Effiziente Laufzeitsysteme für Datenlager

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Diplom-Wirtschaftsinformatiker
Till Westmann
aus Hannover

Mannheim, 2000

Dekan: Professor Dr. Guido Moerkotte, Universität Mannheim
Referent: Professor Dr. Guido Moerkotte, Universität Mannheim
Korreferent: Professor Dr. Georg Lausen, Universität Freiburg

Tag der mündlichen Prüfung: 11. Oktober 2000

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die aktuellen relationalen Datenbankverwaltungssysteme sind für einen hohen Transaktionsdurchsatz optimiert. Sie sind also in der Lage viele „kurze“ Anfragen und atomare Änderungen an dem zu verwaltenden Datenbestand effizient durchzuführen. Dies entspricht den Anforderungen, die die „klassischen“, operationalen Anwendungen an die Datenhaltung stellen. In den letzten Jahren gewinnt jedoch die analytische Verarbeitung der Daten zunehmend an Bedeutung. Die analytische Verarbeitung nutzt üblicherweise historischen Daten, die zu diesem Zweck aus den operationalen Datenbeständen in eine spezielle, von den operationalen Systemen getrennte Datenbank, ein *Datenlager* (engl. *Data Warehouse*), übertragen werden.

Die Anforderungen an ein *Datenlagerverwaltungssystem* unterscheiden sich grundlegend von denen, auf die die aktuellen relationalen Datenbankverwaltungssysteme zugeschnitten sind. In einem Datenlager spielen kleine Änderungen in der Regel keine Rolle, da es historische Daten enthält, die sich nicht mehr ändern. Wenn es in einem Datenlager zu Änderungen des Datenbestands kommt, wird das Datenlager um aktuelle Daten erweitert oder alte Daten werden aus dem Datenlager entfernt. In einem Datenlager überwiegt der lesende Zugriff deutlich.

Betrachten wir als Beispiel eine Fluggesellschaft. Eine typische Aufgabenstellung für eine operationale Anwendung ist: „Reserviere für den 30. Juni 2000 einen Nichtraucherplatz in einem Flug von Frankfurt nach New York.“ Dazu muß zuerst überprüft werden, ob ein solcher Platz noch frei ist und anschließend muß dieser Platz als *belegt* markiert werden. Ein solcher Anwendungsfall muß nur einen sehr kleinen Teil der vorhandenen Daten lesen und – unter Umständen – einen noch kleineren Teil der gelesenen Daten ändern. Eine Aufgabenstellung für eine analytische Anwendung ist: „Wie hoch war der Anteil der Flugscheine erster Klasse an den insgesamt verkauften Flugscheinen auf der Strecke Frankfurt–New York in den letzten zwölf Monaten?“ Für diesen Fall muß ein deutlich größerer Anteil der Daten gelesen werden und Änderungen sind nicht erforderlich.

Das Beispiel zeigt auch, daß analytische Anfragen in der Regel komplexer als Anfragen in operationalen Anwendungen sind. Im Beispiel müssen die von der operationalen Anwendung

benötigten Daten nur gefunden und gelesen werden. Die analytische Anfrage erfordert zusätzlich eine anschließende Aggregation der Daten.

Es ist offensichtlich, daß sich ein Datenbankverwaltungssystem, das analytische Anwendungen optimal unterstützen soll, sich von einen System, das auf operationale Anwendungen zugeschnitten ist, unterscheiden muß. Die Anforderung nach der effizienten Beantwortung von komplexen Anfragen auf großen Datenbeständen muß sich in einem Datenlagerverwaltungssystem niederschlagen.

Das Ziel dieser Arbeit ist die Unterschiede in den Anforderungen und den Gegebenheiten zu identifizieren und für die Anfragebearbeitung auszunutzen. Wir entwickeln dazu eine Reihe von Techniken, die die Eigenschaften analytischer Anwendungen ausnutzen, um die Anfrageauswertung effizienter zu gestalten.

Im Gegensatz zu den meisten anderen Forschungsarbeiten konzentrieren wir uns dabei auf das Laufzeitsystem des Datenlagerverwaltungssystems, da ein effizientes Laufzeitsystem eine notwendige Voraussetzung für die Effizienz des Gesamtsystems ist. Wir betrachten dabei nicht nur die konzeptionellen Probleme, sondern auch die Probleme der Implementierung und haben daher alle entwickelten Techniken in einem Prototyp implementiert und experimentell evaluiert.

Die Arbeit ist wie folgt aufgebaut. In Kapitel 2 erläutern und begründen wir zunächst den Aufbau eines Datenlagers und die sich daraus ergebenden Anforderungen an ein Datenlagerverwaltungssystem. Außerdem erläutern wir in diesem Kapitel die verwendeten Begriffe, und wir geben einen kurzen Überblick über andere Forschungsrichtungen im Bereich der analytischen Anfrageverarbeitung.

Das Thema des Kapitels 3 ist unser Laufzeitsystem für Datenlagerverwaltungssysteme AODB. Wir beschreiben dort die Entwurfsziele und -prinzipien und die Architektur des Systems. Außerdem geben wir einen Einblick in die Funktionsweise und in das Zusammenspiel der einzelnen Komponenten.

In Kapitel 4 wenden wir uns der Integration von Kompressionstechniken in ein Datenlagerverwaltungssystem zu. Wir verwenden Kompression, um die Anfragebearbeitung zu beschleunigen. Da zur Bearbeitung einer Anfrage häufig größere Datenmengen vom Sekundärspeicher in den Primärspeicher gelesen werden müssen, können die Transferkosten – durch die Verwendung einer komprimierten Darstellung der Daten auf dem Sekundärspeicher – reduziert werden. Allerdings muß gleichzeitig sichergestellt werden, daß die Reduktion der Transferkosten nicht durch eine Steigerung der Prozessorkosten (engl. *CPU-costs*) kompensiert wird. Daher konzentrieren wir uns auf Kompressionsverfahren, die möglichst geringe Anforderungen an die Rechenleistung stellen, und nicht auf Verfahren, die möglichst hohe Kompressionsraten erzielen. Außerdem stellen wir ein Verfahren vor, mit dessen Hilfe der Dekomprimierungsvorgang sehr ressourcenschonend erfolgen kann.

In Kapitel 5 beschäftigen wir uns mit Anfragemustern. Anfragemuster sind Muster, die man bei der Betrachtung einer großen Anzahl von Anfragen eines bestimmten Anwendungsbereichs

findet. Wir stellen vier Muster vor, die wir bei der analytischen Anfragebearbeitung entdeckt haben. Bei der Auswahl der Muster ist entscheidend, daß sie speziell genug sind, um Leistungsgewinne zu ermöglichen, und allgemein genug sind, um den Aufwand einer Implementierung zu rechtfertigen. Da analytische Anwendungen häufig mit aggregierten Werten arbeiten, ist es nicht verwunderlich, daß drei der vier Muster Aggregationen enthalten. Wir beschreiben die Muster und wie das Vorhandensein dieser Muster für die Anfrageverarbeitung genutzt werden kann. Wir geben dazu zu jedem Muster einen (relationen-)algebraischen Operator an, der eine besonders effiziente Implementierung von Anfragen, die dem jeweiligen Muster entsprechen, ermöglicht.

Kapitel 6 faßt die Arbeit und ihre Ergebnisse zusammen.

Kapitel 2

Datenlager

In diesem Kapitel werden das Konzept eines Datenlagers und die daraus resultierenden Anforderungen an ein Datenbanksystem kurz dargestellt.

2.1 Transaktionsverarbeitung und Analytische Verarbeitung

Es gibt im wesentlichen zwei Anwendungsarten für Datenbanksysteme im Dialogbetrieb:

- Transaktionsverarbeitung (engl. *On-Line Transaction Processing*, OLTP) und
- Analytische Verarbeitung (engl. *On-Line Analytical Processing*, OLAP).

Eine typische Aufgabenstellung einer OLTP-Anwendung ist die „Verarbeitung einer Bestellung“ bei einem Großhändler. Es handelt sich dabei um eine operationale Anwendung, die *aktuelle Daten* verwendet und eine *kleine Anzahl einzelner Datensätze* liest und ändert. Zentrales Ziel ist hierbei die schnelle und korrekte Durchführung der notwendigen Änderungen. Im Gegensatz dazu dienen OLAP-Anwendungen der Entscheidungsunterstützung und arbeiten in der Regel auf *historischen Daten*. Dabei werden aus *großen Datenmengen* Informationen extrahiert, die die Analyse der vorhandenen Daten ermöglichen. Hierbei liegt die zentrale Forderung in einer schnellen Beantwortung komplexer, lesender Anfragen. Eine typische Fragestellung für den Großhändler ist: „Wie hat sich das Volumen der Bestellungen aus den französischsprachigen Ländern der EU in den letzten drei Jahren entwickelt?“ Aufgrund der unterschiedlichen Aufgaben der Systeme unterscheiden sich auch die Nutzergruppen in Größe und Zusammensetzung. Während OLTP-Anwendungen zum täglichen Geschäft fast aller Mitarbeiter eines Unternehmens gehören, werden OLAP-Anwendungen nur von der deutlich kleineren Gruppe der Entscheidungsträger genutzt. Außerdem ist das OLAP-Anwendungen zugrundeliegende Datenvolumen deutlich größer, da historische Daten über einen längeren Zeitraum vorhanden sein

OLTP-Anwendungen	OLAP-Anwendungen
operationales Geschäft	Entscheidungsunterstützung
aktuelle Daten	historische Daten
Datenvolumen im MB bis GB Bereich	Datenvolumen im GB bis TB Bereich
hauptsächlich Änderungen	hauptsächlich lesender Zugriff
einfache Anfragen	komplexe Anfragen
viele Nutzer	wenige Nutzer

Tabelle 2.1: Unterschiede zwischen OLTP- und OLAP-Anwendungen

müssen. In Tabelle 2.1 sind die Unterschiede zwischen OLAP- und OLTP-Anwendungen zusammengefaßt.

2.2 Datenlager

Aufgrund der Unterschiede in den Anforderungen an die zugrundeliegenden Daten und in der Nutzergruppe geht man heute davon aus, daß für OLAP-Anwendungen ein eigener Datenbestand existieren sollte. Auch die Nutzung eines beiden Datenbestände enthaltenden Systems ist nicht sinnvoll, da:

1. OLTP-Datenbanken und -Datenbankverwaltungssysteme für die schnelle Durchführung von Änderungstransaktionen auf kleinen Datenbeständen optimiert und daher in der Regel nur bedingt für die Verarbeitung großer Datenmengen geeignet sind,

2. das Halten historischer Daten und die Ressourcenanforderungen komplexer analytischer Anfragen, die operationalen Systeme zusätzlich belasten würden und

3. die operationalen Daten häufig über mehrere unterschiedlich organisierte Systeme verteilt und daher einer Analyse nur schwer zugänglich sind.

Man benutzt daher als Grundlage für OLAP-Anwendungen ein spezielles Datenbanksystem, ein *Datenlager* (engl. *Data Warehouse*), das ausschließlich die historischen Daten in integrierter und konsolidierter Form enthält.

Das Datenlager wird aus den operationalen Datenbanken gespeist (vgl. Abbildung 2.1). Es erhält zunächst seinen initialen Datenbestand und wird dann in regelmäßigen Abständen (zum Beispiel täglich oder wöchentlich) um aktuelle Daten erweitert. Um die für Analysen notwendige Datenqualität sicherzustellen, werden die Daten *konsolidiert*, d.h. eventuell fehlenden Daten werden – falls möglich – ergänzt und inkonsistente Daten korrigiert, und *integriert*, d.h. semantisch äquivalente Daten werden in eine einheitliche, dem Schema des Datenlagers entsprechende Darstellung überführt.

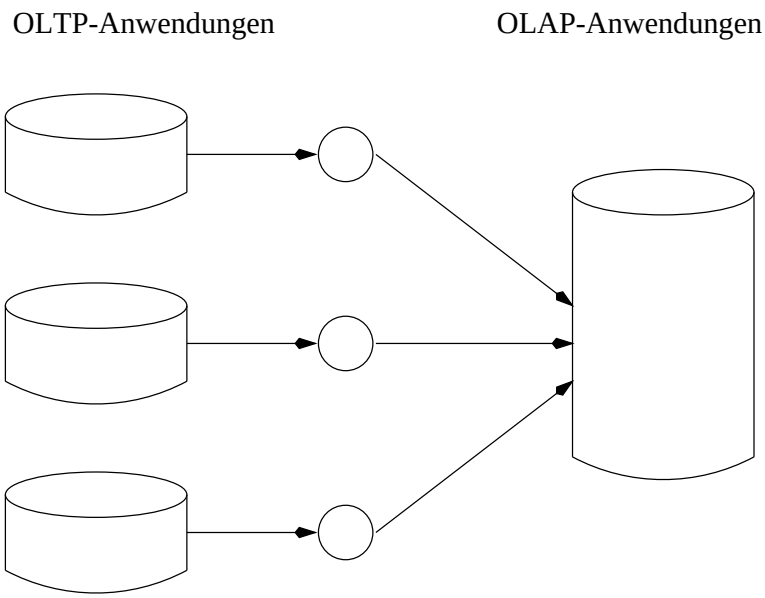


Abbildung 2.1: Operationale Datenbanken und Datenlager

2.3 Datenwürfel

Zur Darstellung und Analyse der Daten wird häufig ein – von der Darstellung in Tabellenkalkulationsprogrammen beeinflußtes – mehrdimensionales Modell verwendet. In diesem Modell gibt es eine Menge *numerischer Werte*, wie zum Beispiel Umsätze, Gewinne oder Lagerbestände, die die zu analysierenden Objekte sind. Jeder Wert hängt von einer Menge von *Dimensionen* ab, die den Kontext des Wertes beschreiben. So können zum Beispiel *Produkt*, *Verkaufsort* und *Verkaufszeitpunkt* im Einzelhandel die zu einer Umsatzzahl gehörenden Dimensionen sein.

Die Dimensionen haben in der Regel mehrere Attribute, die unter Umständen hierarchisch angeordnet werden können. Die Dimension Verkaufszeitpunkt kann zum Beispiel die Attribute Tag, Woche, Monat und Jahr haben. In diesem Fall gibt es zwei hierarchische Anordnungen der Attribute: Tag → Monat → Jahr und Tag → Woche → Jahr. Für die Dimension Produkt sind Produktbezeichnung, Produktkategorie und Hersteller mögliche Attribute und als Attribute für die Dimension Verkaufsort sind Region, Land, Stadt, Adresse geeignet.

Man nimmt an, daß ein Wert durch seinen Kontext eindeutig bestimmt ist. Daher kann jeder Wert als ein Punkt in dem von den Dimensionen aufgespannten Raum betrachtet werden. Eine übliche Darstellung der Daten ist der n-dimensionale *Datenwürfel* (engl. *Data Cube*). Die Zellen eines Datenwürfels enthalten jeweils aggregierte Werte, wie zum Beispiel den gesam-

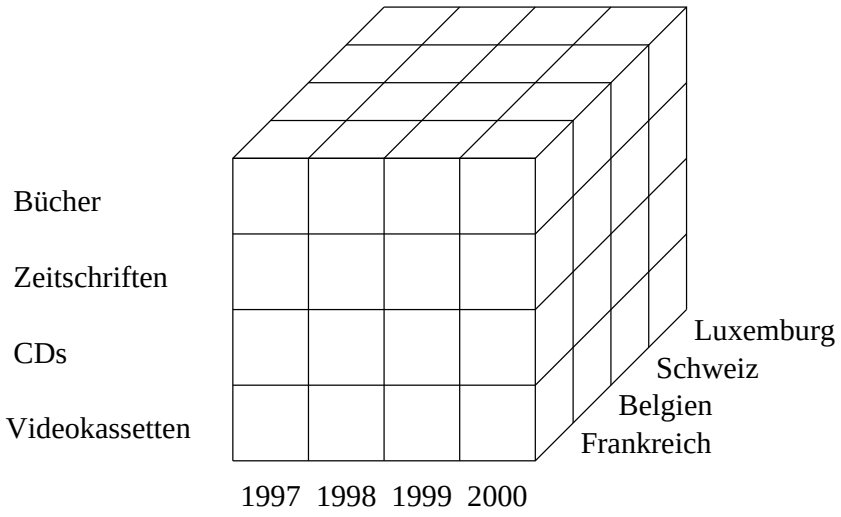


Abbildung 2.2: Datenwürfel

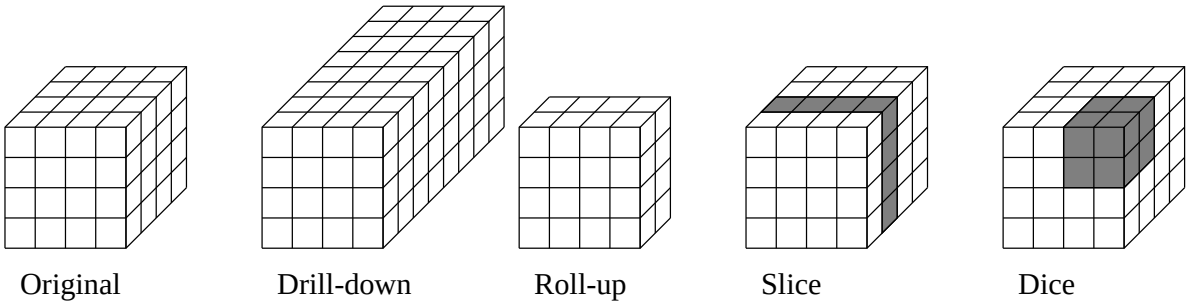


Abbildung 2.3: Operationen auf dem Datenwürfel

ten mit Büchern erzielten Umsatz in Frankreich im Jahr 1999. Die zu aggregierenden Werte werden jeweils durch Gruppierung der Dimensionen nach einem oder mehreren Attributen bestimmt. In Abbildung 2.2 ist ein dreidimensionaler Datenwürfel mit den Dimensionen Produkt, Verkaufszeitpunkt und Verkaufsort dargestellt. Dabei wurde die Dimension Produkt nach Produktkategorie, die Dimension Verkaufszeitpunkt nach Jahr und die Dimension Verkaufsort nach Land gruppiert.

Die meist verwendeten Operationen auf einem solchen Datenwürfel sind *Drill-down*, *Roll-up*, *Slice* und *Dice*.

- Beim drill-down wird die Anzahl der Attribute nach denen gruppiert wird für eine Dimension erhöht. In unserem Beispiel könnte man die Dimension Verkaufszeitpunkt nach Jahr *und* Monat gruppieren. Der dadurch entstehende Datenwürfel ermöglicht es dem Analysten die Verteilung des Gesamtumsatzes eines Jahres auf die einzelnen Monate zu

betrachten. [1]

- Roll-up ist die zu drill-down inverse Operation. Hier werden weniger Attribute zur Gruppierung verwendet. Im Beispiel könnte man das Attribut Produktkategorie weglassen und erhielte so die Gesamtumsatzzahlen der einzelnen Länder für die betrachteten Jahre.

- Slice ist eine Projektion auf eine Untermenge der Dimensionen für feste Werte der anderen Dimensionen. Im Beispiel könnte man nur die Umsatzzahlen für Frankreich betrachten.

- Dice ist die Selektion eines „Teilwürfels“. Im Beispiel ist dies eine Selektion der Produktkategorien Bücher und Videokassetten.

In Abbildung 2.3 sind die vier Operationen grafisch dargestellt.

2.4 MOLAP und ROLAP

Wenn der Datenwürfel in einem das mehrdimensionale Datenmodell direkt unterstützenden Datenbanksystem abgelegt ist, so spricht man von MOLAP (engl. *multidimensional OLAP*) und nennt das Datenbanksystem einen MOLAP Server. Ist er dagegen in einem relationalen Datenbanksystem gespeichert, spricht man von ROLAP (engl. *relational OLAP*) und ROLAP Servern. Obwohl MOLAP Server üblicherweise deutlich kürzere Antwortzeiten liefern, werden sie nur selten für Datenlager verwendet. Der Grund besteht darin, daß sie häufig nicht in der Lage sind die vorhandenen Datenmengen zu verarbeiten. Inmon [Inm96] schlägt als hybride Lösung vor, für das Datenlager ein relationales Datenbanksystem zu verwenden und zur Analyse Teile des Datenlagers in ein mehrdimensionales Datenbanksystem zu übertragen. Diese Lösung ist in der Praxis am häufigsten anzutreffen.

2.5 Sternschema und Schneeflockenschema

Falls die Analyse direkt auf dem relationalen Datenlager aufsetzen soll oder falls ein relationales Datenlager als Grundlage für MOLAP Server benutzt werden soll, muß das mehrdimensionale Datenmodell in das relationale Modell abgebildet werden. Eine Möglichkeit der Abbildung ist das *Sternschema* (vgl. Abbildung 2.4). In einem Sternschema gibt es eine *Faktentabelle*

[1] Häufig wird auch die Verfeinerung einer Dimension als *drill-down* (bzw. die Vergröberung einer Dimension als *roll-up*) bezeichnet. Die von uns gewählte Beschreibung der Operation ist allerdings allgemeiner und schränkt – wie das Beispiel zeigt – die Möglichkeiten der Operation nicht ein.

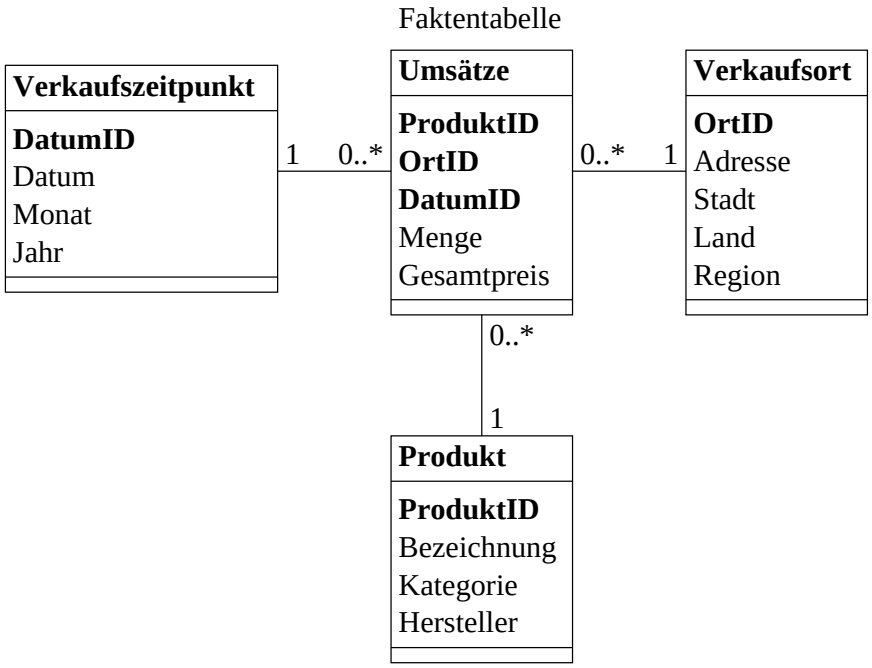


Abbildung 2.4: Sternschema

und für jede Dimension eine *Dimensionstabelle*. Die Attribute der Dimensionstabelle entsprechen den Attributen der Dimension. Jedes Tupel der Faktentabelle enthält die einer Kombination der Dimensionsattribute entsprechenden numerischen Werte, sowie die die Attributkombination beschreibenden Fremdschlüssel. Bei der Betrachtung des Sternschemas fällt auf, daß es – im Gegensatz zu den für OLTP-Anwendungen verwendeten Schemata – nicht normalisiert ist. Die Dimensionstabellen können, unabhängig davon, ob Abhängigkeiten zwischen den Attributen bestehen, alle Attributkombinationen enthalten. Daher gibt es als Verfeinerung des Sternschemas das *Schneeflockenschema*, das man durch Normalisierung der Dimensionstabellen erhält. Abbildung 2.5 zeigt das aus dem Sternschema in Abbildung 2.4 hervorgegangene Schneeflockenschema.

Durch die Normalisierung der Dimensionstabellen erhält man hier eine explizite Darstellung der Attributhierarchien. Ein entscheidender Nachteil des Schneeflockenschemas ist allerdings, daß zur Anfragebearbeitung zusätzliche Verbundoperationen (engl. *Join*) notwendig werden. Das führt zu einem Leistungsverlust bei den zentralen Aufgaben eines Datenlagers: schnelle Anfragebearbeitung und schnelles (inkrementelles) Laden von Daten. Daher ist das Sternschema dem Schneeflockenschema in der Regel vorzuziehen.

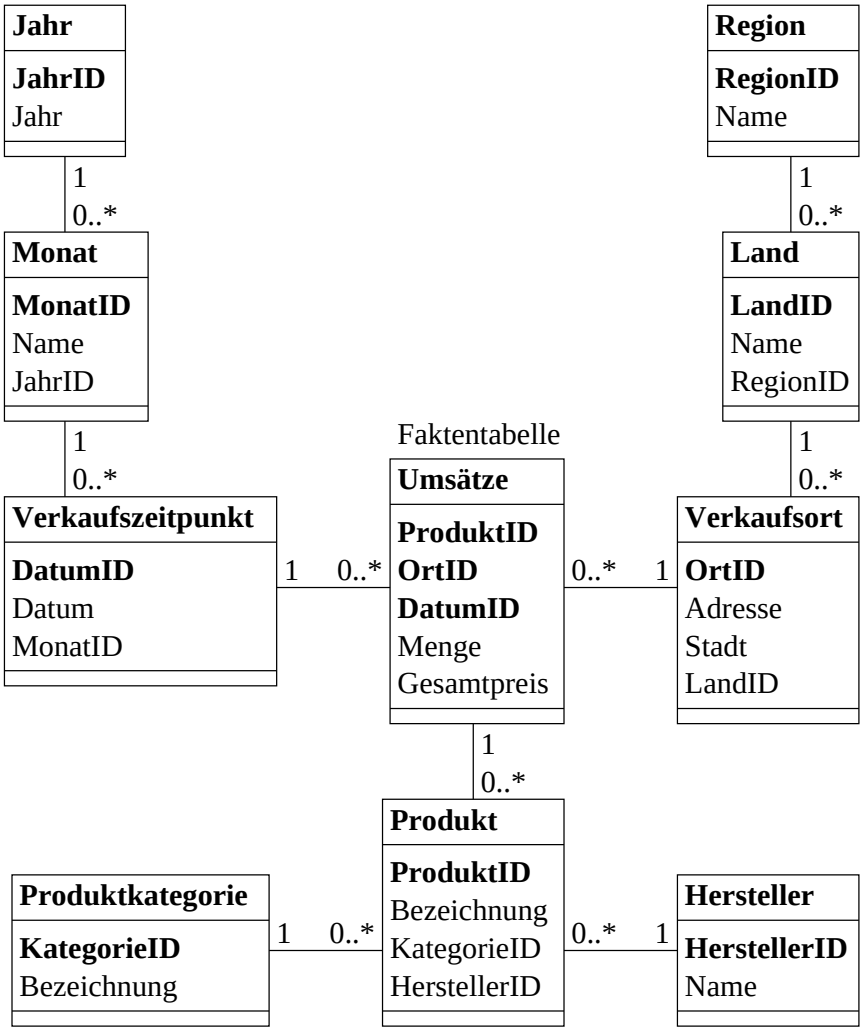


Abbildung 2.5: Schneeflockenschema

2.6 Anforderungen an ein Datenlagerverwaltungssystem

Aus den vorangegangenen Punkten ergeben sich folgende Anforderungen an ein zur Implementierung eines relationalen Datenlagers verwendetes Datenbankverwaltungssystem.

1. Zur Beantwortung der Anfragen werden in der Regel große Datenmengen benötigt. Daher muß das verwendete Datenbankverwaltungssystem entweder in der Lage sein große Datenmengen schnell zu lesen oder Verfahren bereitstellen, die die zu lesende Datenmenge reduzieren, oder beides.

2. Durch die Verwendung des Sternschemas sind Verbundoperationen zwischen der Fakten-

tabelle und den Dimensionstabellen von zentraler Bedeutung. Daher muß ein geeignetes Datenbankverwaltungssystem diese Operation effizient durchführen.

3. Nahezu alle Analysen verwenden Aggregation und/oder Gruppierung. Daher müssen diese Operationen in besonderer Weise unterstützt werden.

2.7 Inhalt dieser Arbeit und andere Ansätze

In dieser Arbeit stellen wir einige Techniken vor, die wir entwickelt haben, um den im letzten Abschnitt formulierten Anforderungen gerecht zu werden. Wir stellen diese Techniken und alternative Ansätze hier kurz vor.

Verarbeitung großer Datenmengen Wir verwenden *Kompression*, um die auf dem Sekundärspeicher abgelegte Datenmenge zu reduzieren. Damit wird auch der zur Übertragung der Daten vom Sekundärspeicher in den Primärspeicher erforderliche Aufwand reduziert und ein schnelles Lesen und Verarbeiten großer Datenmengen ermöglicht. Außerdem wird der zur Übertragung der Daten verwendete Puffer besser ausgenutzt.

Andere Möglichkeiten zur Verarbeitung großer Datenmengen bestehen zum Beispiel in der vertikalen Partitionierung der Basisrelationen, in der Verwendung materialisierter Sichten oder in der Verwendung spezieller Indexstrukturen. Die *vertikale Partitionierung* [RG99] der Basisrelationen führt – wie die Kompression – zu einer Reduktion der zu übertragenden Datenmenge. Der Nachteil dieses Verfahrens besteht allerdings darin, daß viele zusätzliche Verbundoperationen notwendig sind, um die getrennten Daten wieder zusammenzuführen. *Materialisierte Sichten* [GM99] werden verwendet, um den Zugriff auf die Basisrelationen möglichst vollständig zu vermeiden und statt dessen auf zusätzlich angelegte (kleinere oder für die Anfrage spezifischere) Relationen zuzugreifen, die zum Beispiel voraggregierte Werte enthalten. Das zentrale Problem bei der Verwendung materialisierter Sichten besteht darin diese stets auf dem aktuellen Stand zu halten. *Spezielle Indexstrukturen*, wie zum Beispiel SMAs [Moe98], werden zum einen – wie materialisierte Sichten – dazu verwendet, den Zugriff auf die Basisrelationen zu vermeiden, und zum anderen – wie bei Indexstrukturen üblich – um nur auf kleine, relevante Bereiche der Basisrelationen zuzugreifen.

Die beschriebenen Verfahren schließen sich allerdings gegenseitig nicht aus, so daß durchaus auch Kombinationen der Verfahren verwendet werden können. So kann zum Beispiel die Verwendung einer vertikalen Partitionierung die Vorteile der Kompression durchaus verstärken.

Verbund zwischen der Fakten- und den Dimensionstabellen Um die Verbundoperation zwischen der Fakten- und den Dimensionstabellen zu beschleunigen nutzen wir die Tatsache

aus, daß Fakten- und den Dimensionstabellen in der Regel – implizit – nach dem Zeitpunkt des Einfügens der Tupel geballt sind (vgl. Kapitel 5). Wir haben einen Operator entwickelt, der, wenn diese Ballung vorliegt, eine sehr effiziente Berechnung des Verbunds ermöglicht.

Alternativen hierzu sind spezielle Indexstrukturen, die die Berechnung des Verbunds unterstützen [Val87], wie zum Beispiel Bitvektor-Verbund-Indexstrukturen [OG95]. Bei diesen Verfahren werden Bitvektoren verwendet, um nicht qualifizierende Tupel vor der Verbundoperation herauszufiltern und so den Aufwand für die Verbundoperation zu reduzieren. Es ist allerdings zum einen nicht immer möglich, eine große Tupelmenge herauszufiltern, und zum anderen können die Bitvektoren bei Verbundattributen, die eine große Anzahl unterschiedlicher Werte annehmen, sehr groß werden, so daß der mit der Nutzung der Indexstruktur verbundene Aufwand zu groß wird.

Effiziente Gruppierungen und Aggregationen Zur effizienten Unterstützung von Gruppierung und Aggregation haben wir Spezialfälle dieser Operationen identifiziert, die in Datenlagern häufig benötigt werden und die eine effizientere Auswertung, als dies bei herkömmlichen Datenbankverwaltungssystemen der Fall ist, ermöglichen. Wir haben auch hier Operatoren implementiert, die diese effizientere Auswertung realisieren.

Die effiziente Durchführung von Gruppierung und Aggregation kann auch – wie oben schon angedeutet – mit Hilfe materialisierter Sichten oder mit Hilfe spezieller Indexstrukturen unterstützt werden. Bei der Verwendung materialisierter Sichten werden bestimmte Aggregate vorausberechnet, die dann bei der Anfragebearbeitung genutzt werden können. Hierzu muß dann ermittelt werden, welche Aggregate bei der Anfragebearbeitung möglichst flexibel und effizient genutzt werden können [HRU96]. Spezielle Indexstrukturen wie zum Beispiel SMAs können zum einen direkt aggregierte Werte zur Verfügung stellen und zum anderen die Suche nach den zu aggregierenden Werten in den Basisdaten erheblich beschleunigen [Moe98].

Da unsere Techniken die Verwendung von Indexstrukturen nicht ausschließen sind auch hier erfolgversprechende Kombinationen möglich.

Im folgenden Kapitel werden wir zunächst das Laufzeitsystem AODB vorstellen, bevor wir in den Kapiteln 4 und 5 die oben erwähnten Techniken darstellen.

Kapitel 3

AODB

In diesem Kapitel wird das Laufzeitsystem AODB beschrieben. Die in den beiden folgenden Kapiteln 4 und 5 beschriebenen Techniken wurden in das hier beschriebene System integriert und im Rahmen dieses Systems experimentell evaluiert. Nach der Einleitung und der Beschreibung der Entwurfsziele für das System in Abschnitt 3.1 beschreiben wir in Abschnitt 3.2 die Architektur und in den Abschnitten 3.3–3.6 die einzelnen Komponenten des Systems.

3.1 Einleitung

3.1.1 Anfragebearbeitung

Die Bearbeitung einer Anfrage in einem relationalen Datenbanksystem kann grob in zwei Schritte unterteilt werden. Im ersten Schritt wird die deklarative Anfrage, zum Beispiel in SQL [ISO97], vom Anfrageübesetzer mit Hilfe der vorhandenen Metadaten in einen Auswertungsplan übersetzt. Im zweiten Schritt wird dieser Auswertungsplan dann vom Laufzeitsystem bezüglich der in der Datenbank gespeicherten Daten ausgewertet (vgl. Abbildung 3.1).

Ein Auswertungsplan beschreibt die Art und die Reihenfolge der zur Beantwortung einer Anfrage notwendigen Schritte. Das Ziel des Anfrageübersetzers ist, einen zu der gegebenen Anfrage äquivalenten Auswertungsplan zu erzeugen, der bei der Auswertung durch das Laufzeitsystem möglichst wenig Ressourcen benötigt. Dazu muß der Anfrageübersetzer die vom Laufzeitsystem zur Verfügung gestellten Bausteine zur Formulierung eines Auswertungsplans und deren Kosten kennen. Um zu einem leistungsfähigen Gesamtsystem zu gelangen, sollte das Laufzeitsystem effiziente, den Benutzeranforderungen entsprechende Bausteine zur Verfügung stellen.

AODB ist ein Laufzeitsystem für relationale Datenbankverwaltungssysteme, das die Anforderungen von OLAP-Anwendungen in besonderer Weise berücksichtigt, also ein Laufzeitsystem

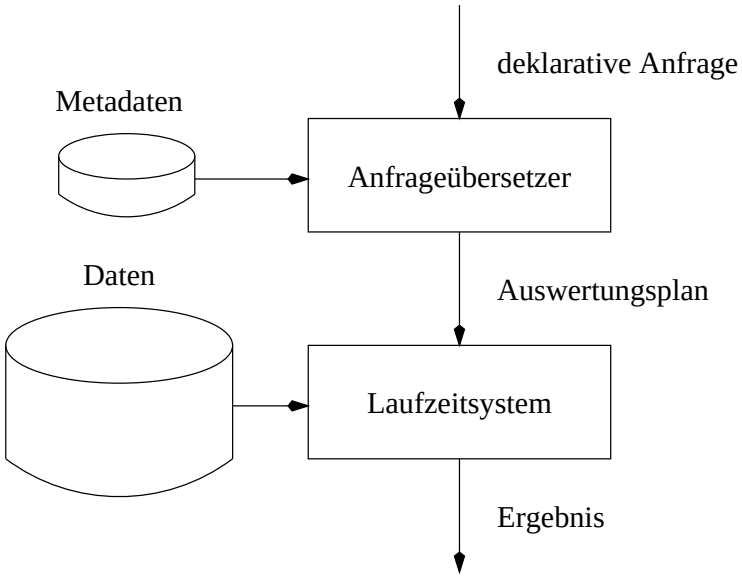


Abbildung 3.1: Anfragebearbeitung

für ein relationales Datenlagerverwaltungssystem. Wegen seiner flexiblen Architektur ist es aber auch als Grundlage für andere relationale Datenbankverwaltungssysteme geeignet. Es existiert zur Zeit ein Forschungsprototyp, der zur Bewertung verschiedener leistungssteigernder Techniken entwickelt wurde.

3.1.2 Entwurfsziele

Die Haupttätigkeit – und damit auch der Hauptgrund für die Ressourcennutzung – des Laufzeitsystems eines Datenbankverwaltungssystems besteht im Kopieren von Daten. Einerseits werden die zu verarbeitenden Daten vom Sekundärspeicher in den Primärspeicher kopiert, andererseits werden sie aber auch im Laufe der Anfragebearbeitung mehrfach im Primärspeicher kopiert und eventuell sogar wieder auf den Sekundärspeicher ausgelagert. Ziel der Entwicklung eines Laufzeitsystems muß es daher sein, den Kopieraufwand auf ein Minimum zu reduzieren. Obwohl es schon relativ früh in der Entwicklung relationaler Datenbanken die Erkenntnis gab, daß ein leistungsfähiges Datenbankverwaltungssystem die Balance zwischen E/A-Last und Prozessorlast halten muß [CAB+81], hat sich die Forschung bisher fast ausschließlich mit der Reduktion der E/A-Last beschäftigt. Beim Entwurf und bei der Konstruktion von AODB war unser Ziel daher die vorhandenen Erkenntnisse zur Reduktion der E/A-Last zu nutzen und neue Ansätze zur Reduktion der Prozessorlast zu finden.

Im folgenden Abschnitt 3.2 wird die Architektur des Systems aus der Sicht eines Nutzerprozesses beschrieben. Anschließend werden in den Abschnitten 3.3–3.6 die einzelnen Komponenten

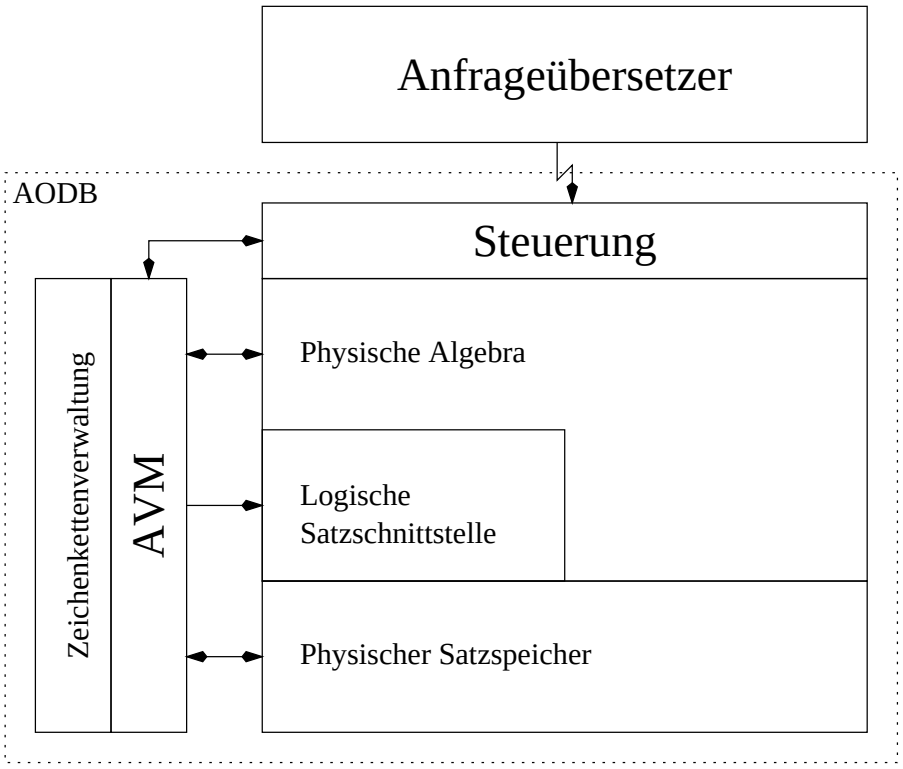


Abbildung 3.2: Architektur von AODB

des Systems dargestellt.

3.2 Architektur

In Abbildung 3.2 ist die Architektur von AODB schematisch dargestellt. Das System erhält als Eingabe Auswertungspläne von einem Anfrageübersetzer und liefert einen Tupelstrom als Ausgabe. Sowohl Ein- als auch Ausgabe erfolgen in textueller Form und sind somit architektur- und betriebssystemunabhängig. Die Umsetzung zwischen der textuellen Darstellung und der internen Darstellung erfolgt durch die als *Steuerung* bezeichnete Komponente.

Das System besteht aus vier Hauptkomponenten:

- Die Komponente *physischer Satzspeicher* enthält alle Systembestandteile, die zur Speicherung und zum Wiederauffinden physischer Sätze benötigt werden. Ein physischer Satz ist dabei eine Folge von Bytes, deren Semantik innerhalb des physischen Satzspeichers nicht bekannt ist. Abschnitt 3.3 enthält eine detailliertere Beschreibung des physischen Satzspeichers.

- Die *logische Satzschnittstelle* besteht aus den Systembestandteilen, die zur Interpretation der in den physischen Sätzen abgelegten Informationen benötigt werden. Sie wird in Abschnitt 3.4 beschrieben.
- AVM, die *AODB Virtual Machine*, wird zur Verarbeitung aller typabhängiger Information in AODB verwendet. Ihre Aufgaben, ihr Aufbau und ihre Funktion werden in Abschnitt 3.5 beschrieben.
- Die Komponente *physische Algebra* enthält die zur Auswertung der SQL-Anweisungen verwendeten (relationen-)algebraischen Operatoren. Deren Beschreibung befindet sich in Abschnitt 3.6.

3.3 Physischer Satzspeicher

In diesem Abschnitt werden die Strukturen beschrieben, mit deren Hilfe in AODB einzelne (physische) Sätze effizient auf sekundären Speichermedien abgelegt, aufgefunden und genutzt werden. Dazu betrachten wir zunächst in Abschnitt 3.3.1 die generelle Organisation des Sekundärspeichers, anschließend in den Abschnitten 3.3.2 und 3.3.3 die im Sekundärspeicher verwendeten Objekte und dann in Abschnitt 3.3.4 die Organisation des Systempuffers, der Schnittstelle zwischen Primär- und Sekundärspeicher. Im folgenden werden Sekundärspeicherdatenstrukturen detaillierter als Primärspeicherdatenstrukturen beschreiben, da eine Sekundärspeicherdatenstruktur üblicherweise das Ergebnis einer Designentscheidung ist, wohingegen eine Primärspeicherdatenstruktur sich oft aus den verwendeten Werkzeugen ergibt.

3.3.1 Organisation des Sekundärspeichers

Der gesamte Sekundärspeicher wird in AODB als eine Menge von Partitionen aufgefaßt. Eine *Partition* ist ein linearer Adreßraum fester Größe mit sichtbaren Seitengrenzen, der üblicherweise einem Festplattenlaufwerk oder einer Partition auf einem Festplattenlaufwerk entspricht. Die Größe der *Seiten* ist in jeder Partition konstant und ist in der Regel ein ganzzahliges Vielfaches der Blockgröße des verwendeten Festplattenlaufwerks. Zusammengehörende Seiten werden in *Segmenten* zusammengefaßt. Um den Aufwand zur Verwaltung der Segmente zu begrenzen, ist ein Segment als eine Menge von Extents organisiert (vgl. 3.3 in [HR99]). Ein *Extent* ist eine Menge von physisch aufeinanderfolgenden Seiten, der durch die erste Seite und die Anzahl der zum Extent gehörenden Seiten spezifiziert werden kann.

Eine Datenbank besteht – bezüglich der auf dem Sekundärspeicher abgelegten Objekte – aus einer Menge von *Relationen* und *Indexstrukturen*. Jede Relation oder Indexstruktur besteht aus mehreren *physischen Sätzen*, die auf den Seiten eines oder mehrerer Segmente gespeichert sind. Ein Segment enthält aber nie Seiten unterschiedlicher Relationen oder Indexstrukturen. Die

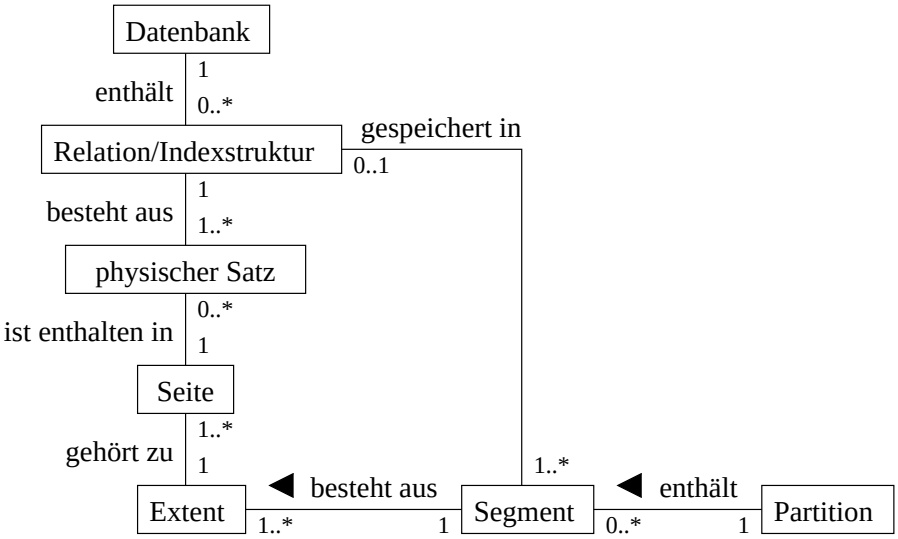


Abbildung 3.3: Organisation des Sekundärspeichers

Zusammenhänge zwischen den Begriffen sind in Abbildung 3.3 in UML-Notation [RJB99] grafisch dargestellt.

3.3.2 Physische Sätze und Seiten

3.3.2.1 Eigenschaften

Wie schon in Abschnitt 3.2 erwähnt ist ein physischer Satz eine Folge von Bytes, die bezüglich des Anwendungsbereichs keine Semantik besitzt. In AODB muß jeder physische Satz vollständig auf einer Seite abgelegt werden, d.h. die Größe eines physischen Satzes kann die Seitengröße der betreffenden Partition nicht übersteigen.

Zur Unterstützung unterschiedlicher Anforderungen kann der Inhalt einer Seite in unterschiedlicher Weise strukturiert werden. Die Art der Strukturierung wird als *Seitentyp* bezeichnet. Die Seitentypen unterscheiden sich in folgenden voneinander unabhängigen Eigenschaften:

- Vorhandensein von Verwaltungsstrukturen für variabel lange Sätze
- Möglichkeit einzelne Sätze zu löschen oder zu verschieben
- Unterstützung der Wiederherstellung im Fehlerfall

Die in Tabelle 3.1 aufgeführten Kombinationen sind in AODB vorhanden[1]. Andere Seitentypen werden zur Zeit zum Beispiel zur Implementierung verschiedener Indexstrukturen entwickelt.

[1]Tatsächlich sind die Seiten der Typen *Append-Only Slotted Page* und *Recoverable Slotted Page* in je zwei

	Plain Page	Recoverable Plain Page	Append-Only Slotted Page	Recoverable Slotted Page
variable Länge	nein	nein	ja	ja
löschen	nein	nein	nein	ja
Wiederherstellung	nein	ja	nein	ja

Tabelle 3.1: Seitentypen

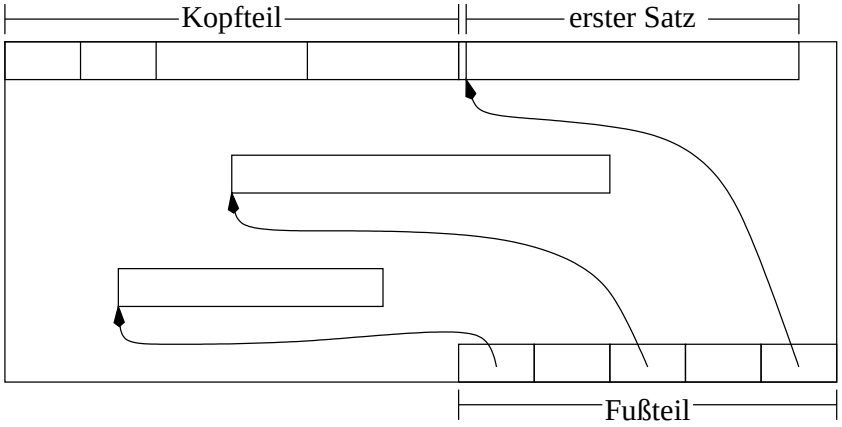


Abbildung 3.4: Seitenaufbau

3.3.2.2 Aufbau

Der prinzipielle Aufbau der Seiten ist in Abbildung 3.4 dargestellt (vgl. auch [GR93]). Die Verwaltungsinformation ist auf einen Kopf- und einen Fußteil verteilt. Der Kopfteil enthält alle Informationen, die in einem Speicherbereich festen Länge abgelegt werden können. Die Länge des Kopfteils ist daher für alle Seiten eines Seitentyps konstant. Beispiele für Informationen, die im Kopfteil abgelegt werden, sind der Seitentyp, die Länge des Fußteils oder auch eine Prüfsumme, die zur Wiederherstellung im Fehlerfall verwendet wird. Der Fußteil besteht aus mehreren Verwaltungssätzen fester Länge, die Informationen über die auf der Seite abgelegten Sätze variable Länge enthalten. Dies sind unter anderem die Länge und ein Zeiger auf den Anfang des variablen Satzes. Die Anzahl der im Fußteil vorhandenen Verwaltungssätze hängt von der Anzahl der auf der Seite abgelegten Sätze und damit von der Größe dieser Sätze ab. Die Länge des Fußteils ist also variabel. Falls eine Seite keine variabel langen Sätze enthält, wie zum Beispiel im Fall der *Recoverable Plain Page*, kann der Fußteil auch wegfallen. Im Sonderfall der *Plain Page* ist auch kein Kopfteil vorhanden.

Versionen vorhanden. Eine Version ordnet die Sätze möglichst platzsparend an, und die andere Version richtet die Sätze an 8 Byte Grenzen aus, um die Verarbeitung im Primärspeicher zu beschleunigen (vgl. auch Abschnitt 3.4)

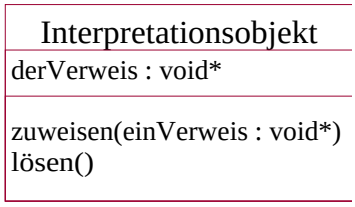


Abbildung 3.5: Minimale Schnittstelle eines Interpretationsobjekts

3.3.2.3 Implementierung

Der Zugriff auf die Informationen einer Seite erfolgt durch *Interpretationsobjekte*. Betrachten wir daher zunächst das

Interpretationsobjekte-Muster . Bei der Implementierung von Datenbankverwaltungssystemen ist es häufig erforderlich im Systempuffer vorhandene Sekundärspeicherdatenstrukturen zu interpretieren. Die einfachste Lösung, die Daten in eine entsprechende Primärspeicherdatenstruktur zu überführen (d.h. zu kopieren), ist allerdings sehr teuer. Wir verwenden daher zu diesem Zweck *Interpretationsobjekte*.

Interpretationsobjekte enthalten das zur Interpretationen einer Sekundärspeicherdatenstruktur notwendige Wissen über den Aufbau dieser Struktur. Um eine Sekundärspeicherdatenstruktur zu interpretieren, wird der entsprechende Speicherbereich im Systempuffer dem Interpretationsobjekt mit *zuweisen* (engl. *attach*) zugewiesen (vgl. auch Abbildung 3.5). Alle Methoden des Interpretationsobjekts beziehen sich dann auf den zugewiesenen Speicherbereich. Der aktuelle Zustand des Objekts ergibt sich also aus diesem Speicherbereich und eventuell aus Parametern, die den Aufbau der Sekundärspeicherdatenstruktur bestimmen. Die Datenelemente eines Interpretationsobjekts bestehen also aus einem Verweis auf den gerade betrachteten Speicherbereich und aus den Parametern.

Da die Parameter sich während der Lebensdauer eines Interpretationsobjekts nicht ändern und mit *zuweisen* jederzeit ein neuer Speicherbereich zugewiesen werden kann, können nacheinander unterschiedliche Speicherbereiche mit Hilfe eines Interpretationsobjekts zu interpretiert werden. Die Lebensdauer der Interpretationsobjekte übersteigt daher in der Regel die Lebensdauer der zu interpretierenden Speicherbereiche im Systempuffer bei weitem.

Es ist zur Leistungssteigerung auch möglich, abgeleitete Informationen nicht bei jeder Verwendung neu zu bestimmen, sondern sie in Datenelementen des Interpretationsobjekts zwischenzuspeichern. Diese müssen dann allerdings bei einem erneuten *zuweisen* aktualisiert werden.

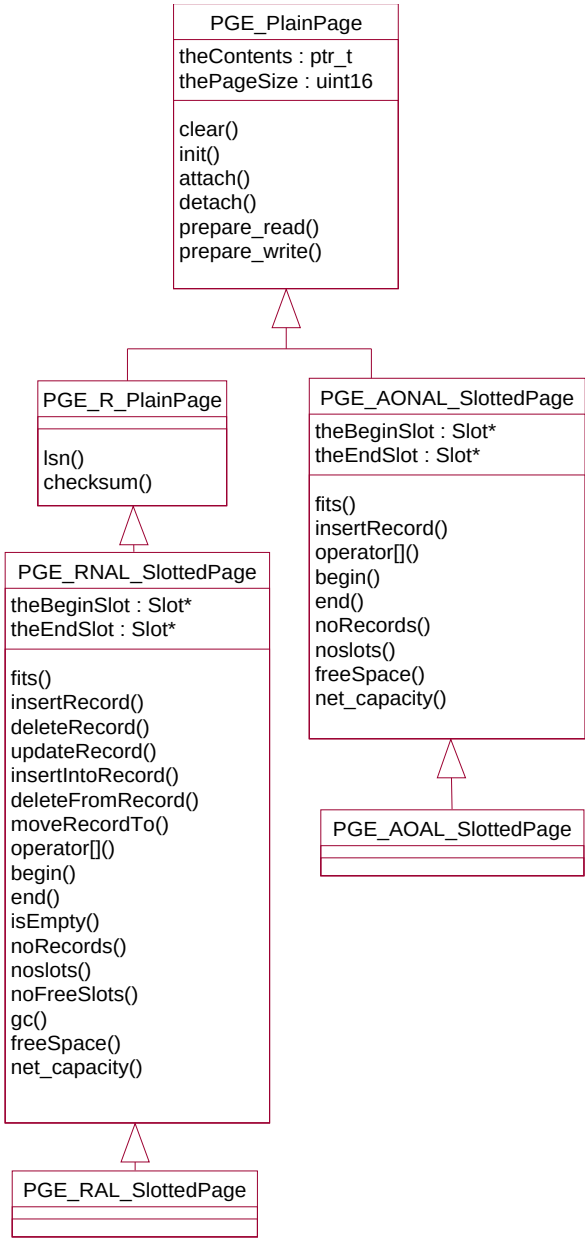


Abbildung 3.6: Schnittstelle der Seitenobjekte

Im Falle der Seiten bestehen die Datenelemente der Interpretationsobjekte aus einem Verweis auf den Seiteninhalt und aus einem Parameter, der Seitengröße (vgl. auch Abbildung 3.6).

Zur Unterscheidung zwischen dem Interpretationsobjekt und dem Bereich im Systempuffer, der den Zustand enthält, sprechen wir im ersten Fall vom *Seitenobjekt* und im zweiten Fall von der *Seite*.

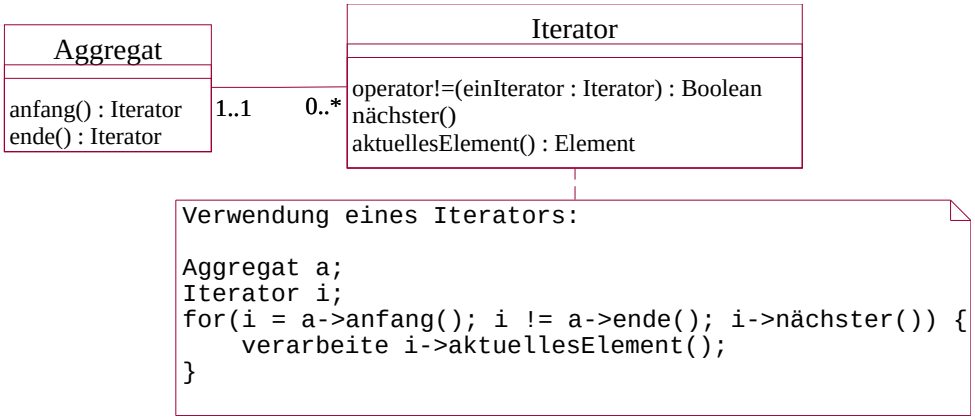


Abbildung 3.7: Iterator-Muster

Die von dem jeweiligen Seitenobjekt zur Verfügung gestellte Schnittstelle hängt von den auf der Seite vorhandenen Verwaltungsstrukturen ab. Einen Überblick gibt Abbildung 3.6, in der die den beschriebenen Seitentypen entsprechenden Klassen in UML-Notation dargestellt sind.

Da in einigen Seitenobjekten zur Leistungssteigerung abgeleitete Informationen im Seitenobjekt zwischengespeichert werden, kann die Ausführung der Methode *zuweisen* relativ aufwendig werden. Daher existieren üblicherweise mehrere Seitenobjekte des gleichen Typs gleichzeitig. (Genauer existiert für jede Seite im Systempuffer genau ein Seitenobjekt. Dieses Seitenobjekt ist jeweils von dem gleichen Typ wie die entsprechenden Seite.)

Falls ein Seitentyp die Verwaltung variabel langer Sätze unterstützt, stellt das entsprechende Seitenobjekt Methoden zum Zugriff auf die Sätze zur Verfügung. Dieser Zugriff kann zum einen direkt, d.h. über die Nummer des Satzes, oder über *Iteratoren* erfolgen.

Allgemein ist das *Iterator-Muster* (vgl. Abbildung 3.7 und [GHJV95]) eine Möglichkeit sequentiell auf die einzelnen Elemente eines aggregierten Objekts zuzugreifen, ohne die Struktur des aggregierten Objekts zu kennen. Das aggregierte Objekt besitzt dazu zwei Methoden *anfang* und *ende*. Die Methode *anfang* gibt dabei einen Iterator zurück, dessen aktuelles Element das erste Element des aggregierten Objekts ist. Die Methode *ende* gibt einen Iterator zurück, der zur Überprüfung, ob ein Iterator bereits alle Elemente betrachtet hat, verwendet werden kann. Der Iterator besitzt den *operator!=* zur Überprüfung, ob das Ende bereits erreicht wurde, die Methode *nächster*, um das nächste Element des aggregierten Objekts zu betrachten, und die Methode *aktuellesElement*, um auf das aktuelle Element zuzugreifen.

Die von den Seitenobjekte zur Verfügung gestellten Iteratoren ermöglichen dementsprechend den sequentiellen Zugriff auf alle Sätze einer Seite, wobei der Benutzer des Iterators den Seitentyp nicht kennen muß. Die Schnittstelle des Iterators ist für alle Seitentypen gleich.

3.3.3 Segmente und Partitionen

3.3.3.1 Eigenschaften

Obwohl prinzipiell Seiten beliebiger Typen in einem Segment zusammengefaßt werden können, sind die Segmente in der Regel homogen, da die Eigenschaften der Seiten nur dann sinnvoll genutzt werden können, wenn sie von allen Seiten eines Segments unterstützt werden. Dementsprechend gibt es analog zu den Seitentypen Segmenttypen, die eine bestimmte Kombination von Eigenschaften beschreiben. Ein Basisrelation wird zum Beispiel in einem oder mehreren Segmenten des Typs *Slotted Page Segment* gespeichert, das Seiten des Typs *Recoverable Slotted Page* enthält. Für eine während der Anfragebearbeitung temporär angelegte Relation genügt hingegen ein *Append Only Segment*. Ein solches Segment besteht aus Seiten des Typs *Append-Only Slotted Page* und unterstützt daher weder eine Wiederherstellung im Fehlerfall noch das Löschen einzelner Sätze. Die bisher vorhandenen Segmenttypen sind in Abbildung 3.11 auf Seite 28 dargestellt.

Zur Verwaltung der zu einer Partition gehörenden Segmente und Seiten gibt es in jeder Partition drei spezielle Segmente: das *Master-Segment*, das *FreeExtent-Segment* und das *FreeSpaceInventory-Segment*. Das Master-Segment enthält spezielle Sätze, die *Master-Records*. Die Master-Records beschreiben die in der Partition gespeicherten Segmente und zwar sowohl die von Benutzern angelegten als auch die drei speziellen Verwaltungssegmente. Das FreeExtent-Segment beschreibt, welche Extents zum gegebenen Zeitpunkt zu keinem Benutzer-Segment gehören. Das FreeSpaceInventory-Segment enthält Informationen über den Belegungsgrad aller Seiten der Partition. Diese Information wird verwendet, um Seiten eines bestimmten Belegungsgrads schnell zu finden. Da die exakte Beschreibung des Belegungsgrads aller Seiten allerdings zu viel Platz benötigen würde, enthält das FreeSpaceInventory-Segment ungenaue Informationen, die bei der Suche nach einer passenden Seite nur als Filter verwendet werden können.

3.3.3.2 Aufbau

Ein Segment besteht aus einer Menge von Seiten, die in Extents zusammengefaßt sind. Alle ein Segment beschreibenden Informationen wie zum Beispiel der Segmenttyp, der Segmentname oder die zu dem Segment gehörenden Extents werden im Master-Segment abgelegt. Ein Segment wird jeweils durch zwei Master-Records beschrieben, einen *Segment-Deskriptor* und eine *Extent-Tabelle* (vgl. Abbildung 3.8).

Ein Segment-Deskriptor enthält alle zur Beschreibung eines Segments notwendigen Informationen fester Länge, sowie einen Verweis auf die Extent-Tabelle. Dieser Verweis wird durch die Aufnahme des systemweit eindeutigen *Satzidentifikators* (*RID* für engl. *record identifier* oder *row identifier*) der Extent-Tabelle in den Segment-Deskriptor realisiert. Ein solcher Satzi-

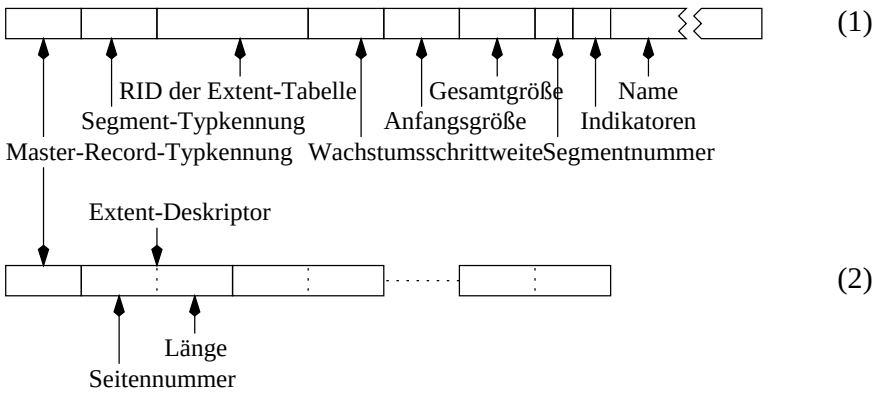


Abbildung 3.8: Master-Records: Segment-Deskriptor (1) und Extent-Tabelle (2)

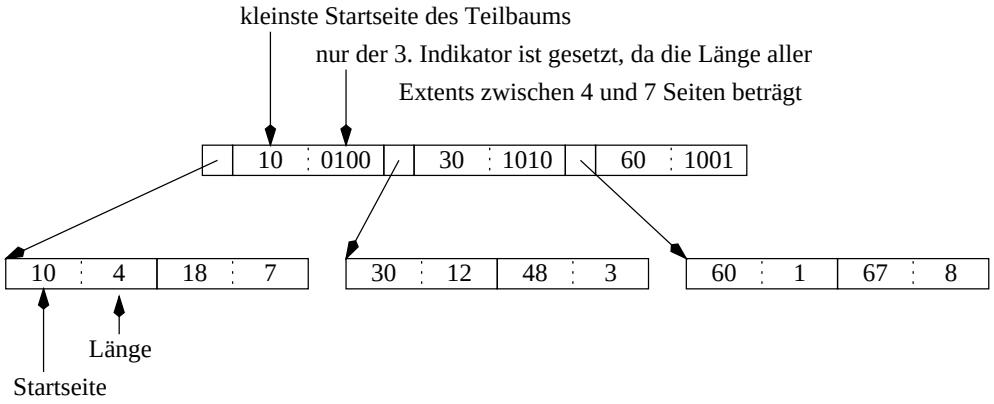


Abbildung 3.9: Aufbau des FreeExtent-Segments

dentifikator besteht aus einer Partitionsnummer, der Seitennummer innerhalb der Partition und Satznummer auf der Seite.

Eine Extent-Tabelle enthält eine variable Anzahl *Extent-Deskriptoren*. Ein Extent-Deskriptor besteht jeweils aus der Nummer der ersten Seite und der Länge des Extents. Die Extent-Deskriptoren sind innerhalb einer Tabelle aufsteigend nach Startseiten sortiert. Die Größe der Extent-Tabelle kann die Größe einer Seite nicht überschreiten. Falls dies erforderlich sein sollte, muß das Segment reorganisiert werden. Durch die Trennung der Extent-Tabelle vom Segment-Deskriptor kann sichergestellt werden, daß ein einmal angelegter Segment-Deskriptor nie verschoben werden muß. Falls eine Extent-Tabelle aufgrund ihres Wachstums nicht mehr auf ihre ursprüngliche Seite paßt, muß sie auf eine andere Seite verschoben werden. Die Position des Segment-Deskriptors bleibt davon unberührt. Daher kann jedes Segment systemweit eindeutig durch den Satzidentifikator seines Segment-Deskriptors identifiziert werden.

Das FreeExtent-Segment enthält die Extent-Deskriptoren aller freien Extents. Diese Extent-Deskriptoren sind je Seite, wie in den Extent-Tabellen, aufsteigend nach der ersten Seite des

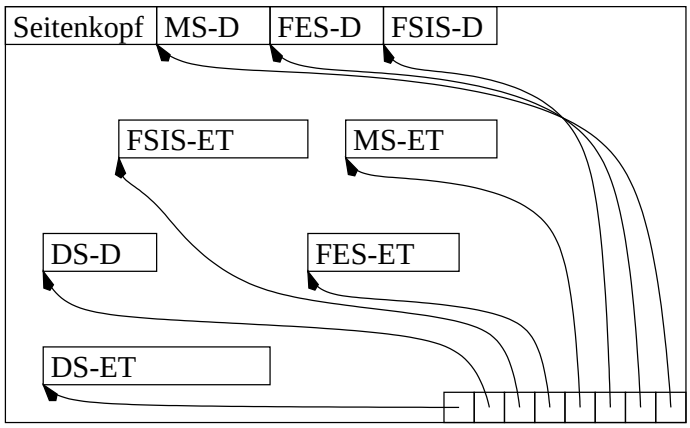
Extents sortiert. Falls eine Seite nicht genug Platz für alle freien Extent-Deskriptoren zur Verfügung stellt, verwendet das FreeExtent-Segment eine B-Baum-ähnliche Struktur (vgl. [Com79]), die die Startseite als Schlüssel verwendet. Dabei enthalten die Blattseiten die Deskriptoren der freien Extents und die inneren Seiten spezielle Deskriptoren, die aus zwei Teilen bestehen. Der erste Teil enthält die kleinste Startseite des zugrundeliegenden Teilbaums. Der zweite Teil enthält n Indikatoren (engl. *flags*). Der erste Indikator gibt an, ob der Teilbaum einen Extent mit einer freien Seite enthält, der zweite Indikator gibt an, ob der Teilbaum einen Extent mit mindestens zwei und weniger als vier freien Seiten enthält, der i-te Indikator gibt an, ob der Teilbaum einen Extent mit mindestens 2^i und weniger als 2^{i+1} freien Seiten enthält und der n-te Indikator gibt an, ob der Teilbaum einen Extent mit mehr als 2^n freien Seiten enthält. In Abbildung 3.9 ist ein Beispiel eines Baums der Höhe 2 mit $n = 4$ dargestellt. Mit Hilfe der im FreeExtent-Segment enthaltenen Information können freie Extents, die zum Vergrößern eines Segments benötigt werden, schnell gefunden werden.

Das FreeSpaceInventory-Segment enthält für jede Seite in der Partition 4 Bit, in denen der Belegungsgrad der Seite kodiert wird. Die Größe des FreeSpaceInventory-Segments wird daher zum Zeitpunkt der Erzeugung der drei Verwaltungssegmente einmalig festgelegt, und sie wird nicht mehr verändert. Die Bedeutung der vier Bit kann jeweils von dem Segment, dem die Seite angehört, frei festgelegt werden. Bei dieser Festlegung muß lediglich die feststehende Bedeutung der Werte 0000 (Seite nicht initialisiert) und 1111 (Seite kann keine weiteren Daten aufnehmen) beachtet werden. Da der exakte Zustand einer Seite in der Regel nicht durch vier Bit dargestellt werden kann, können die Informationen des FreeSpaceInventory-Segments nur als Hinweise interpretiert werden, die vor der Verwendung noch überprüft werden müssen. Da es sich bei dem Belegungsgrad um eine abgeleitete Größe handelt, die jederzeit wiederhergestellt werden kann, ist es nicht notwendig, daß das FreeSpaceInventory-Segment eine Wiederherstellung im Fehlerfall unterstützt. Daher werden hier Seiten vom Typ *Plain Page* verwendet.

Durch Konvention ist festgelegt, daß die erste Seite des Master-Segments stets auf der Seite 0 einer Partition abgelegt wird. Diese Seite enthält die die Segment-Deskriptoren und Extent-Tabellen des Master-Segments, der FreeExtent-Segments und des FreeSpaceInventory-Segments, die durch die Segmentnummern 1, 2 und 3 eindeutig identifiziert werden können (vgl. Abbildung 3.10).

3.3.3.3 Implementierung

Die zur Darstellung der unterschiedlichen Segmenttypen verwendeten Klassen sind in Abbildung 3.11 zusammengefaßt. Die allen gemeinsame Funktionalität der Verwaltung der zu Extents zusammengefaßten Seiten ist in der Basisklasse *SEG_Segment* implementiert. Obwohl prinzipiell alle dazu notwendigen Information dem Segment-Deskriptor entnommen werden kann, werden einige häufig verwendete Informationen wie zum Beispiel die RID der Extent-Tabelle in den (transienten) Instanzen von *SEG_Segment* zwischengespeichert.



MS-D	Master-Segment-Deskriptor
MS-ET	Master-Segment-Extent-Tabelle
FES-D	FreeExtent-Segment-Deskriptor
FES-ET	FreeExtent-Segment-Extent-Tabelle
FSIS-D	FreeSpaceInventory-Segment-Deskriptor
FSIS-ET	FreeSpaceInventory-Segment-Extent-Tabelle
DS-D	Daten-Segment-Deskriptor
DS-ET	Daten-Segment-Extent-Tabelle

Abbildung 3.10: Die erste Seite (des Master-Segments) einer Partition

Der Zugriff auf die einzelnen Seiten eines Segments erfolgt – ebenso wie der Zugriff auf die einzelnen Sätze einer Seite – mit Hilfe von *Iteratoren* (vgl. Darstellung in Abschnitt 3.3.2.3 auf Seite 23). Die Segment-Iteratoren iterieren allerdings nicht über die Seiten, sondern über die *Seitenidentifikatoren* (*PID* für engl. *page identifier*) der zum Segment gehörenden Seiten. Ein Seitenidentifikator besteht aus einer Partitionsnummer und Seitennummer innerhalb der Partition (entspricht also einem Satzidentifikator ohne die Satznummer). Zur Bestimmung der Seitenidentifikatoren müssen lediglich die ein Segment beschreibenden Master-Records betrachtet werden, so daß ein Zugriff auf die Seiten nicht erforderlich ist. In Abbildung 3.12 sind die zur Verwaltung der Seiten einer Partition benötigten Klassen dargestellt.

Um mit Hilfe der Seitenidentifikatoren auf die Seiten selbst und damit auch auf den Seiteninhalt zuzugreifen, wird der im folgenden Abschnitt beschriebene Systempuffer verwendet.

3.3.4 Organisation des Systempuffers

3.3.4.1 Aufgabe und Zielsetzung

Generell ist die Aufgabe des Systempuffers den höheren Schichten des Datenbankverwaltungssystems die zu bearbeitenden Datenobjekte mit möglichst geringem Aufwand im Primärspei-

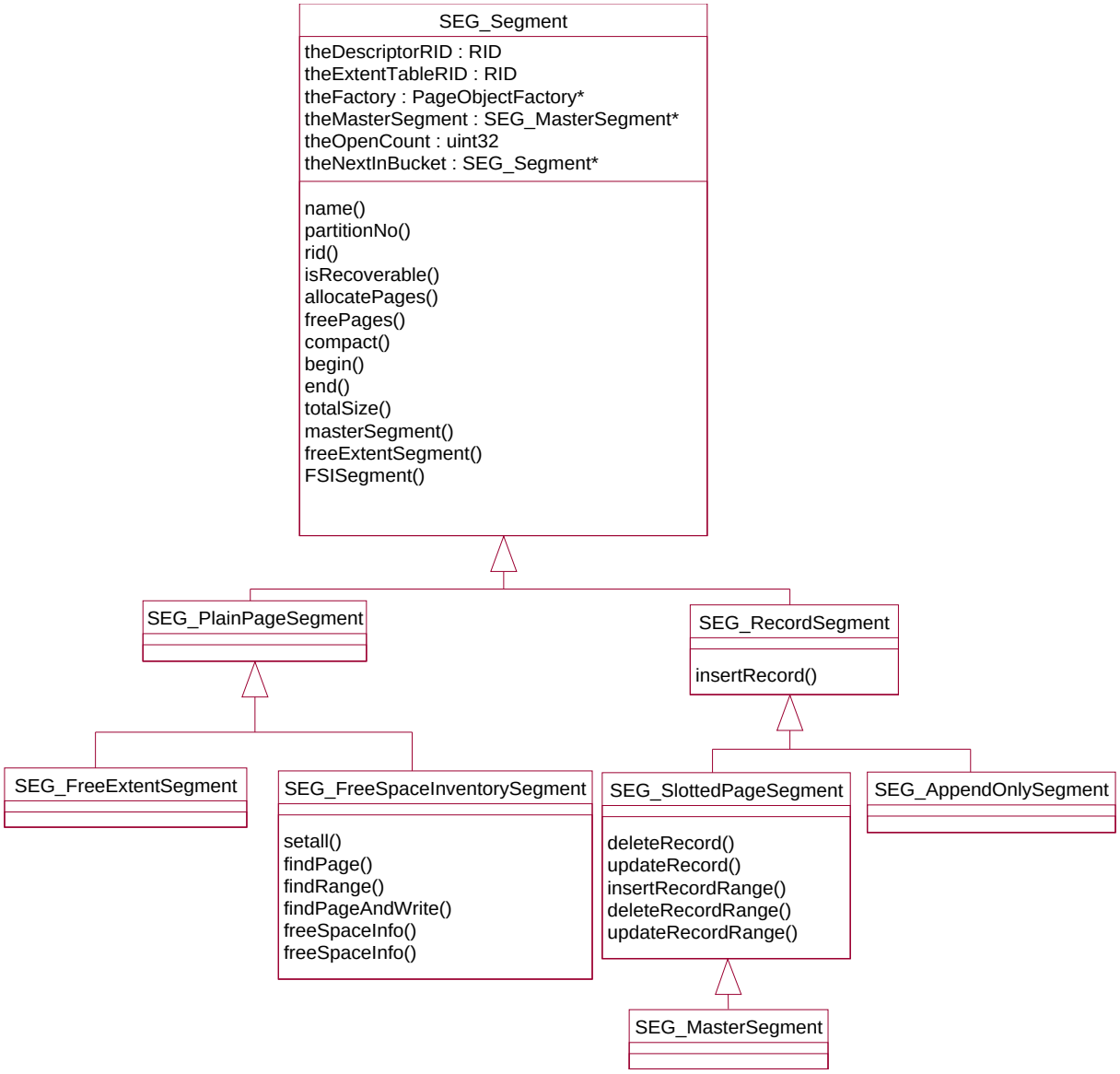


Abbildung 3.11: Schnittstellen der Segmentobjekte

cher zur Verfügung zu stellen. Dies ist notwendig, da Daten nicht direkt auf dem Sekundärspeicher bearbeitet werden können. Da die Größe der Datenbank die Größe des Primärspeichers in der Regel übersteigt, ist es nicht möglich alle einmal in den Primärspeicher geladenen Daten bis zum Ende der Verarbeitung dort zu belassen. Daher muß eine Möglichkeit existieren „alte“ Daten aus dem Primärspeicher zu verdrängen und „neue“ einzulagern. Obwohl aktuelle Betriebssysteme zur Verwaltung des virtuellen Speichers eine ähnliche Funktionalität zur Verfügung stellen, ist es sinnvoll für ein Datenbankverwaltungssystem einen eigenen Systempuffer

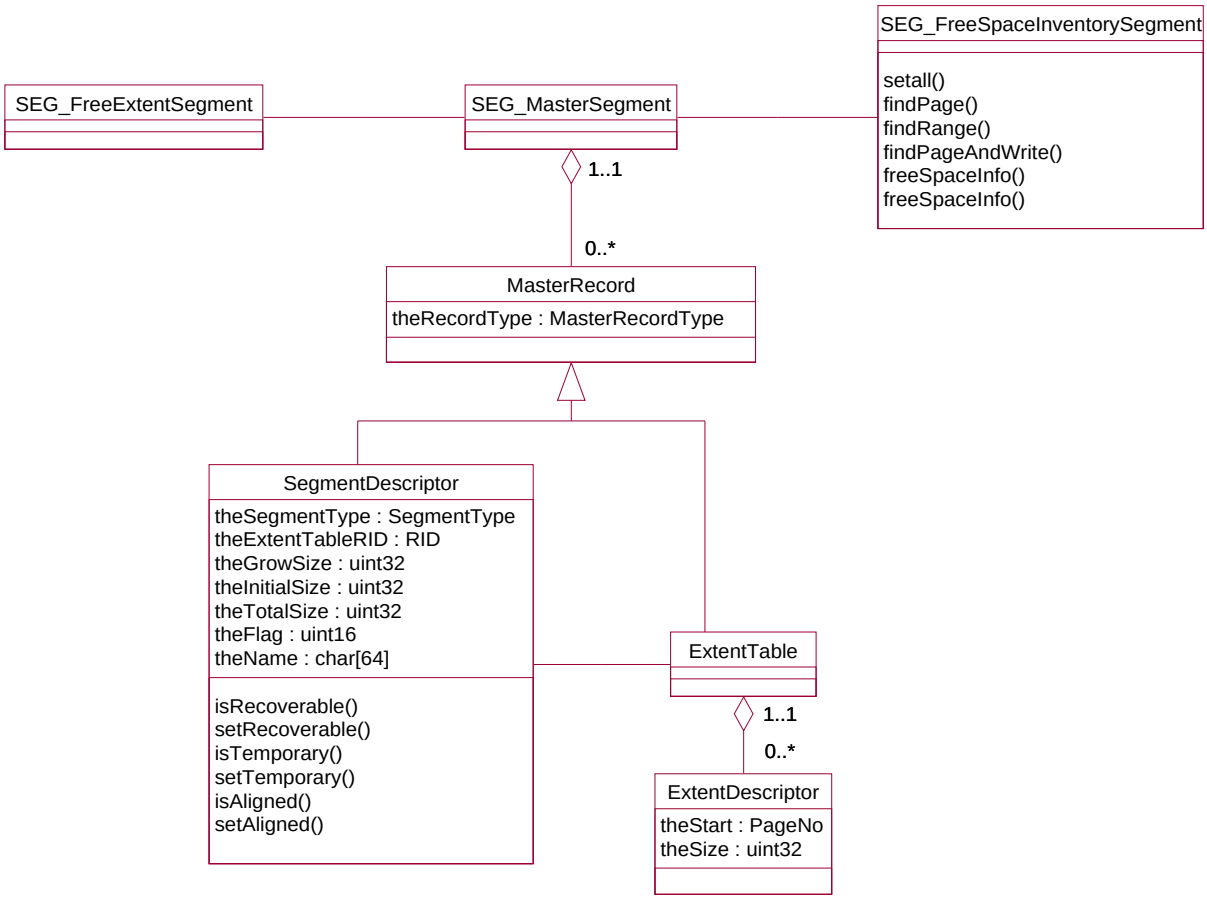


Abbildung 3.12: Verwaltung der Seiten einer Partition

einzurichten. Einige wesentliche Punkte, in denen sich die Anforderungen eines Datenbankverwaltungssystems an den Systempuffer von denen eines Betriebssystems unterscheiden, findet man bei Stonebraker [Sto81].

Das zentrale Ziel bei der Implementierung eines Systempuffers ist die Minimierung der Kosten der E/A-Vorgänge. Einerseits nutzt man die üblicherweise anzutreffende Lokalität der Zugriffe (vgl. [MGST70]) und die in einem Datenbankverwaltungssystem oft vorhandenen Informationen über den zukünftigen Datenbedarf, um die Anzahl der E/A-Vorgänge zu minimieren. Andererseits versucht man möglichst sequentiell auf den Sekundärspeicher zuzugreifen, um die Kosten je Vorgang zu reduzieren.

Um ein weiteres Ziel, die Minimierung des Rechenaufwands, zu erreichen, bemüht man sich um eine einfache Verwaltung des Systempuffers. Dazu legt man zum einen fest, daß Daten nur seitenweise ein- oder ausgelagert werden und zum anderen verwendet man möglichst nur eine Seitengröße (vgl. [HR99]). Die Verwendung von unterschiedlichen Seitengrößen führt nach

Schöning [Sch98] zu einer erheblich aufwendigeren Verwaltung, da zum Beispiel zur Einlagerung einer Seite unter Umständen mehrere im Systempuffer benachbarte Seiten zur Auslagerung bestimmt werden müssen. Dieser zusätzliche Rechenaufwand ist in einem auf OLAP-Anwendungen zugeschnittenen System aufgrund der ohnehin schon hohen Anforderungen an die Prozessorleistung nicht zu rechtfertigen. Die Alternative besteht in der Aufteilung des Systempuffers in mehrere *Pufferbereiche*. In diesem Fall steht für jede verwendete Seitengröße ein eigener Pufferbereich zur Verfügung. Diese Lösung ist zwar sehr einfach zu implementieren und wird daher auch in kommerziellen Datenbankverwaltungssystemen zum Beispiel in DB/2 von IBM verwendet [TG84], hat aber auch einen erheblichen Nachteil: Die „korrekte“ Wahl der Größen der einzelnen Pufferbereiche, die im Falle wechselnder Lastsituationen unter Umständen gar nicht möglich ist, ist für die Leistungsfähigkeit des Gesamtsystems von erheblicher Bedeutung.

3.3.4.2 Aufbau

Der Systempuffer von AODB besteht – falls mehrere Seitengrößen benötigt werden – aus mehreren Pufferbereichen mit festen Seitengrößen. Wir haben diese Lösung aus zwei Gründen gewählt:

- Sie ist mit der geringsten Belastung des Prozessors verbunden.
- Für den häufig anzutreffenden Fall, daß nur eine Seitengröße benötigt wird, ist sie mit keinerlei Nachteilen verbunden.

Jeder Pufferbereich besteht aus einer eigenen Pufferverwaltung und aus einer Menge von Pufferrahmen, die jeweils eine Seite aufnehmen können. Da die Größe der Pufferbereiche fest ist, kann die Verwaltung der einzelnen Teile unabhängig voneinander erfolgen.

Zur Auswahl der zu ersetzenden Seiten wird, wie in vielen Betriebssystemen (vgl. [SG98]) und Datenbankverwaltungssystemen (vgl. zum Beispiel [BJK+97, Sch98]), der LRU-Algorithmus verwendet. LRU steht für *Least Recently Used*, d.h. es wird jeweils die Seite ersetzt, deren letzter Zugriff am weitesten in der Vergangenheit liegt. Die übliche Implementierung dieses Verfahrens ist, die Seiten oder Seitenreferenzen in einer Schlange (engl. *queue*) zu verwalten. Bei jedem Zugriff auf eine Seite, wird diese dann am Anfang der Schlange eingefügt. Die nächste zu ersetzende Seite befindet sich dann stets am Ende der Schlange. Um die von Stonebraker und anderen [Sto81, CD85, SS86] beschriebenen Probleme beim Einsatz von LRU zu umgehen, gibt es allerdings die Möglichkeit der Pufferverwaltung Hinweise (engl. *hints*) zur weiteren Verwendung einer Seite zu geben und so Einfluß auf die Reihenfolge der Seiten in der Ersetzungsschlange (engl. *replacement queue*) zu nehmen. So ist es zum Beispiel möglich Seiten, die mit großer Wahrscheinlichkeit in näherer Zukunft nicht benötigt werden, am Ende der Ersetzungsschlange einzufügen. Außerdem kann die Pufferverwaltung aufgefordert werden,

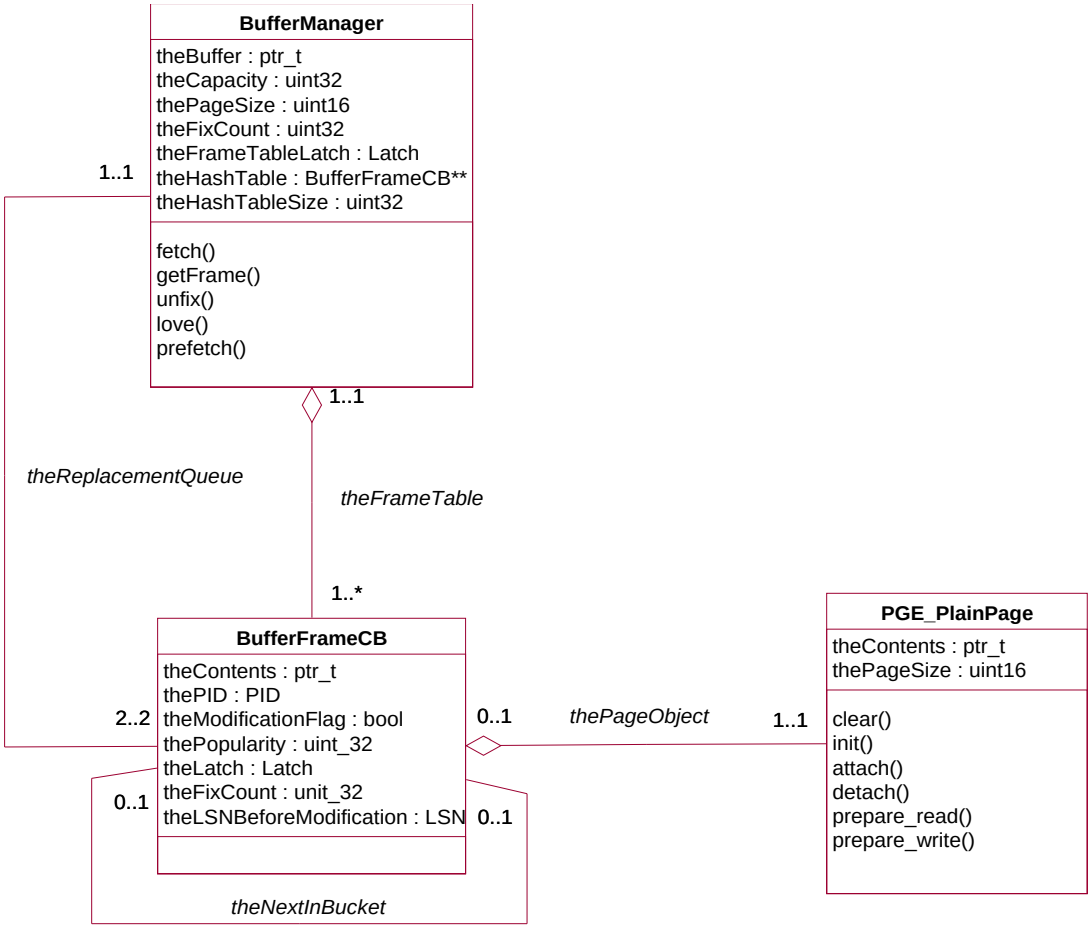


Abbildung 3.13: Klassen zur Pufferverwaltung

asynchron Seiten einzulagern, die vermutlich in naher Zukunft benötigt werden (engl. *prefetching*). So kann das Anwendungswissen genutzt werden und die Pufferverwaltung muß nicht – mit mangelnder Information – aus vergangenen Seitenreferenzen selbstständig auf eventuelle zukünftige Seitenreferenzen schließen.

3.3.4.3 Implementierung

Ein Pufferbereich wird durch ein Objekt der Klasse *BufferManager* verwaltet (vgl. Abbildung 3.13). Da den höheren Schichten des Datenbankverwaltungssystems die Seitenstruktur der Segmente bekannt ist und sie – wie in Abschnitt 3.3.3 dargestellt – explizit Seiten anfordern, kann die Schnittstelle sehr einfach gehalten werden. Sie besteht im wesentlichen aus Methoden

- zur Anforderung einer bestehenden Seite (*fetch*),

- zum Anlegen einer neuen Seite (*getFrame*),
- zur Freigabe eines nicht mehr benötigten Pufferrahmens (*unfix*),
- zur Weitergabe von Hinweise an die Pufferverwaltung (*love*) und
- zur asynchronen Einlagerung von Seiten (*prefetch*).

Da der Nutzer des Systempuffers eine Seite direkt im Systempuffer ändern kann und da die Pufferverwaltung solche Änderungen nicht selbstständig feststellen kann, ist es erforderlich, daß der Nutzer bereits bei der Seitenanforderung angibt, ob die Seite geändert werden soll.

Die zur Verwaltung der zu dem Pufferbereich gehörenden Pufferrahmen benötigten Informationen sind in einer Tabelle zusammengefaßt (*theFrameTable*). Die Tabelle enthält für jeden Pufferrahmen einen Eintrag, den *Pufferrahmenkontrollblock* (*BufferFrameCB*). Dieser enthält:

- die Primärspeicheradresse des Pufferrahmens (*theContents*),
- den systemweit eindeutigen Seitenidentifikator der eingelagerten Seite (*thePID*),
- einen Identifikator, der angibt, ob die Seite seit ihrer Einlagerung verändert wurde (*theModificationFlag*),
- eine Beschreibung des Seitentyps der eingelagerten Seite (*thePageObject*),
- Informationen zur Verwaltung der LRU-Schlange (*theNextInBucket* und *thePopularity*), sowie
- Informationen zur Synchronisation (*theLatch* und *theFixCount*) und
- Informationen für einen Wiederanlauf (*theLSNBeforeModification*).

3.4 Logische Satzschnittstelle

Logische Sätze unterscheiden sich von physischen Sätzen dadurch, daß sie eine Semantik bezüglich des Anwendungsbereichs besitzen. In diesem Abschnitt beschreiben wir, wie physische Sätze auf logische Sätze abgebildet werden.

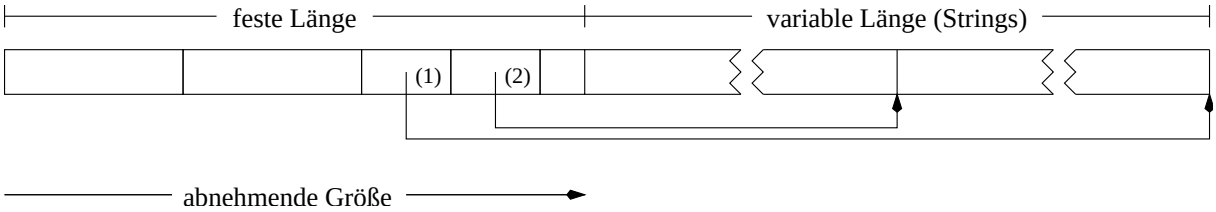


Abbildung 3.14: Aufbau eines Satzes

3.4.1 Eigenschaften

In AODB ist ein logischer Satz ein Tupel von typisierten Werten. Die Aufgabe des logischen Satzschnittstelle besteht darin, den Zugriff auf die einzelnen Attribute eines Tupels zu ermöglichen. Dazu muß die logische Satzschnittstelle die in den physischen Sätzen vorhandenen Informationen interpretieren.

Da in relationalen Systemen alle Tupel einer Relation das gleiche Schema besitzen, werden die Typinformation nicht im physischen Satz selbst, sondern in einem separaten Schema verwaltet. Da in AODB jedes Segment und damit auch jede Seite eindeutig einer Relation zugeordnet ist (vgl. Abschnitt 3.3.1), enthält ein physischer Satz auch keinen Verweis auf sein Schema, d.h. er enthält keinerlei Typinformationen. Zum Zugriff auf einen Attributwert wird also sowohl der physische Satz, der die zu interpretierende Information enthält, als auch das zu dem Satz gehörende Schema benötigt.

3.4.2 Aufbau

In Abbildung 3.14 ist die übliche Strukturierung physischer Sätze dargestellt, die man auch in AODB findet (vgl. [GR93]). Ein physischer Satz wird in zwei Teile geteilt. Der erste Teil enthält alle Attribute deren Werte sich in Feldern fester Länge darstellen lassen und der zweite Teil die Attribute bei denen das nicht möglich ist. Dies sind zum Beispiel Zeichenketten oder auch benutzerdefinierte Typen variabler Länge, wie man sie zum Beispiel in objektrelationalen Systemen findet [SB98].

Im ersten Teil werden die Attribute in der Reihenfolge fallender Darstellungsgrößen abgelegt. Das heißt, daß die Attribute zu deren Darstellung mehr Platz benötigt wird (wie zum Beispiel 8 Byte für doppelt genaue Gleitkommazahlen) vor den Attributen stehen zu deren Darstellung weniger Platz benötigt wird (wie zum Beispiel einzelne ASCII-Zeichen, die nur ein Byte benötigen). Dadurch wird sichergestellt, daß der jeweilige Wert eines Attributs im Primärspeicher stets an den seiner Größe entsprechenden Bytegrenzen ausgerichtet ist, falls der Anfang des physischen Satzes an der größten im Satz auftretenden Bytegrenze ausgerichtet ist. Dies ist erforderlich, um einen direkten Zugriff des Prozessors auf diese Attributwerte im Primärspeicher zu ermöglichen und somit zu vermeiden, daß der jeweilige Attributwert zunächst in einen

entsprechend ausgerichteten Speicherbereich kopiert werden muß, bevor er bearbeitet werden kann.[2] Die sich so ergebende Reihenfolge der Attribute ist unabhängig von der vom Benutzer angegebenen Reihenfolge und sie wird von AODB in Abhängigkeit von der zugrundeliegenden Plattform festgelegt.

Da die Darstellung aller im ersten Teil abgelegten Attribute für alle Tupel einer Relation gleich lang ist, kann auf diese direkt zugegriffen werden. Im Gegensatz dazu muß für jedes Attribut im zweiten Teil die Position des Attributs im Satz einzeln bestimmt werden. Um dies zu ermöglichen werden die Längen der Attribute des zweiten Teils in Feldern fester Länge im ersten Teil abgelegt. So wird in jedem einem Attribut aus dem zweiten Teil zugeordneten Längenfeld, die Summe aus der Länge des entsprechenden Attributs und aller vorangegangenen Attribute variabler Länge abgelegt. Dadurch ist es zur Bestimmung der Position eines Attributs nur notwendig das Längenfeld des vorangehenden Attributs variabler Länge auszulesen und zu der (festen) Länge des ersten Teils hinzuzuaddieren (vgl. (2) in Abbildung 3.14). Eine Ausnahme hiervon bildet das erste Attribut variabler Länge, das stets am Ende des ersten Teils beginnt. In dem diesem Attribut zugeordneten Längenfeld wird daher die Gesamtlänge aller variablen Attribute abgelegt, so daß die Bestimmung des Gesamtlänge des Satzes auch nur einen Zugriff benötigt (vgl. (1) in Abbildung 3.14).

Das zu jeder Relation vorhandene *physische Schema* enthält folgende Informationen:

- die Reihenfolge und die Typen der Attribute in einem physischen Satz und
- die Abbildung zwischen der logischen (vom Benutzer vorgegebenen) und der physischen Reihenfolge der Attribute.

Mit Hilfe dieser Schemainformationen ist es möglich, die bezüglich des logischen Schema formulierten Anforderungen zu erfüllen.

3.4.3 Implementierung

Die Implementierung erfolgt auch hier – wie bei den Seiten – durch Interpretationsobjekte (vgl. Darstellung auf Seite 21). Auch hier unterscheiden wir zwischen dem *Tupelobjekt* und dem *Tupel* oder *Satz*. Die Datenelemente eines normalen Tupelobjekts sind – wie üblich – ein Verweis auf den Speicherbereich, der den Zustand des Objekts enthält, und die Parameter, die in diesem Fall aus einem Verweis auf ein Schema bestehen. Das Schema enthält alle zur Interpretation eines physischen Satzes notwendigen Informationen, wie zum Beispiel die Typen der Attribute oder die Positionen der Attribute fester Länge im Satz.

[2] Diese Erfordernis bezieht sich nur auf einige spezielle Prozessorarchitekturen. Allerdings sind auch bei Prozessorarchitekturen, die in der Lage sind, nicht an Bytegrenzen ausgerichtete Daten zu verarbeiten, solche Zugriffe häufig mit Leistungeinbußen verbunden.

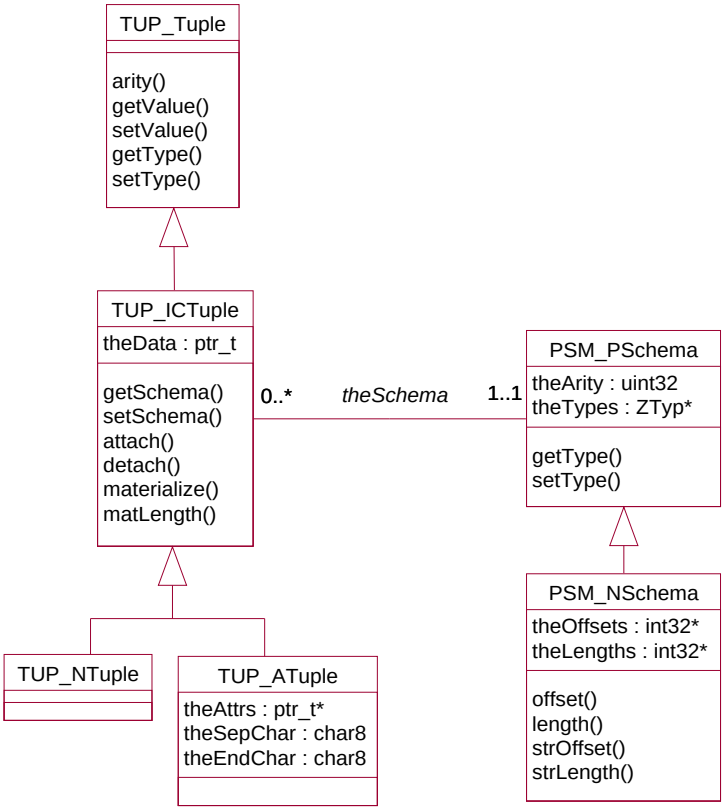


Abbildung 3.15: Schnittstelle der Tupelobjekte

Die Schnittstelle eines Tupelobjekts besteht im wesentlichen aus drei Bereichen (vgl. *TUP_Tuple* in Abbildung 3.15):

- einer Methode zur Bestimmung der Stelligkeit des Tupels (*arity*),
- Methoden zum Auslesen und Setzen des Typs eines Attributs (*getType* und *setType*) und
- Methoden zum Auslesen und Setzen eines Attributwerts (*getValue* und *setValue*).

Die Methoden zum Auslesen und Setzen des Typs eines Attributs existieren an der Tupelschnittstelle, da es prinzipiell auch möglich wäre Tupel zu verarbeiten, die die Typinformation direkt enthalten und zu denen daher kein Schema existiert. AODB nutzt diese Möglichkeit zur Zeit allerdings noch nicht.

Die im letzten Abschnitt beschriebene Tupeldarstellung wird von der Klasse *TUP_NTuple* in Verbindung mit der Klasse *PSM_NSchema* realisiert.

Die Klasse *TUP_ATuple* zeigt, daß aber auch andere Implementierungen der Tupelschnittstelle sinnvoll sein können. Diese Implementierung interpretiert Sätze, die in einer ASCII-Text Dar-

stellung vorliegen. Die erforderlichen Parameter hierfür sind das Zeichen, daß zur Trennung von zwei Attributwerten verwendet wird und das Zeichen, daß zur Trennung von zwei Sätzen verwendet wird. Diese Implementierung ermöglicht uns, zum Beispiel beim Laden von Daten, alle von System zur Verfügung gestellten Mechanismen auch für Textdateien zu verwenden.

Eine weitere alternative Implementierung dieser Schnittstelle stellen wir detailliert in Kapitel 4 vor. Die dort beschriebene Implementierung verwendet eine komprimierte Darstellung der Sätze.

3.5 AODB Virtual Machine

In diesem Abschnitt wird die *AODB Virtual Machine (AVM)* beschrieben (vgl. [WKHM00]). Die zentrale Aufgabe von AVM ist die Auswertung von Ausdrücken wie zum Beispiel Selektionsprädikaten oder Verbundbedingungen. Wie vergleichen zunächst diesen Ansatz zur Auswertung von Ausdrücken mit vorhandenen Alternativen. Anschließend beschreiben wir die virtuelle Maschine und ihre Verwendung zur Verarbeitung der gesamten typisierten Information im System.

3.5.1 Auswertung von Ausdrücken

Prinzipiell gibt es drei Möglichkeiten zur Darstellung und Auswertung von Ausdrücken:

1. Operator-Bäume,
2. assemblerähnliche Programme für eine virtuelle Maschine und
3. Maschinenprogramme für eine reale Maschine.

Die Darstellung von Ausdrücken als Operator-Bäume ist vermutlich am weitesten verbreitet. Als Beispiel betrachten wir den Ausdruck `alter > 30`. Der entsprechende Operator-Baum besteht aus drei Knoten: einem '>'-Knoten an der Wurzel des Baums und zwei Kind-Knoten, die die beiden Operanden `alter` und `30` enthalten. Die Auswertung eines solchen Operator-Baums erfolgt in der Regel durch einen kellerspeicherbasierten Interpreter. Die Vorgehensweise bei dieser Form der Auswertung besteht darin, daß jeder Operator einen oder mehrere Operanden vom aus dem Kellerspeicher entfernt (d.h. aus dem Kellerspeicher kopiert), diese verarbeitet und anschließend das Ergebnis wieder im Kellerspeicher ablegt (d.h. in den Kellerspeicher kopiert).

Im Gegensatz dazu werden assemblerähnliche Programme und Maschinenprogramme durch einen registerbasierten Interpreter bzw. durch einen Prozessor ausgewertet. Hierbei arbeiten

alle Instruktionen auf einem Registersatz, dessen Register sowohl zur Ein- als auch zur Ausgabe verwendet werden können. Dies hat zwei Vorteile:

- Zur Auswertung von Ausdrücken muß wesentlich weniger kopiert werden (was unserem Ziel aus Abschnitt 3.1.2 sehr entgegenkommt) und
- die einfache Faktorisierung von Teilausdrücken wird möglich.

Trotzdem gibt es nur wenige Veröffentlichungen, die einen solchen Ansatz erwähnen. Uns ist lediglich ein alter technischer Bericht von IBM bekannt [LW79], in dem die dritte Alternative verfolgt wird. Lorie und Wade beschreiben dort wie System R System/370-Instruktionen zur Auswertung von Ausdrücken erzeugt. Obwohl diese Vorgehensweise eine sehr effiziente Auswertung von Ausdrücken ermöglicht, ist sie zu wenig flexibel. Um mit unterschiedlichen Betriebssystemen und Prozessorarchitekturen arbeiten zu können und um Einfluß auf den unterstützten Befehlssatz zu haben, verwenden wir zur Darstellung und Auswertung von Ausdrücken assemblerähnliche Programme für eine plattformunabhängige virtuelle Maschine. Obwohl diese Flexibilität natürlich mit Kosten verbunden ist, bemühen wir uns trotzdem um eine hocheffiziente Auswertung. Daher haben wir die virtuelle Maschine durch die Verwendung datenbankspezifischer Instruktionen und durch die Verwendung effizienter Implementierungstechniken optimiert.

3.5.2 Die virtuelle Maschine

In diesem Abschnitt beschreiben wir die Elemente der virtuellen Maschine. Des weiteren führen wir an einem Beispiel die von der Maschine verarbeiteten Programme ein und stellen dar inwieweit die Maschine datenbankspezifisch ist und wie sie sehr effizient implementiert werden kann. In welcher Weise die virtuelle Maschine während der Anfragebearbeitung genutzt wird, beschreiben wir später in Abschnitt 3.6.2.

Interpreter Der zentrale Bestandteil der virtuellen Maschine ist der *AVM-Interpreter*. Dieser führt AVM-Programme auf AVM-Registern aus.

Register Das grundlegende Prinzip beim Entwurf von AVM war das Verhalten eines realen Prozessors nachzuahmen. Genau wie Prozessor-Instruktionen auf Prozessor-Registern arbeiten, arbeiten AVM-Instruktionen auf AVM-Registern. *AVM-Register* sind stets 64 Bit lang und entsprechen damit der Länge aktueller Prozessorregister.

Ein AVM-Register enthält einen atomaren Wert, der entweder ein Attributwert oder ein Wert eines internen Datentyps wie zum Beispiel ein Zeiger auf ein Tupelobjekt (vgl. Abschnitt 3.4)

ist. Genau wie reale Maschinen, die Werte und Zeiger in Register schreiben, um teure Primärspeicherzugriffe zu vermeiden, verwendet unsere virtuelle Maschine ihre Register, um Attributzugriffe zu vermeiden, die auch teuer sein können (insbesondere wenn die Daten komprimiert oder verschlüsselt vorliegen, vgl. Kapitel 4). Außerdem können Register zur Speicherung von Ergebnissen von gemeinsamen Teilausdrücken verwendet werden, die dann nicht mehrfach ausgewertet werden müssen.

Zur Verarbeitung werden jeweils mehrere Register zu einem *Registersatz* zusammengefaßt. Ein AVM-Programm wird vom AVM-Interpreter jeweils bezüglich dreier Registersätze ausgeführt. Diese werden als *erster Registersatz*, *zweiter Registersatz* und *Hilfsregistersatz* bezeichnet. Die Vorteile der Verwendung mehrerer Registersätze werden erst in Abschnitt 3.6 deutlich. Jetzt können wir aber schon feststellen, daß der erste und der zweite Registersatz prinzipiell zur Speicherung von Attributwerten verwendet werden und daß der Hilfsregistersatz in erster Linie zur Speicherung von Zwischenergebnissen und internen Werten verwendet wird. Der erste Registersatz hat eine besondere Bedeutung, da viele Instruktionen nur auf Registern dieses Registersatzes ausgeführt werden können. Diese Einschränkung reduziert die Anzahl der Instruktionen und somit den Platzbedarf des Interpreters.

Programme Ein *AVM-Programm* besteht aus einer Folge von Instruktionen. *AVM-Instruktionen* haben allgemein folgende Form:

Name_Typ_Arg_Erg

Dabei beschreibt **Name** die Funktion des Befehls, **Typ** den Typ der Argumente, **Arg** die Art des Zugriffs auf die Argumente und **Erg** die Art der Verarbeitung des Ergebnisses. Die Teile **Typ**, **Arg** und **Erg** können auch wegfallen, falls es für den jeweiligen Befehl diesbezüglich keine Alternativen gibt.

Zur Erläuterung betrachten wir das Programm in Abbildung 3.16. Bei diesem Programm handelt es sich um das Fortschreibungsprogramm der Aggregation[3] in der ersten Anfrage des TPC-D-Benchmarks [TPC95]. Das Programm besteht im wesentlichen aus drei Teilen: der erste Teil (Zeile 1) erhöht die Anzahl der an dieser Aggregation beteiligten Tupel um eins, der zweite Teil (Zeilen 2–6) lädt die benötigten Attributwerte in Register und im dritten Teil (Zeilen 7–15) werden die aggregierten Werte berechnet. Betrachten wir nun die einzelnen Instruktionen. Die Instruktion in Zeile 1 `INC_UI4` besteht nur aus den Teilen **Name** (`INC` für „increment“) und **Typ** (`UI4` für „4 byte unsigned integer“). Da das Argument der Instruktion genau eine Regi-

[3]Die Aggregation erfolgt in drei Schritten. Im ersten Schritt werden die aggregierten Werte initialisiert. Im zweiten Schritt werden die Attributwerte jedes Tupels mit Hilfe des Fortschreibungsprogramms akkumuliert. Im dritten Schritt wird die Berechnung der aggregierten Werte abgeschlossen, so wird zum Beispiel zur Berechnung der Funktion `AVG` die Summe der Werte durch die Anzahl der Werte dividiert.

```
 1  INC_UI4         0              // erhöhe COUNT(*)
 2  MV_PTR_Y        1        1     // kopiere den Zeiger auf das Tupel
 3  LOAD_SF8_C      4    1   6     // lade L_QUANTITY
 4  LOAD_SF8_C      5    1   7     // lade L_EXTENDEDPRICE
 5  LOAD_SF8_C      6    1   8     // lade L_DISCOUNT
 6  LOAD_SF8_C      7    1   9     // lade L_TAX
 7  ADD_SF8_ZZ_C    6   10  10     // summiere L_QUANTITY auf
 8  ADD_SF8_ZZ_C    7   11  11     // summiere L_EXTENDEDPRICE auf
 9  ADD_SF8_ZZ_C    8   12  12     // summiere L_DISCOUNT auf
10  SUB_SF8_CZ_C  1.0    8  13     // berechne 1-L_DISCOUNT
11  ADD_SF8_CZ_C  1.0    9  14     // berechne 1+L_TAX
12  MUL_SF8_ZZ_C    7   13  17     // berechne L_EXTDPRICE*(1-L_DISC)
13  ADD_SF8_ZZ_C   17   15  15     // summiere L_EXTDPRICE*(1-L_DISC) auf
14  MUL_SF8_ZZ_C   17   14   4     // berechne (...)*(1+L_TAX))
15  ADD_SF8_ZZ_C    4   16  16     // summiere (...)*(1+L_TAX)) auf
16  AVM_STOP
```

Abbildung 3.16: AVM-Programm: Fortschreibungsprogramm

sternummer[4] ist, die das Register bezeichnet, dessen Inhalt um eins erhöht werden soll, werden die Teile **Arg** und **Erg** nicht benötigt. Die Instruktion in Zeile 2 besteht aus den Teilen **Name** (`MV` für „move“), **Typ** (`PTR` für „pointer“) und **Arg** (`Y` für „aus dem zweiten Registersatz“). Sie gibt an, daß der sich in Register 1 des zweiten Registersatzes befindende Zeiger in das Register 1 des ersten Registersatzes kopiert werden soll. Der hier kopierte Zeiger auf ein Tupelobjekt (vgl. Abschnitt 3.4) wird benötigt, um die Instruktionen zum Laden von Attributwerten in den Zeilen 3–6 auszuführen. Diese Ladeinstruktionen bestehen aus den Teilen **Name**, **Typ** (`SF8` für 8 byte signed float) und **Erg** (`C` für „copy“) und laden jeweils den Wert eines Attributs des durch den Zeiger spezifizierten Tupels in ein Register. In den Zeilen 7–15 stehen arithmetische Instruktionen, die aus allen vier Teilen bestehen. Da es sich um binäre Operationen handelt, besteht der **Arg** Teil hier aus zwei Buchstaben, die jeweils den Zugriff auf eines der Argumente spezifizieren (dabei steht `Z` für „aus dem ersten Registersatz“ und `C` für konstante Werte). Die `AVM_STOP` Instruktion in Zeile 16 kennzeichnet das Ende des Programms und hat keine Argumente.

Der Instruktionssatz von AVM ist in zweierlei Hinsicht datenbankspezifisch:

1. Die Instruktionen unterstützen die üblicherweise vom Datenbanksystem zur Verfügung gestellten Datentypen.

2. Es gibt die Möglichkeit zu spezifizieren, in welcher Weise das Ergebnis einer Instruktion weiterverarbeitet werden soll (s.u.).

[4] Die Begriff Register oder Registernummer beziehen sich – sofern nicht anders angegeben – stets auf den ersten Registersatz

```
 1  INC_UI4        0                 // entspricht Zeile 1
 2  MV_PTR_Y       1        1        // entspricht Zeile 2
 3  LOAD_SF8_A     4    1   10       // entspricht Zeilen 3 und 7
 4  LOAD_SF8_B     5    1   7   11   // entspricht Zeilen 4 und 8
 5  LOAD_SF8_B     6    1   8   12   // entspricht Zeilen 5 und 9
 6  LOAD_SF8_C     7    1   9        // entspricht Zeile 6
 7  SUB_SF8_CZ_C  1.0   8   13       // entspricht Zeile 10
 8  ADD_SF8_CZ_C  1.0   9   14       // entspricht Zeile 11
 9  MUL_SF8_ZZ_B   7   13   17   15  // entspricht Zeilen 12 und 13
10  MUL_SF8_ZZ_A  17   14   16       // entspricht Zeilen 14 und 15
11  AVM_STOP
```

Abbildung 3.17: AVM-Programm: verbessertes Fortschreibungsprogramm

Eine weitere Möglichkeit, die von AODB noch nicht genutzt wird, besteht darin, auch die Synchronisationsmechanismen und die für einen Wiederanlauf erforderlichen Maßnahmen durch AVM-Instruktionen zu realisieren. So könnten auch diese Maßnahmen vom Optimierer berücksichtigt und optimiert ausgeführt werden.

Typspezifische Instruktionen Die virtuelle Maschine wird – wie durch das Fortschreibungsprogramm im obigen Beispiel schon angedeutet – nicht nur zur Auswertung von Selektionsprädikaten, sondern zur Verarbeitung der gesamten typisierten Information im System verwendet. Dazu gehört zum Beispiel die Berechnung von Streuwerten (engl. *hash-values*), die Konstruktion von Verbundergebnissen oder eben die dargestellte Aggregation von Werten. Aus dieser Kapselung der Verarbeitung typisierter Information ergibt sich, daß außerhalb der virtuellen Maschine keine typspezifische Verarbeitung erforderlich ist. Dies führt zum einem zu einer vereinfachten, effizienteren und auch robusteren Implementierung der algebraischen Operatoren und zum anderen vereinfacht es die Erweiterung des Systems um weitere Datentypen.

Variable Ergebnisverwendung Es gibt drei Möglichkeiten zu spezifizieren, in welcher Weise das Ergebnis einer Instruktion weiterverarbeitet werden soll:

1. `C` (für „copy“) um – wie schon in der Erläuterung der Abbildung 3.16 erwähnt – das Ergebnis in ein Ergebnisregister zu kopieren,
2. `A` (für „add“) um das Ergebnis zu dem im Ergebnisregister enthaltenen Wert hinzuzuaddieren und
3. `B` (für „both“) um das Ergebnis in ein Ergebnisregister zu kopieren *und* zu dem in einem zweiten Ergebnisregister enthaltenen Wert hinzuzuaddieren.

Mit Hilfe dieser Spezifikationen fällt es leicht gemeinsame Teilausdrücke zu entfernen und und die Anzahl der zu speichernden Zwischenergebnisse zu reduzieren. Das Beispiel in Abbildung

```
LOOP:
  .
  .
switch(instr->cmd) {
  .
  .
case ADD_UI4_ZZ_C:
  {
    register uint32 zui4 = reg1[instr->op1.reg].ui4;
    zui4 += reg1[instr->op2.reg].ui4;
    reg1[instr->res1].ui4  = zui4;
  }
  goto LOOP;
  .
  .
}
```

Abbildung 3.18: Ausschnitt aus dem Quelltext des AVM-Interpreters

3.17 zeigt, daß so die Anzahl der erforderlichen Instruktionen im Vergleich zu Abbildung 3.16 um ein Drittel reduziert werden konnte.

Effiziente Implementierung Um insgesamt zu einer effizienten Auswertung der AVM-Programme zu kommen, ist es nicht nur notwendig die Anzahl der AVM-Instruktionen zu reduzieren, sondern auch die Anzahl der Instruktionen auf der verwendete Maschine selbst zu minimieren. Eine übliche Technik bei der Implementierung von virtuellen Maschinen besteht darin, einen Index in ein Feld von Funktionszeigern als Befehlskode zu verwenden. Dies hat allerdings den entscheidenden Nachteil, daß für jede ausgewertete Instruktion mindestens ein Funktionsaufruf erforderlich ist. Dieser Funktionsaufruf ist häufig aufwendiger, als die Auswertung der Instruktion selbst.[5] Wir verwenden daher eine einfache Sprungtabelle, die auch das Verhalten eines realen Prozessors näher kommt. In Abbildung 3.18 ist ein Ausschnitt aus dem C++-Quelltext des AVM-Interpreters dargestellt. Die Sprungtabelle wird durch eine `switch`-Anweisung implementiert. Die sehr einfache Implementierung des dargestellten Befehls zeigt, daß durch die Verwendung von AVM der Rechenaufwand für eine solche Addition sehr klein gehalten werden kann.

[5] Bei Messungen mit einem SUN UltraSparcII-Prozessor mit 300 MHz war ein Funktionsaufruf zur Durchführung einer Ganzzahl-Addition (ohne die Addition) im günstigsten Fall 1,7-mal so aufwendig, wie die Addition selbst. Der „günstigste Fall“ bezieht sich auf den gewählten C-Übersetzer und die Übersetzeroptionen

3.5.3 Zeichenkettenverwaltung

Da Zeichenketten in der Regel mehr Platz beanspruchen als die atomaren Datentypen und daher nicht in ein Register passen, ist eine separate Behandlung erforderlich. In AODB gibt es daher je Anfrage einen speziellen Puffer, in dem die zur Verarbeitung der Anfrage benötigten Zeichenketten abgelegt werden.

Beim Laden einer Zeichenkette wird diese in den Puffer kopiert und eine Referenz auf die Zeichenkette in dem in der Ladeinstruktion spezifizierten Register abgelegt. Im weiteren Verlauf der Anfragebearbeitung werden dann nur noch Referenzen verwendet, so daß die Zeichenkette selbst nicht mehr kopiert werden muß. Um festzustellen, ob eine Zeichenkette zur weiteren Verarbeitung noch benötigt wird, verwaltet der Puffer außerdem für jede Zeichenkette die Anzahl der jeweils gültigen Referenzen auf diese Zeichenkette. Falls diese Anzahl auf Null sinkt, wird der verwendete Speicherplatz zur erneuten Verwendung wieder freigegeben. Alle während der Anfragebearbeitung nicht freigegeben Zeichenketten werden am Ende der Anfragebearbeitung zusammen mit den Referenzzählern freigegeben.

Der Puffer hat keine Informationen über den Inhalt der Zeichenketten. Es ist daher unter anderem nicht möglich, für Zeichenketten gleichen Inhalts, die zu unterschiedlichen Zeitpunkten angelegt wurden, nur einen Repräsentanten zu verwenden.

3.6 Physische Algebra

In einem relationalen Datenbankverwaltungssystem werden Anfragen in einer deklarativen Anfragesprache (zum Beispiel SQL [ISO97]) üblicherweise in algebraische Ausdrücke übersetzt, die dann vom Laufzeitsystem ausgewertet werden (vgl. Abschnitt 3.1.1). Die in diesen algebraischen Ausdrücken verwendeten Operatoren sind Operatoren der physischen Algebra. Die *physische Algebra* unterscheidet sich unter anderem in folgenden Punkten von der *logischen Algebra* des zugrundeliegenden Datenmodells, also der relationalen Algebra (vgl. [Gra93]):

- Die physische Algebra betrachtet Sequenzen von Tupeln und nicht Mengen. Daher ist es zum Beispiel möglich, daß das Ergebnis der Anwendung eines Operators Duplikate enthält oder sortiert ist.
- Die Operatoren der physischen Algebra sind mit mit spezifischen Algorithmen verbunden, sie legen also nicht nur das Ergebnis der Anwendung des Operators fest, sondern auch, wie dieses Ergebnis bestimmt wird. Daher ist es auch möglich die Kosten der Verwendung physischer Operatoren zu bestimmen.
- Die physische Algebra ist systemabhängig, da jedes System unterschiedliche Algorithmen zur Implementierung der Operatoren verwenden kann. So sind zum Beispiel viele

unterschiedlich Implementierungen des Verbundoperators bekannt, von denen üblicherweise nur ein Teil innerhalb eines Datenbankverwaltungssystems verwendet wird.

In diesem Abschnitt werden der Aufbau und die Funktionsweise der in AODB enthaltenen physischen Operatoren beschrieben.

3.6.1 Iterator-Modell

Die von den Operatoren zu verarbeitenden Tupelsequenzen können sehr groß werden. Es ist daher häufig nicht sinnvoll diese Sequenzen als Einheit zu verarbeiten. Statt dessen werden die zu einer Sequenz gehörenden Tupel einzeln verarbeitet. Der Zugriff auf die einzelnen Tupel einer Sequenz kann mit Hilfe des Iterator-Musters erfolgen (vgl. Darstellung auf Seite 23 in Abschnitt 3.3.2.3). Es ist sogar üblich die physischen Operatoren selbst als Iteratoren zu implementieren. Dieser Ansatz und die Vorteile dieses Ansatzes werden von Graefe [Gra93] ausführlich erläutert. AODB folgt diesem Ansatz.

Im klassischen Iterator-Modell zur Implementierung relationaler Operatoren, wie es von Graefe [Gra93] beschrieben wird, besteht die Schnittstelle zu einem Operator aus den drei Methoden *open*, *next* und *close*. Dabei belegt *open* die zur Auswertung des Operators notwendigen Ressourcen und initialisiert die verwendeten Datenstrukturen. *Next* gibt bei jedem Aufruf ein Tupel des Ergebnisses zurück. Wenn kein weiteres Ergebnistupel mehr vorhanden ist, so wird dies durch einen speziellen Rückgabewert ausgedrückt. *Close* schließt die Auswertung des Operators ab und gibt sie angeforderten Ressourcen wieder frei.

Für die Verwendung in AODB habe wir das klassische Iteratormodell in zweierlei Hinsicht verfeinert. Zum einen wurden die Methoden *open* und *close* durch jeweils drei Methoden ersetzt, die eine feinere Steuerung der Auswertung ermöglichen. Zum anderen haben wir die Schnittstelle der *next*-Methode abgewandelt, um die die Menge der zu kopierenden Daten zu reduzieren und um eine bessere Integration mit AVM zu erreichen.

Die *open*-Methode des klassischen Iterator-Modells erfüllt drei Aufgaben, die am Beispiel eines *Leseoperators* (engl. *Scan-Operator*) dargestellt werden:

1. Belegen von Ressourcen (Öffnen der zu der Relation gehörenden Segmente),
2. Initialisieren des Operators (Festlegung eines Selektionsprädikats) und
3. Initialisieren des Iteratorfunktionalität (Speicherung eines Zeigers auf den ersten Satz im Segment).

Da es aber durchaus möglich ist, daß das Ergebnis eines Operators mehrfach benötigt wird und daß der Operator daher mehrfach ausgewertet werden muß (zum Beispiel für den Max-Operator

aus Abschnitt 5.4), ist die Zusammenfassung dieser drei Aufgaben nicht immer erwünscht. Offensichtlich ist es nicht sinnvoll bei einer Mehrfachauswertung zwischen je zwei Auswertungen die verwendeten Ressourcen erst freizugeben und unmittelbar danach wieder zu belegen. Daher ist eine Trennung der Belegung der Ressourcen von den anderen beiden Aufgaben notwendig. Des weiteren ist es unter Umständen sinnvoll die (semantische) Initialisierung des Operators und die Initialisierung der Iteratorfunktionalität zu trennen. Betrachten wir zum Beispiel den Fall, daß das Ergebnis einer Teilanfrage in einer temporären Relation zwischengespeichert wurde. Falls nun diese Unteranfrage mit einer anderen (semantischen) Initialisierung erneut ausgewertet werden soll, so muß eine neue temporäre Relation erzeugt werden. Falls eine erneute Initialisierung nicht notwendig ist, reicht es hingegen aus, die vorhandene temporäre Relation erneut zu lesen. In AODB wurde die *open*-Methode daher durch die Methoden

1. *create* (Belegen von Ressourcen),
2. *initialize* (Initialisieren des Operators) und
3. *start* (Initialisieren des Iteratorfunktionalität)

ersetzt. Dementsprechend wurde auch *close* durch

1. *finish* (Gegenstück zu *start*),
2. *deinitialize* (Gegenstück zu *initialize*) und
3. *destroy* (Gegenstück zu *create*)

ersetzt. Dadurch ist es möglich stets nur die tatsächlich erforderlichen Schritte durchzuführen. Die resultierende Schnittstelle der Operatoren ist zusammen mit einigen grundlegenden Operatoren in Abbildung 3.19 dargestellt.

Jeder Operator (mit Ausnahme der Leseoperatoren) kann die Ausgabe eines anderen Operators als Eingabe verwenden. Zur Beschreibung der Kommunikation zwischen zwei Operatoren bezeichnen wir den die Eingabe erzeugenden Operator als *Produzenten* und den die Eingabe verarbeitenden Operator als *Konsumenten*. Im klassischen Iterator-Modell erhält der Konsument, beim Aufruf der *next*-Methode des Produzenten ein Tupel zurück und zwar üblicherweise in Form eines Zeigers auf einen Speicherbereich, der dem Produzenten zuzuordnen ist. In AODB wird der Speicherbereich, in dem der Produzent sein Ergebnis konstruiert, in der Regel vom Konsumenten zur Verfügung gestellt. Der Konsument übergibt beim *next*-Aufruf einen Registersatz, der vom Produzenten gefüllt wird. Im Falle eines Leseoperators als Produzent würde dieser also zum Beispiel die benötigten Attribute eines Tupels in den vom Konsumenten übergebenen Registersatz laden. Diese Vorgehensweise bietet einige Vorteile.

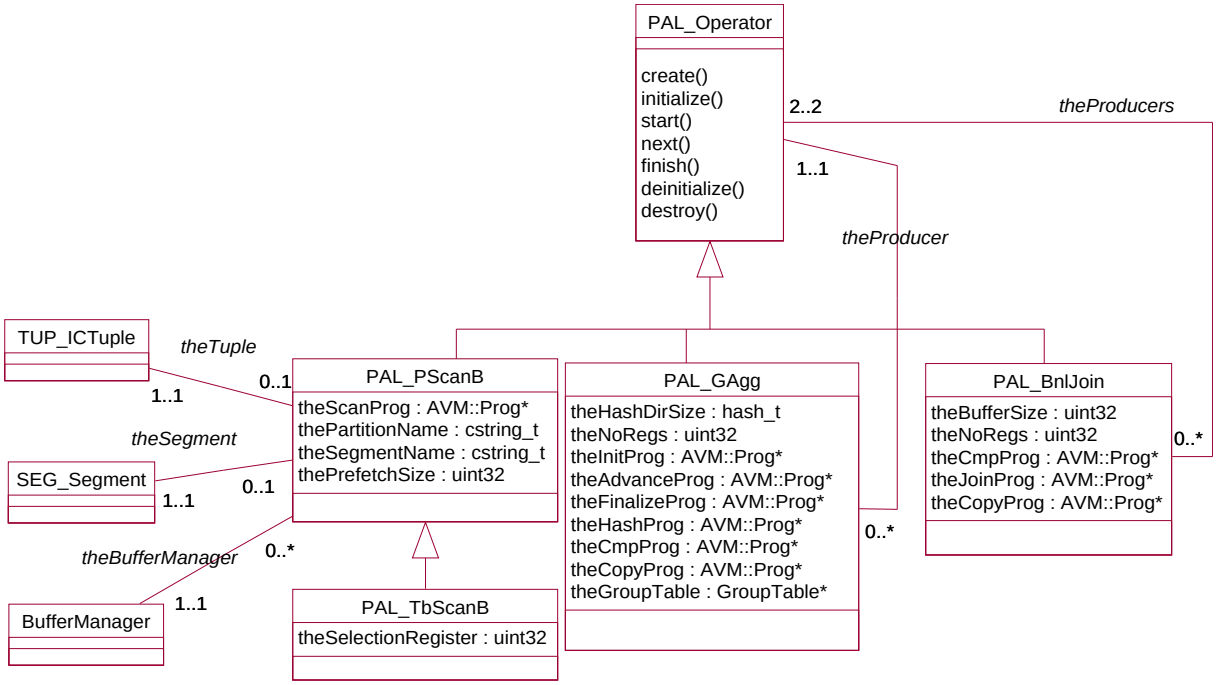


Abbildung 3.19: Einige Operatoren der physischen Algebra

1. Im Idealfall wird nur ein Registersatz für die Auswertung einer Anfrage benötigt, der von jedem Operator an seinen Produzenten weitergegeben wird. Dadurch wird zum einen sehr wenig Speicherplatz verwendet und zum anderen wird der Kopieraufwand reduziert, da die Informationen nicht von Operator zu Operator kopiert werden müssen.

2. Da Register nicht nur Attributwerte, sondern auch interne Datentypen, wie zum Beispiel Zeiger auf Tupelobjekte (vgl. Abschnitt 3.4) enthalten können, ist es möglich das Laden von Attributwerten bis zu dem Zeitpunkt zu verzögern, zu dem sie tatsächlich benötigt werden. Ein Beispiel dafür findet man im Fortschreibungsprogramm in Abbildung 3.16 in Abschnitt 3.5. Durch diese Verzögerung kann sichergestellt werden, daß nur die Attributwerte geladen werden, die zur Auswertung der Anfrage erforderlich sind. Es wird dadurch auch möglich die von Palermo [Pal74] beschriebenen „impliziten Verbunde“ zu implementieren. Bei diesen werden zunächst nur Paare von Tupelreferenzen konstruiert, die dann in einem zweiten Schritt zur Konstruktion der Ergebnistupel verwendet werden.

3. Da alle Daten in Registersätzen übergeben werden, können sie direkt, d.h. ohne Typüberprüfung oder Konversion, vom AVM-Interpreter verarbeitet werden.

Leider ist es nicht immer möglich zur Auswertung einer Anfrage nur einen Registersatz zu verwenden. Einigen Operatoren, wie zum Beispiel der Sortierungsoperator, müssen erst alle vom

Produzenten konstruierten Tupel betrachten, bevor ein Tupel an den Konsumenten zurückgegeben werden kann. In diesem Fall ist es erforderlich, daß der Operator die Tupel des Produzenten zwischenspeichert. Abhängig von der Anzahl der zu speichernden Tupel kann dies entweder im Primär- oder im Sekundärspeicher geschehen. Falls nun ein Operator mehrere Registersätze im Primärspeicher zwischenspeichert, ist es nicht sinnvoll den Inhalt dieser Registersätze bei Verwendung der *next*-Methode in den vom Konsumenten zur Verfügung gestellten Registersatz zu kopieren. Statt dessen kann ein Verweis auf den zwischengespeicherten Registersatz zurückgegeben und damit ein erneuter Kopiervorgang vermieden werden.

3.6.2 Verwendung der AVM

In Abschnitt 3.5.2 wurde bereits erwähnt, daß die AVM ein Programm bezüglich dreier Registersätze auswertet. Diese sind der *erste Registersatz* (oder *Z-Registersatz*), der *zweite Registersatz* (oder *Y-Registersatz*) und der *Hilfsregistersatz* (oder *H-Registersatz*).

Erster Registersatz Der erste Registersatz ist der Registersatz, der im Idealfall vom obersten Operator des Auswertungsplans bis zum zugrundeliegenden Leseoperator übergeben wird und zum Transport der Informationen durch die verschiedenen Operatoren verwendet wird.

Zweiter Registersatz Der zweite Registersatz wird unter anderem für mehrstellige Operatoren, wie zum Beispiel Verbundoperatoren, verwendet. Mehrstellige Operatoren funktionieren in der Regel so, daß je Produzent ein Registersatz existiert, der zur Kommunikation mit dem Produzenten dient und daß aus jeweils zwei Registersätzen mit Hilfe des AVM-Interpreters ein Ergebnisregistersatz konstruiert wird.

Obwohl es prinzipiell auch möglich wäre, nur einen Registersatz zu verwenden und diesen allen Produzenten zur Verfügung zu stellen, hat sich die von uns gewählte Vorgehensweise als günstig erwiesen. Wenn nur ein Registersatz verwendet wird, muß dieser mehr Platz (also mehr Register) zur Verfügung stellen. Bei Operatoren, die mehrere Registersätze temporär zwischenspeichern, führt dies dazu, daß mehr Daten auf den Sekundärspeicher ausgelagert werden müssen, da der Primärspeicher mit (halbleeren) Registersätzen gefüllt ist. Der dadurch hervorgerufene Leistungsverlust überstieg den durch zusätzlichen Kopieraufwand zur Konstruktion der Ergebnisregistersätze hervorgerufenen bei weitem. Außerdem trägt diese Verwendung von mehreren Registersätzen für mehrstellige Operatoren wesentlich dazu bei, daß die Anzahl der Register in einem Registersatz relativ klein gehalten werden kann. So wurden zum Beispiel in unserer Implementierung des TPC-D-Benchmarks [TPC95] nie Registersätze verwendet, die mehr als 22 Register enthielten.

Neben der Auswertung mehrstelliger Operatoren wird der zweite Registersatz, wie schon in Abschnitt 3.5 dargestellt, auch zur Aggregation verwendet. Dabei enthält der erste Registersatz

die aggregierten Werte und der zweite Registersatz die Werte, die zu den Aggregaten hinzugefügt werden sollen. In jedem Fall wird der zweite Registersatz als reiner „Informationslieferant“ verwendet, d.h. die Daten werden aus dem zweiten Registersatz nur gelesen und im ersten Registersatz verarbeitet. Diese Asymmetrie zwischen den Registersätzen bietet den Vorteil, daß nicht alle Befehle für alle Adressierungsarten implementiert werden müssen. Dadurch wird es möglich, daß der ausführbare Kode des AVM-Interpreters klein bleibt und vollständig in den Zwischenspeicher der zweiten Ebene (engl. *second-level-cache*) moderner Prozessoren paßt.

Hilfsregistersatz Im Gegensatz zu den ersten zwei Registersätzen, die Daten verarbeiten, die zwischen unterschiedlichen Operatoren ausgetauscht werden, dient der Hilfsregistersatz zum Austausch von Daten zwischen einem Operator und dem AVM-Interpreter. Zur Auswertung eines Selektionsprädikats werden zum Beispiel vom Operator die benötigten Werte im Hilfsregistersatz dem AVM-Interpreter zur Verfügung gestellt. Dieser wertet das Prädikat dann aus und schreibt das Ergebnis der Auswertung zurück in den Hilfsregistersatz, aus dem der Operator das Ergebnis auslesen kann.

Der Operator erhält den Hilfsregistersatz als Argument des *initialize*-Aufrufs. Dies hat zwei Folgen:

- Ein Konsument muß zum (semantischen) Initialisieren eines Produzenten lediglich die entsprechenden Werte in einen Registersatz schreiben.
- Es wird nur ein Hilfsregistersatz für eine Anfrage benötigt.

Natürlich führt die Verwendung eines Registersatzes für alle Operatoren auch hier zu einem sehr langen Registersatz. In diesem Fall ist das aber unproblematisch, da es niemals notwendig ist, in einem Operator mehrere Hilfsregistersätze zwischenzulagern.

Informationsfluß im Operatorbaum In Abbildung 3.20 ist der Informationsfluß in einem Operatorbaum anhand eines einfachen Beispiels dargestellt. Die den Pfeilen zugeordneten Buchstaben geben an, welcher Registersatz des jeweiligen Produzenten für den Informationsfluß verwendet wird. Diese Zuordnung ist zwar nicht zwingend, sie wird aber von den bisher implementierten Operatoren eingehalten.

3.7 Zusammenfassung

Wir haben unser Laufzeitsystem AODB im Detail beschrieben. Das wesentliche Ziel bei der Entwicklung von AODB war die Reduktion der Prozessorkosten ohne eine Erhöhung der E/A-Kosten gegenüber herkömmlichen Laufzeitsystemen.

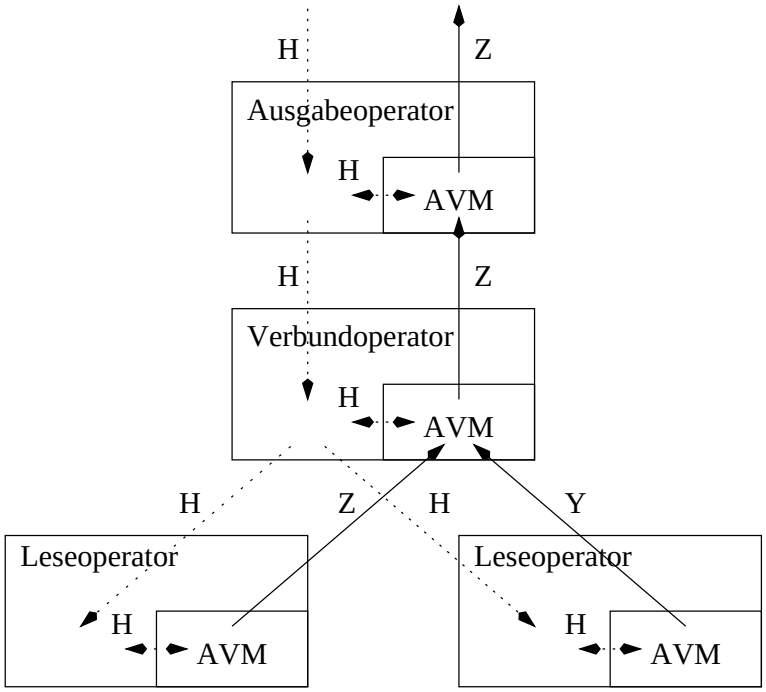


Abbildung 3.20: Informationsfluß im Operatorbaum

Wir haben dieses Ziel durch die Vermeidung nicht notwendiger Prozessorkosten erreicht. Die zentralen Techniken waren dabei die Verwendung des Interpretationsobjekte-Musters zur Bearbeitung von Sekundärspeicherdatenstrukturen und die Verwendung einer virtuellen Maschine zur Verarbeitung typisierter Information im Primärspeicher. Des weiteren konnten wir durch die Verfeinerung der Iteratorschnittstelle die Vorgänge bei der Anfragebearbeitung präziser steuern und so den Aufwand weiter reduzieren.

Die bisher beschriebenen Techniken sind allgemeiner Natur. In den folgenden Kapiteln werden einige Techniken, die speziell die Anfragebearbeitung in Datenlagern unterstützen, eingehender betrachten. Wir folgen dabei dem Weg der Daten vom Sekundärspeicher durch die Schichten des Datenbankverwaltungssystems zum Benutzer, wir beginnen also mit der tiefsten von uns behandelten Schicht.

Kapitel 4

Kompression

In diesem Kapitel beschäftigen wir uns mit der Integration von Kompression in Datenbankverwaltungssystemen im allgemeinen und in AODB im speziellen. Dazu führen wir in Abschnitt 4.1 zunächst in die Problematik ein, betrachten in Abschnitt 4.2 den aktuellen Stand der Forschung und stecken dann in Abschnitt 4.3 den Rahmen, in dem wir uns bewegen, ab. In Abschnitt 4.4 beschreiben wir für den Einsatz in Datenbanksystemen geeignete Kompressionsverfahren und stellen in Abschnitt 4.5 dar, wie diese auf der Ebene der logischen Satzschnittstelle in AODB integriert wurden. In Abschnitt 4.6 stellen wird den von uns zur Leistungsbewertung durchgeführten TPC-D-Benchmark [TPC95] vor.

4.1 Einleitung

Kompression ist eine weit verbreitete Technik in modernen Computersystemen. So verwendet man Kompression zum Beispiel für Audio-, Bild- und Videodaten in Multimediasystemen, zur Erstellung von Sicherungskopien oder auch zum Transport von Dateien durch das Internet.

Kompression hat drei Vorteile:

- sie reduziert die Kosten für Speichermedien (für Primär-, Sekundär- und Tertiärspeicher) und
- sie reduziert die Kosten für den Transfer zwischen den Speicherarten und kann so die Leistungsfähigkeit von E/A-lastigen Anwendungen verbessern und
- sie reduziert die Kosten für den Puffer, der für den Transfer zwischen den Speicherarten benötigt wird.

Der zentrale Nachteil ist, daß Kompression Rechenleistung benötigt, um die Daten einmal zu komprimieren und zu jeder Nutzung zu dekomprimieren. Dies kann die Leistungsfähigkeit rechenlastiger Anwendungen unter Umständen erheblich reduzieren.

Datenlagerverwaltungssysteme können aufgrund der großen Menge der zu verarbeitenden Daten von den Vorteilen der Kompression erheblich profitieren, sie laufen aber auch aufgrund der komplexen Anfragen Gefahr durch die von der Kompression verbrauchten Rechenleistung ausgebremst zu werden.

Da auch Standard-Datenbankanwendungen einen nicht unwesentlichen rechenintensiven Anteil haben (zum Beispiel Verbunde oder Aggregationen), ist Kompression im Bereich relationaler Datenbanken bislang wenig akzeptiert und die Hersteller von Datenbankverwaltungssystemen gehen nur langsam dazu über Kompression in ihre Produkte zu integrieren. So verwendet keines der weit verbreiteten Systeme *DB2 UniversalDatabase*, *Oracle* und *Informix Dynamic Server* Kompressionstechniken. Lediglich in einigen Nischen wie zum Beispiel bei *ADABAS* von der Software AG, *IQ* von Sybase oder *DB2 UniversalDatabase for OS/390*[1] von IBM werden Daten komprimiert abgelegt. Ein möglicher Grund dafür ist die Angst von steigenden Antwortzeiten. Betrachten wir dazu ein Beispiel.

Beispiel Wenn 80 MB Daten verarbeitet werden sollen und die Übertragungsgeschwindigkeit vom Sekundärspeicher in den Primärspeicher 80 MB/s beträgt, so müssen die Daten – zur Erreichung der maximalen Verarbeitungsgeschwindigkeit – in einer Sekunde weiterverarbeitet werden. Wenn die Daten nun in komprimierter Form nur 40 MB benötigen, so erfolgt die Übertragung in einer halben Sekunde. Um diesen Vorteil der Kompression vollständig zu nutzen, ist es erforderlich die Daten innerhalb einer halben Sekunde sowohl zu dekomprimieren als auch zu verarbeiten. □

Es ist offensichtlich, daß dadurch der Prozessor für Systeme, die ohnehin eine große Rechenlast verarbeiten müssen, schnell zum Engpaß werden kann. In diesem Fall können entweder die Vorteile der Kompression nicht genutzt oder sogar ins Gegenteil verkehrt werden. Da die Leistung der kommerziellen Datenbanksysteme in der Regel schon im OLTP-Betrieb durch die Rechenlast begrenzt ist (engl. CPU-bound), ist das Risiko steigender Antwortzeiten hier besonders groß.

Wenn wir also die Kompression zur Steigerung der Leistungsfähigkeit eines Datenlagerverwaltungssystems nutzen wollen, ist es notwendig, sorgfältig zwischen der zu erzielenden Reduktion der Transferkosten und der erforderlichen Steigerung des Rechenaufwands abzuwägen.

Wir werden zeigen, daß es möglich ist die Leistungsfähigkeit eines Datenlagers durch Kompression zu erhöhen. Dazu werden wir zunächst einige sehr einfache und mit wenig Rechenaufwand verbundene Kompressionsverfahren vorstellen. Anschließend zeigen wir, wie man diese

[1] In *DB2 UniversalDatabase for OS/390* erfolgt die Kompression allerdings hardwareunterstützt, so daß eine zusätzlich Prozessorlast hier vermieden wird.

Verfahren in ein Datenlagerverwaltungssystem integrieren und insbesondere wie man effizient komprimierte Information Kodieren und Dekodieren kann. Außerdem beschreiben wir, welche Eigenschaften von AODB für die erfolgreiche Integration nützlich oder gar erforderlich sind.

4.2 Stand der Forschung

Ein Großteil der Arbeiten zum Thema „Kompression in Datenbanken“ beschäftigt sich mit der Entwicklung neuer Kompressionalgorithmen oder mit der Bewertung vorhandener Algorithmen für den Einsatz in Datenbanken (zum Beispiel [Sev83, Cor85, RH93, IW94, ALM96, NR95, GRS98]). Unsere Arbeit unterscheidet sich von diesen Arbeiten in zwei wesentlichen Punkten: Zum einen beschäftigen wir uns mit der Integration von bekannten Verfahren in ein Datenbankverwaltungssystem und nicht mit dem Entwurf neuer Verfahren. Zum anderen liegt unser Hauptaugenmerk auf der Leistungsfähigkeit der Anfrageauswertung und wir werden daher auch die Ergebnisse unserer experimentellen Leistungsbewertung vorstellen. Alle anderen experimentellen Studien untersuchen nur die mögliche Speicherplatzersparnis und die Arbeiten, die sich mit der Leistungsfähigkeit der Verarbeitung befassen (zum Beispiel [GS91, SNG93, RHS95, GRS98]), enthalten keine umfangreichen Ergebnisse von Experimenten.

Es gibt noch zwei weitere Bereiche, in denen Kompression im Zusammenhang mit Datenbanken studiert wurde. Zum einen wurde an speziellen Implementierungstechniken (zum Beispiel für Verbundoperatoren [OG95]) gearbeitet. Zum anderen gibt es eine relativ große Anzahl von Arbeiten über komprimierte Indexstrukturen wie zum Beispiel VSAM [Wag73], Präfixkompression in B-Bäumen [Com79], Kompression der Beschreibungen von Rechtecken in R-Bäumen [GRS98] und Kompression von Bitvektor-Indexstrukturen [MZ92]. Diese Arbeiten sind orthogonal zu unserer Arbeit: Zum einen konzentrieren wir uns auf die Leistungsfähigkeit von Kompression für den Fall, daß weit verbreitete Implementierungstechniken (wie zum Beispiel Streu-Verbunde) verwendet werden, obwohl die Verfahren, die wir vorschlagen, in gleicher Weise für andere Techniken funktionieren. Zum anderen konzentrieren wir uns auf die Kompression von Basisdaten und achten darauf, daß beliebige Indexstrukturen weiterhin verwendet werden können.

4.3 Voraussetzungen

Um Kompression effizient in ein Datenbankverwaltungssystem zu integrieren, müssen zwei grundlegende Voraussetzungen erfüllt sein:

- Die eingesetzten Kompressionsverfahren dürfen nur wenig Rechenleistung beanspruchen.
- Die zu komprimierenden Granulate müssen möglichst klein sein.

Die Bedeutung dieser Voraussetzungen wurde schon von anderen Forschern entdeckt und diskutiert (zum Beispiel in [GS91, SNG93, RHS95, GRS98]). Wir werden die Argumente daher hier nur kurz zusammenfassen.

Die Verfahren dürfen nur wenig Rechenaufwand benötigen, da sonst die Gefahr besteht, daß die Reduktion der Übertragungszeit vom Sekundärspeicher in den Primärspeicher durch die für den zusätzlichen Rechenaufwand benötigte Zeit überkompensiert wird. Ein Beispiel für dieses Phänomen geben Goldstein, Ramakrishnan und Shaft [GRS98], die zeigen, daß es länger dauert eine Seite mit *gunzip* zu dekomprimieren als sie vom Sekundärspeicher in den Primärspeicher zu übertragen.

Des weiteren wird, wie das Beispiel in Abschnitt 4.1 zeigt, durch die Reduktion der vom Sekundärspeicher zu übertragenden Datenmenge bei gleichbleibender Übertragungsgeschwindigkeit die Menge der im Übertragungszeitraum zu verarbeitenden Daten ohnehin erhöht. Es ist daher sinnvoll bei der Auswahl eines Kompressionsverfahrens auf die letzten Prozente der *Kompressionsrate* zu Gunsten eines geringen Rechenaufwands zu verzichten.

Die Kompression der Daten kann prinzipiell auf vier Ebenen erfolgen: auf der Segment-Ebene, auf der Seiten-Ebene, auf der Satz-Ebene oder auf der Attribut-Ebene. Wir komprimieren auf Attribut-Ebene, d.h. einzelne Attribute eine Satzes können ohne Betrachtung anderer Attribute komprimiert und dekomprimiert werden. Wir wählen diesen Ansatz, da alle anderen Alternativen erhebliche Nachteile aufweisen (vgl. auch [GS91, SNG93, RHS95, GRS98]). Die Kompression auf Segment-Ebene ist praktisch nicht durchführbar, da zur Komprimierung und Dekomprimierung jeweils das gesamte Segment bekannt sei muß. Auf der Seiten-Ebene führt Kompression zu unterschiedlich großen komprimierten Seiten, so daß eine zusätzliche Abbildung auf physische Seiten notwendig wird. Des weiteren wird der Primärspeicher schlechter genutzt, da sowohl die physischen aus auch die komprimierten Seiten im Primärspeicher vorhanden sein müssen [GRS98]. Auf allen Ebenen oberhalb der Attribut-Ebene können nur Kompressionsverfahren verwendet werden, die nur Bytefolgen komprimieren und daher in der Regel zu langsam sind, wie zum Beispiel *gzip*. Die Kompression auf Attribut-Ebene ermöglicht dagegen die Semantik der Daten bei der Komprimierung zu berücksichtigen und so schnellere oder weniger aufwendige Verfahren zu verwenden. Außerdem ermöglicht nur die Kompression auf Attribut-Ebene nur die Daten zu dekomprimieren, die tatsächlich verwendet werden (vgl. auch [GS91, SNG93]). Zum Beispiel muß, zur Auswertung einer Anfrage nach den `Namen` aller `Angestellten` die älter als 60 sind, das `Alter`-Attribut aller `Angestellten` dekomprimiert werden, das `Name`-Attribut aber nur für diejenigen, die über 60 Jahre alt sind, und zum Beispiel die `Adresse` überhaupt nicht. Bei Kompression auf Satz-, Seiten- oder gar Segment-Ebene muß zum Zugriff auf ein Attribut immer das gesamte Objekt dekomprimiert werden. Eine weitergehende Beschreibung der Probleme, die beim Versuch Kompression auf höheren Ebenen in ein Datenbankverwaltungssystem zu integrieren auftreten, findet man bei Goldstein, Ramakrishnan und Shaft [GRS98].

Obwohl wir die Kompression auf den kleinstmöglichen Granulaten, den Attributen, verwenden, erfordert unsere Architektur, daß das selbe Verfahren für alle Werte ein Spalte verwendet wird. In obigem Beispiel würde dann das `Alter`-Attribut aller `Angestellten` in gleicher Weise komprimiert. Dies ist erforderlich, da es zum einen für den Benutzer zu aufwendig wäre für jedes Attribut in jedem Satz ein Kompressionsverfahren anzugeben und da es zum anderen auch für das System zu teuer wäre vor jedem Attributzugriff zunächst das zu verwendende Dekompressionsverfahren zu bestimmen. Es ist allerdings möglich unterschiedliche Spalten mit unterschiedlichen Verfahren zu komprimieren oder auch einige Spalten nicht zu komprimieren.

4.4 Konkrete Verfahren

In diesem Abschnitt stellen wir einige konkrete Kompressionsverfahren vor, die wir zur Verbesserung der Leistungsfähigkeit unseres Datenlagerverwaltungssystems verwendet haben. Im einzelnen sind dies die *Numerische Kompression*, die *Zeichenketten Kompression* und die *verzeichnisbasierte Kompression*. Außerdem beschreiben wir den Umgang mit `NULL`-Werten bei der Verwendung von Kompression.

Da alle drei Kompressionsverfahren auf unterschiedliche Fälle anwendbar sind und und außerdem kombiniert werden können, um mehrere Attribute einer Relation unterschiedlich zu komprimieren, gibt es im allgemeinen viele unterschiedliche Möglichkeiten eine Relation zu komprimieren. Obwohl die Wahl des falschen Verfahrens die Leistungsfähigkeit des Datenbankverwaltungssystems erheblich beeinflussen kann, denken wir nicht, daß die Auswahl eine geeigneten Kompressionsverfahrens ein zu großer zusätzlicher Aufwand für den Datenbankadministrator ist, da die richtige Wahl für eine gegebene Anwendung in den meisten Fällen offensichtlich ist. So fiel uns zum Beispiel die Entscheidung für die *verzeichnisbasierte* Kompression (4.4.3) für Aufzählungsattribute oder *Numerische* Kompression (4.4.1) für ganzzahlige Attribute bei unserer Implementierung des TPC-D-Benchmarks nicht schwer.

4.4.1 Numerische Kompression

Dieses von uns zur Kompression ganzzahliger Werte Verfahren basiert auf *Null suppression* und der Kodierung der Länge des komprimierten Wertes [RH93]. Eine ähnliche Technik wird auch in ADABAS, einem relationalen Datenbanksystem der Software AG verwendet [SAG94]. Die grundlegende Idee besteht darin führende Nullen in der Darstellung ganzer Zahlen zu streichen. In den meisten aktuellen Systemen werden ganze Zahlen durch vier Byte dargestellt, so daß die Zahl drei durch 30 auf 0 gesetzte und zwei auf 1 gesetzte Bits repräsentiert wird. Mit einem solchen Kompressionsverfahren könnte die Zahl drei also durch zwei Bits dargestellt werden. Das Problem bei dieser Art der Kompression besteht nun darin, daß die Information, wieviele Bits zur Darstellung verwendet werden, zum dekomprimieren benötigt wird. Wir werden ein

Verfahren darstellen mit dessen Hilfe diese Information effizient kodiert und dekodiert werden kann. Da dieses Verfahren unabhängig von dem verwendeten Kompressionsverfahren ist, geschieht dies allerdings erst in Abschnitt 4.5, nachdem wir die anderen Kompressionsverfahren beschrieben haben. Jetzt können wir allerdings schon festhalten, daß das Kodierungsverfahren nur funktioniert, wenn die komprimierten Daten an Byte Grenzen ausgerichtet sind. Das bedeutet, daß die Zahl drei durch ein Byte und nicht durch zwei Bits dargestellt wird. Das Ausrichten an Byte Grenzen ist ein Beispiel dafür, wie durch Abstriche bei der Kompressionsrate eine Reduktion des Rechenaufwands und damit eine Beschleunigung der Dekomprimierung erreicht werden kann.

Dieses Verfahren zur Kompression ganzer Zahlen kann auch für Datum-Felder verwendet werden, da ein Datum häufig durch die Anzahl der Tage, die es vor oder nach einem bestimmten Stichtag liegt, dargestellt wird. Wenn der Stichtag zum Beispiel der 2. Februar 1999 ist, kann der 4. Februar 1999 durch die Zahl 2 und der 22. Januar 1999 durch die Zahl -11 dargestellt werden. Offensichtlich können diese Zahlen in der gleiche Weise wie alle anderen ganzen Zahlen komprimiert werden.

Für Gleitkommazahlen, die unkomprimiert durch 8 Byte dargestellt werden (`Double`), verwenden wir ein spezielles „Kompressionsverfahren“. Da 8 Byte Gleitkommazahlen oft ohne Informationsverlust in 4 Byte dargestellt werden können, verwenden wir die 4 Byte Darstellung falls dies möglich ist.

Andere Formen numerischer Kompression, die wir nicht berücksichtigt haben, die aber in einigen Situationen sinnvoll sein können, findet man bei Ng und Ravishankar [NR95] und bei Goldstein, Ramakrishnan und Shaft [GRS98].

4.4.2 Zeichenketten Kompression

In SQL [ISO97] können Zeichenketten entweder als `CHAR(n)` oder als `VARCHAR(n)` definiert werden. In den meisten Datenbankverwaltungssystemen werden `CHAR(n)` Attribute durch den Inhalt eines Bereiches der festen Länge n repräsentiert. `VARCHAR(n)` Attribute werden hingegen durch den Inhalt eines Bereiches variabler Größe und durch die Länge dieses Bereiches repräsentiert. Eine einfache Lösung `VARCHAR` Attribute zu komprimieren besteht darin, die Längenangabe mit Hilfe der numerischen Kompression aus dem letzten Abschnitt zu komprimieren. `CHAR` Attribute können durch Umwandlung in `VARCHAR` Attribute und anschließende Kompression der Längenangabe komprimiert werden.

Wenn die Zeichenketten lang sind ist es häufig von Vorteil auch den Inhalt der Zeichenkette selbst zu komprimieren. Wenn die Ordnung der Zeichenketten nicht erhalten werden muß, kann diese weitere Komprimierung mit einem der klassischen Verfahren wie *Huffman coding* [Huf52], dem LZW Verfahren [Wel84] oder *Arithmetic coding* [WNC87] erfolgen. Falls ein ordnungserhaltendes Verfahren benötigt wird, kann das Verfahren von Blasgen, Casey und Es-

waran [BCE76] oder das Verfahren von Antoshenkov, Lomet und Murray [ALM96] verwendet werden. Die Verwendung dieser Verfahren ist unabhängig von der Komprimierung der Längenangabe und kann zusätzlich zu dieser erfolgen .

4.4.3 Verzeichnisbasierte Kompression

Die verzeichnisbasierte Kompression ist ein weit verbreitetes Kompressionsverfahren, das für alle Datentypen verwendet werden kann. Bei diesem Verfahren werden alle Werte, die ein Attribut annehmen kann, in einer separaten Datenstruktur, dem Verzeichnis, abgelegt. Der physischen Satz selbst muß dann nur noch einen Schlüssel enthalten, der den entsprechenden Wert im Verzeichnis eindeutig identifiziert. Das Verfahren ist besonders effizient, wenn ein Attribut nur wenige unterschiedliche Werte annehmen kann. Wenn ein Attribut zum Beispiel nur die Werte „Mannheim“ und „Passau“ annehmen kann, ist zur Repräsentation nur ein Bit erforderlich. Mit Hilfe dieses Bits kann der tatsächliche Attributwert aus dem Verzeichnis extrahiert werden.

Prinzipiell sind viele unterschiedliche Formen der verzeichnisbasierten Kompression vorstellbar. Wir haben eine sehr einfache und eingeschränkte Variante implementiert, bei der die Maximale Größe des Verzeichnisses und damit auch die Anzahl der zur Darstellung eines Attributs notwendigen Bits im voraus bekannt ist. Eine interessante und allgemeinere Variante der verzeichnisbasierten Kompression wurde von Antoshenkov, Lomet und Murray [ALM96] dargestellt.

4.4.4 `NULL`-Werte

In SQL gibt es für alle Datentypen die Möglichkeit des Vorkommens von `NULL`-Werten. Obwohl Integritätsbedingungen in vielen Anwendungen das Vorkommen von `NULL`-Werten unterbinden, muß ein System berücksichtigen, daß `NULL`-Werte vorkommen können und ein Kompressionsverfahren muß `NULL`-Werte eindeutig repräsentieren und identifizieren können.

Bei der verzeichnisbasierten Kompression können `NULL`-Werte einfach als ein weiterer möglicher Wert ins Verzeichnis aufgenommen werden. Im Falle der numerischen Kompression kann ein `NULL`-Wert als ein Attribut der Länge 0 dargestellt werden. Um `NULL` zum Beispiel von der ganzen Zahl 0 zur unterscheiden, wird die Zahl 0 als ein Attribut der Länge eins mit der Wert 0 dargestellt. Bei der Zeichenketten Kompression wird eine leere Zeichenkette als eine Zeichenkette mit der Länge 0 und die `NULL`-Zeichenkette als Zeichenkette mit der Länge `NULL` dargestellt.

4.5 Kodieren und Dekodieren der Kompressionsinformation

In diesem Abschnitt beschreiben wir, wie man die vorgestellten und andere Kompressionsverfahren in das Speichersystem eines Datenbankverwaltungssystems integrieren kann. Wie wir im letzten Abschnitt gesehen haben, ist es für die Effizienz vieler Kompressionsverfahren, die ein Ergebnis variabler Länge produzieren, entscheidend, daß die Längeninformation effizient kodiert und dekodiert werden kann. Außerdem muß der Zugriff auf auf die komprimierten Daten des Attributs im Satz effizient möglich sein. Das bedeutet, daß die Berechnung der Position des Attributs im Satz nur wenig Aufwand erfordern darf. In ADABAS zum Beispiel nimmt der Aufwand für den Zugriff auf ein Attribut mit der Position des Attributs im Satz zu, da zur Berechnung der Position zunächst die Länge aller vorhergehenden Attribute bestimmt werden muß. Um dies zu vermeiden kodiert unser Ansatz die Längeninformation jedes Attributs in einer festen Anzahl Bits und faßt die Längeninformationen aller Attribute in einem speziellen Teil am Anfang des Satzes zusammen. In den folgenden Abschnitten werden wir zunächst den daraus resultierenden Aufbau der komprimierten Sätze und anschließend unsere Kodierungs- und Dekodierungsalgorithmen beschreiben (vgl. [WKHM00]).

4.5.1 Aufbau der komprimierten Sätze

Abbildung 4.1 zeigt den Aufbau eines komprimierten Satzes. Danach kann ein Satz aus bis zu fünf Teilen bestehen:

- Der erste Teil eines Satzes enthält die komprimierten Werte aller Attribute, die durch verzeichnisbasierte Kompression oder durch ein anderes Kompressionsverfahren, das Werte fester Länge erzeugt, komprimiert wurden.
- Der zweite Teil enthält die kodierte Längeninformation für alle Attribute, die mit einem Kompressionsverfahren komprimiert wurden, das Werte variabler Länge erzeugt. Ein Beispiel für ein solches Verfahren ist die numerische Kompression.
- Der dritte Teil enthält die Werte von Attributen fester Länge. Dies sind zum Beispiel ganze Zahlen, Gleitkommazahlen oder Zeichenketten fester Länge (CHAR-Felder). Nicht dazu gehören Zeichenketten variabler Länge (VARCHAR-Felder) oder Zeichenketten fester Länge, die durch Kompression zu Zeichenketten variabler Länge geworden sind.
- Der vierte Teil enthält die komprimierten Werte von Attributen, die mit Hilfe eine Kompressionsverfahrens komprimiert wurden, das Werte variabler Länge erzeugt. Dies sind die Werte, deren Längen im zweiten Teil des Satzes abgelegt sind. Hier würde zum Beispiel auch die Länge eines Zeichenketten variabler Länge abgelegt, falls diese Länge komprimiert wurde. Falls eine solche Länge nicht komprimiert wurde, so wird sie im dritte Teil abgelegt.

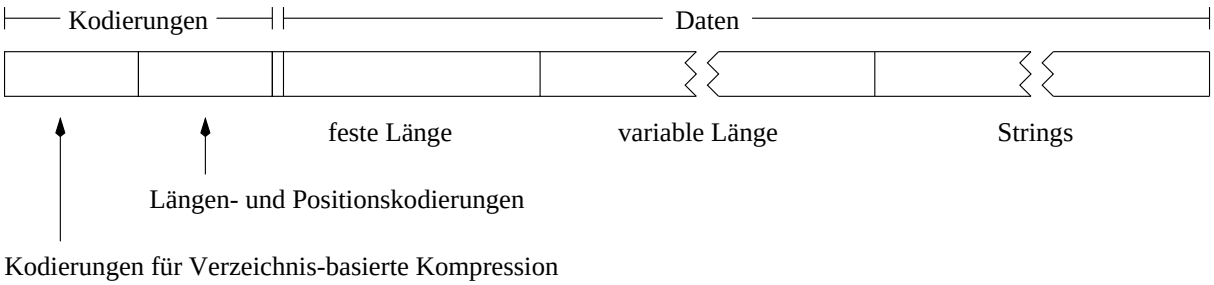


Abbildung 4.1: Aufbau eines komprimierten Satzes

- Der fünfte und letzte Teil eines Satzes enthält die Inhalte von Zeichenketten variabler Länge unabhängig davon, ob diese komprimiert wurden oder nicht.

Obwohl dies alles eher kompliziert aussieht, ist die Aufteilung in fünf Teile sehr natürlich. Wie wir schon in Abschnitt 3.4 gesehen haben, ist es sinnvoll den festen und den variablen Teil eines Satzes zu trennen. Die ersten drei Teile eines Satzes haben eine feste Größe, das bedeutet, daß diese Größe für alle Sätze eines Segments gleich ist. Daraus ergibt sich, daß auf die in den ersten drei Teilen enthaltenen Kompressionsinformationen und/oder Werte ohne Positionsberechnung direkt zugegriffen werden kann. Insbesondere kann auf unkomprimierte Attribute (sofern es sich nicht um Zeichenketten handelt) direkt zugegriffen werden, unabhängig davon, ob andere Attribute komprimiert sind oder nicht. Auch die Zusammenfassung aller Längenkodierungen komprimierter Attribute ist sinnvoll, da man dies in unserem schnellen Dekodierungsalgorithmus ausnuztzen kann (vgl. Abschnitt 4.5.3). Außerdem trennen wir kleine (komprimierte) Attribute variabler Länge von großen Zeichenketten-Attributen variabler Länge, da die Längeninformation für kleine Attribute weniger als ein Byte benötigt und die Längeninformation für große Zeichenketten-Attribute unter Umständen mehr als ein Byte benötigt und daher in einem zweistufigen Verfahren kodiert wird.

Offensichtlich bestehen nicht alle Sätze einer Datenbank aus diesen fünf Teilen. So bestehen zum Beispiel Sätze, die keine komprimierten Attribute enthalten, nur aus dem dritten und eventuell aus dem fünften Teil. (Außerdem haben alle Sätze eine Segments die gleiche Struktur und sie bestehen aus den gleichen Teilen, da alle Sätze eines Segments mit Hilfe der gleichen Verfahren komprimiert werden.)

4.5.2 Kodierung

Aus der Beschreibung des Aufbaus komprimierter Sätze wird offensichtlich, wie auf unkomprimierte und komprimierte Attribute fester Länge zugegriffen wird. Offen ist dagegen noch der Zugriff auf komprimierte Attribute variabler Länge. Im folgenden beschreiben wir, wie die Länge solcher Attribute kodiert und im zweiten Teil des Attributs abgelegt wird. In Abschnitt

Länge	NOT NULL	NULL erlaubt
0	—	000
1	00	001
2	01	010
3	10	011
4	11	100

Tabelle 4.1: Längenkodierung für ganze Zahlen

Länge	NOT NULL	NULL erlaubt
0	—	00
4	0	01
8	1	10

Tabelle 4.2: Längenkodierung für Gleitkommazahlen

4.5.3 beschreiben wir dann, wie diese Information wieder dekodiert wird.

Aus den vorangehenden Abschnitten ergibt sich, daß wir im wesentlichen an der Kodierung der Länge von komprimierten ganzzahligen und Gleitkommawerten interessiert sind. (Datumwerte können als ganzzahlige Werte dargestellt werden und für Zeichenketten ist die Längeninformation auch ein ganzzahliger Wert.) Wegen der Ausrichtung an Byte-Grenzen kann ein komprimierter ganzzahliger Wert 1, 2, 3 oder 4 Byte lang sein. Daher können wir die Länge eines solchen Wertes in zwei Bit kodieren (vgl. linke Spalte von Tabelle 4.1). Wenn das betrachtete Attribut auch NULL-Werte annehmen kann, benötigen wir drei Bits zur Kodierung der Länge, da diese nun 0, 1, 2, 3 oder 4 Byte betragen kann (vgl. rechte Spalte von Tabelle 4.1).

Tabelle 4.2 enthält entsprechende Bitmuster für Gleitkommazahlen. Diese können entweder, falls NULL-Werte verboten sind, in 4 oder 8 Byte dargestellt werden, oder, falls NULL-Werte erlaubt sind, in 0, 4 oder 8 Byte dargestellt werden. Zur Kodierung werden dann entweder ein oder zwei Bits benötigt.

Wenn ein Satz mehrere komprimierte Attribute variabler Länge enthält, so versuchen wir möglichst viele Längenkodes in einem Byte zusammenzufassen. Gleichzeitig stellen wir aber sicher, daß zum Zugriff auf ein Attribut immer nur auf ein Kodierungsbyte zugegriffen werden muß, wir richten also auch die Kodierungsinformation an Byte-Grenzen aus. Wir illustrieren dies mit einem Beispiel. In diesem Beispiel soll ein Tupel mit dem Schema

$$\mathcal{R} = \langle \texttt{a:int, b:int, c:double not null,} \\ \texttt{d:int, e:int, f:int not null} \rangle$$

komprimiert werden. Wenn alle Attribute wie in Abschnitt 4.4.1 beschrieben komprimiert wer-

den, dann werden die Längenkodierungen aller sechs Attribute in der folgenden Weise in zwei Byte zusammengefaßt.

Byte 1	—	Bit_1^a	Bit_2^a	Bit_3^a	Bit_1^b	Bit_2^b	Bit_3^b	Bit_1^c
Byte 2	Bit_1^d	Bit_2^d	Bit_3^d	Bit_1^e	Bit_2^e	Bit_3^e	Bit_1^f	Bit_2^f

Dabei bezeichnet Bit_i^x das i-te Bit der Längenkodierung für Attribut x und „—“ bedeutet, daß das erste Bit von Byte 1 nicht benutzt wird.

4.5.3 Dekodierung

Mit Hilfe des Kodierungsverfahrens aus dem letzten Abschnitt können wir die Länge eine komprimierten Attributs eines Satzes einfach bestimmen: wir müssen lediglich die entsprechenden Bits aus dem Längenkodierungsteil des Satzes extrahieren und können dann die Länge in einer Tabelle wie Tabelle 4.1 oder Tabelle 4.2 nachschlagen. Um den Wert eines Attributs zu bestimmen, müssen wir allerdings mehr tun. Dazu benötigen wir die *Position* des Wertes im Satz, die von den Längen aller Attribute abhängt, die im Satz vor dem betrachteten Attribut gespeichert sind.

Ein einfaches Verfahren, um die Position des i-ten Attributs zu bestimmen, wäre eine Schleife von $j = 1$ bis $i - 1$, in der jeweils die Länge des j-ten Attributs bestimmt wird. Die Position wäre dann die Summe dieser Längen. Diese Vorgehensweise wäre allerdings mit einem sehr hohen Rechenaufwand verbunden, da dazu die Dekodierung der Längeninformation von $i - 1$ Attributen erforderlich wäre. Wir vermeiden diesen Rechenaufwand, indem wir alle möglichen Positionen eines Attributs im Satz in *Dekodierungstabellen* materialisieren. Um die Vorgehensweise zu erläutern, führen wir unser Beispiel aus dem letzten Abschnitt fort und betrachten ein konkretes Tupel aus der Relation $\mathcal{R}$. Wenn der komprimierte Wert des Attributs a 2 Byte lang ist, der von b 0 Byte, der von c 8 Byte, der von d 3 Byte, der von e 0 Byte und der von f 4 Byte, dann sehen die zwei Byte, die die Länge des Tupels kodieren wie folgt aus (die Kodes stammen aus den Tabellen 4.1 und 4.2 und das unbenutzte Bit von Byte 1 ist 0):

Byte 1							
0	0	1	0	0	0	0	1

Byte 2							
0	1	1	0	0	0	1	1

Nun wollen wir die Position des Attributs e in diesem Tupel bestimmen. Dazu betrachten wir zunächst die linke Dekodierungstabelle aus Abbildung 4.2. Aus der dem ersten Byte unseres Tupels entsprechenden Zeile dieser Tabelle erfahren wir, daß die Gesamtlänge der Attribute a, b und c 10 Byte beträgt. Aus der dem zweiten Byte entsprechenden Zeile der rechten Tabelle entnehmen wir, daß wir für den Zugriff auf das Attribut e noch 3 Byte dazuaddieren müssen. Des weiteren erfahren wir aus der rechten Tabelle, daß die Längenkodierung für Attribut e 0 ist,

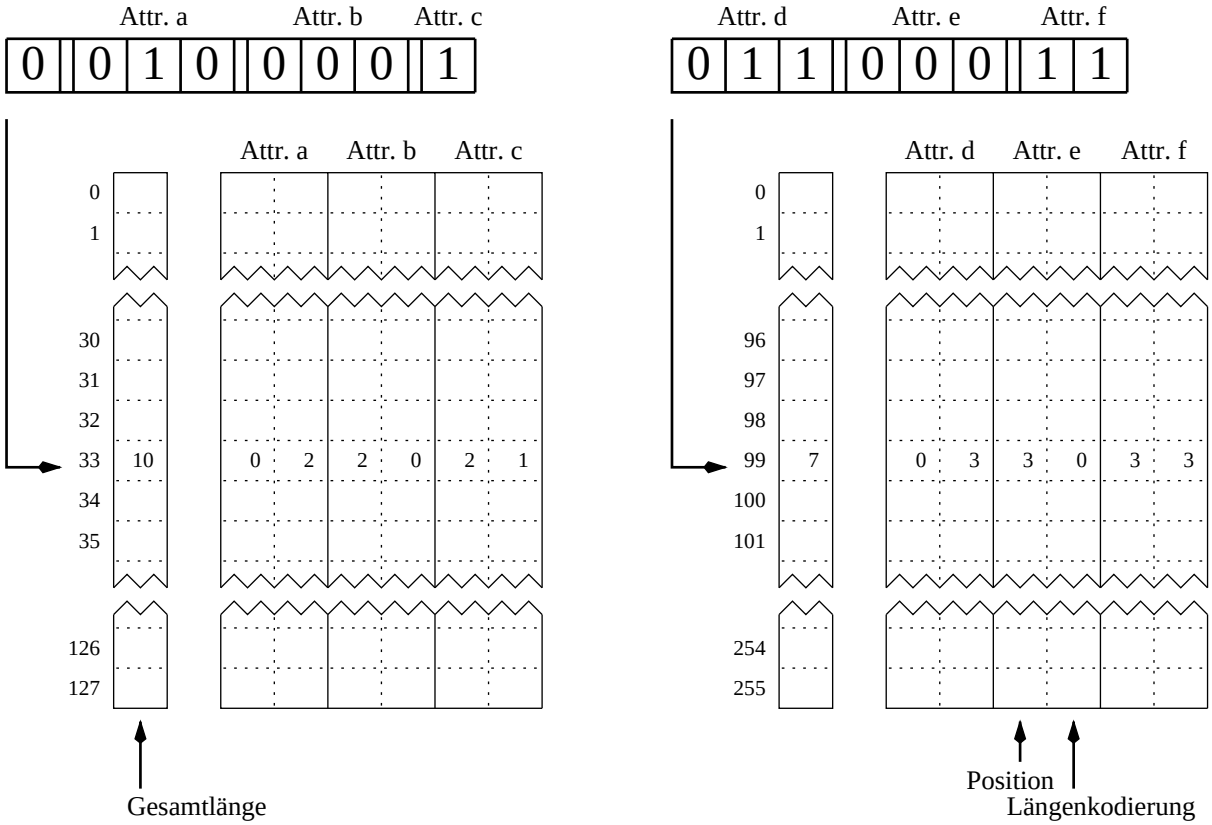


Abbildung 4.2: Dekodierung

was bedeutet (nach Tabelle 4.1), daß es sich um einen NULL-Wert handelt und wir die Position eigentlich nicht benötigen.

Im allgemeinen haben wir d Dekodierungstabellen für eine Relation, für deren Tupel der Längenkodierungsteil (der zweite Teil) d Byte lang ist. Jede Dekodierungstabelle hat 2^b Zeilen, wobei b die Anzahl der Bits ist, die in dem entsprechenden Längenkodierungsbyte verwendet werden. In unserem Beispiel ist $b = 7$ ($\hat{=}$ 128 Zeilen) für das erste und $b = 8$ ($\hat{=}$ 256 Zeilen) für das zweite Byte. Jede Zeile hat ein Feld für die *Gesamtlänge* aller in diesem Byte kodierten Attribute und je kodiertem Attribut ein *Positions-* und ein *Längenfeld*. Wenn diese Tabellen vorhanden sind, können die Position und die Länge eines Attributs im Satz mit dem in Abbildung 4.3 dargestellten Verfahren bestimmt werden.

Wenn wir uns unser Beispiel genauer betrachten, stellen wir auch fest, daß die Dekodierungstabellen sehr klein sind. Die Relation $\mathcal{R}$ aus unserem Beispiel benötigt eine 3,5 KByte (128 Einträge * 28 Byte[2]) große Dekodierungstabelle für Byte 1 und eine 7 KByte (256 Einträge * 28

[2]Diese 28 Byte bestehen aus $3 \cdot 8$ Byte für Position und Länge je eines Attributs und 4 Byte für die Gesamtlänge der Attribute

```
position_und_laenge(
  attr,        /* Attributnummer                                  */
  kodeByte[],  /* Kodierungsinformation eines Tupels              */
  tabelle[][], /* Kodierungstabellenfeld. tabelle[i][j]           */
               /* bezeichnet die j-te Zeile der i-ten Tabelle.    */
  byteNo       /* Nummer des Bytes in KodeByte fuer attr          */
  ) {
  pos    = 0
  laenge = 0
  for(j = 0; j < byteNo - 1; ++j) {
    pos = pos + tabelle[j][kodeByte[j]].gesamtlaenge
  }
  pos    = pos + tabelle[byteNo][kodeByte[byteNo]].pos(attr)
  laenge =       tabelle[byteNo][kodeByte[byteNo]].laenge(attr)
  return <pos, laenge>
}
```

Abbildung 4.3: Verfahren zur Dekodierung der Komprimierungsinformation

Byte) große Dekodierungstabelle für Byte 2, also insgesamt 10,5 KByte. Wir erwarten daher, daß diese Dekodierungstabellen im Primärspeicher gehalten werden können, ebenso wie alle anderen Metadaten. Falls wir bereit wären mehr Primärspeicher zu investieren, könnten wir alle möglichen Positionen und Längen in einer einzigen Tabelle zusammenfassen, statt eine Tabelle pro Byte zu verwenden. Damit wäre es möglich die Dekodierung in konstanter Zeit durchzuführen. Allerdings würde eine solche *universelle* Dekodierungstabelle für die Relation $\mathcal{R}$ etwa 1,6 MByte Primärspeicher (2^{15} Einträge * 52 Bytes[3]) benötigen. (Offensichtlich sind auch noch viele Kompromißmöglichkeiten zwischen den dargestellten Möglichkeiten vorstellbar.)

4.5.4 Integration in AODB

Die Integration der hier beschriebenen Techniken zur Kompression und zum Zugriff auf die komprimierten Daten ist relativ einfach. Alle erforderlichen Änderungen des Laufzeitsystems können innerhalb des in Abschnitt 3.4 beschriebenen Moduls *Logische Satzschnittstelle* erfolgen. Dem der logischen Satzschnittstelle zugrundeliegenden physischen Satzspeicher ist der Inhalt der physischen Sätze unbekannt, so daß der Inhalt auch keinen Einfluß auf die dort stattfindende Verarbeitung hat. Oberhalb der logischen Satzschnittstelle werden die Daten in Registern verarbeitet und transportiert, so daß der Inhalt und Aufbau der physischen Sätze auch hier keine Rolle spielt.

Das System muß also lediglich um eine neue Klasse von Tupelobjekten *TUP_CTuple* und um eine entsprechende Schemaklasse *PSM_CSchema* erweitert werden. Die beiden Klassen sind –

[3] Diese 52 Byte bestehen aus $6 \cdot 8$ Byte für Position und Länge je eines Attributs und 4 Byte für die Gesamtlänge der Attribute

wie *TUP_NTuple* und *PSM_NSchema* – von *TUP_ICTuple* und *PSM_PSchema* abgeleitet. Die Dekodierungstabellen sind dabei im Schema abgelegt, da sie so für jede Relation nur einmal im Primärspeicher vorhanden sein müssen. Bei dem in Abbildung 4.3 dargestellten Verfahren, stammen *tabelle* und *byteNo* also aus dem Schemaobjekt, *kodeByte* aus dem Tupelobjekt und *attr* vom Benutzer.

4.6 Leistungsvergleich

In diesem Abschnitt beschreiben wir die Ergebnisse TPC-D-Benchmarks, der zur Bewertung der in den letzten beiden Abschnitten dargestellten Techniken durchgeführt wurde. Um die durch die Integration von Kompression in ein Datenlagerverwaltungssystem erreichbaren Leistungsveränderungen zu untersuchen, vergleichen wir die Ergebnisses des TPC-D-Benchmarks auf einer unkomprimierten Datenbank mit denen auf einer komprimierten Datenbank. Dazu beschreiben wir in Abschnitt 4.6.1 zunächst die Details unserer Implementierung des Benchmarks. Anschließend betrachten wir die Benchmark-Ergebnisse, die die Größe der Datenbanken, die Ladezeiten und die Laufzeiten der 17 Anfragen und zwei Änderungsoperationen enthalten.

4.6.1 Implementierung der Anfragen und Änderungsoperationen

Wie schon erwähnt, haben wir AODB als Plattform für die Benchmarks verwendet. Bei der Konstruktion der Pläne haben wir uns von Plänen kommerzieller Datenbankverwaltungssysteme leiten lassen. Es handelt sich meist um links tiefe Pläne mit abschließender Gruppierung und Aggregation (wir haben also keine frühe Aggregation wie bei Yan und Larson [YL94, YL95] oder Chaudhuri und Shim [CS94] berücksichtigt). Für die Verbundoperationen haben wir den GRACE-Streu-Verbund und den blockorientierten Geschachtelte-Schleifen-Verbund (engl. *Blockwise-Nested-Loop-Join*) verwendet. Für Gruppierung und Aggregation kamen streubasierte Verfahren zum Einsatz. Die Pläne und die Speicherzuteilung waren für die Anfragen auf der komprimierten und der unkomprimierten Datenbank identisch. Wir haben zunächst einen guten Plan und eine sinnvolle Speicherzuteilung für die unkomprimierte Datenbank bestimmt und diesen Plan dann bezüglich beider Datenbanken ausgewertet. Dieser Ansatz war eher konservativ, da wir auf der komprimierten Datenbank mit spezifischen Plänen bessere Antwortzeiten erzielen können. Die Ergebnisse können daher – bezüglich unserer Implementierung – als untere Grenze der möglichen Leistungssteigerung beim TPC-D-Benchmark gesehen werden. Drei der verwendeten Pläne können Anhang A entnommen werden.

Wir haben die TPC-D Daten mit folgenden Verfahren komprimiert:

- Für alle Attribute der Typen `INTEGER` und `DECIMAL` haben wir die numerische Kompression verwendet (vgl. Abschnitt 4.4.1).

- Für alle Indikatoren haben wir die verzeichnisbasierte Kompression verwendet (vgl. Abschnitt 4.4.3).

- Alle Attribute des Typs CHAR wurden wie Attribute des Typs VARCHAR behandelt und die Längeninformation wurde, wie in Abschnitt 4.4.2 beschrieben, komprimiert. Wir haben keine weiteren Kompressionsverfahren – wie zum Beispiel Huffman-Kodierung [Huf52] – zur weiteren Kompression von Zeichenketten verwendet. Wir hatten den Eindruck, daß die Verwendung solch leistungsfähiger Kompressionsverfahren für lange Zeichenketten (zum Beispiel für die Kommentare in den TPC-D Tabellen) den Vergleich in unangemessenerer Weise zu Gunsten der komprimierten Datenbank beeinflußt hätte. Die Verwendung dieser Verfahren hätte zu hohen Kompressionsraten (und reduzierter E/A-Last) geführt, ohne die Nachteile in Kauf nehmen zu müssen, da diese Attribute in den TPC-D Anfragen kaum genutzt werden.

- Datumsfelder wurden nicht komprimiert.

- Die Relationen *Region* und *Nation* wurde auch nicht komprimiert, da sie auf jeweils eine Seite des Sekundärspeichers passen.

Wir haben alle TPC-D Anfragen und Änderungsoperationen auf komprimierten und unkomprimierten Datenbanken des Skalierungsfaktors 1 ausgeführt. Der verwendete Rechner war eine SUN UltraSparc 2 (300 MHz) mit 256 MByte Primärspeicher. Zur Ausführung der Anfragen haben wir den für Daten verwendeten Primärspeicher allerdings auf 24 MByte begrenzt. Die Datenbanken wurden auf einer 9 GByte Seagate Barracuda Festplatte abgelegt und wir haben eine zweite identische Festplatte zur Speicherung von Zwischenergebnissen verwendet. Das Betriebssystem war Solaris 2.6. Die von AODB verwendete Seitengröße war 4 KByte.

4.6.2 Größe der Datenbanken

In Tabelle 4.3 ist die Größe der komprimierten und der unkomprimierten Datenbank dargestellt. Für die unterschiedlichen Relationen haben wir Kompressionsraten zwischen 45% und 10% erzielt. Die höchste Kompressionsrate haben wir für *Lineitem* erreicht, da diese Relation viele numerische Werte enthält, die wir komprimiert haben. Da der Anteil (langer) Zeichenketten-Attribute, von denen nur die Längeninformation komprimiert wurde, in den anderen Relationen relativ hoch ist, habe wir dort keine hohen Kompressionsraten erzielt. Als Faustregel kann man feststellen, daß für die Attribute, die wir komprimiert haben, die Kompressionsrate bei 50% lag. Wie schon erwähnt, habe wir die Relationen *Region* und *Nation* nicht komprimiert, da sie jeweils auf eine 4-KByte-Seite passen.

Relation	*komprimiert*	*unkomprimiert*
Lineitem	427,360 KByte	758,540 KByte
Order	132,164 KByte	177,900 KByte
Partsupp	112,588 KByte	124,600 KByte
Part	24,120 KByte	29,680 KByte
Customer	22,916 KByte	28,256 KByte
Supplier	1,412 KByte	1,640 KByte
Nation	4 KByte	4 KByte
Region	4 KByte	4 KByte
gesamt	720,568 KByte	1,120,634 KByte

Tabelle 4.3: Größe der komprimierten und unkomprimierten Relationen

Relation	*komprimiert*	*unkomprimiert*
Lineitem	428.5 s	303.6 s
Order	88.5 s	59.1 s
Partsupp	42.9 s	36.2 s
Part	13.9 s	9.2 s
Customer	11.2 s	8.2 s
Supplier	1.2 s	1.0 s
gesamt	586.2 s	417.3 s

Tabelle 4.4: Ladezeiten der komprimierten und unkomprimierten Relationen

4.6.3 Laden der Datenbanken

In Tabelle 4.4 sind die Ladezeiten für die komprimierte und die unkomprimierte Datenbank dargestellt. Wir beobachten dabei, daß die Ladezeiten für die komprimierten Relationen um 20-50% über denen der unkomprimierten Relationen liegen. Die Kompression reduziert zwar die Kosten für das Schreiben neuer Seiten auf den Sekundärspeicher, aber das Laden der Datenbank ist durch die verfügbare Rechenleistung beschränkt, da es die zeichenweise Interpretation der Eingabedatei beinhaltet. Wir haben während des Ladens einer Datenbank typischerweise Prozessorauslastungen von über 90% beobachtet. Da für die Komprimierung ein weiterer (hoher) Rechenaufwand anfällt (der den zur Dekomprimierung erforderlichen Aufwand deutlich übersteigt), ist dieser Anstieg der Ladezeiten nicht verwunderlich. Die Ladezeiten für die Relationen *Region* und *Nation* haben wir nicht angegeben, da zu klein waren, um von unserer Messung erfaßt zu werden.

Anfrage / Änderung	*komprimiert*			*unkomprimiert*		
	Zeit	Prozessor		Zeit	Prozessor	
Q1	42.4 s	33.3 s	78.5 %	73.7 s	22.7 s	30.8 %
Q2	13.7 s	5.0 s	36.5 %	17.9 s	3.0 s	16.8 %
Q3	88.5 s	52.0 s	58.8 %	126.0 s	43.4 s	34.4 %
Q4	67.9 s	38.8 s	57.1 %	113.3 s	33.7 s	29.7 %
Q5	61.5 s	42.3 s	68.8 %	98.4 s	31.6 s	32.1 %
Q6	43.0 s	22.8 s	53.0 %	74.2 s	15.8 s	21.3 %
Q7	71.9 s	46.5 s	67.7 %	105.8 s	34.5 s	32.6 %
Q8	66.8 s	38.4 s	57.5 %	104.1 s	23.6 s	22.7 %
Q9	136.6 s	95.4 s	69.8 %	174.8 s	78.0 s	44.6 %
Q10	96.1 s	45.5 s	47.4 %	131.5 s	34.1 s	25.9 %
Q11	11.4 s	4.6 s	40.4 %	14.9 s	3.0 s	20.1 %
Q12	81.2 s	69.1 s	85.1 %	106.2 s	44.7 s	42.1 %
Q13	57.2 s	27.9 s	48.8 %	97.0 s	20.9 s	21.6 %
Q14	47.8 s	24.0 s	50.2 %	78.6 s	16.6 s	21.1 %
Q15	44.1 s	23.8 s	54.0 %	75.1 s	18.0 s	24.0 %
Q16	14.6 s	13.0 s	89.0 %	16.7 s	11.0 s	65.9 %
Q17	74.4 s	39.6 s	53.2 %	156.9 s	31.0 s	19.8 %
UF1	0.6 s	0.6 s	100.0 %	0.3 s	0.3 s	100.0 %
UF2	124.4 s	30.5 s	24.5 %	216.1 s	36.3 s	16.7 %
gesamt	1237.4 s	694.6 s	56.1 %	1915.5 s	534.7 s	27.9 %

Tabelle 4.5: TPC-D Power Test

4.6.4 Laufzeiten der Anfragen und Änderungsoperationen

In Tabelle 4.5 sind die Laufzeiten, Prozessorkosten und Prozessorauslastung der 17 Anfragen und zwei Änderungsoperationen des TPC-D-Benchmarks dargestellt. Wie in Abschnitt 4.6.1 beschrieben, wurden diese Ergebnisse mit den für die unkomprimierte Datenbank entworfenen Plänen erzielt. Ein erster Blick auf die Ergebnisse der 17 Anfragen zeigt, daß die Anfragen auf der komprimierten Datenbank niemals langsamer als die auf der unkomprimierten Datenbank sind. Das bedeutet, daß für diese Anfragen die Gewinne, die durch die Reduktion der E/A-Last erzielt wurden, die durch zusätzliche Rechenlast erzeugten Verluste übersteigen. Die durch Kompression erzielten Leistungsverbesserungen hängen natürlich von der Art der Anfrage ab, d.h. von der Selektivität der Prädikate, der Anzahl der Verbunde, den an den Verbunden beteiligten Relationen, der Anwesenheit von `ORDER BY` und `GROUP BY` Klauseln und von den in den Ausdrücken und im Ergebnis der Anfrage verwendeten Attributen. In sechs Fällen (Q1, Q4, Q6, Q13, Q15 und Q17) wurde die Laufzeit durch die Kompression um 40% oder mehr verbessert und nur in einem Fall (Q16) unterschritt der Gewinn 20%.

Wenn wir die Prozessorkosten und die Prozessorauslastung der Anfragen betrachten, sehen wir, wie wichtig es war die für die Kompression verwendete Rechenleistung so gering wie möglich zu halten. Die Prozessorauslastung erreicht zwar nie 100%, aber Auslastungen von 70% sind bei Verwendung der komprimierten Datenbank üblich. Allein ein naiver Dekodierungsalgorithmus, der keine Dekodierungstabellen verwendet (vgl. Abschnitt 4.5.3), würde die Prozessorkosten der Dekomprimierung für die meisten Anfragen verdreifachen. Die Verdreifachung dieser Kosten würde nicht nur die Vorteile der reduzierten E/A-Kosten kompensieren, sondern sogar zu längeren Laufzeiten für die komprimierte Datenbank als für die unkomprimierte Datenbank führen.

Bei der Betrachtung der Laufzeiten der Änderungsoperationen stellen wir fest, daß die Kompression die Laufzeit der Änderungsoperation UF1 verdoppelt. UF1 fügt ungefähr 7500 Tupel, die in einer Textdatei vorliegen, in die Datenbank ein. Der Leistungsrückgang durch Kompression entspricht dem, den wir schon bei den Ladezeiten gesehen haben (vgl. Abschnitt 4.4). Die Tupel können mit relativ wenig E/A-Aufwand an die Tabellen *Order* und *Lineitem* angehängt werden, so daß die Kosten der Interpretation der Textdatei die Laufzeit dominieren. Durch den zusätzlichen Aufwand zur Komprimierung der Tupel steigt die Laufzeit im komprimierten Fall an. Im Fall der Änderungsoperation UF2 liegt die komprimierte Datenbank wieder vorne. UF2 löscht ungefähr 7500 *Order*- und *Lineitem*-Tupel und benötigt einige Sekundärspeicherzugriffe, um die zu löschenden Tupel zu finden. Durch die Kompression wird der Aufwand für die Sekundärspeicherzugriffe reduziert und in diesem Fall hat die Kompression einen weiteren Vorteil: es gibt kaum zusätzlichen Rechenaufwand, da die zu löschenden Tupel nicht vorher dekomprimiert werden müssen. (Um die richtigen Tupel zu finden, müssen nur die Primärschlüssel dekomprimiert werden.)

4.7 Ergebnis

Wir haben gezeigt, wie Kompression in das Laufzeitsystem eines Datenbankverwaltungssystems integriert werden kann. Im Gegensatz zu früheren Veröffentlichungen zu diesem Thema haben wir die Integration im Detail beschrieben. Unsere Erfahrungen haben gezeigt, daß es gerade diese Details sind, die über den Erfolg entscheiden. Kompression kann nur erfolgreich, d.h. zur Reduktion der Antwortzeiten, in ein Datenbankverwaltungssystem integriert werden, wenn es gelingt die Prozessorkosten im Griff zu behalten. Wir haben daher der Auswahl der zu verwendenden Kompressionstechniken eher auf eine geringen Rechenaufwand als auf eine gute Kompressionsrate geachtet. Außerdem haben wir gezeigt, wie man ein Tupel, das komprimierte Attribute enthält, so strukturiert, daß ein direkter Zugriff auf Felder fester Länge möglich ist und daß einzelne Attribute – ohne Einfluß auf die Zugriffzeiten für die anderen Attribute – komprimiert werden können. Außerdem haben wir sehr effiziente Kodierungs- und Dekodierungsalgorithmen entwickelt, so daß der Aufwand für den Zugriff auf komprimierte Attribute nur wenig größer ist als der für den Zugriff auf unkomprimierte Attribute.

Wir haben gesehen, das die – insbesondere in Bezug auf den Rechenaufwand – sehr effiziente Implementierung unseres Laufzeitsystems AODB eine sinnvolle oder sogar notwendige Voraussetzung für die erfolgreiche Integration war.

Das die beschriebenen Maßnahmen tatsächlich zu einer erfolgreichen Integration von Kompression in unser Laufzeitsystem geführt hat, haben unsere Experimente gezeigt. Wir haben den TPC-D-Benchmark auf einer komprimierten und einer unkomprimierten Datenbank durchgeführt. Dabei wurden die Antwortzeiten beim Übergang von der unkomprimierten zur komprimierten Datenbank in vielen Fällen halbiert.

Kapitel 5

Anfragemuster

Semantische Anfragemuster sind Muster, die man bei der Betrachtung einer großen Anzahl von Anfragen eines bestimmten Anwendungsbereichs findet. In diesem Kapitel stellen wir einige Muster vor, die wir bei der Nutzung von Datenlagern entdeckt haben, und beschreiben wie das Vorhandensein dieser Muster zur Beschleunigung der Anfrageauswertung genutzt werden kann. Dazu führen wir in Abschnitt 5.1 in das Thema ein und beschreiben anschließend in den Abschnitten 5.2 und 5.4–5.6 vier Anfragemuster und Möglichkeiten diese nutzen. In Abschnitt 5.3 werden gemeinsame Voraussetzungen der Abschnitte 5.4–5.6 dargestellt.

5.1 Einleitung

Einer der Hauptvorteile deklarativer Anfragesprachen ist die Möglichkeit der Anfrageoptimierung. Die Äquivalenz des Kalküls – die formale Grundlage einer deklarativen Anfragesprache – und der Algebra – die Sprache zur Beschreibung von Auswertungsplänen – ist die Grundlage für diese Möglichkeit. Um eine Anfrage tatsächlich optimieren zu können, müssen zur Anfrage äquivalente Auswertungspläne mit unterschiedlichen Kosten existieren. Die Menge der zur ursprünglichen Anfrage äquivalenten Auswertungspläne definiert den Suchraum, den ein Anfrageoptimierer auf der Suche nach einen günstigen Plan durchsucht. Es gibt zwei Gründe dafür, daß der Suchraum mehr als einen Plan enthält:

- Algebraische Äquivalenzen ermöglichen die Konstruktion unterschiedlicher Pläne auf der logischen Ebene.
- Aufgrund unterschiedlicher Implementierungen algebraischer Operatoren gibt es Alternativen auf der physischen Ebene.

Die Auswahl des bezüglich der algebraischen Äquivalenzen optimalen Plans bezeichnet man als *algebraische Optimierung* und die Auswahl der besten Implementierung eines algebrai-

schen Operators als *physische Optimierung*. Allerdings werden diese beiden Optimierungsarten in der Regel nicht unabhängig voneinander durchgeführt, da der Vergleich algebraisch äquivalenter Pläne ohne die Kenntnis der Implementierung und der daraus resultierenden Kosten der verwendeten Operatoren nicht sinnvoll ist.

Traditionell gibt es unterschiedliche Implementierungen für einen algebraischen Operator, so daß eine 1:n-Beziehung zwischen Operatoren und ihren Implementierungen besteht. So gibt es zum Beispiel für den Verbundoperator viele unterschiedliche Implementierungen wie den Geschachtelte-Schleifen-Verbund (engl. *Nested-Loop-Join*), den Sortier-Misch-Verbund (engl. *Sort-Merge-Join*) oder den Streu-Verbund (engl. *Hash-Join*) [Gra93]. Unserer Hypothese ist, daß man weitere Effizienzgewinne durch spezielle Implementierungen für Spezialfälle eines Standard-Operators oder für Operatorkombinationen erzielen kann. Wir werden vier Spezialfälle identifizieren, entsprechende Operatoren definieren und für diese effiziente Implementierungen angeben. Jeder Fall wird eine Absicht des Benutzers darstellen. Wir werden sehen, daß diese Absicht stets durch ein bestimmtes Muster in der Anfrage dargestellt wird. Wir werden diese Spezialfälle daher als *semantische Anfragemuster* bezeichnen. Der erste Operator optimiert einen Spezialfall eines 1:n-Verbundes. Der zweite und der dritte Operator eignen sich für bestimmte Muster von Gruppierung und Aggregation. Der vierte Operator eignet sich für ein Muster, das einer Kombination von Gruppierung, Aggregation und Verbund entspricht.

Offensichtlich muß man bei der Identifizierung solcher semantischer Anfragemuster stets darauf achten, diese nicht nur unter dem Aspekt der effizienten Implementierung sondern auch unter dem Aspekt der möglichst universellen Verwendbarkeit zu betrachten. Es ist einsichtig, daß ein sehr spezifisch auf eine Anfrage zugeschnittenes Muster sehr effizient zu implementieren ist, allerdings ist es dann auch für alle anderen Anfragen nur von geringem Wert. Wir betrachten Fälle, die für OLAP-Anwendungen wichtig sind und auch von anderen als wichtig identifiziert wurden. Alle dargestellten Muster sind im TPC-D-Benchmark [TPC95] enthalten. Ein positiver Seiteneffekt dieser Tatsache ist, daß wir die TPC-D Daten zur Evaluierung der Leistungsfähigkeit unserer Implementierung verwenden konnten.

Die Bedeutung der von uns betrachteten Fälle: Schlüssel-Fremdschlüssel-Verbunde und Gruppierung und Aggregation wird auch am Beispiel des TPC-D-Benchmarks deutlich. Von den 17 Anfragen des Benchmarks enthalten 14 mindestens einen Schlüssel-Fremdschlüssel-Verbund und alle 17 enthalten eine Gruppierung oder eine Aggregation.

Der Rest des Kapitels ist wie folgt aufgebaut. In Abschnitt 5.2 beschreiben wir das erste Anfragemuster, das durch den Diagonalverbund, einen speziellen Algorithmus zur Berechnung des Verbundes, ausgenutzt wird. Der Abschnitt faßt den Stand der Forschung auf dem Gebiet der Algorithmen zur Berechnung des Verbundes zusammen, führt das Muster anhand eines Beispiels ein, beschreibt die Idee und die Implementierung des Operators und enthält ein Kostenmodell für den Operator und einen experimentellen Vergleich mit anderen Algorithmen. Abschnitt 5.3 enthält die Grundlagen für die Abschnitte 5.4–5.6, die die Muster enthalten, die Gruppierung und Aggregation betreffen. In Abschnitt 5.3 findet man daher einen Überblick über den Stand

der Forschung auf dem Gebiet der Gruppierung und Aggregation und die Definition und die Implementierung des konventionellen *Generalized Aggregation* Operators (*GAgg*) [Day87], der die Grundlage für die Implementierung von Gruppierung und Aggregation in relationalen Datenbanksystemen bildet. In den Abschnitten 5.4-5.6 betrachten wir die einzelnen semantischen Anfragemuster. In jedem Abschnitt wird (1) das Muster anhand eines Beispiels eingeführt, (2) ein entsprechender Operator definiert, (3) die Implementierung des Operators skizziert, (4) ein traditioneller Auswertungsplan mit einem Auswertungsplan, der den neuen Operator verwendet, verglichen und (5) der Leistungszuwachs anhand eines Beispiels überprüft. In Abschnitt 5.7 stellen wir kurz dar, wie die neuen Operatoren in einen Anfrageoptimierer integriert werden können. Der Abschnitt 5.8 faßt die Ergebnisse dieses Kapitels zusammen.

5.2 Diagonalverbund

5.2.1 Einleitung

Bei der Auswertung von Anfragen in Datenlagern müssen oft Millionen oder Milliarden Tupel verbunden werden. Verbunde der Faktentabelle mit den Dimensionstabellen oft sind sehr teure Operationen. Daher werden für diese Anwendung schnelle Verbundalgorithmen benötigt.

Die übliche Technik zur Reduktion von Verbundkosten besteht darin, Tupel, zu denen kein Verbundpartner existiert, vor der eigentlichen Operation herauszufiltern. Die Verwendung von Bitvektor-Indizes ist das zu diesem Zweck am häufigsten eingesetzte Verfahren, wie zum Beispiel bei O'Neils und Graefes *Multi-Table-Join* [OG95]. Es ist allerdings nicht immer möglich, eine signifikante Menge von Tupeln herauszufiltern. Außerdem kommen Verbundattribute vor, die eine große Anzahl unterschiedlicher Werte annehmen, so daß die Bitvektoren sehr groß werden und sich der mit dem Filtern verbundene Aufwand nicht mehr lohnt. Es stellt sich daher die Frage, ob bestimmte Eigenschaften von Relationen existieren, die bei einer Verbundoperation ausgenutzt werden können. Bei unserer Analyse haben wir die folgenden beiden Beobachtungen gemacht (vgl. Abschnitt 2.6 und [Inm96, Kim96]). Zum einen werden die Tupel, die in ein Datenlager eingefügt werden üblicherweise an das Ende einer existierenden Relation angehängt. Daher ist die Ordnung nach dem Zeitpunkt des Einfügens – wenn auch implizit – das zentrale Ballungskriterium innerhalb einer Relation. Zum anderen haben wir festgestellt, daß in Datenlagern üblicherweise Schlüssel-Fremdschlüssel Verbunde verwendet werden. Auf der Grundlage dieser Beobachtungen haben wir einen neuen Verbundalgorithmus, den *Diagonalverbund*, entwickelt, der die Ballung nach dem Zeitpunkt des Einfügens für 1:n-Beziehungen ausnutzt.

5.2.2 Stand der Forschung

Seit der Erfindung relationaler Datenbanksysteme wurden mit großem Aufwand effiziente Verbundalgorithmen entwickelt. Nach einem Anfang mit dem einfachen Geschachtelte-Schleifen-Verbundalgorithmus war die Einführung des Misch-Verbunds die erste Verbesserung [BE76]. Später kamen der Streu-Verbund [Bra84, DKO+84] und dessen Verbesserungen [KR96, KNT89, NKT88, SM94] als Alternativen hinzu. (Einen Überblick erhält man zum Beispiel bei Mishra und Eich [ME92] oder bei Shapiro [Sha86]. Vergleiche von Sortier-Misch-Verbundalgorithmen mit Streu-Verbundalgorithmen findet man bei Graefe [Gra94] und bei Graefe, Linville und Shapiro [GLS94].) Mit großem Aufwand wurde auch die Parallelisierung von Verbundalgorithmen betrieben. Dabei wurden sowohl auf Sortierung basierende Verfahren [DNS91, LY89, Men86, STG+90] als auch auf Streuen basierende Verfahren [DG85, FKT86, SD90] entwickelt.

Streubasierte Verbundalgorithmen haben ihre Überlegenheit in vielen Anwendungsbereichen bewiesen. Einer dieser Algorithmen ist der *GRACE-Streu-Verbund* [FKT86, Sha86]. Da er im Folgenden eine zentrale Rolle spielt, beschreiben wir ihn hier kurz:

Um zwei Partitionen R und S zu verbinden, werden sie so partitioniert, daß die folgenden beiden Bedingungen erfüllt sind.

- Jede Partition der kleineren Relation paßt in den zur Verfügung stehenden Primärspeicher.
- Zueinander passende Tupel der beiden Relationen sind in jeweils zwei korrespondierenden Partitionen enthalten.

Der Algorithmus besteht aus den folgenden Schritten (wenn R die kleinere der beiden Relationen ist).

1. Wähle eine Streufunktion h, die R in r ungefähr gleich große Untermengen partitioniert. Lege r temporäre Dateien $R_1 \ldots R_r$ für die Partitionen von R, r temporäre Dateien $S_1 \ldots S_r$ für die Partitionen von S und r Ausgabepuffer an.
2. Lies R und verteile die Tupel mit Hilfe von h auf die Ausgabepuffer. Wenn der Puffer i voll ist, wird sein Inhalt an die Datei R_i angehängt. Wenn alle Tupel aus R gelesen wurden, werden die Inhalte aller Ausgabepuffer an die entsprechenden Dateien angehängt.
3. Wiederhole den letzten Schritt für S.
4. Lies jeweils eine der r Partitionen von R aus der Datei R_i in eine Streutabelle im Primärspeicher. Überprüfe für jedes Tupel aus S_i, ob ein passendes Tupel in der Tabelle vorhanden ist, und konstruiere gegebenenfalls ein Ergebnistupel.

Die Schritte 2 und 3 bezeichnet man als die *Partitionierungsphase* (engl. *Split-Phase*) und den Schritt 4 als die *Suchphase* (engl. *Probe-Phase*).

Symbol	Beschreibung
R_1	die kleinere Relation
R_N	die größere Relation
κ	Attributmenge, die für R_1 Schlüssel und für R_N Fremdschlüssel ist
$\|R_x\|$	Kardinalität (in Tupeln) von R_x (für $x \in \{1, N\}$)
$\|\|R_x\|\|$	Anzahl der für R_x benötigten Seiten
$R_x[j]$	Tupel an der Position j der Relation R_x, $1 \leq j \leq \|R_x\|$

Tabelle 5.1: Verwendete Symbole

Ein weiteres relevantes Forschungsthema ist die Entwicklung von Indexstrukturen zur Verbundunterstützung [Här78, KM94, KKD89, OG95, Val87, XH94]. Wenn es allerdings vor dem Verbund keine oder nur eine Selektion mit geringer Selektivität gibt (d.h. das Ergebnis enthält viele Tupel), ist der Leistungsgewinn durch diese Verfahren eher gering. Dies gilt auch für Bitvektor-Verbund-Indexstrukturen [OG95], die spezifisch für die Anwendung in Datenlagern entwickelt wurden. Daher haben wir nur Standard-Verbundalgorithmen in unserem Leistungsvergleich verwendet.

5.2.3 Bezeichnungen

Im folgenden verwenden wir die in Tabelle 5.1 dargestellten Symbole. Wenn R_1 und R_N die beiden zu verbindenden Relationen sind, nehmen wir an, daß für R_1 die Attributmenge κ ein Schlüssel und für R_N ein Fremdschlüssel ist. Zwischen R_1 und R_N existiert also eine 1:n-Beziehung. $|R_x|$ bezeichnet die Kardinalität (in Tupeln) von R_x (für $x \in \{1, N\}$) und $||R_x||$ die Anzahl der für R_x benötigten Seiten. Außerdem nehmen wir an, daß die Tupel in R_x eine Reihenfolge besitzen und wir bezeichnen das j-te Tupel in R_x mit $R_x[j]$.

5.2.4 Anfragemuster

Wie schon in der Einleitung erwähnt, benötigt der Diagonalverbund zwei Voraussetzungen, die häufig in Datenlagern auftreten:

1. Eine 1:n-Beziehung zwischen den Verbundpartnern.
2. Eine (implizite) Ballung der Verbundpartner nach dem Zeitpunkt des Einfügens.

Wir illustrieren diese beiden Punkte mit einem Beispiel aus dem Buch von Kimball [Kim96]. Alle Firmen, die Produkte verkaufen, müssen diese Produkte an ihre Kunden senden. Daher spielt der Prozeß des Versendens von Produkten eine wichtige Rolle. Nehmen wir an, daß es

Sendungen

Produktnr	*Preis*	*Datum*	*Transport*	*Auftragsnr*
123	24,00	12.10.96	Post	K-323
234	35,00	13.10.96	Luft	K-323
012	97,00	13.10.96	Luft	K-323
635	1298,00	23.11.96	LKW	K-326
534	453,00	23.11.96	LKW	K-326
239	20,00	10.12.96	Luft	K-351
978	10000,00	18.12.96	Bahn	K-351
174	35000,00	20.12.96	Schiff	K-351
…	…	…	…	…

Kundenaufträge

Auftragsnr	*Kundennr*	*Gesamtpreis*	*Datum*
K-323	1943	156,00	10.10.96
K-326	432	1751,00	20.11.96
K-351	129	45020,00	02.12.96
…	…	…	…

Abbildung 5.1: Die Relationen *Sendungen* und *Kundenaufträge*

im Datenlager einer Firma die Faktentabelle *Sendungen* gibt, die Informationen über die durchgeführten Sendungen enthält. Außerdem gibt es eine Dimensionstabelle *Kundenaufträge*, die Informationen über die erhaltenen Aufträge enthält. In Abbildung 5.1 sind Ausschnitte dieser Tabellen dargestellt. Wenn wir nun einen Auftrag eines Kunden in die Tabelle *Kundenaufträge* eintragen, erwarten wir, daß die entsprechenden Tupel kurz danach in *Sendungen* eingefügt werden. Diese beiden Tabellen sind dann nach den miteinander korrespondierenden Einfügezeitpunkten geballt.

Etwas allgemeiner formuliert tritt diese Ballung immer dann auf, wenn

- Objekte (im Beispiel Kundenaufträge) mit mengenwertigen Attributen (im Beispiel Sendungen) zur Speicherung auf mehrere Relationen verteilt werden und
- keine Änderungen am Datenbestand stattfinden, die die beim Einfügen entstandene Ballung wieder zerstören.

Wir betrachten hier also Anfragen in denen

> „aufgespaltene Objekte zusammengeführt werden“

und nennen das Muster *Gesamtheit-Teil-Muster*.

5.2.5 Beschreibung des Operators

Der Diagonalverbund [HWM98] nutzt die beschriebene Ballung aus. Im wesentlichen ist der Diagonalverbund ein Sortier-Misch-Verbund ohne die Sortierphase. Der entscheidende Unterschied ist, daß die Mischphase nicht davon ausgeht, daß die Relationen bezüglich der Verbundattribute sortiert vorliegen. Statt dessen verwendet der Diagonalverbund die physische Ordnung der Tupel, die nach dem Zeitpunkt des Einfügens geballt sind. Die beiden Relationen werden durch einfaches simultanes Lesen verbunden. Der Leseoperator auf der äußeren Relation bewegt ein Gleitfenster fester Größe über die Relation. Die Suche nach Verbundpartnern

für die innere Relation findet nur innerhalb dieses Fensters statt. Ein spezieller Mechanismus befaßt sich mit den Tupeln der inneren Relation, für die innerhalb des Fensters kein Verbundpartner gefunden werden konnte. Wir nennen diese Tupel *Fehlversuche*. Obwohl diese Idee einfach ist, erweist sie sich als sehr effektiv, sofern man einige Details beachtet. Diese liegen in der Pufferverwaltung, der Größe des Fensters, der Organisation des Fensters und der „Gleitgeschwindigkeit“ des Fensters. Wir werden die in diesen Punkten auftretenden Probleme und mögliche Lösungen beschreiben.

Der Diagonalverbund hat gegenüber anderen Verbundverfahren, wie zum Beispiel dem Sortier-Misch-Verbund [BE76], dem GRACE-Streu-Verbund [FKT86, Sha86] oder auch dem hybriden Streu-Verbund [DKO+84, Sha86], einen wesentlichen Vorteil: Auch wenn die Relationen nicht in den Primärspeicher passen, ist der Diagonalverbund in vielen Fällen in der Lage das Erzeugen von großen temporären Relationen zu vermeiden. Bei Implementierung des Verfahrens als Iterator ergibt sich ein weiterer Vorteil. In der ersten Phase wird der Tupelstrom zwischen dem Produzenten und dem Konsumenten des Iterators nicht unterbrochen, so daß zum Beispiel Verweise in den Systempuffer gültig bleiben.

5.2.6 Implementierung

Wir stellen zwei Implementierungen des Operators vor. Die erste Implementierung ist allerdings sehr einfach gehalten und dient nur zur Darstellung des Prinzips. So werden hier zum Beispiel nur einzelne Tupel statt Blöcken von Tupeln betrachtet und wesentliche Probleme einer tatsächlichen Implementierung ausgespart.

Wir beginnen aber mit einer auf beide Implementierungen zutreffenden Vorüberlegung.

5.2.6.1 Vorüberlegung

Nehmen wir an, daß

- ein Tupel in R_1 und alle dazugehörenden Tupel in R_N von *einer* Transaktion erzeugt und „gleichzeitig“ in den beiden Relationen abgelegt werden und daß
- die Anzahl der bei solchen Transaktionen in R_N eingefügten Tupel konstant ist.

Dann können wir zu jedem Tupel aus R_N einfach die physische Position des entsprechenden Tupels in R_1 bestimmen. Wir nennen diese Situation eine *vollkommene Ballung* nach dem Zeitpunkt des Einfügens. Im Fall einer 1:n-Beziehung, falls also zu jedem Tupel aus R_N genau ein Tupel in R_1 existiert, erwarten wir, daß wir das zu $R_N[j]$ gehörende Tupel an der Position $\left\lceil \frac{j}{|R_N|/|R_1|} \right\rceil$ finden. Wenn die Anzahl der Verbundpartner eines Tupels aus R_1 variiert, ist die berechnete Position nur eine Approximation. In Abbildung 5.2 ist die vollkommene Situation

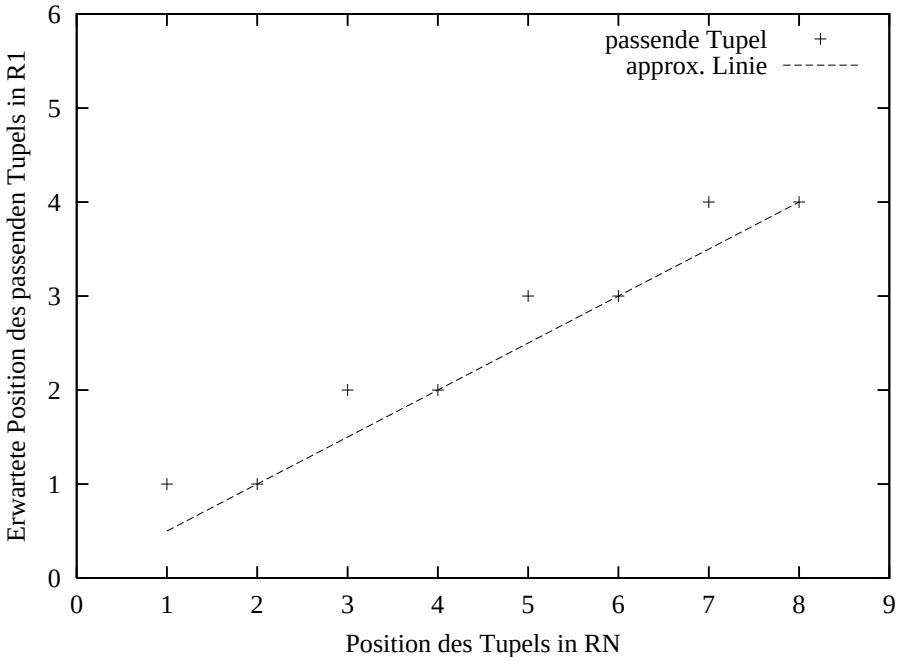


Abbildung 5.2: Vollkommene Ballung

dargestellt. Auf der X-Achse sind die Positionen der Tupel in R_N und auf der Y-Achse die Positionen der Tupel in R_1 aufgetragen. In diesem Beispiel existieren zu jedem Tupel in R_1 genau zwei Tupel in R_N. Den Verbundpartner zu $R_N[5]$ finden wir also an der Position $\left\lceil \frac{5}{8/4} \right\rceil = 3$. Man beachte, daß selbst bei vollkommener Ballung die Relationen in der Regel nicht bezüglich der Verbundattribute sortiert sind.

5.2.6.2 Einfacher Diagonalverbund

Wenn die Tupel in den beiden Relationen vollkommen geballt sind, reicht eine einfache Mischphase aus, um die beiden Relationen zu verbinden. Das ist aber leider nicht immer der Fall. Abweichungen entstehen, wenn

- die Anzahl der zu einem Tupel aus R_1 gehörenden Tupel in R_N variiert,
- die Tupel nicht gleichzeitig in R_1 und R_N eingefügt werden oder
- die Tupel nach dem Einfügen umorganisiert wurden (durch das Löschen von Tupeln, durch das Einfügen neuer Tupel oder durch das Ersetzen vorhandener Tupel).

Daher betrachten wir nicht nur ein Tupel aus R_1, sondern wir halten die g_t Tupel, die sich in der Nähe der berechneten Position befinden, in einem Puffer. Wir nennen den Teil von R_1, der sich

```
Diagonalverbund(R_1, R_N, g_t) {
  /* Phase 1 */

  verhaeltnis  = |R_N| / |R_1|
  aktTup = g_t/2
  fuelle den Puffer mit R_1[1] bis R_1[aktTup]
  for(j = 1; j <= |R_N|; j++) {
    if(im Puffer gibt es ein zu R_N[j] passendes Tupel t) {
      verbinde t mit R_N[j]
      gib das Ergebnis aus
    } else {
      schreibe R_N[j] in tmp
    }
    if(j % verhaeltnis == 0) {
      aktTup++
      if(Puffer ist nicht voll) {
        fuege R_1[aktTup] in den Puffer ein
      } else {
        ersetze das Tupel mit der niedrigsten Position
          mit R_1[aktTup]
      }
    }
  }

  /* Phase 2 */

  verbinde R_1 mit tmp mit einem Standard-Verbundalgorithmus
}
```

Abbildung 5.3: Einfacher Diagonalverbund

Symbol	Beschreibung
t	ein beliebiges Tupel
g_t	die Größe des Fensters/Puffers (in Anzahl Tupel)
g_s	die Größe des Fensters/Puffers (in Anzahl Seiten)
l	Anzahl der Streutabellen im Feld
s	Größe einer Streutabelle in Seiten ($= \frac{g_s}{l}$)
zwOp	reihenfolgeerhaltender Operator zwischen dem Leseoperator auf R_N und dem Diagonalverbund

Tabelle 5.2: Zusätzliche Symbole zur Beschreibung der Implementierung

im Puffer befindet, ein *Fenster* auf R_1.

Zur Beschreibung der Implementierung benötigen wir noch einige Symbole, die in Tabelle 5.2 zusammengefaßt sind. Der *einfache Diagonalverbund* funktioniert wie folgt. Wir initialisieren

das Fenster mit $\left\lceil \frac{g_t}{2} \right\rceil$ Tupeln von $R_1[1]$ bis $R_1[\left\lceil \frac{g_t}{2} \right\rceil]$. Wir erwarten das zu $R_N[1]$ passende Tupel an der Position $R_1[1]$ oder zumindest im Intervall zwischen $R_1[-\left\lceil \frac{g_t}{2} \right\rceil]$ und $R_1[\left\lceil \frac{g_t}{2} \right\rceil]$ zu finden. Da R_1 keine negativen Positionen enthält, fällt dieser Teil weg. Dann wird R_N mit $R_N[1]$ beginnend sequentiell gelesen. Dabei werden mit Ausnahme des aktuellen Tupels keine Tupel von R_N gepuffert. Für jedes Tupel $R_N[j]$ suchen wir das passende Tupel aus R_1 im Fenster. Falls die Suche erfolgreich ist (wir nennen das einen *Treffer*), erzeugen wir sofort ein Ergebnistupel und betrachten das nächste Tupel von R_N. Dies ist möglich, da es wegen der 1:n-Beziehung zwischen R_1 und R_N für jedes Tupel aus R_N höchstens einen Treffer gibt. Falls die Suche nicht erfolgreich ist (ein *Fehlversuch*), schreiben wir $R_N[j]$ in eine temporäre Relation. Nachdem $\frac{|R_N|}{|R_1|}$ Tupel aus R_N verarbeitet wurden, fügen wir ein weiteres Tupel aus R_1 in das Fenster ein. Falls im Fenster kein Platz sein sollte, wird das Tupel mit der niedrigsten Position aus dem Fenster entfernt. Nachdem alle Tupel aus R_N betrachtet wurden, verbinden wir die Tupel aus der temporären Relation (die üblicherweise sehr viel kleiner als $||R_N||$ ist) mit R_1. Dazu verwenden wir einen Standard-Verbundalgorithmus. (Wir werden später den GRACE-Streu-Verbund verwenden.) In Abbildung 5.3 ist der Algorithmus zusammenfassend dargestellt.

Bevor wir eine verbesserte Version des Diagonalverbunds beschreiben, weisen wir noch auf einige Probleme der einfachen Version hin.

1. Der Algorithmus läßt sich so nicht sehr effizient implementieren, da die verwendeten Puffer Tupel als Verarbeitungseinheiten verwenden. Datenbankverwaltungssysteme verwenden üblicherweise Seiten.

2. Die Organisation der Tupel im Fenster ist entscheidend für die Leistungsfähigkeit des Algorithmus und muß daher noch näher betrachtet werden.

3. Der Algorithmus verarbeitet ausschließlich Basisrelationen, d.h. es ist zum Beispiel nicht möglich vor dem Verbund eine Selektion durchzuführen.

Wir werden diese Probleme im folgenden lösen.

5.2.6.3 Verbesserter Diagonalverbund

Wir haben die bisherige Darstellung bewußt einfach gehalten, um das Grundprinzip des Algorithmus darzustellen. Nun wenden wir uns den Details zu.

Organisation des Fensters Wir verwenden statt eines Tupel-orientierten Puffers einen Seiten-orientierten Puffer. Wir lesen also nicht einzelne Tupel in das Fenster, sondern alle Tupel der folgenden s Seiten, da das Lesen von (mehreren) Seiten sehr viel effizienter ist als das Lesen einzelner Tupel. Also müssen wir, wenn das Fenster voll ist, s Seiten ersetzen. Wir nennen s

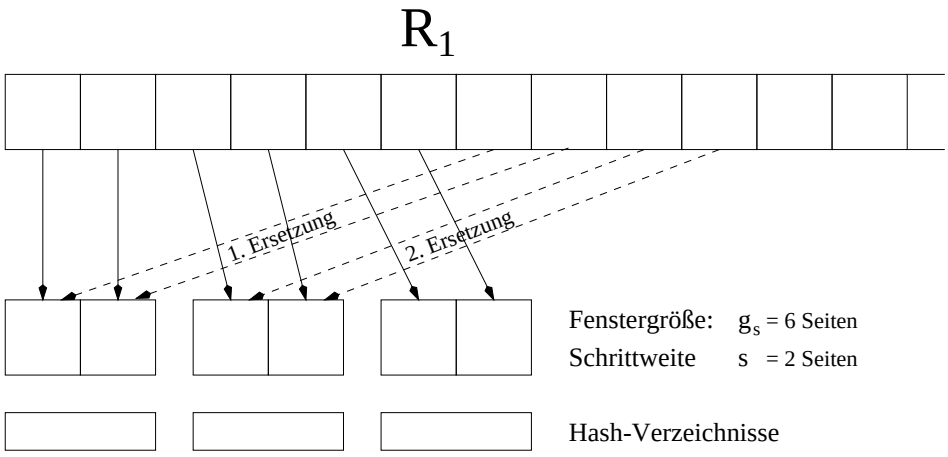


Abbildung 5.4: Organisation des Fensters

die *Schrittweite*. Offensichtlich müssen wir die Tupel immer ersetzen nachdem $s \cdot \frac{||R_N||}{||R_1||}$ Seiten von R_N gelesen wurden.

Da eine sequentielle Suche im Fenster zu teuer ist, verwenden wir Streutabellen, um einen Verbundpartner im Fenster zu finden. Dabei gibt es zwei Alternativen. Wir können entweder eine große Streutabelle, die g_s Seiten groß ist, oder ein Feld mit l Streutabellen, die jeweils $\frac{g_s}{l}$ Seiten groß sind [1], verwenden. Die Verwendung einer Streutabelle ist problematisch. Wenn wir die Schrittweite s genauso groß wie die Fenstergröße g_s wählen, ersetzen wir jeweils auch einen Teil der Tupel, die im letzten Schritt eingefügt wurden und die in diesem Schritt benötigt werden. Wenn wir eine Schrittweite s wählen, die kleiner als die Fenstergröße g_s ist, müssen wir in jedem Schritt viele Tupel einzeln aus der Streutabelle löschen. Wir verwenden daher ein Feld von l Streutabellen. Außerdem wählen wir eine Schrittweite, die der Größe der Streutabellen entspricht, d.h. $s = \frac{g_s}{l}$. Dann können wir in jedem Schritt eine ganze Streutabelle löschen. Das erfordert wesentlich weniger Aufwand als das Löschen einzelner Tupel aus einer Streutabelle.

Das Bewegen des Fensters funktioniert dann wie folgt. Nachdem $s \cdot \frac{||R_N||}{||R_1||}$ Seiten von R_n gelesen wurden, wird das Fenster eine Position nach vorne gerückt. Dazu wird die älteste sich noch im Puffer befindende Streutabelle geleert und mit den nächsten s Seiten von R_1 wieder gefüllt.

In Abbildung 5.4 ist die Organisation des Fensters dargestellt. Die Fenster besteht aus sechs Seiten, die sich aus drei jeweils zwei Seiten großen Blöcken zusammensetzen. Die Schrittweite beträgt also zwei Seiten. Die gestrichelten Linien deuten an, wie die Seiten ersetzt werden, wenn das Fenster voll ist.

Suche im Fenster Bei der Suche nach einem passenden Tupel in dem Feld von l Streutabellen betrachten wir zuerst die mittlere Tabelle an der Position $\left\lceil \frac{l}{2} \right\rceil$ des Feldes. Wenn R_1 und R_N

[1] Wir gehen dabei davon aus, daß g_s und l so gewählt werden, daß $\frac{g_s}{l}$ ganzzahlig ist

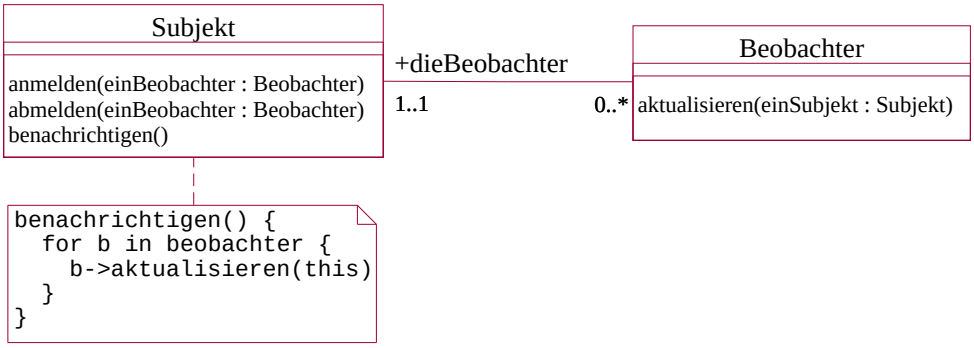


Abbildung 5.5: Beobachter-Muster

vollkommen geballt sind, erwarten wir das passende Tupel in dieser Tabelle. Falls wir es da nicht finden, suchen wir in der Tabelle an der Position $\left\lceil \frac{l}{2} \right\rceil + 1$. Bei einem weiteren Mißerfolg betrachten wir die Positionen $\left\lceil \frac{l}{2} \right\rceil - 1$, $\left\lceil \frac{l}{2} \right\rceil + 2$, $\left\lceil \frac{l}{2} \right\rceil - 2$, und so weiter. Wir nennen diese Vorgehensweise *Zick-Zack-Suche*. Dieses Verfahren eignet sich besonders, wenn die Abweichung der Ballung der Relationen von der vollkommenen Ballung durch die Standardabweichung beschrieben werden kann (vgl. auch Abbildung 5.7). Es empfiehlt sich l ungerade zu wählen und so zu einer symmetrischen Suche zu kommen.

Wie im einfachen Fall können wir auch hier sofort ein Ergbnistupel produzieren, wenn wir in einer der Streutabellen ein passendes Tupel finden. Die Fehlversuche werden wieder in einer temporären Relation gesammelt. Allerdings werden sie nicht direkt auf dem Sekundärspeicher abgelegt, sondern zunächst in einem Puffer gesammelt, um sie seitenweise ausschreiben zu können. Wenn die Anzahl der Fehlversuche klein genug ist, müssen diese also unter Umständen gar nicht ausgeschrieben werden, sondern können direkt im Primärspeicher weiterverarbeitet werden.

Verbinden von Nicht-Basisrelationen Wenn vor der Verbundoperation Tupel – zum Beispiel durch eine Selektion – aus einem der beiden oder aus beiden Verbundpartnern herausgefiltert wurden, kann dies die Synchronisation zerstören. Das bedeutet, daß das Fenster auf R_1 jeweils zu früh oder zu spät bewegt wird und so die falschen Tupel enthält. Daher muß der Diagonalverbund mit den Leseoperatoren auf den Basisrelationen synchronisiert werden. Wir verwenden dazu das *Beobachter-Muster* (engl. *Observer Pattern*, vgl. [GHJV95]).

Allgemein wird das Beobachter-Muster verwendet, wenn mehrere von einem Objekt s (das Subjekt) abhängige Objekte $b_1, \ldots b_n$ (die Beobachter) über eine Zustandsänderung von s informiert werden sollen. Abbildung 5.5 enthält eine Beschreibung des Musters in UML-Syntax. Die Methoden *anmelden* und *abmelden* bauen die Verbindung eines Beobachtes zu einem Subjekt s auf und ab. Wenn s seinen Zustand ändert, benutzt es die Methode *benachrichtigen*, um alle abhängigen Beobachter über die Zustandsänderung zu informieren. Die Methode *benach-*

Symbol	*Definition*
$K_{\text{E/A}}$	Kosten für den Transfer von Seiten zwischen Primär- und Sekundärspeicher
P_x	beliebige Puffergröße
Z_f	Summe der durchschnittlichen Such- und Latenzzeit
Z_t	Zeit zum Transfer einer Seite
Z_s	Zeit zum Streuen eines Tupels
Z_v	Zeit um den Verbundpartner eines Tupels zu finden und die Tupel zu verbinden

Tabelle 5.3: Zusätzliche Symbole für das Kostenmodell

richtigen ruft dazu die Methode *aktualisieren* jedes angemeldeten Beobachters auf. Dabei wird ein Verweis auf s als Parameter übergeben, um dem Beobachter mitzuteilen, welches Subjekt seinen Zustand geändert hat.

In unserem Fall benachrichtigt der Operator, der die Tupel von R_N liest, den Diagonalverbund über die Position der aktuellen Tupel in R_N. Der Diagonalverbund ist dann in der Lage das Fenster in der richtigen Geschwindigkeit weiterzubewegen oder gar einige Seiten auszulassen. Mit dieser Technik ist es möglich beliebige Operatoren zwischen dem Leseoperator auf R_N und und dem Diagonalverbund zu verwenden, solange die relative Reihenfolge der Tupel erhalten bleibt. Eine ähnliche Technik kann verwendet werden, um Operatoren zwischen dem Leseoperator auf R_1 und dem Diagonalverbund zu verwenden. Wenn wir während des Fortbewegens des Fensters eine Streutabelle lesen, so wird diese stets vollständig gelesen. Wenn ein eingefügter Operator sehr viele Tupel verwirft, kann es sein, daß der Leseoperator auf R_1 vorauseilt, um die Streutabelle zu füllen. Wenn der Leseoperator auf R_1 den Diagonalverbund über die Position der aktuellen Tupel informiert, kann der Diagonalverbund diesen Fall erkennen und die Forbewegung des Fensters verzögern, bis der Leseoperator auf R_N aufgeholt hat.

Der verbesserte Algorithmus ist in Abbildung 5.6 zusammenfassend dargestellt. Man beachte dabei, daß sich die jeweilige „mittlere“ Streutabelle nicht immer an der Position $\left\lceil \frac{l}{2} \right\rceil$ des Feldes befindet, da wir die Streutabellen im Feld wiederverwenden.

5.2.7 Kosten des Operators

5.2.7.1 Kostenmodell

Die Grundlage unseres Kostenmodells für den Diagonalverbund sind die von Harris und Ramamohanarao vorgestellten Modelle [HR96]. Die Beschreibung der für die Kostenmodelle benötigten zusätzlichen Symbole befindet sich in Tabelle 5.3.

Die Kosten $K_{\text{E/A}}$ für den Transfer von $||R_x||$ Seiten zwischen Primär- und Sekundärspeicher durch einen Puffer der Größe P_x berechnet man wie folgt.

```
Diagonalverbund(R_1, zwOp(R_N), g_s, l) {
  /* Phase 1 */

  verhaeltnis = |R_N| / |R_1|
  lege feld fld mit l Streutabellen an
  fuelle fld[1] bis fld[l/2] mit Tupeln aus R_1
  do {
    t_N = naechstes Tupel aus zwOp(R_N)
    Zick-Zack-Suche in den Streutabellen nach einem
      passenden Tupel
    if(passendes Tupel gefunden) {
      verbinde t_N mit dem gefundenen Tupel
      gib das Ergebnis aus
    } else {
      schreibe t_N in tmp
    }
    if(Nachricht vom Leseoperator auf der Relation R_N erhalten) {
      if(es gibt noch eine freie Streutabelle) {
        lade die naechsten s Seiten von R_1 in eine
          freie Streutabelle
      } else {
        loesche die aelteste vorhandene Streutabelle
        lade die naechsten s Seiten von R_1 in die
          geloeschte Streutabelle
      }
    }
  } while(zwOp(R_N) enthaelt noch Tupel)

  /* Phase 2 */

  verbinde R_1 mit tmp mit einem Standard-Verbundalgorithmus
}
```

Abbildung 5.6: Verbesserter Diagonalverbund

$$K_{\text{E/A}}(||R_x||, P_x) = \left\lceil \frac{||R_x||}{P_x} \right\rceil \cdot Z_f + ||R_x|| \cdot Z_t \tag{5.1}$$

Dabei ist Z_f die Summe der durchschnittlichen Such- und Latenzzeit und Z_t ist die Zeit, die für den Transfer einer Seite zwischen Primär- und Sekundärspeicher benötigt wird.

Die Kosten des Diagonalverbunds setzen sich aus den Kosten für die erste Phase und den Kosten für die zweite Phase zusammen.

$$K_{\text{Diag}}(R_1, R_N) = K_{\text{Phase1}} + K_{\text{Phase2}} \tag{5.2}$$

In der ersten Phase müssen R_1 und R_2 gelesen werden, die Tupel von R_1 müssen in Streutabellen abgelegt werden, Verbundpartner müssen gefunden und verbunden werden, und die Fehlversuche müssen auf dem Sekundärspeicher abgelegt werden.

$$K_{\text{Phase1}} = K_{\text{Lies } R_1} + K_{\text{Lies } R_N} + K_{\text{Streue } R_1} + K_{\text{Verbinde}} + K_{\text{Fehlversuche}} \tag{5.3}$$

Die Bestandteile von K_{Phase1} werden wie folgt bestimmt:

$$\begin{aligned} K_{\text{Lies } R_1} &= K_{\text{E/A}}(||R_1||, s) && (5.4)\\ K_{\text{Lies } R_N} &= K_{\text{E/A}}(||R_N||, 1) && (5.5)\\ K_{\text{Streue } R_1} &= |R_1| \cdot Z_s && (5.6)\\ K_{\text{Verbinde}} &= |R_N| \cdot Z_v && (5.7)\\ K_{\text{Fehlversuche}} &= K_{\text{E/A}}(||tmp||, 1) && (5.8) \end{aligned}$$

Die Kosten der zweiten Phase hängen von dem dort verwendeten Verbundalgorithmus ab. Wir haben den GRACE-Streu-Verbund verwendet. Daher ist

$$K_{\text{Phase2}} = K_{\text{GRACE}}(R_1, tmp). \tag{5.9}$$

Kostenmodelle für den GRACE-Streu-Verbund findet man zum Beispiel bei Harris und Ramamohanarao [HR96] oder bei Haas et al. [HCLS97].

Wie wir gesehen haben, spielt die Größe der temporären Relation, die die Fehlversuche aufnimmt, eine wesentliche Rolle bei der Bestimmung der Kosten des Diagonalverbunds. Obwohl wir im nächsten Abschnitt eine Abschätzung dieser Größe für eine normalverteilte Abweichung von der vollkommenen Ballung vorstellen, ist die Bestimmung der Größe nicht unproblematisch. Da die Annahme der Normalverteilung wahrscheinlich nicht für alle Anwendungen stimmt, empfehlen wir die folgende Vorgehensweise. In Zeiten geringer Belastung (oder im Rahmen eines Kommandos zur Aktualisierung der Statistiken) kann eine verkürzte Version der ersten Phase des Diagonalverbunds ausgeführt werden. Diese verkürzte Version bestimmt die Größe der temporären Relation, ohne die Relation oder Ergebnistupel zu erzeugen.

Der Anfrageoptimierer eines Datenbankmanagementsystems benötigt das dargestellte Kostenmodell und die Parameter (wie zum Beispiel die Größe der temporären Relation), um eine Entscheidung über die Verwendung des Diagonalverbunds zu treffen. Zur Abschätzung der Kosten eines Verbunds von zwei Basisrelationen kann (5.2) ohne Modifikation verwendet werden. Wenn reihenfolgeerhaltende Zwischenoperatoren auftreten, müssen die Standard-Techniken eines Anfrageoptimierers zur Bestimmung der Kosten komplexer Anfragen angewandt werden (wie zum Beispiel die Bestimmung der Kardinalitäten von Zwischenergebnissen und der temporären Relation des Diagonalverbunds mit Hilfe von Selektivitäten).

Symbol	*Definition*
$N_{\mu,\sigma}(a,b)$	Normalverteilung
$n_{\mu,\sigma}(x)$	Dichtefunktion der Normalverteilung
$j(i)$	erwartete Position des passenden Tupels
$f_{\text{anf}}(i)$	Anfangsposition des Fensters
$f_{\text{end}}(i)$	Endposition des Fensters
$m_{\text{anf}}(i)$	Anfangsposition der mittleren Streutabelle ($f_{\text{anf}} \leq m_{\text{anf}}$)
$m_{\text{end}}(i)$	Endposition der mittleren Streutabelle ($m_{\text{end}} \leq f_{\text{end}}$)
h_t	durchschnittliche Anzahl Tupel pro Streutabelle

Tabelle 5.4: Symbole zur Berechnung der Wahrscheinlichkeit eines Fehlversuchs

5.2.7.2 Wahrscheinlichkeit von Fehlversuchen

In diesem Abschnitt bestimmen wir eine Formel zur Berechnung der *Wahrscheinlichkeit von Fehlversuchen*, das ist die Wahrscheinlichkeit, daß ein beliebiges Tupel aus R_N ein Fehlversuch ist. In Tabelle 5.4 sind die dabei verwendeten Symbole zusammengefaßt.

Mit Hilfe dieser Wahrscheinlichkeit kann die Größe der temporären Relation abgeschätzt werden:

$$|tmp| \quad = \quad P_{\not\phi}(R_N[i] \text{ ist ein Fehlversuch}) \cdot |R_N| \tag{5.10}$$

Wie schon erwähnt, nehmen wir an, daß die Abweichung der Relationen von einer vollkommenen Ballung durch eine Normalverteilung beschrieben werden kann. Die Dichtefunktion der Normalverteilung $n_{\mu,\sigma}(x)$ mit dem Mittelwert μ und der Standardabweichung σ ist definiert durch

$$n_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}}. \tag{5.11}$$

Dementsprechend ist die Wahrscheinlichkeit, daß die normalverteilte Zufallsgröße X in das Intervall $[a,b)$ fällt

$$P(a \leq X < b) = N_{\mu,\sigma}(a,b) = \int_a^b n_{\mu,\sigma}(t)dt. \tag{5.12}$$

Betrachten wir zunächst, was es bedeutet, wenn die Abweichung von der vollkommenen Ballung normalverteilt ist. Für das Tupel $R_N[i]$ an der Position i ($1 \leq i \leq |R_N|$) der Relation R_N

erwarten wir – bei vollkommener Ballung – das passende Tupel an der Position

$$j(i) = \left\lceil i \cdot \frac{|R_1|}{|R_N|} \right\rceil \tag{5.13}$$

der Relation R_1. Die Normalverteilung beschreibt nun die Wahrscheinlichkeit für jede Position k in R_1, daß das zu $R_N[i]$ passende Tupel sich an der Position k befindet. Der Mittelwert und damit die Position, an der sich das passende Tupel mit der größten Wahrscheinlichkeit befindet, ist $j(i)$. In Abbildung 5.7 ist die Wahrscheinlichkeitsverteilung dargestellt. Wenn wir die durchschnittliche Anzahl Tupel pro Streutabelle mit $h_t = \frac{g_t}{l}$ bezeichnen, beginnt die mittlere Streutabelle im Fenster an der Position

$$m_{\text{anf}}(i) = \left(\left\lceil \frac{j(i)}{h_t} \right\rceil - 1 \right) \cdot h_t + 1 \tag{5.14}$$

und endet an der Position

$$m_{\text{end}}(i) = \left(\left\lceil \frac{j(i)}{h_t} \right\rceil \right) \cdot h_t. \tag{5.15}$$

Die Symbole $f_{\text{anf}}(i)$ und $f_{\text{end}}(i)$ bezeichnen die kleinste und die größte Position der Tupel des Fensters. Wenn l ungerade ist bedeutet das:

$$f_{\text{anf}}(i) = m_{\text{anf}}(i) - \left\lfloor \frac{l}{2} \right\rfloor \cdot h_t \tag{5.16}$$

$$f_{\text{end}}(i) = m_{\text{end}}(i) + \left\lfloor \frac{l}{2} \right\rfloor \cdot h_t \tag{5.17}$$

Um die Darstellung übersichtlicher zu halten, betrachten wir die Spezialfälle am Anfang und am Ende von R_1 nicht.

Die Wahrscheinlichkeit dafür, daß $R_N[i]$ für ein beliebiges $1 \leq i \leq |R_N|$ ein Fehlversuch ist, ist die Wahrscheinlichkeit, daß das passende Tupel nicht im Fenster ist:

$$P(R_N[i] \text{ ist ein Fehlversuch}) = 1 - N_{j(i),\sigma}(f_{\text{anf}}(i), f_{\text{end}}(i)) \tag{5.18}$$

Während wir die einzelnen Tupel von R_N betrachten, ändert sich diese Wahrscheinlichkeit, da $j(i)$ sich von $m_{\text{anf}}(i)$ nach $m_{\text{end}}(i)$ durch die mittlere Streutabelle bewegt. Wenn $j(i)$ $m_{\text{end}}(i)$ erreicht, wird das Fenster der Schrittweite entsprechend weiterbewegt und $j(i)$ beginnt wieder bei $m_{\text{anf}}(i)$ der dann aktuellen mittleren Streutabelle. In Tabelle 5.5 ist beispielhaft ein Ausschnitt aus den sich für $\frac{|R_1|}{|R_N|} = \frac{1}{3}$ und $h_t = 4$ ergebenden Positionen dargestellt.

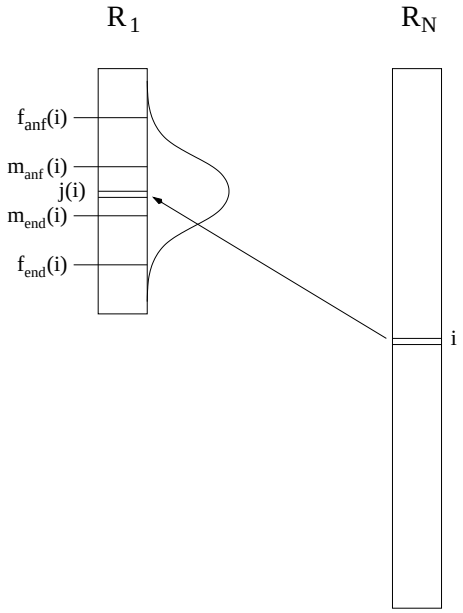


Abbildung 5.7: Normalverteilte Abweichung von der perfekten Ballung

i	120	121	122	123	124	125	126	127	128	129	130	131	132	133
$j(i)$	40	41	41	41	42	42	42	43	43	43	44	44	44	45
$m_{\text{anf}}(i)$	37	41	41	41	41	41	41	41	41	41	41	41	41	45
$m_{\text{end}}(i)$	40	44	44	44	44	44	44	44	44	44	44	44	44	48

Tabelle 5.5: Der Weg durch die mittlere Streutabelle

Die durchschnittliche Wahrscheinlichkeit dafür, daß das Tupel an der Position i einen Fehlversuch ist, während $j(i)$ sich innerhalb der mittleren Streutabelle eines gegebenen Fensters bewegt ist:

$$P_{\phi}(R_N[i] \text{ ist ein Fehlversuch}) = \sum_{j=m_{\text{anf}}(i)}^{m_{\text{end}}(i)} \frac{1 - N_{j,\sigma}(f_{\text{anf}(i)}, f_{\text{end}}(i))}{m_{\text{end}}(i) - m_{\text{anf}}(i)} \tag{5.19}$$

Da diese Wahrscheinlichkeit unabhängig von der Wahl des Fensters ist, wählen wir das Fenster von Position 1 bis Position g_t und erhalten:

$$P_{\phi}(R_N[i] \text{ ist ein Fehlversuch}) = \sum_{j=\lfloor \frac{l}{2} \rfloor \cdot h_t + 1}^{\lceil \frac{l}{2} \rceil \cdot h_t} \frac{1 - N_{j,\sigma}(1, g_t)}{h_t} \tag{5.20}$$

Nun interessiert uns weiter, wie groß man das Fenster wählen muß, um eine Wahrscheinlichkeit für einen Fehlversuch zu erhalten, die unter einer vorgegebenen akzeptablen Schranke $P_{\text{akzeptabel}}$

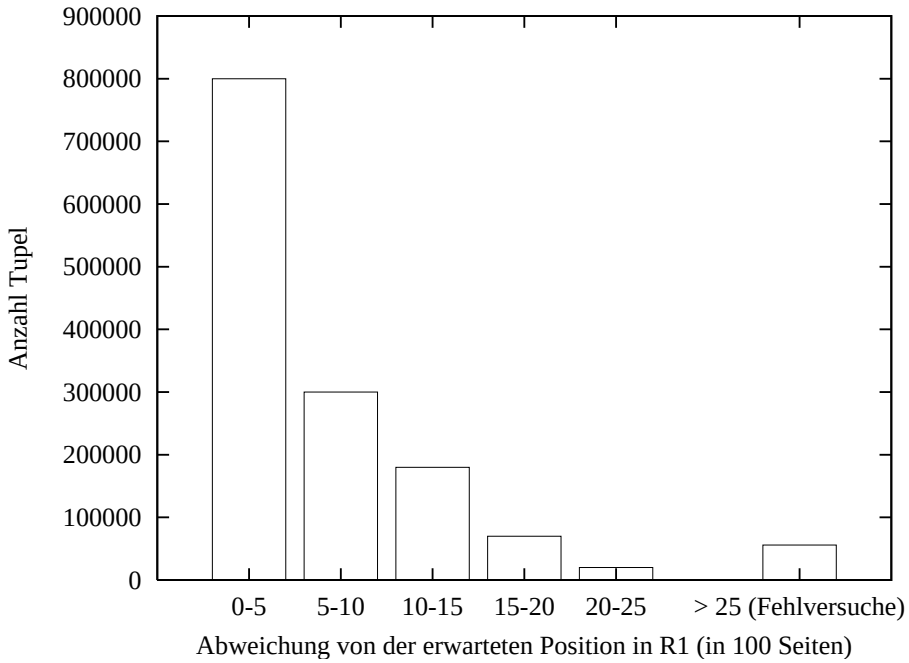


Abbildung 5.8: Histogramm zur Messung der Abweichung von der vollkommenen Ballung

liegt. Wir wollen also zu einem gegebenen $P_{\text{akzeptabel}}$ das entsprechende g_t berechnen. Dies ist mit Hilfe von (5.20) zwar möglich, aber unpraktisch, da

1. $N_{j,\sigma}(1, g_t)$ nur numerisch bestimmt werden kann, so daß wir keine geschlossenen Formel zur Berechnung von g_t herleiten können, und da
2. wir bisher keine Möglichkeit haben σ genau zu bestimmen.

Daher empfehlen wir Histogramme zu verwenden. Um Histogramme zu ermitteln genügt es mit dem größtmöglichen Puffer einmal R_1 und R_N zu lesen. Für jedes Tupel aus R_N wird dann der Betrag der Differenz zwischen der erwarteten und der tatsächlichen Position des passenden Tupels in R_1 in den entsprechenden Bereich des Histogramms eingefügt. Fehlversuche werden separat gezählt. Das resultierende Histogramm (wie zum Beispiel das in Abbildung 5.8) kann dann zur Bestimmung einer Fenstergröße für eine gegebene Wahrscheinlichkeit $P_{\text{akzeptabel}}$ verwendet werden.

5.2.8 Leistungsvergleich

Dieser Abschnitt besteht aus zwei Teilen. Im ersten Teil beschreiben wir Inhalt und Umgebung des Vergleichs. Im zweiten Teil stellen wir die Ergebnisse dar und analysieren sie.

Order
O_Orderkey
O_Custkey
O_Orderstatus
O_Totalprice
O_Orderdate
O_Orderpriority
O_Clerk
O_Shippriority
O_Comment

Lineitem
L_Orderkey
L_Partkey
L_Suppkey
L_Linenumber
L_Quantity
L_Extendedprice
L_Discount
L_Tax
L_Returnflag
L_Linestatus
L_Shipdate
L_Commitdate
L_Receiptdate
L_Shipinstruct
L_Shipmode
L_Comment

Abbildung 5.9: Schemata der Relationen Order und Lineitem

5.2.8.1 Beschreibung

Der Vergleich wurde auf einer SUN UltraSparc 1 (143 MHz) mit 288 MByte Primärspeicher unter Solaris 2.5.1 durchgeführt. Der Algorithmus wurde als physischer Operator in AODB implementiert. Dabei wurde von der temporären Relation, die die Fehlversuche aufnimmt, jeweils eine Seite im Primärspeicher gepuffert. Für die zweite Phase des Diagonalverbunds haben wir den GRACE-Streu-Verbund verwendet [FKT86, Sha86].

Die verwendeten Daten waren die des TPC-D-Benchmarks [TPC95] für den Skalierungsfaktor 1 (entspricht einer Datenbankgröße von 1 GByte). Wir haben die Relationen *Order* und *Lineitem* verbunden. Die Schemata der Relationen sind in Abbildung 5.9 dargestellt. Die Relation *Order* war bezüglich des Attributs *O_Orderdate* und die Relation *Lineitem* bezüglich des Attributs *L_Shipdate* sortiert. Man beachte, daß dies *nicht* zu einer Sortierung auf einem der Verbundattribute *O_Orderkey* und *L_Orderkey* führt, aber eine Imitation der *Ballung nach dem Zeitpunkt des Einfügens* ist.

In einem ersten Schritt haben wir einige Parameter des Diagonalverbunds, wie zum Beispiel die Anzahl der zu verwendenden Streutabellen, optimiert. Dann haben wir die Gesamtkosten, Prozessorkosten und E/A-Kosten des Diagonalverbunds mit denen eines blockorientierten Geschachtelte-Schleifen-Verbunds und denen eines GRACE-Streu-Verbunds verglichen. Dabei haben wir unterschiedliche Puffergrößen betrachtet. Wir haben den hybriden Streu-Verbund nicht betrachtet, da er bei dem von uns betrachteten Verhältnis der Primärspeichergröße zur Men-

Parameter	*Wert*
Seitengröße	4 KByte
Größe von *Order*	44,475 Seiten
Kardinalität von *Order*	1,500,000 Tupel
Größe von *Lineitem*	189,635 Seiten
Kardinalität von *Lineitem*	6,001,215 Tupel
Fenstergröße für den Diagonalverbund Schrittweite (Fenstergröße/5)	300 - 4000 Seiten (1.17 MByte - 15.62 MByte) 60 - 800 Seiten
Puffergröße für Geschachtelte-Schleifen-Verbund	300 - 4000 Seiten (1.17 MByte - 15.62 MByte)
Puffergröße für GRACE-Streu-Verbund	300 - 4000 Seiten (1.17 MByte - 15.62 MByte)

Tabelle 5.6: Parameter des Leistungsvergleichs

ge der verarbeitenden Daten keinen Vorteil gegenüber dem GRACE-Streu-Verbund hat [HR96, Sha86, HCLS97]. Wie Tabelle 5.6, die die Parameter des Leistungsvergleichs zusammenfaßt, zeigt, beträgt die Größe des verwendeten Puffers höchstens $\frac{1}{50}$ der Größe der Relationen. Dies ist für große Datenlager eine realistische Annahme.

5.2.8.2 Ergebnisse

Bestimmung der Parameter Um zwei Relationen mit dem Diagonalverbund zu verbinden, müssen wir zunächst eine Fenstergröße und eine Schrittweite wählen. Bei der Wahl der Fenstergröße ist zu erwarten, daß die Laufzeit bei steigender Fenstergröße sinkt. Bei der Wahl der Schrittweite sind zwei Effekte zu berücksichtigen. Wenn wir eine große Anzahl Streutabellen verwenden (also eine kleine Schrittweite), vermeiden wir passende Tupel, die sich in der Nähe der erwarteten Position befinden, abzuschneiden. Allerdings wird die Zick-Zack-Suche aufwendiger, je mehr Streutabellen wir verwenden.

Betrachten wir zunächst die Meßergebnisse in Abbildung 5.10. Dort ist für feste Puffergrößen jeweils der *Anteil der Fehlversuche* in Abhängigkeit von der Anzahl der verwendeten Streutabellen dargestellt. Die Ergebnisse sind wie erwartet. Generell sinkt die Anzahl der Fehlversuche für größere Puffer- bzw. Fenstergrößen, da die Wahrscheinlichkeit, das passende Tupel in einer der Streutabellen zu finden, steigt. Für kleine Fenstergrößen ist die Schrittweite eher unerheblich. In diesem Fall ist die Anzahl der auf eine große Schrittweite zurückzuführenden Fehlversuche im Vergleich zur Gesamtzahl der Fehlversuche klein. Für große Fenstergrößen ist die Gesamtanzahl der Fehlversuche dagegen klein und unterschiedliche Schrittweiten machen sich deutlich bemerkbar. (Die mit (t) markierten Graphen beschreiben die theoretischen Werte

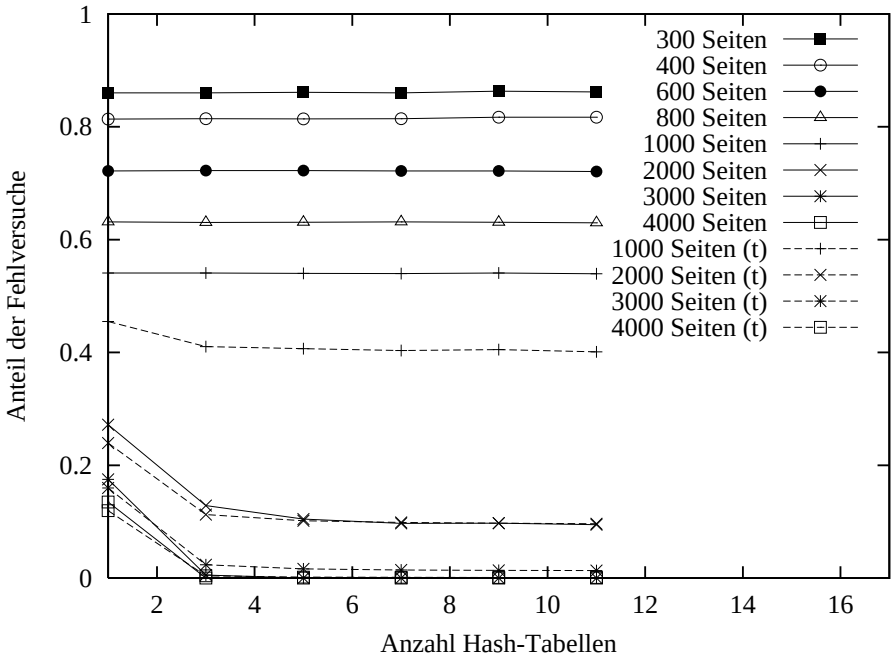


Abbildung 5.10: Auswirkungen der Parameter: Anteil der Fehlversuche

für den Fall, daß die Abweichung von der vollkommenen Ballung – wie in Abschnitt 5.2.7.2 dargestellt – durch eine Normalverteilung beschrieben werden kann.)

Kommen wir nun zu den Meßergebnissen in Abbildung 5.11. Dort ist für feste Puffergrößen jeweils die *Gesamtlaufzeit* in Abhängigkeit von der Anzahl der verwendeten Streutabellen dargestellt. Die Gesamtlaufzeit für eine feste Puffergröße wird für eine größere Anzahl von Streutabellen von der Zick-Zack-Suche dominiert und steigt daher mit der Anzahl der Streutabellen. Für kleine Puffergrößen führt dies wegen der konstanten Fehlversuchs-Rate zu einem monotonen Anstieg der Gesamtlaufzeit mit der Anzahl der Streutabellen. Für große Puffergrößen überlagern sich die beiden Effekte, so daß die optimale Anzahl größer als eins ist. Aufgrund der Messungen haben wir uns für einen Kompromiß entschieden und für unseren weiteren Messungen das Fenster in fünf Tabellen eingeteilt.

Vergleich mit anderen Verfahren Wir vergleichen den Diagonalverbund mit dem blockorientierten Geschachtelte-Schleifen-Verbund und dem GRACE-Streu-Verbund. Die Ergebnisse für die Gesamtlaufzeit der drei Algorithmen beim Verbund von *Order* und *Lineitem* auf dem Attribut *Orderkey* sind in Abbildung 5.12 dargestellt.

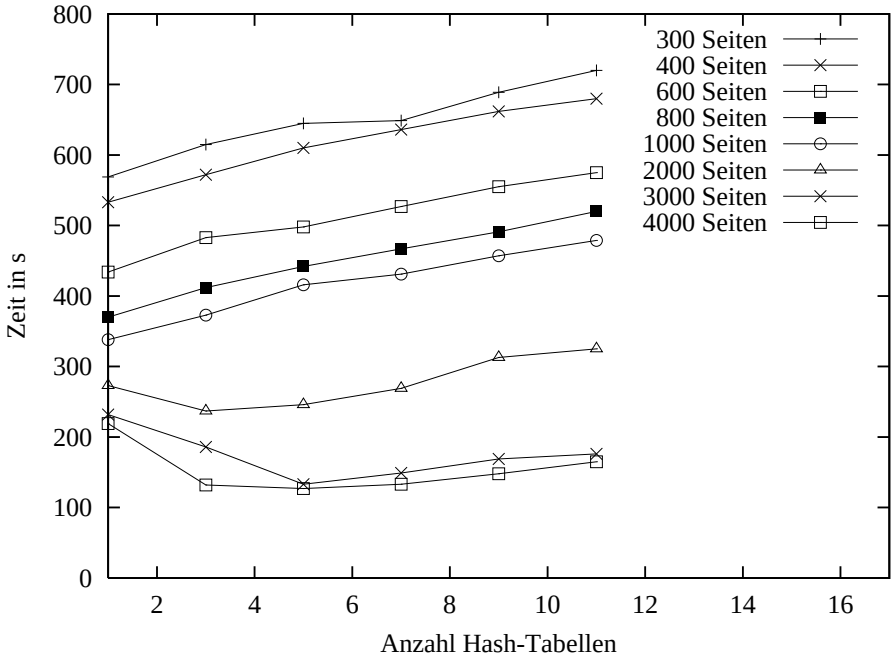


Abbildung 5.11: Auswirkungen der Parameter: Gesamtlaufzeit

Gesamtkosten Der blockorientierte Geschachtelte-Schleifen-Verbund zeigt die geringste Leistung. Das ist nicht überraschend, da das Verhältnis von Puffergröße zu Datengröße sehr ungünstig ist.

Für hinreichend große Puffergrößen (> 3000 Seiten oder 6% von $||R_1||$) ist der Diagonalverbund dem GRACE-Streu-Verbund deutlich überlegen, da in diesem Fall alle Tupel in der ersten Phase verbunden werden und keine zweite Phase zur Verarbeitung der Fehlversuche erforderlich ist. Für mittlere Puffergrößen (zwischen 1000 und 3000 Seiten ist der Diagonalverbund immer noch schneller und nur für sehr kleine Puffergrößen (< 1000 Seiten oder 2% von $||R_1||$) ist der GRACE-Streu-Verbund ein wenig schneller als der Diagonalverbund. Die erste Phase des Diagonalverbunds verursacht einen relativ kleinen Zusatzaufwand, ist aber in der Lage einige Tupel zu verbinden und so die Last des GRACE-Streu-Verbunds in der zweiten Phase zu reduzieren (vgl. Abbildung 5.13). Allerdings ist diese Reduktion nicht ausreichend, um den Zusatzaufwand der ersten Phase auszugleichen.

Prozessorkosten Betrachten wir nun die Prozessorkosten der Verbundalgorithmen in Abbildung 5.14.

Je mehr Puffer zur Verfügung steht, desto geringer fallen die Kosten für den blockorientierten Geschachtelte-Schleifen-Verbund aus. Dies ist offensichtlich, da sich die Anzahl der erforderlichen Schleifen mit dem Steigen der Puffergröße reduziert.

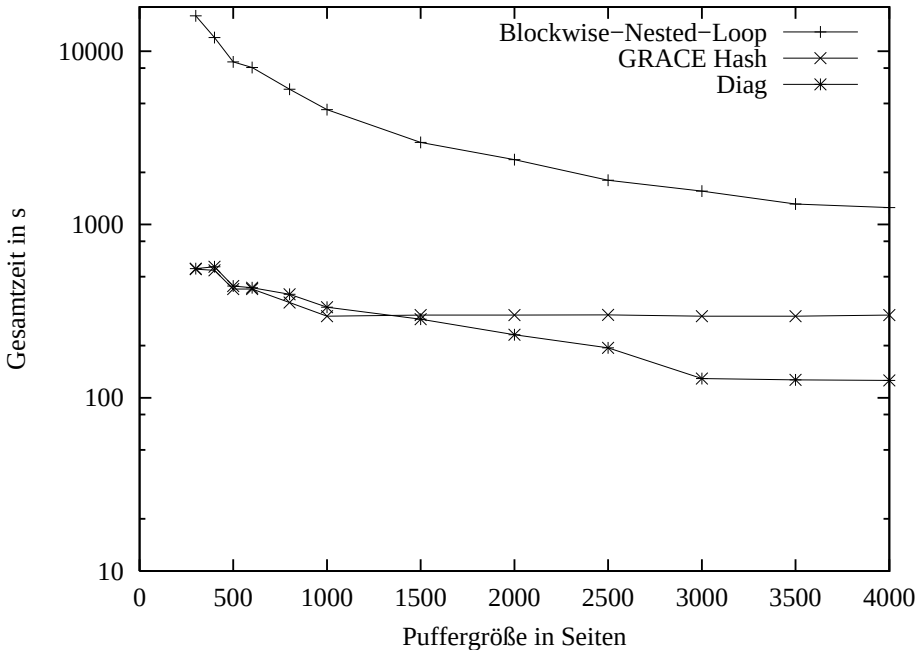


Abbildung 5.12: Gesamtlaufzeit der Algorithmen

Die Größe der Streuverzeichnisse ist für die Prozessorkosten des GRACE-Streu-Verbunds irrelevant, solange sie hinreichend groß sind. Die Prozessorkosten des GRACE-Streu-Verbunds bestehen aus den Kosten alle Tupel aus *Order* zu streuen, alle Tupel aus *Lineitem* zu streuen, die Tupel aus *Order* noch einmal während der Suchphase zu streuen und $|Lineitem|$ Einträge in der Streutabelle zu suchen. Daher sind die Kosten fast konstant.

Die Prozessorkosten der ersten Phase des Diagonalverbunds sind – unabhängig von der Puffergröße – nahezu konstant, da die beiden Relationen *Order* und *Lineitem* nur einmal gelesen werden (vgl. Abbildung 5.15). Der leichte Anstieg entsteht durch die Kosten für das Verbinden der Tupel. Je mehr Puffer zu Verfügung steht, desto mehr Tupel finden ihren Verbundpartner in der ersten Phase. (Wir haben bei dieser Messung die Fehlversuche nicht in einer temporären Relation abgelegt.) Die insgesamt abfallenden Prozessorkosten des Diagonalverbunds (vgl. Abbildung 5.14) entstehen durch die die Reduktion der Größe der temporären Relation und die dadurch fallenden Kosten des GRACE-Streu-Verbunds in der zweiten Phase.

E/A-Kosten In Abbildung 5.16 sind die E/A-Kosten der drei Algorithmen dargestellt. Für den blockorientierten Geschachtelte-Schleifen-Verbund ergibt sich das gleiche Bild wie bei den Prozessorkosten. Je größer die Puffergröße wird, desto geringer wird die Anzahl der Schleifen und desto geringer werden die Kosten.

Auch für den GRACE-Streu-Verbund gehen die E/A-Kosten mit wachsender Puffergröße zu-

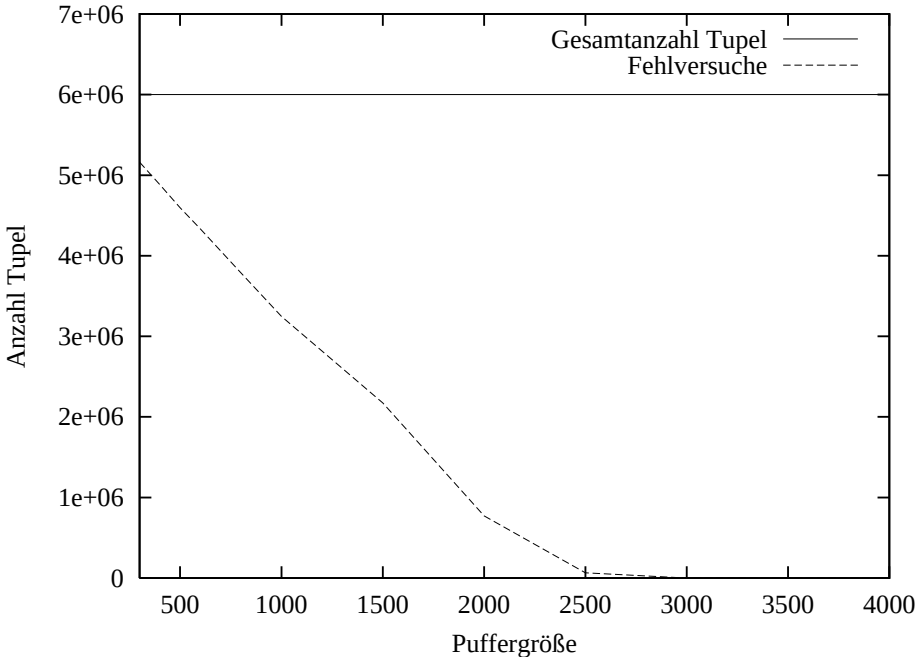


Abbildung 5.13: Gesamtanzahl der Fehlversuche

rück. Ab einer bestimmten Puffergröße (ungefähr 1000 Seiten) und damit ab einer bestimmten Größe der Partitionen, werden die Such- und Latenzzeiten sehr klein und die E/A-Kosten werden von den reinen Transferkosten bestimmt. Diese sind beim GRACE-Streu-Verbund allerdings konstant, da die beiden Relationen in jedem Fall zweimal vom Sekundärspeicher gelesen und einmal auf dem Sekundärspeicher zwischengespeichert werden. Daher sind die E/A-Kosten für Puffergrößen über 1000 Seiten nahezu konstant.

Beim Diagonalverbund sind die E/A-Kosten ab 3000 Seiten (6% von $||$*Order*$||$) konstant. Dann ist das Fenster ausreichend groß, um alle Tupel in der ersten Phase zu verbinden. In diesem Fall müssen *Order* und *Lineitem* nur einmal gelesen werden und das ist für diesen Algorithmus die Untergrenze. Wenn der Puffer kleiner als 3000 Seiten ist, müssen die beiden Relationen auch einmal in der ersten Phase gelesen werden. Zusätzlich müssen dann aber die Fehlversuche aus *Lineitem* in eine temporäre Relation geschrieben werden, die dann noch einmal mit *Order* verbunden wird. Durch eine Reduktion der Puffergröße wächst die temporäre Relation und damit auch die Kosten des GRACE-Streu-Verbunds in der zweiten Phase.

5.2.8.3 Zusammenfassung des Leistungsvergleichs

Wenn die Relationen nach dem Zeitpunkt des Einfügens geballt sind, ist der Diagonalverbund sehr leistungsfähig. Er war in unseren Messungen bis zu $2\frac{1}{2}$ mal schneller als der GRACE-Streu-

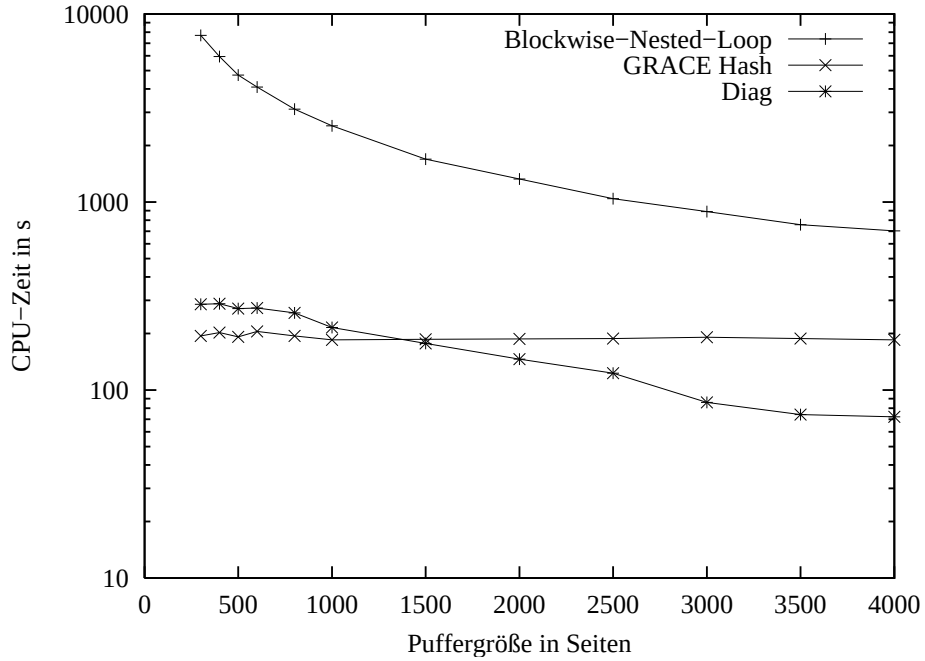


Abbildung 5.14: Prozessorkosten der Algorithmen

Verbund und bis zu 28 mal schneller als der blockorientierte Geschachtelte-Schleifen-Verbund. Er benötigt ausreichend Speicher (etwa 6% von $||R_1||$ in unserem Beispiel), um sein Potential voll auszuschöpfen, ist aber auch bei kleineren Puffergrößen zufriedenstellend.

Die Tatsache, daß 6% der Größe der *kleineren* Relation R_1 ausreichen, um das Auftreten von Fehlversuchen zu vermeiden, ist natürlich eine Folge der spezifischen Verteilung unserer Daten. Beim Versuch Relationen, die nicht nach dem Zeitpunkt des Einfügens geballt sind, zu verbinden, wird der Diagonalverbund versagen. In diesem Fall werden nur wenige Tupel in der ersten Phase verbunden und viele Fehlversuche auf dem Sekundärspeicher abgelegt. Die Größe der temporären Relation ist dann im Durchschnitt $(1 - \frac{\text{Puffergröße}}{||R_1||}) \cdot ||R_N||$. Im Optimalfall unserer Messungen (die Puffergröße beträgt 6% von $||R_1||$) würde die temporäre Relation 94% aller Tupel aus R_N enthalten. Das bedeutet, daß die erste Phase ein einmaliges, überflüssiges Kopieren der größeren Relation ist.

Des weiteren ist festzustellen, daß die Größe der temporären Relation zwar linear von der Größe von R_N abhängt, aber die Frage, ob überhaupt Fehlversuche auftreten, hängt von dem Verhältnis zwischen der Verteilung der Tupel in R_1 und der Größe des Fensters ab. Der Diagonalverbund kommt also – bei entsprechender Ballung der Relationen – mit Primärspeichergrößen zurecht, die im Vergleich zu den zu verbindenden Relationen sehr klein sind.

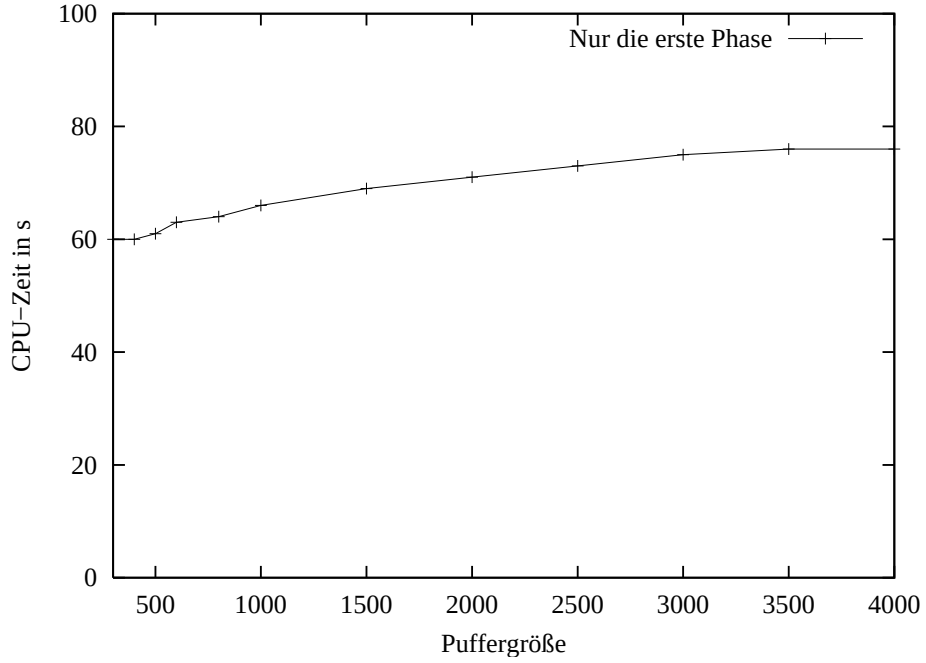


Abbildung 5.15: Prozessorkosten der ersten Phase des Diagonalverbunds

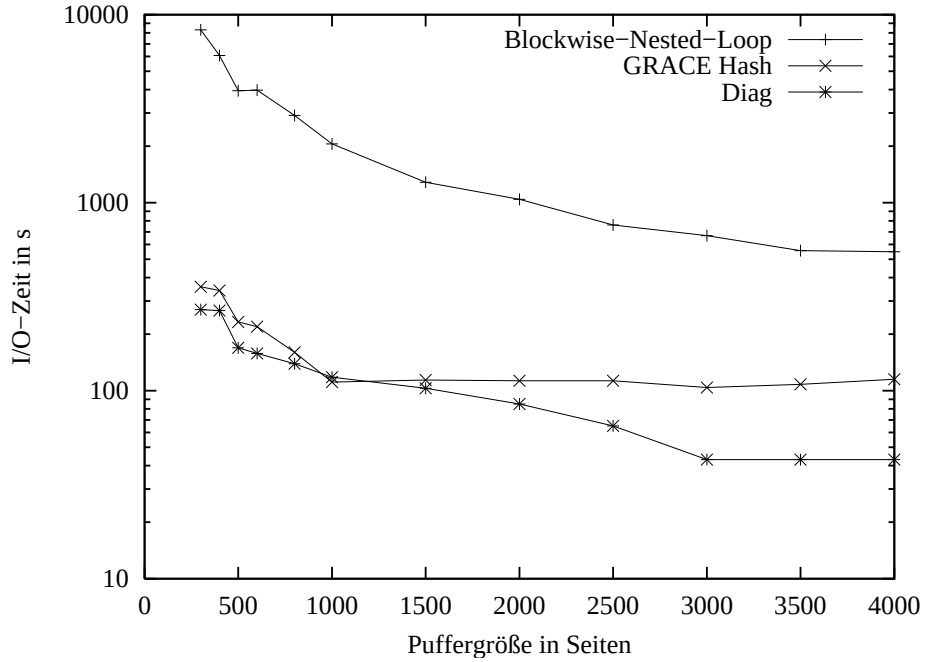


Abbildung 5.16: E/A-Kosten der Algorithmen

5.3 Gruppierung und Aggregation

In diesem Abschnitt betrachten wir die gemeinsamen Grundlagen der Abschnitte 5.4–5.6, die drei Muster beschreiben, die Spezialfälle von Gruppierung und Aggregation behandeln.

5.3.1 Einleitung

Zu Beginn der Entwicklung relationaler Datenbankverwaltungssysteme wurde auf die effiziente Implementierung von Gruppierung und Aggregation wenig Wert gelegt, da die Systeme für OLTP-Anwendungen konzipiert waren. Die zur Gruppierung und Aggregation verwendeten Operatoren wurden vom Anfrageübersetzer nicht in die Optimierung einbezogen, sondern stets am Ende des Auswertungsplans eingebaut, und die Operatoren wurden durch Sortierung implementiert. In der Mitte der neunziger Jahre wurden Entscheidungsunterstützungssysteme, also OLAP-Anwendungen, zum Gegenstand der Forschung und man begann, Gruppierung und Aggregation genauer zu untersuchen. Dazu wurden die Operatoren bei der algebraischen Anfrageoptimierung berücksichtigt, [YL94, CS94, YL95, GHQ95], parallele Algorithmen zur Implementierung der Operatoren entwickelt [SN95] und vorhandene Anfragesprachen um Konzepte zur einfacheren Formulierung von entsprechenden Anfragen erweitert [GBLP96, CR96].

Unser Ansatz besteht darin, die Möglichkeiten der algebraischen Optimierung um die der physischen Optimierung zu erweitern. Dazu stellen wir Algorithmen vor, die für bestimmte Anfragemuster eine Alternative zu den bekannten Gruppierungs- und Aggregationsalgorithmen darstellen. Die Möglichkeiten der Parallelisierung der verwendeten Algorithmen oder der Erweiterung der Anfragespachen werden davon nicht beeinflußt.

5.3.2 Stand der Forschung

Die formale Betrachtung von Gruppierung und Aggregation begann Klug [Klu82], der als erster präzise Definitionen für Aggregatfunktionen angab und die relationale Algebra und das Relationenkalkül erweiterte, um diese zu unterstützen. Ceri und Gottlob [CG85] haben dann Klugs *Aggregate formation*-Operator verallgemeinert, um eine Übersetzung von SQL in die relationale Algebra zu ermöglichen. Dayal [Day87] hat zum einen eine Implementierung dieses *Generalized Aggregation*-Operators (*GAgg*) angegeben und zum anderen mögliche Vorgehensweisen zur Berücksichtigung der Aggregation bei der Anfrageoptimierung beschrieben. Alle semantischen Anfragemuster, die wir identifizieren, enthalten den *GAgg*-Operator. Implementierungtechniken für den *GAgg*-Operator findet man im Memorandum von Epstein [Eps79], im Papier von Dayal [Day87] und im Übersichtsartikel von Graefe [Gra93]. Shatdal und Naughton beschreiben Implementierungstechniken für parallele Versionen des Operators. Wir beschränken uns allerdings auf sequentielle Implementierungen. Chatziantoniou und Ross beschreiben eine Erweiterung von SQL, die die Formulierung einiger Anfragen, die Gruppierung und Aggregation

enthalten, erleichtert. Außerdem führen sie den relationalen Operator Φ ein, der die Suche nach einer effizienten Implementierung für die betrachteten Anfragen erleichtert [CR96]. Allerdings führt der beschriebene Algorithmus – wenn er anwendbar ist – in den von unseren Mustern betrachteten Fällen zu den üblichen Implementierungen. Die Verwendung unserer Operatoren führt im Vergleich zur Verwendung der Operatoren aus Graefes Übersichtsartikel [Gra93] zu einer Verbesserung um den Faktor zwei in Bezug auf Geschwindigkeit oder Speicherverbrauch.

5.3.3 Bezeichnungen

In SQL können Werte mit oder ohne vorheriger Gruppierung aggregiert werden. Als Beispiel für den zweiten Fall betrachten wir die Anfrage

select sum(Gehalt), max(Gehalt)
from Angestellter

wobei *Angestellter* eine Relation mit dem Schema

Angestellter(AngID, Name, Gehalt, Abteilung).

ist. Die Anfrage addiert die Gehälter aller Angestellten und bestimmt das maximale Gehalt. Hier wird ein Tupel bestehend aus zwei Aggregatfunktionen auf eine Menge von Tupeln atomarer Werte, also eine Relation, angewandt. Epstein [Eps79] unterscheidet *scalar aggregates*, wie zum Beispiel *count*, *sum* oder *avg*, die einen einzelnen Wert für eine Relation ergeben, und *aggregate functions*, die eine Menge von Werten für eine Relation liefern. Wir werden hierfür die Begriffe *skalare Aggregatfunktion* und *gruppierende Aggregatfunktion* verwenden. In der Algebra bezeichnen wir die Anwendung eines Tupels skalarer Aggregatfunktionen *agg* auf eine Relation mit dem Operator γ_{agg} („klein Gamma“), der unten definiert wird.

Als ein Beispiel für Aggregation mit vorheriger Gruppierung betrachten wir die Anfrage

select sum(Gehalt), max(Gehalt), Abteilung
from Angestellter
group by Abteilung

Hier wird ein Tupel aus zwei gruppierenden Aggregatfunktionen auf eine Relation bzw. ein Tupel aus zwei skalaren Aggregatfunktionen auf jede Gruppe einer Relation angewandt. Jede Gruppe besteht dabei aus der Menge aller Tupel aus *Angestellter*, bei denen das in der *group by*-Klausel angegebene Attribut den gleichen Wert annimmt. Diese Kombination der Gruppierung mit anschließender Anwendung eines Tupels skalarer Aggregatfunktionen auf die Gruppen wird durch den Operator $\Gamma_{A;agg}$ bezeichnet. Dabei bezeichnet A die Attributmenge, nach der

gruppiert wird, und *agg* – wie oben – ein Tupel skalarer Aggregatfunktionen. Dieser Operator entspricht Dayals *GAgg*-Operator [Day87].

Für die folgenden Definitionen dieser beiden Operatoren verwenden wir die Notation von Maier [Mai83]: Kleinbuchstaben werden für Tupel und Relationen und Großbuchstaben für Attributmengen und Relationenschemata verwendet, $r(R)$ bedeutet entweder, daß r eine Relation mit Schema R ist, oder, daß wir die Projektion von r auf die in R enthaltenen Attribute betrachten. $\mathcal{P}(m)$ bezeichnet die Potenzmenge von m und $\circ$ den Tupelkonstruktionsoperator. Damit sind die beiden Operatoren wie folgt definiert:

Seien $r(R)$ und $o(O)$ Relationen, $agg : \mathcal{P}(r) \rightarrow o$ ein Tupel aus skalaren Aggregatfunktionen und $A \subseteq R$ eine Attributmenge. Dann ist

$$\begin{aligned} \gamma_{agg}(r) &:= \{agg(r)\} \\ \Gamma_{A;agg}(r) &:= \{t(A) \circ a : t \in r \wedge a = agg(\{s \in r : s(A) = t(A)\})\}. \end{aligned}$$

Beispiel Sei a eine Relation, die Informationen über Angestellte enthält, mit dem Schema

Angestellter(AngID, Name, Gehalt, Abteilung).

und der Ausprägung

$$\begin{aligned} a = \{ \quad & (1, \text{„Hinz“}, 3000.00, \text{„Produktion“}), (2, \text{„Kunz“}, 2800.00, \text{„Produktion“}), \\ & (3, \text{„Meier“}, 3300.00, \text{„Vertrieb“}), (4, \text{„Müller“}, 3500.00, \text{„Vertrieb“}), \\ & (5, \text{„Schmidt“}, 3100.00, \text{„Vertrieb“}) \quad \}. \end{aligned}$$

Wenn wir *agg* für alle $q \in \mathcal{P}(a)$ durch

$$agg(q) := \left(\sum_{t \in q} t.Gehalt \ , \ \max_{t \in q}(t.Gehalt) \right),$$

definieren, können wir die Summe aller Gehälter und das maximale Gehalt mit Hilfe des γ-Operators bestimmen:

$$\gamma_{agg}(a) = \{(15700.00, 3500.00)\}.$$

Wenn wir die Summe aller Gehälter und das maximale Gehalt *für jede Abteilung* benötigen, benutzen wir den Γ-Operator:

$$\Gamma_{(Abteilung);agg}(a) = \{(\text{„Produktion“}, 5800.00, 3000.00), (\text{„Vertrieb“}, 9900.00, 3500.00)\}.$$

Obwohl $\Gamma_{\emptyset;agg}(r) = \gamma_{agg}(r)$ ist, behalten wir beide Operatoren bei, da dies die weitere Darstellung vereinfacht.

5.3.4 Implementierung

Bisher gibt es drei grundlegende Methoden zur Implementierung des Γ-Operators: die erste basiert auf geschachtelten Schleifen, die zweite auf Sortierung und die dritte auf der Verwendung von Streufunktionen [Gra93]. Da die auf geschachtelten Schleifen beruhenden Algorithmen bei weitem nicht so leistungsfähig sind wie die beiden anderen Verfahren und da die speziellen Eigenschaften dieser Verfahren in normalen relationalen Datenbankverwaltungssystemen nicht benötigt werden, kommen diese selten zum Einsatz. Graefe [Gra93] schreibt, daß die auf Sortierung und Streuen beruhenden Algorithmen etwa gleich leistungsfähig sind, so daß es in einem relationalen Datenbankverwaltungssystem üblicherweise ein Modul gibt, das Gruppierung, Aggregation und und die Entfernung von Duplikaten mit einem dieser beiden Ansätze implementiert. Die Situation ist der des Verbundoperators sehr ähnlich. Für einen algebraischen Operator gibt es unterschiedliche Implementierungen. Falls nicht anders bemerkt, verwenden wir Streu-Implementierungen der Operatoren.

In den folgenden Abschnitten werden wir algebraische Operatoren für semantische Anfragemuster vorstellen, die spezielle Fälle von γ, Γ und eine Kombination von Γ und Verbund abdecken.

5.4 Der Max-Operator

Dieser und die beiden folgenden Abschnitte bestehen aus jeweils fünf Teilen. Wir beginnen mit der Beschreibung eines semantischen Anfragemusters und definieren dann einen Operator, der die Semantik des Musters erfaßt. Anschließend skizzieren wir eine mögliche Implementierung des Operators. Der vierte Teil enthält den Vergleich von zwei Auswertungsplänen. Der erste Plan ist jeweils ein traditioneller Plan und der zweite Plan nutzt den neuen Operator. Der fünfte Teil dokumentiert die Möglichkeiten des Konzepts durch den Nachweis von Leistungsverbesserungen.

Zur Durchführung der Experimente wurden die neuen Operatoren in unser Laufzeitsystem AODB integriert. Die Anfragen wurden an eine TPC-D Datenbank mit dem Skalierungsfaktor 1 [TPC95] gestellt. AODB lief dabei auf einer SUN UltraSparc 2 mit 256 MB Primärspeicher unter Solaris 2.6.

5.4.1 Semantisches Anfragemuster

Beispiel Sei *Angestellter* eine Relation mit dem Schema

Angestellter(AngID, Name, Gehalt, AbtID).

An diese Relation stellen wir folgende Anfrage:

Welcher Angestellte bezieht das höchste Gehalt ?

Die Übersetzung in SQL ergibt:

```
select  Name
from    Angestellter
where Gehalt = ( select max(Gehalt)
                 from  Angestellter)
```

Die meisten kommerziellen Datenbankverwaltungssysteme (zumindest alle, die wir testen konnten) interpretieren diese SQL-Anfrage sehr direkt. Das Ergebnis wird in zwei Schritten erzeugt. Im ersten Schritt wird die Unteranfrage verwendet, um den maximalen Wert des Attributs *Gehalt* zu bestimmen. Im zweiten Schritt werden alle Tupel aus *Angestellter*, die das Prädikat erfüllen, bestimmt. Beide Schritte können entweder mit Hilfe eines Index[2] oder durch sequentielles Lesen der Relation *Angestellter* durchgeführt werden. In jedem Fall muß das System die Gehälter und die entsprechenden Tupel zweimal betrachten. Offensichtlich ist dies weder wünschenswert noch notwendig, insbesondere falls die Anzahl der resultierenden Tupel klein ist und daher problemlos in den Primärspeicher paßt (was typischerweise der Fall ist). Diese Situation ist noch unangenehmer, falls die betrachtete Relation keine Basisrelation, sondern das Ergebnis einer Unteranfrage ist (zum Beispiel weil *Angestellter* eine Sicht ist). In diesem Fall gibt es mit Sicherheit keinen Index in dem man die Tupel nachschlagen kann, so daß entweder die gesamte Unteranfrage noch einmal ausgeführt oder das Ergebnis der Unteranfrage in einer temporären Relation materialisiert werden muß.

Das allgemeine semantische Anfragemuster wählt

„alle Tupel, für die ein gegebener Ausdruck maximal ist“.

Die folgende Diskussion behandelt nur die Maximierung. Die Minimierung kann analog behandelt werden. Daher nennen wir dieses Muster *globales Extremum-Muster*.

5.4.2 Definition des Operators

Dieses Muster führt uns direkt zu der folgenden Definition des Max-Operators (vgl. [WM99]), der eine Anpassung des von Cluet und Moerkotte [CM93] vorgeschlagenen Max-Operators an den relationalen Kontext ist.

Sei $r(R)$ eine Relation, X eine geordnete Menge atomarer Werte und $exp : r \rightarrow X$ ein Ausdruck, der maximiert werden soll. Dann definieren wir

$$\text{Max}_{exp}(r) \quad := \quad \{t \in r : exp(t) = \max(exp(r))\}.$$

[2] Die Möglichkeiten der Nutzung von Indexstrukturen werden in Abschnitt 5.4.4 erörtert.

Im Gegensatz zu dieser Definition ist das Ergebnis des Operators aus [CM93] geschachtelt. Ein solches Ergebnis ist für unsere Anwendung ungeeignet, da Relationen im traditionellen relationalen Modell flach sind. Außerdem haben Cluet und Moerkotte keine Implementierung angegeben.

Wir schlagen den Max-Operator zur Auswertung von Anfragen, die dem globalen Extremum-Muster entsprechen, vor.

5.4.3 Implementierung des Operators

Eine einfache, effiziente, aber auch naive Implementierung dieses Operators ist:

```
MAX(r,exp)         /* Relation r, zu maximierender Ausdruck exp */
M = leereMenge
waehle m aus r
fuege m zu M hinzu
foreach(t aus r) {
  if(exp(t) > exp(m)) {
    M = leereMenge
    m = t
    fuege m zu M hinzu
  } else if(exp(t) = exp(m)) {
    fuege t zu M hinzu
  }
}
return M
```

Diese Implementierung betrachtet jedes Tupel aus r genau einmal und ist daher effizienter als das übliche „2-Schritt-Verfahren“. Das einzige Problem bei dieser Implementierung kann die Größe von M sein. Es gibt kein Problem, solange genügend Primärspeicher für ganz M zur Verfügung steht. Aber das muß nicht einmal dann der Fall sein, wenn das Ergebnis vollständig in den Primärspeicher paßt. Als Beispiel betrachten wir die Relation *Angestellter*. Angenommen, jeder Angestellte bezieht eines der Gehälter DM 20.000, DM 200.000 oder DM 2.000.000 und die Relation *Angestellter* ist zufällig nach steigenden Gehältern sortiert. Außerdem beziehen die meisten der 10 Millionen Angestellten DM 20.000, weniger DM 200.000 und nur einige DM 2.000.000. Dann würde unsere Implementierung des Operators M zuerst mit allen Angestellten, die DM 20.000 erhalten, füllen. Sobald der erste Angestellte mit DM 200.000 gefunden wird, wird M gelöscht und und mit diesem Angestellten initialisiert. Dann beginnt die ganze Prozedur wieder. Also wird M nacheinander an alle Angestellten mit DM 20.000, DM 200.000 und DM 2.000.000 gebunden. Wenn eine dieser Mengen nicht in den Primärspeicher paßt, gibt es ein Problem. Obwohl dieser Fall eher unwahrscheinlich ist sollte er doch berücksichtigt werden.

Ein Ausweg wäre die Tupel auf dem Sekundärspeicher statt im Primärspeicher zu speichern. Dies hat allerdings den entscheidenden Nachteil, daß es schreibenden Zugriff auf den Sekun-

därspeicher erfordert. Es gibt allerdings eine einfache Möglichkeit dynamisch auf die Speicherarmut zu reagieren, wie die folgende modifizierte Implementierung zeigt:

```
  MAX(r,exp)       /* Relation r, zu maximierender Ausdruck exp */
  M = leereMenge
  waehle m aus r
  fuege m zu M hinzu
* ueberlauf = false
  foreach(t aus r) {
    if(exp(t) > exp(m)) {
      M = leereMenge
*     ueberlauf = false
      m = t
      fuege m zu M hinzu
    } else if(exp(t) = exp(m)) {
*     if(M nicht voll) {
        fuege t zu M hinzu
*     } else {
*       ueberlauf = true
*     }
    }
  }
* if(ueberlauf = false) {
    return M
* } else {
*   waehle m aus M
*   return alle Tupel t aus r mit exp(t) = exp(m)
* }
```

Die zentrale Idee dieser Modifikation ist auf das „2-Schritt-Verfahren“ zurückzugreifen, falls der Primärspeicher nicht ausreicht. Sobald zu viele Tupel in M eingefügt wurden, wird der *Überlauf-Indikator* gesetzt und es werden keine Tupel mehr für den aktuellen Maximalwert gesammelt. Wenn ein Tupel mit einen neuen Maximalwert gefunden wird, können wir mit einer kleinen Menge M wieder anfangen. Am Ende des Algorithmus betrachten wir die aktuelle Situation und stellen fest, ob M das vollständige Ergebnis enthält oder ob wir r noch einmal lesen müssen. Dieses Verfahren zur Begrenzung des erforderlichen Primärspeichers werden wir bei der Implementierung des nächsten Operators auch anwenden.

5.4.4 Vergleich der Auswertungspläne

In Abbildung 5.17 werden die beiden möglichen Auswertungspläne gezeigt. Das „2-Schritt-Verfahren“ besteht – in relationaler Algebra ausgedrückt – aus einem γ und einem Verbundoperator. Da der kleinere der beiden Partner des Verbundes aus nur einem Tupel besteht, eignet sich jede einfache Implementierung des Verbundoperators. Die modifizierte Implementierung

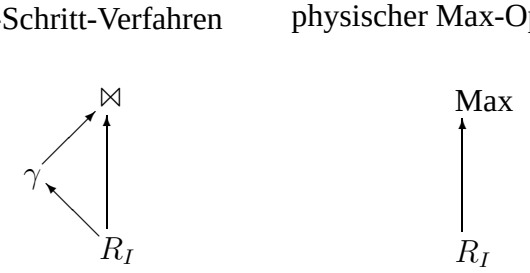


Abbildung 5.17: Auswertungspläne für den (logischen) Max-Operator

des Max-Operators betrachtet jedes Tupel, je nach Größe des Ergebnisses, einmal oder zweimal. Da die Bestimmung des Maximums keinen großen Rechenaufwand erfordert, beansprucht die Produktion der Tupel aus R_I üblicherweise den größten Teil der Gesamtausführungszeit. Daher erwarten wir von der Verwendung des Max-Operators für Anfragen, die dem globalen Extremum-Muster entsprechen, eine Reduktion der Gesamtausführungszeit um 50%, wenn das Ergebnis in den Primärspeicher paßt. Selbst wenn das Ergebnis nicht in den Primärspeicher paßt, greift die modifizierte Implementierung auf das „2-Schritt-Verfahren“ zurück und ist daher nicht langsamer als der konventionelle Plan.

Bemerkung Falls ein Index für das Attribut *Gehalt* existiert, kann dieser offensichtlich zur Beantwortung der Beispielanfrage verwendet werden. Ein solcher Plan wird höchstwahrscheinlich effizienter als unser Plan sein. Allerdings kann man nicht davon ausgehen, für jedes Attribut einen Index zu haben. Außerdem gibt es zwei Fälle, in denen wir sicher keinen Index haben: (1) Wenn die betrachtete Relation das Ergebnis einer Unteranfrage und keine Basisrelation ist, und (2) wenn der zu maximierende Ausdruck nicht einfach nur ein Attribut, sondern ein komplexerer Ausdruck ist.

5.4.5 Leistungsvergleich

Um zu zeigen, daß der Max-Operator den versprochenen Leistungszuwachs tatsächlich erbringt, haben wir die folgende Anfrage zweimal ausgewertet – einmal mit Hilfe der „Zwei-Schritt-Verfahrens“ und einmal mit Hilfe unserer Implementierung des Max-Operators.

```
select O_Clerk, O_TotalPrice
from   Order
where O_TotalPrice = (select max(O_TotalPrice)
                      from  Order)
```

Plan	Gesamt-zeit	Prozessor-zeit
„Zwei-Schritt-Verfahren“	39 s	9.0 s
Max-Operator	19 s	5.7 s

Tabelle 5.7: Ausführungszeiten für den Max-Operator

Die Ergebnisse in Tabelle 5.7 zeigen die erwartete Verbesserung um 50%.

5.5 Der Γ^{max}-Operator

5.5.1 Semantisches Anfragemuster

Beispiel Wir betrachten die folgende Anfrage:

> Gib mir zu jeder Abteilung den Namen des höchstbezahlten Angestellten.

Auf der Grundlage der Relation *Angestellter(AngID, Name, Gehalt, AbtID)* aus dem letzten Abschnitt ist

```
select  a1.Name, a1.AbtID
from    Angestellter a1
where a1.Gehalt = (select  max(a2.Gehalt)
                   from    Angestellter a2
                   where a1.AbtID = a2.AbtID).
```

eine geeignete SQL Übersetzung dieser Anfrage. Je nach Qualität des Optimierers gibt es zwei Möglichkeiten, wie diese Anfrage von einem aktuellen Datenbankverwaltungssystem ausgewertet wird. Die erste Möglichkeit ist die Unteranfrage tatsächlich einmal für jedes Tupel aus *Angestellter* auszuführen. Dies ist offensichtlich sehr langsam, unabhängig davon, ob Indizes für *Angestellter* existieren. Die zweite Möglichkeit ist die Anfrage in zwei Schritten in ähnlicher Weise wie im letzten Abschnitt auszuwerten. Im ersten Schritt wird *Angestellter* nach *AbtID* gruppiert und *max(Gehalt)* für jede Gruppe bestimmt. Im zweiten Schritt wird das Ergebnis des ersten Schritts mit *Angestellter* auf den Attributen *AbtID* und *Gehalt* verbunden.

Das allgemeine semantische Anfragemuster ist dem des letzten Abschnitts sehr ähnlich. Es wählt

„alle Tupel, für die ein gegebener Ausdruck innerhalb einer Gruppe maximal ist“.

Die Minimierung kann auch hier wieder analog behandelt werden, also nennen wir dieses Muster *lokales Extremum-Muster*.

5.5.2 Definition des Operators

Wir definieren den $\Gamma^{\max}$-Operator wie folgt (vgl. [WM99]):

Sei $r(R)$ eine Relation, $A \subseteq R$ eine Attributmenge, X eine geordnete Menge atomarer Werte und $exp : r \rightarrow X$ ein Ausdruck, der maximiert werden soll. Dann definieren wir

$$\Gamma^{\max}_{A;exp}(r) \quad := \{ \quad t \in r : exp(t) = \max(exp(\{s \in r : s(A) = t(A)\})) \quad \}$$

Der $\Gamma^{\max}$-Operator bestimmt für jede durch die Attribute in A definierte Gruppe die Menge der Tupel, für die den Wert von *exp* innerhalb der Gruppe maximal ist. Die Definition des Operators spiegelt also die Semantik des lokalen Extremum-Musters direkt wieder.

5.5.3 Implementierung des Operators

Die Implementierung dieses Operators ist eine Erweiterung der Implementierung des Max-Operators. Anstelle einer globalen Menge maximaler Tupel verwendet diese Implementierung eine Menge maximaler Tupel je Gruppe. Jedes Tupel wird also zunächst einer Gruppe zugeordnet, bevor es in der gleichen Weise wie beim Max-Operator verarbeitet wird. Es ergibt sich folgende Implementierung:

```
GAMMAMAX(r,exp,A) /* Relation r, zu maximierender Ausdruck exp */
                  /* A Menge der Gruppierungsattribute         */
foreach(t aus r) {
  if(fuer t existiert bereits eine Gruppe g) {
    M = Menge der maximalen Elemente fuer g
    waehle m aus M
    if(exp(t) > exp(m)) {
      M = leereMenge
      fuege t zu M hinzu
    } else if(exp(t) = exp(m)) {
      fuege t zu M hinzu
    }
  } else {
    lege eine neue Gruppe mit der Menge M an
    M = leereMenge
    fuege t zu M hinzu
  }
}
return alle Mengen aller Gruppen
```

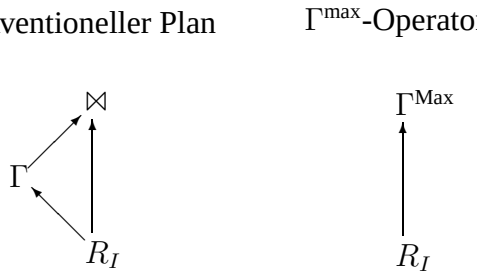


Abbildung 5.18: Auswertungspläne für den (logischen) Γ^{max}-Operator

Die Gruppierung wird – wie schon in Abschnitt 5.3.4 erwähnt – durch Streuen implementiert.

Die Behandlung eines Speicherüberlaufs erfolgt auf zwei Ebenen. Zunächst können wir, wie im Fall des Max-Operators, Indikatoren verwenden, um einen Überlauf festzustellen und dynamisch zum konventionellen Plan zurückzukehren. Dabei ist es sinnvoll, einen Indikator je Gruppe zu verwenden, so daß nur die Gruppen verbunden werden müssen, die tatsächlich einen Überlauf hatten. Wenn die Anzahl der Gruppen zu groß wird, um sie im Primärspeicher zu halten, müssen wir einen weiteren Schritt zurück machen und auf hybrides Streuen zurückgreifen [GBC98].

5.5.4 Vergleich der Auswertungspläne

Wir werden die erste in Abschnitt 5.5.1 beschriebene Alternative, die geschachtelte Ausführung, hier nicht diskutieren, da sie $|R_I|+1$-maliges Lesen von R_I erfordert und damit für fast alle Größen von R_I eine nicht akzeptable Leistung zeigt. Die beiden verbleibenden Auswertungspläne werden in Abbildung 5.18 gezeigt. Der konventionelle Plan besteht aus einem Γ- und einem Verbundoperator. Wie wir bereits bei der Betrachtung des „Zwei-Schritt-Verfahrens“ im letzten Abschnitt festgestellt haben, greift dieser konventionelle Plan auf jedes Tupel aus R_I zweimal zu. Der Γ^{max}-Operator muß hingegen auf die Tupel nur dann ein zweites Mal zugreifen, wenn das Ergebnis nicht in den Primärspeicher paßt. Da die Bestimmung der Gruppe zu einem Tupel den erforderliche Rechenaufwand nicht deutlich erhöht, beansprucht die Produktion der Tupel aus R_I auch hier den größten Teil der Gesamtausführungszeit. Daher können wir in diesem Fall mit der gleichen Leistungssteigerung wie beim Max-Operator rechnen (50%). Es ist hier aber unwahrscheinlicher, daß dies tatsächlich gelingt, da die Wahrscheinlichkeit, daß das Ergebnis in den Primärspeicher paßt, abnimmt. Allerdings hat der Γ^{max}-Operator auch im schlechtesten Fall keinen Nachteil gegenüber dem konventionellen Plan, da er in der Lage ist sanft in den konventionellen Plan überzugehen.

Bemerkung Die Möglichkeiten des Einsatzes von Indizes sind hier nicht so offensichtlich wie im letzten Fall. Wir betrachten daher zunächst eine alternative Implementierung des Γ^{max}-Operators. Dazu nehmen wir an, daß die Relation sowohl bezüglich der Gruppierungsattribute aus auch bezüglich der zu maximierenden Attribute sortiert ist. Dann kann eine effiziente Implementierung des Γ^{max}-Operators einfach alle Tupel iterativ betrachten und jeweils die ersten Tupel jeder Gruppe, die die maximalen Werte aufweisen, auswählen. Diese Implementierung ist sehr effizient und benötigt keinen eigenen Speicher. Wenn sie für eine unsortierte Relation verwendet werden soll, wird die Laufzeit daher durch den Sortieraufwand bestimmt (vgl. Tabelle 5.8). Wenn allerdings ein Mehrattributindex für die Gruppierungsattribute und die zu maximierenden Attribute vorhanden ist, kann die Sortierung durch einfaches Auslesen des Indexes erreicht werden. Wenn nur die Gruppierungsattribute mit einem Index indiziert sind, kann die erforderliche Sortierung durch ein weniger aufwendiges Sortierverfahren, daß nur die zu einer Gruppe gehörenden Tupel sortiert, erzielt werden.

5.5.5 Leistungsvergleich

Für diesen Leistungsvergleich haben wir die Anfrage aus Abschnitt 5.4 so modifiziert, daß wir *für jedes Jahr* den Bearbeiter und den Betrag der Bestellung, die den höchsten Umsatz erzielt hat, erhalten.

```
select o1.O_Clerk, year(o1.O_Orderdate) as year, o1.O_TotalPrice
from   Order o1
where o1.O_TotalPrice = (select  max(o2.O_TotalPrice)
                         from    Order o2
                         where year(o1.O_Orderdate) = year(o2.O_Orderdate))
```

Wir haben diese Anfrage mit einem konventionellen Plan, mit der Streu-Implementierung des Γ^{max}-Operators und mit der Sortier-Implementierung des Γ^{max}-Operators ausgewertet. Bezüglich des Vergleichs des konventionellen Plans mit der Streu-Implementierung des Γ^{max}-Operators, ergeben sich aus den Resultaten in Tabelle 5.8 zwei Dinge. (1) Die erwartete Verbesserung um 50% kann tatsächlich erreicht werden und (2) der durch die Gruppierung entstehende zusätzliche Prozessoraufwand beträgt weniger als 25% und wirkt sich daher nicht auf die Gesamtlaufzeit der Anfrage aus (vgl. Tabelle 5.7). Bezüglich der Bewertung der Sortier-Implementierung des Γ^{max}-Operators ergeben sich zwei weitere nennenswerte Ergebnisse. Zum einen stellen wir fest, daß diese Implementierung noch langsamer als der konventionelle Plan ist. Zum anderen stellen wir aber auch fest, daß dies an dem hohen Sortieraufwand liegt. Da zur Auswertung der Anfrage ausreichend Primärspeicher zur Verfügung stand, um die Relation im Primärspeicher zu sortieren, erscheint uns die Verwendung dieser Implementierung nur sinnvoll, wenn ein vorheriges Sortieren nicht erforderlich ist (zum Beispiel, weil wir die sortierten Tupel von einem geballten Index erhalten).

Plan	Gesamt-zeit	Prozessor-zeit
konventioneller Plan	39 s	10.9 s
Γ^{max}-Operator	19 s	7.4 s
Γ^{max} mit Sortierung	80 s	67 s
nur sortieren	78 s	65 s

Tabelle 5.8: Ausführungszeiten für den Γ^{max}-Operator

5.6 Der $\Gamma^{\text{add-in}}$-Operator

5.6.1 Semantisches Anfragemuster

Beispiel Die einführende Anfrage für unser drittes Muster ist:

> Bestimme zu jeder Abteilung das durchschnittliche Gehalt aller Angestellten dieser Abteilung.

Im Gegensatz zum letzten Abschnitt wollen wir allerdings nicht nur den Identifikator *AbtID* sondern auch den Namen der Abteilung. Wenn wir die beiden Relationen

> *Angestellter(AngID, Name, Gehalt, AbtID)* und *Abteilung(AbtID, Name)*

haben, lautet die Anfrage in SQL:

```
select   ab.Name, ab.AbtID, avg(an.Gehalt)
from     Abteilung ab, Angestellter an
where    ab.AbtID = an.AbtID
group by ab.Name, ab.AbtID
```

Wie schon Yan und Larson [YL94, YL95], Chaudhuri und Shim [CS94] und Gupta, Harinarayan und Quass [GHQ95] bemerkt haben, kann die Auswertungszeit für diese Anfrage erheblich reduziert werden, indem man – entgegen der üblichen Auswertungsreihenfolge – die Gruppierung und die Aggregation vor den Verbund „schiebt“. Das bedeutet, daß zunächst *Angestellter* nach *AbtID* gruppiert und *avg(Gehalt)* berechnet wird und das Ergebnis der Aggregation dann mit Abteilung verbunden wird. Wenn nun streubasierte Operatoren für Gruppierung, Aggregation und Verbund verwendet werden, läuft die Verarbeitung folgendermaßen ab: Zunächst wird eine Streutabelle für Gruppierung und Aggregation mit *an.AbtID* als Streuschlüssel konstruiert

und anschließend wird eine Streutabelle für den Verbund mit *an.AbtID* als Streuschlüssel konstruiert. Es werden also zwei Streutabellen mit dem gleichen Inhalt und der gleichen Struktur konstruiert. Offensichtlich kann man den erforderlichen Aufwand reduzieren, wenn man nur eine Streutabelle für beide Aufgaben verwendet. Graefe, Bunker und Cooper haben für „Hash-Teams“ auch Streutabellen wiederverwendet [GBC98]. Allerdings suggeriert ihre Darstellung, daß bei einem Hash-Team, das aus einem Verbund und einem Gruppierungs- und Aggregationsoperator besteht, der Gruppierungs- und Aggregationsoperator stets zum Schluß ausgeführt wird. Außerdem fordern sie, daß die Gruppierungs- und die Verbundattribute identisch sind. Dies ist zur Anwendung des $\Gamma^{\text{add-in}}$-Operators nicht erforderlich. Wir fordern nur, daß es zumindest ein gemeinsames Attribut gibt.

Die generelle Problemstellung, die wir hier betrachten, sind Anfragen, die es ermöglichen

> „den Gruppierungsoperator vor dem Verbundoperator auszuführen und die gemeinsame Gruppierungs- und Verbundattribute haben“.

Wir nennen dieses Muster das *gemeinsame Gruppierungs- und Verbundattribute Muster*.

5.6.2 Definition des Operators

Wir schlagen vor solche Anfragen mit Hilfe des $\Gamma^{\text{add-in}}$-Operators auszuwerten, der wie folgt definiert ist (vgl. [WM99]):

Seien $r(R)$, $q(Q)$ und $p(P)$ Relationen, A und B Attributmengen mit $A \subseteq R$ und $B \subseteq Q$ und $agg : \mathcal{P}(r) \rightarrow p$ ein Tupel skalarer Aggregatfunktionen. Dann definieren wir:

$$\begin{aligned}\Gamma^{\text{add-in}}_{A;agg;B}(r,q) &:= \Gamma_{A;agg}(r) \bowtie_B q \\ &= \{t(A) \circ a \circ u : t \in r \wedge \\ &\quad a = agg(\{s \in r : s(A) = t(A)\}) \wedge \\ &\quad u \in q \wedge (t(A) \circ a)(B) = u(B)\}.\end{aligned}$$

Diese Definition ist relativ allgemein: die gewünschte effiziente Implementierung ist nur möglich, wenn $A \cap B \neq \emptyset$. Wenn diese Bedingung erfüllt ist, kann eine Streutabelle mit dem Streuschlüssel $A \cap B$ für Gruppierung, Aggregation und Verbund verwendet werden. Obwohl die Menge der zu betrachtenden Anfragen durch diese Bedingung eingeschränkt wird, verbleiben einige interessante Anfragen in ihr, da sowohl Verbunde als auch Gruppierung auf Fremdschlüsseln in OLTP-Anwendungen (wie zum Beispiel dem TPC-D-Benchmark [TPC95]) häufig vorkommen.

5.6.3 Implementierung des Operators

Eine einfache Implementierung des Operators könnte so aussehen:

```
GAMMAADDIN(r, s, A, agg, B) /* Die Relation r wird nach der    */
                            /* Attributmenge A gruppiert, die  */
                            /* Aggregationsfunktion agg wird   */
                            /* auf die einzelnen Gruppen       */
                            /* angewandt und das Resultat wird */
                            /* mit der Relation s auf der      */
                            /* Attributmenge B verbunden.      */
foreach(t aus r) {
  if(fuer t existiert bereits eine Gruppe g) {
    fuege t zu g hinzu
  } else {
    lege eine neue Gruppe g an
    initialisiere g mit Werten aus t
  }
}
schliesse die Aggregationen aller Gruppen ab
foreach(t aus s) {
  if(eine zu t passende Gruppe g existiert) {
    verbinde t mit g
    fuege die verbundenen Tupel zum Ergebnis hinzu
  }
}
```

In der ersten Schleife wird zu jedem Tupel die entsprechende Gruppe gesucht. Dazu wird eine Streutabelle verwendet, deren Streuwerte nur aus den Attributen aus $A \cap B$ berechnet werden. Zur Überprüfung, ob ein Tupel tatsächlich zu einer Gruppe gehört, werden aber alle Attribute aus A verwendet. Nach der ersten Schleife werden die Abschlußberechnungen einiger Aggregatfunktionen (wie zum Beispiel *avg*) durchgeführt (vgl. auch [Day87]). In der zweiten Schleife wird die gleiche Streutabelle zur Suche nach Verbundpartnern verwendet. Auch hier werden die Streuwerte nur aus den Attributen aus $A \cap B$ berechnet, aber zur Überprüfung des Verbundprädikats werden hier alle Attribute aus B verwendet.

Das Problem der Behandlung von Überläufen ist orthogonal zum Prinzip dieser Implementierung und kann daher mit den üblichen Mechanismen wie zum Beispiel hybridem Streuen [GBC98] gelöst werden. Die wesentliche Verbesserung besteht darin, daß die Suche nach Verbundpartnern für die Tupel der zweiten (ungruppierten) Eingaberelation mit Hilfe der Streutabelle ausgeführt wird, die für die Gruppierung konstruiert wurde.

5.6.4 Vergleich der Auswertungspläne

Die drei möglichen Auswertungspläne für den $\Gamma^{\text{add-in}}$-Operator sind in Abbildung 5.19 dargestellt. Der traditionelle Plan verbindet erst R_I mit R_J und gruppiert und aggregiert die Tupel anschließend. Das Ausführen von Gruppierung und Aggregation vor dem Verbund führt üblicherweise zur einer Reduktion der Größe eines Verbundpartners und damit zu einer Reduktion der zur Ausführung der Operation notwendigen Zeit [YL94]. Falls R_I zu groß ist, um im

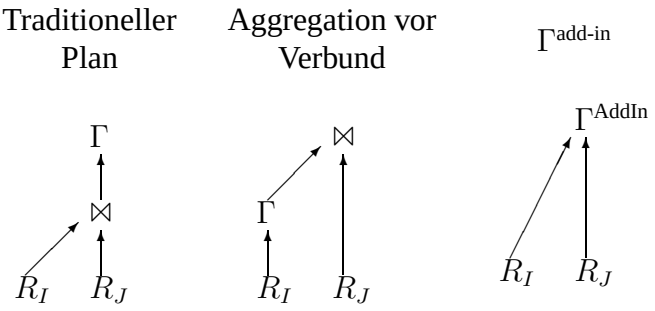


Abbildung 5.19: Auswertungspläne für den $\Gamma^{\text{add-in}}$-Operator

Primärspeicher gehalten zu werden, kann diese Vorgehensweise auch zu einer Reduktion der E/A-Kosten führen. Die Verwendung des $\Gamma^{\text{add-in}}$-Operators bringt darüberhinaus zwei weitere Vorteile. Zum einen müssen die Tupel nicht von einer Streutabelle in die andere kopiert werden, so daß Prozessorkosten gespart werden können. Zum anderen – und dies ist der wichtigere Vorteil – wird zur Auswertung im Vergleich zu den konventionellen Operatoren nur die halbe Menge Primärspeicher benötigt. Daher ist es auch möglich E/A-Kosten zu sparen, wenn die Implementierung auf hybridem Streuen basiert.

5.6.5 Leistungsvergleich

Zur Demonstration der durch den Einsatz des $\Gamma^{\text{add-in}}$-Operators möglichen Leistungssteigerungen verwenden wir die folgende Anfrage:

select P_Partkey, P_Mfgr, P_Brand, P_Type, P_Retailprice,
avg(PS_Supplycost) as avg_supplycost
from Part, Partsupp
where P_Partkey = PS_Partkey
group by P_Partkey, P_Mfgr, P_Brand, P_Type, P_Retailprice
having avg(PS_Supplycost) $>$ 0.9 * P_Retailprice

Die Anfrage liefert die Teile für die die durch die Lieferung anfallenden Kosten mehr als 90% des Verkaufspreises ausmachen. Da die Antwortzeit für alle drei Pläne durch die E/A-Kosten bestimmt wurde, können wir bei den Gesamtlaufzeiten in Tabelle 5.9 keine Verbesserung feststellen. Allerdings können wir wesentliche Reduktionen bei den Prozessorkosten (48% gegenüber dem traditionellen Plan und 16% gegenüber der Aggregation vor dem Verbund) und beim Speicherbedarf (51% gegenüber dem traditionellen Plan und 34% gegenüber der Aggregation vor dem Verbund) feststellen. Die Tatsache, daß die Aggregation vor dem Verbund weniger

Plan	Gesamt-zeit	Prozessor-zeit	Größe der Streutabellen
Traditioneller Plan	20 s	10.4 s	21.6 MB
Aggregation vor Verbund	20 s	6.4 s	16.3 MB
$\Gamma^{\text{add-in}}$	20 s	5.4 s	10.7 MB

Tabelle 5.9: Ausführungszeiten für den $\Gamma^{\text{add-in}}$-Operator

Speicher beansprucht als der traditionelle Plan, ergibt sich daraus, daß zwischen Aggregation und Verbund eine Projektion durchgeführt werden konnte, die die Stelligkeit der zu verarbeitenden Tupel reduziert hat.

5.7 Optimierung

In diesem Abschnitt beschreiben wir, welche Auswirkungen die Integration unserer neuen Operatoren auf einen Anfrageoptimierer hat.

5.7.1 Optimierungsprozeß

Wie schon in Abschnitt 5.1 erwähnt, gehen wir davon aus, daß die algebraische und die physische Optimierung nicht voneinander getrennt werden können. Unser Optimierungsprozeß umfaßt also beide Optimierungsarten.

Der Prozeß besteht aus drei Phasen:

1. erste Anfragetransformationsphase,
2. Plangenerierungsphase und
3. zweite Anfragetransformationsphase.

In der *ersten Transformationsphase* werden regelbasierte Transformationen der Anfrage durchgeführt. Diese sind im wesentlichen Entschachtelungen und die Auflösung von Sichten. In der *Plangenerierungsphase* werden unterschiedliche, aus den Operatoren der physischen Algebra (vgl. Abschnitt 3.6) konstruierte, Auswertungspläne generiert und anhand eines Kostenmodells der beste Plan ausgewählt. In der *zweiten Transformationsphase* werden weitere regelbasierte Transformationen des generierten Plans, wie zum Beispiel das Verschieben des Gruppierungsoperators von einen Verbund (vgl. [YL94, YL95, CS94, GHQ95]), durchgeführt.

5.7.2 Diagonalverbund

Zur Integration des Diagonalverbundes muß nur die Plangenerierungsphase angepaßt werden. Hier kann der Diagonalverbund – wie alle anderen Implementierungen des Verbunds – in den generierten Plänen verwendet werden. Natürlich müssen dazu überprüft werden, ob an der entsprechenden Stelle des Auswertungsplans die Voraussetzungen für die Verwendung des Diagonalverbunds erfüllt sind.

5.7.3 Max-Operator und $\Gamma^{\mathbf{max}}$-Operator

Die Verwendung des Max-Operators führt für Anfragen, die dem globalen Extremum-Muster entsprechen, niemals zu einer Verschlechterung der Ausführungszeit. Daher kann er schon – zusammen mit anderen Entschachtelungen – in der ersten Transformationsphase in den Auswertungsplan eingeführt werden. Die Integration dieses Operators führt daher nicht zu einer Vergrößerung des vom Anfrageoptimierer betrachteten Suchraums.

Das gleiche trifft für den Γ^{max}-Operator für Anfragen, die dem lokalen Extremum-Muster entsprechen, zu. Auch hier kann eine Integration in der ersten Transformationsphase ohne eine Vergrößerung des Suchraums erfolgen.

5.7.4 $\Gamma^{\mathbf{add\text{-}in}}$-Operator

Der $\Gamma^{add\text{-}in}$-Operator kann als eine verbesserte Implementierung der Kombination aus einer verschobenen Gruppierung mit einem anschließenden Verbund betrachtet werden. Daher kann diese Kombination in der zweiten Transformationsphase durch den $\Gamma^{add\text{-}in}$-Operator ersetzt werden. Da diese Ersetzung stattfindet, *nachdem* in der Plangenerierungsphase der optimale Plan bestimmt wurde, können wir sicher sein, daß die Einführung des $\Gamma^{add\text{-}in}$-Operators die Ausführungszeit nicht erhöht. Des weiteren können wir feststellen, daß auch die Einführung des $\Gamma^{add\text{-}in}$-Operators den Suchraum nicht vergrößert.

5.8 Ergebnis

Wir haben das Konzept der semantischen Anfragemuster beschrieben und gezeigt, wie diese zur Anfrageverarbeitung genutzt werden können. Wir haben vier Muster der analytischen Anfrageverarbeitung identifiziert und Operatoren, die diese Muster unterstützen, beschrieben. Außerdem haben wir für die Operatoren Implementierungen angegeben. Wir haben die Überlegenheit dieser Implementierungen gegenüber den bisher verwendeten Implementierungen begründet und experimentell in AODB nachgewiesen. Des weiteren haben wir dargestellt, wie ein Anfrageoptimierer erweitert werden muß, um diese Operatoren zu nutzen.

Das zentrale Ergebnis ist allerdings, daß Effizienzsteigerungen bei der Anfrageauswertung nicht nur durch neue Implementierungen bekannter Operatoren sondern auch durch Erweiterungen der Algebra möglich sind. Dieser Ansatz eröffnet ein neues Optimierungspotential. Wir hoffen daher, daß in Zukunft weitere Muster gefunden werden, die speziell genug für eine effizientere Implementierung und allgemein genug für die Integration in ein Datenbankverwaltungssystem sind.

Kapitel 6

Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war die Konstruktion eines effizienten Laufzeitsystems für Datenlager. Dazu haben wir zunächst in Kapitel 2 die Charakteristiken von Datenlagern und die sich daraus ergebenden Anforderungen an ein Datenlagerverwaltungssystem dargestellt.

In Kapitel 3 haben wir dann unser Laufzeitsystem AODB vorgestellt, das die notwendigen Voraussetzungen für die Konstruktion eines Laufzeitsystems für Datenlager mitbringt. Diese sind Flexibilität und sparsamer Umgang mit Ressourcen (insbesondere mit der Prozessorzeit). Ein großer Schritt zur Erreichung beider Eigenschaften war dabei die Verwendung der virtuellen Maschine AVM zur Verarbeitung der gesamten typisierten Information im System.

Anschließend haben wir in den Kapiteln 4 und 5 Techniken vorgestellt, die die Anfragebearbeitung in Datenlagern in besonderer Weise unterstützen. In Kapitel 4 haben wir uns mit der Nutzung von Kompression zur Beschleunigung des Lesens großer Datenmengen beschäftigt. Dabei haben wir festgestellt, daß eine Beschleunigung nur dann möglich ist, wenn es gelingt, die Prozessorkosten im Griff zu behalten. Wir haben daher Techniken entwickelt, die die im Zusammenhang mit der Integration von Kompression entstehenden Prozessorkosten reduzieren.

In Kapitel 5 haben wir uns semantischen Anfragemustern gewidmet. Wir haben dort einige Muster, die in Datenlagern auftreten, identifiziert und spezielle Operatoren entwickelt, die diese Muster ausnutzen und so eine effizientere Anfrageverarbeitung ermöglichen.

Wir haben alle in den Kapiteln 4 und 5 beschriebenen Techniken in AODB implementiert und experimentell in Anlehnung an den TPC-D-Benchmark evaluiert. Bei den Experimenten konnten wir erhebliche Leistungssteigerungen feststellen: Sowohl die Verwendung von Kompression als auch die Ausnutzung der Anfragemuster konnten – sofern sie anwendbar waren – die Antwortzeiten halbieren.

Eine spezifische Eigenschaft von Datenlagern, die wir in Zukunft untersuchen wollen, ist die Mehrbenutzersynchronisation. Es stellt sich die Frage, wie man den lesenden Zugriff mehrerer Benutzer so synchronisiert, daß gemeinsam genutzte Daten nur einmal in den Primärspeicher

transferiert werden müssen. Durch die Verwendung eines Systempuffers kann es zwar vorkommen, daß dies zufällig passiert, allerdings ist ein systematischeres Vorgehen hier wünschenswert.

Des weiteren wollen wir überprüfen, ob die beschriebenen Techniken für den Einsatz in parallelen und verteilten Datenbanksystemen geeignet sind, und sie – falls das nicht der Fall sein sollte – für einen solchen Einsatz anpassen.

Literaturverzeichnis

[ALM96] ANTOSHENKOV, GENNADY, DAVID B. LOMET und JAMES MURRAY: *Order Preserving Compression*. In: *ICDE '96* [ICD96], Seiten 655–663.

[BCE76] BLASGEN, MIKE W., RICHARD G. CASEY und KAPALI P. ESWARAN: *An Encoding Method for Multifield Sorting and Indexing*. Technical Report RJ 1753, IBM Research, San Jose, CA, USA, 1976.

[BE76] BLASGEN, MIKE W. und KAPALI P. ESWARAN: *On the Evaluation of Queries in a Relational Database System*. Technical Report RJ 1745, IBM Research, San Jose, CA, USA, 1976.

[BJK+97] BRIDGE, WILLIAM, ASHOK JOSHI, M. KEIHL, TIRTHANKAR LAHIRI, JUAN LOAIZA und N. MACNAUGHTON: *The Oracle Universal Server Buffer Manager*. In: *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, Seiten 590–594, Athen, Griechenland, August 1997.

[Bra84] BRATBERGSENGEN, KJELL: *Hashing Methods and Relational Algebra Operations*. In: *Proceedings of the Tenth International Conference on Very Large Data Bases (VLDB)*, Seiten 323–333, Singapur, Singapur, August 1984.

[CAB+81] CHAMBERLIN, DONALD D., MORTON M. ASTRAHAN, MIKE W. BLASGEN, JIM GRAY, W. FRANK KING, BRUCE G. LINDSAY, RAYMOND A. LORIE, JAMES W. MEHL, THOMAS G. PRICE, GIANFRANCO R. PUTZOLU, PATRICIA G. SELINGER, MARIO SCHKOLNICK, DONALD R. SLUTZ, IRVING L. TRAIGER, BRADFORD W. WADE und ROBERT A. YOST: *A History and Evaluation of System R*. Communications of the ACM, 24(10):632–646, Oktober 1981.

[CD85] CHOU, HONG-TAI und DAVID J. DEWITT: *An Evaluation of Buffer Management Strategies for Relational Database Systems*. In: *VLDB '85* [VLD85], Seiten 127–141.

[CG85] CERI, STEFANO und GEORG GOTTLOB: *Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries*. IEEE Transactions on Software Engineering (TSE), 11(5):324–344, April 1985.

[CM93] CLUET, SOPHIE und GUIDO MOERKOTTE: *Nested Queries in Object Bases*. In: *Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*, Seiten 226–242, New York City, NY, USA, 1993.

[Com79] COMER, DOUGLAS: *The Ubiquitous B-tree*. ACM Computing Surveys, 11(2):121–137, Juni 1979.

[Cor85] CORMACK, GORDON V.: *Data Compression in a Database System*. Communications of the ACM, 28(12):1336–1342, Dezember 1985.

[CR96] CHATZIANTONIOU, DAMIANOS und KENNETH A. ROSS: *Querying Multiple Features of Groups in Relational Databases*. In: *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, Seiten 295–306, Bombay, Indien, September 1996.

[CS94] CHAUDHURI, SURAJIT und KYUSEOK SHIM: *Including Group-By in Query Optimization*. In: *VLDB '94* [VLD94], Seiten 354–366.

[Day87] DAYAL, UMESHWAR: *Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers*. In: *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, Seiten 197–208, Brighton, England, September 1987.

[DG85] DEWITT, DAVID J. und ROBERT H. GERBER: *Multiprocessor Hash-Based Join Algorithms*. In: *VLDB '85* [VLD85], Seiten 151–164.

[DKO+84] DEWITT, DAVID J., RANDY H. KATZ, FRANK OLKEN, LEONARD D. SHAPIRO, MICHAEL STONEBRAKER und DAVID A. WOOD: *Implementation Techniques for Main Memory Database Systems*. In: *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seiten 1–8, Boston, MA, USA, Juni 1984.

[DNS91] DEWITT, DAVID J., JEFFREY F. NAUGHTON und DONOVAN A. SCHNEIDER: *Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting*. In: *Proceedings of the International Conference on Parallel and Distributed Information Systems*, Seiten 280–291, Miami Beach, FL, USA, Dezember 1991.

[Eps79] EPSTEIN, ROBERT: *Techniques for Processing of Aggregates in Relational Database Systems*. UCB/ERL Memorandum M79/8, Univ. of California at Berkeley, Februar 1979.

[FKT86] FUSHIMI, SHINYA, MASARU KITSUREGAWA und HIDEHIKO TANAKA: *An Overview of The System Software of A Parallel Relational Database Machine GRACE*. In: *VLDB '86* [VLD86], Seiten 209–219.

[GBC98] GRAEFE, GOETZ, ROSS BUNKER und SHAUN COOPER: *Hash Joins and Hash Teams in Microsoft SQL Server*. In: *VLDB '98* [VLD98], Seiten 86–97.

[GBLP96] GRAY, JIM, ADAM BOSWORTH, ANDREW LAYMAN und HAMID PIRAHESH: *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total*. In: *ICDE '96* [ICD96], Seiten 152–159.

[GHJV95] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[GHQ95] GUPTA, ASHISH, VENKY HARINARAYAN und DALLAN QUASS: *Aggregate-Query Processing in Data Warehousing Environments*. In: *VLDB '95* [VLD95], Seiten 358–369.

[GLS94] GRAEFE, GOETZ, ANN LINVILLE und LEONARD D. SHAPIRO: *Sort versus Hash Revisited*. IEEE Trans. on Data and Knowledge Eng., 6(6):934–944, Dezember 1994.

[GM99] GUPTA, ASHISH und INDERPAL SINGH MUMICK (Herausgeber): *Materialized Views : Techniques, Implementations, and Applications*. The MIT Press, Cambridge, MA, USA, Juni 1999.

[GR93] GRAY, JIM und ANDREAS REUTER: *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1993.

[Gra93] GRAEFE, GOETZ: *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys, 25(2):73–170, Juni 1993.

[Gra94] GRAEFE, GOETZ: *Sort-Merge-Join: An Idea Whose Time Has(h) Passed?* In: *ICDE '94* [ICD94], Seiten 406–417.

[GRS98] GOLDSTEIN, JONATHAN, RAGHU RAMAKRISHNAN und URI SHAFT: *Compressing Relations and Indexes*. In: *Proceedings IEEE Conference on Data Engineering*, Seiten 370–379, Orlando, FL, USA, 1998.

[GS91] GRAEFE, GOETZ und LEONARD D. SHAPIRO: *Data Compression and Database Performance*. In: *Proc. ACM/IEEE-CS Symp. on Applied Computing*, Kansas City, MO, Apr 1991.

[Här78] HÄRDER, THEO: *Implementing a Generalized Access Path Structure for a Relational Database System*. ACM Transactions on Database Systems, 3(3):285–298, September 1978.

[HCLS97] HAAS, LAURA M., MICHAEL J. CAREY, MIRON LIVNY und AMIT SHUKLA: *Seeking the Truth About ad hoc Join Costs*. The VLDB Journal, 6(3):241–256, Juli 1997.

[HR96] HARRIS, EVAN P. und KOTAGIRI RAMAMOHANARAO: *Join Algorithm Costs Revisited*. The VLDB Journal, 5(1):64–84, Januar 1996.

[HR99] HÄRDER, THEO und ERHARD RAHM: *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer-Verlag, Berlin, Heidelberg, New York, 1999.

[HRU96] HARINARAYAN, VENKY, ANAND RAJARAMAN und JEFFREY D. ULLMAN: *Implementing Data Cubes Efficiently*. In: *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seiten 205–216, Montreal, Kanada, Juni 1996.

[Huf52] HUFFMAN, DAVID A.: *A Method for the Construction of Minimum-redundancy codes*. Proc. IRE, 40(9):1098–1101, September 1952.

[HWM98] HELMER, SVEN, TILL WESTMANN und GUIDO MOERKOTTE: *Diag-Join: An opportunistic Join Algorithm for 1:N Relationships*. In: *VLDB '98* [VLD98], Seiten 98–109.

[ICD94] *Proceedings IEEE Conference on Data Engineering*, Houston, TX, USA, 1994.

[ICD96] *Proceedings IEEE Conference on Data Engineering*, New Orleans, LA, USA, 1996.

[Inm96] INMON, W. H.: *Building the Data Warehouse (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[ISO97] ISO, INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Information technology – Database languages – SQL*. ISO/IEC 9075:1992, September 1997. http://www.iso.ch/.

[IW94] IYER, BALAKRISHNA R. und DAVID WILHITE: *Data Compression Support in Databases*. In: *VLDB '94* [VLD94], Seiten 695–704.

[Kim96] KIMBALL, RALPH: *The Data Warehouse Toolkit*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[KKD89] KIM, W., K. C. KIM, und A. DALE: *Indexing techniques for object-oriented databases*. In: KIM, W. und F. H. LOCHOVSKY (Herausgeber): *Object-Oriented Concepts, Databases, and Applications*, Seiten 371–394. Addison-Wesley, Reading, MA, USA, 1989.

[Klu82] KLUG, ANTHONY: *Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions*. Journal of the ACM, 29(3):699–717, Juli 1982.

[KM94] KILGER, CHRISTOPH und GUIDO MOERKOTTE: *Indexing Multiple Sets*. In: *VLDB '94* [VLD94], Seiten 180–191.

[KNT89] KITSUREGAWA, MASARU, MASAYA NAKAYAMA und MIKIO TAKAGI: *The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method*. In: *VLDB '89* [VLD89], Seiten 257–266.

[KR96] KAMATH, MOHAN und KRITHI RAMAMRITHAM: *Bucket Skip Merge Join: A scalable Algorithm for Join Processing in Very Large Databases using Indexes*. Technical Report CS-TR-96-20, University of Massachusetts, 1996.

[LW79] LORIE, RAYMOND A. und BRADFORD W. WADE: *The Compilation of a High Level Data Language*. Technical Report RJ 2598, IBM Research, San Jose, CA, 1979.

[LY89] LORIE, RAYMOND A. und HONESTY C. YOUNG: *A Low Communication Sort Algorithm for a Parallel Database Machine*. In: *VLDB '89* [VLD89], Seiten 125–134. auch veröffentlicht als: IBM Technical Report RJ 6669, Februar 1989.

[Mai83] MAIER, DAVID: *The Theory of Relational Databases*. Computer Science Press, Rockville, MD, USA, 1983.

[ME92] MISHRA, PRITI und MARGARET H. EICH: *Join Processing in Relational Databases*. ACM Computing Surveys, 24(1):63–113, März 1992.

[Men86] MENON, JAI: *A Study of Sort Algorithms for Multiprocessor Database Machines*. In: *VLDB '86* [VLD86], Seiten 197–206.

[MGST70] MATTSON, RICHARD L., JAN GECSEI, DONALD R. SLUTZ und IRVING L. TRAIGER: *Evaluation Techniques for Storage Hierarchies*. IBM Systems Journal, 9(2):78–117, 1970.

[Moe98] MOERKOTTE, GUIDO: *Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing*. In: *VLDB '98* [VLD98], Seiten 476–487.

[MZ92] MOFFAT, ALISTAIR und JUSTIN ZOBEL: *Parameterised Compression for Sparse Bitmaps*. In: *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seiten 274–285, Kopenhagen, Dänemark, Juni 1992.

[NKT88] NAKAYAMA, MASAYA, MASARU KITSUREGAWA und MIKIO TAKAGI: *Hash-Partitioned Join Method Using Dynamic Destaging Strategy*. In: *Proceedings of the Fourteenth International Conference on Very Large Data Bases (VLDB)*, Seiten 468–478, Los Angeles, California, USA, August 1988.

[NR95] NG, WEE KEONG und CHINYA V. RAVISHANKAR: *Relational Database Compression Using Augmented Vector Quantization*. In: *Proceedings IEEE Conference on Data Engineering*, Seiten 540–549, Taipeh, Taiwan, 1995.

[OG95] O'NEIL, PATRICK und GOETZ GRAEFE: *Multi-Table Joins Through Bitmapped Join Indices*. ACM SIGMOD Record, 24(3):8–11, Oktober 1995.

[Pal74] PALERMO, FRANK P.: *A Database Search Problem*. In: *Information Systems*, Seiten 67–101. Plenum Publ., New York, NY, USA, 1974.

[RG99] RAMAKRISHNAN, RAGHU und JOHANNES GEHRKE: *Database Management Systems*. McGraw-Hill, Inc., New York, San Francisco, Washington, D.C., USA, August 1999.

[RH93] ROTH, MARK A. und SCOTT J. VAN HORN: *Database Compression*. ACM SIGMOD Record, 22(3):31–39, September 1993.

[RHS95] RAY, GAUTAM, JAYANT R. HARITSA und S. SESHADRI: *Database Compression: A Performance Enhancement Tool*. In: *Proceedings of the International Conference on Management of Data (COMAD)*, Pune, Indien, Dezember 1995.

[RJB99] RUMBAUGH, JAMES, IVAR JACOBSON und GRADY BOOCH: *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, USA, 1999.

[SAG94] SOFTWARE AG, München: *ADABAS im Überblick*, Januar 1994.

[SB98] STONEBRAKER, MICHAEL und PAUL BROWN: *Object-Relational DBMSs*. Morgan-Kaufmann Publishers, San Mateo, CA, USA, 2. Auflage, September 1998.

[Sch98] SCHÖNING, HARALD: *The ADABAS Buffer Pool Manager*. In: *VLDB '98* [VLD98], Seiten 675–679.

[SD90] SCHNEIDER, DONOVAN A. und DAVID J. DEWITT: *Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines*. In: *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*, Seiten 469–480, Brisbane, Queensland, Australien, August 1990.

[Sev83] SEVERANCE, DENNIS G.: *A practitioner's guide to data base compression*. Information Systems, 8(1):51–62, 1983.

[SG98] SILBERSCHATZ, ABRAHAM und PETER BAER GALVIN: *Operating System Concepts*. Addison-Wesley, Reading, MA, USA, 5. Auflage, 1998.

[Sha86] SHAPIRO, LEONARD D.: *Join Processing in Database Systems with Large Main Memories*. ACM Transactions on Database Systems, 11(3):239–264, September 1986.

[SM94] SHIN, DONG KEUN und ARNOLD CHARLES MELTZER: *A New Join Algorithm*. ACM SIGMOD Record, 23(4):13–18, Dezember 1994.

[SN95] SHATDAL, AMBUJ und JEFFREY F. NAUGHTON: *Adaptive Parallel Aggregation Algorithms*. In: *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seiten 104–114, San Jose, CA, USA, Juni 1995.

[SNG93] SHAPIRO, LEONARD D., SHENGSONG NI und GOETZ GRAEFE: *Full-Time Data Compression: An ADT for Database Performance*. Technical Report, Portland State University, OR, USA, 1993.

[SS86] SACCO, GIOVANNI MARIA und MARIO SCHKOLNICK: *Buffer Management in Relational Database Systems*. ACM Transactions on Database Systems, 11(4):473–498, Dezember 1986.

[STG+90] SALZBERG, BETTY, ALEX TSUKERMAN, JIM GRAY, MICHAEL STEWART, SUSAN UREN und BONNIE VAUGHAN: *FastSort: A Distributed Single-Input Single-Output External Sort*. In: *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seiten 94–101, Atlantic City, NJ, USA, Juni 1990.

[Sto81] STONEBRAKER, MICHAEL: *Operating System Support for Database Management*. Communications of the ACM, 24(7):412–418, Juli 1981.

[TG84] TENG, J. Z. und R. A. GUMAER: *Managing IBM Database 2 buffers to maximize performance*. IBM Systems Journal, 23(2):211–218, 1984.

[TPC95] TPC, TRANSACTION PROCESSING PERFORMANCE COUNCIL: *TPC Benchmark D (Decision Support)*. Standard Specification 1.0, Transaction Processing Performance Council (TPC), Mai 1995. http://www.tpc.org/.

[Val87] VALDURIEZ, PATRICK: *Join Indices*. ACM Transactions on Database Systems, 12(2):218–246, Juni 1987.

[VLD85] *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB)*, Stockholm, Schweden, August 1985.

[VLD86] *Proceedings of the Twelfth International Conference on Very Large Data Bases (VLDB)*, Kyoto, Japan, August 1986.

[VLD89] *Proceedings of the Fifteenth International Conference on Very Large Data Bases (VLDB)*, Amsterdam, Niederlande, August 1989.

[VLD94] *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, Santiago, Chile, September 1994.

[VLD95] *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB)*, Zürich, Schweiz, September 1995.

[VLD98] *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB)*, New York, NY, USA, August 1998.

[Wag73] WAGNER, ROBERT E.: *Indexing Design Considerations*. IBM Systems Journal, 12(4):351–367, 1973.

[Wel84] WELCH, TERRY A.: *A Technique for High Performance Data Compression*. IEEE Computer, 17(6):8–19, Juni 1984.

[WKHM00] WESTMANN, TILL, DONALD KOSSMANN, SVEN HELMER und GUIDO MOERKOTTE: *The Implementation and Performance of Compressed Databases*. ACM SIGMOD Record, 29(3), September 2000.

[WM99] WESTMANN, TILL und GUIDO MOERKOTTE: *Variations on Grouping and Aggregation*. Reihe Informatik 11/1999, Universität Mannheim, 1999.

[WNC87] WITTEN, IAN H., RADFORD M. NEAL und JOHN G. CLEARY: *Arithmetic Coding for Data Compression*. Communications of the ACM, 30(6):520–540, Juni 1987.

[XH94] XIE, ZHAOHUI und JIAWEI HAN: *Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases*. In: *VLDB '94* [VLD94], Seiten 522–533.

[YL94] YAN, WEIPENG P. und PER-ÅKE LARSON: *Performing Group-By Before Join*. In: *ICDE '94* [ICD94], Seiten 89–100.

[YL95] YAN, WEIPENG P. und PER-ÅKE LARSON: *Eager Aggregation and Lazy Aggregation*. In: *VLDB '95* [VLD95], Seiten 345–357.

Anhang A

Auswertungspläne für den TPC-D-Benchmark

Dieser Anhang enthält drei der siebzehn Auswertungspläne, die für den in Abschnitt 4.6 beschiebenen TPC-D-Benchmark verwendet wurden. Da die Darstellung aller siebzehn Pläne etwa 90 Seiten erfordert, haben wir drei (einfachere) Pläne ausgewählt.

A.1 Query 1

SQL-Anfrage

```
SELECT
      L_RETURNFLAG, L_LINESTATUS, SUM(L_QUANTITY) AS SUM_QTY,
      SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE,
      SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS SUM_DISC_PRICE,
      SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE,
      AVG(L_QUANTITY) AS AVG_QTY, AVG(L_EXTENDEDPRICE) AS AVG_PRICE,
      AVG(L_DISCOUNT) AS AVG_DISC, COUNT(*) AS COUNT_ORDER
   FROM LINEITEM
   WHERE L_SHIPDATE <= DATE '1998-12-01' - INTERVAL '90' DAY
   GROUP BY L_RETURNFLAG, L_LINESTATUS
   ORDER BY L_RETURNFLAG, L_LINESTATUS
```

Auswertungsplan

```
[ 10
  NO_AUX_REGS 1
  CREATE_PARTITION_OBJECT
```

```
{
  filepartition
  'TPCD'
  '/export/lab1/TPC-D/segment_sf1/TPCD.part'
  4096
  true // do not use OS buffers
} 0
MOUNT_PARTITION 0
EVAL 22 4
( print
  ( sgroup
    ( tbscanb
      'TPCD'
      'lineitem'
      { ntuple '/export/lab1/TPC-D/schema/lineitem.nschema' }
      0    // operator-register (aux)
      1    // selection-register (aux)
      32   // prefetch size
      [ 20
        LOAD_PTR      0              1
        LOAD_SC1_C    8           1  2 // L_RETURNFLAG
        LOAD_SC1_C    9           1  3 // L_LINESTATUS
        LOAD_DAT_C   10           1  4 // L_SHIPDATE
        LEQ_DAT_ZC_C 4 '1998-09-02'  1
        AVM_STOP
      ]
    )
    10 // htsize
    22 // noRegs
    2  // result reg of cmp-prg (aux)
    3  // result reg of sort-prg (aux)
    [ 100 // init
      MV_UI4_C_C    1              0    // COUNT(*) = 0
      LOAD_SF8_C    4           1  6    // L_QUANTITY
      LOAD_SF8_C    5           1  7    // L_EXTENDEDPRICE
      LOAD_SF8_C    6           1  8    // L_DISCOUNT
      LOAD_SF8_C    7           1  9    // L_TAX
      MV_SF8_Z_C    6             10    // SUM/AVG(L_QUANTITY)
      MV_SF8_Z_C    7             11    // SUM/AVG(L_EXTENDEDPRICE)
      MV_SF8_Z_C    8             12    // AVG(L_DISCOUNT)
      SUB_SF8_CZ_C  1.0         8 13    // 1 - L_DISCOUNT
      ADD_SF8_CZ_C  1.0         9 14    // 1 + L_TAX
      MUL_SF8_ZZ_C  7          13 15    // SUM(L_EXTDPRICE * (1 - L_DISC))
      MUL_SF8_ZZ_C 15          14 16    // SUM((...) * (1 + L_TAX))
      AVM_STOP
    ]
    [ 100 // advance
      INC_UI4       0                  // inc COUNT(*)
      MV_PTR_Y      1              1
```

```
    LOAD_SF8_C    4           1  6    // L_QUANTITY
    LOAD_SF8_C    5           1  7    // L_EXTENDEDPRICE
    LOAD_SF8_C    6           1  8    // L_DISCOUNT
    LOAD_SF8_C    7           1  9    // L_TAX
    MV_SF8_Z_A    6             10    // SUM/AVG(L_QUANTITY)
    MV_SF8_Z_A    7             11    // SUM/AVG(L_EXTENDEDPRICE)
    MV_SF8_Z_A    8             12    // AVG(L_DISCOUNT)
    SUB_SF8_CZ_C  1.0         8 13    // 1 - L_DISCOUNT
    ADD_SF8_CZ_C  1.0         9 14    // 1 + L_TAX
    MUL_SF8_ZZ_B  7          13 17 15 // SUM(L_EXTDPRICE * (1 - L_DISC))
    MUL_SF8_ZZ_A 17          14 16    // SUM((...) * (1 + L_TAX))
    AVM_STOP
  ]
  [ 100 // finalize
    UIFC_C        0             18
    DIV_SF8_ZZ_C  10         18 19    // AVG(L_QUANTITY)
    DIV_SF8_ZZ_C  11         18 20    // AVG(L_EXTENDEDPRICE)
    DIV_SF8_ZZ_C  12         18 21    // AVG(L_DISCOUNT)
    AVM_STOP
  ]
  [ 100 // hash
    HASH_SC1      2
    HASH_SC1      3
    AVM_STOP
  ]
  [ 100 // cmp
    CMPA_SC1_ZY_C 2          2  2
    EXIT_NEQ      2
    CMPA_SC1_ZY_C 3          3  2
    AVM_STOP
  ]
  [ 100 // copy
    MV_PTR_Y      1             1
    MV_SC1_Y_C    2             2
    MV_SC1_Y_C    3             3
    AVM_STOP
  ]
  [ 100 // sort
    CMPA_SC1_ZY_C 2          2  3
    EXIT_NEQ      3
    CMPA_SC1_ZY_C 3          3  3
    AVM_STOP
  ]
)
[ 100
  PRINT_STR_C     'L_RETURNFLAG'    13
  PRINT_STR_C     'L_LINESTATUS'    13
  PRINT_STR_C     'SUM_QTY'         12
  PRINT_STR_C     'SUM_BASEPRICE'   14
```

```
      PRINT_STR_C     'SUM_DISC_PRICE'  15
      PRINT_STR_C     'SUM_CHARGE'      12
      PRINT_STR_C     'AVG_QTY'         8
      PRINT_STR_C     'AVG_PRICE'       10
      PRINT_STR_C     'AVG_DISC'        10
      PRINT_STR_C     'COUNT_ORDER'     12
      AVM_STOP
    ]
    [ 100
      PRINT_SC1_Z     2          13
      PRINT_SC1_Z     3          13
      PRINT_SF8_Z     10         12
      PRINT_SF8_Z     11         14
      PRINT_SF8_Z     15         15
      PRINT_SF8_Z     16         12
      PRINT_SF8_Z     19         8
      PRINT_SF8_Z     20         10
      PRINT_SF8_Z     21         10
      PRINT_UI4_Z     0          12
      AVM_STOP
    ]
  )
  AVM_STOP
]
```

A.2 Query 4

SQL-Anfrage

```
SELECT
      O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
   FROM ORDER
   WHERE O_ORDERDATE >= DATE '1993-07-01'
      AND O_ORDERDATE < DATE '1993-07-01' + INTERVAL '3' MONTH
      AND EXISTS (SELECT *
                  FROM LINEITEM
                  WHERE L_ORDERKEY = O_ORDERKEY
                     AND L_COMMITDATE < L_RECEIPTDATE
                  )
   GROUP BY O_ORDERPRIORITY
   ORDER BY O_ORDERPRIORITY;
```

Auswertungsplan

```
[ 10
  NO_AUX_REGS 1
  CREATE_PARTITION_OBJECT
  {
    filepartition
    'TPCD'
    '/export/lab1/TPC-D/segment_sf1/TPCD.part'
    4096
    false // do not use OS buffers
  } 0
  MOUNT_PARTITION 0
  EVAL 18 7
  ( print
    ( sgroup
      ( semijoin
        ( tbscanb
          'TPCD'
          'lineitem'
          { ntuple '/export/lab1/TPC-D/schema/lineitem.nschema' }
          0    // operator-register (aux)
          1    // selection-register (aux)
          16   // prefetch size
          [ 20
            LOAD_PTR        0              0
            LOAD_DAT_C     11          0   1  // L_COMMITDATE
            LOAD_DAT_C     12          0   2  // L_RECEIPTDATE
            LT_DAT_ZZ_C     1          2   1  // Vergleichen des L_SHIPDATE
            EXIT_F          1
```

```
      LOAD_SI4_C      0           0   1   // L_ORDERKEY
      AVM_STOP
    ]
  )
  ( tbscanb
    'TPCD'
    'order'
    { ntuple '/export/lab1/TPC-D/schema/order.nschema' }
    2    // operator-register (aux)
    3    // selection-register (aux)
    16   // prefetch size
    [ 20
      LOAD_PTR      2                0
      LOAD_DAT_C    4            0   1  // O_ORDERDATE
      LEQ_DAT_CZ_C  '1993-07-01' 1   3  // Vergleichen des O_ORDERDATE
      EXIT_F        3
      LT_DAT_ZC_C   1 '1993-10-01'   3  // Vergleichen des O_ORDERDATE
      EXIT_F        3
      LOAD_SI4_C    0            0   1  // O_ORDERKEY
      LOAD_STB_C    5            0   2  // O_ORDERPRIORITY
      AVM_STOP
    ]
  )
  1       // # Join Attr.
  100001  // Hashtablesize
  3       // # Register
  4       // result-register (aux)
  [ 10  // Hash Prog
    HASH_SI4       1                 // hash auf ORDERKEY
    AVM_STOP
  ]
  [ 10  // Comp Prog
    CMPA_SI4_ZY_C 0          1  4     // Vergleich von ORDERKEY
    AVM_STOP
  ]
  [ 10   // Copy Prog
    MV_SI4_Y_C   1     0
    AVM_STOP
  ]
)  // Ende des Joins
5000 // Hashtablesize
2    // # Regs
5    // result-register (aux)
6    // sort-register (aux)
[ 10  // Init Prog
  MV_UI4_C_C  1      1
  AVM_STOP
]
[ 10  // Advance Prog
```

```
        INC_UI4 1
        AVM_STOP
      ]
      [ 10   // Finalize Prog
        AVM_STOP
      ]
      [ 10  // Hash Prog
        HASH_STR    2
        AVM_STOP
      ]
      [ 10  // Cmp Prog
        CMPA_STR_ZZ_C 2          2  5 // L_ORDRKEY
        AVM_STOP
      ]
      [ 10  // Copy Prog
        MV_STB_Y_C    2          2
        AVM_STOP
      ]
      [ 10  // Sort Prog
        CMPA_STR_ZY_C  2          2  6
        AVM_STOP
      ]
    ) // Ende der Gruppierung
    [ 20
      PRINT_STR_C     'O_ORDERPRIORITY '         16
      PRINT_STR_C     'ORDER_COUNT     '         16
      AVM_STOP
    ]
    [ 20
      PRINT_STR_Z     2          16  // O_ORDERPRIORITY
      PRINT_UI4_Z     1          16  // ORDER_COUNT
      AVM_STOP
    ]
  ) // print
  AVM_STOP
]
```

A.3 Query 17

SQL-Anfrage

```
SELECT
      SUM(L_EXTENDEDPRICE)/7.0 AS AVG_YEARLY
   FROM LINEITEM, PART
   WHERE P_PARTKEY = L_PARTKEY
      AND P_BRAND = 'Brand#23'
      AND P_CONTAINER = 'MED BOX'
      AND L_QUANTITY < (SELECT
                           0.2 * AVG(L_QUANTITY)
                        FROM LINEITEM L1
                        WHERE L_PARTKEY = P_PARTKEY
                        );
```

Auswertungsplan

```
[ 20
  NO_AUX_REGS 2
  CREATE_PARTITION_OBJECT
  {
    filepartition
    'TPCD'
    '/export/lab1/TPC-D/segment_sf1/TPCD.part'
    4096
    false // do not use OS buffers
  } 0
  MOUNT_PARTITION 0
  CREATE_PARTITION_OBJECT
  {
    filepartition
    'TMP'
    '/export/lab1/TPC-D/tmp.part'
    4096
    false // do not use OS buffers
  } 1
  GROW_PARTITION   1 10000
  FORMAT_PARTITION 1 'TMP'
  MOUNT_PARTITION  1
  CREATE_SEGMENT { spsegment 'aTmpSeg' 'TMP' true 16 16 }
  EVAL 4 4
  ( tempb
    ( bnljoin
      ( tbscanb
        'TPCD'
        'part'
```

```
  { ntuple '/export/lab1/TPC-D/schema/part.nschema' }
  0  // operator-register (aux)
  1  // selection-register (aux)
  16 // prefetch-size
  [ 20
    LOAD_PTR    0                   0
    LOAD_STR_C  3            0      1
    EQ_STR_ZC_C 1     'Brand#23'    1
    EXIT_F      1
    LOAD_STR_C  6            0      1
    EQ_STR_ZC_C 1     'MED BOX'     1
    EXIT_F      1
    LOAD_SI4_C  0            0      1  // P_PARTKEY
    AVM_STOP
  ]
)
( pscanb
  'TPCD'
  'lineitem'
  { ntuple '/export/lab1/TPC-D/schema/lineitem.nschema' }
  2  // operator-register (aux)
  16 // prefetch-size
  [ 20
    LOAD_PTR       2               0
    LOAD_SF8_C     5            0   2  // L_EXTENDEDPRICE
    LOAD_SF8_C     4            0   3  // L_QUANTITY
    LOAD_SI4_C     1            0   1  // L_PARTKEY
    AVM_STOP
  ]
)
4096      // Pagesize
2000      // # Pages
5000      // Hashtablesize
4         // # Register
3         // result-register (aux)
[ 20        // Hash Programm
  HASH_SI4       1 //hash auf PARTKEY
  AVM_STOP
]
[ 20        // Comp. Programm
  CMPA_SI4_ZY_C 1          1  3
  AVM_STOP
]
[ 20         // Join Programm
  MV_SF8_Y_C   2          2  // L_EXTENDEDPRICE
  MV_SF8_Y_C   3          3  // L_QUANTITY
  AVM_STOP
]
[ 10          // Copy Prog.
```

```
      MV_SI4_Y_C  1            1  //
      AVM_STOP
    ]
  ) // Ende part x lineitem
  'TMP'
  'aTmpSeg'
  { ntuple 3
    int32_e   // PARTKEY
    float64_e // L_EXTENDEDPRICE
    float64_e // L_QUANTITY
  }
  4 // noRegs
  { matspec  3   1 2 3 }
)
EVAL 6 4
( print
  ( aggr
    ( bnljoin
      ( pscanb
        'TMP'
        'aTmpSeg'
        { ntuple 3
          int32_e   // PARTKEY
          float64_e // L_EXTENDEDPRICE
          float64_e // L_QUANTITY
        }
        0  // operator-register (aux)
        16 // prefetch size
        [ 20
          LOAD_PTR      0               0
          LOAD_SI4_C    0            0  1  // PARTKEY
          LOAD_SF8_C    1            0  2  // L_EXTENDEDPRICE
          LOAD_SF8_C    2            0  3  // L_QUANTITY
          AVM_STOP
        ]
      )
      ( group
        ( pscanb
          'TMP'
          'aTmpSeg'
          { ntuple 3
            int32_e   // PARTKEY
            float64_e // L_EXTENDEDPRICE
            float64_e // L_QUANTITY
          }
          1  // operator-register (aux)
          16 // prefetch size
          [ 20
            LOAD_PTR      1               0
```

```
      LOAD_SI4_C    0            0   1  // PARTKEY
      LOAD_SF8_C    1            0   2  // L_EXTENDEDPRICE
      LOAD_SF8_C    2            0   3  // L_QUANTITY
      AVM_STOP
    ]
  )
  600   // Hashtablesize
  6     // noRegs
  2     // result-register (aux)
  [ 10   // init
    MV_SF8_C_C    0.0            4
    ADD_SF8_ZZ_C  3              4    4
    MV_SI4_C_C    1              5
    AVM_STOP
  ]
  [ 20  // Advance Prog
    MV_SF8_Y_A    3              4
    ADD_SI4_ZC_C  5              1    5
    AVM_STOP
  ]
  [ 20 // Finalize Prog
    SIFC_C        5              5
    DIV_SF8_ZZ_C  4              5    4
    MUL_SF8_ZC_C  4            0.2    3
    AVM_STOP
  ]
  [ 10  // hash
    HASH_SI4    1   // PARTKEY
    AVM_STOP
  ]
  [ 10 // cmp
    CMPA_SI4_ZY_C  1        1  2
    AVM_STOP
  ]
  [ 10 // Copy
    MV_SI4_Y_C     1        1
    MV_SF8_Y_C     2        2
    MV_SF8_Y_C     3        3
    AVM_STOP
  ]
) // Ende Gruppierung
4096    // Pagesize
100     // # Pages
500     // Hashtablesize
6       // # Register
3     // result-register (aux)
[ 20      // Hash Programm
  HASH_SI4       1 //hash auf PARTKEY
  AVM_STOP
```

```
        ]
        [ 20      // Comp. Programm
          CMPA_SI4_ZY_C 1          1  3
          EXIT_NEQ      3
          CMPA_SF8_ZY_C 3          3  3
          EXIT_LT       3
          CMPD_SF8_ZZ_C 3          3  3
          AVM_STOP
        ]
        [ 10   // Join Prog
          MV_SF8_Y_C 3             4     //
          AVM_STOP
        ]
        [ 10          // Copy Prog.
          MV_SI4_Y_C     1         1
          //MV_SF8_Y_C     2         2
          MV_SF8_Y_C     3         3
          AVM_STOP
        ]
      ) // bnljoin
      4         // noRegs
      [ 10   // init
        AVM_STOP
      ]
      [ 10 // advance
        MV_SF8_Y_A   2        2    //
        AVM_STOP
      ]
      [ 10 // finalize
        DIV_SF8_ZC_C   2      7.0   2
        AVM_STOP
      ]
      [ 10  // copy
        MV_SF8_Y_C   2        2    // L_EXTENDEDPRICE
        AVM_STOP
      ]
    )
    [ 10
      PRINT_STR_C     'AVG_YEARLY     '        15
      AVM_STOP
    ]
    [ 10
      PRINT_SF8_Z     2          15
      AVM_STOP
    ]
  ) // print
  DELETE_SEGMENT 'TMP' 'aTmpSeg'
  AVM_STOP
]
```