Diesbezüglich werden von Bauer et al. in [BG04] zwei grundlegenden Eigenschaften von Data-Warehouse-Anfragen aufgezeigt. Diese Eigenschaften sind, dass

- Anfragen an eine sehr große Datenmenge gestellt werden und
- Anfragen Restriktionen enthalten.

Restriktionen sind Einschränkungen der Ergebnismenge von Anfragen, welche bezüglich der Dimensionen und Kennzahlen durchgeführt werden können. Die Restriktionen spielen neben den Aggregationen im multidimensionalen Modell die wichtigste und zeitaufwendigste Rolle [BG04]. Im Zusammenhang mit Dimensionen werden Restriktionen bezüglich bestimmter Klassifikationsknoten realisiert. Zum Beispiel kann eine Anfrage die Dimension Zeit erhalten, wobei nur Daten in die Ergebnismenge der Anfragen aufgenommen werden, wenn die Bedingung „Jahr = 2006“ erfüllt ist. Eine Restriktion für eine Kennzahl Verkauf ist unter anderem die Bedingung „Verkauf $>$ 10“.

Typen von Anfragen

Es existieren unterschiedliche Typen von Anfragen, je nachdem welche Restriktionen auf den Dimensionen durchgeführt werden. Diese Anfragen werden im Folgenden erläutert. Die Beispiele beziehen sich auf einen Datenwürfel, der durch die Dimensionen Zeit und Produkt aufgespannt wird. Die Dimension Zeit enthält die Klassifikationsstufen Tag und Jahr, während bei der Dimension Produkt die Stufen Artikel und Kategorie sind. Der Datenwürfel umfasst darüber hinaus die Kennzahl Verkauf.

- *Bereichsanfragen* sind Anfragen, die in jeder Dimension durch ein Intervall bestimmt sind. Ein Beispiel ist die Anfrage nach dem Verkauf aller Artikel an den unterschiedlichen Tagen. Es ist bei Bereichsanfragen allerdings auch möglich ein bestimmten Klassifikationsknoten bzw. eine Spanne von Klassifikationsknoten anzugeben. Bereichsanfragen sind der allgemeinste Typ von Anfragen und beinhalten alle folgenden Typen.
- *Partielle Bereichsanfragen* entsprechen Bereichsanfragen, allerdings werden nicht alle Dimensionen durch Intervalle eingeschränkt. Eine solche Anfrage ist beispielsweise der Verkauf einer bestimmten Menge von Artikeln in Beachtung der Dimension Zeit. Die Dimension Zeit ist in diesem Fall nicht eingeschränkt.

- *Partial-Match-Anfragen* schränken einen Teil der Dimensionen auf einen Punkt ein, während andere uneingeschränkt bleiben. Der Verkauf eines bestimmten Artikels in Beachtung der Dimension Zeit ist eine Partial-Match-Anfrage. Die Dimension Zeit ist nicht eingeschränkt, während die Dimension Produkt auf einen bestimmten Klassifikationsknoten limitiert ist.

- *Punktanfragen* schränken alle Dimensionen auf einen Klassifikationsknoten ein. Zum Beispiel der Verkauf eines bestimmten Artikels an einem angegebenen Tag.

2.2.2. Relationale Umsetzung multidimensionaler Anfragen

„Die konkrete Umsetzung einer multidimensional formulierten Anfrage in eine SQL-Anweisung hängt davon ab [BG04]“, welches Schema zur Speicherung des multidimensionalen Datenmodells verwendet wird. In der vorliegenden Diplomarbeit wird das Star-Schema eingesetzt, welches aus einer Faktentabelle und einer Dimensionstabelle für jede Dimension des Datenwürfels besteht. Diese Tabellen werden mit Hilfe von Schlüssel/Fremdschlüsselbeziehungen untereinander verknüpft. Anfragen an des Datenmodell beinhalten somit den Verbund zwischen den Dimensionstabellen und der Faktentabelle, wobei Restriktionen bezüglich der Dimensionen vorgenommen werden. Ein solcher Mehrfachverbund wird in Anlehnung an das zu Grunde liegende Sternschema als Star-Join bezeichnet [BG04].

Star-Join

Ein Star-Join ist ein (n+1)-Wege-Verbund zwischen einer zentralen Faktentabelle und n Dimensionstabelle. Mit Hilfe des Sternschemas in Abbildung 2.5 wird nachfolgend ein Star-Join am Beispiel dargestellt werden.

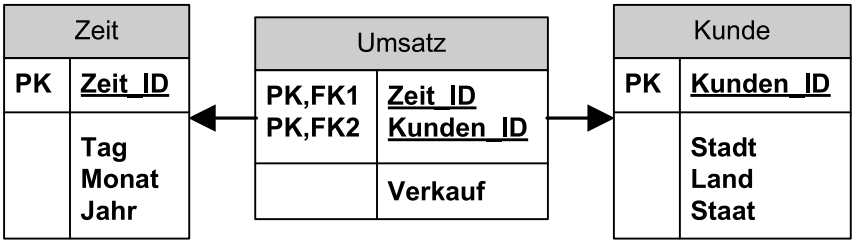

Abbildung 2.5.: Beispiel eines Sternschemas zur Erläuterung des Star-Joins

Das Sternschema enthält die Dimensionstabellen Zeit und Kunde und die Faktentabelle Umsatz, in der die Kennzahl Verkauf gespeichert ist. Die Dimensionstabelle Zeit enthält die Hierarchiestufen Tag, Monat und Jahr. Der Primärschlüssel der Relation Zeit ist Zeit_ID, der gleichzeitig als Fremdschlüssel in der Faktentabelle enthalten ist. Die Tabelle Kunde hingegen enthält die Stufen Stadt, Land und Staat. Kunden_ID ist der Primärschlüssel dieser Relation und ebenfalls in der Faktentabelle enthalten. Die

Fremdschlüssel Zeit_ID und Kunden_ID bilden zusammen den Primärschlüssel der Faktentabelle.

Eine mögliche multidimensionale Anfrage an das Sternschema aus Abbildung 2.5 ist zum Beispiel folgende Anfrage:

„Wie viel wurde im Jahr 2006 pro Monat verkauft?“

Diese multidimensional formulierte Anfrage kann wie folgt als SQL-Anweisung umgesetzt werden:

```
SELECT Zeit.Monat, SUM(Umsatz.Verkauf)
FROM Zeit, Umsatz
WHERE Zeit.Zeit_ID = Umsatz.Zeit_ID AND
      Zeit.Jahre = 2006
GROUP BY Zeit.Monat
```

Die SQL-Anweisung enthält verschiedene Klauseln, mit deren Hilfe die unterschiedlichen Bestandteile des Sternschemas adressiert werden. Die Abbildung 2.6 illustriert die Zuordnung der SQL-Klauseln zu den Sternschema Bestandteilen.

SQL-Klausel	Inkludierte Bestandteile
SELECT	Aggregierte Kennzahlen
	Ergebnisgranularität
FROM	Dimensionstabellen
	Faktentabelle
WHERE	Verbundbedingungen
	Restriktionen
GROUP BY	Ergebnisgranularität

Abbildung 2.6.: SQL-Klauseln mit zugeordneten Sternschema Bestandteilen

Die SELECT-Klausel enthält die aggregierte Kennzahl Verkauf der Faktentabelle Umsatz. Es wurde die Aggregatfunktion „SUM“ verwendet. Die Granularität des Ergebnisses wird ebenfalls angegeben, wobei die Granularität den Verdichtungsgrad der Daten bestimmt. Das heißt die Ergebnisgranularität gibt an, welche Hierarchiestufe ausgegeben werden soll. In dem Beispiel ist dies die Stufe Monat der Dimension Zeit. Die Dimensionstabellen und die Faktentabelle werden in der FROM-Klausel angegeben, wobei in der WHERE-Klausel die entsprechenden Verbundbedingungen definiert werden. Die Verbundbedingung ist im Beispiel „Zeit.Zeit_ID = Umsatz.Zeit_ID“. Des Weiteren wird durch die Bedingung „Zeit.Jahre = 2006“ eine Restriktion vorgenommen. Die Anfrage ist somit eine Bereichsanfrage, speziell eine Partial-Match-Anfrage. Weiter gehende Einschränkungen bezüglich der Kennzahlen sind ebenfalls denkbar,

sind im Beispiel aber nicht angegeben. Zum Beispiel könnte durch die Bedingung „Umsatz.Verkauf $>$ 100“ festgelegt werden, dass nur Monate mit Verkaufszahlen größer als 100 in das Ergebnis aufgenommen werden sollen. Die letzte SQL-Klausel ist GROUP BY, welche ebenfalls die Ergebnisgranularität angibt. Die Klausel bestimmt, welche Attribute gruppiert werden.

3. Related Work

3.1. Materialisierte Views

Eine materialisierte View ist in der vorliegenden Diplomarbeit ein relational gespeicherter Datenwürfel, der das Ergebnis einer multidimensionalen Anfrage an ein Data Warehouse ist. Dabei wird der Datenwürfel des Data Warehouses mit Hilfe des Sternschemas relational verwaltet und mittels SQL abgefragt (siehe Abschnitt 2.2.2).

Gupta et al. geben in [GM99] einen Überblick über die Gesamtthematik der materialisierten Views. Es werden dabei die „wichtigen Arbeiten [GM99]“ anderer Autoren gesammelt und vorgestellt, sodass [GM99] als Einstieg in die Thematik genutzt werden kann. Nachfolgend werden an den entsprechenden Stellen jeweils die Originalartikel referenziert werden, damit detaillierter erkennbar ist, welche Ansätze von welchen Autoren entwickelt bzw. beschrieben wurden.

3.1.1. Restrukturierung von Anfragen

Mit der Thematik der Views im Kontext des relationalen Modells hat sich Roussopoulos in [Rou98] auseinander gesetzt. Der Autor gibt einen Überblick über das Potential von Views, da seiner Meinung nach dieses Themengebiet noch in keiner Literatur den zentralen Fokus hatte. Auch der Begriff der materialisierten Views wird benutzt, allerdings nicht eindeutig definiert.

Mit den Möglichkeiten der Optimierung von Anfragen unter Verwendung von Views hat sich Halevy in [Hal01] bzw. in [Hal00] beschäftigt. Mit Hilfe von motivierenden Beispielen wird gezeigt, wie Anfragen umgeschrieben werden können, welche Probleme sich ergeben und welche Theorien der Problematik der Anfrageoptimierung zur Grunde liegen. Des Weiteren werden von Halevy in [Hal01] konkrete Algorithmen für die Beantwortung von Anfragen vorgestellt und Möglichkeiten erläutert, wie Anfrageoptimierer verändert werden können um auch Views zu berücksichtigen. Auf den genauen Inhalt von [Hal01] wird nachfolgend an entsprechender Stelle detaillierter eingegangen.

View Usability Problem

Materialisierte Views können zur Optimierung von Anfragen genutzt werden. Die dabei umgesetzte Idee ist: Anstatt Anfragen an die Relationen einer Datenbank zu stellen, werden die extrahierten Ergebnisse der materialisierten Views zur Beantwortung verwendet. Dabei wird vorausgesetzt, dass die ursprünglich adressierten Relationen einer Datenbank im Vergleich zu den Relationen der Views wesentlich umfangreicher bzgl. der Tupelanzahl sind. Des Weiteren wird angenommen, dass die Relationen der materialisierten Views alle zur Beantwortung notwendigen Daten enthalten. Der Vorteil

dieses Vorgehens liegt „in einer meist deutlichen Reduzierung der Anfrageausführungszeiten [BG04]“. Damit ergibt sich die Frage bzw. das Problem, welches in der Literatur als „View Usability Problem“ bekannt ist:

„Given views $V_1, ..., V_n$ and a query Q over a fixed database schema, can Q be reformulated using the views so that it does not use any (or as few as possible) of the relations in the database [JLVV01].“

Das heißt, kann eine Anfrage Q an ein festgeschriebenes Datenbankschema bei einer gegebenen Menge von Views $V_1, ..., V_n$ so unter Verwendung der Views umgeschrieben werden, dass keine bzw. so wenig wie möglich Relationen der Datenbank verwendet werden.

Komplexität des View Usability Problems

Das „View Usability Problem“ wird von Halevy in [Hal01] unterteilt in die Probleme des „Query Answering“ und des „Query Rewriting“. Dabei sind beide Probleme nicht trivial, sondern ohne entsprechende Einschränkungen von Sichtdefinitionen und möglichen Anfragen NP-vollständig [Hal01].

Bauer et al. sagen in [BG04], dass „für den allgemeinen Fall relationaler Anfragen das Problem der Existenz einer gültigen Ersetzung NP-vollständig [BG04]“ ist. Eine gültige Ersetzung wird wie folgt definiert:

„Eine Anfrage Q' ist eine gültige Ersetzung der Anfrage Q unter Verwendung der Materialisierung M, wenn Q und Q' das gleiche Multimengenergebnis liefern [BG04].“

„Eine Multimenge ist eine Menge, in der Elemente mehrfach auftreten können [BG04]“. Dabei ist eine gültige Ersetzung die Voraussetzung, dass eine restrukturierte Anfrage äquivalent zur ursprünglichen Anfrage ist [BG04].

Von Jarke et al. wird in [JLVV01] das Problem der äquivalenten Ersetzung als „Equivalence Problem“ bezeichnet, einem Spezialfall des „View Usability Problems“. Auch in dieser Literatur wird gesagt, dass das „View Usability Problems“, genau wie das „Equivalence Problem“ NP-vollständig sind.

Conjunctive Queries

Es werden aufgrund der vorliegenden Komplexität Einschränkungen bezüglich der Anfragen an eine Datenbank vorgenommen. Eine mögliche Einschränkung ist die Verwendung von „Conjunctive Queries“, auch „Select-Project-Join Queries“ genannt. Solche Anfragen haben in SQL den folgenden Aufbau:

```
SELECT <Attribute>
FROM <Relationen>
WHERE <Verbundbedingungen> AND
      <Restriktionen>
```

Eine weitere Einschränkung bei „Conjunctive Queries“ besteht darin, das die Verbundbedingungen und Restriktionen jeweils nur als Vergleichsoperator „=“ und als Verknüpfung untereinander „AND“ erlauben. Darüber hinaus können in „Conjunctive Queries“ keine Aggregationen und Gruppierungen vorgenommen werden, welche essentiell für Data Warehouses sind (siehe Abschnitt 2.2).

Algorithmen für Conjunctive Queries

Für „Conjunctive Queries“ existieren verschiedene Algorithmen, mit denen Anfragen an eine Datenbank umgeschrieben werden können, sodass diese materialisierte Views verwenden. Halevy stellt in [Hal01] den „Bucket Algorithm“ und „Inverse-Rules Algorithm“ vor, wobei der Begriff des „Subgoals“ eingeführt wird. Ein Subgoal ist eine Restriktion oder Verbundbedingung, welche in der WHERE-Klausel der „Conjunctive Query“ vorkommt. Das heißt jede aufgeführte Bedingung, jeweils getrennt durch „AND“, ist ein Subgoal.

Der „Bucket Algorithm“ versucht für jedes Subgoal isoliert eine entsprechende materialisierte View zu finden, aus diesen „Conjunctive Queries“ zu erstellen und am Ende die einzelnen Anfragen so zu kombinieren, dass alle Subgoals der ursprünglichen Anfrage erfüllt sind.

Der „Inverse-Rules Algorithm“ konstruiert hingegen eine Menge von Regeln, welche die Sichtdefinitionen beschreiben. Anhand der Regeln wird danach versucht, die erwarteten Tupel der Anfrage zu analysieren und durch die entsprechenden Views, welche die gleichen Tupel liefern, zu ersetzen.

Pottinger et al. stellen in [PL00] den „MiniCon Algorithm“ vor und vergleichen diesen experimentell mit dem „Bucket Algorithm“ und „Inverse-Rules Algorithm“. Der „MiniCon Algorithm“ ist eine Erweiterung des „Bucket Algorithm“, der strengere Bedingungen an die Auswahl von Views im Zusammenhang mit Subgoals stellt. Es werden mit Hilfe von detaillierteren Beschreibungen („MiniCon Descriptions“) mehr Informationen über die einzelnen verwendbaren Views gespeichert als beim „Bucket Algorithm“, was am Ende zur einer verbesserten Auswahl der verwendbaren Views führt. Zum Beispiel wird explizit gespeichert, welche Variablen wie ersetzt werden können, welche Subgoals eine View besitzt, etc.. Der letzte Schritt ist erneut die Restrukturierung der ursprünglichen „Conjunctive Query“ unter Verwendung der materialisierten Views. Die Autoren von [PL00] sagen, dass eine Erweiterung des „MiniCon Algorithm“ möglich ist. Das heißt, dass der Algorithmus nicht nur „Conjunctive Queries“ sondern auch komplexere Anfragen bearbeiten kann. Details dazu werden allerdings nicht genannt.

Ein kostenbasierter Ansatz für „Conjunctive Queries“ wird von Chaudhuri et al. in [CKPS95] vorgestellt. Die Autoren nutzen eine „MapTable“, die eine Liste von Ersetzungsregeln enthält. Jede Ersetzungsregel beschreibt dabei, wie welche Bestandteile einer Anfrage durch eine View ersetzt werden können. Die „MapTable“ wird danach mit einem von Chaudhuri et al. in [CKPS95] erläuterten Algorithmus durchsucht. Der Algorithmus liefert als Ergebnis die kostengünstigste Umformung der ursprünglichen Anfrage, einen so genannten Ersetzungsplan. Die Kosten werden dabei durch die Anzahl der Verbunde bestimmt, die eine Anfrage benötigt. Das heißt aber gleichzeitig, dass

eventuell auch keine Views zur Beantwortung einer Anfrage genutzt werden können. Ein weiterer Aspekt ist die Speicherung von guten Ersetzungsplänen. Daraus folgt beim Auftreten einer neuen Anfrage, für welche schon in der Vergangenheit ein Ersetzungsplan gefunden wurde, wird der alte Plan als Ergebnis des Algorithmus ausgegeben. Es ist allerdings unklar was passiert, wenn ein alter Plan nicht gut war aber dennoch gespeichert wurde.

Anfragen mit Aggregationen

„Im Data-Warehouse-Bereich besonders interessant sind Aggregatanfragen [BG04]“, das heißt Anfragen mit folgendem Aufbau:

```
SELECT <Gruppierungsattribute>,
       <Aggregatfunktionen>
FROM <Faktentabelle>,
     <Dimensionstabellen>
WHERE <Verbundbedingungen> AND
      <Restriktionen>
GROUP BY <Gruppierungsattribute>
```

Diese Anfragen werden als Star-Queries bzw. Sternanfragen bezeichnet. Star-Queries sind somit multidimensionale Anfragen an ein Sternschema, wobei Aggregatfunktionen über Kennzahlen verwendet werden.

Gupta et al. beschreiben in [GHQ95] ein Verfahren, mit dem Star-Queries so restrukturiert werden, dass diese Anfragen materialisierte Views verwenden. Das vorgestellte Verfahren arbeitet mit Anfragebäumen. Das heißt, dass Anfragen an das Data Warehouse und gespeicherte Views normalisiert als Graph dargestellt werden. Die Elemente eines Baumes sind dabei die unterschiedlichen Bestandteile der Star-Query. Für diese Elemente wurden Regeln definiert, das heißt es wird beschrieben, wie welche Bestandteile eines Baumes transformiert, verschoben, umgeschrieben etc. werden können. Unter Verwendung der Regeln sollen Teilbäume entstehen bzw. extrahiert werden, welche durch die Anfragebäume von materialisierten Views ersetzt werden können. Am Ende wird aus dem veränderten Anfragebaum eine restrukturierte Anfrage konstruiert, die nach Bauer et al. in [BG04] eine gültige Ersetzung ist. Der Algorithmus von Gupta et al. in [GHQ95] sieht vor, dass die Restriktionen der Anfragen genauso bzw. größer ein müssen, als die Restriktionen der materialisierten Views.

Die Autoren von [SD96] beschreiben einen Ansatz, der diese Einschränkungen von Gupta et al. in [GHQ95] nicht besitzt. Es wird erläutert, unter welchen Umständen und Voraussetzungen Anfragen mit Hilfe von Views umgeschrieben werden können. Für den Fall, dass eine Aggregatanfrage und als materialisierte Views nur „Conjunctive Queries“ vorliegen, wird die Idee der Multiblockersetzung eingeführt. Das heißt, dass die Selektionsbedingungen derart aufgeteilt werden, dass unterschiedliche Views für die verschiedenen Selektionsbedingungen verwendet werden können. Die dadurch entstehenden Teilanfragen, das heißt Anfragen die nicht das komplette Ergebnis der Aggregatanfrage repräsentieren, werden am Ende vereinigt und als restrukturierte Anfrage ausgegeben. Srivastava et al. betrachten in [SD96] außerdem noch die Fälle, dass

sowohl Anfragen als auch gespeicherte Views Aggregatanfragen sind, und dass Anfragen „Conjunctive Queries“ sind, während die Views Ergebnisse von Aggregatanfragen repräsentieren. Für den letzten Fall wird festgestellt, dass eine Restrukturierung einer Anfrage nicht möglich ist. Die Autoren von [SD96] setzen neben den Anforderungen an die Struktur der Views und Anfragen voraus, dass die Spalten der SELECT-Klausel global eindeutig sind. Des Weiteren wird gefordert, dass eine Anfrage bzw. bei der Multiblockersetzung jede der Teilanfragen erfüllbar sein muss. Die Behandlung nicht erfüllbarer Anfragen ist nach Srivastava et al. in [SD96] nicht für das Verständnis notwendig, wird demnach auch nicht thematisiert.

Komplexe Anfragen

Zaharioudakis et al. stellen in [ZCL+00] einen weiterer Ansatz zur Restrukturierung von Anfragen vor. Die Autoren führen das „Query Graph Model“ ein, welches die Basis für deren Arbeit ist. Das „Query Graph Model“ ist die grafische Repräsentation einer Anfrage, welche Blätter, Knoten und ein Wurzelelement besitzt. Die Blätter sind die Basisrelationen einer Anfrage, während die Knoten Operationen auf diesen Relationen repräsentieren. Jeder Knoten ist dabei eine Box, die explizit Ein- und Ausgaben angibt. Die Wurzel enthält als Ausgabe die restrukturierte Anfrage. Mit Hilfe des „Query Graph Models“ und einem „Matching Framework“ werden die Beziehungen zwischen den Anfragen und den materialisierten Views untersucht. Das heißt es wird analysiert, wo und wie welche Knoten ersetzt werden können und welche Kompensationen vorgenommen werden müssen. Eine Kompensation ist eine Menge von Operationen, die einen Knoten derart abändern, sodass er ersetzt werden kann. Eine Besonderheit ist, dass nicht nur Unteranfragen, sondern auch Anfragen mit benutzerdefinierte Funktionen und multidimensionale Aggregationen verarbeitet werden können.

Dieser Ansatz wurde laut der Autoren von [ZCL+00] implementiert und getestet, wobei allerdings Details nicht veranschaulicht werden. Es wird darüber hinaus nicht beschrieben, wie der „Matching Function“ des „Matching Frameworks“ genau abläuft. Diese Funktion wird von einem „Navigator“ betrieben, der das „Query Graph Model“ durchsucht und anhand von nicht exakt definierten Mustern äquivalente Knoten herausfindet. Des Weiteren erwähnen Zaharioudakis et al. in [ZCL+00], dass deren Algorithmus in der Lage sei zu entscheiden, wann ein Attribut einer View äquivalent zu einem einer Anfrage sei. Details werden diesbezüglich nicht dargestellt, genau wie nicht erläutert wird, wie generell Kompensationen erkannt und definiert werden können.

3.1.2. OLAP Optimierung mit materialisierten Views

Die Autoren von [NT04] beschreiben einen metadaten-gestützten Ansatz zur Optimierung von OLAP Anfragen unter Verwendung materialisierter Views. Es wird dabei eine Architektur zur Behandlung des „View Usability Problem“ vorgestellt. Diese Architektur enthält unterschiedliche Komponenten, die folgende Aufgaben haben:

- Abschätzung der Größe von Views
- Auswahl der Views
- Wartung der Views
- Optimierung von Anfragen unter Verwendung von Views

Die unterschiedlichen Komponenten und deren Aufgaben werden von Nadeau et al. in [NT04] näher thematisiert, wobei insbesondere auf die Restrukturierung von Anfragen unter Verwendung materialisierter Views eingegangen wird.

Schemainformationen und Relationen

Nadeau et al. verwenden in [NT04] ein Sternschema, an welches Star-Queries gestellt werden. Das heißt, es existiert eine Faktentabelle und mehrere Dimensionstabellen. Des Weiteren wird eine Relation Metadata eingeführt, welche einen Eintrag für jede materialisierte View enthält und dieser eine eindeutige ViewID zuordnet. Jede materialisierte View wird darüber hinaus durch eine Relation mit dem Namen AST<ViewID> repräsentiert, wobei Informationen zur Größe der View in der Relation Metadata gespeichert werden. Die Größe wird in diesem Fall durch die Anzahl der Blöcke angegeben, die eine View zur Speicherung von deren Tupel benötigt. Zusätzlich zur ViewID und der Größe einer View enthält die Relation Metadata jeweils eine Spalte für jede Dimensionstabelle. Es ergibt sich folgendes Relationenschema:

$$metadata\ (viewid,\ blocks,\ d_1, ..., d_n)$$

Die Relation Metadata enthält somit sowohl das Attribut ViewID, als auch das die Größe angebende Attribut „blocks“, als auch n unterschiedliche Dimensionen.

Klassifikationshierarchien

Die Einträge der Dimensionen sind Integerzahlen, welche die in der View gespeicherten Klassifikationsstufen darstellen [NT04]. Das heißt, dass den unterschiedlichen Stufen der Klassifikationshierarchie Werte zugeordnet werden. Für eine Dimension Zeit mit den Stufen Tag, Monat und „Top“ und eine Dimension Kunde mit den Stufen Stadt, Land und „Top“ könnte folgende Zuordnung vorgenommen werden:

$$\textbf{\textit{Zeit}}\textit{: Tag} \mathrel{\widehat{=}} \textit{0}\ ,\ \textit{Monat} \mathrel{\widehat{=}} \textit{1}\ ,\ \textit{Top} \mathrel{\widehat{=}} \textit{2}\ ;\ \textbf{\textit{Kunde}}\textit{: Stadt} \mathrel{\widehat{=}} \textit{0}\ ,\ \textit{Land} \mathrel{\widehat{=}} \textit{1}\ ,\ \textit{Top} \mathrel{\widehat{=}} \textit{2}$$

Es können mit dieser Zuordnung einfache Hierarchien dargestellt werden, wobei eine Ordnung eingehalten werden muss. Das heißt je kleiner die Integerzahl der Dimension, je kleiner ist die Hierarchiestufe.

Algorithmus zur Optimierung

Die Autoren von [NT04] nutzen die vorhandenen zusätzlichen Informationen über die materialisierten Views für die Optimierung der OLAP Anfragen. Sie verwenden dabei die Abhängigkeiten zwischen den einzelnen Klassifikationsstufen einer Dimension. Die zugrunde liegende Idee ist:

> *Falls eine View V zur Optimierung gebraucht wird, welche allerdings nicht materialisiert ist, dann versuche eine andere View V' zu nutzen, bei der die Klassifikationsstufen kleiner oder gleich der Klassifikationsstufe von V ist.*

Diese Idee wird anhand eines Beispiels von Nadeau et al. in [NT04] illustriert. Es wird bei der Vorgehensweise ausgenutzt, dass eine Hierarchiestufe bei einfachen Hierarchien die Aggregation aller Dimensionselemente der jeweils niedrigeren Hierarchiestufe ist.

Im Folgenden wird vereinfacht das Vorgehen der Autoren von [NT04] beschrieben. Die folgende Star-Query Q wird gestellt:

```
SELECT Zeit.Monat, Kunde.Land,
       SUM(Faktentabelle.Kennzahl)
FROM Zeit, Kunde, Faktentabelle
WHERE <Verbundbedingungen>
GROUP BY Zeit.Monat, Kunde.Land
```

Des Weiteren existiert eine materialisierte View V mit „ViewID = 1“, welche aus der folgenden Anfrage erstellt wurde:

```
SELECT Zeit.Tag, Kunde.Stadt,
       SUM(Faktentabelle.Kennzahl)
FROM Zeit, Kunde, Faktentabelle
WHERE <Verbundbedingungen>
GROUP BY Zeit.Tag, Kunde.Stadt
```

Es ist ersichtlich, dass mit den bisher vorgestellten Methoden die View V nicht zur Restrukturierung der Anfrage Q verwendet werden kann. Allerdings unter Ausnutzung der zusätzlichen Informationen über die Klassifikationshierarchien kann die View V dennoch eingesetzt werden, was nachfolgend erläutert wird.

Nadeau et al. beginnen in [NT04] damit, dass sie die Klassifikationsstufen der Anfrage analysieren. In dem Beispiel wäre das unter Beachtung obiger Zuordnung die Kombination (1,1). Wie allerdings entschieden wird, welche Stelle welcher Dimension entspricht, wird nicht erwähnt. In dem Beispiel entspricht die erste Zahl der Dimension Zeit und die zweite der Dimension Kunde. Nach Feststellung der Kombination wird die Relation Metadata wie folgt abgefragt:

```
SELECT ViewID
FROM Metadata
WHERE Zeit <= 1 AND Kunde <= 1
ORDER BY Blocks
```

Als Ergebnis würde zum Beispiel die View V mit „ViewID = 1“ zurück geliefert werden, da diese View die Kombination (0,0) besitzt. Gesetzt den Fall es existieren weitere Views, dann würde durch „ORDER BY Blocks“ die View mit dem geringsten Speicherverbrauch gewählt werden. Mit der Voraussetzung, dass eine materialisierte View durch die Relation AST<ViewID> repräsentiert wird, würde die Anfrage Q wie folgt restrukturiert werden:

```
SELECT Zeit.Monat, Kunde.Land,
       SUM(AST1.Kennzahl)
FROM Zeit, Kunde, AST1
WHERE <Verbundbedingungen>
GROUP BY Zeit.Monat, Kunde.Land
```

Es kann somit auf den kostenintensiven Zugriff auf die Faktentabelle verzichtet werden, die aus vielen Millionen Einträgen bestehen kann [NT04].

Weitere Herausforderungen

Der Ansatz von Nadeau et al. in [NT04] ist eine Möglichkeit zur Optimierung von OLAP Anfragen unter Ausnutzung von semantischen Informationen. Allerdings werden einige Aspekte nicht behandelt. Zum Beispiel wird nicht gesagt, was passiert, wenn die Aggregatfunktionen oder Kennzahlen der Views nicht denen der Anfragen entsprechen, oder woher ein System die genauen Zuordnungen der Hierarchiestufen zu den Werten wissen soll. Des Weiteren wird nicht erwähnt, wie die Kombinationen der Klassifikationsstufen entstehen bzw. verwaltet werden. Das Problem ist, dass beim Auftreten von n Dimensionen immerhin n! verschiedene Kombinationen existieren. Darüber hinaus schränken die Autoren die Views diesbezüglich ein, dass jede View generell alle möglichen Dimensionen enthalten muss. Views, die nicht alle Dimensionen selektiert haben, passen nicht in das Konzept und werden daher nicht thematisiert.

3.2. Auswahl materialisierter Views

In Abschnitt 3.1 wurden verschiedene Verfahren und Lösungsmöglichkeiten vorgestellt, wie Anfragen mit Hilfe materialisierter Views optimiert werden können. Es wurde allerdings nicht erwähnt, welche materialisierte Views gespeichert werden sollen. Mit dieser Problematik befassen sich unter anderem die Autoren von [HRU96].

Die Autoren von [HRU96] erläutern einen kostenbasierten Ansatz, der in Beachtung des vorhandenen Speichers eine Menge zu materialisierender Views ausgibt.

Lattice Framework

Das „Lattice Framework“ bzw. Gitternetzwerk ist ein Graph, dessen Knoten mögliche Anfragen an einen Datenwürfel repräsentieren. Diese Anfragen sind Kandidaten, die als materialisierte Views gespeichert werden können. Zwischen den Knoten werden beim

Vorhandensein von Abhängigkeiten Kanten hinzugefügt. Im Zusammenhang mit Klassifikationshierarchien könnten zum Beispiel die Knoten miteinander verbunden werden, welche unterschiedliche Hierarchiestufen einer Hierarchie repräsentieren. Es existiert im Graph jeweils ein höchster und niedrigster Knoten. Während der höchste Knoten einer Anfrage entspricht, die alle Werte des Datenwürfels ausgibt, entspricht der niedrigste Knoten einer Anfrage ohne Werte.

Kostenmodell

Den Knoten des Gitternetzwerkes werden Kosten zugeordnet, damit die einzelnen zu materialisierenden Views untereinander verglichen werden können. Diese Kosten entsprechen bei Harinarayan et al. in [HRU96] dem zusätzlichen Speicher, der durch die materialisierte View belegt werden würde. Im Kontext von [HRU96] schätzen Harinarayan et al. die Kosten anhand der Anzahl der Tupel einer View ab. Es wird dabei ein lineares Kostenmodell verwendet, das heißt die Zeit zur Beantwortung einer Anfrage ist proportional zur Größe der verwendeten View.

Greedy Algorithm

Unter Verwendung des „Lattice Frameworks“ und des Kostenmodells wird von Harinarayan et al. in [HRU96] der „Greedy Algorithm“ vorgestellt. Dieser Algorithmus wählt anhand des Gitternetzwerks eine vorher definierte Anzahl von Views aus, je nachdem wie hoch der Nutzen einer View unter Beachtung schon vorhandener Views ist. Der Nutzen einer View ist dabei die Differenz der bisherigen Gesamtkosten aller materialisiertem Views gegen die Gesamtkosten mit und ohne neu zu materialisierender View.

Im ersten Durchlauf vom Algorithmus wird der Knoten in die Menge der zu materialisierenden Views aufgenommen, der die meisten Detaildaten liefert [BG04]. Dieser Knoten ist der höchste Knoten im Gitternetzwerk und ermöglicht, „die Auswertbarkeit aller möglichen Anfragen sicherzustellen [BG04]“. In jedem weiteren Durchlauf wird daraufhin eine weitere View in die Lösungsmenge bzw. Viewmenge aufgenommen, bis die vorher definierte Anzahl der Views erreicht ist.

Modifikationen

Eine mögliche Modifikation vom „Greedy Algorithm“ ist der Verzicht auf die festgeschriebene Anzahl der Views. Der resultierende Algorithmus wird BPUS genannt [SDN98], wobei BPUS ein Akronym für „Benefit Per Unit Space“ (Nutzen je Element Größe) ist. BPUS verwendet anstatt einer festgeschriebenen Anzahl von Views eine maximale Speichergröße, die der Viewmenge zur Verfügung steht. Des Weiteren wird von den Autoren von [HRU96] vorgeschlagen, dass das Anfrageverhalten stärker beachtet werden soll. Der „Greedy Algorithm“ geht nämlich davon aus, dass alle Knoten des Gitternetzwerks die gleiche Zugriffswahrscheinlichkeit haben.

PBS und PBS-U

Die Zugriffwahrscheinlichkeit von Knoten des Gitternetzwerks wird im Algorithmus PBS-U der Autoren von [SDN98] berücksichtigt. PBS ist ein Akronym für „Pick By Size“ (Auswahl durch Größe). PBS-U ist eine Erweiterung von PBS, bei der das Zugriffsverhalten von einzelnen Knoten beachtet werden kann. Dabei steht „-U“ für „Non-Uniform“ (Nicht einheitlich).

Der Algorithmus PBS hat als Grundlage ebenfalls obiges Gitternetzwerk und Kostenmodell. Im Gegensatz zum „Greedy Algorithm“ und BPUS errechnet PBS allerdings nicht den Nutzen einer View im Zusammenhang der bereits vorhandenen Views, sondern nimmt generell die kleinste View in jedem Durchlauf in die Viewmenge auf. Dieses Vorgehen ermöglicht PBS wesentlich schneller eine Menge zu materialisierender Views zu berechnen, garantiert allerdings nicht die Qualität der materialisierten Views.

Menge ausgewählter Views

Shukla et al. stellen in [SDN98] anhand durchgeführter Experimente dar, wie die durchschnittlichen Antwortzeiten von Anfragen bei Verwendung unterschiedlicher Mengen von Views waren. Dabei wurde unter anderem festgestellt, dass die Antwortzeit bei Nutzung PBS-ausgewählter Viewmengen deutlich kürzer war als im Vergleich zu BPUS. Unter bestimmten Umständen wurden zwar die gleichen Viewmengen durch die unterschiedlichen Algorithmen geliefert, aber die Zeit war in dem Fall entscheidend, die zum Finden der Menge der Views erforderlich war. Ein Beispiel wird erwähnt in dem PBS 0.37 Sekunden benötigte, während BPUS erst nach 64 Tagen eine Viewmenge lieferte. Wie diese extremen Abweichungen zustande gekommen sind, wird nicht erklärt.

3.3. Wartung materialisierter Views

Nach der Beschreibung der Möglichkeiten von materialisierten Views und deren Auswahl „darf nicht vergessen werden, dass diese Sichten auch aktualisiert werden müssen [BG04]“. Das Problem besteht darin, dass falls die Relationen des zugrunde liegenden Sternschemas bzw. der Basisrelationen aktualisiert werden, müssen diese Änderungen an die entsprechenden materialisierten Views propagiert werden. Unter Aktualisierung des Sternschemas wird in diesem Zusammenhang das Einfügen, Löschen oder Verändern von Datenwerten der Basisrelationen verstanden.

Die Autoren von [GM95] beschäftigen sich mit der Thematik der Wartung von materialisierten Views bzw. dem „View Maintenance Problem“. Sie definieren das „View Maintenance Problem“, stellen Algorithmen anderer Autoren vor und gehen auf mögliche Anwendungen von Algorithmen und materialisierte Views ein.

Dimensionen der Wartung

Gupta et al. teilen in [GM95] das „View Maintenance Problem“ in unterschiedliche Dimensionen ein, die zur Charakterisierung von Updatealgorithmen verwendet werden können. Dazu gehören, die Dimensionen der

- Information,
- Änderung,
- Sprache und
- Instanz.

Die *Informationsdimension* gibt an, wie viele und welche Daten bei einem Update zur Verfügung stehen. Das heißt, dass zum Beispiel geklärt werden muss, ob Updatealgorithmen auf die Basisrelationen und Sichtdefinitionen zugreifen können oder nicht. Dabei gilt, dass „die Menge der zur Verfügung stehenden Informationen entscheidenden Einfluss darauf hat, ob ein bestimmter Algorithmus angewendet werden kann [BG04]“. Die *Änderungsdimension* beschäftigt sich mit den erlaubten Modifikationen der Basisrelationen, die ein Algorithmus vornehmen kann. Dazu zählt zum Beispiel, ob ein Update direkt vollzogen wird oder ob ein Update eine Folge von Löschen und Einfügen entsprechender Tupel ist. Unter *Sprachdimension* versteht man die Angabe der Mächtigkeit der zugrunde liegenden Sprache. Das heißt, sind Sichten „Conjunctive Queries“, Aggregatanfragen, rekursive Anfragen oder eventuell noch komplexerer Natur. Als letzte Dimension wurde die *Instanzdimension* genannt. Diese Dimension gibt an, ob ein Algorithmus mit allen oder nur mit einigen Instanzen von Datenbanken und Modifikationen umgehen kann. Das heißt in Berücksichtigung der anderen Dimensionen wird geklärt, ob ein Algorithmus sämtliche Möglichkeiten der einzelnen Dimensionen abdeckt oder nur bestimmte Teile. Zum Beispiel könnten alle Informationen zur Verfügung stehen und Updates direkt ausgeführt werden, während allerdings nur Aggregatanfragen behandelt werden können.

Zeitpunkt der Wartung

Bauer et al. erläutern in [BG04] ein weiteres Unterscheidungsmerkmal zwischen den Updatealgorithmen. Dieses betrifft den Zeitpunkt, wann Änderungen propagiert werden sollen. Die folgenden Möglichkeiten werden genannt:

- Sofortige Aktualisierung
- Verzögerte Aktualisierung
- Snapshot Aktualisierung

Eine *sofortige Aktualisierung* bedeutet, dass eine Modifikationtransaktion synchron zum Update der Basisrelationen alle davon abgeleiteten Sichten aktualisiert. Die *verzögerte Aktualisierung* verzichtet auf eine direkte Aktualisierung der Sichten. Das heißt,

dass eine materialisierte Sicht erst beim Aufrufen der entsprechenden zugehörigen Relation aktualisiert wird. Eine *Snapshot Aktualisierung* hingegen aktualisiert Sichten nach „anwendungsspezifischen Gesichtspunkten [BG04]“. Zum Beispiel könnte festgelegt werden, dass Aktualisierungen von Sichten alle 24 Stunden erfolgen sollen. Damit wird allerdings toleriert, dass materialisierte Sichten veraltete Daten enthalten.

Mehrversionenansatz

Quass et al. stellen in [QW97] den Algorithmus 2VNL vor, wobei dies ein Akronym für „Two-Version No Locking“ (2 Versionen ohne Sperren) ist. 2VNL verwaltet zwei Versionen des gleichen Datensatzes und ermöglicht dem Anwender eines Data Warehouses den durchgehenden Zugriff auf einen konsistenten Zustand der gespeicherten Daten. Dieser Zugriff erfolgt mittels Aggregatanfragen, welche mit Hilfe von SQL formuliert werden. Des Weiteren erfolgen die Zugriffe eines Anwenders in Sessions. Eine Session ist die Dauer einer Sitzung, während ein Anwender auf die Daten des Data Warehouses zugreift. Ausgehend davon, dass sämtliche Informationen über die Relationen eines Data Warehouses vorhanden sind, werden die entsprechenden Schemata verändert bzw. ergänzt. Es werden nicht nur für jedes Datum eine Versionsnummer eingeführt, sondern auch Informationen zu den durchgeführten Operationen und den vorherigem Werten gespeichert. Es ergibt sich folgendes Schema:

$$schema\ (VN,\ op,\ A_1, ..., A_n,\ M_1, ..., M_n,\ pre\_M_1, ...,\ pre\_M_n)$$

VN gibt die aktuelle Versionsnummer des Tupels an, während *op* die entsprechende Operation ist, welche als letztes auf diesem Tupel durchgeführt wurde. A_x bezeichnet ein Selektionsattribut, welches nicht aktualisiert wird, während M_x das Ergebnis einer Aggregation ist. M_x können im Gegensatz zu A_x aktualisiert werden, wobei pre_M_x die Werte der letzten Version von M_x enthalten. Es ist des Weiteren auch möglich, dass die Selektionsattribute ebenfalls aktualisiert werden können. Dieser Fall wird allerdings von Quass et al. in [QW97] nicht thematisiert.

2VNL ermöglicht eine sofortige Aktualisierung von Tupeln. Dafür wird die Versionsnummer des Tupels auf die aktuelle, global eindeutige Versionsnummer des Data Warehouses gesetzt und die entsprechende Updateoperation im Operationsattribut eingetragen. Gleichzeitig wird der Wert von M_x unter pre_M_x und der aktualisierte Wert unter M_x gespeichert. Obwohl der Algorithmus ohne Sperren auskommen soll, müssen dennoch bei einer Aktualisierung die entsprechenden Tupel kurzfristig gesperrt werden. Ansonsten könnten trotz der Konsistenzgarantie fehlerhafte Daten abgefragt werden.

Mit Hilfe der Versionsnummern wird nun der Zugriff auf die Tupel überwacht. Dafür wird vor jedem lesenden Zugriff überprüft, ob die aktuelle Session noch gültig oder schon abgelaufen ist. Quass et al. beschreiben in [QW97] die unterschiedlichen Möglichkeiten für die Operationen Update, Einfügen und Löschen. Das heißt, es wird angegeben, unter welchen Umständen und Voraussetzungen ein Tupel gelesen werden kann bzw. wann ein konsistenter Zugriff nicht mehr möglich ist. Dies wird dabei in Abhängigkeit der letzten Operation auf dem Tupel und den Versionsnummern der Session, des Tupels selber und des Systems realisiert.

Eine Erweiterung von 2VNL ist der Algorithmus nVNL, bei dem n Versionen eines Tupels abgespeichert werden. Dies soll verhindern, dass lang laufende Sessions ablaufen und somit vorzeitig abgebrochen werden. Eine dynamische Anpassung der Anzahl der Versionen ist allerdings nicht möglich. Das heißt, dass schon beim Entwurf eines Datenschemas festgelegt werden muss, wie viele Versionen verwaltet werden sollen. Des Weiteren bedeutet eine Erhöhung der Versionsanzahl auf n Versionen im schlechtesten Fall, das heißt falls alle Attribute aktualisierbar sind, immer eine n-fache Erhöhung des Speicherbedarfs eines Tupels.

Summary-Delta Table

Mumick et al. stellen in [MQM97] den Ansatz der „Summary-Delta Table“ vor. Eine „Summary-Delta Table“ ist eine Relation, welche alle Updates einer materialisierten Sicht bzw. einer „Summary Table“ enthält. Das Verfahren unterstützt die Operationen Update, Einfügen und Löschen und benötigt die Informationen über die Basisrelationen und Sichtdefinitionen. Die Basisrelationen sind als Sternschema organisiert und werden mit Hilfe von SQL formulierten Aggregatanfragen abgefragt. Der vorgestellte Algorithmus wird in die folgenden zwei Phasen eingeteilt:

- propagate (propagieren)
- refresh (aktualisieren)

In der ersten Phase wird die „Summary-Delta Table“ aufgebaut. Das heißt, anstatt Aktualisierungen direkt auf den entsprechenden Tupeln von Views vorzunehmen, werden diese Aktualisierungen gesammelt. In der zweiten Phase werden dann mit Hilfe der „Summary-Delta Table“ die materialisierten Views aktualisiert. Dabei werden die entsprechenden Views gesperrt, dass heißt die materialisierten Views können in dieser Phase nicht gelesen werden. Die Autoren gehen von einer Snapshot Aktualisierung aus, wobei das Data Warehouse eine fest definierte Zeit zur Aktualisierung haben soll. In dieser Zeitspanne wird die Aktualisierung der materialisierten Views vorgenommen, während außerhalb der Spanne Updates gesammelt werden. Ein Anwender arbeitet demnach auf potentiell veralteten Daten.

Neben der Vorstellung des Ansatzes der „Summary-Delta Table“ wird von Mumick et al. in [MQM97] der Begriff der „Self-Maintainable Aggregates“ (selbstpflegende Aggregate) definiert. Ein „Self-Maintainable Aggregate“ ist eine Aggregatfunktion, die nur unter Verwendung des alten Wertes und der vorhandenen Updates aktualisiert werden kann. Das heißt, es muss nicht erneut die entsprechende Faktentabelle für eine Neuberechnung des Wertes abgefragt werden. Die Funktion „COUNT“ ist ein „Self-Maintainable Aggregate“, denn bei einem Update wird der alte Wert beibehalten, beim Einfügen erhöht und beim Löschen verringert.

Das vorgestellte Verfahren orientiert sich an der Aktualisierung der Faktentabelle. Das heißt, es wird nicht beachtet, ob die Dimensionstabellen des Sternschemas aktualisiert werden oder nicht. Allerdings ist laut der Autoren von [MQM97] eine Erweiterung des Ansatzes denkbar und sinnvoll.

3.4. Fazit

In den vorherigen Abschnitten wurde erläutert, wie materialisierte Views zur Optimierung genutzt werden können und wie entschieden werden kann, welche Anfragen materialisiert werden sollen. Des Weiteren wurden Verfahren präsentiert, mit denen vorhandene Views aktualisiert werden können.

Einschränkungen der Anfragen

Es ist durch die Autoren von [Hal01], [BG04] und [JLVV01] gezeigt worden, dass aufgrund der Komplexität des „View Usability Problems“ Einschränkungen bezüglich möglicher Anfragen an ein Data Warehouse gemacht werden müssen. Es wurden zum Beispiel „Conjunctive Queries“ verwendet, welche allerdings keine typischen Anfragen darstellen. Dennoch haben die unterschiedlichen Algorithmen für „Conjunctive Queries“ gezeigt, dass eine Erhöhung der Menge gespeicherter Informationen wesentlich zur Leistungsfähigkeit eines Algorithmus beitragen kann. Durch den Vergleich des „Bucket Algorithm“ mit dem „MiniCon Algorithm“ wurde ebenfalls illustriert, dass die Leistungsfähigkeit von der Menge der verfügbaren Informationen abhängt [PL00].

Star-Queries bzw. Aggregatanfragen sind typische Anfragen an ein Data Warehouse [BG04]. Mit dieser Thematik haben sich die Autoren von [GHQ95], [SD96] und [ZCL+00] beschäftigt. Es wurden dabei Graphen-basierte Ansätze präsentiert, mit denen definiert wurde, unter welchen Umständen und Voraussetzungen eine Anfrage restrukturiert werden kann. Dabei stellte sich des Weiteren heraus, dass ein modularer Aufbau bzw. die Unterteilung der Problematik der materialisierten Views in Teilprobleme sinnvoll ist [NT04]. Durch dieses Vorgehen ist der Austausch von Komponenten zu einem späteren Zeitpunkt möglich, zum Beispiel wenn andere Autoren neue Erkenntnisse publizieren bzw. wenn Teilprobleme durch andere, effektivere Verfahren gelöst werden können.

Kostenbasierter Ansatz

Nadeau et al. stellen in [NT04] einen Ansatz vor, bei dem Views nach deren Kosten ausgewählt und untereinander verglichen werden. Der Ansatz beschreibt eine Möglichkeit zur Verwaltung und Nutzung funktionaler Abhängigkeiten zwischen einzelnen materialisierten Views. Dies führt unter anderem dazu, dass weniger Views benötigt werden und die Charakteristika von Hierarchien in Dimensionen ausgenutzt werden können. Ein weiterer kostenbasierten Ansatz wird von Chaudhuri et al. in [CKPS95] vorgestellt, bei dem eine „MapTable“ Ersetzungsregeln für Bestandteile einer „Conjunctive Queries“ enthält. Ein kostenbasierter Ansatz ermöglicht unter anderem aber auch, dass auf die Restrukturierung von Anfragen verzichtet werden kann, falls der Nutzen dieses Prozesses zu gering ist.

Es wurden in Abschnitt 3.2 Algorithmen vorgestellt, mit denen eine zu materialisierende Menge von Views gefunden werden kann. Es wurden diesbezüglich unter anderem der „Greedy Algorithm“ von Harinarayan et al. in [HRU96] und Erweiterungen von Shukla et al. in [SDN98] vorgestellt. Diese Algorithmen können zum Finden

materialisierter Views verwendet werden. In der vorliegenden Diplomarbeit wird allerdings vorausgesetzt, dass eine Menge von materialisierter Views vorhanden ist und somit genutzt werden kann. Die Idee eines Kostenmodells und der Vergleichbarkeit von Views kann aber dennoch verwendet werden.

Wartung von Views

Nachdem Views vorhanden sind und genutzt werden können, müssen diese in Beachtung der zugrunde liegenden Basisrelationen aktualisiert werden. In Abschnitt 3.3 wurden die Arbeiten der Autoren von [GM95], [QW97] und [MQM97] vorgestellt. Während Gupta et al. in [GM95] einen Überblick über die Thematik präsentierten, wurden von Quass et al. in [QW97] und Mumick et al. in [MQM97] Algorithmen erläutert. Erneut ist der Konsens, dass „die Menge der zur Verfügung stehenden Informationen entscheidenden Einfluss darauf hat, ob ein bestimmter Algorithmus angewendet werden kann [BG04]“. Des Weiteren wurde die Idee der „Snapshot Aktualisierung“ eingeführt. Das heißt, es ist durchaus denkbar, dass Updates nicht direkt ausgeführt werden müssen, sondern gesammelt und zu „Ruhezeiten“ eines Data Warehouses durchgeführt werden.

4. Konzept

In den vorherigen Kapiteln wurden die technischen Grundlagen und Arbeiten anderer Autoren vorgestellt. Daraus abgeleitet und unter Verwendung der eingeführten Terminologie wird nachfolgend ein Konzept entwickelt, mit dem Aggregatanfragen an ein Star-Schema mit Hilfe materialisierter Views optimiert werden können.

4.1. Aggregatanfragen

Anfragen mit Aggregationen entsprechen den typischen Anfragen an ein Data Warehouse. Der allgemeine Aufbau wurde im Abschnitt 3.1.1 erklärt und sieht in SQL-Syntax wie folgt aus:

```
SELECT <Gruppierungsattribute>,
       <Aggregatfunktionen>
FROM <Faktentabelle>,
     <Dimensionstabelle>
WHERE <Verbundbedingungen> AND
      <Restriktionen>
GROUP BY <Gruppierungsattribute>
```

Des Weiteren wurde definiert, dass eine materialisierte View das relational gespeicherte Ergebnis einer Aggregatanfrage an ein Sternschema ist. Das heißt, dass die verschiedenen Bestandteile der Aggregatanfragen von vorhandenen Views und gestellten Anfragen verglichen werden müssen. Dadurch sollen die materialisierten Views ermittelt werden, die zur Restrukturierung einer Anfrage geeignet sind. Eine restrukturierte Anfrage ist dabei eine Star-Query, die unter Verwendung materialisierter Views derart umgeschrieben wurde, dass keine bzw. so wenig wie möglich Basisrelationen des Data Warehouses verwendet werden. Folgende Fragestellungen müssen diesbezüglich beantwortet werden: „Welche

- Dimensionen,
- Klassifikationsstufen,
- Kennzahlen,
- Aggregatfunktionen,
- Gruppierungsattribute und
- Restriktionen

wurden in der Aggregatanfrage einer materialisierten View verwendet und somit gespeichert?“ Materialisierte Views, für die die unterschiedlichen Fragestellungen positiv beantwortet wurden, werden in eine Menge von Views aufgenommen, die zur Restrukturierung einer Anfrage geeignet sind. Diese Menge soll nachfolgend ***PosMV*** genannt werden, ein Akronym für „Possible Materialized Views“ (Mögliche materialisierte Sichten).

Ein Überblick über sämtliche Mengen und eine Übersicht der Charakteristika der Mengen, welche nachfolgend verwendet werden, ist in Anhang A.1 dargestellt. Des Weiteren wird ein detailliertes Beispiel in Abschnitt 5.2 vorgestellt.

4.1.1. Dimensionen

Aggregatanfragen werden an ein Star-Schema gestellt. Das bedeutet, dass die Faktentabelle und eine bestimmte Anzahl von Dimensionstabellen adressiert werden. Die Information, welche Dimensionstabellen in einer materialisierten View verwendet wurden, muss gespeichert werden. Dies ist der Fall, da mit Hilfe des Relationenschemas einer View nachträglich nicht zweifelsfrei entschieden werden kann, welche Dimensionstabellen bei der Erstellung abgefragt wurden. Der Grund dafür ist, das unter Verwendung von Umbenennungen, die mit Hilfe von Alias-Werten realisiert werden können, Attribute nicht zwingend die Namen aus deren Dimensionstabellen besitzen bzw. nicht eindeutig sein müssen.

Die genutzten Dimensionstabellen sind darüber hinaus ein Indiz dafür, welche materialisierten Views zur Restrukturierung einer Anfrage verwendet werden können. Angenommen es existieren n verschiedene Dimensionstabellen, ***DimMV*** ist die Menge der Dimensionstabellen einer materialisierten View und ***DimQ*** ist die Menge der Dimensionstabellen einer Anfrage. Es sind drei Möglichkeiten vorstellbar:

1. $\mathbf{DimMV} = \mathbf{DimQ}$
2. $\mathbf{DimMV} \supset \mathbf{DimQ}$
3. $\mathbf{DimMV} \subset \mathbf{DimQ}$

„$\mathbf{DimMV} = \mathbf{DimQ}$“ bedeutet, dass sowohl die materialisierte View als auch die Anfrage die gleichen Dimensionstabellen verwendet haben bzw. abfragen. Alle Views, die diese Bedingung erfüllen, sollen in die Menge $\boldsymbol{TempDim}_1$ aufgenommen werden. „$\mathbf{DimMV} \supset \mathbf{DimQ}$“ besagt, dass eine View mehr Dimensionstabelle als eine Anfrage enthält. Solche Views werden in der Menge $\boldsymbol{TempDim}_2$ gesammelt.

Als letzte Möglichkeit kann eine View weniger Dimensionstabellen enthalten als eine Anfrage („$\mathbf{DimMV} \subset \mathbf{DimQ}$“). In diesem Fall kann eine View nicht alleine zur Restrukturierung einer Anfrage herangezogen werden, sondern benötigt andere, ergänzende Views. Es könnten Kombinationen von Views gebildet werden, welche zusammen alle notwendigen Informationen zur Restrukturierung enthalten. Dieses wird allerdings in der vorliegenden Diplomarbeit nicht betrachtet, da die Bedingungen zu streng sind, welche an die einzelnen Views gestellt werden würden. Zum Beispiel müsste jede verwendete View paarweise mit einer anderen View der Kombination ein Verbundattribut

besitzen, welches Primärschlüsseleigenschaften besitzt. Das heißt, dass keine Werte mehrfach auftreten dürften und dieses Attribut als identifizierendes Merkmal dienen kann. Der Aufbau der Faktentabelle eines Sternschemas erlaubt ein solches Attribut allerdings nicht, denn der Primärschlüssel einer Faktentabelle ist die Konkatenation aller Fremdschlüssel der verbundenen Dimensionstabellen. Das heißt, das Werte von Attributen mehrfach auftreten können. Es könnte zwar ein künstlicher Schlüssel erzeugt werden, mit dem jedes Tupel eindeutig in der Faktentabelle identifiziert werden kann. Dann müsste allerdings beim Verbund der Views auf die Faktentabelle zugegriffen werden, was vermieden werden soll.

4.1.2. Klassifikationsstufen

Nachdem geprüft wurde, welche Dimensionstabellen zur Verfügung stehen, werden die Klassifikationsstufen betrachtet. Das heißt es wird analysiert, welche Hierarchiestufen in einer materialisierten View verwendet wurden und welche für eine Restrukturierung benötigt werden. Dafür werden die Gruppierungsattribute der SELECT-Klausel der Aggregatanfrage untersucht. Drei Möglichkeiten sind denkbar:

1. Hierarchiestufen View = Hierarchiestufen Anfrage
2. Mindestens eine Hierarchiestufe View $<$ Hierarchiestufe Anfrage
3. Mindestens eine Hierarchiestufe View $>$ Hierarchiestufe Anfrage

Mindestens eine Hierarchiestufe View $<$ *Hierarchiestufe Anfrage* inkludiert, dass die anderen, nicht niedrigeren Hierarchiestufen ebenfalls kleiner oder gleich, auf keinen Fall aber größer im Vergleich zur Anfrage sind.

Die Möglichkeiten werden auf die Mengen $\boldsymbol{TempDim}_1$ und $\boldsymbol{TempDim}_2$ angewendet. Das heißt, dass die in den Mengen enthalten materialisierten Views bzw. Elemente bezüglich deren Klassifikationsstufen untersucht werden.

$\boldsymbol{TempDim}_1$ enthält die Views, deren Dimensionstabellen genau denen einer Anfrage entsprechen. Unter der Voraussetzung, dass die Hierarchiestufen eines Elements von $\boldsymbol{TempDim}_1$ denen der Anfrage entspricht, wird dieses Element in die Menge $\boldsymbol{TempKlass}_1$ aufgenommen. Falls mindestens eine Hierarchiestufe eines Elements kleiner als die Hierarchiestufe der zugeordneten Dimensionstabellen ist, müssen nachträglich unter Verwendung der funktionalen Abhängigkeiten zwischen den Hierarchieelementen einer Klassifikationshierarchie Aggregationen vorgenommen werden. Das heißt, es müssen entlang von Klassifikationspfaden alle Elemente einer niedrigeren Hierarchiestufe zu denen einer höheren Hierarchiestufen zusammengefasst werden. Elemente bei denen solche Schritte notwendig sind, sollen in die Menge $\boldsymbol{TempKlass}_2$ aufgenommen werden. Die letzte Möglichkeit ist, dass mindestens eine Hierarchiestufe eines Elements größer ist als die einer Anfrage. Solche materialisierten Views können nicht zur Restrukturierung einer Anfrage genutzt werden. Dies ist der Fall, da eine nachträgliche Zerlegung einer höheren Hierarchiestufe in die aggregierten Werte der entsprechenden niedrigeren Hierarchiestufen nur durch den Zugriff auf die Faktentabelle und die entsprechende Dimensionstabelle möglich wäre. Auf solche Zugriffe, speziell auf den Zugriff auf die Faktentabelle, soll aber verzichtet werden.

Die Menge $\boldsymbol{TempDim}_2$ enthält die Views, die mehr Dimensionstabellen als die Anfrage enthalten. Durch Beschränkung auf die Dimensionstabellen, die zur Beantwortung einer Anfrage notwendig sind, können die Elemente von $\boldsymbol{TempDim}_2$ wie die der Menge $\boldsymbol{TempDim}_1$ behandelt werden. Das heißt, dass die zusätzlichen Gruppierungsattribute vernachlässigt werden, da sie nicht notwendige Informationen enthalten. Die Elemente von $\boldsymbol{TempDim}_2$ werden somit entweder in $\boldsymbol{TempKlass}_1$ oder $\boldsymbol{TempKlass}_2$ aufgenommen oder nicht weiter beachtet. Im weiteren Verlauf sollte allerdings eine Kostenfunktion in der Lage sein, Views der Menge $\boldsymbol{TempDim}_2$ schlechter zu bewerten als Elemente von $\boldsymbol{TempDim}_1$.

4.1.3. Kennzahlen

Eine Aggregatanfrage enthält neben den Gruppierungsattributen auch eine Menge von Kennzahlen, welche unter Verwendung von Aggregatfunktionen selektiert werden. Die Elemente der Mengen $\boldsymbol{TempKlass}_1$ und $\boldsymbol{TempKlass}_2$ müssen daher analysiert werden, ob

1. alle bzw. mehr Kennzahlen,
2. ein Teil der Kennzahlen oder
3. keine der Kennzahlen

enthalten sind. Falls keine bzw. nicht die richtigen Kennzahlen inkludiert sind, dann werden die entsprechenden Elemente aus den Mengen entfernt. Sind alle bzw. mehr als die notwendigen Kennzahlen enthalten, dann werden die Elemente von $\boldsymbol{TempKlass}_1$ in eine Menge $\boldsymbol{TempKenn}_1$ und Elemente von $\boldsymbol{TempKlass}_3$ in $\boldsymbol{TempKenn}_3$ aufgenommen. Die zusätzlichen Kennzahlen, die nicht benötigt werden, werden wiederum nicht betrachtet. Eine Kostenfunktion sollte allerdings erneut in der Lage sein, materialisierte Views mit der passenden Anzahl von Kennzahlen denen vorzuziehen, die zu viele selektiert haben.

Die letzte Möglichkeit ist, dass materialisierte Views nur einen Teil der Kennzahlen einer Anfrage umfassen. Es werden dementsprechend die Kombinationen von Views gebildet, die zusammen die notwendigen Kennzahlen besitzen. Ob die Kombinationen allerdings zur Restrukturierung verwendet werden können, kann erst nach Analyse der noch offenen Fragestellungen geklärt werden. Daher ist es nicht sinnvoll an dieser Stelle im Voraus die unterschiedlichen Viewkombinationen explizit mit Hilfe von Verbundoperationen zu verknüpfen. Die Elemente der Menge $\boldsymbol{TempKlass}_1$ werden als Kombinationen in die Menge $\boldsymbol{TempKenn}_2$ aufgenommen. Da Elemente von $\boldsymbol{TempKlass}_1$ sowohl die benötigte Anzahl von Dimensionstabellen als auch die passenden Klassifikationsstufen besitzen, könnten in diesem Fall die unterschiedlichen Views mit Hilfe eines Verbunds über die Klassifikationsstufen verknüpft werden. Die durch den Verbund erzeugte View würde somit zusätzlich zu den Dimensionstabellen und Klassifikationsstufen die notwendigen Kennzahlen enthalten. Elemente der Menge $\boldsymbol{TempKlass}_2$ werden in die Menge $\boldsymbol{TempKenn}_4$ als Kombinationen aufgenommen, vorausgesetzt die Elemente der Kombination enthalten die gleichen Klassifikationsstufen.

4.1.4. Aggregatfunktionen

Die in der vorliegenden Diplomarbeit betrachteten Aggregatfunktionen sind: COUNT, AVG, SUM, MAX und MIN. Die Mengen $\boldsymbol{TempKenn}_1$, $\boldsymbol{TempKenn}_2$, $\boldsymbol{TempKenn}_3$ und $\boldsymbol{TempKenn}_4$ werden analysiert, ob die Aggregatfunktionen der Views denen der Anfrage

1. entsprechen,
2. kombiniert entsprechen oder
3. nicht entsprechen.

Das heißt, es wird anhand der Verträglichkeitsmatrix der Abbildung 4.1 geprüft, ob eine View bzw. ein Element der obigen Mengen zur Restrukturierung einer Anfrage verwendet werden könnte oder nicht. Elemente die nicht verwendet werden können, werden wiederum entfernt.

Anfrage \ View	AVG	SUM	MAX	MIN	COUNT
AVG	√	Ⓥ	–	–	Ⓥ
SUM	Ⓥ	√	–	–	Ⓥ
MAX	–	–	√	–	–
MIN	–	–	–	√	–
COUNT	Ⓥ	Ⓥ	–	–	√

Symbol	Bedeutung
√	Mögliche Konstellation
Ⓥ	In Kombination möglich
–	Nicht mögliche Konstellation

Abbildung 4.1.: Verträglichkeitsmatrix der Aggregatfunktionen

Die Abbildung illustriert eine Matrix, die beschreibt, welche Konstellationen zwischen materialisierten Views und einer Anfrage möglich sind. Gesetzt den Fall, dass eine Anfrage Q die Aggregatfunktion SUM enthält, dann könnten Views verwendet werden die ebenfalls die Funktion SUM beinhalten. Daher ist in der Matrix die Zeile und Spalte SUM abgehakt. Falls mögliche Views nur MIN bzw. MAX einbeziehen, dann können diese nicht zur Restrukturierung von Q herangezogen werden. Ausgedrückt wird dies in der Matrix durch einen horizontalen Strich. Als letzte Möglichkeit könnte die Anfrage Q beantwortet werden, wenn zwei Views existieren, wobei View V_1 die Aggregatfunktion AVG enthält, während die View V_2 COUNT genutzt hat. Somit könnte SUM durch die Division der Werte von V_1 durch V_2 berechnet werden. Dargestellt wird dies in der Abbildung 4.1 durch die eingekreisten Haken. Voraussetzung für die letzte Möglichkeit ist allerdings, dass die Views V_1 und V_2 jeweils die gleichen Dimensionstabellen, Klassifikationsstufen und Kennzahlen enthalten. Zur letzten Möglichkeit gehört auch der Fall, dass nicht eine Kombination von Views benötigt wird, sondern dass eine einzige View alle benötigten Aggregate beinhaltet. Das heißt, eine View V_1 hat sowohl AVG

als auch COUNT selektiert, sodass SUM nur unter Nutzung von V_1 berechnet werden kann. Aufgrund einer besseren Übersichtlichkeit wird im folgenden Kontext allerdings davon ausgegangen, dass eine Kombination nötig ist. Dies bedeutet, dass die Kombination nur aus den Views V_1 besteht. Bei der Restrukturierung einer Anfrage wird dieser Aspekt erneut aufgegriffen werden.

Sonderstellung von AVG

Für die Aggregatfunktion AVG gelten weitere Einschränkungen bezüglich der Verwendbarkeit von Views, die in der Abbildung 4.1 aber nicht dargestellt werden. Eine View alleine kann nur dann verwendet werden, falls die View die gleichen Klassifikationsstufen wie die Anfrage besitzt. Ist dies nicht Fall, können auch durch eine spätere Aggregation der Elemente einer niedrigeren Hierarchiestufe nicht die notwendigen Werte berechnet werden, ohne auf die entsprechende Faktentabelle zuzugreifen. Im Gegensatz zu SUM, MAX, MIN und COUNT könnten die AVG-Werte nur aus den Durchschnittswerten niedrigerer Hierarchiestufen berechnet werden, falls eine Wichtung nach den einzelnen Durchschnittswerten möglich wäre. Das heißt, es müsste auf die Faktentabelle zugegriffen werden und die Anzahl der Tupel jedes Dimensionselementes niedrigerer Hierarchiestufe gezählt werden. Danach könnte der Durchschnittswert der höheren Hierarchiestufe („hS“) als gewichtete Summe der Durchschnittswerte der niedrigeren Hierarchiestufen („nS“) berechnet werden. Das Gewicht jedes Summanden wäre in dem Fall der Quotient aus Anzahl der Tupel des Dimensionselementes („$|\mathrm{DE}_x|$“) und der Gesamtanzahl der Tupel aller Dimensionselemente. Es ergibt sich folgende Formel, wobei das Dimensionselement höherer Hierarchiestufe durch die Aggregation der n verschiedenen Dimensionselemente niedrigerer Stufe berechnet werden kann:

$$AVG_{hS} = \frac{|DE_1|}{|DE_1| + ... + |DE_n|} AVG_{1,nS} + ... + \frac{|DE_n|}{|DE_1| + ... + |DE_n|} AVG_{n,nS}$$

Da allerdings auf den Zugriff auf die Faktentabelle verzichtet werden soll, können Anfrage mit AVG mit nur einer View nicht restrukturiert werden, die niedrigere Klassifikationsstufen enthält und AVG verwendet hat. Das heißt aber gleichzeitig auch, dass falls eine weitere View vorhanden wäre, die die gleichen Klassifikationsstufen selektiert wie die der scheinbar nicht verwendbare AVG View, könnte aus dieser Kombination dennoch der AVG Wert der Anfrage nachträglich aggregiert werden.

Eine ausführlichere Erläuterung der Berechnung des Durchschnittes aus den Durchschnittswerten niedrigerer Klassifikationsstufen wird in Anhang A.2 dargestellt.

Anwendung der Verträglichkeitsmatrix

Unter Anwendung der Verträglichkeitsmatrix wird die Menge $\boldsymbol{TempKenn}_1$ aufgespalten in die Mengen $\boldsymbol{TempAgg}_1$ und $\boldsymbol{TempAgg}_2$. Die erste Menge enthält dabei alle Views, bei denen die Aggregatfunktionen einer Anfrage und einer View übereinstimmen, während die zweite Menge Kombinationen von Views zur Berechnung der notwendigen Funktion enthält. Das heißt, falls eine Anfrage AVG benötigt, enthält die

Menge $\boldsymbol{TempAgg_2}$ Paare aus Views, die einerseits SUM andererseits COUNT verwendet haben.

Die Menge $\boldsymbol{TempKenn_2}$ wird analog zu $\boldsymbol{TempKenn_1}$ in zwei Mengen aufgespalten. Diese Mengen sind $\boldsymbol{TempAgg_3}$ und $\boldsymbol{TempAgg_4}$, wobei die Zuordnungskriterien mit denen von $\boldsymbol{TempAgg_1}$ und $\boldsymbol{TempAgg_2}$ korrespondieren.

Nach der Betrachtung der Elemente der Mengen $\boldsymbol{TempKenn_1}$ und $\boldsymbol{TempKenn_2}$ werden die Menge $\boldsymbol{TempKenn_3}$ und $\boldsymbol{TempKenn_4}$ analysiert. Die Elemente der Menge $\boldsymbol{TempKenn_3}$ werden eingeteilt in $\boldsymbol{TempAgg_5}$, $\boldsymbol{TempAgg_6}$ und $\boldsymbol{TempAgg_7}$. Unter der Voraussetzung, dass eine Anfrage die Aggregatfunktionen COUNT, SUM, MAX oder MIN verwendet hat und diese Funktionen auch in einer View von $\boldsymbol{TempKenn_3}$ eingesetzt wurden, werden diese Views in $\boldsymbol{TempAgg_5}$ gespeichert. Falls allerdings eine Anfrage AVG genutzt hat, werden wie oben gezeigt, zwei Views benötigt, bei denen die Klassifikationstufen untereinander gleich sein müssen. Das heißt, die Menge $\boldsymbol{TempAgg_6}$ enthält Paare von Views, wobei die eine View AVG und die andere View COUNT verwendet hat. Die letzte Menge, in die Elemente aus $\boldsymbol{TempKenn_3}$ eingeordnet werden, ist $\boldsymbol{TempAgg_7}$. Es werden in dieser Menge Paare von Views gespeichert, mit denen die notwendigen Aggregatfunktionen eine Anfrage berechnet werden können. Das heißt zum Beispiel, wenn eine Anfrage SUM benötigt, werden Paare von Views gespeichert, bei denen die eine View AVG und die zweite View COUNT genutzt hat. Auch in diesem Fall müssen die Views eines Paares die gleichen Klassifikationstufen besitzen.

Die letzte zu betrachtende Menge ist $\boldsymbol{TempKenn_4}$. Diese Menge wird unter Verwendung der gleichen Einordnungskriterien wie bei der Menge $\boldsymbol{TempKenn_3}$ in die Mengen $\boldsymbol{TempAgg_8}$, $\boldsymbol{TempAgg_9}$ und $\boldsymbol{TempAgg_{10}}$ untergliedert.

4.1.5. Gruppierungsattribute

Nachdem die SELECT- und FROM-Klauseln einer Aggregatanfrage analysiert wurden, müssen die WHERE- und GROUP BY-Klauseln von möglichen Views untersucht werden. Die Gruppierungsattribute entsprechen denen der SELECT-Klausel und damit den selektierten Klassifikationsstufen. Eine View kann im Vergleich zu einer Anfrage

1. die selben Gruppierungsattribute oder
2. nicht die selben Gruppierungsattribute

besitzen. Diese beiden Fälle werden auf die Mengen $\boldsymbol{TempAgg_x}$ mit x $\in$ {1,...,10} angewendet. Es entstehen somit die Mengen $\boldsymbol{TempGroup_x}$ mit x $\in$ {1,...,10}, wobei zum Beispiel Elemente der Menge $\boldsymbol{TempAgg_1}$ bei passenden Gruppierungsattributen in $\boldsymbol{TempGroup_1}$ aufgenommen werden. Wenn die Gruppierungsattribute nicht passen, dass heißt, falls eine View weniger, mehr oder andere Attribute gruppiert hat, wird diese View entfernt. Falls eine View allerdings niedrigere Klassifikationsstufen enthält, dann können die Gruppierungsattribute der View niedrigere Hierarchiestufen im Vergleich zur Anfrage sein. Das heißt, dass anstatt nach den notwendigen höheren Hierarchiestufen einer Anfrage zu gruppieren, könnten die entsprechend niedrigeren Stufen der gleichen Dimension in einer View verwendet werden. Dennoch muss nach der gleichen Anzahl und den gleichen Dimensionen gruppiert worden sein. Ist dies nicht

der Fall, dann passen die Gruppierungsattribute wiederum nicht, sodass die entsprechenden Views entfernt werden.

4.1.6. Restriktionen

Als letzte Fragestellung muss beantwortet werden, ob die Restriktionen einer View mit denen einer Anfrage zusammenpassen. Die Restriktionen werden in der WHERE-Klausel formuliert und schränken die Ergebnismenge einer Aggregatanfrage ein, indem Bedingungen an die Klassifikationsstufen und Kennzahlen gestellt werden. Die Abbildung 4.2 illustriert die Syntax der Restriktionen in Erweiterter Backus-Naur-Form (EBNF), die im Kontext der Arbeit behandelt werden. Für eine verbesserte Übersicht wird in Abbildung 4.2 nicht auf die Möglichkeiten von Klammerungen eingegangen.

```
<Restriktionen>   = ( <Restriktion> <Verbindung> )* <Restriktion>
<Restriktion>     = [ "not" ] <Attribut> <Operation> <Wert>
<Verbindung>      = "and" | "or"
<Attribut>        = <Klassifikationsstufe> | <Kennzahl>
<Operation>       = "=" | "<" | ">"
<Wert>            = <Werte aus Wertebereich des Attributes>
```

Abbildung 4.2.: Syntax von Restriktionen in EBNF

Es werden somit Bereichsanfragen behandelt, wie sie in Abschnitt 2.2.1 vorgestellt wurden. Zum Beispiel könnten Restriktionen wie „Jahr $<$ 2007 AND Verkauf $=$ 180“ thematisiert werden. Die Bedingung „Jahr $=$ 2007“ ist in diesem Beispiel eine Restriktion der Klassifikationsstufe der Dimension Zeit, die mit Hilfe der Verbindung „AND“ mit der Restriktion der Kennzahl „Verkauf $=$ 180“ verbunden wurde.

Durch die Festlegung, dass Attribute konkrete Werte des eigenen Wertebereichs annehmen, wird sichergestellt, dass Restriktionen von Anfragen und Views verglichen werden können. Dabei wird angenommen, dass falls ein Attribut einer View dem einer Anfrage entspricht, diese Attribute den gleichen Datentyp haben. Diese Bedingung stellt dabei keine selbstdefinierte Einschränkung dar, sondern ist die Schlussfolgerung daraus, dass sowohl View als auch Anfrage an die gleichen Basisrelationen gestellt wurden. Die Basisrelationen, das heißt die Relationen des Sternschemas, legen somit den Datentyp fest.

Beziehungen von Restriktionen

Restriktionen von Anfragen können auf verschiedene Art und Weise in Beziehung mit den Restriktionen von Views stehen. Restriktionen von Anfragen können

1. strenger,
2. gleich oder
3. weniger streng

im Vergleich zu den Restriktionen von Views sein. Die unterschiedlichen Fälle sollen anhand eines Beispiels der Hierarchiestufe Jahr der Dimension Zeit erläutert werden. Eine *strengere Restriktion* ist, wenn eine Anfrage die Jahre 2006 bis 2007 und eine View hingegen die Jahre 2005 bis 2008 selektiert hat. *Gleich strenge Restriktionen* sind unter anderem, wenn weder Anfrage noch View jegliche Einschränkungen bezüglich der Jahreszahlen vorgenommen haben bzw. auf das gleiche Jahr limitiert sind. Als letzte Möglichkeit könnten *weniger strenge Restriktionen* vorliegen. Das heißt, dass eine Anfrage die Jahre 2006 bis 2007 ausgewählt hat, während eine View nur die Werte des Jahres 2006 enthält. Des Weiteren wäre es bei weniger strengen Restriktionen denkbar, dass keine View verwendet werden kann. Das heißt, dass zum Beispiel keine verfügbare View die notwendigen Daten umfasst.

Allgemeiner formuliert bedeutet der letzte Fall, dass eine Menge von Views gesucht wird, welche die zur Beantwortung der Anfrage notwendigen Daten enthält. Dieses Problem wird dadurch kompliziert, dass Restriktionen bezüglich sämtlicher Attribute vorgenommen werden können und dass die Tupelmengen der Views untereinander disjunkt sein müssen. Die Disjunktheit ist erforderlich, weil sonst beim Verknüpfen der Views untereinander nicht ohne Zugriffe auf die Faktentabelle entschieden werden könnte, ob die notwendigen Daten im Vergleich zur gestellten Anfrage vorhanden sind. Es wird somit nach einer exakten Überdeckung gesucht. Das Problem der exakten Überdeckung ist NP-vollständig und wird in der vorliegenden Diplomarbeit nicht betrachtet. Das heißt, falls eine Anfrage weniger strenge Restriktionen enthält als die vorhandenen Views, dann wird nicht nach einer Menge von Views gesucht, mit deren Hilfe die Anfrage restrukturiert werden kann.

Anwendung der Restriktionsmöglichkeiten

Die Elemente der Mengen $\boldsymbol{TempGroup}_x$ mit x $\in \{1,...,10\}$ werden untersucht, inwiefern die vorhandenen Views zur Restrukturierung verwendet werden können. Da die Analyse der Restriktionen die letzte Fragestellung ist, welche positiv beantwortet werden muss, entsteht somit die Menge $\boldsymbol{PosMV}$.

Die Mengen $\boldsymbol{TempGroup}_1$, $\boldsymbol{TempGroup}_2$, $\boldsymbol{TempGroup}_3$ und $\boldsymbol{TempGroup}_4$ enthalten jeweils Views, bei denen die Klassifikationsstufen passen. Das heißt, diese haben die gleichen Gruppierungsattribute wie eine gestellte Anfrage gewählt und können somit direkt verglichen werden.

Elemente der Menge $\boldsymbol{TempGroup}_1$ werden in die Menge $\boldsymbol{TempRest}_1$ aufgenommen, wenn die Restriktionen die gleichen sind wie in der Anfrage. Die nächste Möglichkeit ist, dass die Restriktionen einer Anfrage strenger sind als die einer View. Das heißt, dass die View mehr Informationen als die Anfrage enthält und somit nochmals überprüft werden muss. Es muss analysiert werden, ob die Restriktionen der Anfrage auf die entsprechende View anwendbar sind oder nicht. Zum Beispiel könnte eine Anfrage die Jahre einer Dimension Zeit selektiert haben, allerdings bezüglich der Hierarchiestufe Monat Restriktionen enthalten. Solche Restriktionen könnten im Bezug auf eine View nur mit einem Zugriff auf die Basisrelationen angewendet werden. Auf den Zugriff auf die Faktentabelle soll aber verzichtet werden. Das heißt, dass bei strengeren Restrik-

tionen nur Views verwendet werden können, bei denen die Restriktionen einer Anfrage die selektierten Gruppierungsattribute und Kennzahlen betreffen. Diese Views werden in die Menge $\boldsymbol{TempRest}_2$ aufgenommen.

Die Views der Mengen $\boldsymbol{TempGroup}_2$, $\boldsymbol{TempGroup}_3$ und $\boldsymbol{TempGroup}_4$ werden analog zu den Elementen der Menge $\boldsymbol{TempGroup}_1$ behandelt. Das heißt, Elemente von $\boldsymbol{TempGroup}_2$ werden in $\boldsymbol{TempRest}_3$ oder $\boldsymbol{TempRest}_4$, Elemente von $\boldsymbol{TempGroup}_3$ in $\boldsymbol{TempRest}_5$ oder $\boldsymbol{TempRest}_6$ und Elemente von $\boldsymbol{TempGroup}_4$ in $\boldsymbol{TempRest}_7$ oder $\boldsymbol{TempRest}_8$ eingeordnet. Dabei ist aber zu beachten, dass die Menge $\boldsymbol{TempGroup}_2$ und $\boldsymbol{TempGroup}_4$ jeweils aus Kombinationen von Views bestehen. Das heißt, dass die Bedingungen zur Einordnung in die entsprechende Menge $\boldsymbol{TempRest}_x$ mit x $\in$ {3,4,7,8} jeweils für jede View einer Kombination gelten muss. Des Weiteren dürfen die Restriktionen nur die Gruppierungsattribute betreffen. Ist dies nicht der Fall, dann können die Views nicht zur Restrukturierung verwendet werden, da durch die Verknüpfung der Views einer Kombination nicht alle notwendigen Daten erhalten werden können. Es muss bei den Elementen der Mengen $\boldsymbol{TempGroup}_3$ und $\boldsymbol{TempGroup}_4$ darüber hinaus beachtet werden, dass keine Restriktionen bezüglich der nicht betrachteten Kennzahlen vorgenommen wurden.

Die Elemente der Mengen $\boldsymbol{TempGroup}_x$ mit x $\in$ {5,...,10} müssen nachaggregiert werden, weil diese Views niedrigere Klassifikationsstufen enthalten als eine Anfrage. Daraus folgt, dass diese Views keine Restriktionen bezüglich der vorhandenen Kennzahlen haben dürfen, da sonst nachträgliche Aggregationen nicht vollständig möglich wären. Für die Mengen $\boldsymbol{TempGroup}_8$, $\mathbf{TempGroup}_9$ und $\boldsymbol{TempGroup}_{10}$ gilt darüber hinaus, dass auch nicht betrachtete Kennzahlen keine Einschränkungen enthalten dürfen. Ist dies doch der Fall, dann können die entsprechenden Views der Mengen nicht verwendet werden. Die Mengen $\boldsymbol{TempGroup}_x$ mit x $\in$ {6,7,9,10} enthalten Paare von Views, mit denen nachträglich die erforderlichen Aggregatfunktionen berechnet werden sollen. Für jede View eines Paares gilt erneut, dass jeweils die gleiche Restriktionen vorliegen müssen, damit eine spätere Restrukturierung einer Anfrage mit diesen Kombinationen ermöglicht wird. Eine weitere Einschränkung ist, dass eine Anfrage keine Dimensionselemente einer Hierarchiestufe einschränken darf, die niedrigerer als die einer möglichen View sind. Eine Ausnahme stellt dabei die Situation dar, in der eine View genau die gleichen Restriktionen enthält wie die zu restrukturierende Anfrage.

Unter Beachtung der vorgestellten Einschränkungen werden die Elemente der Mengen $\boldsymbol{TempGroup}_x$ mit x $\in$ {5,...,10} eingeteilt in die Mengen $\boldsymbol{TempRest}_x$ mit x $\in$ {9,...,20}. Die Views der Menge $\boldsymbol{TempGroup}_5$ werden somit in die Menge $\boldsymbol{TempRest}_9$ bzw. $\boldsymbol{TempRest}_{10}$ aufgenommen, falls die Restriktionen übereinstimmen bzw. wenn die Anfrage strengere Restriktionen enthält. Die Elemente der Menge $\boldsymbol{TempGroup}_6$ werden analog in $\boldsymbol{TempRest}_{11}$ oder $\boldsymbol{TempRest}_{12}$, $\boldsymbol{TempGroup}_7$ in $\boldsymbol{TempRest}_{13}$ oder $\boldsymbol{TempRest}_{14}$, $\boldsymbol{TempGroup}_8$ in $\boldsymbol{TempRest}_{15}$ oder $\boldsymbol{TempRest}_{16}$, $\boldsymbol{TempGroup}_9$ in $\boldsymbol{TempRest}_{17}$ oder $\boldsymbol{TempRest}_{18}$ und $\boldsymbol{TempGroup}_{10}$ in $\boldsymbol{TempRest}_{19}$ oder $\boldsymbol{TempRest}_{20}$ eingeordnet.

Die Mengen $\boldsymbol{TempRest}_x$ mit x $\in$ {1,...,20} enthalten somit alle Views, die zur Restrukturierung einer Anfrage verwendet werden können. Diese Teilmengen bilden zusammen die Menge $\boldsymbol{PosMV}$. Dabei ist die detaillierte Einteilung der Views notwendig,

da die Zugehörigkeit zu der entsprechenden *TempRest*-Menge Einfluß auf das Restrukturieren einer Anfrage hat. Es muss nachfolgend des Weiteren anhand einer Kostenfunktion geklärt werden, welche der möglichen Views auch tatsächlich eingesetzt werden sollen.

4.2. Kostenfunktion

Materialisierte Views können zur Restrukturierung von Anfragen an ein Sternschema verwendet werden. Dabei ist es möglich, dass nicht nur eine sondern eine Menge von Views geeignet ist. Eine Kostenfunktion wird demnach benötigt, mit welcher die Zugriffszeiten der Views beschrieben werden können. Mit Hilfe dieser Funktion wäre es daraufhin möglich, die kostengünstigste View aus der Menge der materialisierten Views auszuwählen. Wie in Abschnitt 3.4 erläutert wurde, ist das Materialisieren von möglichen Views nicht Bestandteil der Arbeit. Es wurden diesbezüglich aber in Abschnitt 3.2 Algorithmen vorgestellt, die verwendet werden können.

Die Voraussetzung, dass Views vorhanden sind, schränkt die Möglichkeiten der Definition von Parametern einer Kostenfunktion ein. Das heißt, es können keine Informationen in die Funktion einfließen, welche unmittelbar mit der Zeitspanne der Materialisierung zusammen hängen. Zum Beispiel wäre es denkbar, dass die Dauer der Erzeugung einer View gemessen und als Kostenfunktionswert genutzt wird. Solche Informationen sind allerdings nicht verfügbar und können daher nicht berücksichtigt werden. Es können nur Parameter in der Kostenfunktion verwendet werden, die unmittelbar für eine Relation zur Verfügung stehen bzw. berechnet werden können. Die folgenden Parameter sollen zur Berechnung der Kosten einer materialisierten View genutzt werden:

1. Tupelanzahl
2. Seitenanzahl
3. Zugriffszeiten

Die *Tupelanzahl* beschreibt, wie viele Tupel bei Verwendung einer View verarbeitet werden müssen. Diese Variante stellt gleichzeitig die einfachste zu realisierende Methode zur Abschätzung der Größe einer View da.

Die *Seitenanzahl* wird berechnet aus der Größe der Datenbankseiten und der Indexseiten. Die Datenbankseiten enthalten dabei die eigentlichen Datensätze, während die Indexseiten die Zugriffsstrukturen speichern. Welche Indexstrukturen allerdings genutzt werden, spielt bei der Abschätzung der Größe von Views keine Rolle. Die Größe einer Seite kann in unterschiedlichen Datenbanksystemen variieren und sollte deshalb gespeichert werde. Im Bezug auf Abschnitt 4.1 kann die Seitenanzahl unter anderem dazu verwendet werden, um Views zu bevorzugen, die im Vergleich zu anderen Views zum Beispiel nur die notwendige Anzahl von Kennzahlen oder Dimensionstabellen besitzen.

Der letzte Parameter sind die *Zugriffszeiten*, die allerdings nur indirekt die Größe einer View abschätzen. Es soll für den Fall eines verteilten Datenbanksystems die

notwendige Zeit abgeschätzt werden, die zur Rekonstruktion einer auf verschiedene Stationen verteilten View benötigt wird. Eine andere Möglichkeit wäre, dass die Views auf unterschiedlichen Netzwerkressourcen gespeichert wurden und zu einem zentralen Knoten übertragen werden müssen. Die Zugriffszeit könnte in dem Fall auch verwendet werden, um die benötigte Dauer der Übertragung zu ermitteln.

4.2.1. Parameter im verteilten Szenario

Zur Berechnung der Zugriffszeiten im verteilten Szenario werden unterschiedliche Parameter benötigt. Diese Parameter sind:

1. Verbindungsverzögerung
2. Antwortverzögerung
3. Übertragungsrate
4. Tupelgröße
5. Tupelanzahl

Die *Verbindungsverzögerung* gibt an, wie viel Zeit beim Verbindungsaufbau zu einer Ressource vergehen kann. Das heißt es wird angegeben, wie lange es dauert, um auf die Daten einer Netzwerkressource oder Station zugreifen zu können. Die *Antwortverzögerung* ist die Zeitdauer, bevor die jeweilige CPU der Ressource eine Abfrage bearbeiten kann. Dieser Wert repräsentiert demnach die CPU-Auslastung der entsprechenden Ressource. Nachdem eine Verbindung aufgebaut wurde und die notwendigen Daten einer View bereit stehen, müssen diese Daten für eine Rekonstruktion oder anderweitige Nutzung übertragen werden. Die Dauer dieser Übertragung hängt von der *Tupelanzahl*, *Tupelgröße* und *Übertragungsrate* ab, somit also von der Menge zu sendender Daten.

Einteilung der Zugriffszeiten

Mit Hilfe der Verbindungs- und Antwortverzögerung und der Übertragungsdauer wird der entsprechende Kostenfunktionswert einer View berechnet. Dabei sollte allerdings darauf geachtet werden, dass es unterschiedliche Fälle bei den Zugriffszeiten geben kann. Die folgenden drei Fälle sind denkbar:

1. Best Case (Bester Fall)
2. Average Case (Durchschnittlicher Fall)
3. Worst Case (Schlechtester Fall)

Beim *Best Case* ist die Abschätzung der Zugriffszeit optimistisch. Das heißt, dass die Verzögerungen minimal angenommen werden, während die Übertragungsrate maximal eingeschätzt wird. Dies resultiert in einem dementsprechend geringen Wert der Zugriffszeit. Im Gegensatz zum besten Fall ist der *Worst Case* eine pessimistische Abschätzung.

Daraus folgt, dass angenommen wird, dass die Daten mit nur geringer Übertragungsrate und großen Verzögerungen übermittelt werden können. Somit ist der Zugriffswert wesentlich höher. Der Durchschnitt wird repräsentiert durch den *Average Case*. Die Zugriffszeit sollte bei diesem Fall zwischen denen vom besten und schlechtesten Fall liegen.

Damit ein Nutzer entscheiden kann, welcher Fall als Grundlage zur Berechnung dienen soll, müssen die verschiedenen Parameter des verteilten Szenarios für alle drei Fälle vorliegen. Somit kann nachträglich entschieden werden, ob ein eher optimistischer, pessimistischer oder ein durchschnittlicher Wert zum Vergleich von Views genutzt werden soll.

4.2.2. Berücksichtigung der Basisrelationen

In den bisherigen Betrachtungen zum Thema Kostenfunktion wurden Möglichkeiten zum Vergleich von Views untereinander vorgestellt. Das heißt, es soll mit Hilfe von Kostenfunktionswerten die kostengünstigste materialisierte View aus einer Menge von potentiell verwendbaren Views ermittelt werden. Dabei wurden die Basisrelationen allerdings noch nicht berücksichtigt, speziell die Faktentabelle.

Es ist die Situation denkbar, dass ein Sternschema nur wenige Dimensionstabellen und eine relativ kleine Faktentabelle enthält. In diesem Fall kann es sinnvoll sein, die unterschiedlichen Parameter, welche in Abschnitt 4.2 vorgestellt wurden, im Verhältnis zur Faktentabelle darzustellen. Das heißt, es wird zum Beispiel ein Quotient der Tupelanzahl der View und der Faktentabelle gebildet. Dieses Vorgehen hat den Vorteil, dass eine Anfrage nicht restrukturiert wird, falls der Quotient einen Wert gleich eins annimmt. Dies würde nämlich bedeuten, dass eine wichtige Voraussetzung der Motivation zur Restrukturierung einer Anfrage nicht gegeben wäre. Diese Voraussetzung ist, dass der Zugriff auf die Faktentabelle wesentlich teurer im Vergleich zur Verwendung einer materialisierten View ist. Im Zusammenhang mit der Tupelanzahl bedeutet teurer, dass sehr viel mehr Tupel bearbeitet werden müssten.

Die Abbildung 4.3 illustriert die verschiedenen Möglichkeiten, mit denen die Kosten einer materialisierten View beschrieben werden können. Die Auswahl beschreibt dabei den Wert, mit denen im Kapitel 5 der gewünschte Kostenfunktionswert ausgewählt werden wird.

Es werden die unterschiedlichen Möglichkeiten dargestellt, welche im Vorfeld schon beschrieben wurden. Die Abkürzung „FT“ steht dabei für Faktentabelle. Die Spalte „Auswahl“ dient der besseren Übersicht und der Selektion des Kostenfunktionswertes. Das heißt, dass zum Beispiel der Quotient aus der Tupelanzahl einer View und der Faktentabelle berechnet wird, falls die Auswahl „4“ getätigt wird. Welcher Wert allerdings verwendet werden soll, muss zusätzlich gespeichert werden. Falls die Auswahl „3“ oder „6“ benutzt wird, dann muss explizit angegeben werden, welcher Fall Verwendung finden soll. Das heißt, dass angeführt werden muss, ob der „Best Case“, „Worst Case“ oder „Average Case“ zur Berechnung dienen soll.

Kostenfunktionswert	Auswahl
Tupelanzahl	1
Seitenanzahl	2
Zugriffszeit	3
Tupelanzahl / Tupelanzahl FT	4
Seitenanzahl / Seitenanzahl FT	5
Zugriffszeit / Zugriffszeit FT	6

Abbildung 4.3.: Mögliche Kostenfunktionswerte

4.3. Restrukturierung einer Anfrage

Es wurde in Abschnitt 4.1 ein Konzept vorgestellt, mit dem materialisierte Views nach deren Verwendbarkeit bezüglich der Restrukturierung einer Anfrage in Mengen eingeteilt wurden. Von der Einteilung in die unterschiedlichen Mengen hängt ab, wie eine Anfrage umgeschrieben wird, damit diese materialisierte Views verwendet anstatt auf die entsprechende Faktentabelle zuzugreifen. Welche View in einer Menge von möglichen Views ausgewählt wird, wird dabei mit Hilfe der in Abschnitt 4.2 vorgestellten Kostenfunktion ermittelt.

Im Folgenden wird dargestellt, wie aus einer Anfrage Q unter Verwendung materialisierter Views eine Anfrage Q' konstruiert werden kann. Diesbezüglich wird ein detailliertes Beispiel allerdings erst im Abschnitt 5.2 vorgestellt. Es werden nachfolgend die unterschiedlichen Klauseln einer Aggregatanfrage untersucht und falls nötig abgeändert. Die ursprüngliche Anfrage wird dabei nicht umgeschrieben, damit im Fehlerfall oder bei auftretenden Problemen das Original ausgeführt werden kann bzw. zur Behebung der Probleme zur Verfügung steht.

4.3.1. Relationen der restrukturierten Anfrage

Die FROM-Klausel einer Aggregatanfrage Q enthält die Dimensionstabellen und die Faktentabelle, da Anfragen an ein Sternschema gestellt werden. Bei der Konstruktion der restrukturierten Anfrage Q' werden folgende Relationen in die FROM-Klausel aufgenommen:

1. Relationen aller notwendigen Views
2. Relationen aller notwendigen Dimensionstabellen

Relationen aller notwendigen Views bedeutet, dass die Namen der Relationen der Views in die FROM-Klausel von Q' aufgenommen werden. Es kann dabei unterschieden werden, ob nur eine View notwendig ist oder mehrere Views benötigt werden. Zum letzten Fall gehören Kombinationen von Views, mit denen notwendigen Aggregate noch berechnet werden müssen. Das heißt zum Beispiel, falls die Anfrage Q die Aggregatfunktion

AVG enthält, dann könnte Q unter anderem mit Hilfe zweier Views beantwortet werden. Dabei könnte die erste View die Funktion COUNT enthalten, während die zweite die Funktion SUM verwendet hat. Gesetzt den Fall, dass eine View mehrfach in einer Kombination auftritt, dann wird die entsprechende View nur einmal übernommen. Diese Situation kann eintreten, falls eine View die unterschiedlichen Aggregate, die zur Berechnung von zum Beispiel AVG benötigt werden, komplett enthält. *Relationen aller notwendigen Dimensionstabellen* besagt, dass auch Dimensionstabellen zur Restrukturierung adressiert werden müssen. Dies kann unter anderem der Fall sein, falls die Klassifikationsstufen einer Dimension einer View niedriger als die der Anfrage Q sind. Somit muss bei den noch durchzuführenden Aggregationen auf die entsprechenden Dimensionstabelle zugegriffen werden.

Die Tabelle 4.4 illustriert den Zusammenhang der *TempRest*-Mengen und der FROM-Klausel von Q':

Sichten \ DT	nein	ja
eine	1, 2	9, 10
eine / viele	3, 4	11 - 14
viele	5 - 8	15 - 20

Abbildung 4.4.: Restrukturierung der FROM-Klausel

In der Tabelle bezeichnet „DT“ die Dimensionstabellen. Die Einträge entsprechen den Indizes der *TempRest*-Mengen, sodass „1, 2“ für die Mengen $\boldsymbol{TempRest}_1$ bzw. $\boldsymbol{TempRest}_2$ steht.

Die Tabelle besagt, dass zum Beispiel für ein Element der Menge $\boldsymbol{TempRest}_{13}$ eine oder mehrere Relationen von materialisierten Views in die FROM-Klausel von Q' hinzufügt werden müssen. Elemente von $\boldsymbol{TempRest}_{13}$ enthalten im Vergleich zur Anfrage Q die notwendigen Dimensionstabellen, mindestens eine niedrigere Klassifikationsstufe und die notwendigen Kennzahlen. Allerdings enthält Q eine Aggregatfunktion, welche erst mit Hilfe von mehreren Views berechnet werden muss. Daher wurde ein Paar von Views gebildet, wobei diese Views jeweils untereinander die gleichen Klassifikationsstufen besitzen und zusammen die Funktion von Q berechnen können. Die Relationen der beiden Views werden demnach in die FROM-Klausel von Q' aufgenommen. Eine Alternative ist, dass das Paar einer Kombination aus denselben Views besteht. Zum Beispiel kann dieser Fall eintreten, falls eine View die notwendigen Aggregate zur Berechnung der Aggregatfunktion von Q komplett alleine enthält. In diesem Fall wird nur eine View in die FROM-Klausel übernommen.

Die Dimensionstabellen der Klassifikationsstufen, welche niedriger als die notwendigen Stufen von Q sind, werden zusätzlich in die FROM-Klausel der Anfrage Q' übernommen.

Verbundbedingungen

Die Verbundbedingungen, welche in die WHERE-Klausel der restrukturierten Anfrage Q' übernommen werden, hängen direkt von der schon betrachteten FROM-Klausel von Q' ab. Dabei kann unter Anwendung der Abbildung 4.4 unterschieden werden, ob Verbundbedingungen zwischen

1. Views oder
2. Views und Dimensionstabellen

benötigt werden. Falls die FROM-Klausel von Q' nur eine View und keine Dimensionstabellen umfasst, dann werden dementsprechend keine Verbundbedingungen aufgenommen. Für den Fall, dass mehrere *Views* enthalten sind, wird eine Verbundbedingung für jede View und Klassifikationsstufe übernommen. Diese unterschiedlichen Bedingungen werden mit Hilfe von „AND" verknüpft. Mehrere Views bedeutet, dass Kombinationen von Views ausgewählt wurden, deren Klassifikationsstufen mit denen der Anfrage Q übereinstimmen, allerdings nicht alle notwendigen Daten enthalten. Zum Beispiel könnten nicht alle geforderten Kennzahlen in einer View V_1 vorhanden sein. Daher wurde eine weitere View V_2 aufgenommen, die die fehlenden Kennzahlen liefert. V_1 und V_2 müssen demnach über deren sämtliche Klassifikationsstufen verknüpft werden. Das heißt, dass die Verbundbedingungen der WHERE-Klausel wie folgt aussehen könnten:

```
SELECT v1.jahr, v1.produkt, v1.sell, v2.cost
FROM ast1 v1, ast2 v2
WHERE v1.jahr = v2.jahr AND
      v1.produkt = v2.produkt AND
      v1.kunde = v2.kunde
```

In dem Beispiel sind die Klassifikationsstufen „jahr", „produkt" und „kunde", derweil sind die Kennzahlen „sell" und „cost". Die erste Kennzahl ist in V_1 bzw. „ast1" enthalten, während die zweite von V_2 bzw. „ast2" geliefert wird. Verknüpft werden die Views über die vorhandenen Klassifikationsstufen. Es muss dabei beachtet werden, dass nicht alle Klassifikationsstufen zwingend im SELECT-Teil von Q' selektiert sein müssen. Zum Beispiel wird „kunde" nicht selektiert, aber dennoch muss für die richtige Zuordnung der Werte der Kennzahlen zu den entsprechenden Tupeln der anderen View ebenfalls über diese Klassifikationsstufe verknüpft werden.

Sofern die FROM-Klausel von Q' *Views und Dimensionstabellen* enthält, dann müssen die Views wiederum über die Klassifikationsstufen untereinander verbunden werden. Des Weiteren muss für jede Dimensionstabelle geprüft werden, welche Klassifikationsstufe zu welcher View gehört, damit die entsprechende View mit der Dimensionstabelle vereinigt werden kann. Dieser Fall tritt ein, falls Views niedrigere Klassifikationsstufen als die Anfrage Q selektiert haben. Das heißt gleichzeitig, dass eine View nachaggregiert werden muss. Die dafür notwendigen Informationen sind dabei in der Dimensionstabelle enthalten. Das folgende Beispiel illustriert eine mögliche WHERE-Klausel:

```
SELECT zeit.jahr, v1.produkt, SUM(v1.sell)
FROM ast1 v1, zeit
WHERE v1.woche = zeit.woche
GROUP BY zeit.jahr, v1.produkt
```

In dem Beispiel enthält die View V_1 bzw. „ast1“ die Hierarchiestufen „woche“ und „produkt“ und die Kennzahl „sell“. Es wird allerdings durch eine Anfrage Q nicht die Klassifikationsstufe „woche“, sondern „jahr“ abgefragt. Deswegen werden V_1 und „zeit“ über die vorhandene Klassifikationsstufe „woche“ miteinander verknüpft und „sell“ dementsprechend nachaggregiert. Des Weiteren wird die Verbundbedingung in die WHERE-Klausel von Q’ übernommen.

4.3.2. Gruppierungsattribute und Aggregatfunktionen

Die SELECT- und GROUP BY-Klausel enthalten die Gruppierungsattribute bzw. Klassifikationsstufen. Des Weiteren enthält die SELECT-Klausel die Kenngrößen mit den entsprechenden Aggregatfunktionen.

Es wird im folgenden Kontext davon ausgegangen, dass nur eine Aggregatfunktion in der SELECT-Klausel einer Anfrage Q auftritt. Das heißt, bei den Erläuterungen wird aufgrund der verbesserten Übersichtlichkeit und Verständlichkeit nur jeweils eine Aggregatfunktion behandelt. Das Konzept ist allerdings auch auf eine Menge von Aggregatfunktionen anwendbar. Es müsste diesbezüglich iterativ jede Aggregatfunktion mit Hilfe der nachfolgenden Vorgehensweise behandelt werden.

Gruppierungsattribute

Beim Umschreiben muss im Zusammenhang mit den Klassifikationsstufen entschieden werden, aus welcher Relation die entsprechenden Informationen erhalten werden können. Einerseits könnten die notwendigen Daten aus einer View stammen, andererseits aus einer Dimensionstabelle. Der erste Fall tritt ein, falls die Klassifikationsstufen einer Anfrage Q mit denen einer View übereinstimmen, während der zweite Fall für alle Klassifikationsstufen eintritt, die niedriger als die entsprechenden Stufen von Q sind.

Die Gruppierungsattribute der SELECT-Klausel sind ebenfalls in der GROUP BY-Klausel enthalten. Das heißt für die Anfrage Q’, dass die Gruppierungsattribute der Anfrage Q übernommen werden können. Im Zusammenhang mit den Klassifikationsstufen muss allerdings wiederum entschieden werden, aus welchen Relationen die notwendigen Informationen erhalten werden können. Diesbezüglich können die Gruppierungsattribute der SELECT-Klausel von Q’ direkt in die GROUP BY-Klausel übernommen werden.

Aggregatfunktionen

Eine Aggregatfunktion kann in der SELECT-Klausel der Anfrage Q' die folgenden Erscheinungsformen haben:

1. Kennzahl
2. Funktion über eine Kennzahl
3. Mehrere Kennzahlen
4. Funktion über mehrere Kennzahlen

Eine Aggregatfunktion besteht nur aus einer *Kennzahl*, falls eine View V die notwendigen Daten einer Anfrage Q bereits enthält. Das heißt, falls V im Vergleich zu Q die gleichen Klassifikationsstufen, Kennzahlen, Aggregatfunktionen, Dimensionstabellen, Restriktionen und Gruppierungsattribute enthält, dann kann V komplett zur Beantwortung von Q genutzt werden. Es müssten daher keine nachträglichen Aggregationen vorgenommen werden. *Eine Funktion über eine Kennzahl* wird in die SELECT-Klausel von Q' übernommen, falls die vorhandenen Werte einer View V erneut aggregiert werden müssen. Dies wäre unter anderem der Fall, falls V niedrigere Klassifikationsstufen enthält als eine Anfrage Q. Zum Beispiel könnte Q die Summe der jährlichen Verkäufe abfragen, während V die monatlichen Verkäufe enthält. Somit müssten die summierten Werte von V nachträglich erneut summiert werden. Die Aggregatfunktion einer Anfrage Q kann aus *mehreren Kennzahlen* einer oder verschiedener Views berechnet werden, falls die zur Restrukturierung von Q notwendigen Daten nur mit Hilfe der Aggregatfunktionen einer oder verschiedener Views erhalten werden können. Einerseits wäre es denkbar, dass Q den Durchschnitt der jährlichen Verkäufe abfragt, während eine View V_1 die Summe und eine andere View V_2 die Anzahl der jährlichen Verkäufe enthält. Der Durchschnitt könnte demnach durch die Division der Aggregate von V_1 durch V_2 erhalten werden. Andererseits könnte V_1 sowohl die Summe als auch die Anzahl der jährlichen Verkäufe selektiert haben, sodass der Durchschnitt alleine mit Hilfe von V_1 berechnet werden kann. *Eine Funktion über mehrere Kennzahlen* wird eingefügt, falls die Aggregatfunktion einer Anfrage Q aus den Aggregaten mehrerer Views erst berechnet werden muss. Zum Beispiel könnte Q die Summe der jährlichen Verkäufe benötigen, während vorhandene Views mit niedrigeren Klassifikationsstufen den monatlichen Durchschnitt und die Anzahl beinhalten. Somit könnte die monatliche Summe aus den Views berechnet werden, die daraufhin nachträglich mit Hilfe der Aggregatfunktion SUM aggregiert werden muss. Wiederum könnte eine View alle notwendigen Daten enthalten, sodass die Summe mit nur einer View errechnet werden könnte.

AVG über mehrere Kennzahlen

Die Situation, dass eine Anfrage Q die Aggregatfunktion AVG verwendet und nur Views zur Restrukturierung verfügbar sind, die niedrigere Klassifikationsstufen enthalten, stellt einen Sonderfall dar. Das Problem ist, dass eine nachträgliche Aggregation mit nur einer den Durchschnitt enthaltenden View V_1 nicht ohne Weiteres möglich ist.

Es wird eine zweite View V_2 benötigt, welche die gleichen Klassifikationsstufen wie V_1 enthält. Gesetzt den Fall V_1 enthält den monatlichen Durchschnitt der Verkäufe und V_2 die monatliche Anzahl der Verkäufe, dann kann der jährliche Durchschnitt der Verkäufe der Anfrage Q wie folgt berechnet werden:

```
SELECT z1.jahr, SUM(v1_avg.sell * v2_count.sell) /
   (SELECT SUM(v3_count.sell)
    FROM zeit z2, ast2 v3_count
    WHERE <Verbundbedingungen> AND
          z2.jahr = z1.jahr
    GROUP BY <Gruppierungsattribute>)
FROM zeit z1, ast1 v1_avg, ast2 v2_count
WHERE <Verbundbedingungen> AND
      <Restriktionen>
GROUP BY z1.jahr
```

Es werden in dem dargestellten SQL-Ausdruck die Dimensionstabelle „zeit“ und die Views „ast1“ und „ast2“ verwendet. Die View „ast1“ enthält die monatliche Anzahl der Verkäufe „sell“, während „ast2“ die Anzahl der Verkäufe zur Verfügung stellt. Durch den inneren SQL-Ausdruck bzw. die Subquery wird die Anzahl aller Dimensionselemente ermittelt, welche als Divisor dient. Der Dividend hingegen ist die Summe aller Verkäufe eines Jahres, welche aus der Multiplikation der entsprechenden Durchschnittswerte und der Anzahl der Dimensionselemente eines Monats berechnet wird.

Alternativ kann die View V_1 sowohl den Durchschnitt, als auch die entsprechende Anzahl von Verkäufen enthalten. Obiges Beispiel würde in dem Fall wie folgt abgeändert werden:

```
SELECT z1.jahr, SUM(v1.sell_avg * v1.sell_count) /
   (SELECT SUM(v2.sell_count)
    FROM zeit z2, ast1 v2
    WHERE <Verbundbedingungen> AND
          z2.jahr = z1.jahr
    GROUP BY <Gruppierungsattribute>)
FROM zeit z1, ast1 v1
WHERE <Verbundbedingungen> AND
      <Restriktionen>
GROUP BY z1.jahr
```

In dem Beispiel wurde zur Unterscheidung der Verkaufsattribute eine Umbenennung vorgenommen. Das heißt, dass „sell_avg“ den Durchschnitt und „sell_count“ die Anzahl der Verkäufe von View V_1 bzw. „ast1“ enthält.

Eine ausführlichere Erläuterung der Berechnung des Durchschnittes aus den Durchschnittswerten niedrigerer Klassifikationsstufen wird in Anhang A.2 dargestellt.

Anwendung auf die *TempRest*-Mengen

Die Tabelle 4.5 illustriert den Zusammenhang der *TempRest*-Mengen und der SELECT- bzw. GROUP BY-Klausel von Q':

Gruppierungsattribute / Aggregatfunktionen	View.Stufe	View.Stufe & DT.Stufe
Kennzahl	1, 2, 5, 6	-
Ke <op> Ke	3, 4, 7, 8	-
Funktion(Ke)	-	9, 10, 15, 16
Funktion(Ke <op> Ke)	-	13, 14, 19, 20
AVG(Ke <op> Ke) / SQ	-	11, 12, 17, 18

Abbildung 4.5.: Restrukturierung der SELECT- und GROUP BY-Klausel

In der Tabelle bezeichnet „DT" die Dimensionstabellen. Die Einträge entsprechen den Indizes der *TempRest*-Mengen, sodass „1" für die Menge $\boldsymbol{TempRest}_1$ steht. Des Weiteren steht „Ke" als Abkürzung für Kennzahl, „<op>" für Operation und „SQ" für Subquery. Die Operationen können eine Multiplikation bzw. Division sein, je nachdem welche Aggregatfunktionen verwendet wurden. Dabei wird ausgenutzt, dass der Durchschnitt der Quotient aus Summe und Anzahl ist. Die Multiplikation wird als Operation verwendet, falls eine Anfrage Q SUM und die Views AVG und COUNT selektiert haben. Die Division hingegen wird als Operation benötigt, falls einerseits Q AVG und die Views SUM und COUNT anderseits Q COUNT und die Views AVG und SUM selektiert haben.

Für die Elemente der Menge $\boldsymbol{TempRest}_9$ bedeutet dies zum Beispiel, dass deren Gruppierungsattribute sowohl Daten aus den Dimensionstabellen als auch aus den verwendeten Views enthalten. Die Daten der Dimensionstabellen werden dabei jeweils für jede Klassifikationsstufe benötigt, die niedriger im Vergleich zu Q ist und somit nachaggregiert werden muss. Die nachträgliche Aggregation ist ebenfalls der Grund dafür, dass die Aggregatfunktion in der SELECT-Klausel der Anfrage Q' die dargestellte Erscheinungsform „Funktion(Ke)" hat.

4.3.3. Restriktionen

Die WHERE-Klausel einer Aggregatanfrage besteht neben den Verbundbedingungen aus Restriktionen. Die Verbundbedingungen verknüpfen dabei die Faktentabelle mit den Dimensionstabellen, während die Restriktionen die Ergebnismenge einer Anfrage einschränken.

Im Bezug auf die *TempRest*-Mengen wurden in Abschnitt 4.1 verschiedene Fälle der Beziehungen zwischen den Restriktionen einer Anfrage Q und denen von Views vorgestellt und analysiert. Eine Anfrage kann in der vorliegenden Diplomarbeit entweder

1. strengere oder

2. gleiche

Restriktionen im Vergleich zu einer View besitzen. Falls eine Anfrage Q die *gleichen* Restriktionen enthält, dann werden diese nicht in die WHERE-Klausel von Q' übernommen. Dies ist damit zu begründen, dass Views die entsprechenden Restriktionen schon auf deren Ergebnismenge angewendet haben. Eine Anfrage Q kann aber *strengere* Restriktionen im Bezug auf Views haben, sodass diese auf Q' angewendet werden müssen. Es muss dabei beachtet werden, welche Klassifikationsstufen eingeschränkt werden. Einerseits können Attribute von Dimensionstabellen, andererseits Attribute von Views betroffen sein. Der erste Fall tritt ein, falls zum Beispiel eine Anfrage Q die Jahreszahlen von Verkäufen einschränkt, eine View V_1 allerdings nur die wöchentlichen Verkäufe selektiert hat. Somit müsste V_1 nachaggregiert und die Restriktion bezüglich der Werte der zugehörigen Dimensionstabelle vorgenommen werden. Falls Attribute von Q begrenzt werden, die in V_1 direkt enthalten sind, dann müssen diese Restriktionen in Q' dementsprechend auf die Werte von V_1 angewendet werden.

Beim Auftreten von mehreren Restriktionen in der WHERE-Klausel von Q' werden diese analog zur Anfrage Q miteinander verbunden. Das heißt, dass gemäß Abbildung 4.2 entweder „AND“ oder „OR“ verwendet wird.

Das folgende Beispiel illustriert den Aufbau einer restrukturierten Anfrage Q':

```
SELECT z1.jahr, v1.produkt, SUM(v1.sell)
FROM zeit z1, ast1 v1
WHERE z1.woche = v1.woche AND
      z1.jahr < 2008 AND
      v1.produkt = 'tisch'
GROUP BY z1.jahr, v1.produkt
```

In dem Beispiel soll eine Anfrage Q restrukturiert werden, die die jährlichen Verkäufe eines Produktes vor einem angegebenen Jahr abfragt. Es steht zur Beantwortung eine View V_1 bzw. „ast1“ zur Verfügung, wobei die View die wöchentlichen Verkäufe aller Produkte enthält. Die Restriktionen der Anfrage sind somit im Vergleich zu V_1 strenger und müssen in Q' übernommen werden. Neben der Verbundbedingung „z1.woche = v1.woche“ enthält Q' die Restriktionen „z1.jahr < 2008“ und „v1.produkt = 'tisch'“, welche mit „AND“ verbunden sind. Die erste Restriktion schränkt die Klassifikationsstufe der Dimensionstabelle „zeit“ ein, da V_1 nur die Wochen selektiert hat. Es können demnach auf V_1 keine direkten Einschränkungen im Zusammenhang mit dem Jahr 2008 getätigt werden. Die zweite Restriktion wird allerdings unmittelbar auf die Klassifikationsstufe von V_1 angewendet, da V_1 diesbezüglich die gleiche Stufe enthält wie die Anfrage Q.

Anwendung auf die *TempRest*-Mengen

Die Tabelle 4.6 illustriert den Zusammenhang der *TempRest*-Mengen mit den WHERE-Klauseln der Anfrage Q und der restrukturierten Anfrage Q':

Anfrage Q / Anfrage Q'	Keine	View.Stufe	View.Stufe & DT.Stufe
gleiche	1, 3, 5, 7, 9, 11, 13, 15, 17, 19	-	-
strengere	-	2, 4, 6, 8	10, 12, 14, 16, 18, 20

Abbildung 4.6.: Restriktionen der WHERE-Klausel

In der Tabelle bezeichnet „DT“ die Dimensionstabellen, während die Einträge den Indizes der *TempRest*-Mengen entsprechen Das heißt, dass zum Beispiel „1“ für die Menge $\boldsymbol{TempRest}_1$ steht. Es wird dargestellt, welchen Aufbau die Restriktionen der restrukturierten Anfrage Q' besitzen, wenn die entsprechende Anfrage Q gleiche bzw. strengere Restriktionen im Vergleich zu den Elementen der *TempRest*-Mengen hat.

Ein Element der Menge $\boldsymbol{TempRest}_{10}$ kann demzufolge zum Beispiel Restriktionen bezüglich der Klassifikationsstufen der Dimensionstabellen oder auch bezüglich der Stufen der View enthalten, weil die Anfrage Q strengere Einschränkungen besitzt.

4.4. Wartung materialisierter Views

Die unterschiedlichen materialisierten Sichten müssen aktualisiert werden, falls die zugrunde liegenden Relationen des Sternschemas verändert werden. Eine Änderung ist dabei entweder das Einfügen, Löschen oder Verändern bzw. Update von Datenwerten der Basisrelationen. Es wird im folgenden Kontext eine „Snapshot Aktualisierung“ beschrieben, mit der Views zu Ruhezeiten des Data Warehouses aktualisiert werden können. Dabei stehen im Bezug auf Abschnitt 3.3 sämtliche Informationen der Basisrelationen und Sichten zur Verfügung. Des Weiteren sind die Sichten die gespeicherten Ergebnisse von Aggregatanfragen.

Trigger und Datenbankviews

Es wird ein Mechanismus benötigt, mit dem Änderungen an den Basisrelationen erkannt werden können. Dabei muss unterschieden werden, ob eine Änderung ein Einfügen, Löschen oder Update ist. Trigger bieten diese Funktionalität. Ein Trigger ist eine Anweisung oder Prozedur, die beim Eintreten eines bestimmten Ereignisses automatisch vom Datenbank Management System (DBMS) ausgeführt wird. Das heißt, dass für jede Relation vom Sternschema entsprechende Trigger definiert werden müssen. Aufgrund der „Snapshot Aktualisierung“ werden Änderungen allerdings nicht sofort an die materialisierten Views weitergeleitet, sondern es wird protokolliert, ob ein Einfügen, Löschen oder Update durchgeführt wurde. Somit werden Änderungen der Basisrelatio-

nen erst in den Ruhezeiten des Data Warehouses an die vorhandenen materialisierten Views weitergegeben.

Nach der Erkennung von Änderungen müssen die materialisierten Views angepasst werden. Das heißt, dass die gespeicherten Views so abgeändert werden müssen, dass deren Datenwerten mit denen der Basisrelationen übereinstimmen. Die Idee ist, dass neben den materialisierten Views ebenfalls klassische Datenbankviews verwendet werden. Datenbankviews fragen beim Aufruf die zugrunde liegenden Basisrelationen ab, sodass diese Views die aktuellen Datenwerte des Data Warehouses enthalten. Falls Änderungen an den Basisrelationen vollzogen worden sind, dann können in den Ruhezeiten des Data Warehouses die materialisierten Views mit den Datenbankviews verglichen und entsprechend abgeändert werden. Das heißt, dass zur Aktualisierung von materialisierten Views auf die Basisrelationen zugegriffen wird.

4.4.1. Faktentabelle

Im Zusammenhang mit der Faktentabelle und den Änderungen sind die folgenden Fälle denkbar:

1. Einfügen von Tupeln
2. Löschen von Tupeln
3. Update von Tupeln

Das *Einfügen von Tupeln* bedeutet, dass neue Kennzahlen hinzugefügt werden. Diese Kennzahlen werden allerdings von keiner materialisierten View berücksichtigt, sodass die Aggregate der Sichten angepasst werden müssen. Es kann diesbezüglich unterschieden werden, ob eine View entweder die niedrigsten Klassifikationsstufen oder mindestens eine höhere Stufen einer Dimension selektiert hat. Falls alle Klassifikationsstufen die niedrigsten sind, dann entsprechen diese Stufen den Primärschlüsseln der Dimensionstabellen und sind somit direkt in den Tupeln der Faktentabelle enthalten. Das heißt, dass die neu eingefügten Tupel der Faktentabelle in eine entsprechende materialisierte View übernommen werden können. Wenn mindestens eine Klassifikationsstufe einer View V höher ist, dann enthält V die aggregierten Werte der Tupel der Faktentabelle. Es müsste somit nachvollzogen werden, wie die neuen Kennzahlen in die vorhandenen Tupel von V eingefügt werden müssen. Das heißt, dass die Faktentabelle und die materialisierten Views abgefragt werden. Des Weiteren würde beim abschließenden Vergleich der Datenbankview und der entsprechenden materialisierten View erneut auf die Faktentabelle zugegriffen werden. Daher ist es sinnvoller, dass alle Tupel einer materialisierten View entfernt und aus der Datenbankview neu übernommen werden, falls mindestens eine Klassifikationsstufe höher ist.

Beim *Löschen von Tupeln* aus der Faktentabelle muss wiederum entschieden werden, auf welche Art und Weise die materialisierten Views betroffen sind. Analog zum Einfügen von Tupeln entscheiden die Klassifikationsstufen, ob aus der Faktentabelle entfernte Tupel direkt in den materialisierten Views gelöscht werden können, oder ob

es sinnvoller ist, alle Tupel zu löschen und danach die Tupel der Datenbankview zu übernehmen.

Das *Update von Tupeln* betrifft die Kennzahlen der Faktentabelle, während die Attributwerte der Schlüssel-/Fremdschlüsselbeziehungen bei den vorhandenen Tupeln nicht verändert werden. Eine Änderung dieser Attributwerte wird durch das Einfügen neuer Tupel realisiert, da beim Update noch benötigte Informationen verloren gehen könnten. Bei einem Update der Kennzahlen können die aggregierten Werte einer materialisierten View V unabhängig von den Klassifikationsstufen übernommen werden. Dabei müssen die Tupel von V mit der entsprechenden Datenbankview verglichen und gegebenenfalls abgeändert werden.

Änderungen von materialisierten Views

Es werden drei Trigger zum Überwachen der möglichen Änderungen benötigt, welche die durchgeführten Änderungsoperationen anzeigen. Durch dieses Vorgehen kann in der Ruhephase des Data Warehouses entschieden werden, wie eine materialisierten View aktualisiert werden kann. Die Abbildung 4.7 illustriert, wie materialisierte Views bei Änderungen der Faktentabelle verändert werden:

Änderung / Klassifikationsstufe	Einfügen	Löschen	Update	Kombination
Nur niedrigsten	Einfügen	Löschen	Update	Löschen & Einfügen
Min. eine höhere	Löschen & Einfügen	Löschen & Einfügen	Update	Löschen & Einfügen

Abbildung 4.7.: Wartung bei Änderungen der Faktentabelle

Es wird dargestellt, wie eine materialisierte View unter Berücksichtigung der Klassifikationsstufen geändert wird. Das heißt unter anderem, dass beim Entfernen von Tupeln aus der Faktentabelle und beim Vorhandensein höherer Klassifikationsstufen zuerst alle Tupel aus der materialisierten View gelöscht werden. Danach werden die Tupel der entsprechenden Datenbankview übernommen. Wenn eine Kombination verschiedener Änderungen vorliegt, dann werden generell alle Tupel gelöscht und neu eingefügt. Das heißt zum Beispiel, wenn in der Faktentabelle sowohl Tupel gelöscht als auch aktualisiert wurden, dann werden alle Tupel der materialisierten View wiederum zuerst gelöscht und dann aus der Datenbankview übernommen. Dieses Vorgehen ist sinnvoller, da ansonsten beim angeführten Beispiel zuerst die entsprechenden Tupel gelöscht werden müssten, um danach die noch vorhandenen Tupel zu aktualisieren. Das bedeutet, dass mindestens zwei Mal mit Hilfe der Datenbankviews auf die Basisrelationen zugegriffen werden würde, während beim kompletten Löschen aller Tupel mit anschließender Übernahme der Datenwerte aus der Datenbankview nur ein Zugriff benötigt wird.

4.4.2. Dimensionstabellen

Es können unterschiedliche Fälle auftreten, je nachdem welche Änderungen auf einer Dimensionstabelle vollzogen werden. Nachfolgend wird aufgrund der Übersichtlichkeit von nur einer Dimensionsrelation ausgegangen, die Ergebnisse können aber auf alle Dimensionen übertragen werden. Die folgenden Situationen sind denkbar:

1. Einfügen oder Löschen von Tupeln
2. Update von vorhandenen Tupeln

Das *Einfügen oder Löschen von Tupeln* aus den Dimensionstabellen wird nicht mit Triggern überwacht. Durch das Einfügen werden Tupel hinzugefügt, welche noch nicht in der entsprechenden Faktentabelle enthalten sind. Das heißt, es existieren diesbezüglich keine Kennzahlen, die in den entsprechenden materialisierten Views enthalten sind und somit verändert werden müssen. Falls die Primärschlüssel neu eingefügter Tupel einer Dimension mit entsprechenden Kennzahlen in die Faktentabelle übernommen werden, dann wird dies durch den entsprechenden Trigger der Faktentabelle erkannt und behandelt. Beim Löschen von Tupeln aus den Dimensionstabellen sind zwei Fälle denkbar. Es können Tupel betroffen sein, die entweder als Fremdschlüssel in der Faktentabelle enthalten sind oder nicht. Falls die Schlüssel der Tupel nicht in der Faktentabelle verwendet wurden, dann können die entsprechenden Tupel ohne Auswirkungen auf die materialisierten Views in den Dimensionstabellen gelöscht werden. Sind die Schlüssel der gelöschten Tupel allerdings als Fremdschlüssel in der Faktentabelle enthalten, dann müssen die entsprechenden Tupel wegen den vorliegenden Schlüssel-/Fremdschlüsselbeziehungen ebenfalls aus der Faktentabelle entfernt werden. Dies wird wiederum durch den Trigger der Faktentabelle erkannt und behandelt.

Beim *Update von vorhandenen Tupeln* werden die Datenwerte von den unterschiedlichen Attributen aktualisiert. Dies stellt allerdings einen Sonderfall dar, da aufgrund der Persistenz der Data Warehouse-Daten das erneute Einfügen veränderter Daten dem Update vorhandener Tupel vorgezogen wird. Das Ändern von vorhandenen Datenwerten ist nämlich dem Verlust von Informationen gleichzusetzen, die in Zukunft eventuell noch benötigt werden könnten. Deshalb wird das Update von vorhandenen Tupeln in der vorliegenden Diplomarbeit nicht thematisiert.

5. Umsetzung

Zur Realisierung des in Kapitel 4 vorgestellten Konzepts werden verschiedene Relationen zur Speicherung von Metadaten materialisierter Views benötigt, welche nachfolgend erklärt werden. Des Weiteren wird anhand eines Beispiels der Optimierungsprozess einer Anfrage vorgestellt und erläutert, wobei ebenfalls auf die Wartung vorhandener materialisierter Views eingegangen wird.

5.1. Speicherung von Metadaten

Abbildung 5.1 illustriert die unterschiedlichen Relationen, die zur Optimierung von Aggregatanfragen an ein Data Warehouse benötigt werden. Die unterschiedlichen Bestandteile werden nachfolgend in den entsprechenden Abschnitten näher erläutert. Des Weiteren sind im Anhang B.1 die SQL-Statements zur Erzeugung der Relationen und in B.2 Anfragen zur Ermittlung der Kostenfunktionswerte aufgelistet.

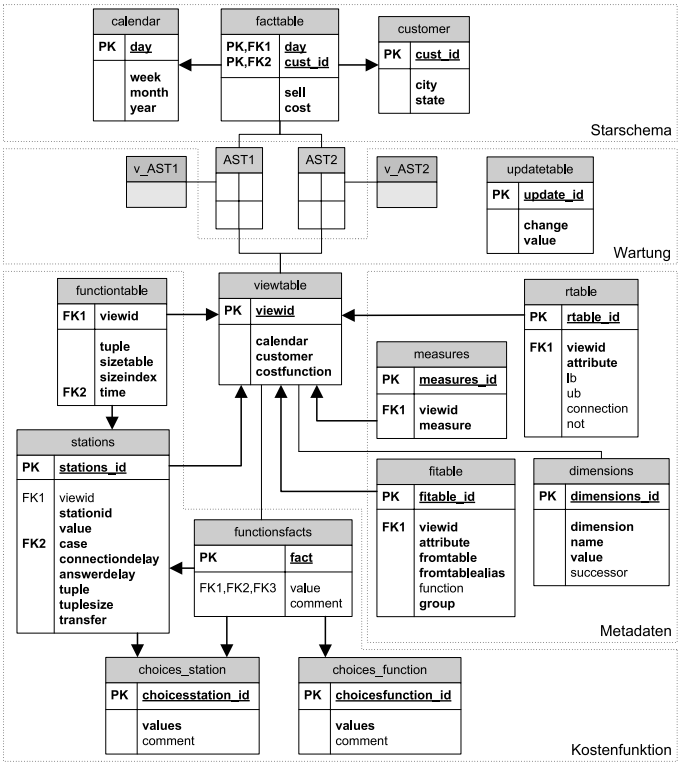

Abbildung 5.1.: Relationenschema zur Speicherung von Metadaten

5.1.1. Starschema

Das in Abbildung 5.1 dargestellte Starschema ist die Grundlage für die folgenden Erläuterungen. Die Dimensionstabellen sind „calendar“ (Kalender) und „customer“ (Kunde) und die Faktentabelle ist „facttable“, welche die Kennzahlen „sell“ (Verkauf) und „cost“ (Kosten) enthält. Die Dimensionen bilden die in Abbildung 5.2 dargestellten Hierarchien.

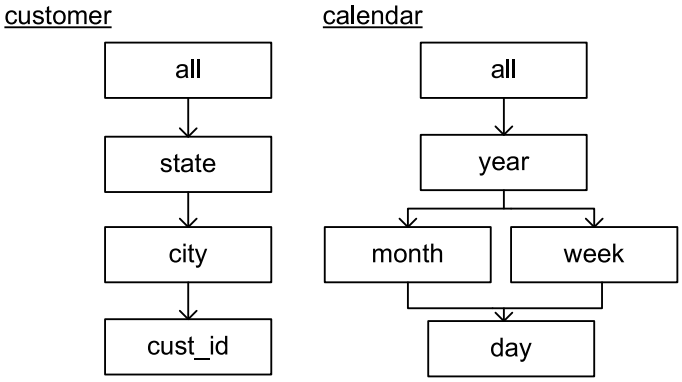

Abbildung 5.2.: Hierarchien der Dimensionstabellen

Die Dimension „customer“ bildet eine einfache Hierarchie mit nur einem Konsolidierungspfad, während „calendar“ eine parallele Hierarchie mit zwei Konsolidierungspfaden besitzt. Die Dimensionselemente „week“ (Woche) und „month“ (Monat) der Dimension „calendar“ sind unabhängige Hierarchiestufen, sodass die Stufe „year“ (Jahr) entweder durch die Aggregation aller Monate oder durch die Aggregation aller Wochen erhalten werden kann.

5.1.2. Verwaltung materialisierter Views

An das Starschema aus Abschnitt 5.1.1 werden Aggregatanfragen gestellt, wobei die Ergebnisse als materialisierte Views gespeichert werden können. Die folgenden Views stehen zur Optimierung von Anfragen zur Verfügung:

```
CREATE TABLE 'da'.'AST1'
    SELECT calendar.month, customer.city,
           SUM(facttable.cost) AS cost,
           SUM(facttable.sell) AS sell_sum,
           COUNT(facttable.sell) AS sell_count
    FROM calendar, customer, facttable
    WHERE calendar.day = facttable.day AND
          customer.cust_id = facttable.cust_id AND
          calendar.year < 2008
    GROUP BY calendar.month, customer.city;
```

Die materialisierte View AST1 selektiert die monatlichen Kosten, die Summe der Verkäufe und die Anzahl der Verkäufe in Abhängigkeit der entsprechenden Städte. Dabei werden nur die Kennzahlen berücksichtigt, die vor dem Jahr 2008 gesammelt wurden. Das Akronym „da“ steht für „Diplomarbeit“ und bezeichnet das Datenbankschema, in dem die verschiedenen Relationen aus Abbildung 5.1 gespeichert werden.

```
CREATE TABLE 'da'.'AST2'
  SELECT calendar.day, customer.cust_id,
         SUM(facttable.cost) AS cost,
         AVG(facttable.sell) AS sell_avg,
         COUNT(facttable.sell) AS sell_count
  FROM calendar, customer, facttable
  WHERE calendar.day = facttable.day AND
        customer.cust_id = facttable.cust_id AND
        calendar.year < 2008 AND
       NOT (calendar.month = 'November') AND
       NOT (calendar.month = 'Juni')
  GROUP BY calendar.day, customer.cust_id;
```

Die materialisierte View AST2 selektiert die täglichen Kosten, Durchschnittsverkäufe und die Anzahl der Verkäufe von Kunden. Dabei werden nur die Kennzahlen berücksichtigt, die vor dem Jahr 2008 gesammelt wurden, und nicht den Monaten November und Juni zugeordnet werden können.

Viewtable

Die materialisierten Views werden in der Relation „viewtable“ verwaltet und gespeichert. Dabei ist die ViewID („viewid“) sowohl Primärschlüssel als auch als Konkatenation aus „AST“ und <ViewID> im Name der Relation enthalten. Des Weiteren wird jeder View ein Kostenfunktionswert zugeordnet, der im Attribut „costfunction“ gespeichert ist. Wie dieser Wert berechnet und ausgewählt wird, wird detailliert in Abschnitt 5.1.4 behandelt.

Die Attribute „calendar“ und „customer“ enthalten die Klassifikationsstufen der entsprechenden Dimensionen. Das heißt, dass jeder materialisierten View eine Kombination von Werten zugeordnet wird. Die Werte und die zugeordneten Hierarchiestufen werden dabei in der Relation „dimensions“ gespeichert. Durch die Festlegung der ViewID als Primärschlüssel, können auch Views verwaltet werden, die nicht alle verfügbaren Dimensionen selektiert haben. Falls eine View eine verfügbare Dimension nicht selektiert hat, dann wird beim entsprechenden Dimensionsattribut der „viewtable“ ein Nullwert eingefügt.

5.1.3. Metadaten materialisierter Views

Um analysieren zu können, ob eine materialisierte View zur Restrukturierung einer Aggregatanfrage verwendet werden kann, werden unterschiedliche Metadaten benötigt. Diese werden in den Relationen gespeichert, welche in Abbildung 5.3 dargestellt sind.

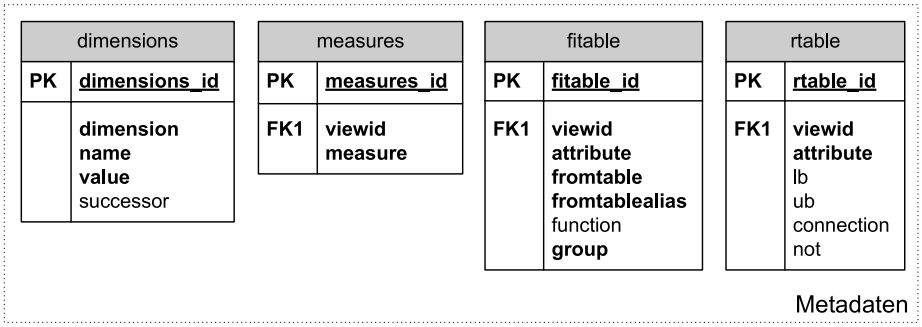

Abbildung 5.3.: Metadaten des Relationenschemas

Die Relation „dimensions“ enthält Informationen über die Klassifikationshierarchien und Dimensionen des Starschemas. Es sind die Vorgänger jeder Hierarchiestufe angegeben („successor“), wodurch einfache und parallele Hierarchien analysiert werden können. Des Weiteren werden den Hierarchiestufen die entsprechenden Werte zugeordnet („value“), welche eine Klassifikationsstufe in den Dimensionsattributen der Relation „viewtable“ annehmen kann. Die Abbildung 5.4 illustriert die Relation „dimensions“.

dimensions_id	dimension	name	value	successor
1	calendar	day	0	NULL
2	calendar	week	1	0
3	calendar	month	2	0
4	calendar	year	3	0
5	calendar	year	3	1
6	calendar	year	3	2
7	calendar	all	4	0
8	calendar	all	4	1
9	calendar	all	4	2
10	calendar	all	4	3
11	customer	cust_id	0	NULL
12	customer	city	1	0
13	customer	state	2	0
14	customer	state	2	1
15	customer	all	3	0
16	customer	all	3	1
17	customer	all	3	2

Abbildung 5.4.: Hierarchiestufen der Dimensionstabellen

Unter Beachtung der dargestellten Werte besitzt die View AST1 die Kombination (2,1), während AST2 die Kombination (0,0) hat. Dabei ist zu beachten, dass die Position innerhalb einer Kombination unter Berücksichtigung der lexikalischen Sortierung der Dimensionen vergeben wird. Das heißt, weil „calendar“ kleiner im Vergleich zu „customer“ ist, entspricht die erste Stelle der Kombination der Klassifikationsstufe von Kalender, während die zweite der Stufe der Dimension Kunde zugeordnet werden kann.

Kennzahlen

Die Kennzahlen einer View werden in der Relation „measures“ gespeichert, welche in Abbildung 5.5 dargestellt ist. Das heißt, dass in dieser Tabelle zu jeder View die Kennzahlen im Vergleich zu den vorhandenen Kenngrößen der Faktentabelle enthalten sind. Die Relation „measures“ enthält darüber hinaus als Fremdschlüssel die ViewID einer materialisierten View.

measures_id	viewid	measure
1	1	sell
2	1	cost
3	2	sell
4	2	cost

Abbildung 5.5.: Kennzahlen der materialisierten Views

Es werden die Kennzahlen gespeichert, welche in einer View verwendet wurden. Dabei wird allerdings nicht gespeichert, welche Aggregatfunktionen auf eine Kennzahl angewendet wurden. Dies wird unter anderem in der Relation „fitable“ vermerkt.

Zusätzliche Informationen

Die Relation „fitable“ ist ein Akronym für „Further Information Table“ (Tabelle mit zusätzlichen Informationen) und wird in Abbildung 5.6 dargestellt.

fitable_id	viewid	attribute	fromtable	fromtablealias	function	group
1	1	month	calendar	month	NULL	1
2	1	city	customer	city	NULL	1
3	1	cost	facttable	cost	sum	0
4	1	sell_sum	facttable	sell	sum	0
5	1	sell_count	facttable	sell	count	0
6	2	day	calendar	day	NULL	1
7	2	cust_id	customer	cust_id	NULL	1
8	2	cost	facttable	cost	sum	0
9	2	sell_avg	facttable	sell	avg	0
10	2	sell_count	facttable	sell	count	0

Abbildung 5.6.: Zusätzliche Informationen von materialisierten Views

Die Tabelle „fitable“ enthält Informationen über die Gruppierungsattribute („group“), Aggregatfunktionen („function“) und Alias-Werte. Dabei wird als Fremdschlüssel wiederum die ViewID der Relation „viewtable“ verwendet. Das Attribut „group“ gibt an, nach welchen Klassifikationsstufen einer View gruppiert wurde. Das heißt, es wird angegeben, welche Klassifikationsstufen in der GROUP BY-Klausel der entsprechenden Aggregatanfrage einer View verwendet wurden. Dabei besagt der Wert „1“, dass nach dem entsprechenden Attribut gruppiert wurde, während „0“ das Gegenteil bedeutet.

Das Attribut „function“ gibt an, welche Aggregatfunktion auf eine Kennzahl angewendet wurde. Das heißt, dass zum Beispiel MIN, MAX, AVG, SUM oder COUNT

angegeben wird. Enthält ein Tupel einen Nullwert, dann ist das entsprechende Attribut eine Klassifikationsstufe der SELECT-Klausel einer Aggregatanfrage.

Durch die Attribute „attribute“, „fromtable“ und „fromtablealias“ können Alias-Werte verwendet werden. Es werden zu jedem Attribut der entsprechenden Relation einer View gespeichert, welchen aktuellen Namen eine Spalte hat („attribute“), aus welcher Relation das entsprechende Attribut selektiert wurde („fromtable“) und welchen Namen das Attribut in der „Ursprungsrelation“ hatte („fromtablealias“).

Restriktionen

Die Restriktionen, welche eine materialisierte View enthält, werden in der Relation „rtable“ („Restriction Table“) gespeichert, wobei eine View wiederum durch deren ViewID identifiziert wird (siehe Abbildung 5.7).

rtable_id	viewid	attribute	lb	ub	connection	not
1	1	calendar.year	NULL	2008	AND	0
2	2	calendar.year	NULL	2008	AND	0
3	2	calendar.month	November	November	AND	1
4	2	calendar.month	Juni	Juni	AND	1

Abbildung 5.7.: Restriktionen von materialisierten Views

Die Spalte „attribute“ enthält den Namen des eingeschränkten Attributs, der entweder ein Wert der Spalte „attribute“ der Relation „fitable“ ist, oder eine Zeichenkette bestehend aus „<Relation>.<Attributname>“. Das Attribut „lb“ steht für „Lower Bound“ (Untere Grenze), während „ub“ die „Upper Bound“ (Obere Grenze) bezeichnet. Des Weiteren geben die Spalten „connection“ (Verbindung) und „not“ an, wie unterschiedliche Restriktionen untereinander verbunden sind. Es ist zu beachten, dass falls Restriktionen bezüglich eines Wertes gemacht werden, dieser Wert jeweils in „ub“ und „lb“ aufgenommen werden muss. Zum Beispiel ist die View AST2 bezüglich der Monate Juni und November eingeschränkt, sodass die Werte von „lb“ und „ub“ beim entsprechenden Eintrag in der Relation „rtable“ gleich sind. Des Weiteren sind die Restriktionen von AST2 mittels AND verbunden und mit NOT versehen. Daher enthält die Spalte „not“ den Wert „1“, während der Wert vom Attribut „connection“ „AND“ ist.

Bei der Restrukturierung einer Anfrage Q ist bezüglich der „connection“ zu beachten, dass falls eine Restriktion in eine restrukturierte Anfrage Q' ergänzt wird, aber keine Verbundbedingungen oder andere Restriktionen in der WHERE-Klausel von Q' enthalten sind, wird die entsprechende Verbindung nicht mit übernommen.

Die vorgestellte Form der Speicherung macht beim Vorhandensein von Klammerungen eine Ausklammerung der vorhandenen Restriktionen erforderlich. Das heißt, dass eine Normalisierung vorgenommen werden muss, damit analog zur EBNF der Abbildung 4.2 die Restriktionen verwaltet werden können.

5.1.4. Kostenfunktion

Zur Realisierung der Kostenfunktion werden verschiedene Relationen benötigt, welche in Abbildung 5.8 dargestellt sind. Es werden dabei die in Abschnitt 4.2 vorgestellten Berechnungsmöglichkeiten von Kostenfunktionswerten realisiert.

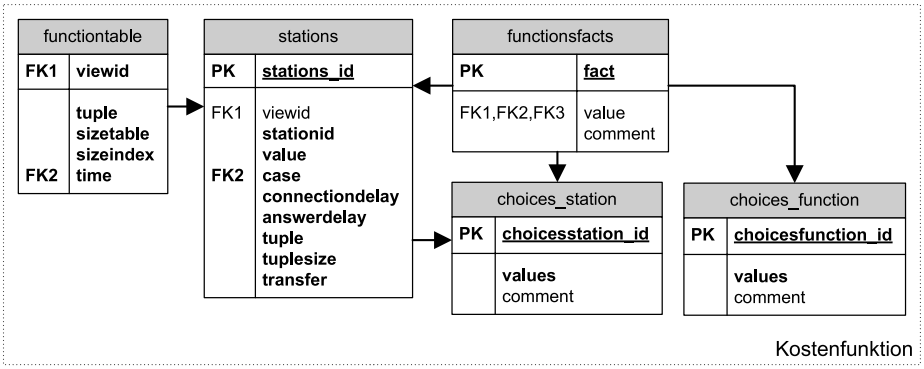

Abbildung 5.8.: Kostenfunktion des Relationenschemas

Die Relation „functiontable“ enthält zu jeder ViewID die verschiedenen Kostenfunktionswerte, welche zum Vergleich von materialisierten Views verwendet werden können. Dazu zählen die Tupelanzahl („tuple“), die benötigte Seitenanzahl („sizetable“, „sizeindex“) und die Zugriffszeit („time“). Abbildung 5.9 illustriert eine mögliche Belegung für die Views AST1 und AST2.

viewid	tuple	sizetable	sizeindex	time
1	15	1	0	0
2	14	1	0	0

Abbildung 5.9.: Beispiel der Relation „functiontable“

Zur Berechnung der Zugriffszeiten im verteilten Szenario werden unterschiedliche Parameter benötigt, welche in der Relation „stations“ gespeichert werden. Mit Hilfe dieser Parameter wird eine Zugriffszeit ermittelt und im Attribut „value“ festgeschrieben. Diese Werte sind dabei als Fremdschlüssel im Attribut „time“ der Relation „functiontable“ enthalten. Wie in Abschnitt 4.2 erläutert wurde, können unterschiedliche Fälle der Zugriffszeiten auftreten. Diese Fälle sind in der Relation „choices_station“ gespeichert (siehe Abbildung 5.10) und im Attribut „case“ der Tabelle „stations“ als Fremdschlüssel enthalten.

choicesstation_id	values	comment
1	worstcase	Use the worst case for time calculation
2	bestcase	Use the best case for time calculation
3	avg	Use the average case for time calculation

Abbildung 5.10.: Fälle der Zugriffszeiten von materialisierten Views

Die Relation „choices_function“ enthält die unterschiedlichen Möglichkeiten, mit denen die Kostenfunktionswerte berechnet werden können. Die Abbildung 5.11 illustriert die in Abschnitt 4.2 vorgestellten Möglichkeiten.

choicesfunction_id	values	comment
1	tuplecount	Use the number of tuples for cost function
2	pagecount	Use the number of pages (table + index) for cost function
3	time	Use calculated time for cost function
4	tuplequotient	Use tuplecount in comparison with facttable for cost function
5	pagequotient	Use pagecount in comparison with facttable for cost function
6	timequotient	Use time in comparison with facttable for cost function

Abbildung 5.11.: Kostenfunktionswerte von materialisierten Views

Es kann somit entschieden werden, ob die Tupelanzahl, Seitenzahl, Zugriffszeit oder der entsprechende Wert im Vergleich zur Faktentabelle als Kostenfunktionswert dienen soll. Dafür muss in der Relation „functionsfacts“ der entsprechende Wert gesetzt werden. Die Abbildung 5.12 stellt die Relation „functionsfacts“ dar.

fact	value	comment
choice_function	1	The choice of table choices_function
choice_station	1	The choice of table choices_station
pagesize	16384	The size of one page in the database
size	2	The number of pages (table + index) of the facttable
time	0	The calculated time of the facttable
tuplecount	24	The number of tuples of the facttable

Abbildung 5.12.: Parameter der Kostenfunktionswerte

Die Tabelle enthält neben den Einstellungen, welche Zugriffsfälle („fact = 'choices_stations'“) und Kostenfunktionswerte („fact = 'choices_function'“) verwendet werden sollen, auch die entsprechenden Kostenfunktionswerte der Faktentabelle. Das heißt, falls die Kostenfunktionswerte der materialisierten Views im Vergleich zur Faktentabelle betrachtet werden, können die entsprechenden Werte der Faktentabelle aus der Relation „functionsfacts“ ausgelesen werden. Um die Zugriffszeiten im verteilten Szenario berechnen zu können, müssen die entsprechenden Parameter ebenfalls für die Faktentabelle vorhanden sein. Diesbezüglich werden die Werte der Faktentabelle in der Relation „stations“ gespeichert, allerdings enthält das Attribut „viewid“ anstatt einer ViewID einen Nullwert.

Viewtable

Die Abbildung 5.13 veranschaulicht die Relation „viewtable“, wobei als Einträge die Views AST1 und AST2 enthalten sind.

Die Werte des Attributs „costfunction“ werden aus der Relation „functiontable“ übernommen, je nachdem welche Möglichkeit ausgewählt wurde. Im Beispiel wird die Tupelanzahl als Kostenfunktionswert verwendet, da in der Relation „functionsfacts“ der

viewid	calendar	customer	costfunction
1	2	1	15
2	0	0	14

Abbildung 5.13.: Beispiel der Relation „viewtable“

entsprechende Wert gesetzt wurde („choices_stations = 1“). Die Kombination der Views wird anhand der Tabelle „dimensions“ ermittelt und abgespeichert.

5.1.5. Wartung

Materialisierte Views werden aktualisiert, falls die zugeordneten Basisrelationen verändert wurden. Wie in Abschnitt 4.4 vorgestellt wurde, werden zur Wartung Datenbanksichten benötigt. Die Abbildung 5.14 illustriert die notwendigen Relationen und Datenbanksichten.

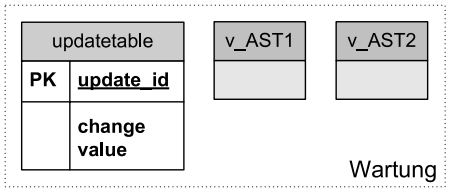

Abbildung 5.14.: Wartung des Relationenschemas

Die notwendigen Datenbanksichten werden erzeugt, indem aus den Aggregatanfragen zuerst Datenbanksichten und danach materialisierte Views erstellt werden. Der Name einer Datenbankview wird gebildet aus der Konkatenation „v_“ und dem entsprechenden Namen der materialisierten View. Das folgende SQL-Statement illustriert dies am Beispiel von AST1:

```
CREATE VIEW `da`.`v_AST1` AS
    SELECT calendar.month, customer.city,
           SUM(facttable.cost) AS cost,
           SUM(facttable.sell) AS sell_sum,
           COUNT(facttable.sell) AS sell_count
    FROM calendar, customer, facttable
    WHERE calendar.day = facttable.day AND
          customer.cust_id = facttable.cust_id AND
          calendar.year < 2008
    GROUP BY calendar.month, customer.city;

CREATE TABLE `da`.`AST1`
    SELECT * FROM v_AST1;
```

Neben den Datenbanksichten wird die Relation „updatetable“ benötigt, welche die auf den Basisrelationen vollzogenen Änderungen vermerkt. Die Abbildung 5.15 stellt die Relation „updatetable“ dar.

update_id	change	value
1	insert	0
2	delete	0
3	update	0

Abbildung 5.15.: Wartung von materialisierten Views

Das Attribut „change“ der Relation „updatetable“ enthält die verschiedenen Änderungen, welche mit Hilfe von Triggern überwacht werden sollen. Dies ist das Einfügen, Löschen und Update von Tupel der Faktentabelle. Falls eine derartige Änderung vorgenommen wird, ändert der entsprechende Trigger das Attribut „value“. Der Wert „1“ bedeutet, dass eine Änderung vorgenommen wurde, während „0“ das Gegenteil ausdrückt.

Die Definition der Trigger und die Wartung der materialisierten Views wird in Abschnitt 5.3 detaillierter behandelt.

5.2. Optimierung einer Anfrage

Es wird unter Verwendung des in Abschnitt 5.1 vorgestellten Relationenschemas nachfolgend eine Aggregatanfrage restrukturiert. Dabei soll unter anderem gezeigt werden, auf welche Art und Weise die vorhandenen materialisierten Views im Bezug auf Abschnitt 4.1 analysiert und in die Menge $PosMV$ einsortiert werden. Die Abbildung 5.16 illustriert den Ablauf einer Optimierung. Im Folgenden sind zur verbesserten Übersicht die wichtigsten Erkenntnisse den entsprechenden Abschnitten vorangestellt (als kursive Aufzählung).

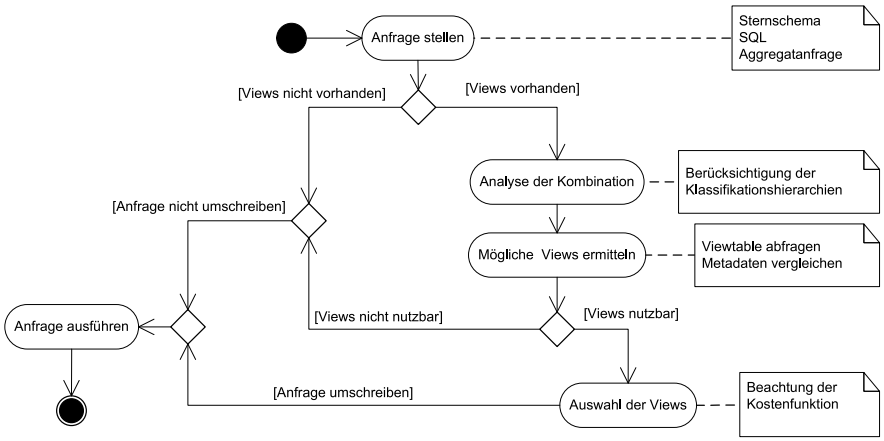

Abbildung 5.16.: Ablauf einer Optimierung

5.2.1. Anfrage stellen

- *Formulierung einer Beispielanfrage*
- *Überprüfung des Vorhandenseins materialisierter Views*

Die folgende in SQL formulierte Aggregatanfrage Q wird an das Starschema von Abschnitt 5.1 gestellt:

```
SELECT calendar.month, customer.city,
       SUM(facttable.cost) AS cost,
       AVG(facttable.sell) AS sell
FROM calendar, customer, facttable
WHERE calendar.day = facttable.day AND
      customer.cust_id = facttable.cust_id AND
      calendar.year < 2008 AND
      NOT (calendar.month = 'November') AND
      NOT (calendar.month = 'Juni')
GROUP BY calendar.month, customer.city;
```

Es wird nach den monatlichen Durchschnittsverkäufen und Kosten der unterschiedlichen Städte gefragt. Dabei sollen allerdings nur die Kennzahlen berücksichtigt werden, die vor dem Jahr 2008, aber nicht in den Monaten Juni oder November, gesammelt wurden.

Nachdem eine Anfrage gestellt wurde, muss überprüft werden, ob materialisierte Views vorhanden sind oder nicht. Dafür könnte zum Beispiel die folgende Anfrage gestellt werden:

```
SELECT COUNT(*) FROM viewtable;
```

Falls der zurückgegebene Wert „0“ ist, dann stehen keine Views zur Verfügung. Somit kann die Anfrage Q nicht restrukturiert und damit direkt ausgeführt werden. Ist der Wert allerdings größer oder gleich „1“, dann sind materialisierte Views vorhanden, die zur Optimierung genutzt werden könnten. Es sind die Views AST1 und AST2 vorhanden, sodass Q potentiell restrukturiert werden kann.

5.2.2. Analyse der Kombination

- *Ermittlung der Kombination der Anfrage durch Zugriff auf „viewtable“*

Nachdem geprüft wurde, ob materialisierte Views vorhanden sind, wird die Kombination der gestellten Anfrage Q analysiert. Dafür wird die Relation „dimensions“ benötigt, in der den Klassifikationsstufen Werte zugeordnet werden. Des Weiteren müssen die Klassifikationsstufen mit den entsprechenden Dimensionen von Q ermittelt werden. Das heißt, dass die FROM-Klausel von Q durchlaufen werden muss und alle genutzten Dimensionstabellen vermerkt werden müssen. Danach muss unter Verwendung der vermerkten Dimensionstabellen die SELECT-Klausel durchlaufen werden, um die selektierten Klassifikationsstufen den entsprechenden Dimensionen zu zuordnen.

Die folgende Anfrage kann danach gestellt werden, die die Kombination einer Anfrage zurückgibt:

```
SELECT dimension, value
FROM dimensions
WHERE (dimension = <Dimension> AND name = <Stufe>)
ORDER BY dimension;
```

Im Beispiel ergibt sich für Q die folgende Anfrage P:

```
SELECT dimension, value
FROM dimensions
WHERE (dimension = 'customer' AND name = 'city') OR
      (dimension = 'calendar' AND name = 'month')
ORDER BY dimension;
```

Die Anfrage P liefert für das Beispiel und die Anfrage Q die Tupel (calendar, 2) und (customer, 1). Dies entspricht im Beispiel der Kombination (2,1). Allerdings ist die Zuordnung der Dimensionen zu den Werten der Stufen erforderlich, da falls eine Anfrage nicht alle Dimensionstabellen des Starschemas verwendet, müssten Nullwerte in die Kombination eingebaut werden. Das heißt, falls zum Beispiel eine Dimension „channel“ (Kanal) existiert, die lexikalisch zwischen „calendar“ und „customer“ liegt, dann müsste obige Kombination (2,NULL,1) lauten. Daher wird auf die verkürzende Schreibweise verzichtet.

5.2.3. Ermittlung möglicher Views

- *Vergleich der möglichen Views mit der Beispielanfrage*
- *Einteilung der Views in TempRest-Mengen*

Um mögliche Views zu ermitteln, werden die unterschiedlichen Metadaten-Relationen des Relationenschemas benötigt. Unter Verwendung dieser Relationen wird die Menge $PosMV$ aufgebaut, die die zur Restrukturierung geeigneten materialisierten Views enthält. Falls beim Aufbau der Menge die Situation eintreten sollte, dass alle entsprechenden Untermengen von $PosMV$ keine Views mehr enthalten, dann sind die vorhandenen materialisierten Views nicht nutzbar. Daraus folgt, dass die gestellte Anfrage nicht umgeschrieben und somit ausgeführt werden kann. Die Untermengen sind $TempDim_x$, $TempKlass_x$, $TempKenn_x$, $TempAgg_x$, $TempGroup_x$ und $TempRest_x$.

Dimensionstabellen und Klassifikationsstufen

- *Übernahme AST1 in $TempKlass_1$ - Dimensionstabellen: gleich, Klassifikationsstufen: gleich*
- *Übernahme AST2 in $TempKlass_2$ - Dimensionstabellen: gleich, Klassifikationsstufen: niedriger*

Es wird zuerst geprüft, welche Views die gleichen oder mehr Dimensionstabellen und die passenden Klassifikationsstufen besitzen (siehe Abschnitt 4.1.2). Dafür wird die Relation „dimensions“ benötigt, um ausgehend von der Kombination der Anfrage Q die möglichen Werte der Hierarchiestufen zu ermitteln. Die folgende Anfrage P liefert alle Werte der Dimensionen, die zur Beantwortung von Q dienen können:

```
SELECT dimension, successor
FROM dimensions
WHERE (value = 1 AND dimension = 'calendar') OR
      (value = 1 AND dimension = 'customer');
```

Es werden im Beispiel die Tupel (calendar, 0) und (customer, 0) zurückgegeben. Das bedeutet, dass falls eine View niedrigere Hierarchiestufen als Q besitzt, dann dürfen diese Stufen bezüglich der Dimensionen „calendar“ und „customer“ jeweils den Wert „0“ haben. Unter Verwendung der Kombination von Q und den möglichen Werten niedrigerer Hierarchiestufen wird die Relation „viewtable“ wie folgt abgefragt:

```
SELECT viewid, calendar, customer
FROM viewtable
WHERE (calendar = 2 OR calendar = 0) AND
      (customer = 1 OR customer = 0);
```

Als Ergebnis werden alle materialisierten Views zurückgegeben, deren Klassifikationsstufen und Dimensionstabellen mit denen von Q vergleichbar sind. Diese Views bilden die Elemente der Menge *TempDim*. Eine explizite Unterscheidung zwischen $TempDim_1$ und $TempDim_2$ ist an dieser Stelle nicht notwendig, da im nächsten Schritt beide Mengen vereinigt werden (siehe Abschnitt 4.1.2). Es muss unterschieden werden, ob die Klassifikationsstufen im Vergleich zu Q passen, oder mindestens eine Stufe kleiner ist. Daher wird mit obiger Anfrage nicht nur die ViewID zurückgegeben, sondern ebenfalls die Dimensionen und deren Hierarchiestufen. Falls mindestens eine Klassifikationsstufe kleiner ist, wobei die anderen vorkommenden Stufen kleiner oder gleich sind, werden die entsprechenden Views in die Menge $TempKlass_2$ eingeordnet. Im Beispiel wäre dies die View AST2. Im Gegensatz dazu werden alle Views, deren Hierarchiestufen mit denen der Anfrage übereinstimmen, in die Menge $TempKlass_1$ aufgenommen. AST1 erfüllt diese Bedingung und wird daher $TempKlass_1$ zugeordnet.

Kennzahlen

- *Übernahme AST1 aus* $TempKlass_1$ *in* $TempKenn_1$ *- Kennzahlen: gleich*
- *Übernahme AST2 in* $TempKlass_2$ *in* $TempKenn_3$ *- Kennzahlen: gleich*

Nach der Analyse der Dimensionstabellen und Hierarchiestufen, werden die Kennzahlen betrachtet. Dafür wird die Relation „measures“ abgefragt. Es werden dabei jeweils nur die materialisierten Views untersucht, die in den *TempKlass*-Mengen vorhanden sind. Es ergibt sich die folgende Anfrage P:

```
SELECT viewid, measure
FROM measures
WHERE viewid = 1 OR viewid = 2;
```

Die Anfrage P liefert alle Kennzahlen, welche in den Views AST1 und AST2 vorhanden sind. Diese Kennzahlen werden mit denen der Anfrage Q verglichen, wobei die Kennzahlen von Q jeweils in den Aggregatfunktionen der SELECT-Klausel enthalten sind. Es muss unterschieden werden, ob einerseits alle bzw. mehr oder andererseits nur ein Teil der Kennzahlen in einer View vorliegen. Im Beispiel enthalten beide vorhandenen Views jeweils alle notwendigen Kennzahlen. Das heißt, analog zu Q sind sowohl „sell“ als auch „cost“ selektiert worden. Somit wird AST1 aus $TempKlass_1$ in $TempKenn_1$ und AST2 aus $TempKlass_2$ in $TempKenn_3$ übernommen.

Aggregatfunktionen

- *Übernahme AST1 aus $TempKenn_1$ in $TempAgg_2$ (SUM(cost) in $TempAgg_1$) - Aggregatfunktionen: Kombination (bzgl. SUM(cost) gleich)*
- *Übernahme AST2 aus $TempKenn_3$ in $TempAgg_6$ (SUM(cost) in $TempAgg_5$) - Aggregatfunktionen: gleich (AVG) (bzgl. SUM(cost) gleich)*

Die Views in den *TempKenn*-Mengen werden untersucht, welche Aggregatfunktionen auf die Kennzahlen angewendet wurden. Dabei wird unterschieden, ob die Funktionen passen oder als Kombination verschiedener Funktionen berechnet werden (siehe Abschnitt 4.1.4). Es wird diesbezüglich die Relation „fitable“ mit folgender Anfrage P abgefragt:

```
SELECT viewid, attribute, fromtable, fromtablealias,
       function, fitable.group
FROM fitable
WHERE viewid = 1 OR viewid = 2;
```

Unter Verwendung der erhaltenen Tupel kann entschieden werden, ob die Aggregatfunktionen von Q mit denen vorhandener Views übereinstimmen. Dafür muss analysiert werden, welche Aggregatfunktionen in Q auf welche Kennzahlen angewendet werden. Im Beispiel sind dies „SUM(cost)“ und „AVG(sell)“. Es werden für jedes Element der *TempKenn*-Mengen jeweils die Tupel von P untersucht, welche untereinander die gleiche ViewID besitzen und als Wert des Attributs „function“ keinen Nullwert enthalten. Diese Tupel beinhalten dabei die Informationen über die in einer View verwendeten Aggregatfunktionen. Danach wird mit Hilfe des Attributs „fromtablealias“ überprüft, welche Kennzahlen von den entsprechenden Funktionen aggregiert wurden.

AST1 enthält im Vergleich zu Q zwar „SUM(cost)“, allerdings nicht „AVG(sell)“. Somit muss AST1 die notwendigen Aggregationen nachvollziehen. Dies ist zum Beispiel möglich, indem „AVG(sell)“ aus der Division von „SUM(sell)“ und „COUNT(sell)“ berechnet wird. AST1 enthält sowohl „SUM(sell)“ als auch „COUNT(sell)“, sodass keine zweite View zur Berechnung von „AVG(sell)“ genutzt werden muss. Gesetzt den Fall, dass AST1 nicht beide aggregierte Kennzahlen enthalten hätte, würde AST1 nicht in

PosMV aufgenommen werden können. Dies ist damit zu begründen, dass aufgrund des Fehlens anderer, kombinierbarer Views nicht alle aggregierten Kennzahlen zur Berechnung vorhanden wären. Im Beispiel wird AST1 allerdings aus $TempKenn_1$ in $TempAgg_2$ übernommen. Dabei muss allerdings vermerkt werden, dass AST1 bezüglich „SUM(cost)“ in die Menge $TempAgg_1$ übernommen worden wäre. Dies ist für eine spätere Restrukturierung notwendig (siehe Abschnitt 5.2.5). Nachfolgend wird daher die Einteilung der entsprechenden View im Bezug auf „SUM(cost)“ ebenfalls thematisiert.

AST2 enthält sowohl „SUM(cost)“ als auch „AVG(sell)“, sodass die Aggregatfunktionen im Vergleich zu Q passen. Allerdings besitzt AST2 zwei niedrigere Klassifikationsstufen, sodass eine nachträgliche Aggregation notwendig ist. Wie im Vorfeld gezeigt wurde, werden diesbezüglich aufgrund der Sonderstellung von AVG nicht nur „AVG(sell)“, sondern auch „COUNT(sell)“ benötigt. Wiederum enthält AST2 beide notwendigen aggregierten Kennzahlen, sodass keine zweite View zur Bildung einer Kombination gesucht werden muss. AST2 wird im Beispiel aus $TempKenn_3$ in $TempAgg_6$ übernommen. Analog zu AST1 muss wiederum vermerkt werden, dass AST2 im Bezug auf „SUM(cost)“ in die Menge $TempAgg_5$ aufgenommen worden wäre.

Gruppierungsattribute

- *Übernahme AST1 aus $TempAgg_2$ in $TempGroup_2$ ($SUM(cost)$ in $TempGroup_1$) - Gruppierungsattribute: gleich*
- *Übernahme AST2 aus $TempAgg_6$ in $TempGroup_6$ ($SUM(cost)$ in $TempGroup_5$) - Gruppierungsattribute: gleiche Dimensionen*

Die Tupel der obigen Anfrage P können nicht nur zur Analyse der Aggregatfunktionen verwendet werden, sondern auch für die Gruppierungsattribute der GROUP BY-Klausel. Dafür wird das Attribut „group“ der Relation „fitable“ benötigt. Es muss im Bezug auf die GROUP BY-Klausel der Anfrage Q geprüft werden, ob in den Elementen der *TempAgg*-Mengen die gleichen Gruppierungsattribute verwendet wurden oder nicht. Falls niedrigere Klassifikationsstufen in einer View vorliegen, dann müssen die Stufen in der GROUP BY-Klausel der View vorhanden sein, die aggregiert die höheren Stufen und somit die Gruppierungsattribute von Q ergeben. Das heißt, dass anstatt „month“ und „city“ zum Beispiel „day“ und „cust_id“ vorliegen müssten. In der View AST2 ist dies genau der Fall, was durch den Wert „1“ im Attribut „group“ angezeigt wird. Daher wird AST2 aus $TempAgg_6$ in $TempGroup_6$ übernommen (bezüglich „SUM(cost)“ aus $TempAgg_5$ in $TempGroup_5$).

Die materialisierte View AST1 enthält dieselben Klassifikationsstufen wie die Anfrage Q. Somit müssen die Gruppierungsattribute von AST1 und Q übereinstimmen, was auch der Fall ist. Deswegen wird AST1 aus $TempAgg_2$ in $TempGroup_2$ aufgenommen (bezüglich „SUM(cost)“ aus $TempAgg_1$ in $TempGroup_1$).

Restriktionen

- *Übernahme AST1 aus $TempGroup_2$ in $TempRest_4$ (SUM(cost) in $TempRest_2$) - Restriktionen: weniger streng*
- *Übernahme AST2 aus $TempGroup_6$ in $TempRest_{11}$(SUM(cost) in $TempGroup_9$) - Restriktionen: gleich*

Zum Aufbau der Menge *PosMV* müssen die Restriktionen der Anfrage Q mit denen der Elemente aus den *TempRest*-Mengen verglichen werden. Es wird diesbezüglich unterschieden, ob Q strengere oder die gleichen Restriktionen im Vergleich zu den Views besitzt. Die Restriktionen der materialisierten Views sind in der Relation „rtable“ gespeichert und können mit folgender Anfrage P erhalten werden:

```
SELECT viewid, attribute, lb, ub,
       connection, rtable.not
FROM rtable
WHERE viewid = 1 OR viewid = 2;
```

Es können im Attribut „attribute“ von Anfrage P zwei unterschiedliche Wertbelegungen vorkommen. Einerseits können die dort gespeicherten Werte Einträge der „fitable“ enthalten, andererseits nicht in den Views selektierte, aber eingeschränkte Hierarchiestufen. Im ersten Fall sind die Werte die Namen, die im Attribut „fromtablealias“ der Relation „fitable“ vorkommen. Im zweiten Fall bestehen die Einträge aus einer Konkatenation aus <Dimension> und <Hierarchiestufe>, getrennt durch einen Punkt. Im Beispiel entsprechen die Restriktionen der materialisierten Views dem zweiten Fall.

Die Restriktionen der Anfrage Q schränken die Hierarchiestufen „year“ und „month“ der Dimension „calendar“ ein. Dabei werden die Einschränkungen bezüglich des Wertebereichs der entsprechenden Stufen vorgenommen. Die Hierarchiestufe „year“ wird mit Hilfe einer oberen Grenze („year $<$ 2008“) eingeschränkt. Das heißt, dass für die vorhandenen Views geprüft werden muss, ob die obige Anfrage P ein Tupel enthält, bei dem „ub“ gleich „2008“ ist. Dabei muss der Wert des Attributs der unteren Grenze („lb“) ein Nullwert sein, da ansonsten nicht alle notwendigen Daten zur Beantwortung von Q in der View enthalten wären. Des Weiteren ist es nicht möglich, dass eine View einen größeren Wert als „2008“ in „ub“ umfasst, zum Beispiel „2009“. Das würde heißen, dass Q eine strengere Restriktion besitzt. Solche Anfrage können nur dann mit Hilfe von materialisierten Views beantwortet werden, falls die entsprechende Restriktion von Q die selektierten Klassifikationsstufe einer View einschränkt (siehe Abschnitt 4.1.6). Dies ist im Beispiel aber nicht der Fall, da die Restriktion von Q im Vergleich zu den Views die nicht selektierte Klassifikationsstufe „year“ beinhaltet. Sowohl AST1 als auch AST2 besitzen nach obiger Anfrage P ein Tupel, bei dem „ub“ gleich „2008“ ist. Des Weiteren sind die Werte des Attributs „connection“ gleich, sodass die Restriktionen der vorhandenen Views der von Q entsprechen.

Die Anfrage Q enthält des Weiteren Restriktionen bezüglich der Hierarchiestufe „month“. Da AST1 keine weiteren Restriktionen umfasst, wird diese View aus der Menge $TempGroup_2$ in die Menge $TempRest_4$ übernommen (bezüglich „SUM(cost)“

aus $TempGroup_1$ in $TempRest_2$). Dies ist allerdings nur möglich, da die nicht vorhandenen Restriktionen von Q Klassifikationsstufen betreffen, welche durch AST1 selektiert wurden. Das heißt, dass eine nachträgliche Einschränkung der Tupel von AST1 ohne Zugriff auf die Faktentabelle möglich ist. Gesetzt den Fall die Einschränkungen von Q hätten wiederum die Hierarchiestufe „year“ eingeschränkt, dann wäre AST1 nicht in die Menge $PosMV$ eingegangen.

Die Restriktionen von Q besagen, dass zwei festgelegte Monate nicht in die Ergebnismenge eingehen sollen. Das heißt, dass geprüft werden muss, ob eine View zwei Restriktionen enthält, die sowohl in „lb“ als auch „ub“ denselben Wert aufweisen. Diese Werte müssen dabei genau denen der Anfrage entsprechen, also einerseits „Juni“ anderseits „November“ sein. Des Weiteren müssen die Restriktionen im Attribut „not“ den Wert „1“ beinhalten und bezüglich des Attributs „connection“ gleich sein. Für die View AST2 gelten diese Bedingungen.

Es wäre auch denkbar, dass AST2 die besagten Restriktionen nicht enthält bzw. andere Restriktionen besitzt. Im ersten Fall wäre die Anfrage strenger als die View, könnte aber dennoch verwendet werden, obwohl die Restriktionen nicht selektierte Klassifikationsstufen betrifft. Dies ist damit zu begründen, dass AST2 die niedrigsten Klassifikationsstufen selektiert hat und durch die notwendigen nachträglichen Aggregationen die entsprechenden Restriktionen von Q nachvollziehen kann. Würde die von der Einschränkung betroffene Dimension nicht die niedrigste Stufe enthalten, wäre dies ohne Zugriff auf die Faktentabelle nicht möglich. Falls AST2 andere Restriktionen beinhalten würde, dann wäre die Anfrage weniger streng und somit AST2 nicht zur Beantwortung von Q geeignet (siehe Abschnitt 4.1.6). Andere Restriktionen würden auch dann vorliegen, falls zum Beispiel die Attribute „connection“ oder „not“ nicht mit der Anfrage übereinstimmen würden.

Im Beispiel entsprechen die Restriktionen von Q denen von AST2, sodass diese View von $TempGroup_6$ in die Menge $TempRest_{11}$ übernommen wird (bezüglich „SUM(cost)“ aus $TempGroup_5$ in $TempRest_9$).

5.2.4. Auswahl der materialisierten Views

- *Anwendung der Kostenfunktion zum Vergleich der Views*

Nach dem Durchlauf des Abschnitt 5.2.3 besteht $PosMV$ aus den Mengen $TempRest_4$ (AST1) und $TempRest_{11}$ (AST2). Somit kann die Anfrage Q unter Verwendung verschiedener materialisierter Views restrukturiert werden. Daher wird unter Verwendung der Kostenfunktionswerte entschieden, welche View genutzt wird. Die folgende Abfrage ermittelt unter Anwendung der Relation „viewtable“ die kostengünstigste View aus der Menge $PosMV$:

```
SELECT viewid
FROM viewtable
WHERE viewid = 1 OR viewid = 2
ORDER BY costfunction;
```

In Beispiel ist AST2 die kostengünstigere View. Dies ist damit zu begründen, dass in der Relation „functionsfacts“ die Tupelanzahl als Kostenfunktionswert ausgewählt wurde. Das heißt, dass das Tupel mit dem Eintrag „choices_function“ als Wert des Attributs „value“ „1“ enthält. Da AST1 15 Tupel und AST2 14 Tupel beinhalten, wird AST2 zur Restrukturierung genutzt werden.

5.2.5. Restrukturierung der Anfrage

- *Umschreiben der Beispielanfrage mittels Views*

Die Anfrage Q kann mit Hilfe der materialisierten View AST2 restrukturiert werden. Es wird daher im Folgenden erläutert, wie aus Q die restrukturierte Anfrage Q_2 erzeugt wird. Obwohl die Anfrage AST1 aufgrund des höheren Kostenfunktionswertes nicht verwendet werden würde, soll dennoch nachfolgend auch gezeigt werden, wie die restrukturierte Anfrage Q_1 mit Hilfe von Q und AST1 konstruiert wird. Auf welche Art und Weise die Restrukturierung ablaufen wird, wurde konzeptionell im Abschnitt 4.3 beschrieben.

Dimensionstabellen und Views

- *Aufbau der FROM-Klausel*
- *Aufbau der WHERE-Klausel - Übernahme von Verbundbedingungen*

Es muss ermittelt werden, welche Views und Dimensionstabellen in eine restrukturierte Anfrage Q' übernommen werden müssen. Durch die Einordnung in die *TempRest*-Mengen ergibt sich für die Views, dass sowohl für AST1 als auch AST2 eine oder mehrere Sichten notwendig sind. Da allerdings keine der vorhandenen *TempRest*-Mengen Kombinationen verschiedener Views enthält, wird jeweils die entsprechende View zur FROM-Klausel von Q_1 und Q_2 hinzugefügt. Wie in Abschnitt 5.2.3 gezeigt wurde, umfassen die Views jeweils alle notwendigen Aggregatfunktionen und Kennzahlen, sodass keine weiteren, ergänzenden Views benötigt wurden.

Für AST2 gilt, dass aufgrund nachträglich erforderlicher Aggregationen die Dimensionstabellen aller niedrigeren Klassifikationsstufen benötigt werden. Da sowohl „day“ als auch „cust_id“ niedriger als die von Q abgefragten Hierarchiestufen sind, müssen die Dimensionstabellen „calendar“ und „customer“ in Q_2 aufgenommen werden. Dies bedeutet aber gleichzeitig, dass die jeweiligen Verbundbedingungen zwischen AST2 und den Dimensionstabellen in die WHERE-Klausel von Q_2 eingefügt werden müssen. Es ergeben sich zu diesem Zeitpunkt die folgenden restrukturierten Anfragen:

```
SELECT
FROM ast1
WHERE
GROUP BY ;
```

```
SELECT
FROM ast2, calendar, customer
WHERE calendar.day = ast2.day AND
      customer.cust_id = ast2.cust_id
GROUP BY ;
```

Gruppierungsattribute

- *Aufbau der SELECT-Klausel - Übernahme von Klassifikationsstufen*
- *Aufbau der GROUP BY-Klausel - Übernahme von Gruppierungsattributen*

Im Bezug auf die Gruppierungsattribute muss entschieden werden, aus welcher Relation die entsprechenden Daten erhalten werden können. Falls eine nachträgliche Aggregation aufgrund passender Klassifikationsstufen nicht notwendig ist, wird für jede Klassifikationsstufe von Q in der SELECT-Klausel von Q' ein Eintrag der Form „<View>.<Stufe>“ getätigt. Passen die Klassifikationsstufen allerdings nicht, muss für jede niedrigere Hierarchiestufe ein Eintrag der Form „<Dimension>.<Stufe>“ erfolgen, während wiederum für die passenden Stufen „<View>.<Stufe>“ hinzugefügt wird. Die Gruppierungsattribute können danach, unter Beachtung der Anfrage Q, ebenfalls in die GROUP BY-Klausel aufgenommen werden. Es ergibt sich für Q_1 und Q_2 folgender Aufbau:

```
SELECT ast1.month, ast1.city
FROM ast1
WHERE
GROUP BY ast1.month, ast1.city;

SELECT calendar.month, customer.city
FROM ast2, calendar, customer
WHERE calendar.day = ast2.day AND
      customer.cust_id = ast2.cust_id
GROUP BY calendar.month, customer.city;
```

Es muss bezüglich der Verwendung von Alias-Werten beachtet werden, dass eventuell Umbenennungen vorgenommen werden müssen. Zum Beispiel könnte ein Attribut in einer View nicht wie von Q gewollt „month“ sondern „Monat“ heißen. Das heißt, dass unter Verwendung der Relation „fitable“ der entsprechende Attributname ermittelt werden muss. Angenommen „month“ wäre durch eine Anfrage Q gefordert, „Monat“ allerdings gespeichert. Somit würde in der Relation „fitable“ für die View ASTx folgendes Tupel enthalten sein: (...,'x','Monat',<Dimension>,'month',...). Für die restrukturierte Anfrage würde sich somit die SELECT-Klausel wie folgt ändern: „SELECT ASTx.Monat as month ...“.

Die Relation „fitable“ könnte im Bezug auf die Alias-Werte der Views AST1 und AST2 mit folgender Anfrage P abgefragt werden:

```
SELECT viewid, attribute, fromtable,
       fromtablealias, function
FROM fitable
WHERE attribute <> fromtablealias AND
      (viewid = 1 OR viewid = 2);
```

Aggregatfunktionen

- *Ergänzung der SELECT-Klausel - Übernahme von Kennzahlen und Funktionen*

Die Zugehörigkeit der vorhandenen Views zu den *TempRest*-Mengen hat ebenfalls Einfluss auf die Struktur der Aggregatfunktionen, welche in eine restrukturierte Anfrage Q' übernommen werden. Wie im Abschnitt 5.2.3 dargestellt wurde, muss im Bezug auf die aggregierten Kennzahlen unterschieden werden, ob „SUM(cost)“ oder „AVG(sell)“ der Anfrage Q betrachtet wird.

Im Zusammenhang mit „SUM(cost)“ wurde vermerkt, dass eine Einordnung von AST1 in $TempRest_2$ und von AST2 in $TempRest_9$ hätte erfolgen müssen. Somit ist die Erscheinungsform der Aggregatfunktion „SUM(cost)“ im Bezug auf Abbildung 4.5 in Q_1 eine „Kennzahl“, während in Q_2 die Form „Funktion(Kennzahl)“ aufgenommen wird. Unter Beachtung möglicher Alias-Werte, welche mit obiger Anfrage P ermittelt werden können, ergeben sich die vorläufigen restrukturierten Anfragen Q_1 und Q_2:

```
SELECT ast1.month, ast1.city,
       ast1.cost AS cost
FROM ast1
WHERE
GROUP BY ast1.month, ast1.city;

SELECT calendar.month, customer.city,
       SUM(ast2.cost) AS cost
FROM ast2, calendar, customer
WHERE calendar.day = ast2.day AND
      customer.cust_id = ast2.cust_id
GROUP BY calendar.month, customer.city;
```

Neben den Alias-Werten, die in einer View verwendet wurden, kann auch die Anfrage Q solche Umbenennungen enthalten. Zum Beispiel soll „SUM(cost)“ den Attributnamen „cost“ erhalten. Daher wurde diese Umbenennung („as cost“) zu den entsprechenden aggregierten Kennzahlen hinzugefügt.

Zusätzlich zu „SUM(cost)“ wird durch Q „AVG(sell)“ selektiert. Bezüglich dieser Funktion wurde AST1 in $TempRest_4$ und AST2 in $TempRest_{11}$ eingeordnet. Das bedeutet im Bezug auf Abbildung 4.5, dass in Q_1 diese aggregierte Kennzahl die Erscheinungsform „Kennzahl <op> Kennzahl“ hat. Da der Durchschnitt berechnet werden soll, ist die Operation die Division. Des Weiteren wurden Alias-Werte in AST1 verwendet, wobei „SUM(sell)“ zu „sell_sum“ und „SUM(count)“ zu „sell_count“ umbenannt wurde. Dies könnte wiederum mit Hilfe der obigen Anfrage P abgefragt werden.

In Q_2 wird „AVG(sell)“ mit Hilfe einer Subquery berechnet, welche im Abschnitt 4.3.2 dargestellt wird. Wiederum muss beachtet werden, welche Alias-Werte in AST2 und in Q verwendet wurden. Die Anfragen Q_1 und Q_2 können wie folgt erweitert werden:

```
SELECT ast1.month, ast1.city,
       ast1.cost AS cost,
       ast1.sell_sum / ast1.sell_count AS sell
FROM ast1
WHERE
GROUP BY ast1.month, ast1.city;

SELECT calendar.month, customer.city,
       SUM(ast2.cost) AS cost,
       SUM(ast2.sell_avg * ast2.sell_count)/
       (SELECT SUM(ast2_1.sell_count)
        FROM calendar cal1, customer cus1, ast2 ast2_1
        WHERE cal1.day = ast2_1.day AND
              cus1.cust_id = ast2_1.cust_id AND
              cal1.month = calendar.month AND
              cus1.city=customer.city
        GROUP BY cal1.month, cus1.city) AS sell
FROM ast2, calendar, customer
WHERE calendar.day = ast2.day AND
      customer.cust_id = ast2.cust_id
GROUP BY calendar.month, customer.city;
```

Restriktionen

- *Ergänzung der WHERE-Klausel - Übernahme von Restriktionen*

Als letzten Schritt müssen die entsprechenden Restriktionen behandelt werden. Es ist möglich, dass die Anfrage Q im Vergleich zu einer View entweder gleich strenge oder strengere Restriktionen enthält. Im ersten Fall sind die entsprechenden Einschränkungen schon auf die Ergebnismenge der materialisierten View angewendet worden, sodass die Einschränkungen nicht erneut ausgeführt werden müssen. Das heißt, dass die restrukturierte Anfrage Q_2 nicht weiter ergänzt werden muss, da AST2 die gleichen Restriktionen wie Q besitzt ($TempRest_4$). Die Anfrage Q ist allerdings strenger als die View AST1 ($TempRest_{11}$), sodass die noch fehlenden Restriktionen auf die Ergebnismenge von Q_1 angewendet werden müssen. Diese Restriktionen waren „NOT (calendar.month = 'November'“ und „NOT (calendar.month = 'Juni')“. Wie bereits gezeigt wurde, können diese Restriktionen auf die View AST1 angewendet werden. Dies wurde damit begründet, dass die Einschränkungen selektierte Klassifikationsknoten von AST1 betreffen und somit nachvollzogen werden können. Die restrukturierten Anfragen Q_1 und Q_2 lauten daher abschließend wie folgt:

```
SELECT ast1.month, ast1.city,
       ast1.cost AS cost,
       ast1.sell_sum / ast1.sell_count AS sell
FROM ast1
WHERE NOT (ast1.month = 'November') AND
      NOT (ast1.month = 'Juni')
GROUP BY ast1.month, ast1.city;

SELECT calendar.month, customer.city,
       SUM(ast2.cost) AS cost,
       SUM(ast2.sell_avg * ast2.sell_count)/
        (SELECT SUM(ast2_1.sell_count)
         FROM calendar cal1, customer cus1, ast2 ast2_1
         WHERE cal1.day = ast2_1.day AND
               cus1.cust_id = ast2_1.cust_id AND
               cal1.month = calendar.month AND
               cus1.city=customer.city
         GROUP BY cal1.month, cus1.city) AS sell
FROM ast2, calendar, customer
WHERE calendar.day = ast2.day AND
      customer.cust_id = ast2.cust_id
GROUP BY calendar.month, customer.city;
```

Die Aggregatanfrage Q wurde unter Verwendung der materialisierten Views AST1 und AST2 in die restrukturierten Anfragen Q_1 und Q_2 umgeschrieben. Aufgrund des geringeren Kostenfunktionswertes würde im Beispiel allerdings nur die Aggregatanfrage Q_2 restrukturiert und ausgeführt werden, wodurch der in Abbildung 5.16 dargestellte Ablauf einer Optimierung abgeschlossen ist.

5.3. Wartung materialisierter Views

In Abschnitt 4.4 wurde dargestellt, auf welche Art und Weise die materialisierten Views gewartet werden sollen. Es wurde ein auf Triggern basierendes Konzept entwickelt, welches eine „Snapshot Aktualisierung“ ermöglicht. Des Weiteren wurde beschrieben, bei welchen Änderungen der Basisrelationen welche Operationen auf den materialisierten Views durchgeführt werden. In Abschnitt 5.1.5 wurden darüber hinaus die Relation „updatetable“ und die Möglichkeit der Verwendung von Datenbankviews vorgestellt.

Nachfolgend soll anhand obiger materialisierter Views und deren Datenbankviews v_AST1 und v_AST2 erläutert werden, wie die entsprechenden Änderungen der Basisrelationen erkannt, propagiert und auf AST1 bzw. AST2 angewendet werden können.

5.3.1. Definition der Trigger

Es werden Trigger benötigt, welche das Einfügen, Löschen und Update von Tupeln der Faktentabelle registrieren und vermerken, sodass in einer Ruhephase eines Data

Warehouses die materialisierten Views gegebenenfalls verändert werden können. Die folgenden Trigger werden definiert:

```
CREATE TRIGGER insert_FT
 AFTER INSERT ON facttable FOR EACH ROW
 UPDATE updatetable SET value = 1
 WHERE updatetable.change = 'insert';

CREATE TRIGGER update_FT
 AFTER UPDATE ON facttable FOR EACH ROW
 UPDATE updatetable SET value = 1
 WHERE updatetable.change = 'update';

CREATE TRIGGER delete_FT
 AFTER DELETE ON facttable FOR EACH ROW
 UPDATE updatetable SET value = 1
 WHERE updatetable.change = 'delete';
```

Es werden somit drei Trigger erzeugt, welche bei Änderungen der angegebenen Basisrelation die entsprechenden Werte der Relation „updatetable“ modifizieren. Zum Beispiel wird durch den Trigger „insert_FT“, nach dem Einfügen eines Tupels in die Faktentabelle, der Attributwert von „value“ im Tupel mit „change = 'insert'“ auf „1“ gesetzt.

5.3.2. Aktualisierung der materialisierten Views

Die Aktualisierungen sind abhängig von den Klassifikationsstufen und Änderungen, welche auf den Basisrelationen vollzogen wurden. Wie in Abschnitt 4.4.1 erklärt wurde, wird eine materialisierte View entweder durch eine Modifikation der vorhandenen Tupel oder durch das komplette Löschen und neue Einfügen aller Tupel aus der entsprechenden Datenbankview aktualisiert.

Modifikation vorhandener Tupel

Die Modifikation der Tupel erfordert, dass die materialisierten Views nur die niedrigsten Klassifikationsstufen selektiert haben. Des Weiteren dürfen nur exklusiv Lösch-, Einfüge- oder Updateoperationen auf der Faktentabelle durchgeführt worden sein, sodass die Relation „updatetable“ nur einmal den Wert „1“ im Attribut „value“ enthält. Die Bedingungen können wie folgt abgefragt werden:

```
SELECT viewid
FROM viewtable
WHERE calendar = 0 AND customer = 0;

SELECT COUNT(*)
FROM updatetable
WHERE value = 1;
```

Die erste Anfrage liefert alle materialisierten Views (im Beispiel AST2), bei denen nur die niedrigsten Hierarchiestufen selektiert wurden, während die zweite Anfrage die Exklusivität von Aktualisierungen prüft. Wird ein höherer Wert als „1“ zurückgegeben, wurden unterschiedliche Operationen außerhalb der Ruhezeit des Data Warehouses durchgeführt. Angenommen, es wurden ausschließlich Lösch-, Einfüge- oder Updateoperationen auf der Faktentabelle umgesetzt, dann können die nachfolgenden Aktualisierungen entsprechend der vorliegenden Operation realisiert werden. Es wird dabei die View AST2 unter Verwendung von v_AST2 aktualisiert:

```
UPDATE ast2, v_ast2
SET ast2.cost = v_ast2.cost,
    ast2.sell_avg = v_ast2.sell_avg,
    ast2.sell_count = v_ast2.sell_count
WHERE ast2.day = v_ast2.day AND
      ast2.cust_id = v_ast2.cust_id;

INSERT INTO ast2
SELECT v_ast2.day, v_ast2.cust_id, v_ast2.cost,
       v_ast2.sell_avg, v_ast2.sell_count
FROM v_ast2
WHERE NOT EXISTS (
 SELECT *
 FROM ast2
 WHERE ast2.day=v_ast2.day AND
       ast2.cust_id=v_ast2.cust_id);

DELETE FROM ast2
USING v_ast2, ast2
WHERE NOT EXISTS (
 SELECT *
 FROM v_ast2
 WHERE ast2.day = v_ast2.day AND
       ast2.cust_id = v_ast2.cust_id);
```

Anhand der verwendeten Verbundbedingungen zwischen der materialisierten View AST2 und deren Datenbankview v_AST2 wird deutlich, warum nur Views mit den niedrigsten Klassifikationsstufen auf obige Weise aktualisiert werden können. Die einzelnen Tupel werden mit Hilfe der Fremdschlüssel der Dimensionstabellen identifiziert, sodass eindeutig erkannt wird, welche Tupel jeweils gelöscht, aktualisiert oder eingefügt wurden.

Kopieren der Datenbankview

Die andere Möglichkeit eine materialisierte View zu aktualisieren ist, deren Tupel zu löschen und unter Verwendung der zugehörigen Datenbankview neu einzufügen. Dieses Verfahren wird angewendet, falls mindestens eine Klassifikationsstufe einer View nicht

die niedrigste ist, oder die Exklusivität der Änderungsoperationen nicht gegeben ist. Die obigen Anfragen können wie folgt abgeändert werden:

```
SELECT viewid
FROM viewtable
WHERE calendar <> 0 OR customer <> 0;

SELECT COUNT(*)
FROM updatetable
WHERE value = 1;
```

Die erste Anfrage liefert alle materialisierten Views, die mindestens eine höhere Klassifikationsstufe besitzen (im Beispiel AST1), während die zweite prüft, ob unterschiedliche Aktualisierungen durchgeführt wurden. Dabei ist zu beachten, dass falls die zweite Anfrage einen Wert größer als „1“ ausgibt, auch Views mit nur den niedrigsten Hierarchiestufen erst geleert und dann neu gefüllt werden. Das heißt, dass zum Beispiel in AST2 ebenfalls alle Tupel gelöscht und aus der Datenbankview v_AST2 neu eingefügt werden würden. Für die View AST1 und deren Datenbankview v_AST1 ergibt sich die folgende Aktualisierung:

```
DELETE FROM ast1;

INSERT INTO ast1
SELECT *
FROM v_ast1;
```

Es werden, wie oben beschrieben, alle Tupel der Relation AST1 gelöscht, bevor die Tupel der Datenbankview v_AST1 erneut eingefügt werden.

5.3.3. Aktualisierung der Kostenfunktionswerte

Es werden verschiedene Informationen über die materialisierten Views und die Faktentabelle gespeichert, um mit diesen die Kostenfunktion zu realisieren. Dazu zählen unter anderem die Tupelanzahl, die Anzahl benötigter Datenbank- und Indexseiten, sowie die Zugriffszeiten im verteilten Szenario. Diese Informationen müssen bei der Aktualisierung der Basisrelationen ebenfalls geprüft und aktualisiert werden.

Die in der Faktentabelle gespeicherten Werte können mit folgender Anfrage aktualisiert werden:

```
UPDATE functionsfacts ff, functionsfacts ff1,
       functionsfacts ff2, information_schema.tables it
SET ff1.value = it.table_rows,
    ff.value = (it.data_length + it.index_length)/ff2.value
WHERE it.table_schema = 'da' AND
      it.table_name = 'facttable' AND
      ff.fact = 'size' AND
      ff1.fact = 'tuplecount' AND
      ff2.fact = 'pagesize';
```

Zur Aktualisierung wird die Relation „tables“ des Schemas „information_schema“ verwendet, da in dieser Tabelle alle notwendigen Informationen enthalten sind. Dazu zählen zum Beispiel die Tupleanzahl („table_rows“) und die Anzahl der Datenbankseiten („data_length“) und Indexseiten („index_length“). Die Anzahl der entsprechenden Seiten wird allerdings in Bytes angegeben, sodass dieser Wert durch die Seitengröße geteilt werden muss. Die Seitengröße ist dabei in der Relation „functionsfacts“ im Tupel mit „fact = 'pagesize'“ gespeichert.

Die Kostenfunktionswerte der unterschiedlichen materialisierten Views können ebenfalls mit Hilfe der Relation „tables“ aktualisiert werden. Am Beispiel der View AST1 ergibt sich folgende Updateoperation:

```
UPDATE functionsfacts ff, functiontable ft,
       information_schema.tables it
SET ft.tuple = it.table_rows,
    ft.sizetable = it.data_length / ff.value,
    ft.sizeindex = it.index_length / ff.value
WHERE it.table_schema = 'da' AND
      ff.fact = 'pagesize' AND
      it.table_name = 'ast1' AND
      ft.viewid = 1;
```

Im verteilten Szenario können analog zur obigen Anfrage die Anzahl der Tupel und die durchschnittliche Tupelgröße in der Relation „stations“ verändert werden. Dabei müssen unter Angabe der entsprechenden „stationid“ und ViewID die Attribute „tuple“ und „tuplesize“ aktualisiert werden. Die Informationen für die durchschnittliche Tupelgröße sind im Attribut „avg_row_length“ gespeichert. Die Parameter „connectiondelay“, „answerdelay“ und „transfer“ müssen allerdings bei Bedarf durch den Anwender verändert werden. Die Ermittlung dieser Werte wird demnach nicht thematisiert. Nach Angabe dieser Parameter kann die Zugriffszeit der unterschiedlichen Zugriffsfälle (siehe Abschnitt 4.2.1) ermittelt werden durch: „value“ = „connectiondelay“ + „answerdelay“ + („tuple“ * „tuplesize“) / „transfer“. Unter Berücksichtigung des ausgewählten Zugriffsfalls kann der entsprechende Wert einer materialisierten View in die Relation „functiontable“ ins Attribut „time“ übernommen werden. Welcher Fall ausgewählt werden soll, ist in der Relation „functionsfacts“ im Tupel mit „fact = 'choices_station'“ gespeichert.

Übernahme der Kostenfunktionswerte

Nachdem die unterschiedlichen Werte einer View in der Relation „functiontable“ aktualisiert wurden, muss die „viewtable“ ebenfalls angepasst werden. Das heißt, dass den unterschiedlichen materialisierten Views ein Kostenfunktionswert zugeordnet werden muss (siehe Abschnitt 4.2.2). Welcher Wert übernommen werden soll, ist in der Relation „functionsfacts“ im Tupel mit „fact = 'choices_function'“ gespeichert. Es ergeben sich die folgenden Updateoperationen:

```
UPDATE functiontable ft, viewtable vt
SET vt.costfunction = <Berechnung>
WHERE ft.viewid = vt.viewid;

UPDATE functiontable ft, viewtable vt,
       functionsfacts ff
SET vt.costfunction = <Berechnung>
WHERE ft.viewid = vt.viewid AND
      <Bedingung>;
```

Die erste Anfrage P_1 kann verwendet werden, falls die Werte der Faktentabelle nicht in die Berechnung der Kostenfunktion einfließen sollen. Das heißt, dass die Tupelanzahl, die Anzahl der Datenbank- und Indexseiten oder die Zugriffszeit einer View direkt in die Relation „viewtable“ übernommen wird. Dies entspricht den Werten „1“, „2“ oder „3“ von „choices_function“. Falls allerdings die Kostenfunktionswerte im Verhältnis zur Faktentabelle berechnet werden sollen, kann die zweite Anfrage P_2 verwendet werden. Dafür muss „choices_function“ die Werte „4“, „5“ oder „6“ beinhalten.

Die Abbildung 5.17 illustriert, welche Berechnungen und Bedingungen in Abhängigkeit der unterschiedlichen Möglichkeiten in die Anfragen P_1 und P_2 ergänzt werden müssen:

Choice_function	< Berechnung >	< Bedingung >
1	ft.tuple	-
2	ft.sizetable + ft.sizeindex	-
3	ft.time	-
4	ft.tuple / ff.value	ff.fact = 'tuplecount'
5	(ft.sizetable + ft.sizeindex) / ff.value	ff.fact = 'size'
6	ft.time / ff.value	ff.fact = 'time'

Abbildung 5.17.: Aktualisierung der Kostenfunktionswerte

Die Abbildung besagt, dass falls „Choices_function“ den Wert „1“ besitzt, die Berechnung „ft.tuple“ in P_1 ergänzt werden muss. Es wird somit die Tupelanzahl ohne Berücksichtigung der Faktentabelle als Kostenfunktionswert berechnet. Eine zusätzliche Bedingung ist dabei nicht notwendig. Es ergibt sich somit folgende Anfrage:

```
UPDATE functiontable ft, viewtable vt
SET vt.costfunction = ft.tuple
WHERE ft.viewid = vt.viewid;
```

Nachdem alle Aktualisierungen durchgeführt wurden, müssen die Werte des Attributs „value“ der Relation „updatetable“ auf „0“ zurückgesetzt werden. Das heißt, dass nach der Modifikation der materialisierten Views und der Anpassung der Kostenfunktionswerte die Wartung der Views abgeschlossen ist. Somit endet die auf Triggern basierende „Snapshot Aktualisierung“, sodass im Bezug auf die materialisierten Views die Ruhephase des Data Warehouses beendet werden kann.

6. Zusammenfassung

6.1. Bewertung

Es wird in der vorliegenden Diplomarbeit ein Verfahren vorgestellt, mit dem Aggregatanfragen an ein relational verwaltetes Data Warehouse unter Verwendung materialisierter Views optimiert werden. Dabei wurden unter anderem die Problemstellungen aus Abschnitt 1.2 betrachtet, welche nachfolgend in Klammern hervorgehoben sind.

Es werden im vorgestellten Optimierungsprozess Aggregatanfragen vor deren Ausführung analysiert (*Analyse der gestellten Aggregatanfragen*) und mit den Metadaten vorhandener materialisierter Views verglichen (*Verwaltung der Metadaten materialisierter Views*). Falls eine materialisierte View zur Beantwortung der ursprünglichen Anfrage verwendet werden kann, wird diese Aggregatanfrage dementsprechend restrukturiert (*Restrukturierung der gestellten Anfrage*). Das heißt, es werden die voraggregierten Datensätze einer materialisierten View verwendet, statt auf die Basisrelationen des entsprechenden Star-Schemas, speziell auf die Faktentabelle, zuzugreifen. Eine Kostenfunktion wurde darüber hinaus entwickelt, mit der unterschiedliche Views untereinander verglichen werden können (*Vergleich materialisierter Views*). Dadurch ist es möglich, dass die jeweils günstigste View bezüglich des Kostenfunktionswertes verwendet wird, falls mehrere Views potentiell zur Beantwortung dienen können. Des Weiteren wurde ein Verfahren eingeführt, mit dem vorhandene materialisierte Views bei der Änderung der zugrunde liegenden Basisrelationen aktualisiert werden können (*Wartung materialisierter Views*).

6.1.1. Konzeptuelle Betrachtung

Das in Abschnitt 4.1 vorgestellte Verfahren illustriert detailliert, auf welche Art und Weise materialisierte Views bezüglich einer gestellten Aggregatanfrage unterschieden werden. Dafür wird die Menge $PosMV$ eingeführt, welche alle zur Beantwortung verwendbaren Views enthält. Beim Aufbau der Menge werden die unterschiedlichen Bestandteile einer Aggregatanfrage nacheinander untersucht. Das heißt, es werden die Dimensionen, Klassifikationsstufen, Kennzahlen, Aggregatfunktionen, Gruppierungsattribute und Restriktionen verglichen. Falls ein Bestandteil dabei nicht passt, wird die entsprechende View nicht weiter betrachtet und analysiert. Ob die gewählte Reihenfolge der Analyse dabei die schnellste Variante zur Ermittlung von $PosMV$ ist, muss untersucht werden. Zum Beispiel wäre es denkbar, dass die Gruppierungsattribute (GROUP BY-Klausel) früher betrachtet werden. Es wird bei den Gruppierungsattributen unterschieden, ob die entsprechenden Attribute passen oder nicht. Durch eine frühere Betrachtung, zum Beispiel direkt nach der Analyse der Dimensionen, könnten

nicht passende Views frühzeitiger entfernt werden. Pauschale Aussagen, welche Reihenfolge der Analyse die sinnvollste oder effektivste ist, können aber nicht getroffen werden und müssten kontextbezogen analysiert werden.

Unter der Voraussetzung, dass mehrere materialisierte Views zur Beantwortung verwendet werden können, werden wahlweise drei Kostenfunktionen angewandt (siehe Abschnitt 4.2). Dabei kann unterschieden werden, ob die Tupelanzahl, Seitenanzahl, Zugriffszeit oder die vorangegangenen Werte im Vergleich zur Faktentabelle als Kostenfunktionswerte dienen. Während die Tupel- und Seitenanzahl mit Hilfe der Metadaten einer Datenbank erlangt werden können, ist zur Ermittlung der Zugriffszeiten das Eingreifen eines Anwenders erforderlich. Das heißt, dass die Verbindungsverzögerungen, Antwortverzögerungen und Übertragungsraten bekannt sein müssen. Des Weiteren ist eine Unterscheidung zwischen Best, Average und Worst Case erforderlich. Inwiefern dieser Aufwand sinnvoll ist, oder ob ein anderer, weniger aufwendiger Kostenfunktionswert im verteilten Szenario verwendet werden kann, muss geklärt werden. Eine Ergänzung der Kostenfunktion wäre diesbezüglich vorstellbar. Der Austausch der Kostenfunktion ist ebenfalls möglich, da diese Funktion nur zur Auswahl einer View, nicht aber beim Aufbau von *PosMV* oder zur Restrukturierung verwendet wird.

Mit Hilfe der Einordnung in *PosMV*, speziell der Zugehörigkeit materialisierter Views zu den *TempRest*-Mengen, kann eine Anfrage restrukturiert werden (siehe Abschnitt 4.3). Dabei werden die Erkenntnisse angewandt, mit denen Views zuvor eingeteilt wurden. Das heißt, falls eine View mindestens eine niedrigere Klassifikationsstufe enthält, dann müssen bei der restrukturierten Anfrage die zugehörigen Dimensionstabellen in die FROM-Klausel aufgenommen werden. Die Trennung der Analyse und Auswahl der materialisierten Views und der Restrukturierung der Anfrage ist dabei sinnvoll, da somit unnötige Operationen verhindert werden. Das heißt, dass nur ein Restrukturierungsprozess vollzogen wird, anstatt diesen Prozess mit Views zu vollziehen, die nicht durch die Kostenfunktion ausgewählt oder im Verlauf der Ermittlung von *PosMV* entfernt werden. Die Restrukturierung ist dennoch direkt abhängig von der Einteilung der Views in die *TempRest*-Mengen. Das heißt, dass falls Änderungen oder Ergänzungen im Bezug auf *PosMV* vorgenommen werden, dann ist eine Überarbeitung des Restrukturierungsprozesses ebenfalls notwendig.

Im Abschnitt 4.4 wird eine Trigger-basierende „Snapshot Aktualisierung“ vorgestellt, während deren Realisierung in 5.3 erläutert wird. Diese Aktualisierung ermöglicht es, die vorhandenen materialisierten Views in den Ruhezeiten eines Data Warehouses zu warten. Ein Problem könnte entstehen, falls die Ruhezeiten eines Data Warehouses im Vergleich zur Menge der zu wartenden Views zu kurz ist. Das heißt, dass die Ruhezeit nicht ausreichen könnte, um alle Views zu aktualisieren. In diesem Fall ist es notwendig, dass entweder die aktualisierten Views markiert werden, um diese von den veralteten Views zu unterscheiden, oder Änderungen während das laufenden Betriebs vorgenommen werden. Beide Möglichkeiten sind denkbar, wurden aber nicht in der Diplomarbeit betrachtet. Allerdings ist eine Erweiterung des Wartungskonzeptes möglich, da dieser Prozess losgelöst von der Optimierung einer Aggregatanfrage behandelt wurde und somit ausgetauscht werden kann.

6.1.2. Realisierung und Veranschaulichung des Konzeptes

Zur Verwaltung materialisierter Views wird in Abschnitt 5.1 ein Relationenschema vorgestellt, mit dem die unterschiedlichen Metadaten der Views gespeichert werden. Darüber hinaus wurden Relationen zur Speicherung, Berechnung und Verwaltung einer Kostenfunktion und zur Realisierung der Wartung der materialisierten Views eingeführt.

Das Relationenschema wurde in den Datenbanken MySQL (Version: 5.0.45) und DB2 (Stufe: v9.5.0.808) umgesetzt, wobei die SQL-Statements zur Erzeugung der Relationen in MySQL im Anhang B.1 aufgelistet werden. Aus Platzgründen wurde allerdings auf die Übernahme der entsprechenden DB2-Statements in die Diplomarbeit verzichtet. Darüber hinaus wurden die Abfragen und SQL-Statements aus Kapitel 5 in MySQL getestet und danach direkt in die Arbeit übernommen. Dazu zählen unter anderem auch die Triggerdefinitionen aus Abschnitt 5.3.

Im Bezug auf den Optimierungsprozess wurde ermittelt, dass das Relationenschema den Anforderungen aus Kapitel 4 entspricht. Das heißt, alle notwendigen Informationen über Views können gespeichert und abgefragt werden. Dies ist unter anderem damit zu begründen, dass das Schema im Anschluss an das Konzept entwickelt wurde. Dadurch kann die klare Trennung der Relationen zur Speicherung von Metadaten, Verwaltung der Kostenfunktion und Wartung der materialisierten Views ebenfalls erklärt werden. Durch dieses Vorgehen können, wie in Abschnitt 6.1.1 erklärt wird, die unterschiedlichen Relationen ausgetauscht bzw. ergänzt werden. Im Bezug auf den Optimierungsprozess, ohne Betrachtung der Wartung materialisierter Views, ist die zentrale Relation die Viewtable. In dieser wird einer View eine eindeutige ViewID zugeordnet, die in den anderen Relationen des Relationenschemas zur Identifizierung dient und im Namen einer View wiedergefunden werden kann. Dadurch ist es möglich, alle gespeicherten Informationen über eine View zu ermitteln, neue Views aufzunehmen und nicht mehr benötigte Views zu entfernen. Inwiefern allerdings der Optimierungsprozess bei einer größeren Anzahl vorhandener Views beeinflusst wird, muss wiederum betrachtet werden. Das heißt, dass der Einfluss der umgesetzten Relationen untersucht werden muss. Dabei wären besonders die Größe der Verwaltungsrelationen und der daraus resultierende Mehraufwand bezüglich des Optimierungsprozesses interessant.

Im Abschnitt 5.2 wird unter Verwendung des Relationenschemas und des Konzepts eine Aggregatanfrage restrukturiert. Diesbezüglich werden zwei Views untersucht, um die unterschiedlichen Arbeitsschritte vom Optimierungsprozess zu veranschaulichen. Die entsprechenden Views wurden dabei so konstruiert und ausgewählt, dass möglichst viele Aspekte des Konzepts erläutert werden können. Das heißt, dass zum Beispiel die eine View passende Klassifikationsstufen besitzt, während die zweite niedrigere Stufen enthält. In diesem Zusammenhang wäre eine Betrachtung weniger akademischer Beispiele interessant. Des Weiteren wird ein Star-Schema verwendet, welches nur zwei Dimensionen besitzt. Dies ist für die Übersichtlichkeit hilfreich, dennoch wäre eine realitätsnahere Betrachtungsweise, vor allem die Verwendung eines komplexeren Star-Schemas, umfangreicherer Datensätze und einer höheren Anzahl materialisierter Views, wünschenswert.

6.2. Ausblick

In der vorliegenden Diplomarbeit wird ein Verfahren beschrieben, mit dem Aggregatanfragen an ein Data Warehouse optimiert werden können. Es wird diesbezüglich erläutert, wie materialisierte Views verwaltet, ausgewählt und zur Restrukturierung verwendet werden können. Ein dafür notwendiges Relationenschema wurde implementiert. Des Weiteren wurden in den vorangegangenen Kapiteln Anfragen formuliert, mit denen notwendige Informationen aus dem Schema abgefragt werden können.

In zukünftigen Arbeiten wäre es denkbar, dass alle vorgestellten Teilprozesse der Optimierung in einem System integriert und vereinigt werden könnten. Das heißt, dass die Analyse einer Aggregatanfrage, die Auswahl einer View, die Restrukturierung der Anfrage, die Verwaltung der Views und die Wartung automatisiert ablaufen würden. Ein solches System könnte eine Aggregatanfrage an ein Data Warehouse vor dessen Ausführung abfangen, analysieren und selbstständig optimieren. Ein Anwender würde in diesem Szenario nur das gewünschtes Resultat in Form einer Menge von Datensätzen erhalten, höchstens noch die Mitteilung, dass die Anfrage optimiert wurde. Notwendige Einstellungen könnten beim erstmaligen Initialisieren des System eingestellt und gespeichert werden, zum Beispiel welcher Kostenfunktionswert verwendet wird.

Darüber hinaus könnten zusätzliche Funktionalitäten ergänzt werden. Es wurden Verfahren präsentiert, mit denen im Vorfeld einer Optimierung eine Menge von materialisierten Views ausgewählt wird. Dieses könnte ebenfalls in obiges System integriert werden. Des Weiteren wäre es möglich, verschiedene weitere Kostenfunktionen umzusetzen und kontextbezogen auszuwählen. Die modulare Betrachtungsweise ermöglicht diese Vorgehensweise. Eine Ergänzung und Erweiterung des Relationenschemas wäre diesbezüglich notwendig.

Bei der Aktualisierung der Basisrelationen, das heißt beim Einfügen, Löschen oder Update von Tupeln, werden die materialisierten Views angepasst. Auf welche Art und Weise allerdings auf strukturelle Veränderungen des Star-Schemas reagiert werden soll, wurde nicht behandelt. Dazu zählen zum Beispiel das Einfügen oder Löschen von Dimensionstabellen oder auch das Hinzufügen von neuen Hierarchiestufen. Eine Erweiterung mit Fokus auf die Wartung der Views ist somit ebenfalls denkbar.

Des Weiteren wäre es interessant, das vorgestellte Verfahren an realen Daten zu testen. Das heißt, dass mit Hilfe verschiedener Benchmarks untersucht wird, in welchen Szenarien eine Optimierung die Antwortzeiten einer Anfrage so negativ beeinflusst, dass auf die Restrukturierung komplett verzichtet werden sollte. Zum Beispiel ist es denkbar, dass in einem Star-Schema mit nur wenigen, „tupelarmen“ Relationen die notwendigen Optimierungsschritte (Abfragen der Metadaten, Restrukturierung, u.s.w.) länger dauern könnten als die direkte Ausführung der entsprechenden Aggregatanfrage.

Bei der Umsetzung des Verfahrens in einem „Komplettsystem“, bei der Erweiterung der Funktionalitäten oder bei der Lösung weiterführender Fragestellungen und offener Probleme sollte allerdings der folgende Ausspruch von Johann Wolfgang von Goethe berücksichtigt werden:

„Jede Lösung eines Problems ist ein neues Problem.“

Literaturverzeichnis

[BG04] Bauer, Andreas und Holger Günzel: *Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung.* Dpunkt.Verlag GmbH, 2004.

[CKPS95] Chaudhuri, Surajit, Ravi Krishnamurthy, Spyros Potamianos und Kyuseok Shim: *Optimizing queries with materialized views.* Seiten 190–200, 1995.

[For07] Forbrig, Peter: *Objektorientierte Softwareentwicklung mit UML.* Hanser, 2007.

[GHQ95] Gupta, Ashish, Venky Harinarayan und Dallan Quass: *Aggregate-query processing in data warehousing environments.* In: *In Proceedings of the International Conference on Very Large Databases*, Seiten 358–369, 1995.

[GM95] Gupta, Ashish und Inderpal Singh Mumick: *Maintenance of materialized views: Problems, Techniques, and applications.* IEEE Data Engineering Bulletin, 18:3–18, 1995.

[GM99] Gupta, Ashish und Iderpal Singh Mumick (Herausgeber): *Materialized views: techniques, implementations, and applications.* MIT Press, Cambridge, MA, USA, 1999.

[Hal00] Halevy, Alon Y.: *Theory of answering queries using views.* SIGMOD Record, 29:40–47, 2000.

[Hal01] Halevy, Alon Y.: *Answering Queries Using Views: A Survey*, 2001.

[HGZS02] Helfert, Markus, Ch-St. Gallen, Gregor Zellner und Carlos Sousa: *Data Quality, Data Quality Management, Data Warehouse Systems 1*, 2002.

[HH02] Helfert, Markus und Clemens Herrmann: *Proactive Data Quality Management for Data Warehouse Systems.* DMDW, 2002:97–106, 2002.

[HRU96] Harinarayan, Venky, Anand Rajaraman und Jeffrey D. Ullman: *Implementing Data Cubes Efficiently.* Seiten 205–216, 1996.

[IWIP02] Inmon, W. H., John Wiley, W. H. Inmon und Wiley Computer Publishing: *Data Warehouse Third Edition Building the Data Warehouse Third Edition*, 2002.

[JLVV01] Jarke, Matthias, Maurizio Lenzerini, Yannis Vassiliou und Panos Vassiliadis: *Fundamentals of Data Warehouses*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[Kle07] Klettke, Meike: *Vorlesung Datenbanken III (Data Warehouses, Data Integration, Data Mining)*. Stud.IP, Universität Rostock, 2007.

[MQM97] Mumick, Inderpal Singh, Dallan Quass und Barinderpal Singh Mumick: *Maintenance of data cubes and summary tables in a warehouse*. SIGMOD Rec., 26(2):100–111, 1997.

[NT04] Nadeau, Thomas P. und Toby J. Teorey: *OLAP Query Optimization in the Presence of Materialized Views*, 2004.

[PL00] Pottinger, Rachel und Alon Levy: *A scalable algorithm for answering queries using views*. In: *In Proc. of VLDB*, Seiten 484–495, 2000.

[QW97] Quass, Dallan und Jennifer Widom: *On-line warehouse view maintenance*. In: *In Proceedings of SIGMOD*, Seiten 393–404, 1997.

[Rou98] Roussopoulos, Nick: *Materialized Views and Data Warehouses*. SIGMOD Record, 27:21–26, 1998.

[SD96] Srivastava, Divesh und Shaul Dar: *Abstract Answering Queries with Aggregation Using Views*, 1996.

[SDN98] Shukla, Amit, Prasad M. Deshpande und Jeffrey F. Naughton: *Materialized view selection for multidimensional datasets*. Seiten 488–499, 1998.

[ZCL+00] Zaharioudakis, Markos, Roberta Cochrane, George Lapis, Hamid Pirahesh und Monica Urata: *Answering complex SQL queries using automatic summary tables*. In: *In SIGMOD*, Seiten 105–116, 2000.

Abbildungsverzeichnis

Glossar

Basisrelationen

Die Basisrelationen bezeichnen die Fakten- und Dimensionstabellen des Star-Schemas.

Data Warehouse

Ein Data Warehouse ist eine themenorientierte, integrierte, persistente und zeitvariante Sammlung von Daten aus zum Zweck der Analyse und Entscheidungsunterstützung.

Datenwürfel

Der Datenwürfel ist eine mehrdimensionale Matrix, deren Achsen durch die Dimensionen aufgespannt werden und deren einzelne Zellen eine oder mehrere Kennzahlen repräsentieren.

Dimension

Eine Dimension ist innerhalb des multidimensionalen Datenmodells eine ausgewählte Entität und beschreibt eine mögliche Sicht auf die zugeordneten Kennzahlen. Dimensionen bilden Klassifikationshierarchien.

Dimensionalität

Die Dimensionalität ist die Anzahl der Dimensionen, die die mehrdimensionale Matrix aufspannen.

Dimensionselement

Dimensionselemente sind die Klassifikationsknoten der niedrigsten Hierarchiestufe.

ETL

ETL ist ein Akronym für Extraktion, Transformation und Laden und bezeichnet den Prozess, mit dem heterogene Daten aus verschiedenen Quellen im Data Warehouse zusammengeführt werden.

Granularität

Die Granularität gibt den Verdichtungsgrad von Daten im Datenwürfel an. Niedrigere Hierarchiestufen haben einen größeren Verdichtungsgrad als höhere Stufen.

Kennzahl

Eine Kennzahl ist eine verdichtete Messgröße, die betriebswirtschaftliche Zusammenhänge beschreibt.

Klassifikationshierarchie

Eine Klassifikationshierarchie ist eine baumartige Struktur, deren Elemente Klassifikationsknoten sind. Klassifikationshierarchien ermöglichen verschiedene Sichten auf die Dimensionselemente.

Klassifikationsknoten

Klassifikationsknoten sind Elemente der Klassifikationshierarchie.

Klassifikationsschema

Ein Klassifikationsschema ist die abstrakte Beschreibung von Klassifikationshierarchien und enthält verschiedene Hierarchiestufen.

Materialisierte View

Eine materialisierte View ist ein relational gespeicherter Datenwürfel, der das Ergebnis einer multidimensionalen Anfrage an ein Data Warehouse ist. Das Data Warehouse wird mit Hilfe des Star-Schemas relational verwaltet.

OLAP

OLAP steht für Online Analytical Processing und ist eine Methode der Analyse auf Grundlage des multidimensionalen Datenmodells.

Restrukturierte Anfrage

Eine restrukturierte Anfrage ist eine Star-Query, die unter Verwendung materialisierter Views derart umgeschrieben wurde, dass keine bzw. so wenig wie möglich Basisrelationen des Data Warehouses verwendet werden.

Star-Join

Ein Star-Join ist ein (n+1)-Wege-Verbund zwischen einer zentralen Faktentabelle und n Dimensionstabelle.

Star-Query

Eine Star-Query bzw. Sternanfrage ist eine multidimensionale Anfrage an ein Sternschema, wobei Aggregatfunktionen über Kennzahlen verwendet werden.

Star-Schema

Das Star-Schema oder auch Sternschema ist ein Datenmodell, mit dem multidimensionale Datenwürfel in einer relationalen Datenbank gespeichert werden können. Das Sternschema enthält eine zentrale Faktentabelle und eine Dimensionstabelle für jede Dimension. Die Tabellen werden mit Hilfe von Schlüssel/Fremdschlüsselbeziehungen verknüpft und Basisrelationen genannt.

A. Anhang Konzept

A.1. Mengenübersicht der Aggregatanfragen

Im Abschnitt 4.1 werden die materialisierten Views in unterschiedliche Mengen eingeteilt. Die Einteilung der Views ist notwendig, da die Zugehörigkeit zu der entsprechenden Menge Einfluß auf das Restrukturieren einer Anfrage hat. Der allgemeine Aufbau einer Aggregatanfrage sieht in SQL-Syntax wie folgt aus:

```
SELECT <Gruppierungsattribute>,
       <Aggregatfunktionen>
FROM <Faktentabelle>,
     <Dimensionstabellen>
WHERE <Verbundbedingungen> AND
      <Restriktionen>
GROUP BY <Gruppierungsattribute>
```

Die Tabelle A.1 illustriert, welche Bestandteile einer Aggregatanfrage analysiert werden, wobei sich die Mengen auf die Abbildung A.2 beziehen:

Mengen	SQL-Klausel	Analyse
①	FROM	Dimensionstabellen
②	SELECT	Klassifikationsstufen
③	SELECT	Kennzahlen
④	SELECT	Aggregatfunktionen
⑤	GROUP BY	Gruppierungsattribute
⑥	WHERE	Restriktionen

Abbildung A.1.: Zuordnung der Mengen zu SQL-Klauseln

Die Tabelle besagt zum Beispiel, dass die Dimensionstabellen der möglichen materialisierten Views als Einteilungskriterium in die Mengen $\boldsymbol{TempDim}_1$ und $\boldsymbol{TempDim}_2$ (also Menge 1) verwendet werden. Dabei sind die entsprechenden Informationen über die Dimensionstabellen in der FROM-Klausel enthalten.

In Abbildung A.2 stehen die $\boldsymbol{TR}_x$ mit x $\in$ {1,...,20} stellvertretend für die Mengen $\boldsymbol{TempRest}_x$ mit x $\in$ {1,...,20}, wobei eine Übersicht der Charakteristika der entsprechenden Mengen in Abbildung A.3 dargestellt wird. Dabei werden die Charakteristika der Mengen im Vergleich einer materialisierten View zu einer Anfrage illustriert. Das heißt, dass zum Beispiel der Eintrag „Weniger streng“ bedeutet, dass eine View der entsprechenden *TempRest*-Menge weniger strenge Restriktionen im Vergleich zu einer Anfrage enthält.

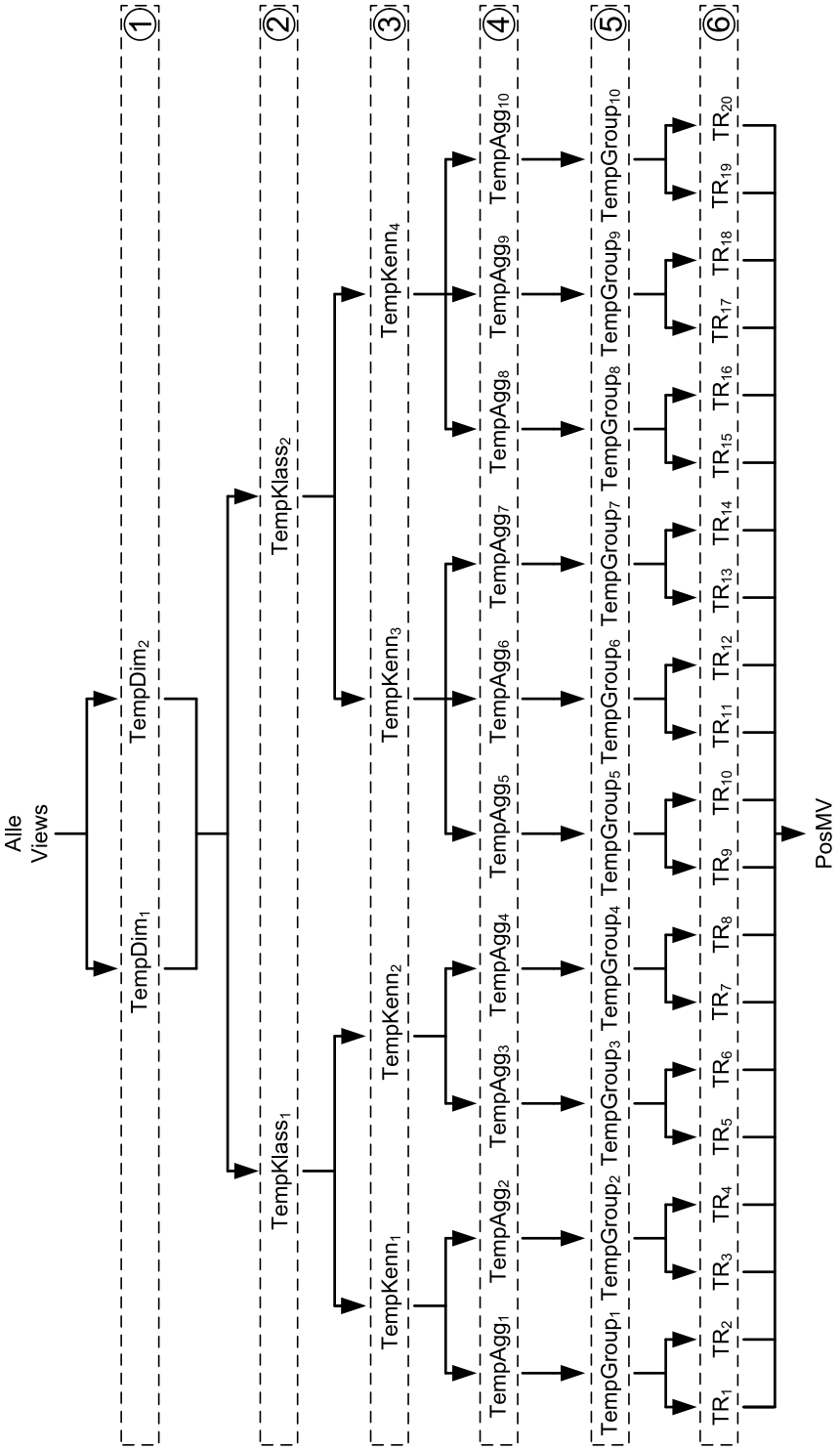

Abbildung A.2.: Mengenübersicht der Aggregatanfragen

Mengen \ Analyse	Dimensionstabellen	Klassifikationsstufen	Kennzahlen	Funktionen	Gruppierung	Restriktionen
$TempRest_1$	Gleich oder mehr	Gleich	Gleich	Gleich	Gleich	Gleich
$TempRest_2$	Gleich oder mehr	Gleich	Gleich	Gleich	Gleich	Weniger streng
$TempRest_3$	Gleich oder mehr	Gleich	Gleich	Kombination	Gleich	Gleich
$TempRest_4$	Gleich oder mehr	Gleich	Gleich	Kombination	Gleich	Weniger streng
$TempRest_5$	Gleich oder mehr	Gleich	Kombination	Gleich	Gleich	Gleich
$TempRest_6$	Gleich oder mehr	Gleich	Kombination	Gleich	Gleich	Weniger streng
$TempRest_7$	Gleich oder mehr	Gleich	Kombination	Kombination	Gleich	Gleich
$TempRest_8$	Gleich oder mehr	Gleich	Kombination	Kombination	Gleich	Weniger streng
$TempRest_9$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Gleich	Gleich	Gleiche Dimensionen	Gleich
$TempRest_{10}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Gleich	Gleich	Gleiche Dimensionen	Weniger streng
$TempRest_{11}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Gleich	Gleich (AVG)	Gleiche Dimensionen	Gleich
$TempRest_{12}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Gleich	Gleich (AVG)	Gleiche Dimensionen	Weniger streng
$TempRest_{13}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Gleich	Kombination	Gleiche Dimensionen	Gleich
$TempRest_{14}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Gleich	Kombination	Gleiche Dimensionen	Weniger streng
$TempRest_{15}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Kombination	Gleich	Gleiche Dimensionen	Gleich
$TempRest_{16}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Kombination	Gleich	Gleiche Dimensionen	Weniger streng
$TempRest_{17}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Kombination	Gleich (AVG)	Gleiche Dimensionen	Gleich
$TempRest_{18}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Kombination	Gleich (AVG)	Gleiche Dimensionen	Weniger streng
$TempRest_{19}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Kombination	Kombination	Gleiche Dimensionen	Gleich
$TempRest_{20}$	Gleich oder mehr	Gleich oder niedriger (min. eine niedriger)	Kombination	Kombination	Gleiche Dimensionen	Weniger streng

Abbildung A.3.: Charakteristika der *TempRest*-Mengen

A.2. Berechnung des Durchschnitts

In den Abschnitten 4.1 und 4.3 wurde die Sonderstellung der Aggregatfunktion AVG im Zusammenhang mit der Restrukturierung einer Anfrage dargestellt. Nachfolgend soll ausführlicher erläutert werden, wie der Durchschnitt einer höheren Klassifikationsstufe aus den Durchschnittswerten der niedrigeren aggregiert werden kann. Das Beispiel der Abbildung A.4 wird für die Erläuterungen verwendet:

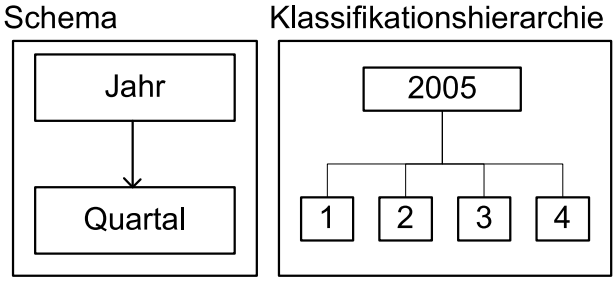

Abbildung A.4.: Klassifikationsschema und -hierarchie für AVG-Erläuterung

Es sind ein Klassifikationsschema und eine zugeordnete Klassifikationshierarchie dargestellt. Zur Erläuterung wird des Weiteren eine Kennzahl „Verkauf“ verwendet, wobei die Aggregation aller Verkäufe der Hierarchiestufe „Quartal“ den Verkauf der Hierarchiestufe „Jahr“ ergeben. Die Dimensionselemente „1“, „2“, „3“ und „4“ stehen stellvertretend für die entsprechenden Quartale, während „2005“ ein Jahr repräsentiert.

Der Durchschnitt vom „Verkauf“ wird berechnet als Quotient aller Einzelverkäufe und der Anzahl der Verkäufe. Es ergibt sich im Zusammenhang mit der Kennzahl „Verkauf“ folgende Formel:

$$Durchschnitt_{Verkauf} = \frac{(Summe\ der\ Einzelverkäufe)_{Verkauf}}{(Anzahl\ der\ Verkäufe)_{Verkauf}}$$

Diese Formel kann auf die unterschiedlichen Dimensionselemente angewendet werden. Dabei wird die Aggregatfunktion AVG für den „Durchschnitt“, SUM für „Summe der Einzelverkäufe“ und COUNT für „Anzahl der Verkäufe“ verwendet. Des Weiteren beziehen sich die Erläuterungen nachfolgend auf die Kennzahl „Verkauf“, sodass diese für eine verbesserte Übersichtlichkeit weggelassen wird. Es ergibt sich folgende Formel:

$$AVG_x = \frac{SUM_x}{COUNT_x} \qquad x \in \{1, 2, 3, 4\} \tag{A.1}$$

Der Durchschnitt des Dimensionselementes „2005“ kann nun unter Beachtung der funktionalen Abhängigkeiten zwischen den Klassifikationsstufen mit folgender Formel berechnet werden:

$$AVG_{2005} = \sum_{y=1}^{4} \frac{SUM_y}{\sum_{x=1}^{4} COUNT_x} \tag{A.2}$$

Mit Hilfe der Gleichung (A.2) illustrieren die nachfolgenden Formeln, warum der Durchschnitt der Hierarchiestufe „2005“ nicht die Summe der Durchschnittswerte von „1“, „2“, „3“ und „4“ ist. Das heißt, es wird gezeigt, warum aus der Summe der Durchschnittswerte einer niedrigeren Klassifikationsstufen nicht der Durchschnitt einer höheren Stufe berechnet werden kann:

$$\begin{aligned} AVG_{2005} &= \sum_{y=1}^{4} \frac{SUM_y}{\sum_{x=1}^{4} COUNT_x} \\ &\Updownarrow \\ AVG_{2005} &\neq \frac{SUM_1}{COUNT_1} + \frac{SUM_2}{COUNT_2} + \frac{SUM_3}{COUNT_3} + \frac{SUM_4}{COUNT_4} \\ &\Updownarrow \\ AVG_{2005} &\neq AVG_1 + AVG_2 + AVG_3 + AVG_4 \end{aligned}$$

Durch die Umformung der Gleichung (A.2), das heißt, durch die Ersetzung von SUM durch die Multiplikation von AVG und COUNT gemäß (A.1), ergibt sich folgende Formel:

$$\begin{aligned} AVG_{2005} &= \sum_{y=1}^{4} \frac{SUM_y}{\sum_{x=1}^{4} COUNT_x} \\ &\Updownarrow \\ AVG_{2005} &= \sum_{y=1}^{4} \frac{AVG_y * COUNT_y}{\sum_{x=1}^{4} COUNT_x} \end{aligned} \tag{A.3}$$

Die Gleichung (A.3) stellt dar, warum eine Anfrage Q, welche als Aggregatfunktion AVG verwendet, nicht alleine mit einer View V_1 beantwortet werden kann, deren Klassifikationsstufen niedriger sind und die nur AVG enthält. Es wird eine zweite View V_2 benötigt, welche die gleichen Klassifikationsstufen im Vergleich zu V_1 besitzt und COUNT selektiert hat. Das heißt, dass im Bezug auf (A.3) V_1 die Daten für AVG_y und V_2 die Daten für $COUNT_y$ und $COUNT_x$ liefert. Alternativ könnte V_1 sowohl AVG als auch COUNT ausgewählt haben, sodass die Daten für AVG_y, $COUNT_y$ und $COUNT_x$ komplett von V_1 stammen. Unter Verwendung der Views V_1 und V_2 bzw. der View V_1 alleine könnte die Anfrage Q somit beantwortet werden.

B. Anhang Umsetzung

B.1. SQL-Statements des Relationenschemas

In Abschnitt 5.1 wird das Relationenschema beschrieben, welches zur Optimierung von materialisierten Views verwendet wird. Nachfolgend werden die unterschiedlichen SQL-Statements zur Erzeugung der Relationen aufgelistet. Die entsprechenden Statements beziehen sich auf eine MySQL Datenbank (MySQL-Version: 5.0.45).

Starschema

```
CREATE TABLE `da`.`facttable` (
    `day` INTEGER UNSIGNED NOT NULL DEFAULT 0,
    `cust_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
    `sell` INTEGER UNSIGNED NOT NULL DEFAULT 0,
    `cost` INTEGER UNSIGNED NOT NULL DEFAULT 0,
   PRIMARY KEY (`day`, `cust_id`),
   CONSTRAINT `FK_facttable_calendar_day`
     FOREIGN KEY `FK_facttable_calendar_day` (`day`)
     REFERENCES `calendar` (`day`)
       ON DELETE RESTRICT
       ON UPDATE RESTRICT,
   CONSTRAINT `FK_facttable_customer_cust_id`
     FOREIGN KEY `FK_facttable_customer_cust_id` (`cust_id`)
     REFERENCES `customer` (`cust_id`)
       ON DELETE RESTRICT
       ON UPDATE RESTRICT
) ENGINE = InnoDB;

CREATE TABLE `da`.`calendar` (
    `day` INTEGER UNSIGNED NOT NULL DEFAULT 0,
    `week` INTEGER UNSIGNED NOT NULL,
    `month` VARCHAR(45) NOT NULL,
    `year` INTEGER UNSIGNED NOT NULL,
   PRIMARY KEY (`day`)
) ENGINE = InnoDB;
```

```
CREATE TABLE `da`.`customer` (
   `cust_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
   `city` VARCHAR(45) NOT NULL,
   `state` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`cust_id`)
) ENGINE = InnoDB;
```

Viewtable und Views

```
CREATE TABLE `da`.`viewtable` (
   `viewid` INTEGER UNSIGNED NOT NULL DEFAULT 0,
   `calendar` INTEGER UNSIGNED NOT NULL,
   `customer` INTEGER UNSIGNED NOT NULL,
   `costfunction` REAL NOT NULL DEFAULT 0,
  PRIMARY KEY (`viewid`)
) ENGINE = InnoDB;

CREATE VIEW `da`.`v_AST1` AS
  SELECT calendar.month, customer.city,
         SUM(facttable.cost) AS cost,
         SUM(facttable.sell) AS sell_sum,
         COUNT(facttable.sell) AS sell_count
  FROM calendar, customer, facttable
  WHERE calendar.day = facttable.day AND
        customer.cust_id = facttable.cust_id AND
        calendar.year < 2008
  GROUP BY calendar.month, customer.city;
CREATE TABLE `da`.`AST1`
  SELECT * FROM v_AST1;

CREATE View `da`.`v_AST2`
  SELECT calendar.day, customer.cust_id,
         SUM(facttable.cost) AS cost,
         AVG(facttable.sell) AS sell_avg,
         COUNT(facttable.sell) AS sell_count
  FROM calendar, customer, facttable
  WHERE calendar.day = facttable.day AND
        customer.cust_id = facttable.cust_id AND
        calendar.year < 2008 AND
       NOT (calendar.month = 'November') AND
       NOT (calendar.month = 'Juni')
  GROUP BY calendar.day, customer.cust_id;
CREATE TABLE `da`.`AST2`
  SELECT * FROM v_AST2;
```

Metadaten

```
CREATE TABLE `da`.`dimensions` (
    `dimensions_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
    `dimension` VARCHAR(45) NOT NULL,
    `name` VARCHAR(45) NOT NULL,
    `value` INTEGER UNSIGNED NOT NULL,
    `successor` INTEGER UNSIGNED,
  PRIMARY KEY (`dimensions_id`)
) ENGINE = InnoDB;

CREATE TABLE `da`.`fitable` (
    `fitable_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
    `viewid` INTEGER UNSIGNED NOT NULL,
    `attribute` VARCHAR(45) NOT NULL,
    `fromtable` VARCHAR(45) NOT NULL,
    `fromtablealias` VARCHAR(45) NOT NULL,
    `function` VARCHAR(45) DEFAULT NULL,
    `group` SMALLINT UNSIGNED NOT NULL DEFAULT 0,
  PRIMARY KEY (`fitable_id`),
  CONSTRAINT `FK_fitable_viewtable_viewid`
    FOREIGN KEY `FK_fitable_viewtable_viewid` (`viewid`)
    REFERENCES `viewtable` (`viewid`)
      ON DELETE RESTRICT
      ON UPDATE RESTRICT
) ENGINE = InnoDB;

CREATE TABLE `da`.`rtable` (
    `rtable_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
    `viewid` INTEGER UNSIGNED NOT NULL,
    `attribute` VARCHAR(45) NOT NULL,
    `lb` VARCHAR(45) DEFAULT NULL,
    `ub` VARCHAR(45) DEFAULT NULL,
    `connection` VARCHAR(45) DEFAULT NULL,
    `not` SMALLINT UNSIGNED DEFAULT 0,
  PRIMARY KEY (`rtable_id`),
  CONSTRAINT `FK_rtable_viewtable_viewid`
    FOREIGN KEY `FK_rtable_viewtable_viewid` (`viewid`)
    REFERENCES `viewtable` (`viewid`)
      ON DELETE RESTRICT
      ON UPDATE RESTRICT
) ENGINE = InnoDB;
```

```
CREATE TABLE `da`.`measures` (
  `measures_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
  `viewid` INTEGER UNSIGNED NOT NULL,
  `measure` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`measures_id`),
  CONSTRAINT `FK_measures_viewtable_viewid`
    FOREIGN KEY `FK_measures_viewtable_viewid` (`viewid`)
    REFERENCES `viewtable` (`ViewID`)
      ON DELETE RESTRICT
      ON UPDATE RESTRICT
) ENGINE = InnoDB;
```

Kostenfunktion

```
CREATE TABLE `da`.`functiontable` (
  `viewid` INTEGER UNSIGNED NOT NULL DEFAULT 0,
  `tuple` BIGINT UNSIGNED NOT NULL DEFAULT 0,
  `sizetable` BIGINT UNSIGNED NOT NULL DEFAULT 0,
  `sizeindex` BIGINT UNSIGNED NOT NULL DEFAULT 0,
  `time` REAL NOT NULL DEFAULT 0,
  CONSTRAINT `FK_functiontable_viewtable_viewid`
    FOREIGN KEY `FK_functiontable_viewtable_viewid` (`viewid`)
    REFERENCES `viewtable` (`viewid`)
      ON DELETE RESTRICT
      ON UPDATE RESTRICT
) ENGINE = InnoDB;

CREATE TABLE `da`.`choices_station` (
  `choicesstation_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
  `values` VARCHAR(45) NOT NULL,
  `comment` VARCHAR(100) DEFAULT NULL,
  PRIMARY KEY (`choicesstation_id`)
) ENGINE = InnoDB;

CREATE TABLE `da`.`choices_function` (
  `choicesfunction_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
  `values` VARCHAR(45) NOT NULL,
  `comment` VARCHAR(100) DEFAULT NULL,
  PRIMARY KEY (`choicesfunction_id`)
) ENGINE = InnoDB;
```

```
CREATE TABLE `da`.`functionfacts` (
   `fact` VARCHAR(45) NOT NULL,
   `value` REAL DEFAULT 0,
   `comment` VARCHAR(100) DEFAULT NULL,
PRIMARY KEY (`fact`)
) ENGINE = InnoDB;

CREATE TABLE `da`.`stations` (
   `stations_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
   `viewid` INTEGER UNSIGNED DEFAULT 0,
   `stationid` INTEGER UNSIGNED NOT NULL,
   `value` REAL NOT NULL DEFAULT 0,
   `case` INTEGER UNSIGNED NOT NULL,
   `connectiondelay` REAL NOT NULL,
   `answerdelay` REAL NOT NULL,
   `tuple` INTEGER UNSIGNED NOT NULL,
   `tuplesize` BIGINT UNSIGNED NOT NULL,
   `transfer` REAL NOT NULL,
  PRIMARY KEY (`stations_id`),
  CONSTRAINT `FK_stations_choices-station`
    FOREIGN KEY `FK_stations_choices-station` (`case`)
    REFERENCES `choices_station` (`choicesstation_id`)
      ON DELETE RESTRICT
      ON UPDATE RESTRICT,
  CONSTRAINT `FK_stations_viewtable_viewid`
    FOREIGN KEY `FK_stations_viewtable_viewid` (`viewid`)
    REFERENCES `viewtable` (`viewid`)
      ON DELETE RESTRICT
      ON UPDATE RESTRICT
) ENGINE = InnoDB;
```

Wartung

```
CREATE TABLE `da`.`updatetable` (
   `update_id` INTEGER UNSIGNED NOT NULL DEFAULT 0,
   `change` VARCHAR(100) NOT NULL,
   `value` SMALLINT UNSIGNED NOT NULL DEFAULT 0,
  PRIMARY KEY (`update_id`)
) ENGINE = InnoDB;
```

B.2. Ermittlung der Kostenfunktionswerte

In Abschnitt 5.1.4 werden die Relationen vorgestellt, mit denen die Kostenfunktion realisiert wird. Für die Kostenfunktion werden unterschiedliche Informationen bzw. Kostenfunktionswerte benötigt, welche in einer MySQL Datenbank (MySQL-Version: 5.0.45) wie folgt ermittelt werden können:

Größe von Datenbank- und Indexseiten

```
SHOW STATUS WHERE variable_name = 'innodb_page_size';
```

Die Größe der Datenbankseiten wird in Byte angegeben und für die Speicher-Engine „InnoDB“ ermittelt. Der Wert kann in die Relation „functionsfacts“ eingetragen werden.

Anzahl der Tupel, Datenbank- und Indexseiten

```
SELECT table_rows, data_length, index_length
FROM information_schema.tables
WHERE table_schema = 'da' AND
      table_name = <Relation>;
```

Die Anzahl der Tupel einer Relation sind im Attribut „table_rows“ gespeichert. Die Anzahl der Datenbank- und Indexseiten kann mit Hilfe der Attribute „data_length“ bzw. „index_length“ berechnet werden. Diese Werte müssen dabei durch die Größe der Seiten der entsprechenden Speicher-Engine geteilt werden, um die genaue Anzahl der Datenbank- und Indexseiten zu bekommen. Die ermittelten Werte können in die Relationen „functiontable“ oder „functionsfacts“ übernommen werden.

Zugriffszeiten

```
SELECT table_rows, avg_row_length
FROM information_schema.tables
WHERE table_schema = 'da' AND
      table_name = <Relation>;
```

Im verteilten Szenario wird die durchschnittliche Tupelgröße zur Berechnung der Zugriffszeiten benötigt, welche im Attribut „avg_row_length“ gespeichert ist und in das Attribut „tuplesize“ der Relation „stations“ übernommen werden kann. Die Anzahl der Tupel einer Station ist wiederum in „table_rows“ enthalten und kann in „tuple“ gesichert werden. Die Parameter „connectiondelay“, „answerdelay“ und „transfer“ müssen durch den Nutzer eingetragen werden. Ist dies geschehen kann das Attribut „value“ der Relation „stations“ wie folgt berechnet werden: „value“ = „connectiondelay“ + „answerdelay“ + („tuple“ * „tuplesize“) / „transfer“. Der berechnete Wert kann daraufhin unter Beachtung des Falls (siehe Abschnitt 4.2.1) in die Relation „functiontable“ ins Attribut „time“ übernommen werden.

Selbständigkeitserklärung

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

Ort, Datum Unterschrift

Thesen

1. Typische Anfragen an ein Data Warehouse sind Aggregatanfragen, die häufig eine große Anzahl von Tupeln auswerten.
2. Aggregatanfragen an ein Star-Schema können optimiert werden.
3. Der Zugriff auf die Faktentabelle eines Star-Schemas ist kostenintensiv, da Faktentabellen in realen Szenarien Millionen von Einträgen enthalten.
4. Voraggregierte Relationen sind das Ergebnis früherer Aggregatanfragen und werden materialisierte Views genannt. Sie können zur Beantwortung von Anfragen an ein Data Warehouse genutzt werden.
5. Der Zugriff auf voraggregierte Relationen ist in der Regel weniger kostenintensiv, weil diese Relationen das Ergebnis von Aggregatanfragen sind. Somit vereinfachen sich die Berechnungen.
6. Die formale Beschreibung der materialisierten Views muss vorliegen und kann in dem in der Diplomarbeit vorgestellte Relationenschema gespeichert und verwaltet werden.
7. Materialisierte Views, deren Metadaten nicht komplett mit einer Aggregatanfrage übereinstimmen, können unter bestimmten Voraussetzungen dennoch zur Optimierung genutzt werden. Zum Beispiel können durch eine nachträgliche Aggregation benötigte Datensätze ermittelt werden, sodass die Klassifikationsstufen einer materialisierten View niedriger sein können als in der gestellten Anfrage.
8. Es werden sowohl parallele als auch einfache Klassifikationshierarchien beachtet.
9. Alias-Werte in den materialisierten Views oder gestellten Aggregatanfragen können erkannt und entsprechend behandelt werden.
10. Beim Fehlen oder bei mangelnder Eignung materialisierter Views wird die ursprüngliche Anfrage ausgeführt.
11. Es werden in der Diplomarbeit mehrere Kostenfunktionen vorgestellt. Mit Hilfe dieser Funktionen wird die günstigste materialisierte View ausgewählt. Falls nur eine materialisierte View vorhanden ist, wird diese durch die Kostenfunktionen selektiert.
12. Der Restrukturierungsprozess liefert syntaktisch korrekte und ausführbare SQL-Statements. Eine Implementation des Verfahrens als Komplettsystem ist realisierbar.

13. Das Verfahren ist modular aufgebaut, wodurch der Austausch und die Ergänzung vorgestellter Funktionalitäten des Optimierungsprozesses möglich sind.

14. Die vorgestellte Aktualisierungsmethode ist ein Snapshot-Verfahren, das in der Ruhezeit eines Data Warehouses ausgeführt werden kann. Die Veränderungen der Basisrelationen, beruhend auf dem Einfügen, Löschen oder Update von Tupeln, werden erkannt und behandelt.