Simulation objektrelationaler Join-Verfahren in parallelen Datenbanksystemen

Diplomarbeit von Lew Bessonow

August 2000

Universität Leipzig, Fachbereich Informatik
Abteilung DBS, Prof. Dr. Erhard Rahm

Betreuer: Dipl.-Inform. Holger Märtens

Inhaltsverzeichnis:

Abbildungsverzeichnis:

1. Einleitung

Datenbanken ist ein umfassendes Informatikgebiet, das seit einigen Jahrzehnten existiert und immer weiter entwickelt wird. DBS (Datenbanksysteme) gehören zu den wichtigsten Produkten der Software-Industrie; kaum eine größere Informatikanwendung ist ohne Datenbankunterstützung denkbar.

Die möglichen Anwendungen, wo DBS zum Einsatz kommt, sind spezifische Informationssysteme wie Krankenhaus-Informationssysteme, Geographische-Informationssysteme etc. sowie rechnergestützte Entwurfs- und Fertigungssysteme wie CAD/CAM, CASE, CIM etc. Eine starke Zunahme an Datenbanken ergibt sich ferner durch die enorm wachsenden Datenmengen, die im Internet weltweit zugänglich bereitgestellt werden, verbunden mit einer sehr großen Benutzerzahl und entsprechenden Leistungserfordernissen [HR99].

Durch 30-jährige Erfahrung hat man bereits eine Menge von Wissen über den Aufbau und die grundlegenden Konzepte der Datenverarbeitung in den DB-Systemen gesammelt. Der Bedarf an neuen Forschungsarbeiten und guten kommerziellen Systemen im DBS-Bereich besteht immer noch. Die neuen Arten der DB-Systeme werden ins Leben gerufen, neue Standards werden gesetzt. Das erfordert die Entwickelung von neuen Algorithmen und Techniken für Datenverarbeitung, sowie das Sammeln von Erfahrungen und Kenntnissen über Hardware-Komponenten und Konfigurationen, die gemeinsam eine kosteneffektive Lösung für jede Aufgabe, die einem DBS gegenüber gestellt wird, liefern sollen.

Die DBS-Abteilung, Universität Leipzig, wo diese Diplomarbeit geschrieben wird, führt Forschungsarbeit durch, die sich unter anderen auf die Untersuchung der Eigenschaften und besonderen Merkmale einer parallelen DBS-*Shared-Disk (SD)*-Architektur richtet. Diese DBS-Architektur verspricht gewisse Vorteile, ist aber immer noch nicht genug erforscht.

Eine spezifische Erweiterung wird im Rahmen eines umfassenden *SimPaD*-Simulationssystems (*SimPaD* steht für *"Simulations of Parallel Databases"*) für das Testen paralleler ORDB-Systeme erstellt, die sich als eine wichtige Ergänzung zum Gesamtpaket der bereitgestellten Algorithmen und Methoden zu der Untersuchung der *SD*-Architektur betrachten läßt.

Das Erstellen der OR-Erweiterung ist nicht lediglich das einzige Ziel dieser Arbeit, wobei mit ihrer Hilfe die Untersuchungen in verschiedene Richtungen unter den vorgegebenen Rahmenbedingungen möglich wären. Das System, das im Laufe der Arbeit implementiert wird, erlaubt die Ausführung der ganzen Menge von Vergleichen, welche die diversen Verfahren für die OR-Datenallokation sowie ihre Bearbeitungsroutinen und Parameterlisten zwischeneinander testen. Der Vergleich von mehreren komplexen gleichzeitig ablaufenden OR-Anfragen (logischer Mehrbenutzerbetrieb) wird im Rahmen dieses Systems auch möglich und stellt ein typisches Beispiel für ihre Stärke beim Simulieren dar. Die Anzahl der Parameter, womit eine Simulation konfiguriert wird, wird groß genug gewählt, um die tiefgreifenden Prozesse in einem realen ORDBS nachbilden zu können.

Ein großes Teil dieser Arbeit widmet sich deswegen dem Simulationssystem selbst. Im **Kap. 2** werden die wichtigsten theoretischen Grundlagen zum Erstellen der OR-Erweitung des Simulationssystems dargelegt. **Kap. 3** erläutert hauptsächlich die Vorschläge dafür, wie die bisherigen Kenntnisse aus Theorie in unser Modell übertragen werden können. Darüber hinaus werden dort auch einige unserer Ideen zum Thema gestellt und zum Einsatz angeboten. Im **Kap. 4** erklärt man die genaue Funktionsweise einer OR-Simulation und den Aufbau ihrer wichtigsten Module. Die notwendigen Erklärungen zu der Parametrisierung der ganzen Simulation findet man in den **Kap. 3, 4** und **5**.

Als Untersuchungsschwerpunkt dieser Arbeit wird der Vergleich zwischen den verschiedenen Join-Strategien in ORDBS, welche im **Kap. 3** vorgestellt werden, gezogen. Es schien für uns sehr interessant und spannend zu sein, die bestehenden Meinungen über die Effizienz einzelner Join-Verfahren im relationalen Model (R-Model) mit unseren Ergebnissen in der parallelen Umgebung der *SD*-Architektur für ein OR-Schema zu vergleichen. Außerdem konnten zugleich unsere Ansätze für den Allokationsalgorithmus und die Verarbeitungsmethoden getestet werden. Die aus diesen Vergleichen entstehenden Ergebnisse werden im **Kap. 5** präsentiert**.** Das **Kap. 6** bietet dem Leser eine kurze Zusammenfassung, welche die wichtigsten Ergebnisse dieser Arbeit wieder hervorheben sollte.

Wir hoffen, daß nachdem man sich mit dieser Arbeit und den dort beschriebenen Ansätzen vertraut gemacht hat, ist man dann in der Lage, mit den praktischen Untersuchungen im Bereich paralleler ORDBS zu beginnen. Das Abspielen der Queries für die diversen Join-Verfahren liefert nicht nur die ersten Ergebnisse über die Effizienz der einzelnen Join-Algorithmen in parallelen ORDBS, welche mit der *SimPaD*-Erweiterung für OR-Daten im Rahmen dieser Arbeit erhalten werden. Es bietet dem Leser auch ein anschauliches Beispiel dafür, wie man die unterschiedlichen Vorgängen aus der realen Welt der parallelen ORDBS nachbildet.

Unabhängig davon, ob man unser Werkzeug für die weiteren Simulationen in dem Bereich verwendet oder sich mit dem Aufbau eines eigenen ähnlichen Tools für ORDBS befaßt, sollte diese Arbeit von Nutzen sein.

2. Theoretische Grundlagen

Dieses Kapitel bietet einige Informationen darüber, wie paralleles ORDBS organisiert wird und warum es so wichtig ist, diese Art der DBS-Systeme weiter zu erforschen. Die Kenntnisse über ORDB-Systeme werden dann im Laufe der Arbeit eingesetzt, um ihren Prototyp in Form eines Simulationsmodels zu erstellen.

2.1 ORDBS als neue Generation von DBS

Objekt-Relationale Datenbank-Verwaltungssysteme (ORDBVS) oder anders genannt Objekt-Relationale Datenbank-Managementsysteme (ORDBMS) wurden in [Sto96] als “The next great wave” bezeichnet. Als Expansionsursache wurden die zwei folgenden Gründe genannt:

- *Wachstumsrate für neue Multimediale Datenverarbeitung;*
- *Umgestalten von mehreren bereits existierenden DBS-Applikationen.*

Es wurde behauptet [OHE99], daß insbesondere im Business-Bereich, wo auch das Internet immer mehr an Bedeutung gewinnt, es zu den ausschlaggebenden Änderungen kommen wird. Die neue Generation der Software für verschiedene Netzwerkplattformen und Technologien, welche die jüngsten Internet-Anwendungen unterstützt (sog. Middleware), wird bereits mittels OO-Programmen unter der Verwaltung von kommerziellen CORBA- oder COM/DCOM-Werkzeugen geschrieben und beruht auf ein Konzept der verteilten Objekte. Deswegen stellt man sich folglich vor, die weitere Entwicklung der OO-Technologien auf den DBS-Bereich zu übertragen.

Ein Beispeil hierfür stellt das neue Segment von “Business to Business”-Anwendungen dar, wo durch ein Einsatz von Objekteigenschaften das Integrieren von verteilten Daten in ein geschlossenes kommerzielles System erleichtert werden kann. Die kommerziellen *Data-Mining*-Anwendungen, die nach bestimmten Intelligenzmustern in Datenbanken suchen, können auch von der Komplexität der OR-Datenstrukturen profitieren.

Nach der Meinung des Autoren [Sto96] gibt es gar keine andere Alternative zu der Ausbreitung von DBS als in die Richtung von ORDBS. Die RDBVS seien viel zu unflexibel aus der Sicht des Nutzers, um sich den neu anstehenden Herausforderungen in der Ära der OO-Technologien gegenüberzustellen. Obwohl die weitere Nutzung von RDB-Systemen sich fortsetzen sollte, kann es keinesfalls mit dem Tempo verglichen werden, wie ORDBS sich in die nächsten Jahren durchsetzen werden. Die OR-Features sind bereits in mehreren DBS vorhanden, die als RDB-Systeme allgemein bekannt sind.

Die anderen Alternativen wie OODBS, die heuzutage noch bekannt sind, stellen keine Konkurrenz zu den zwei o.g. Ansätzen dar. “O Vendor” Markt (der Markt für Objekt-Orientierte DB-Systeme) bleib zwar bestehen, ist aber kaum mit dem von RDBS zu vergleichen. Die Transaktionsbearbeitung und der Mehrbenutzerbetrieb werden bei den OODBS nur im beschränkten Umfang unterstützt, was dazu führen kann, daß die kommerziellen Produkte darunter leiden werden. Die Entwicklung der ORDB-Systeme wird hingegen durch die ganze Reihe der renommierten Unternehmen wie IBM (DB2), Oracle (Oracle8), Informix (Universal Server) etc. propagiert, was im Endeffekt die entscheidende Rolle beim Erwerben der Marktanteile spielen wird.

An dem Beispiel von OO-Programmiersprachen wie C++ oder Delphi, kann nachvollzogen werden, wie schnell neue Produkte, die mit OO-Tools geschrieben werden, sich auf dem Markt etablieren können. Solche Produkte sind viel einfacher zu produzieren und zu warten, weil die Programmiersprachen wesentlich benutzerfreundlicher und flexibler sind. Die neuen Technologien setzen sich immer schneller durch, vorausgesetzt wird eine entsprechende Unterstützung der wissenschaftlichen und wirtschaftlichen Gesellschaften rechtzeitig gewährt. Im Falle der ORDB-Systeme spricht man von dem SQL3 Standard **Absch. 2.1.2**, der im Jahre 99 offiziell im Betrieb aufgenommen wurde und die Grundlage für den Aufbau eines beliebigen ORDBMS geschaffen hat.

2.1.1 Objekt-Relationale Eigenschaften

Mittlerweile ist es ohne Zweifel, daß die RDB-Systeme ihre führende Rolle im Bereich der Datenverarbeitung behaupten. Da ein RDBS lediglich in der Lage ist, nur bestimmte Anfragearten über die einfacheren vordefinierten Datentypen effizient auszuführen, verbreitete man immer weiter Vorschläge, wie diese Unvollkommenheit behoben werden könnte. Eine der Lösungen ist es, ein DBMS mit wesentlich komplexeren Anfrageaufbau zu kreieren.

Aus Effizienzgründen ist es jedoch nicht möglich, alle vorstellbaren Anforderungen an ein DBMS zu stellen. Es wurde versucht, die relationalen Konzepte zu erweitern, ohne die dabei entstehenden Performanzverluste hinnehmen zu müssen. So sollte ein neues ORDBS in der Lage sein, wesentlich komplexere Queries ausführen zu können, ohne die Effizienz des gesamten DBS zu beeinträchtigen.

Die folgenden wichtigen Eigenschaften werden durch ORDBS zusätzlich zu relationalen unterstützt:

- *Vererbungshierarchie;*
- *Erweiterungen zu vordefinierten Datentypen;*
- *ADTs (Abstract Data Typs);*
- *UDFs (User Defined Functions);*
- *Verweisbasierteadressierung;*
- *Kollektionstype;*
- *Regelnsysteme (active Databases).*

Eine Vererbungshierarchie stellt eine der grundlegenden OO-Konzepte dar. Beim SQL 3 Standard wurde mehrfache Vererbung (im Gegensatz zu Java) sowohl für Daten als auch für Methoden erlaubt. Die Knappheit und Inflexibilität der vordefinierten Datentypen macht es nicht leicht eine beliebige Anfrage zu gestalten. Für die einfachen Graphikanwendungen wäre es z.B. sogar von Vorteil, einen solchen Datentyp zu besitzen, den ein Punkt, eine Linie oder ein Kreis als einfaches Attribut bei der Anfrageausführung referenzieren läßt. Mit dem neuen Standard ist man nun in der Lage, dieses Funktionalitätsproblem durch Definition von neuen Datentypen zu bewältigen.

Das Einführen von abstrakten Datentypen, komplexen Klassen und Funktionen, welche vom Nutzer definierten werden (sog. *UDFs*), kann als weitere Ergänzung zu dem o.g. Konzept der Verallgemeinerung von Datenstrukturen aufgefaßt werden. Einige Methoden können beispielsweise folgendermaßen deklariert werden, daß sie ihre volle Funktionalität nicht sofort - also zu dem Zeitpunkt der Deklaration - erwerben müssen, sondern sich dynamisch einbinden lassen. Wie die üblichen OO-Programmiersprachen, können solche Funktionen mit verschiedenen Bibliotheken geliefert werden, was manchmal wesentlich effizienter und einfacher ist als sie über eine SQL-Einweisung zu implementieren. Es ist hier auch zu bemerken, daß dieses Konzept ein höheres Maß an Sicherheit mit sich bringen kann [EM99].

Ein Primärschlüsselattribut wird in einem RDBS immer gebraucht. Die in einer OR-Datenbank gespeicherten Klassen benötigen keins. Um die verschiedenen Objekte während einer Anfrage zusammenzuführen, wird bei jedem Objekt eine Reihe von Verweisen mitgespeichert. Diese Information über die Speicherung von Referenzen bildet eine Basis für jede Art der Join-Anfrage und ermöglicht die Suche nach den passenden Verbindungspartnern, **Absch. 2.2.2** und **3.1.5**.

Die Art, wie solche Verweise in einer Klasse zusammengehalten werden, hängt ausschließlich von dem Aufbau der Klasse selbst ab. Jedoch ist es möglich, diese Speicherungsanordnung zu beeinflussen, indem man sog Kollektionstype[1] wie *List*, *Set* oder *Bag* deklariert. Die Kollektionstypen haben allerdings auch den Vorteil, daß sie die genaue Menge von Daten[2] nicht im voraus zu wissen brauchen.

Alle o.g. OR-Eigenschaften sind im SQL3/99 Standard festgesetzt, der gemeinsam bei zwei der größten internationalen Organisationen für Standards ANSI und ISO in dem letzten Jahr verabschiedet wurde [EM99]. Außerdem wird im Standard eine Reihe der Eigenschaften unterstützt, die nicht unbedingt ORDB-spezifisch sind, aber auch dort ihren Einsatz finden.

1. Bedauerlicherweise wurde im SQL 99 Standard lediglich ein der Kollektionstypen vorgesehen, nämlich *Set*.
2. Ein Kollektionstyp kann sowohl ganze Objekte als auch Verweisen als Bestandteile haben.

2.1.2 Neuer Standard SQL3 / 99

SQL3 / 99 wurde als weitere Form des bereits existierenden SQL-Standards zweiter Generation entwickelt, der allgemein unter dem Namen SQL 92 bekannt war. Der neue Standard wird von der ganzen Reihe der DBS-Herstellern **Absch. 2.1** unterstützt. Er bildet nicht lediglich seinen Vorgänger nach und fügt ihm nur einige typische OO-Eigenschaften hinzu, sondern für die Datenbankwelt ist es wesentlich wichtiger, daß sein Konzept eine Ordnung in die Vielfalt der Interpretationen und Realisierungsvorschläge für die typischen OR-Eigenschaften bringt.

Der neue Standard wird aus zwei Gruppen von Eigenschaften zusammengebaut. Die erste Gruppe besteht aus R-Eigenschaften, die bereits im SQL 92 -Standard festgelegt wurden. Die zweite enthält die neuen OR-Eigenschaften, die sich durch den Einsatz der neuen Operatoren und Predikatnotazionen bei der Beschreibung von OR-Anfragen, durch einige Änderungen in der Semantik und dem Gestalten von Tabellen, welche neue OR-Datentypen beinhalten, zeigen lassen.

Zu den neuen OR-Notationen gehören die “DOT”- und “REF”-Notationen, welche die Zusammensetzung von einzelnen Klassenkomponenten bzw. ganzen Klassen wesentlich vereinfachen sollen. Bei der Semantik wird man auch einige wesentliche Unterschiede bemerken. Ein Beispiel hierfür ist eine Rekursionsanfrage, die erst mit dem neuen Standard möglich wird:

```
WITH RECURSIVE
        Q1 AS SELECT  ... FROM ... WHERE
        Q2 AS SELECT .... FROM .... WHERE
SELECT .... FROM Q1, Q2  WHERE
```

Ein höheres Maß an Sicherheit wird auch dadurch erreicht, daß man die Veränderungen bei Datenbeständen in einem OR-DBS mit Hilfe von Regeln leichter kontrollieren kann. Solche Regeln beschützen die Integrität der Daten [Sto96]. Eins der passenden Werkzeuge, das für das Ausführen einer Regel benötigt wird, heißt *Trigger.*

Viele DBS-Hersteller haben *Triggers* nach ihrer Art und Funktionsweise sehr spezifisch gebaut, was für die Anwendungsentwickler sehr irreführend ist. Mit dem neuen Standard wurde die Syntax festgelegt, wie in Zukunft die *Triggers*-spezifischen Anfragen gestaltet werden sollen. Allerdings sind nicht nur die *Trigger* dem neuen Standard nach zu spezifizieren. Die anderen wichtigen DBS-Werkzeuge wie z.B. *Stored Procedures*, die schon längst existiert hatten, wurden mit dem neuen Standard offiziell eingeführt.

Die Frage, inwieweit die Datenbankhersteller dem neuen Standard in Zukunft folgen werden, kann nur im Laufe der Zeit beantwortet werden. Meiner Ansicht nach ist der Standard noch viel zu “frisch”, um darüber eine bestimmte Aussage treffen zu können.

2.2 Zentrale OR-Verarbeitungs- und Speicherungskonzepte

Die Grundlagen der DBS-Verarbeitungs- und Speicherungskonzepte werden in vielen Werken gründlich beschrieben. In dieser Arbeit werden vor allem die Informationen aus dem Buch [HR99] verwenden, da dort mehrere moderne Ansätze konzipiert wurden, die für RDBS geeignet sind und sich leicht auf ORDB-Systeme übertragen lassen. Der Umgang mit komplexen und zusammengesetzten Datentypen, der für eine gute ORDB-Allokation im zentralen Fall sehr wichtig ist, wurde im [HR99] übersichtlich dargestellt.

2.2.1 Mögliche Aufbaukonzepte eines ORDBS

In [Sto96] wurden verschiedene Konzepte für den Aufbau und die Architektur eines ORDBS aufgezeigt. Mehrere der DBS-Hersteller besitzen bereits eine gute Implementierung eines RDBS, so daß sie nicht ganz auf die Eigenschaften ihrer früheren Versionen verzichten wollten und konnten. Außerdem wurde ganz richtig bemerkt, daß viele Anfragen, welche an ein ORDB-System gestellten werden, nach ihrer Natur nicht den OR- sondern einfachen R-Charakter haben. Für diese Anfragearten ist eine relationale Architektur oft mehr geeignet.

Somit kommt man in [Sto96] zu der Untersuchung der verschieden Strategien für den Aufbau eines ORDBS, für welche es ganz natürlich wäre, die o.g. Arten der R-Anfragen generisch zu unterstützen. Auf dem **Bild 1** sind zwei Alternativen zum Übergang von einem RDBS zu einem ORDBS zu sehen. Die erste Variante sieht ein *Wrapper*-Anwendung vor - also ein Werkzeug, das die an das ORDBS gestellten Anfragen beim Bedarf in R-Queries umwandelt und an den relationalen Kern des Systems weiterleitet.

Als Alternative wird die sog. OR-*Gateway*-Architektur für ein ORDBS vorgeschlagen. Hierbei handelt es sich um die beiden ORDB und RDB-Systeme, die ihre Aufgaben bezüglich Datenallokation und Anfrageausführung immer zwischeneinander aufteilen sollen. Darüber hinaus wird jedes der Systeme für seine Art der Anfragen und Daten verantwortlich gemacht. Die Kommunikation zwischen den beiden Teilsystemen erfolgt durch eine API-Brücke, die klug und schnell genug sein sollte, um die Daten nach ihrer Natur zu unterscheiden, zu trennen und wieder zusammenzusetzen.

Am Ende der Diskussion kommt man in [Sto96] jedoch zu dem Schluß, daß das generische ORDBS, d.h. das System, welches von Anfang an als ORDBS konstruiert wird und sowohl die Speicherung aller Datentypen als auch die Bearbeitung aller möglichen Anfragetypen erlaubt, im Endeffekt am effizientesten sein müsse. Aufgrunddessen werden in dieser Arbeit keine weiteren Ansätze in Betracht gezogen.

2.2.2 Konzepte der Datenspeicherung für OR-Daten

Es sind viele verschiedene Alternativen möglich, wie die Daten innerhalb eines generischen ORDBS gespeichert werden könnten (Beispiel [TB94]). Darüber hinaus bieten DBS-Hersteller den DBS-Administratoren immer mehr mögliche Varianten an, durch die gezielt und geschickt gewählten Allokationsstrategien die Datenspeicherungsstrukturen zu beeinflussen ([HR99], [JM98]).

Die Auswahl der Speicherungsstrategie wurde durch die Fülle der verschiedenen Vorschlägern erschwert[1]. Die Zahl von Speicherungskonzepten ist heuzutage unendlich groß. Man kann behaupten, daß es für kein der existierenden kommerziellen Systeme vorstellbar ist, alle Speicherungsstrategien in ihrer DBS-Implementierung zu umfassen. Als Folge daraus wird bei jedem einzelnen Vendor die Entscheidung getroffen, welche der möglichen Variationen zu realisieren und zu unterstützen ist.

1. Unsere Auswahl an Speicherungsstrategien wird im Laufe der Arbeit **Absch. 3.1.5** auf dem Beispiel des Bucky-Schemas geklärt.

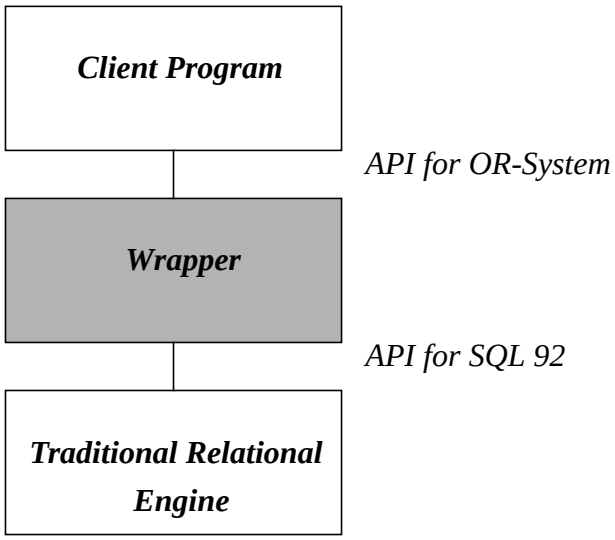


Bild 1a. *Wrapper-Architektur* (nach [St96])

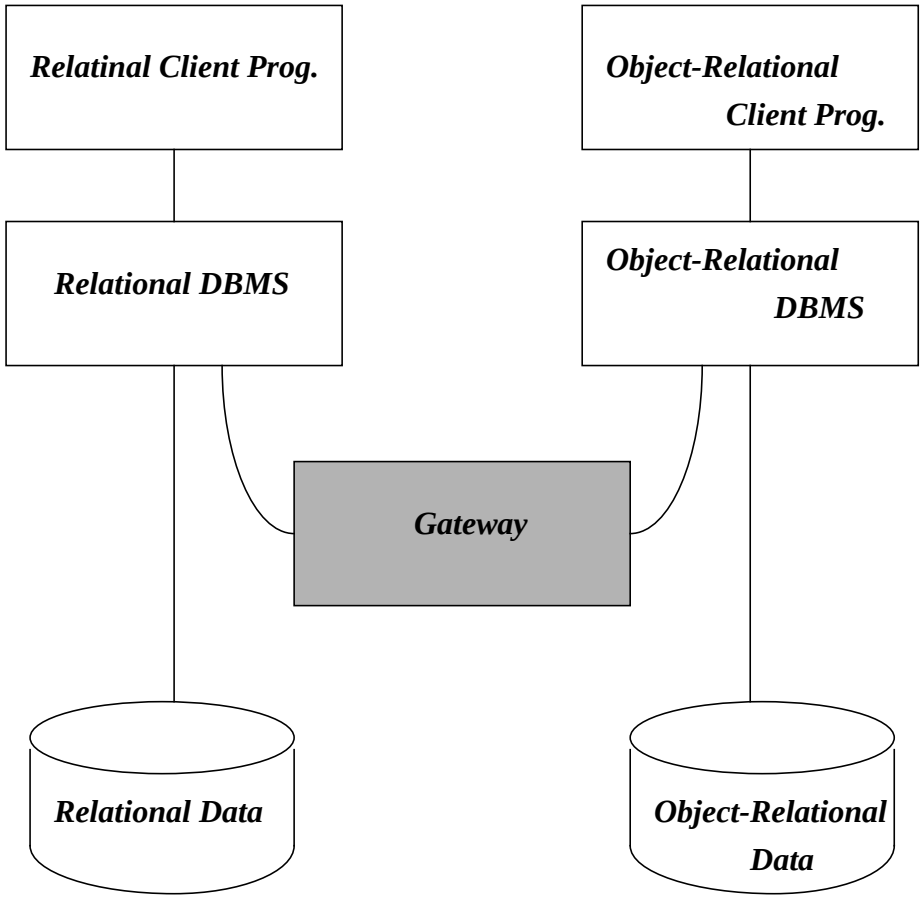


Bild 1b. *Gateway-Architektur* (nach [St96])

Diese Vorgehensweise wird in dieser Arbeit übernommen. Schon am Beginn der Konzeptionstätigkeit trifft man die Entscheidung über die zu unterstützenden Speicherungsmethoden. Mehrere verschiedene Alternativen werden in Betracht gezogen und aus Effizienzgründen verworfen. Alle dabei entstandenen Überlegungen werden hier nicht beschrieben, weil sie den Rahmen dieser Arbeit gesprengt hätten. Die anderen Vorschläge, die für das Konzept dieser Arbeit interessant gefunden sind, werden in unser Simulationssystem generisch integriert.

Die Kriterien, welche die zu treffenden Entscheidung über die Datenspeicherung in einem zentralen DBS beeinflussen können, sind die folgenden:

- *Segmentierung und Clustering einzelner Relationen bzw. Klassen;*
- *Effizienz beim Plattenzugriff während der Anfrageausführung;*
- *entstehender Aufwand bei Speicherung gegenseitiger Referenzen [CD96a];*
- *Anzahl der Verweise, die für das vorgegebene Speicherungsschema benötigt werden;*
- *der gesamte Speicherplatz, der für die Plazierung von Objekten gebraucht wird;*
- *Einsatzmöglichkeiten für Indexstrukturen.*

Die Daten, die logisch zusammengehören, sollten in einem DBS auf der Platte so gehalten werden, daß sie die Grenzen eines Segments nicht überschreiten. Hiermit werden sog. *Clusters* von Daten gebildet. Der Zugriff auf die benachbarten Seiten wird durch geschickte Plazierung beschleunigt, da nicht nur eine sondern mehrere Seiten während eines Lesevorgangs und eines Zugriffes auf denselben Datensegment gelesen werden können. Eine der Techniken, die das erlaubt, heißt das *Prefetching*.

Es ist bekannt, daß Zugriffe auf Platten und den damit verbundenen E/A-Aufwand oft als Flaschenhals eines Rechnersystems, - unter anderen auch eines DBS, - betrachtet werden. Also, wird für jedes DBS das Ziel angestrebt, die Anzahl von externen Zugriffen auf der Hardware-Ebene soweit wie möglich niedrig zu halten. In einem DBS versucht man deswegen die wichtigsten Daten, die mehrmals vom System angefordert werden können, in den Puffern des Systems zu fixieren, um das mehrfache Lesen von den Platten für sie zu vermeiden.

Ein wichtiger Bestandteil jedes DBS ist der *Anfrageoptimierer.* Er sorgt dafür, daß eine beliebige Anfrage in der effizientesten Reihenfolge abgewickelt wird. Das läßt sich unter anderem durch die am wenigsten auftretenden gegenseitigen Störungen von Prozessorknoten bei den Plattenzugriffen erreichen. Eine Hauptaufgabe eines *Anfrageoptimierers* besteht also darin, eine geschickte Reihenfolge für die Teiloperatoren einer Anfrage bzw. einer Transaktion vorzugeben, um gleichzeitige Zugriffe auf eine Platte seitens zwei oder mehrerer CPUs soweit wie möglich zu verhindern.

Die Speicherung von Verweisen ist für die meisten ORDB-Systeme wesentlich kritischer als für ihre R-Konkurrenten. Ein RDB enthält keine Referenzen, die dem logischen Aufbau eines DB-Schemas helfen. Die Referenzen in einem RDBS sind lediglich für interne Zwecke da, wie beispielsweise für Freispeicherverwaltung, Metadatenverwaltung und haben keinen direkten Einfluß auf die Schemarelationen.

Ein ORDBS unterstütz hingegen alle möglichen Typen von Referenzen. Vor allem ist es wichtig, die entgegengerichteten Verweise eines OR-Schemas zu pflegen. Die Schwierigkeiten für diesen Referenztyp entstehen dadurch, daß bei erstem Anlegen der Datenbank solche Verweise auf die nicht existierenden Objekte zeigen sollen und damit nicht vollständig spezifiziert werden können. Nur dann, wenn der Speicherungsvorgang für alle Klassen des Schemas komplett abgeschlossen ist, wird das ganze System erst in einen konsistenten Zustand gebracht. Die Zahl der benötigten Verweise in einem ORDB-Schema spiegelt sich in einem zusätzlichen Speicherplatzbedarf wieder, was keine positiven Einflüsse auf die Performanz des ganzen DBS haben kann. Durch ein gewähltes Speicherungskonzept läßt sich jedoch diese Zahl manipulieren.

Die Index-Strukturen werden eingesetzt, um den E/A-Aufwand zu vermindern und bei der Suche nach wenigen benötigten Seiten, die durch die Ausprägungen einzelner Attribute bestimmt werden, schneller ans Ziel zu kommen. Da in einem ORDBS nicht nur Attribute sondern auch Verweise die Träger der entscheidenden Informationen sind, wird man sie auch indexieren können.

2.3 Parallele DBS-Architekturen

Nach Definition besteht ein paralleles DBS aus mehreren Prozessorknoten, Platten und einem integrierten Schema, das für jede DBMS-Instanz in allen Prozessorknoten allgemeingültig ist. Die Prozessoren und Platten sind durch eine passende Netzwerkstruktur eng verbunden und befinden sich in einer lokalen Entfernung von einander [Ra94]. Die Hauptanforderungen, die an die DBS-Architekturen gestellt werden, sind sowohl die Lösung des Performanzproblems - also die viel schnellere Abwicklung von Transaktionen und einzelnen Anfragen (hohe Leistungsfähigkeit),- als auch die mögliche Skalierbarkeit (Modulare Wachstumsfächigkeit des Systems), effiziente Datenabsicherung gegen Platten bzw. CPUs-Ausfällen[1] und hohe Kosteneffektivität.

Zu der letzten Eigenschaft ist anzumerken, daß die große Anzahl von Mikroprozessoren heuzutage einen Kostenvorteil gegenüber Großrechnern anbietet, vorausgesetzt ihre Ressourcen können durch eine geeignete Konfiguration voll genutzt werden. Als Folge daraus sucht man gezielt nach den Architekturen, die keine Großrechner benötigen, sondern aus den mehreren PCs zusammengebaut werden können. Die hier vorgestellten Ansätze können ohne Einsatz eines großen Rechnersystems auskommen.

2.3.1 SN, SD und SE parallele DBS-Architekturen

Man unterscheidet zwischen *Shared-Nothing* (*SN*), *Shared-Disk* (*SD*) und *Shared-Everything* (*SE*) parallelen DBS-Architekturen [Ra94]. Alle drei Architekturen sind auf dem **Bild 2** grob skizziert. Fast alle heuzutage existierenden DBS-Architekturen können einer der o.g. Alternativen zugeordnet werden bzw. durch ihre Eigenschaften grob charakterisiert werden.

SE-Architektur basiert auf einem Multiprozessor-System mit gemeinsamen Hauptspeicher. Die Skalierbarkeit des System ist durch eine bestimmte Zahl von Prozessorknoten im System stark beschränkt. Für sehr große Datenbanken kann deswegen diese Art der Architektur kaum eingesetzt werden.

1. Das ist meistens durch eine Daten und Hardware-Replikation erreichbar.

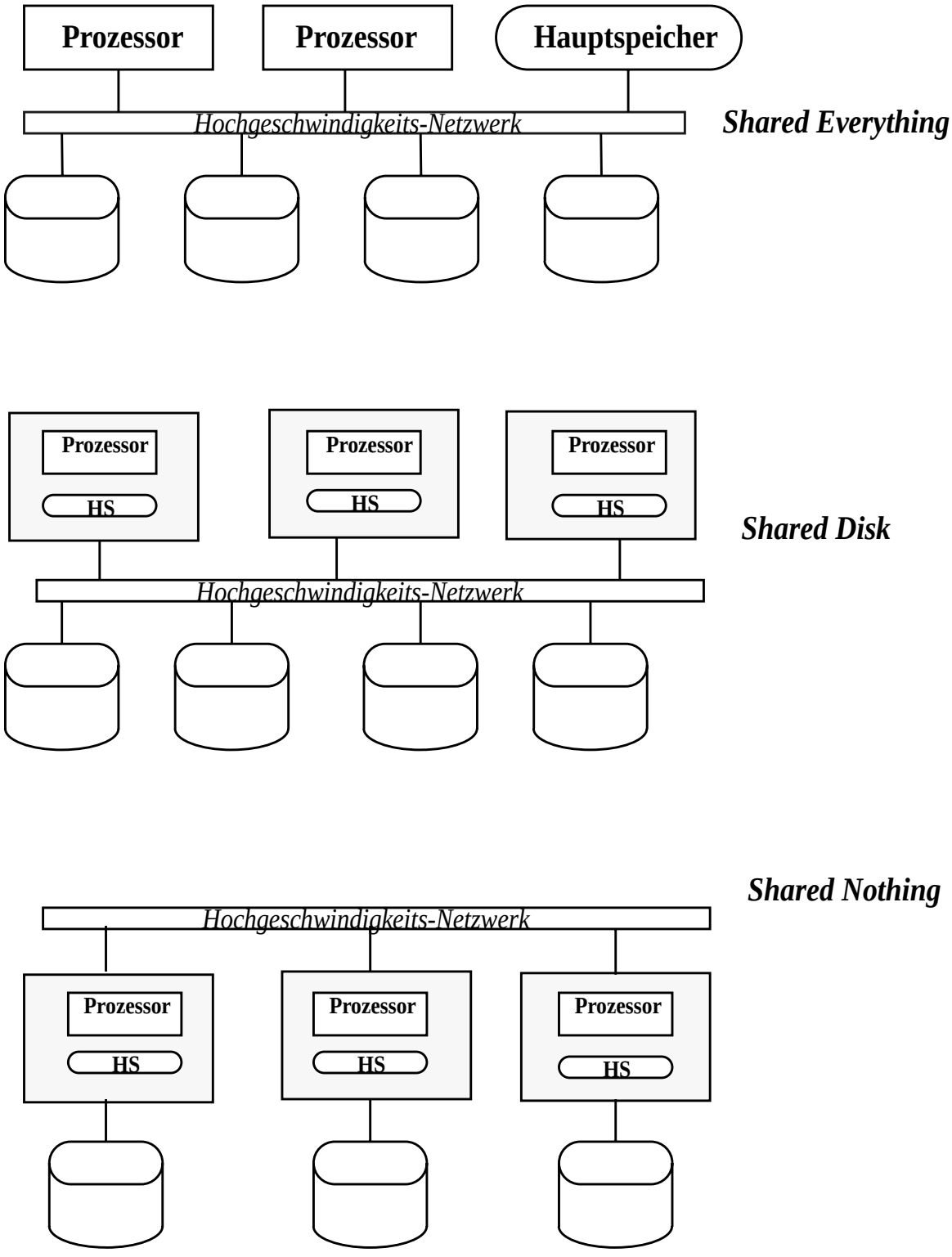


Bild 2. *Parallele DBS-Architekturen* (nach [Ra94])

Bei den beiden *SD*- und *SN*-Architekturen liegt in jedem Rechnerknoten eine vollständige Instanz des DBMS, so daß jeder Prozessor ohne Hilfe von außen parallel mit einander arbeiten kann. Bei einer *SN*-Architektur soll jedoch die Allokation von Daten statisch erfolgen, weil jede Platte einem Prozessorknoten eindeutig zugeordnet wird.

Im Gegensatz zu *SN* bietet die *SD*-Architektur einem parallelen DBVS die Möglichkeit, von beliebigem Prozessorknoten aus auf jede Platte zugreifen zu können. Im Vergleich zu der *SE* hat die *SD*-Architektur ein höheres Maß an Skalierbarkeit, Flexibilität, Leistungsfähigkeit und ist nicht mehr so kostspielig. DBS mit *SD*-Architektur weisen verglichen mit *SN*-Architektur ein höheres Potential zur dynamischen Lastbalancierung auf, das heuzutage noch nicht in vollem Maße genutzt wird. Als typische Beispiele für die parallelen DBS mit *SD*-Architektur sind Oracle Parallel Server, DB2/MVS (IBM) etc. hier zu nennen.

In der Arbeit wird lediglich eine Art der o.g. Architekturen, nämlich die *SD*-Architektur, näher untersucht. Die *SD*-Architektur hat bereits in vielen existierenden Systemen ihren Einsatz gefunden, ist aber immer noch nicht genug erforscht. Vor allem wurden noch keine Literaturquelle entdeckt, wo die Instanzen eines ORDBMS als Untersuchungsobjekt im Rahmen der *SD*-Architektur ins Visier genommen werden. Das OR-Simulationssystem, das während dieser Arbeit entwickelt wird, leistet diesem Untersuchungsaspekt einen Beitrag.

2.3.2 Arten von Parallelität

In der Literatur spielen die Arten von Parallelität eine bedeutende Rolle. Hierbei handelt es sich nicht unbedingt um die Parallelitätsarten einer ganzer Transaktion, sondern es kann von einer Anfrage- bzw. einer Query-Parallelität oder sogar einer Operator-Parallelität die Rede sein. Die Parallelität von Daten wird allerdings noch hinzugezählt. Die Hierarchie zwischen den einzelnen Parallelitätsarten ist auf dem **Bild 3** zu sehen.

Eine Inter-Transaktion-Parallelität erfordert eine parallele Ausführung von mehreren Transaktionen, die im Rahmen eines Mehrbenutzerbetriebs entstehen. Bei der Inter-Query-Parallelität handelt es sich um die Queries, die innerhalb einer Transaktion zur Ausführung gebracht werden und alle entweder erfolgreich beendet werden oder scheitern müssen (*ACID*-Eigenschaften).

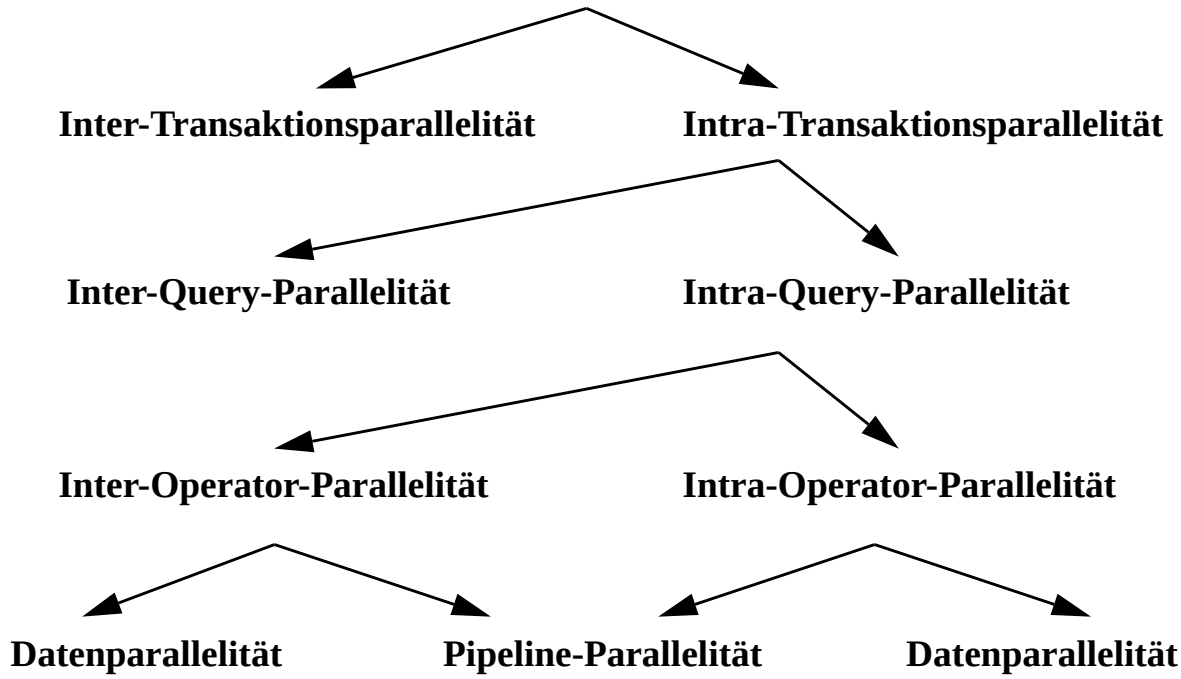


Bild 3. *Arten der parallelen DB-Verarbeitung* (nach [Ra94])

Bei der Intra-Query-Parallelität werden verschiedene Operatoren derselben Transaktion parallel bearbeitet. Man muß dabei beachten, daß die parallele Bearbeitung der Operatoren nur dann erfolgen kann, wenn sie voneinander ganz unabhängig sind. Unter einer Inter-Operator-Parallelität versteht man eine gleichzeitige Ausführung von mehreren Operatoren einer Anfrage. Auch hier wird die Abhängigkeit zwischen den Operatoren streng kontrolliert.

Es ist aus Effizienzgründen **Absch. 2.2.2** äußerst wichtig, daß ein paralleles DBS sowohl Daten- als auch Pipeline-Parallelität bei einer Anfrageausführung generisch unterstützt. Die Datenbestände werden in einem parallelen System partitioniert, so daß verschiedene Operatoren bzw. Teiloperatoren auf disjunkten Datenpartitionen arbeiten. Pipeline-Parallelität sieht eine überlappende Ausführung verschiedener Operatoren bzw. Teiloperatoren vor [Ra94], so daß die Ergebnisse immer weitergeleitet werden, ohne das Ende der ganzen DBS-Operation abzuwarten. Somit wird die Datenverarbeitung wesentlich beschleunigt. Außerdem trägt die Pipeline-Parallelität dazu bei, die einmal gescannten Daten nicht auf die Platten ausschreiben zu müssen und hiermit den E/A-Aufwand zu reduzieren.

In [JM98] wurde gezeigt, wie eine mögliche Lösung des Parallelisierungsproblems im Falle eines ORDBS, wo die Aggregationsartige-*UDFs* einer speziellen Behandlungsroutine unterlagen, aussehen soll. Im Gegensatz zu [JM98] befaßt man sich hier mit der Suche nach:

- geeigneter Datenallokation, die genügend Datenparallelität aufweist und als Basis für die Parallelen Operatoren dienen kann;

- Verfahren, die effiziente parallele Ausführung einzelner OR-Operatoren erlauben;

- Faktoren, welche die möglichen Ablaufszenarien für eine OR-Anfrage im Rahmen paralleler SD-Architektur beeinflussen können.

2.3.3 Intra-Operator-Parallelität

Unter der Intra-Operator-Parallelität versteht man die parallele Ausführung einzelner Operatoren während einer Anfrage. Im Gegensatz zur Inter-Operator-Parallelität, die dem Query-Optimisatoren auch bekannt ist, wird hier nicht der ganze Anfragebaum durchsucht, um die Ausführungsreihenfolge von einzelnen Operatoren zu bestimmen. Bei Intra-Operator-Parallelität handelt es sich eher um eine bestimmte Zahl der vorgegebenen parallelen Algorithmen, die es ermöglichen, jeden einzelnen Operator parallel durchzuführen.

Selbstverständlich ist es auch möglich, jeden der Operatoren zentral - also bei dem Involvieren lediglich eines Prozessoren und eines Puffers - zu bearbeiten. Diese Variante ist aber im Falle eines parallelen DBS inakzeptabel. Eine geeignete Datenallokation ist schon deswegen die wichtigste Voraussetzung für die parallele Verarbeitung, da bereits beim Lesen von Daten der gesamte Bearbeitungsprozeß beeinflußt wird. Nur dann, wenn die zu lesenden Relationen (Klassen) in mehreren horizontalen Fragmenten zerlegt werden, läßt sich ein Scan-Operator parallelisieren. Diese wichtige Eigenschaft muß in einem guten Allokationsschema vertreten sein.

Darüber hinaus wird eine Intra-Operator-Parallelität aus den Überlegungen zur Lastbalancierung gefordert. Bei einer *SD*- oder *SE*-Architektur **Absch. 2.3.1** werden die Teiloperatoren zwischen den Verarbeitungsrechnern so dynamisch verteilt, daß die Prozessoren, die am wenigsten ausgelastet sind, die größten Datenmengen zu bearbeiten bekommen. Im Falle einer *SN*-Architektur legt man eine eindeutige Zuordnung zwischen Prozessorknoten und Datenfragmenten fest und versucht, durch Lokalität der Teiloperatoren die Kommunikations-, E/A- und Synchronisationskosten zu reduzieren.

Beliebige Fragmentierung der zu lesenden Relation (Klasse) gewehrleistet aber keinen sicheren Erfolg, auch wenn die Anzahl der Partitionen vorbestimmt und entsprechend gut festgelegt wird. Oft werden sog. *Skew-Effekte* auftreten, die den gesamten Bearbeitungsprozeß - also sowohl Lesen der Daten, das weitere Pipelining etc. - beeinflussen. Die *Skew-Effekte* entstehen meistens durch die Asymmetrie einzelner Anfragearten.

Es ist eine anspruchsvolle Aufgabe sowohl für ein gutes DBMS als auch für einen DB-Administratoren, die *Skew-Effekte* soweit wie möglich zu eliminieren. Ein DB-Administrator kann die Prozesse nur beschränkt beeinflussen, indem er aufgrund der vorliegenden Statistiken über die einzelnen Anfragearten (ihre Häufigkeit, Wichtigkeit, Umfang etc.) eine geeignete statische Allokation vornimmt. Ein gutes DBMS stellt nicht lediglich dem Administratoren eine Reihe von Werkzeugen zur Verfügung, mit welchen er die gute Allokationsvariante selbst auswählen kann. Es müßte auch in der Lage sein, die entstehenden *Skews* dynamisch korrigieren zu können, indem es den am wenigsten belasteten *Hardware*-Komponenten auch die größten Aufgaben erteilt.

3. Problemanalyse

In diesem Kapitel werden die theoretischen Probleme diskutiert, die während des Aufbaus unseres OR-Simulationssystems zu lösen waren. Die getroffenen Entscheidungen werden auf dem umfassenden Beilspiel des Bucky-Benchmarks illustriert. Die Literaturquellen, die unsere Ansätze begründet haben, werden hier angegeben. An einigen Stellen überschneidet sich dieses Kapitel mit dem **Kap. 2.** Es war manchmal nicht leicht, bei einigen Aspekten des Systemaufbaus sich auf die Konzepte aus der Theorie zu beziehen, da dort nicht immer eine gewisse Antwort auf die von uns gestellten Fragen gefunden werden kann.

3.1 Bucky-Benchmark

Es ist bekannt, daß die Güte eines Produkts, insbesondere eines beträchtlichen kommerziellen Systems, mit Hilfe von verschiedenen Benchmarks auf dem Markt bewertet wird. Für ein ORDBS hat man auch nach den verschiedenen Schemata und Anfragearten gesucht, die einem fairen Vergleich zwischen einzelnen Systemen helfen sollten.

Das Bucky-Benchmark (beschrieben in [CD96a]), welches das Resultat von Bemühungen und der gründlichen Analyse der verschiedenen Spezialisten (Informix, Wisconsin-Madison Universität, USA etc.) im DBS-Bereich ist, könnte als Kriterium zur Auswertung eines ORDBS genommen werden. Dieses Benchmark wird teilweise hier verwendet, weil es noch kein Äquivalent gibt, das für ORDB-Systeme als Prüfungsmuster allgemein bekannt ist und auf das wir uns mit Sicherheit verlassen könnten. Das Bucky-Benchmark beinhaltet sowohl eine R- als auch eine OR-Spezifikation der Struktur einer Universität-Miniwelt, die für das DB-Schema zugrunde gelegt wurde. Die Kardinalitäten der Relationen bzw. Klassen wurden in dem Bucky-Schema ebenfalls gegeben.

Die Beschreibung einer Universitätsminiwelt wird als typisches Beispiel für den Bereich, wo DB-Systeme zum Einsatz kommen, bei vielen Professoren und Autoren sehr oft angegeben. Der Grund dafür ist die Übersichtlichkeit dieses Beispiels, das für alle verständlich ist und alle möglichen Arten der Beziehungen zwischen den in dem Schema vorhandenen Relationen (Klassen) enthält. Aufgrunddessen wird auch in dieser Arbeit die Analyse und der Konsistenztest für die Ergebnisse auf dem Beisiel vom Bucky-Schema durchgeführt.

Die Autoren des Benchmarks haben versucht, die wichtigsten Eigenschaften eines OR-Schemas in das Benchmark mit einzubeziehen. Andererseits blieben alle R-Eigenschaften in dem Benchmark erhalten, so daß das Resultat, das nach dem Vergleich der beiden Schemata herauskam, unabhängig von jedem der gewählten Schemavariationen sein sollte. Am Beispiel von ORDBS-Illustra wurde die Vorgehensweise beim Umgang mit dem Benchmark gezeigt und die ersten Ergebnisse interpretier.

Da man sich in dieser Arbeit mehrmals auf das Bucky-Benchmark bezieht, werden einige Fragmente aus dem Benchmark besser erklärt. Dort, wo die mangelhafte Behandlung von für diese Arbeit relevanten OR-Eigenschaften festgestellt wurde, wurde das Schema sogar ergänzt und erweitert. Die Anfragen zum Schema werden hier neu formuliert, da sie in neuer Form unseren Erfordernissen besser entsprechen **Absch. 3.1.3**. Mit ihrer Hilfe wird eine umfassende Prüfung der Systemkomponenten durchgeführt und die Fähigkeit des Simulationssystems getestet, OR-Eigenschaften zu unterstützen.

3.1.1 Erläuterung zu dem Bucky-Schema

Auf **Bild 4** wurde nicht das ganze Bucky-Schema abgebildet, wie es ursprünglich in der Quelle [CD96a] gewesen war, sondern nur ein Fragment davon. In der nachstehenden Erklärung zu dem Allokationsalgorithmus werden lediglich die Klassen benötigt, die in dem Teilschema vorkommen. Da die Prüfung der Fähigkeiten des Simulationssystems kein Schwerpunkt dieser Arbeit ist und für die endgültige Ergebnisauswertung **Kap. 5** wiederum nur die Daten aus dem Teilschema benötigt werden, wurde darauf verzichtet, alle Details aus der ursprünglichen Bucky-Variante hier zu erklären.

Außerdem wurde das Benchmark an einigen wenigen Stellen erweitert, wo meiner Ansicht nach der Bedarf dafür bestand. Die Autoren des Benchmarks haben einige wenige Kleinigkeiten nicht in Betracht gezogen, da sie vielleicht ein anderes Ziel für sich gestellt hatten. Um das Schema zu vervollständigen wurde beispielsweise die BLOB-Attribute anstelle einfacher Char-Arrays für die Klasse *Person* eingeführt, da sie meiner Ansicht nach für die Abbildung von Bildern, Attribut *Picture*, besser geeignet sind.

Um die Anfrageverarbeitung bei der Simulation des ORDBS unter korrekten Bedingungen zu testen, wird das Schema mit den Instanzenzahlen gefüllt, die das Verhältnis zu den Kardinalitäten einzelner Klassen in [CD96a] ohne Verzerrungen reflektieren. Die numerischen Angaben sind die folgenden: Studentenzahl - 500000, Anzahl von Abteilungen = 2500, Anzahl von HiWis -250000, Professorenzahl - 250000, Kinderzahl - 2500000. Die Kinder können durch das Verfahren in [CD96a] automatisch generiert und den einzelnen Personen zugewiesen werden.

Den Leser soll es nicht wundern, daß die numerischen Angaben nicht ganz den Verhältnissen entsprechen, die man in der Realität findet. Eine prinzipielle Bedeutung tragen die Zahlen nicht. Es ging mehr darum, die Schemaklassen so zu füllen, daß die Ergebnisse, die nach den Simulationsläufen erhalten werden, als aussagekräftig interpretiert werden können. Somit wird dieses Verzehnfachen der Werte gerechtfertigt.

3.1.2 OR-Spezifikation der Bucky-Klassen

CREATE ROW TYPE ***Person*** *(*

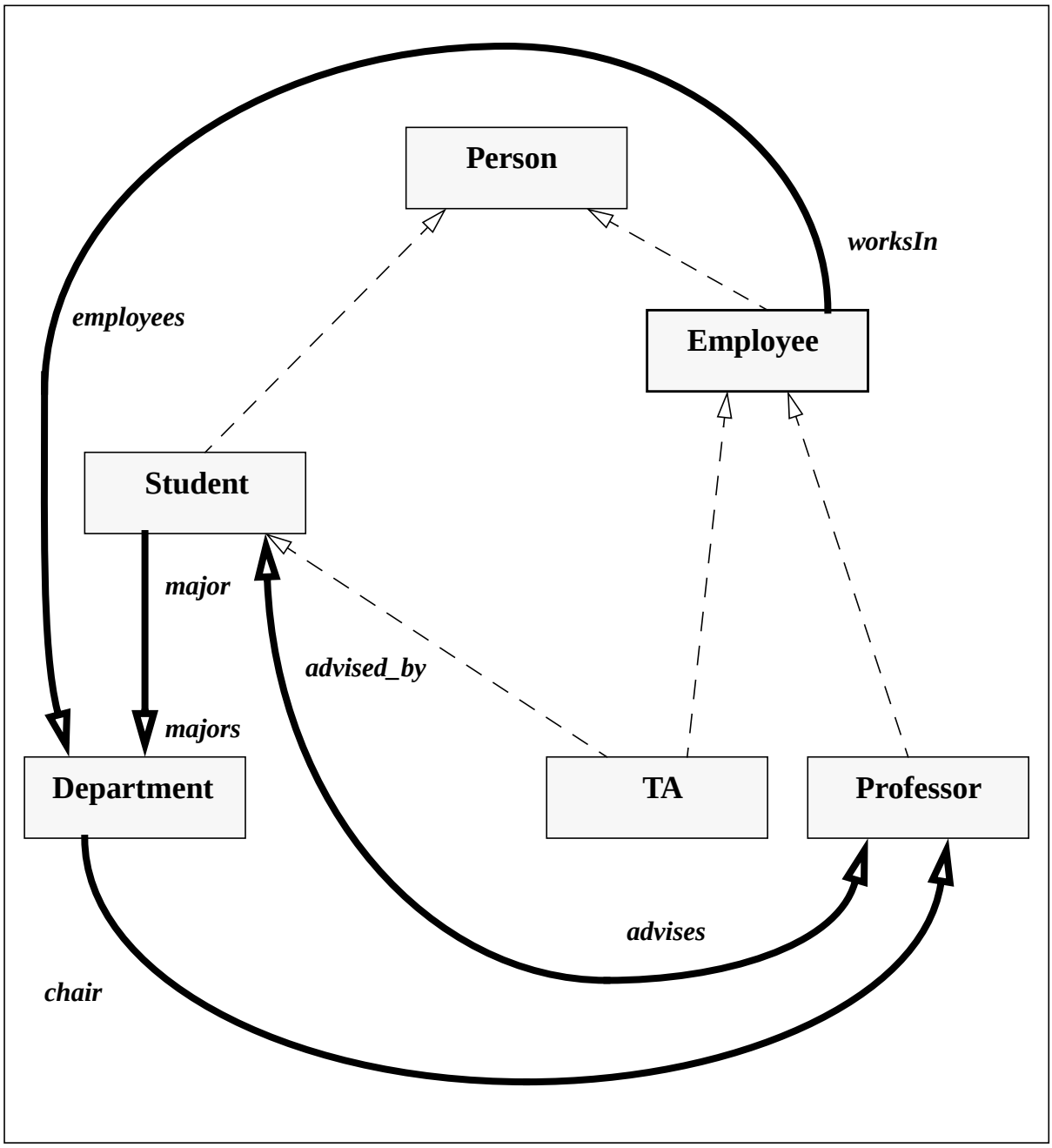
id	*Integer,*
name	*Varchar(20),*
street	*Varchar(20),*
city	*Varchar(20),*
picture	*BLOB,*
kidNames	*Set(Varchar(10)),*
place	*LocationADT);*

CREATE TABLE ***PERSON*** *OF ROW TYPE* ***Person*** *... ;*

CREATE ROW TYPE ***Department*** *(*

depatNo	*Integer,*
name	*Varchar(10),*
building	*Varchar(10),*
budget	*Integer,*
place	*LocationADT,*
chair	*Ref(**Professor**),*
employees	*Set(Ref(**Employee**),*
majors	*Set(Ref(**Student**));*

CREATE TABLE ***DEPARTMENT*** *OF ROW TYPE* ***Department*** *... ;*



- - - ▷ - *Vererbungshierarhie*

——▶ - *1:N Beziehung*

◀——▶ - *N:M Beziehung*

Bild 4. *Bucky-Teilschema*

*CREATE ROW TYPE **Student** (*
studentNo Integer,,
*major Ref(**Department**),*
*advised_by Set(Ref(**Professor**))*
*UNDER **Person**;*
*CREATE TABLE **STUDENT** OF ROW TYPE **Student** UNDER **PERSON**... ;*

*CREATE ROW TYPE **Employee** (*
annualSalary Integer,
*worksIn Ref(**Department**))*
*UNDER **Person**;*
*CREATE TABLE **EMPLOYEE** OF ROW TYPE **Employee** UNDER **PERSON**. ... ;*

*CREATE ROW TYPE **Professor** (*
*advisees Set(Ref(**Student**))*
*UNDER **Employee**;*
*CREATE TABLE **PROFESSOR** OF ROW TYPE **Professor** UNDER **EMPLOYEE** ... ;*

*CREATE ROW TYPE **TA** (*
status Integer)
*UNDER **Employee**;*
*CREATE TABLE **TA** OF ROW TYPE **TA** UNDER **Employee** ... ;*

3.1.3 Gewählte Anfragen zum Schema

Eine Reihe der Anfragen zu dem Bucky-Schema wurde bereits im Benchmark spezifiziert. Es wäre möglich gewesen, sie alle ohne weitere Änderungen einfach zu übernehmen. Jedoch haben wir uns für eine andere Strategie entschieden. In diesem Abschnitt werden die Anfragen spezifiziert, die zwecks Korrektheitsprüfung des ganzen Simulationssystems sowie seiner richtigen Parametrisierung **Absch. 5.1** gewählt wurden. Die Anfragen, die für das Erzielen der Schlüsselergebnisse eingesetzt werden, findet man dann im **Absch. 5.3**.

Die Komplexität der hier definierten Anfragen wurde während der Untersuchung sukzessiv ausgebaut. Der Einsatz von immer größeren Operatormengen in verschiedenen Kombinationen erlaubte es solche Übergangswerte zu finden, bei denen die Änderungen in der Performanz erwartet werden können. Außerdem war es für uns wichtig, ein solches Lastprofil zu erschaffen, daß die einzelnen Allokations- und Balancierungsstrategien möglichst gut isoliert werden. Darüber hinaus sollten die Nebeneffekte, die beim Testen auftreten könnten, sowohl in einzelnen als auch in verschiedenen Kombinationsvarianten geprüft werden. Die zu untersuchenden OR-Eigenschaften sollten immer den Boden für unsere Forschungsarbeit bilden.

Als Ergebnis unserer Überlegungen, haben wir die folgenden Eigenschaften für die einzelnen Anfragen angestrebt:

- *gleichzeitiger Zugriff auf vererbte und nicht vererbte Klassenattribute (d.h. über mehrere Hierarchieebenen);*
- *gleichzeitiger Zugriff auf ausgelagerte mengenwertige und andere Attribute;*
- *Zugriff auf Attribute von Relationen, die per Referenz verknüpft sind;*
- *Zugriff auf vererbte, mengenwertige bzw. Referenzattribute sowohl einfach lesend als auch selektierend;*
- *Eignung für die Basisverfahren Marge-Join, Hash-Join bzw. verschränkter Scan;*
- *Selektion über indizierte und nicht indizierte Attribute*[1]*;*
- *schwache Selektivität (so daß größere Tupelmengen verarbeitet werden, die eine Parallelisierung rechtfertigen)*[2]*;*
- *unterschiedlich tiefe Operatorbäume zur Kombination mehrerer solcher Eigenschaften.*

Alle Anfragen lassen sich auf dem Bucky-Teilschema spezifizieren und sind die folgenden:

I. Einstufige Anfragen

Q1. *einfache Anfrage, keine Selektion*

Finde die Adressen aller Personen aus Leipzig.

Q2. *Anfrage über Hierarchie, Selektion "oben"*[3]

Finde die Adressen aller Studenten aus Leipzig.

Q3. *Anfrage über Hierarchie, Selektion "unten"*

Finde die Adressen aller Studenten mit Matrikelnummern 200.000 - 500.000[4].

1. Dieses Kriterium kann umgekehrt durch geeignete Auswahl von Indizes bei gegebener Anfragelast erfüllt werden.
2. Erst im Mehrbenutzerbetrieb wird auch eine Vielzahl kleiner, stark selektiver Queries interessant.
3. Die Bezeichnungen "oben" und "unten" beziehen sich auf die Hierarchie
4. Selektion wird durch "Selectivity" in Prozent angegeben

II. Zweistufige Anfragen

Q4. *Anfrage über Hierarchie und Mengenattribut, keine Selektion*
Finde die Kinder aller Informatikstudenten.

Q5. *einfacher Join*
Finde alle Paare von Personen mit gleichem Geburtsdatum.

Q6. *Anfrage über Hierarchie mit Join "oben"*
Finde alle Paare von Mitarbeitern mit gleichem Geburtsdatum.

III. Mehrstufige Anfragen

Q7. *Anfrage über mehrere, mehrfach verzweigte Hierarchieebenen, keine Selektion*
Finde alle HiWis mit Name, Gehalt und Matrikelnummer.

Q8. *Komplexe Anfrage mit Selektion auf mittlerer Ebene*
Finde alle HiWis mit Name, Gehalt und Matrikelnummer, deren Gehalt unter DN 500/Monat liegt.

Q9. *Anfrage über mehrere, teils mengenwertige Referenzattribute mit Join*
Finde alle Paare von Studenten und betreuenden Professoren, bei denen der Professor Abteilungsleiter für das Hauptfach des Studenten ist.

3.1.4 Mögliche Bewertungsvarianten eines ORDBS

Im [CD96a] wurden zwei Kriterien für die Bewertung eines ORDBS vorgeschlagen. Der **Effizienzindex** wird eingesetzt, wenn die Performanz eines ORDBS im Vergleich mit einer von RDBS gemessen wird. Um den **Effizienzindex** auszurechnen, multipliziert man die Ausführungszeiten jeder Anfrageart zwischen einander und nimmt die ***n***-fache Wurzel aus dem Ergebnis, zunächst für ein ORDBS und dann für das entsprechende RDBS, wobei ***n*** die Anzahl der auszuführenden Anfragen ist. Abschließend wird das Ergebnis für ORDBS durch das von RDBS geteilt.

Das zweite Kriterium aus dem Benchmark war das **Powerrating. Powerrating** von Bucky dient dem Messen der absoluten Performanz eines ORDBS und ist stark an das Bucky-Schema und die dort beschriebenen Anfragearten gebunden. Die geometrische Mitte, die auch bei dem Kalkulieren des **Effizienzindexs** berechnet wird**,** wird auch als Grundlage für das **Powerrating** genommen. Da das **Powerrating** um so größer sein soll, je schnelle die Anfragen ausgeführt werden, wird hundert durch das erhaltene Resultat geteilt.

Für diese Arbeit ist der **Effizienzindex** nicht weiter relevant, da ihrer Schwerpunkt nicht im einfachen Vergleich zwischen zwei Typen von DBS liegt. Jedoch könnte das ein interessanter Untersuchungsaspekt werden, der mit Hilfe des *SimPaD*-Simulationssystems **Absch. 4.1.1**, das die Prototypen beider DBS-Varianten enthält, leicht zu bewerkstelligen wäre. Da alle Tests dieser Arbeit querie-weise für die Auswertung einzelner OR-Verfahren durchgeführt werden, findet auch das **Powerrating** keinen Einsatz in unserer Forschung.

Die Effizienzschätzung von Operatoren, die sich an der Verarbeitung ausgewählter Anfragen beteiligen, findet aufgrund der Statistiken statt, die unser Simulationssystem selbst liefert, **Kap. 5**. Alle Simulationskomponente wie Puffer, Platten oder Prozessorknoten werden mit den speziellen Funktionen versorgt, die am Ende jedes Simulationslaufes die Parameter wie CPU-Auslastung oder Trefferrate des Puffers bereitstellen. Die Realisierung der Statistiken wurde sowohl in den Modulen für Hardware-Komponenten des Simulationssystems als auch in dem Simulationstool “CSIM” fest verankert. Nach einem Simulationslauf werden die Ergebnisse in die Resultatsdatei umgeleitet, so daß für einen Nutzer keine Schwierigkeiten entstehen, die zu weiterer Analyse erforderlichen Werte abzulesen und mit Analogien aus den anderen Simulationsläufen zu vergleichen.

3.1.5 Zwei Alternativen zu der Bucky-Speicherung

In diesem Abschnitt werden zwei mögliche alternative Ansätze zur Speicherung von Bucky-Teilschema dargestellt. Am Beispiel der beiden Modelle wird gezeigt, wie unterschiedlich die möglichen Speicherungskonzepte für ein OR-Schema werden können. Einige Vorteile und Nachteile der Konzepte werden hier diskutiert und die endgültige Entscheidung, die zweite Variante mit einigen wenigen Ausnahmen für das Erstellen des Simulationsprogrammes als Muster zu nehmen, begründet.

In beiden Speicherungsvarianten wird jedes der vorkommenden Objekte beim Bedarf in zwei oder mehrere Teile zerlegt. Das erste Teil eines Objekts (als Kopfteil oder Instanz der ersten Klassenkomponente bezeichnet) enthält hauptsächlich nur Ausprägungen der Attribute begrenzter Länge. Die anderen Attribute wie *Sets, BOLBs* etc. werden ausgelagert, falls sie bei den meisten Anfragen nicht betroffen werden. Da die OIDs eine feste Länge besitzen, sind sie mit den Attributen des Kopfteils gleich zu behandeln.

Es ist jedoch manchmal möglich und nötig, auch einige weitere Attribute eines Objekts der ersten Klassenkomponente zuzuordnen. Das lohnt sich vor allem dann, wenn der Zugriff auf den Kopfteil eines Objekts fast ausschließlich das Lesen anderer Attribute erfordert. In solchen Fällen trägt der DB-Administrator die Verantwortung für die Entscheidung, wie und wo die einzelnen Attribute zu plazieren sind.

Die konzeptuelle Entscheidung, wie die einzelnen Komponenten plaziert und zusammengesetzt werden sollen, mußte auch im Rahmen unseres Simulationsmodells getroffen werden. Bei der Auswahl des Verfahrens hat man sich auf die Vorschläge aus der Literatur wie in [JM98], [HR99] gestützt. Dort wurden einige existierende Ansätze vorgestellt, wie verschiedene Objektsteile zusammengehalten werden können, **Bild 5**. Die Speicherung von einzelnen Klassenkomponenten kann sowohl *referenziert* als auch *materialisiert* sein. Man sollte dem Administratoren eines ORDBS ein Werkzeug in die Hand geben, womit er eine passende Verbindungsvariante für jede Klasse in dem DB-Schema wählen kann.

Im Allgemeinen können Komponenten 2, 3, etc. gemeinsam als Rest bezeichnet werden. Sie stellen das zweite, dritte etc. Teil einer Klasse dar und werden aus dem *Cluster* der Daten des Kopfteils entfernt. Auf den Rest wird durch Referenzen verwiesen. Andererseits hat man auch die Möglichkeit, ausgehend von den Instanzen der Restkomponenten das Kopfteil des Objekt zu referenzieren. Die Ergebnisse unserer Überlegungen sieht man auf **Bildern 6 und 7**. Das BLOB-Attribut *Picture* wird beispielweise bei der Klasse *Person* von den Attributen *id* und *name* getrennt und ausgelagert. Die Restteile einer Klasse bilden dann auch ihr eigenes *Cluster.*

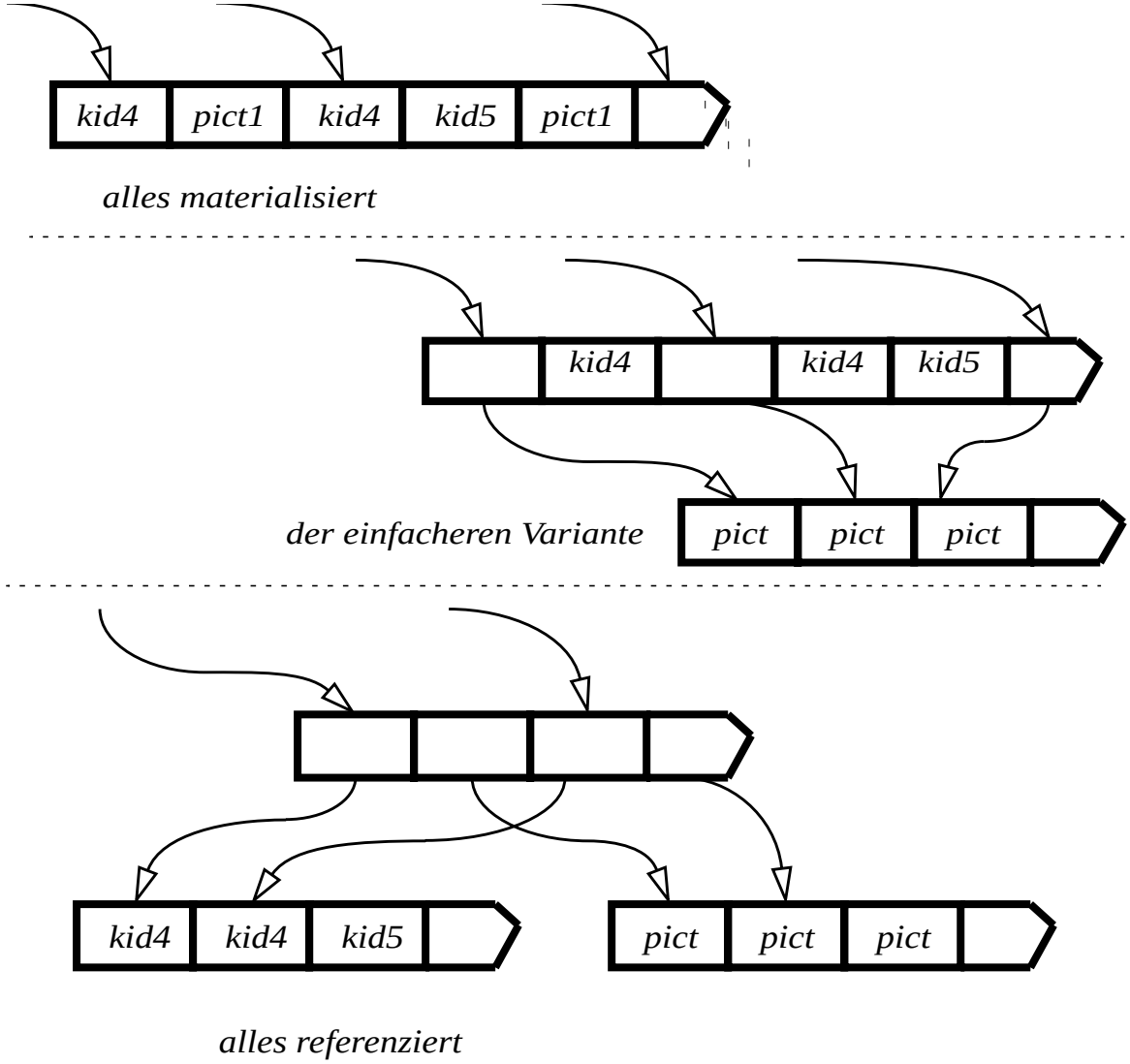


Bild 5. *Materialisierte und referenzierte Speicherung von Objekten*

Für Kollektionstypen wie *Sets* soll in jedem einzelnen Fall eine separate Entscheidung getroffen werden. Man könnte durch Einführen eines Kriteriums für eine durchschnittliche bzw. maximale Länge eines *Sets*[1] bei der Lösung ausgehen. Die Größe dieses Kriteriums kann mit mehreren Simulationsläufen ermittelt werden. Eine andere Alternative zur Behandlung der Kollektionstypen könnte durch Separieren einzelner Attribute eines Kollektionstypes in Gruppen entstehen (wie auf den Bildern, Attribut *Kids*). Die Ausprägungen der ersten Gruppe, deren gemeinsame Länge nicht ein bestimmtes Kriterium überschreiten soll, sollte in dem Kopfteil des Objekts bleiben. Die zweite sollte ausgelagert und mit dem Rest gleich behandelt werden.

1. Im Simulationssystem wird die Größe eines Kollektionstypes durch die Anzahl der Bestandteile und ihre durchschnittliche Länge vorbestimmt.

Die letzte Möglichkeit wird in dem Simulationsmodul nicht unterstützt, da in dieser Arbeit auf das Längelimit des Kopfteils ganz verzichtet wird. Es wird aber versucht, ausgehend von der Häufigkeit und Nützlichkeit der Attribute in den vordefinierten Quereis, eine logische Aufteilung der Objekte zu erzielen. In unserem Modell läßt man allerdings auch nicht zu, daß ein Kollektionstyp wie *Set* auf die gleiche Weise wie ein ganzes Objekt in zwei oder mehrere Teilen zerlegt wird.

Die erste Datenspeicherungstrategie auf **Bild 6** sieht vor, daß die Objekte, die durch Vererbungshierarchien miteinander verbunden sind, naturgemäß zusammengehalten werden. Dieser Ansatz geht aus dem sogenannten Molekül-Atom-Modell [HÄRD90] hervor, das besagt, daß Moleküle, welche aus zusammengehörenden Atomen (Objekten) bestehen und von vielen Anfragen gemeinsam verarbeitet werden sollen, als ungeteilte Granulate gespeichert werden. Bei dem Ansatz kann man sogar einige Speicherplatzreduktionen erreichen, da die Verweise für die Vererbungshierarchien nicht mehr gebraucht werden.

Diese Strategie ist jedoch für viele Anfragearten zu ineffizient. Der Nachteil des Modells liegt darin, daß die Objekte, die sich nicht notwendigerweise auf dem niedrigsten Niveau der Vererbungshierarchie befinden, auch die anderen Objekten in das Verarbeitungsverfahren mit einbeziehen, was folglich zu sehr hohem zusätzlichen Aufwand führt. Das heißt, daß für viele Anfragearten, die keine hierarchischen Beziehungen benötigen, die Performanz einfach schlecht werden kann.

Deshalb wurde sowohl aufgrund von verschiedenen Vorschlägen aus der Literatur ([Sto96], [EM99]) als auch unserer eigenen Überlegungen die zweite Variante des Speicherungskonzeptes vorgezogen, **Bild 7**. Diesem Konzept wird in dieser Arbeit auch weiter gefolgt, da es aus unserer Sicht mehr Flexibilität aufweist und weniger von den Arten der gestellten Anfragen abhängt. Darüber hinaus kann das Einlesen und das Ausschreiben von Objekten, die auf den verschiedenen Hierarchieebenen liegen, parallel erfolgen, was für parallele ORDBS ausschlaggebend ist. Die Kosten für das Einführen von zusätzlichen Referenzen, welche den hierarchischen Schemaaufbau unterstützen sollen, sind durchaus verträglich.

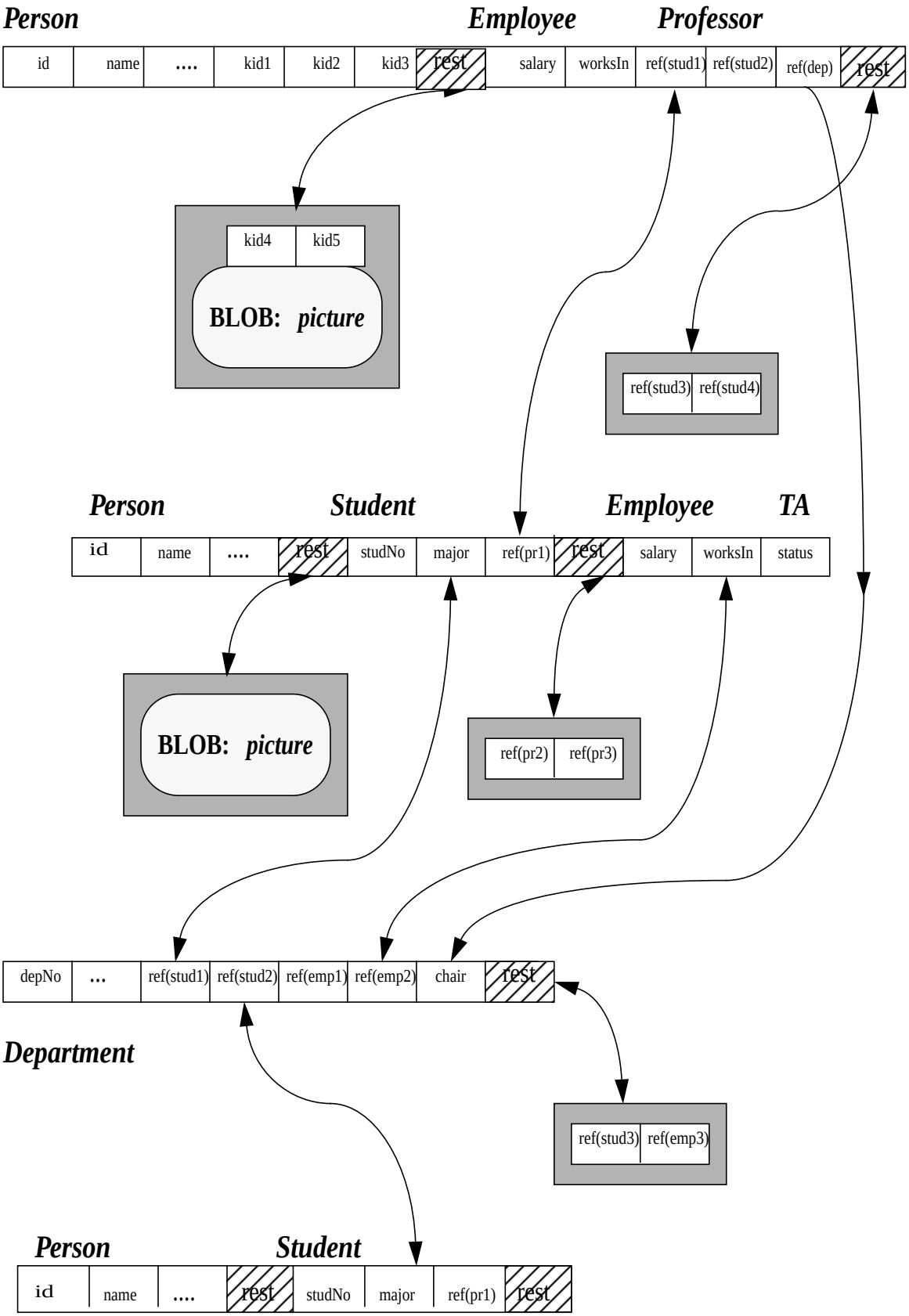


Bild 6. *Speicherung des Bucky-Teilschemas*

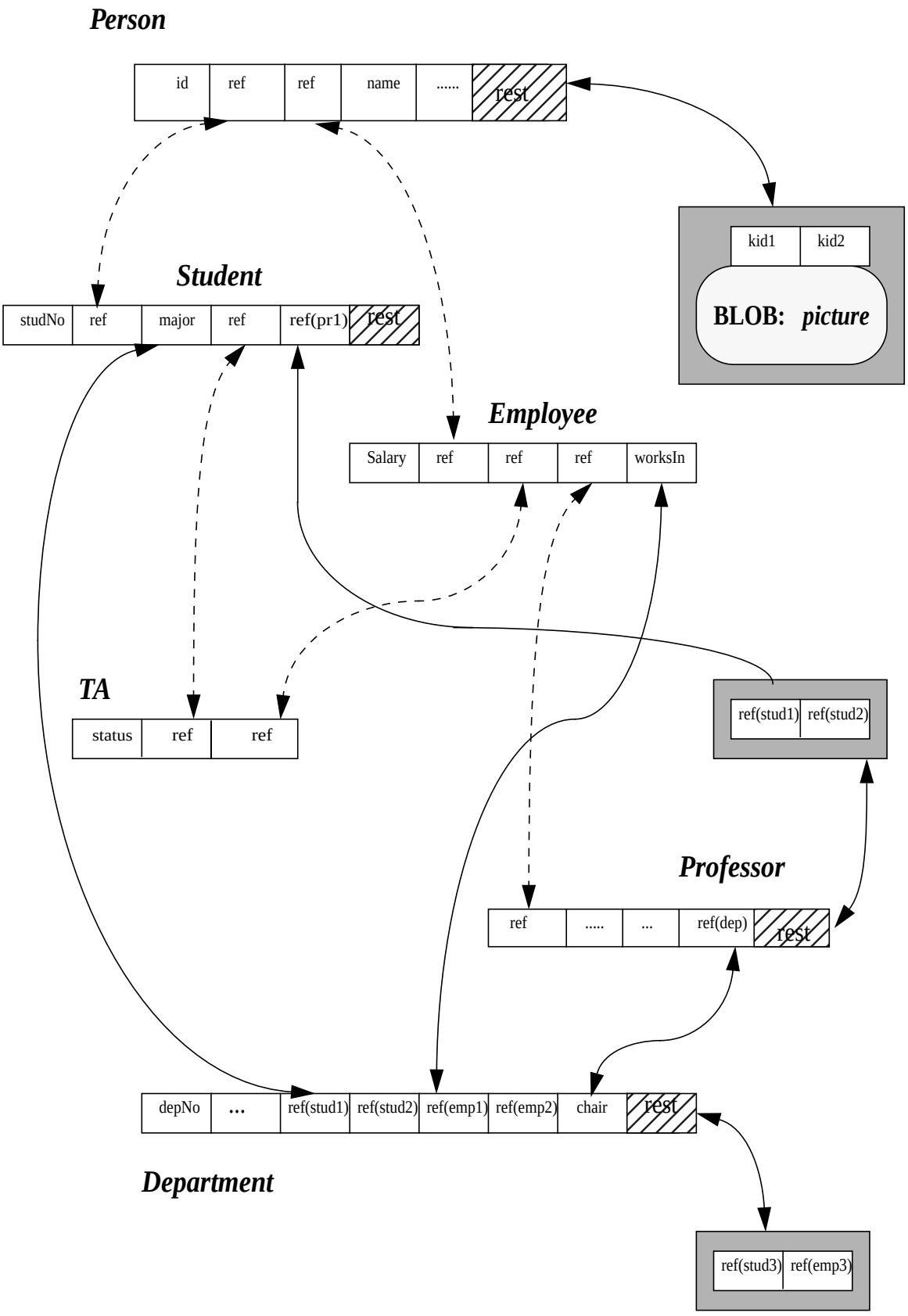


Bild 7. *Alternative Speicherung des Bucky-Teilschemas*

3.2 Verteilte Datenallokation von OR-Daten

Das Problem der Speicherung von OR-Daten in einem zentralen Fall wurde im vorigen Kapitell diskutiert. Wie die Zerlegung von Datenbeständen am günstigsten auf eine parallele OR-DB übertragen werden kann, damit die an das ORDBS gestellten Anforderungen in Erfüllung gehen, wird hier betrachtet.

3.2.1 Ziel und Arten der Datenpartitionierung in parallelen DBS

Wie es schon bereits erwähnt wurde **Absch. 2.2.2**, stellen die E/A-Vorgänge eine Art Bremsen eines existierenden DBS dar. Der Lese-Schreibkopf kann sich nur mit beschränkter Geschwindigkeit auf den Platten bewegen, so daß die Zugriffszeiten im Bereich von *ms* liegen. Wenn die Daten sich bereits in dem Hauptspeicher eines Prozessors befinden (noch besser in seinem *Cache*), wird die Bearbeitung lediglich einige n*s* dauern. Da es in vielen existierenden parallelen Systemen jedoch möglich ist, Daten aus den verschiedenen externen Datenquellen gleichzeitig zu lesen und den Prozessorknoten zwecks weiterer Verarbeitung bereitzustellen, versucht man die Datenbestände so horizontal zu partitionieren, daß sie auf mehreren Datenträgern (Platten) möglichst gleichmäßig in Bezug auf die Zugriffsmöglichkeiten verteilt werden.

Eine Gleichmäßigkeit bei dem Partitionierungsvorgang kann mittels eines *Round-Robin*-Verfahrens erreicht werden, so daß jede Platte die gleiche (ähnliche) Zahl der Objekte zu speichern bekommt. Eine einfache *Round-Robin*-Partitionierung liefert selten eine zufriedenstellende Lösung für die gesamte Datenbank. Das liegt vor allem daran, daß es genügend Anfragearten gibt, die nicht alle Objektinstanzen in die Verarbeitungsroutine einbeziehen, sondern nur einen kleinen Teil davon (sog. Selektive- oder Bereichsanfragen).

Die der Queryverarbeitung unterliegenden Objekte sollen in der Regel einige bestimmte Eigenschaften aufweisen bzw. einen begrenzten Attributwertebereich haben. Bei selektiven Anfragen kann die Zahl der zu scannenden Objekte im Falle einer *Round-Robin*-Allokation ohne Indexhilfe kaum reduziert werden. Auch mit Index sind meist alle Partitionen betroffen, was folglich einen gewaltigen Overheard verursacht.

Die *Hash*- und *Bereichs*partitionierungsmethoden versuchen diesen Nachteil abzuschwächen. Sie definieren die Datenverteilung über die bestimmten Werte eines Attributs bzw. einer Menge von Attributen [Ra94]. Die Gleichmäßigkeit bei der Datenverteilung kann auch hier erreicht werden, was im allgemeinen schwieriger ist als mit *Round-Robin*. Bei der Hash-Partitionierung wird die Zerlegung einer Relation durch eine Hash-Funktion auf dem Verteilattribut bestimmt. Die Bereichspartitionierung verwendet Wertebereichaufteilung auf disjunkte und vollständige Mengen, die sich auf das Verteilattribut anstelle einer *Hash*-Funktion beziehen.

Wenn man bei der Wahl zwischen den *Hash*- und *Bereichs*partitionierungsmethoden steht, soll analysiert werden, welche Typen im parallelen DBS am häufigsten vorkommen, um die geeignete Partitionierungsvariante für die Daten festzulegen. Eine Hash-Partitionierung unterstützt sehr gut die *Exact-Match*-Anfragen während die Bereichspartitionierung zusätzlich für die Bereichsanfragen geeignet ist.

Eine *Mehrdimensionale*-Datenfragmentierung, die eine erweiterte Form der Eindimensionalen-Bereichspartitionierung ist, versucht die Nachteile der o.g. Methoden abzuschwächen, indem die Tupel nach Bereichen von mehreren und nicht nur einem Verteilattribut partitioniert werden. Das führt folglich zu einer Erhöhung der Fragmentenzahl, welche die Anzahl der Platten weitgehend übertreffen kann. Die höhere Zahl von Fragmenten muß aber nichts schlimmeres für ein paralleles DBS bedeuten, da sie unabhängig von der Dimensionalität für die Verbesserung der Lastbalancierungsstrategien **Absch. 2.3.3, 3.4** verwendet werden kann.

3.2.2 Verweisbasierte-Mehrdimensionale-Fragmentierung von OR-Daten

Auch für ein paralleles ORDBS erweist sich eine *Mehrdimensionale*- zusammen mit einer passenden *Verweisbasierten*-Datenfragmentierung als nützlich. Die Fragmentierungsziele bleiben bestehen. Es wird immer noch versucht, durch eine geeignete Allokation den schnellsten parallelen Zugriff zu gewähren. Für ORDB-Systeme wird die allgemeine Allokationsaufgabe durch die Komplexität des Hierarchiebaumes und Vielfalt der in einem ORDB-Schema existierenden entgegengerichteten Verweisen erschwert. Die komplexen Beziehnungen zwischen Klassen bzw. innerhalb einer Klasse dürfen nicht außer Acht gelassen werden. Der in diesem Abschnitt vorgeführte Ansatz hilft dieses Problem zu lösen.

In der Arbeit wird der Vorschlag zu der *Verweisbasierten-Mehrdimensionalen*-Datenfragmentierung auf den Klassen des Bucky-Teilschemas kurz illustriert. Bei der Datenallokation werden nicht nur die Domäne der Verteilattribute einer Klasse berücksichtigt, sondern auch die Wertebereiche anderer Klassen, die mit der ersten in einer hierarchischen Beziehung stehen.

Nehmen wir an, daß man Instanzen der Klassen *Person*, *Student*, *Employee*, *TA* und *Professor* aus dem Bucky-Schema zwischen drei Platten zu verteilen hat. Die komplette Aufgabe der Allokation könnte daraus bestehen, außer Hierarchiebeziehungen zwischen den einzelnen Objekten auf die folgenden Verteilattribute der Klassen Rücksicht zu nehmen: die Studentennummer für die Klasse *Student* (kleiner und größer als 25.000) und das Gehalt für die Klasse *Employee* (kleiner als 5.000 DM; zwischen 5.000 und 50.000 DM; darüber).

Auf **Bild 8** ist eine prinzipiell mögliche *Verweisbasierte-Mehrdimensionale*-Datenfragmentierung für die Objekte des Schemas zu sehen. Man notiert, daß nach dieser Allokationsmethode eine universelle Plazierung von Objekten erreicht werden kann. Der Allokationsvorgang wird übersichtshalber in zwei Phasen aufgeteilt[1]. Als erstes werden die einzelnen Objekte aus den Super- und Sub-Klassen logisch zusammengesetzt und die Granulate für die Speicherung von Hierarchien gebildet[2] (*Verweisbasierte*-Fragmentierung).

Die Objekte, die einem solchen Hierarchiegranulaten gehören, werden dann möglichst gleichmäßig zwischen allen Datenträgern verteilt (Person1 -> Platte1, Student1 -> Platte2 **;** Person3 -> Platte1, Student3 -> Platte3, HiWi3 -> Platte2 etc.). Die zweite Phase wird dazu benötigt, um die bereits entstandenen Fragmente entsprechend den Wertebereichen ihrer Verteilattribute wie Gehalt für Mitarbeiter weiter aufzuteilen (einfache *Mehrdimensionale*-Datenfragmentierung*)*. Anders ausgedrückt wird das Ergebnis der ersten Fragmentierungsphase einem weiteren Partitionierungsvorgang unterzogen, der sich der bekannten *Mehrdimensionalen*-Datenfragmentierung für R-Daten ähnelt. Bei einer solchen Fragmentierungsstrategie verläuft das parallele Scannen ohne Konflikte unabhängig davon, ob die Anfrage sich auf das Zusammensetzen von Objekten in der Hierarchie oder auf die Verteilattribute der Objekte richtet.

1. Es ist möglich, die beiden Phase zu dem gleichen Zeitpunk auszuführen.
2. Das ist möglich, weil einem Objekt aus der Basis-Klasse genau ein aus der Sub-Klasse entspricht.

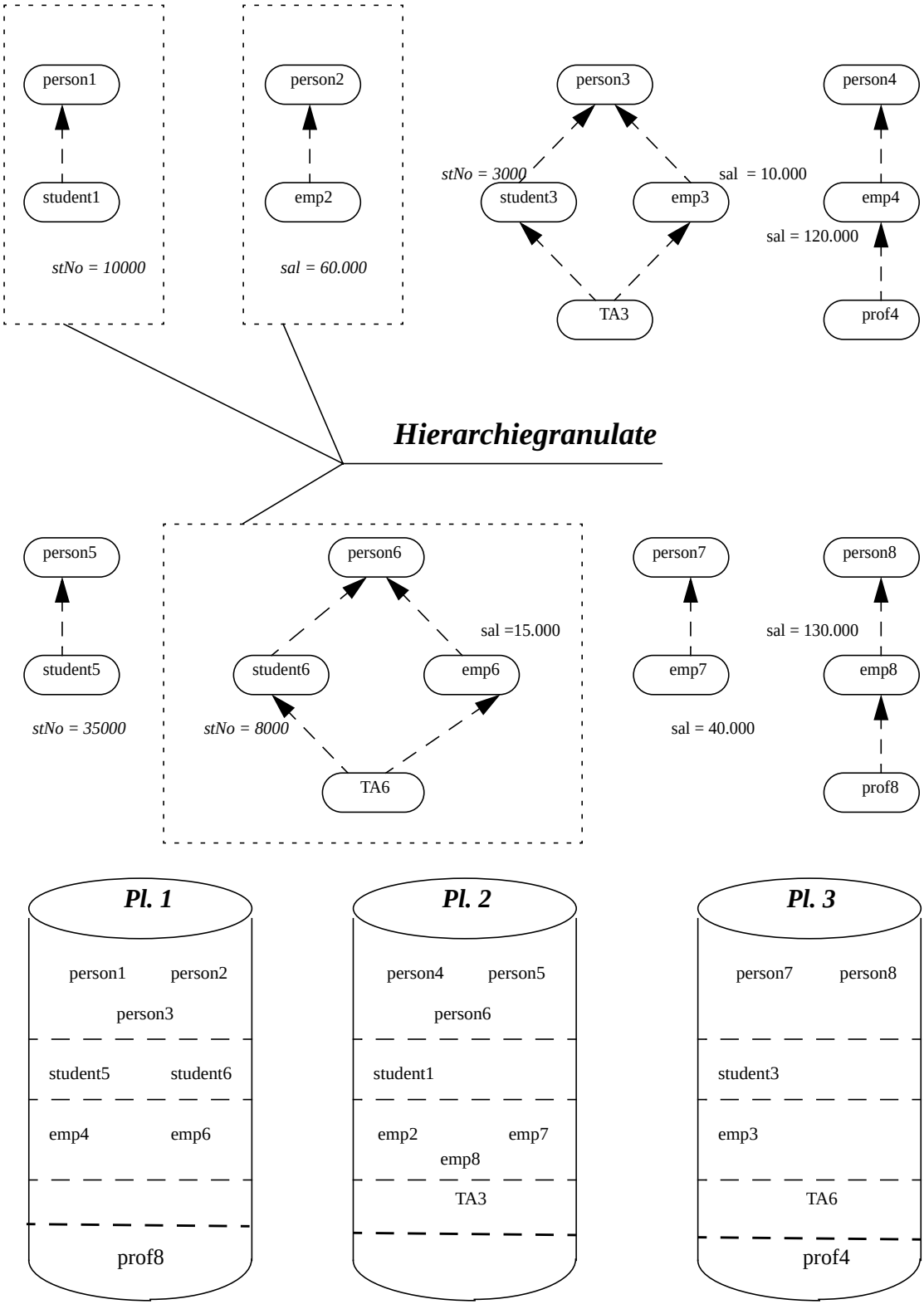


Bild 8. *Plattenbezogene Werteverteilung*

In dieser Arbeit wird keine Forschung der *Mehrdimensionalen*-Datenfragmentierung in dem OR-Simulationssystem durchgeführt, da die bereits im Rahmen mehrerer RDBS-Untersuchungen gründlich untersucht wurde [Ra94]. Dabei bleibt uns wesentlich mehr Zeit für das Testen der spezifischen Aspekte von OR-Algorithmen erspart. Außerdem wird es wesentlich leichter, die aus den Simulationsreihen erhaltenen Ergebnisse zu interpretieren. Die Untersuchung der Hierarchiestrukturen in einem ORDBS stellt sich hingegen als umfassendes Gebiet dar, wo die Forschungsarbeiten erst beginnen und nur wenige Erfolge vermeldet wurden. Dort haben wir unsere Bemühungen verstärkt.

3.2.3 Mögliche OR-Allokation *(Vorschlaganalyse)*

Nachdem im letzten Kapitel die Grundprinzipien einer *Verweisbasierten-Mehrdimensionalen*-Datenfragmentierung für OR-Daten grob erläutert wurden, wird in diesem Kapitel unser Vorschlag zum Erzeugen einer universellen OR-Datenallokation in Form eines Algorithmus präsentiert. Der Algorithmus erstellt eine eindeutige Zuordnung von Daten zu den Platten, wo die Objekte untergebracht werden sollen. Die Abhängigkeit zwischen der Plattenzahl und der Anzahl von Prozessorknoten wird in dem gegebenen Algorithmus mitberücksichtigt.

Im **Anhang 1** ist der C++-Code der vollständigen Version des OR-Allokationsalgorithmus zu sehen. Jede Komponente einer Klasse wird vor dem eigentlichen Allokationsvorgang einer Vorbereitungsroutine unterzogen, welche die Grenzen jedes Fragments zurückgibt. Die Anzahl der dabei entstehenden Fragmente bekommt man einfach als Resultat des Produkts ***n * (n - 1)***, wobei ***n*** die Anzahl der zur Verfügung stehenden Platten ist.

Dieses Ergebnis ist durch die Überlegungen über die mögliche gleichzeitige Ausführung von parallelen Scan-Operatoren entstanden, die zu einem gleichen Zeitpunkt auf mehrere Objekte bzw. Objektteile einer Klasse zugreifen können. Bei dem Faktor ***n - 1*** ist die maximale Anzahl von den Platten gemeint, die mit jeder Platte, auf der ein Teil des Objekts allokiert werden kann, nicht im Konflikt stehen. Dadurch sollten die meisten Konflikte zwischen verschiedenen Prozessoren sowie die Engpässe bei den Plattenzugriffen reduziert oder sogar ganz vermieden werden.

Diese Formel gilt sowohl für die Menge aller Platten als auch für jede ihrer möglichen Submenge. Manchmal kann es sinnvoll sein, die Platten zwischen den Prozessoren statisch - also vor dem Beginn des Verarbeitungsprozesses - aufzuteilen **Absch. 2.3.3**. Diese Möglichkeit wird in dieser Arbeit berücksichtigt und in dem SimOR-Simulationssystem **Absch. 4.1.3** implementiert. Bei der zweiten Variante der Datenallokation wird allerdings die Anzahl der Fragmente für jede Plattensubmenge separat berechnet. Dementsprechend wird der Parameter ***n*** als Anzahl der in der Submenge liegenden Platten angesehen.

Der Vergleich zwischen den beiden Allokationsschemata wird hier am folgenden Beispiel illustriert. Übersichtshalber wird die Anzahl von Objekten in dem Bucky-Teilschema um Faktor 1000 verkleinert. Wird angenommen, daß die Klasse *Student* (500 Instanzen von 0 bis 499), die von der Klasse *Person* (1000 Instanzen von 0 bis 999) abgeleitet ist, auf sechs Platten allokiert wird. Für Verarbeitung stehen dem System zwei Prozessorknoten zur Verfügung. Die Klasse *Student* befindet sich auf der zweiten Hierarchieebene, so daß eine Verschiebung in der Anordnung der Student-Objekte zu den Platten (sog. *disk offset oder shift)* die Werte zwischen eins und fünf annehmen kann[1].

Die Klasse *Person* soll als Hauptklasse[2] in der Hierarchie auftreten und wird deshalb nach dem Allokationsalgorithmus einer initialen Plattenverteilung unterliegen. Hier wird zunächst übersichtshalber ihre Fragmentierungsart gezeigt[3]:

Person1

disk 0	**disk 1**	**disk 2**	**disk 3**	**disk 4**	**disk 5**
0 - 32	166 - 199	333 - 365	500 - 532	666 - 699	833 - 865
33 - 65	200 - 232	366 - 399	533 - 565	700 - 732	866 - 899
66 - 99	233 - 265	400 - 432	566 - 599	733 - 765	900 - 932
100 - 132	266 - 299	433 - 465	600 - 632	766 - 799	933 - 965
133 - 165	300 - 332	466 - 499	633 - 665	800 - 832	966 - 999

1. Die Erklärung für den Verschiebungsgrad kommt später in diesem Abschnitt.
2. Als Hauptobjekte werden die Objekte der ersten Hierarchieebene bezeichnet.
3. Im Algorithmus wird keine mix-Funktion angewendet, wenn man bei einer ursprünglichen Datenallokation bleiben will.

Die Plattenzahl bei der ersten Allokation beträgt sechs. Daraus resultiert die Anzahl von Fragmenten: **6 * 5 = 30**. Die Allokationsvariante für die Klasse *Student*, die keine Rücksicht auf die Anzahl der Prozessoren im System nimmt, sieht anschließend folgendermaßen aus:

Student 1

disk 0	**disk 1**	**disk 2**	**disk 3**	**disk 4**	**disk 5**
300 - 332	**0 - 32**	**33 - 65**	**66 - 99**	**100 - 132**	**133 - 165**
433 - 465	466 - 499	166 - 199	200 - 232	233 - 265	266 - 299
566 - 599	600 - 632	633 - 665	333 - 365	366 - 399	400 - 432
700 - 732	733 - 765	766 - 799	800 - 832	500 - 532	533 - 565
833 - 865	866 - 899	900 - 932	933 - 965	966 - 999	666 - 699

Die zweite von der Prozessorknotenzahl abhängige Variante sieht man in der nächsten Tabelle. Die Strichlinie, welche die Platten in zwei Submengen zerlegt, ist nur eine mögliche Trennungsvariante. Ihre Position kann sowohl nach links als auch nach rechts rücken, abhängig davon welcher von der Prozessoren schneller ist und damit mehr Arbeit innerhalb eines gleiches Zeitfensters bewältigt.

Da jedoch von uns angenommen wird, daß alle in dem Simulationssystem vorhandenen Prozessorknoten gleich schnell sind, wird die imaginäre Trennungslinie für das Beispiel mit zwei Prozessoren automatisch in die Mitte gesetzt und spaltet hiermit die ganze Personen- bzw. Studententabelle in zwei gleichen Teile. Die Anzahl von Fragmenten ändert sich entsprechend. Die Variable **n** nimmt den Wert 3 an. Daraus resultiert die gemeinsame Fragmentenzahl: **3 * 2 + 3 * 2 = 12**.

Wie bei der ersten Allokation wird zunächst übersichtshalber die initiale Personentabelle (Komponente 1)[1] dargestellt:

1. Die Allokation für die einzelnen Objektskomponente ist anders als die des Kopfteils.

Person 2

	CPU 1			CPU 2	
disk 0	**disk 1**	**disk 2**	**disk 3**	**disk 4**	**disk 5**
0 - 82	166 - 249	333 - 415	**500 - 582**	666 - 749	833 - 915
83 - 165	250 - 332	416 - 499	**583 - 665**	750 - 832	916 - 999

Die zweite prozessorabhängige Allokationsvariante der Klasse *Student* ist in der nächsten Tabelle zu sehen:

Student 2

	CPU 1			CPU 2	
disk 0	**disk 1**	**disk 2**	**disk 3**	**disk 4**	**disk 5**
250 - 332	**0 - 82**	**83 - 165**	750 - 832	**500 - 582**	**583 - 665**
333 - 415	416 - 499	166 - 249	833 - 915	916 - 999	666 - 749

Nach dem Vergleich von den Ergebnissen der Allokationsstrategien notiert man als erstes, daß die Allokationswertebereiche der beiden beteiligten Klassen in allen vier Tabellen unabhängig von ihrer Objektzahl gleich sind und der Zahl der Objekte in der Hauptklasse entsprechen. Alle Objekte werden dann - unabhängig von ihrer Anzahl - durch ihre *PersonID* repräsentiert, die in der Hierarchie vererbt wird und auf allen Ebenen als Verteilattribut gilt und somit überall denselben Wertebereich hat.

Dieses Ergebnis ist auf die Komplexität von Hierarchiestrukturen zurückzuführen. Alle abgeleiteten Klassen wie *Student* bilden mit der Klasse *Person* zusammen ein logisches Gemeinsames. Die Hierarchiegranulate **Absch. 3.2.2** werden dann von vielen Queries nachgefragt. Folglich stellt es sich als nötig heraus, sich eine gemeinsame Kardinalität für alle Hierarchiegranulate auszudenken, die für den ganzen Hierarchiebaum allgemeingültig wird und der maximal vorstellbaren Instanzen-Zahl jeder Klasse im Baum entspricht. Da die Hauptklasse *Person* die o.g. Anforderungen komplett erfüllt, wird die Kardinalität des Verteilattributs *PersonID* dafür eingesetzt.

In beiden Allokationsvarianten wird *disk_offset* für die Klasse *Student* als eins genommen. Den Verschiebungsgrad kann man durch die Position des ersten Fragments in der Allokationstabelle feststellen. Im allgemeinen werden zwei oder mehrere Klassen mit diversen *disk_offset* allokiert. Das Zusammensetzten der Objekte, die zu den verschiedenen Klassen in dem Hierarchiebaum gehören, wird damit wesentlich erleichtert, was den Lesevorgang stark beschleunigen sollte. Bei dem Verschiebungsgrad gleich eins, findet man beispielsweise die erste Spalte der Allokationstabelle der Hauptklasse in der ersten Zeile der Tabelle der ersten abgeleiteten Klasse wieder[1].

Um das ganze Konzept noch ein mal zu verdeutlichen, wird als nächstes die Verschiebung zweiten Grades auf die Objekte der *Employee1*-Klasse angewendet und die daraus resultierende Allokationstabelle hier gezeigt:

Employee1

disk 0	**disk 1**	**disk 2**	**disk 3**	**disk 4**	**disk 5**
266 - 299	**133 - 165**	**0 - 32**	**33 - 65**	**66 - 99**	**100 - 132**
400 - 432	433 - 465	300 - 332	166 - 199	200 - 232	233 - 265
533 - 565	566 - 599	600 - 632	466 - 499	333 - 365	366 - 399
666 - 699	700 - 732	733 - 765	766 - 799	633 - 665	500 - 532
966 - 999	833 - 865	866 - 899	900 - 932	933 - 965	800 - 832

1. In der Tabellen werden diese Werte hervorgehoben.

Wie man sieht, liegt das letzte Fragment 133 -165 (erste Spalte initialer Allokation der Hauptklasse) nicht auf der ersten Platte, wie es im Falle der einfachen *Round-Robin*-Allokation gewesen wäre, sondern auf der zweiten. Das wird durch die Besonderheit des Algorithmus erreicht. Man notiert, daß diese Art der Allokation von Mitarbeitern sich auch nicht mit der von Studenten in Bezug auf die parallele Verarbeitung überschneidet, so daß alle drei Objekte ohne die dabei entstehenden Konflikte von mehreren Prozessoren gleichzeitig gelesen und zusammengeführt werden.

Es wird angenommen, daß für eine Anfrage einige HiWis mit ihren Gehalt, Matrikelnummer und Wohnort gefunden und ausgegeben werden sollen, **Bild 4**. Nach dem Lesen der HiWi-Objekte (die kleinste Klasse) stellt man fest, daß nur die Bereiche 133-165, 600-632, 933-965 betroffen sind. Diese Bereiche werden dann für alle drei Basis-Klassen seitens drei Prozessoren nacheinander gelesen, so daß keine Konflikte zwischen den Prozessorknoten bei den gleichzeitigen Plattenzugriffen auftreten:

Bereich 133-165 ->

Person-Objekte auf der Platte 0;
Student-Objekte auf der Platte 5;
Mitarbeiter-Objekte auf der Platte 1;

Bereich 600-632 ->

Person-Objekte auf der Platte 3;
Student-Objekte auf der Platte 1;
Mitarbeiter-Objekte auf der Platte 2;

Bereich 933-965 ->

Person-Objekte auf der Platte 5;
Student-Objekte auf der Platte 3;
Mitarbeiter-Objekte auf der Platte 4;

Es wird offensichtlich, daß die maximale Allokationsverschiebung nur den Wert der Plattenzahl erreichen kann. Hingegen kann ein Hierarchiebaum beliebig tief sein. Dadurch werden die Allokationsmöglichkeiten zunächst erschöpft und für die Bäume, welche diese Tiefgrenze unterschreiten, nicht ausreichend sein. Solche Anomalien kommen jedoch selten vor, weil die Anzahl von Platten im System meistens groß genug gewählt wird.

Falls doch diese Art des Konfliktes auftreten sollte, werden bei der Allokation einer tieferliegenden Klasse nur die Allokationsmöglichkeiten aller Vorfahren in Betracht gezogen, und nicht die von den Klassen, welche auf der gleichen Hierarchieebene mit der zu allokierenden Klasse liegen. Die Lücken werden mit den Verschiebungswerten von fehlenden (herausgenommenen) Klassen gefüllt. Als weitere und endgültige Maßnahme zum Beheben des Allokationskonflikts wird der Vergleich von Kardinalitäten der Vorfahren vorgeschlagen. Die zu allokierende Klasse wird so einen Grad der Verschiebung bekommen, der für alle anderen Objekte ihrer Vorfahren am wenigsten zum Einsatz kommt.

Ähnlich geht man mit den Komponenten einer Klasse um. Alle Komponenten einer Klasse bauen gemeinsam eine ganze Hierarchiestruktur ähnlich wie die Schemaklassen auf. Die vorgegebene Allokation der Klasse *Person1* ist nur für das Bestimmen der Fragmente für die erste Komponente der *Person*-Klasse geeignet. Die zweite Komponente bekommt eine ähnliche Allokation wie *Student1*, die dritte wie *Employee1* etc.

Am Ende der Diskussion über den Allokationsvorschlag muß man noch einiges bemerken. Der in diesem Abschnitt vorgestellte Algorithmus hat seinen Einsatz in dem OR-Simulationssystem gefunden und ist unserer Ansicht nach eine gute Voraussetzung dafür, die nicht proportionale Datenspeicherung für Hierarchieanfragen statisch auszugleichen. Die *Skew-Effekte* werden jedoch durch die dynamische Asymmetrie einzelner Anfragearten, durch starke Divergenz bei der Zahl der Klasseninstanzen, die auf den gleichen Hierarchieebenen liegen, sowie durch die unvergleichbare Zweigtiefe der Branchen des Hierarchiebaumes, die zu derselben Klasse führen, immer noch auftreten.

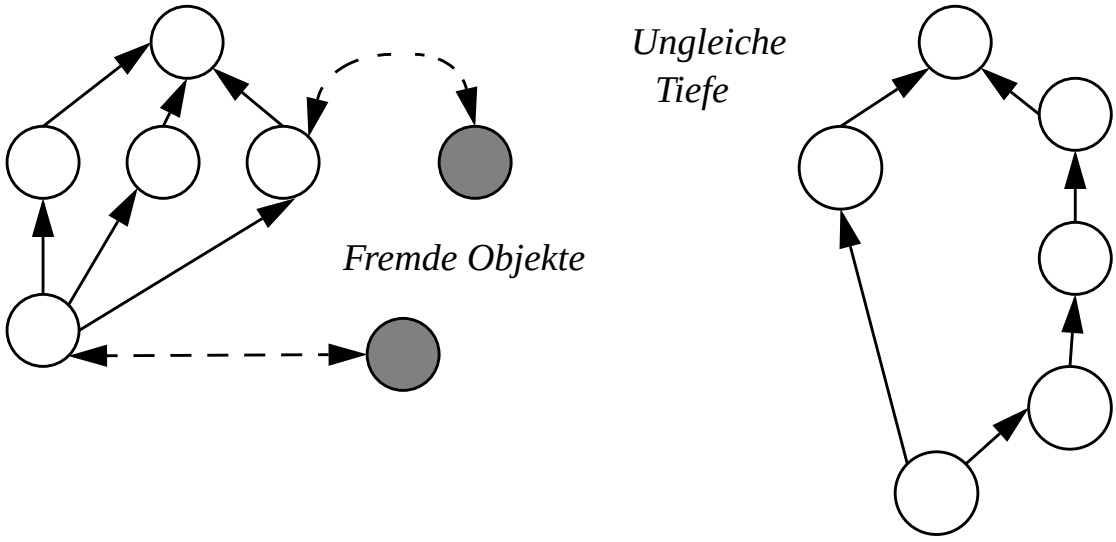


Bild 9. *Unregelmäßigkeiten bei der OR-Allokation*

Auf **Bild 9** sind zwei von den o.g. Anomalien skizziert. Die Objekte, die außerhalb von den zu bearbeitenden Hierarchiegranulaten liegen (sog. fremde Objekte), können die Bearbeitungsreihenfolge einer Hierarchieanfrage durch die Referenzzugriffe so beeinflussen, daß es keine Ordnung mehr gibt (das linke Teilbild). Bei solchen Störungen nutzt eine gute statische Allokation kaum noch.

Auf dem zweiten Teilbild sind die Unregelmäßigkeiten innerhalb einer Hierarchie zu sehen. Eine Klasse wird durch verschiedene Wege, die ungleich lang sind, im Hierarchiebaum erreicht. Es wird angennomen, daß die Hierarchietiefe einer Klasse dem längsten der Wege im Baum entspricht. Zwar helfen auch hier die vorgeschlagenen Allokationsmethoden eine ausgewogene Fragmentierung zu bekommen, es kann aber nicht behauptet werden, daß bei dieser Allokation keine *Skews* enthalten sind. Die Verfahren zur dynamischen Lastbalancierung und Scheduling **Absch. 3.4** sollten die o.g. Unregelmäßigkeiten, die durch die statische Allokation noch nicht verschwunden sind, bereinigen.

Der Algorithmusvorschlag, der in dieser Arbeit beschrieben wurde, ist nicht die einzige mögliche Variante zur Allokation von OR-Daten. Das Problem wird zur Zeit von mehreren Wissenschaftlern parallel zu uns gründlich erforscht. In den neuesten Werken wie [KL00], [Go00], unterbreitet man die Vorschläge zu den *Vertikalen* und *Horizontalen* Partitionierungsmethoden mit verschiedenen spezifischen Eigenschaften und Techniken. In [Go00] hat man sogar eine theoretische Grundlagen für das Messen der Effizienz einer *AHCP (Assocoated Horisontal Class Partitioning)* in einer *Data Warehousing* Umgebung geschaffen. *AHCP* ähnelt auch unserer Partitionierugsmethode. Jedoch haben die Autoren in [Go00] keine Funktion definiert, welche die horizontale Partitionierung von Objekten und die Zuordnung der Partitionen zu Platten eindeutig bestimmen sollte. Das wurde in unserem OR-Model konkretisiert.

3.3 Vergleich zwischen R- und OR-Operatoren

Nachdem die Entscheidung über die Allokation von OR-Daten gefallen ist und die Implementierung von komplexen Fragmentierungsmethoden abgeschlossen wurde, wurden die bekannten diversen Techniken zu der parallelen Verarbeitung von Daten untersucht. In dem Buch [Ra94] findet man die grundlegenden Aspekte der Realisierung einzelner paralleler Operatoren anschaulich dargestellt. In diesem Kapitel werden nicht nur die bekannten Verfahren für die Ausführung der Operatoren beschrieben, sondern versucht, eine Parallele zwischen den verteilten R- und OR-Operatoren festzustellen um zu sehen, inwieweit die bereits bekannten Methoden für Verarbeitung von R-Operatoren auf die OR-Systeme generisch übertragen werden können. Abschließend wird das *Referenz*-Join-Verfahren angesprochen, das in der R-Welt überhaupt keinen Äquivalent findet.

3.3.1. OR- vs. R-Scan und Sort

Der R- Scan gehört zu unären Operatoren, welche die einfachste Art aus der ganzen Operatorenpalette darstellt. Die zentrale Aufgabe eines Scan-Operatoren ist es, Daten so schnell wie möglich von den Platten zu lesen und den Prozessorknoten zwecks weiterer Bearbeitung zu übergeben. Das Scannen kann sogar in einem zentralen Fall auf verschiedene Art und Weise erfolgen abhängig davon, wie die zu lesenden Daten auf den Platten gespeichert sind und inwiefern sie dann für die weitere Verarbeitung benötigt werden.

Als erstes muß unterschieden werden, ob überhaupt alle Daten gelesen werden sollen. Die selektiven Anfragen s.o. benötigen keinen totalen Scan von ganzen Datenbeständen. Der Scan-Operator muß in der Lage sein, die zu lesenden Tupel (Objekte) während eines Scan-Vorganges zu selektieren und zu projizieren.

Falls die zu scannenden Daten mit Hilfe von Zugriffspfadinformationen wie B-Bäumen, B*-Bäumen, Hash-Tabellen etc. gespeichert wurden, sollte ein Scan-Operator diese Informationen entschlüsseln und mit ihrer Hilfe navigieren können. Grundsätzlich ist es möglich, die Zugriffspfadinformation in die Speicherungsstrukturen der Sätze einzubetten oder sie vollkommen separat zu speichern [HR99]. So wird beispielsweise zwischen *Clustered Index Scan* und *NonClustered Index Scan* unterschieden, die genau diesen zwei Prinzipien folgen.

Die weitere Information, über die man verfügen soll und die für die weitere Verarbeitung manchmal unentbehrlich ist[1], betrifft das Sortieren. Falls die zu lesenden Daten bereits sortiert sind, darf man nicht zulassen, daß diese wichtige Eigenschaft während eines Scan-Vorganges verloren geht. Eine explizite dynamische Sortierung erfordert wesentlich mehr Aufwand und kostet mehr Zeit. Falls sie für die weitere Verarbeitung jedoch nötig wird, muß man die dazu entstehenden Kosten bei der Effizientsschätzung der Gesamtanfrage miteinberechnen. Das Verfahren für das parallele Sortieren wurde in [Ra94] anschaulich beschrieben.

Die Frage, was man nun unter einem OR-Scan-Operatoren verstehen soll und welche Eigenschaften ein OR-Scan aufweisen soll, hängt wesentlich davon ab, wie man ein OR-Scan definiert. In der Literatur haben wir keine Hinweise gefunden, die auf diese Frage eine direkte Antwort geben könnten. Wie haben uns entschieden, eine eigene Definition zu erarbeiten.

Laut unserer Definition wird einem OR-Scan-Operatoren eine beliebige Anfrage zugeteilt, die im Rahmen einer SQL 3/99 Anweisung mit einem Satz von maximal drei Schüsselwörtern *SELECT, FROM* und *WHARE* beschrieben wird, wobei in der *FROM*-Klausel lediglich das Einfügen des Namens einer Klasse zugelassen wird. Darüber hinaus wird der Definition entsprechend das Verwenden von “Pfeil”-Notationen komplett verboten.

1. Als Beispiel ist Merge-Join hier zu nennen.

Nach unserer Definition soll ein OR-Scan alles können, was schon ein R-Scan gekonnt hat, d.h die Daten einer Klasse Lesen, Selektieren, Projizieren etc. Hinzu kommt noch die spezifische Eigenschaft, zwei oder mehrere Klassen in der Vererbungshierarchie zu vereinigen, was schon eher zu einem R-Join-Operatoren gehörte. Da nach unserem Allokationsvorschlag die Daten in der Hierarchie wesentlich leichter zusammenzusetzten sind, da sie bereits sortiert waren, wird diese zusätzliche Eigenschaft des OR-Scans nicht als Fähigkeit angesehen, Join-artige Anfragen komplett abarbeiten zu können. Wie die weiteren Überlegungen über die Join-Operatoren auch zeigen s.u., läßt sich der OR-Scan nur teilweise einem vollständigen Join-Verfahren zuordnen, nämlich *Sort-Merge*-Join.

3.3.2 Vergleich zwischen verschiedenen Join-Verfahren

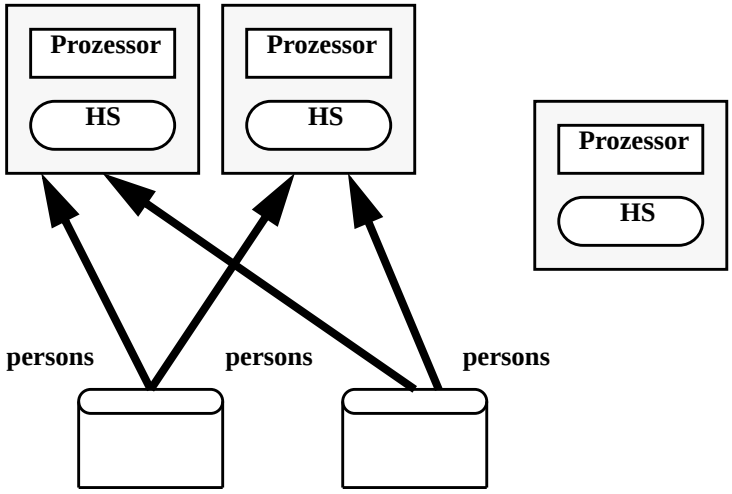
Als nächstes werden verschiedene Join-Verfahren unter die Lupe genommen. Der Join-Operator ist ein wichtiges Instrument bei der DBS-Anfrageverarbeitung und eignet sich hauptsächlich dafür, die Daten aus den verschiedenen Relation (Klassen) zu vereinigen[1]. Ein Join-Operator kann sowohl in binärer als auch in mehrdimensionaler Form vorkommen. Die drei wichtigsten Kategorien des Join-Operatoren, die sich nach dem Ausführungsplan grob unterscheiden lassen, sind *Nested-Loop*-, *Sort-Merge*- und *Hash*-Join.

Die *Nested-Loop-Join*-Technik ist die einfachste der drei und kommt folgendermaßen zustande. Als erstes nimmt man eine Relation (Klasse) als äußere Tabelle und geht alle dort enthaltenen Tupels (Objekte) der Reihe nach durch. Für jeden Tupel (Objekt) werden die Verbundpartner in der zweiten Tabelle (Klasse) ermittelt, indem man von jedem Tupel (Objekt) aus der zweiten Tabelle das Erfüllen der Anfragebedingung verlangt.

Der *Hash*-Join wird in zwei Schritten durchgeführt. Bei dem ersten Schritt der sog. *Build*-Phase wird die innere Relation (Klasse) in eine Hash-Tabelle im Hauptspeicher gebracht, wobei eine Hash-Funktion auf das Join-Attribut angewendet wird. Bei dem zweiten Schritt der sog. *Probe*-Phase wird die äußere Relation (Klasse) gelesen. Für jeden Tupel (Objekt) aus zweiter Relation (Klasse) wird wieder die gleiche Hash-Funktion auf das Join-Attribut angewendet, und geprüft, ob sich dort die passenden Verbundpartner aus der ersten Relation (Klasse) finden.

1. Als Spezialfall kann man hier die komplexe Zusammensetzung von Daten der selben Relation erlauben.

Hash-Join (Build-Phase) searching for kids



Hash-Join (Probe-Phase) searching for kids

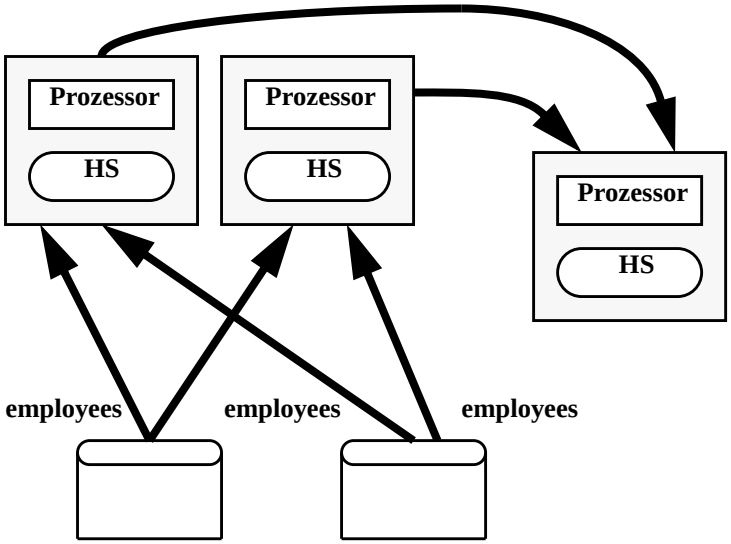


Bild 10. *Paralleler Hash-Join*

Die dritte alternative Technik ist *Sort-Merge*-Join. In dem Fall werden zunächst die beiden Relationen (Klassen) unabhängig von einander auf dem Join-Attribut sortiert. Dann wird das *Merge*-Verfahren gestartet, das Schritt für Schritt die beiden Relationen parallel durchgeht und den Verbundpartner für jeden Attributwert solange sucht, bis er ihm nicht mehr findet.

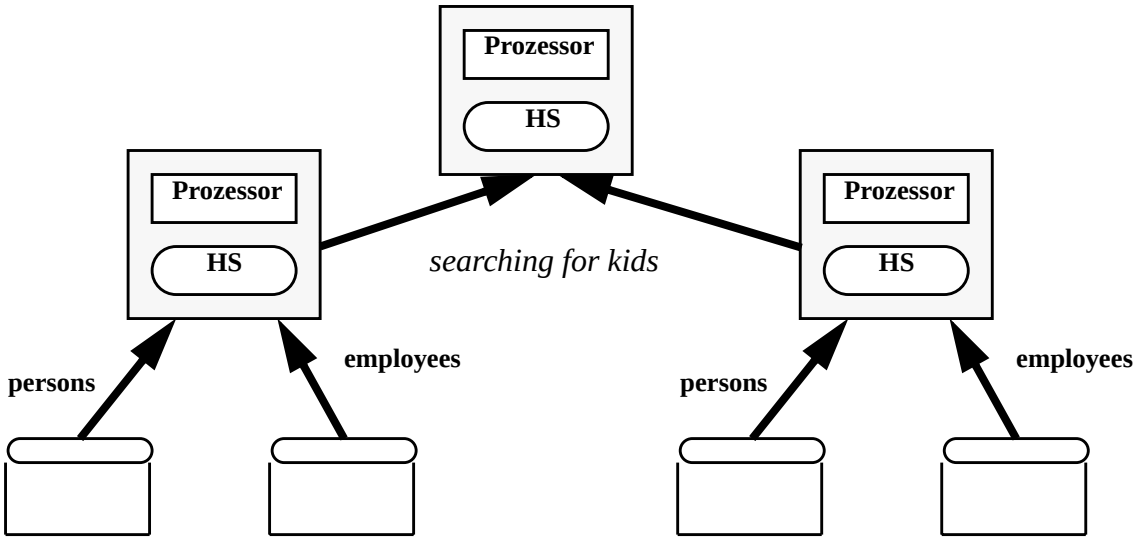


Bild 11. *Paralleler Merge-Join*

Die o.g. Join-Techniken bestimmen nur das Muster einer Ausführungsreihenfolge. Die genaue Betrachtung der Methode, wonach ein Join durchgeführt wird, hängt im wesentlichen von mehreren Komponenten und Feinheiten ab, die in dem konkreten Join-Algorithmus implementiert wurden. So lassen sich die Join-Operatoren aus der *Nested-Loop*-Kategorie bei ihrer Fähigkeit, mit Index-Strukturen umzugehen, unterscheiden. Die bekannten Variationen für den *Hash*-Joins-Algorithmus sind beispielsweise *CRACE*-Join, *Hybrid*-Hash, Hash-Join mit der *Bit-Vektor-Filterun*g etc.[HR96], [Ra94]. Die *Sort-Merge*-Join-Algorithmen können auch verschieden sein. Sie werden dann durch die Ausführungsart der Sort-Teiloperation sowie durch die Verhältnisse bei den Eingabedatenmengen unterschieden. Außer *Quicksort* kommen z.B. bei dem einfachen Sortieren im zentralen Fall die *Replacment-Selection*- oder *Polyphase*-Algorithmen in Frage [GLS94].

Alle o.g. Join-Verfahren können sowie zentral als auch parallel durchgeführt werden, **Bilder 10** und **11**. Das Untersuchen der parallelen Algorithmen ist ausschlaggebend für das Erstellen der OR-Simulation. Es ist beispielsweise sehr wichtig, die Zeiten für das Verschicken von Daten in Form von Nachrichten zwischen verschiedenen Platten und Prozessoren im Vergleich zu dem CPU-Aufwand, der für dieses oder jenes Join-Verfahren entsteht, korrekt einkalkulieren zu können. Außerdem war es wichtig festzulegen, auf welchen der Prozessoren im System eine weitere Operation stattfindet. Spezifisch für den parallelen Hash-Join wurden im System zwei Kollektortypen realisiert **Absch. 4.2.7,** die sich durch das Verschicken der ersten Relation (Klasse) - also ihre Replikation - für den Aufbau von Hash-Tabellen unterscheiden lassen.

Die Frage, welche von den Join-Algorithmen in unserem OR-Simulationssystem zu implementieren sind, haben wir uns auch gestellt. Die Antwort darauf wurde der Literatur entnommen [GLS94], [Gr94], [HR96], [Sto96], [Gr99]. Die Effizienzauswertungen mehrerer Join-Algorithmen werden in der Literatur seit einigen Jahrzehnten heftig diskutiert. Die Vorschlagspalette ist sehr breit, Schätzungen unterscheiden sind und weichen von einander ab. Jedoch war es nötig, die wichtigsten und aktuellsten Informationen aus den Werken herauszukristallisieren. Der Artikel [Gr99], der die jüngste Studie zu der Frage liefert, führt z.B. einen vollständigen Vergleich auf den Queries des bekannten TPC-D Benchmarks in einem zentralen Fall durch[1]. Unsere Aufgabe bestand also darin, die Argumente zum Thema zusammenzufassen, zu vergleichen und zu versuchen, das wichtigste aus der Diskussion herauszuziehen.

Das Fazit läßt sich meiner Ansicht nach in einigen Sätzen folgendermaßen formulieren:

a) falls eine Sortierung der Datenbestände bereits vor dem Algorithmusbeginn existiert, ist der *Merge-Join*-Algorithmus ohne Zweifel der schnellste [Gr94];

b) *Nested-Loop-Join* verliert gegen die beiden Alternativen bei den meisten Query-Arten [Gr99];

c) ein *Hash*-Join schneidet wesentlich schlechter ab, wenn die Datenverteilung auf dem Verteilattribut von der gleichmäßigen Verteilung stark abweicht [HR96];

d) falls es vorkommen sollte, daß für ein *Hash*-Join eine Überlaufbehandlung erforderlich ist, wird es zu einem totalen Performanzverlust führen [Gr94];

1. Alle Messungen fanden auf dem Microsoft SQL Server 7.5 statt, wo alle Vertreter der drei Join-Kategorien vorhanden waren.

e) ein *Sort-Merge*-Join schneidet wesentlich besser ab, wenn die beiden Eingaberelationen eine vergleichbare Größe besitzen [Gr94];

f) die DBS-*Anfrageoptimizatoren* befinden sich aktuell auf einem Niveau, so daß sie nicht immer in der Lage sind die eine oder andere Technik zu bevorzugen und damit richtig entscheiden, welcher der gegebenen Join-Algorithmen beansprucht werden soll, um eine bestimmte Anfrage am effizientesten ausführen zu können [Gr99].

Die o.g. Beschlüsse zusammen mit unseren eigenen Überlegungen über die Fähigkeit, in der für diese Arbeit vorgegebenen Zeit robuste Simulationsmodule zu erzeugen, ergaben, daß man sich bei den gegebenen Umständen auf die komplette Implementierung des *Sort-Merge*-Joins und *Nested-Loop*-Joins-Algorithmen durchaus verzichten kann. *Merge*-Join kann ohne Sortieren eingesetzt und vollständig in das Simulationsmodul integriert werden. Hiermit können die Datenbestände, die mit Hilfe unseres Allokationsalgorithmus geschickt partitioniert worden waren, so schnell wie möglich wieder zusammengesetzt werden. Die Implementierung eines *Hash-Join*-Operatoren wurde in zwei Varianten vorgenommen, nämlich mit und ohne *Replikation* der ganzen *Hash*-Tabelle.

3.3.3 OR-Referenz-Join

Der *OR-Referenz*-Join ist eine ganz neue Join-Art, der in RDB-Systemen nicht existierte. Die Funktionsweisen eines *OR-Referenz*-Joins ist sehr einfach: jeder einzelnen Referenz, aufgrund deren das Zussamensetzen von Objekten stattfindet, wird gefolgt, und damit die Verbundpartner ermittelt.

Die Aufgabe, die Vereinigung von Objekten mit Hilfe eines *OR-Referenz*-Joins zustande zu bringen, wird für den OR-*Anfrageoptimizatoren* durch die Entscheidung erschwert, welches der beteiligten Klassen als Ausgangsobjekt und welches als Zielobjekt für Verweise zu nehmen sind. Die allgemeine Vorgehensweise sieht vor, daß das ganze Ausgangsobjekt in den Speicher gebracht werden soll. Die Instanzen des Zielobjekts werden nur dann gelesen, wenn sie sich als passende Komponente eines Resultatssatzes erweisen.

Bei der Effizienzberechnung des ganzen Operatoren muß man davon ausgehen, daß bei jedem Zugriff eine Suche nach der entsprechenden Seitennummer stattfindet, was im allgemeinen zu Performanzverlusten führt. Anders formuliert, wird der *OR-Referenz*-Join um so besser abschneiden, je weniger Zugriffe für den gesamten Join nötig werden. Das Sortieren kann auch in diesem Fall seinen Beitrag zur Performanzverbesserung leisten, indem die durch *Prefetching* gelesenen Seiten bessere Trefferraten aufweisen. Das kann allerdings nur unter bestimmten Umständen möglich sein. Als Beispeil hierfür ist die Allokation für den Hierarchieaufbaum **Absch. 3.2.3** zu nennen.

Der *OR-Referenz*-Join kann als eine Alternative zu den anderen o.g. Join-Techniken vor allem dann in Frage kommen, wenn die Anzahl von Instanzen des Zielobjekts, die von den Platten gelesen werden, der Zahl aller Instanzen wesentlich unterlegen ist. Die CPU-Kosten sind in dem Fall gering und können der einfachen Zahl der gefolgten Verweise gleichgestellt werden. Die E/A-Kosten werden stark davon abhängen, ob die referenzierten Fragmente des Zielobjekts auf die gleiche Art und Weise wie die Seiten des Ausgangsobjekts sortiert sind und ob im System das *Prefetching* verwendet wird. Der Frage, unter welchen Umständen ein Referenz-Join eine bessere Performanz als die anderen Join-Operatoren aufweist und bei welchen Anfragearten man ihn einsetzen sollte, wird der **Absch. 5.3.** gewidmet.

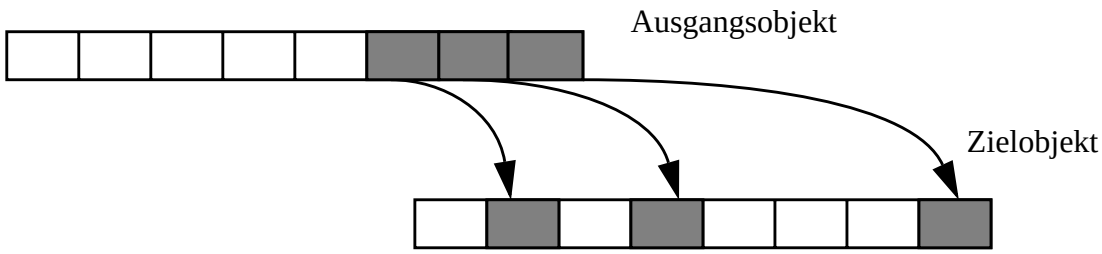


Bild 12. *OR-Referenz-Join (Sortiert)*

3.4 Dynamische Lastbalancierung

Die Notwendigkeit der Verfahren zur dynamischen Lastbalancierung wurde bereits im **Absch. 3.2.3** begründet. Unter der dynamischen Lastbalancierung in einem parallelen System wird vor allem eine effektive Nutzung von Rechnern und Platten während der Datenverarbeitung verstanden. Eine gute Datenallokation ist die grundlegende Voraussetzung für den Erfolg der Lastbalancierungsstrategien. Es ist aber nicht möglich, durch eine geeignete Datenallokation für jede Art der Anfrage, unabhängig von ihrer Komplexität und ohne Rücksicht auf den aktuellen Stand des Systems, zufriedenstellende Ergebnisse zu erzielen. Es soll zur Laufzeit entschieden werden, welche Komponente des Systems mit welchen Aufgaben beauftragt werden sollen.

Bei der Planung der Abarbeitungsreihenfolge einer Datenbankanfrage wird die gleichmäßige Auslastung aller Hardware-Ressourcen wie CPUs und Platten sowie die konfliktarme Verarbeitung konkurrierender Teilaufgaben angestrebt. Als Resultat soll die Verarbeitung des Anfragebaumes möglichst parallel verlaufen. Das läßt sich vor allem auf der Abarbeitungszeit der gesamten Anfrageverarbeitung sichtbar machen. Bei den guten Strategien zur Lastbalancierung einer Anfrage bleibt ihre Abarbeitungszeit am geringsten.

Für die fast-optimale Verteilung einer Menge von Aufgaben auf eine Menge von Prozessoren hat sich die LTP-Heuristik (*longest processing time first*) bewährt, wonach die Liste der Aufträge absteigend nach ihrer Größe geordnet wird. Dann werden Aufgaben vom Kopf der Liste beginnend an die Prozessoren verteilt, wobei die jeweils nächste Aufgabe an den Prozessor vergeben wird, der bisher insgesamt am wenigsten Arbeit zugewiesen bekommen hat. Andererseits führt man das Sortieren von Scan-Aufträgen so durch, daß die Zugriffe auf die gleichen Platten möglichst isoliert bleiben und sich nicht gegenseitig stören.

In dem OR-Simulationsmodul wird versucht [Mä00], die beiden o.g. Anforderungen unter einen Hut zu bringen. Das Sortieren der Scan-Aufträge wird während der Simulation nach der Formel s.u. durchgeführt, welche die Ziele der verschiedenen Lastbalancierungsstrategien in sich vereinigt und somit allgemein wird. Sie beschreibt eine Kostenfunktion, die das Gewicht eines Scan-Auftrags im Vergleich zu den anderen berechnen läßt. Laut dieser Formel hat ein Auftrag die oberste Priorität nur dann, wenn seine Kostenfunktion minimal ist.

$$M(j) = p(j)^{\alpha_p} \bullet \left(\alpha_l \bullet k(j, L) - \alpha_r \bullet k(j, R - j) - \alpha_c\right)$$

(min.)

Die einzelnen Parameter dieser Formel sind folgendermaßen zu interpretieren:

- ***M(j)*** ist die Kostenfunktion für den Scan-Auftrag ***j;***
- ***p(j)*** ist die Anzahl Seiten, die nach dem Scan-Auftrag ***j*** auf allen Platten gelesen werden sollen;
- **L** ist die Menge aller Aufträge, die aktuell bearbeitet werden;
- **R** ist die Menge aller Aufträge, dessen Bearbeitung noch nicht begonnen hat;
- **k** ist die gesamte Konkurrenz zwischen dem Scan-Auftrag ***j*** einer Menge der anderen Scan-Aufträgen;
- α sind die speziellen Konstanten, die für die Berechnung der Kostenfunktion nötig sind und Scheduling-Strategien bestimmen.

Die Einzelheiten zu den Definitionen der α-Parameter sowie anderer Parameter dieser Formel findet man im **Anhang 7.2**. Die Formel wurde so allgemein zusammengestellt, daß der Nutzer durch eine geschickte Auswahl der α-Parameter auf der *SimPaD*-Oberfläche (allgemeines Modul für die Anfrageverarbeitung) die Möglichkeit bekommt, die eine oder andere *Scheduling*-Strategie zu bevorzugen **Absch. 4.2.6**. Zu den *Scheduling*-Strategien, zwischen denen gewählt wird **Anhang 7.3**, gehören die folgenden:

- *Scheduling nach Größe **M1***[1] ;

$$M_1(j) = p(j) \qquad \textbf{(max.)}$$

- *Scheduling nach Konflikten mit laufenden Aufträgen **M2;***

$$M_2(j) = k(j, L)$$
$$M_2'(j) = k(j, L) \cdot p(j) \qquad \textbf{(min.)}$$

1. ***Mi*** sind die Abbreviaturen, womit die Schedulingsstrategien in der Anhangtabellen gekennzeichnet sind.

*- Scheduling nach Konflikten mit künftigen Aufträgen **M3**;*

$$M_3(j) = k(j, R - j)$$
$$M_3'(j) = k(j, R - j) \cdot p(j) \qquad \textbf{(max.)}$$

*- Scheduling nach Konflikten mit allen Aufträgen **M4.***

$$M_4(j) = M_2(j) - M_3(j)$$
$$M_4'(j) = \alpha_l \cdot M_2(j) - \alpha_r \cdot M_3(j) \qquad \textbf{(min.)}$$
$$M_4''(j) = \alpha_l \cdot M_2'(j) - \alpha_r \cdot M_3'(j)$$

Bei den Untersuchungen im **Kap. 5** wird die ***M2***-Kostenfunktion als Basis für die Berechnungen der *Scheduling*-Strategie eingesetzt, die sich nach unseren Experimenten als gut erwiesen hat. Nach dieser Strategie werden die entstehenden Scan-Aufträge nach ihren möglichen Konflikten mit bereits laufenden Scan-Operatoren stets bewertet und sortiert. Das Risiko, daß sich die am konfliktträchtigsten Aufträge am Ende der ganzen Verarbeitungsroutine stauen, besteht aber immer noch. Alle α-Parameter nehmen bei der Strategie den Wert Null an mit Ausnahme vom α_l-Parameter, der den Wert eins zugewiesen bekommt.

4. Wichtige Aspekte der Implementierung

Die ganze Implementierungsaufgabe dieser Arbeit besteht darin, die neue unabhängige *SimOR*-Simulation im Rahmen des *SimPaD*-Simulationssystems zu erstellen, womit man sämtliche Abläufe eines parallelen ORDBS mit einer *SD*-Architektur anschauen und abschätzen könnte. Die Implementierung wird daher in die folgenden Bestandteilen unterteilt:

- *Untersuchung der gegebenen SimPaD-Module, Oberflächenelemente und Datenstrukturen, die in die neue OR-Simulation aufgenommen werden können;*
- *Entwurf und Implementierung der Datenstrukturen für die Speicherung eines ORDB-Schemas;*
- *Entwurf und Implementierung des Allokationsalgorithmus;*
- *Erzeugen eines OR-Anfragegenerators;*
- *Übernahme und Anpassung der bereits existierenden Scan-Operatoren und Indexstrukturen an die OR-Daten;*
- *Entwurf und Implementierung eines gemeinsamen Scheduling-Verfahrens für alle Scan-Operatoren;*
- *Entwicklung und Implementierung des Konzeptes zur Abarbeitung einer beliebig zugelassenen Operatorbaumstruktur;*
- *Entwurf und Implementierung von Algorithmen zur Verarbeitung aller Typen der Operatorknoten, die anstelle der Blätter eines Operatorbaumes eingesetzt werden;*
- *Tests, Bug-Fixing und Tuning.*

In diesem Kapitel werden die wichtigsten inneren Besonderheiten des OR-Simulationsmoduls erklärt. Am Anfang verlieren wir einige Worte über die Umgebung, wohin das OR-Modul integriert wird. Dann wird Ursprung, Bestimmung und Funktionsweise der einzelnen Modulkomponenten skizziert. Alles zusammen sollte dem Leser einen guten Überblick über den Gesamtaufbau des OR-Simulationssystems sowie ihre Eigenschaften und Fähigkeiten geben.

4.1 Umgebungsuntersuchung und Integration

Das OR-Simulationssystem sollte nicht als unabhängiges allein stehendes Programm erstellt werden. Man kam zu diesem Schluß, nachdem die aus der Integration in das bereits existierende *SimPaD*-System entstehenden Vorteile und Nachteile verglichen worden sind. Einerseits verursachte die Integration des OR-Simulationsmoduls einen Overhead an Arbeit, die an mehreren Stellen bei der Anpassung an die existierenden Simulationskomponenten geleistet werden sollte. Andererseits konnten aber die Module, die auch für die anderen Simulationen verwendet wurden, ohne Leistungs- und Performanzverluste übernommen werden. Dies war ausschlaggebend, sich für die Übernahme der Integrationsaufgabe zu entscheiden.

4.1.1 SimPaD-Simulationssystem

Das *SimPaD*-Simulationssystem wird von den Mitarbeitern und Studenten der Abteilung DBS Universität Leipzig entwickelt, um verschiedene Konzepte und Charakteristika der parallelen DBS-Architekturen, der *SD*-Architektur unter anderen, zu erforschen. Einige Ideen zu der Implementierung dieses Systems gehen aus [St93] hervor. Das System läuft auf dem UNIX-Solaris Betriebsystem. Das Entwicklungslabor wurde mit den Sun-Rechnern ausgestattet. Die Module sind mittels C++-Programmiersprache geschrieben. Eine spezifische Simulationserweiterung für C++-Klassen wurde den Entwicklern in Form eines kommerzielles Produkts namens “CSIM” (beschrieben in [Me97] [Me97a]) bereitgestellt und an mehreren Stellen für die Abbildung von parallelen Prozessen verwendet.

Das System hat ein GUI, das *SimPaDManager* genannt wurde, **Bild 13**. Dort kann man die Parameter für Hardware-Komponente, sowie für die zu untersuchenden Verfahren bequem angeben und damit die ganze anstehende Simulation korrekt konfigurieren. Beim Starten des *SimPaDManagers* sind drei Fenster zu sehen. In dem linken Fenster sieht man die graphischen Icons für alle Module, welche in der zu untersuchenden DBS-Konfiguration vorhanden sind. Die Icons bilden eine baumartige Struktur zusammen, die den Namen des ganzen *SimOR*-Hauptmoduls an der Spitze hat. Mit einem Klick läßt sich jede Icon in dem zweiten oberen Fenster öffnen. Dort wird die Parameterliste für die gewählte Simulationskomponente erscheinen und mit den vorgegebenen Werten belegt. In dem dritten Fenster kann man eine Simulation zum Laufen bringen.

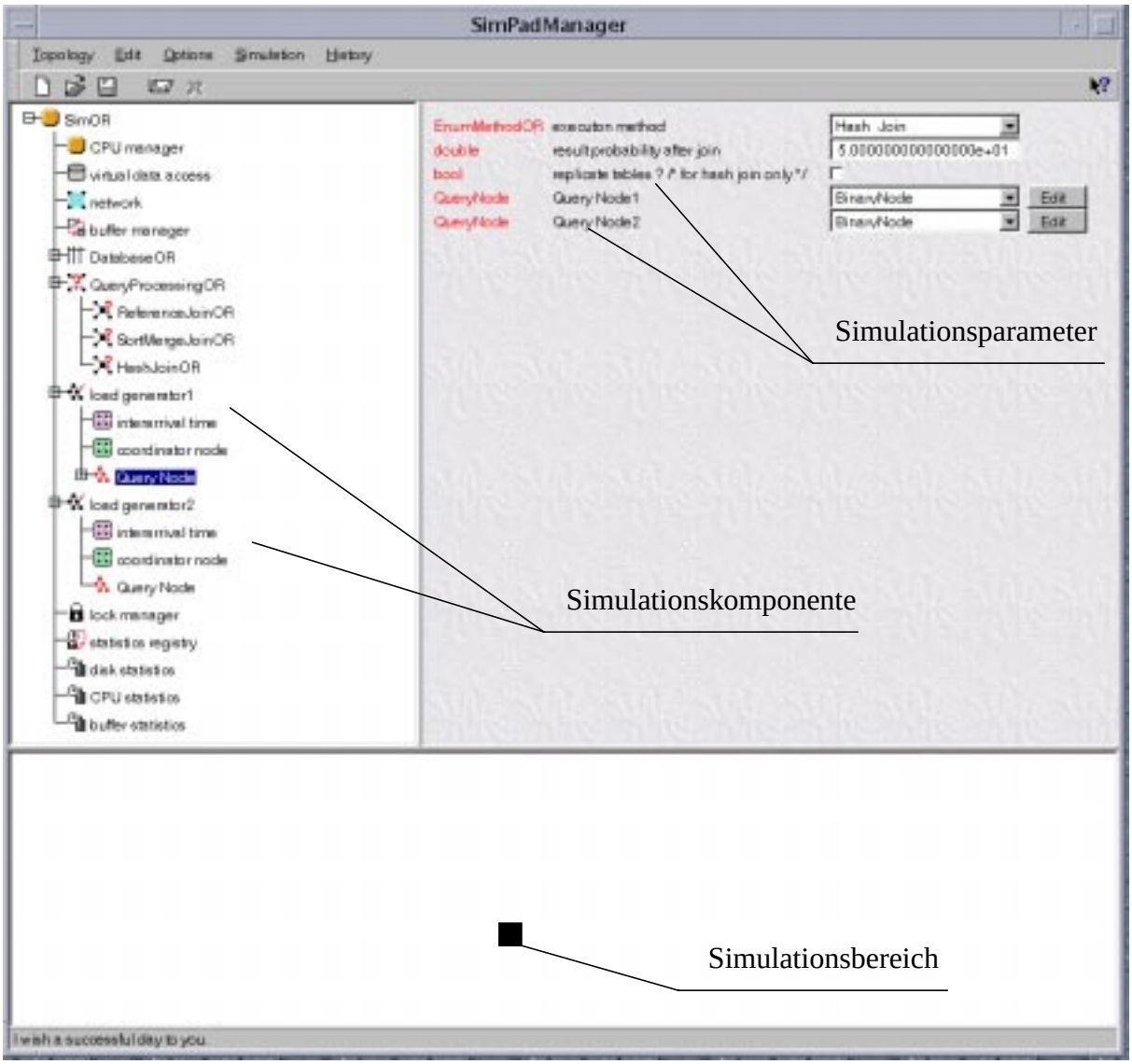


Bild 13. *Hauptfenster des SimPaD-Managers*

Es besteht auch die Möglichkeit, eine Simulation im Hintergrund zu starten, so daß ein neuer UNIX-Prozeß dabei erzeugt wird. Eine Simulation kann auch von der Kommandozeile gestartet werden. Das passiert, wenn die in dem *SimPaD*-Manager integrierte Anwendung namens *Csim-MiniClient* mit der gewünschten Topologie als Parameter aufgerufen wird.

Damit man mit dem inneren Aufbau des *SimPaD*-Managers zurecht kommt und die Module für das *SimPaD*-Simulationssystem erfolgreich weiter entwickelt, sollten beim Einstieg in die Arbeit alle geschriebenen internen Papiere sorgfältig gelesen werden. Die wichtigsten von denen sind [Ja98a], [Ja98b]. Außerdem war es notwendig, mehrere der vorhandenen komplexen Quelldateien schrittweise zu untersuchen, um an die dahinter liegenden komplexen Ideen und Gedanken heranzukommen.

4.1.2 Elemente des SimPaD-Managers

Bei dem Kreieren neuer *SimOR*-Komponenten auf der Oberfläche des *SimPaD*-Managers werden die Klassen verwendet, deren Beschreibung man in [Ja98b] findet. Diese Klassen erlauben sowohl das Erzeugen der neuen graphischen Elemente auf der Oberfläche z.B. Icons und Aufschriften, als auch das Plazieren der Felder für die Parameterangabe. Auf der Oberfläche des *SimPaD*-Managers wird die Parameterangabe aller wichtigsten Standarddatentypen generisch unterstützt.

Darüber hinaus wird einem die Möglichkeit gegeben, einige bisher unbekannte Aufzählungstypen und Arrays von Elementen für die Simulationsmodule zu spezifizieren. Dafür werden die neuen C++-Klassen gebraucht. Sie sollen auf besondere Art und Weise angelegt und dann in bestimmten Dateien angemeldet sein, um dem ganzen System später bekannt zu werden. Die wichtigste Bedingung für das Erzeugen eines neues Oberflächenelements im *SimPaD*-Manager blieb immernoch die Erbschaft von der Klasse *StandardSimObject,* die dem *SimPaD*-Manager allgemein bekannt ist und alle notwendigen Grundvoraussetzungen einer Oberflächekomponente bereits erfüllt.

Alle o.g. Möglichkeiten wurden bei der Konfiguration des OR-Simulationsmoduls in Anspruch genommen. Ein typisches Beispiel einer Aktion beim Erzeugen der *SimPaD*-Manager-Elemente war das Kreieren eines neuen Aufzählungstyps für alle Arten der Operatoren, woraus eine Anfrage bestehen kann. Das zweite mögliche Beispiel hierfür stellt das Anlegen eines Parameters der Simulation in Form eines Arrays **Bild 14** dar.

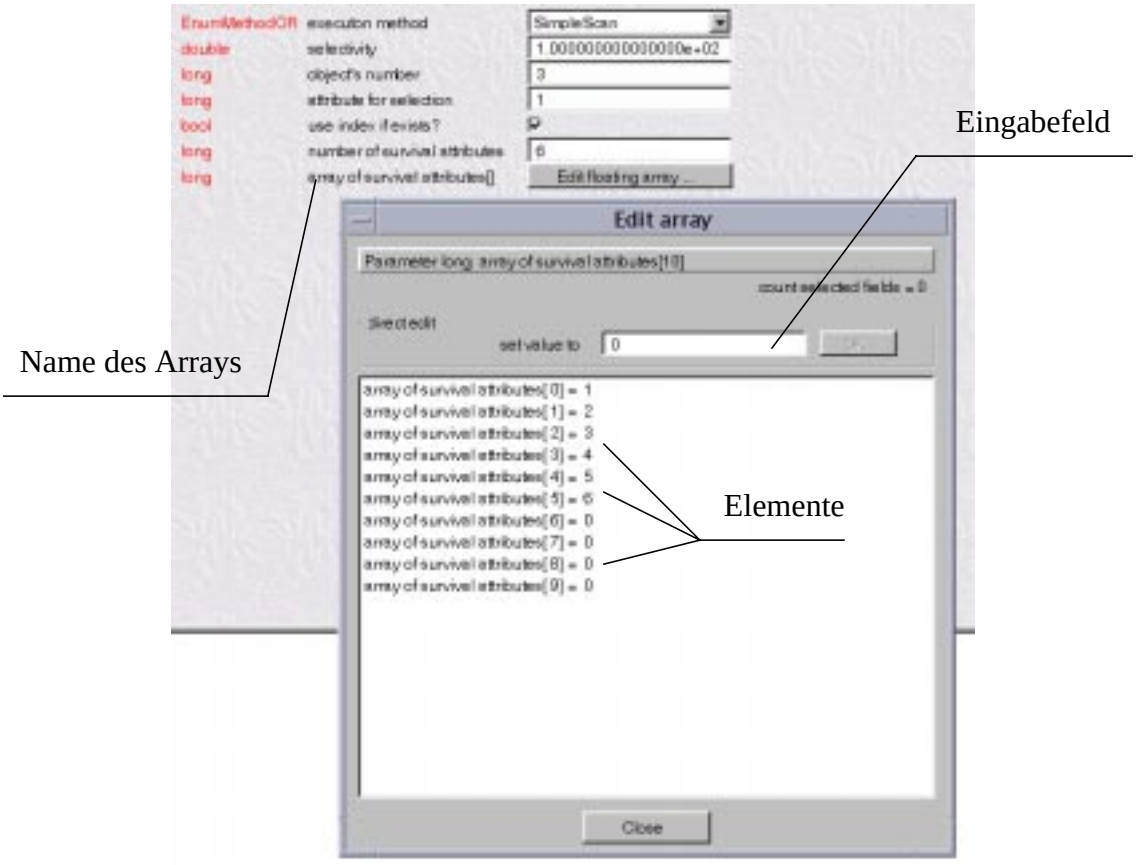


Bild 14. *Konfigurieren eines Arrays im SimPaD-Manager*

Nicht nur die Elemente des *SimPaD*-Managers erforderten eine geeignete Parametrisierung. Einige wenige Konfigurationsparameter, von denen die Stärke der Simulation sowie ihre Ablaufgeschwindigkeit direkt abhängt, sind auch in dem Quellcode zu finden, da sie dort leichter zu manipulieren sind. So läßt sich z.B. die maximale Zahl der Schemaobjekte und Schemaattribute durch die Änderungen der Konstanten in der Datei *DatabaseOR.h* variieren[1].

Die allererste Parameteranalyse erfolgt gleich bei der Parameterangabe. Die Parameter dürfen nur innerhalb eines Wertebereiches liegen, der im Programm durch die Bereichsgrenzwerte sowie ihr Format festgelegt wird. Die weiteren Parameterprüfungen finden während der eigentlichen Anfrageverarbeitung statt, wenn die Nutzerangaben einem Konsistenztest laut der vorgegeben Konfiguration des Gesamtsystems unterliegen.

1. Ein paar weitere wichtige Konstanten findet man allerdings dort auch.

Zum Schluß sind noch einige Worte über die Übergabemöglichkeiten der erzeugten Simulationskomponenten im *SimPaD*-Manager zu sagen. Sobald der Parametrisierungsvorgang einer Simulation komplett abgeschlossen ist, werden die einzelnen Komponenten des Systems nacheinander kreiert. Zunächst erzeugt man die Hardware-Module der Simulation **Absch. 4.1.3**. Erst dann kreiert man die weiteren Objekte für das Generieren von Anfragen und die Abarbeitung von Querybäumen (*LoadGeneratorOR, QueryProcessingOR* **Absch. 4.2.2, 4.2.5, 4.2.6** u.s.w.), die gemeinsam den Kern des Simulationssystems bilden. Ihnen wird die spezielle Routine *getParameterFromParent* seitens des *SimPaD*-Managers bereitgestellt, womit man die Verweise auf die Instanzen der Hardware-Module den anderen Modulen übergibt und in die übrigen Simulationsobjekte einbindet[1].

4.1.3 Die gegebenen SimPaD-Module

Einige Module, welche für das Erstellen der *SimOR*-Simulation benötigt werden, existierten bereits in dem *SimPaD*-Simulationssystem. Sie werden auch von anderen Simulationen genutzt. Die wichtigsten davon sind die Prozessor-, Puffer-, Datenzugriffs[2]- und Netzwerk-Module. Sie werden komplett in die OR-Simulation übernommen. Ihre Schnittstellen werden auf den HTML-Seiten über die Gestaltung der Simulationsmodule[SD00] in allen Einzelheiten beschrieben, so daß bei ihrer Übernahme kaum Probleme entstehen konnten.

Dem CPU-Modul liegt eine *CSIM-Facility*-Klasse zugrunde [Me97]. Der Funktionsaufruf, womit die Hauptsimulation den Prozessorknoten anspricht, hat eine gewisse Verzögerung in der Simulationszeit zur Folge, deren Dauer abhängig von der aktuellen CPU-Auslastung sowie der Anzahl der Instruktionen, die auf dem Rechnerknoten auszuführen sind, gemacht wird. Nach ähnlichen Prinzipien agiert das Netzwerk-Modul, wo die Nachrichtenschlangen in Abhängigkeit von der Belastung einzelner Verbindungskanäle gebildet werden. Die Verzögerung, die durch die Lesevorgänge verursacht und beim Datenzugriffsmodul registriert wird, kommt durch die zahlreichen

1. Unter dem *Parent* ist hier wieder die Klasse *StandardSimObject* zu verstehen, von der jede Klasse eines *SimPaD*-Oberflächenelements direkt oder indirekt abgeleitet wird.
2. Der Prozessor-Modul trägt in dem SimPaDManager den Namen "*CPU Manager*", der Puffer-Modul heißt "*Buffer Manager*", die Netzwerk- und Datenzugriffsmodule heißen jeweils "Network" und "*Virtual Data Access*".

Seitenzugriffe auf die Platten, sowie durch die Zeit, den Lese/Schreib-Kopf auf einer Platte zu bewegen, zustande.

Die wichtigsten Funktionsaufrufe, mit denen man sich ans Buffer-Modul wendet, sind *allokate*, *deallocate, fix* und *unfix*. Die *allokate*- und *deallocate*-Funktionen beanspruchen die Seiten eines Puffers gruppenweise. Diese Aufrufe sind meistens für die Speicherung von Zwischenergebnissen und das Anlegen der Hash-Tabellen nötig. Die *fix*- und *unfix*-Funktionen benötigen als Übergabeparameter eine *PageID*-Struktur (passive C++-Klasse[1]), die jede einzelne Seite durch ihre Zugehörigkeit zu einer bestimmten Plattenpartition sowie ihre innere Seitennummer innerhalb der Partition eindeutig identifizieren läßt.

```
class pageID {
    public:
        dbfile*     f;      // pointer to the disk the page belongs to
        long        p;      // page number inside a disk partition
};
```

Falls eine Seite bereits im Puffer liegt, kommt es bei einem Zugriff auf sie zu keiner Simulationszeitverzögerung. Wenn jedoch eine Seite im Puffer nicht gefunden wird, findet folglich ein Zugriff auf die Platte statt, was zur Verzögerung führt, welche seitens des Datenzugriffsmoduls eindeutig bestimmt wird. Auf **Bild 15** ist das Zusammenspiel der hier beschriebenen Hardware-Module mit dem *SimOR*-Modul zu sehen.

Alle in diesem Abschnitt erwähnten Module spiegeln die Bestandteile einer echter *SD*-Architektur wieder. Da die OR-Simulation auch keine Ausnahme bezüglich der DBS-Architektur zu den anderen in dem *SimPaD*-Simulationssystem vertretenden Simulationsarten macht, werden die Module für Hardware-Komponenten aus früheren Versionen ohne weitere Änderungen übernommen. Die anderen Simulationskomponenten wie DB-Schema-, Anfrageverarbeitungs- und Lastgenerierungsmodule sind für die OR-Simulation spezifisch und werden entweder geändert oder sogar ganz neu geschrieben.

1. Hier und später werden die C++-Klassen als passiv bezeichnet und immer durch das Wort Struktur ersetzt, wenn sie keine Methoden beinhalten, die für die Funktionsweise des Gesamtsystems wichtig sind.

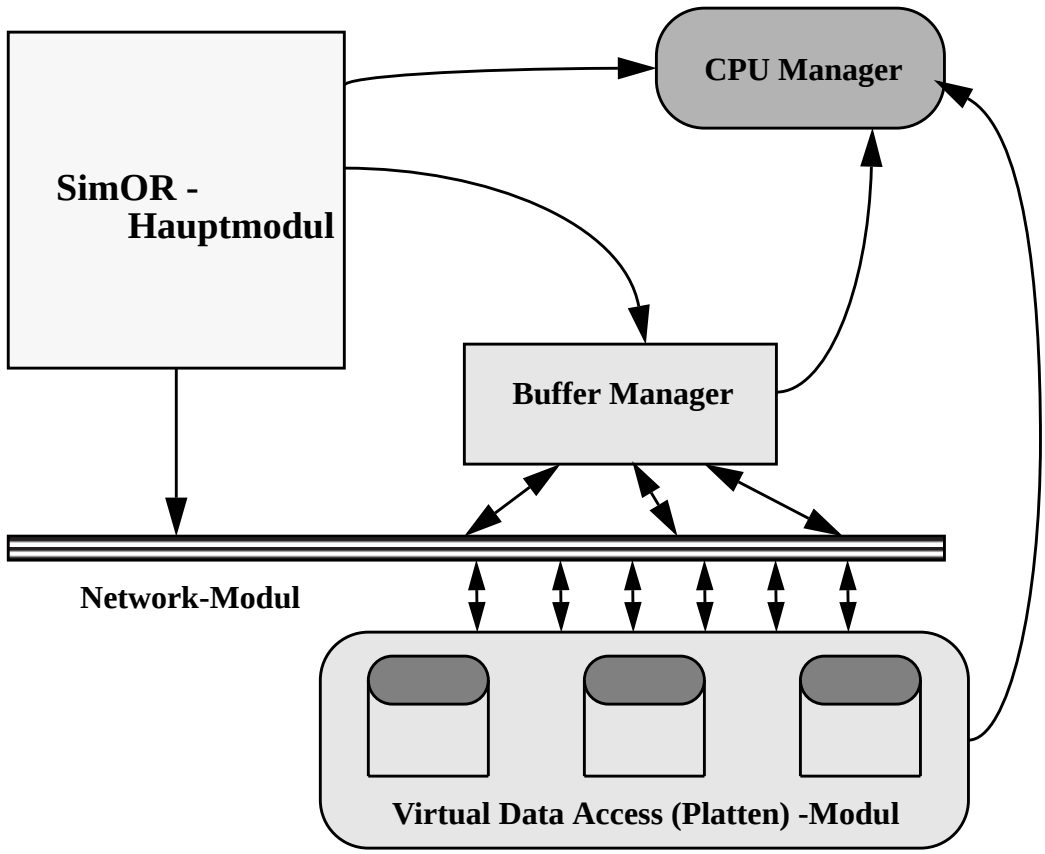


Bild 15. *Das Zusammenspiel der Module*

4.2 Implementierungsbesonderheiten der OR-Simulation

In den folgenden Abschnitten werden die wichtigsten Aspekte der Implementierung von Modulen skizziert, welche den spezifischen Kern der OR-Allokation und Anfrageverarbeitung bilden. Wir beginnen mit der Erklärung der Klassen und Methoden, welche die Abbildung des OR-Schemas auf die inneren Datenstrukturen ermöglichen. Dann gehen wir zur Beschreibung der Gesamtorganisation der OR-Anfrageverarbeitung sowie einzelner OR-Operatoren über. Das Kapitel endet mit der Beschreibung der Module, welche für die Übertragung des Datenflusses in dem Querybaum eingesetzt werden.

4.2.1 Implementierung des ORDB-Schemas

Der OR-Allokationsalgorithmus wurde im **Absch. 3.2.3** bereits beschrieben. Dem OR-Schema wird jedoch zunächst eine passende Klassenhierarchie zugrunde gelegt, wo alle von dem Nutzer auf der *SimPaD*-Oberfläche angegebenen Parameter für Schemaklassen und ihre Attribute dynamisch gespeichert werden. Diese Klassenhierarchie, die hauptsächlich aus drei Klassen gebildet wird, nämlich *attributeOR*, *relationOR* und *DatabaseOR*, dient als Behälter für alle möglichen Schemaparameter.

Die Klassen *attributeOR* und *relationOR* sind vom *SimPaD*-Manager durch die spezielle Schnittstelle - Datenstrukturen *SchemaObjectTable* und *SchemaObjAttr* - abgekoppelt. Somit haben sie die Fähigkeit erworben, die bereits in dem R-Datenbankmodul deklarierten Klassen *attribute* und *dataset* zu vererben, ohne die Icons der *SimPaD*-Oberfläche zu beeinflussen. Für die *DatabaseOR*-Klasse war es aber nicht möglich, ihre Analogie aus der R-Simulation zu übernehmen und zu ergänzen. *DatabaseOR* wurde daher komplett neu gestaltet.

Die Werteverteilung der OR-Schemaattribute wird von der *attribute*-Klasse, R-Simulation, geerbt. Die Verteilungsarten, zwischen denen man wählen kann, sehen die folgenden Variationen vor: *pseudo-uniform-, uniform-, scalar- und zipf*-Distribution. Für die Verteilung von Hierarchieattributen in der OR-Simulation ist die *pseudo-uniform*-Distribution zu bevorzugen. Zu den anderen wichtigsten geerbten Eigenschaften gehören unter anderen die Größe eines Attributes, sein Typ, Anzahl der Attributen innerhalb einer Relation (Klasse) etc.

Die Instanzen der Schemaklassen bzw. Schemaattribute werden mittels einer *Depth-First*-Strategie produziert. Zunächst wird das *DatabaseOR*-Objekt für die ganze Schemaklassenhierarchie erzeugt. Dann kreiert man das erste Objekt einer OR-Schemaklasse und erzeugt die Instanzen für alle ihre Schemaattribute, **Bild 16**. Erst danach wird mit dem Kreieren der zweiten Schemaklassen begonnen u.s.w.

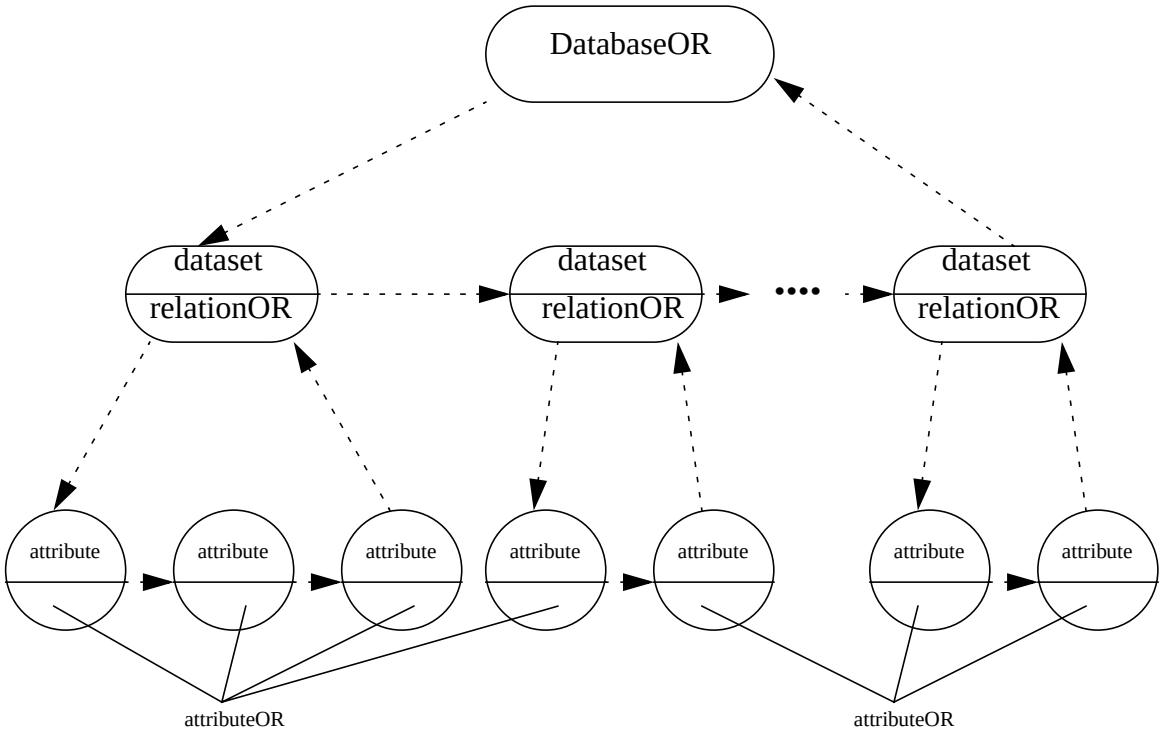


Bild 16. *Depth-First-Strategie beim Kreieren des OR-Schemas*

Die Zuordnung von Fragmenten zu den Platten wird anderes als im R-Fall durchgeführt. Laut des vorgeschlagenen Allokationsalgorithmus **Absch. 3.2.3** werden die Grenzen des Wertebereichs einer Schemaklasse eindeutig festgelegt. Außerdem werden die Zahl der entstehenden Fragmente und die Plattennummer, wo jedes Fragment liegt, in Form eines Arrays von *mix_element*-Strukturen vom Algorithmus **Anhang 1** zurückgeliefert. Die *mix_element*-Struktur hat die folgende Gestalt:

```
struct mix_element {
        long    part;       // number of fragment
        long    disk;       // disk, which the fragment belongs to
        long    from;       // begin value of fragment's domain
        long    to;         // end   value of fragment's domain
};
```

Die Grenzen eines Fragments werden dann separat nach dem Allokationsvorgang als Instanzen der *partitionOR*-Klasse abgelegt. Im Gegensatz zum R-Fall werden diese Werte während der Simulation nicht mehr neu berechnet. Die *partitionOR*-Klasse ist jedoch so aufgebaut, daß die Kompatibilität zu dem R-Vorgänger - *partition*-Klasse aus dem R-Datenbankmodul - nicht absolut verschwindet. Eine *partitionOR*-Klasse wird durch das Zusammensetzen von mehreren R-Partitionen[1] gebildet, wobei die Instanzen einer R-Partition immer noch als Grundsteine der OR-Allokation angesehen werden. Die Anzahl der R-Partitionen in einer *partitionOR*-Klasse ist der Zahl der Komponente in einer Schemaklasse identisch.

Als zusätzliche Information zu der Allokation einer Schemaklasse wird die Länge jeder Objektkomponente, sowie die Zahl der Attribute, die diese Komponente bilden, in der *declusteringOR*-Klasse gespeichert. Die *declusteringOR*-Klasse verfügt über Kenntnisse über alle *partitionOR*-Komponenten und somit über die ganze Allokationsanordnung der Schemaobjekte. Als Folge daraus kann für jedes Schemaobjekt aus der Instanz der *declusteringOR*-Klasse die Plattenummer für jede Klassenkomponente in Abhängigkeit von der Werteverteilung ermittelt werden. Diese Fähigkeit wird von den anderen OR-Simulationsmodulen während einer Scan-Operation stets genutzt, um den Zugriff auf die Objekte einer Schemaklasse zu gewähren.

In meinem Simulationsmodell werden dem Nutzer auch einige Parameter zur Verfügung gestellt, die zwecks einer korrekten Speicherung der Klassenattribute eingeführt wurden. Vor allem wird durch den booleschen Parameter "*combined level storage*" für die korrekte Art der Komponentenplazierung unmittelbar gesorgt. Wenn der Parameter den Wert "true" annehmen sollte, sollte es heißen, daß alle Attribute, die zum gleichen Level in der Attributhierarchie einer Klasse gehören, innerhalb einer Komponente *materialisiert* **Absch. 3.1.5** zusammengespeichert werden. Bei "false" wird dann jedes Attribut eine eigene Komponente bilden, und durch die Komponente der höheren Levels referenziert. Das Level eines Attributes wird durch den Parameter "*attribut's level in object's structure*" eindeutig bestimmt.

1. Genauer gesagt, enthält die Klasse *partitionOR* ein Array von Verweisen auf die Objekte der *partition*-Klasse.

In dem OR-Simulationsmodul werden verschiedene Varianten der Allokation von Schemaklassen und ihrer Komponenten angeboten und implementiert. Für jede Schemaklasse wird das Level in der Hierarchie (ausgehend vom Null), ihre Nummer (ausgehend vom Eins) und das Array der direkten Vorfahren spezifiziert[1]. Darüber hinaus wird die Art der nachstehenden Allokation der Objektteile durch den Parameter *"Round-Robin distribution"* manipuliert. Selbstverständlich kann auch eine andere Form der Allokation von Objekten vorgenommen werden. Dafür setzt man den Wert *"distribute automatically"* auf "false" und vergibt manuel den Verschiebungsgrad jeder Schemaklasse, sowie die Domänengrößen aller Schemaattributen. Diese Variante ist aber nicht zu empfehlen, da während einer komplexen Allokation massenweise Fehler produziert werden können.

4.2.2 Anfragegenerator

Nachdem die Konfiguration des Allokationsschemas erfolgreich abgeschlossen wird und die für die weitere Verarbeitung benötigten Daten über die Schemaobjekte und ihre Attributwerte aus dem *SimPaD*-Manager-Fenster geladen und geprüft werden, werden sie einer Routine unterzogen, welche dem Bearbeitungsvorgang einer Anfrage ähnelt. Da ein Anfragebaum beliebig tief sein und aus mehreren verzweigten Knoten und Branchen bestehen kann, muß das für das Erzeugen des Querybaumes zuständige Simulationsmodul namens *LoadGeneratorOR* die Fähigkeit besitzen, seine Datenstrukturen rekursiv gestalten zu lassen. Der *SimPaD*-Manager gibt dem Nutzer die Möglichkeit, durch bestimmte Elemente der Oberfläche, wie z.B. die Felder eines Aufzählungstypen für die Querybaumknoten, eine beliebig vorstellbare Anfragestruktur zu kreieren, **Bild 17**.

In der existierenden Version des OR-Simulationsmoduls werden zuerst zwei solchen Anfragegeneratoren dem Nutzer zur Verfügung gestellt. Sie lassen zwei diverse Anfragebäume konstruieren, die während einer Simulation zugleich erzeugt, gestartet und auf dieselben Datenbestände angewendet werden können. Die Zahl der Anfragegeneratoren im System kann jedoch leicht den neuen Erfordernissen angepaßt werden, indem ein kleiner Eingriff in die Methoden der *SimOR*-Hauptsimulationsklasse vorgenommen wird.

1. Die maximale Anzahl der Objekte darf den *"object_number"* -Parameter nicht überschreiten. Dasselbe gilt für die Attribute eines Objekts, Parameter *"attr_number"*.

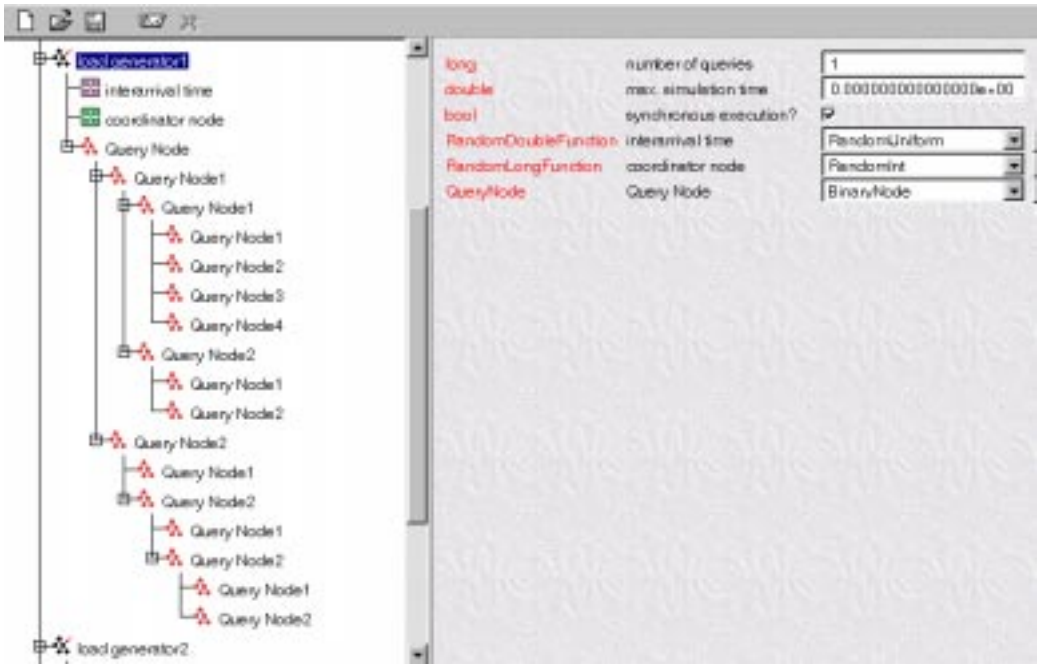


Bild 17. *OR-Querybaum als Bestandteil eines LoadGenerators*

Bei dem Erweiterungsbedarf erscheinen nach den Änderungen im Quellcode und dem abschließenden Kompilieren neue Anfragegeneratorelemente im *SimPaD*-Manager, welche die gleichen Eigenschaften und dieselbe Funktionalität wie die ersten Exemplare besitzen. Über das Ende der ganzen Simulation wird dann durch eine Benachrichtigung der *start*-Methode entschieden, die als Ausgangsfunktion für alle Anfragegeneratoren im System aufgefaßt werden kann. Die Nachrichten werden mittels CSIM-*Mailbox* [Me97] produziert und zwischen den einzelnen CSIM-Prozessen ausgetauscht.

Die Anzahl der Anfragen, die bei einem Anfragegenerator zum Ausführen gebracht werden, ist beliebig groß und kann durch den Parameter "*number of queries*" manipuliert werden. Hiermit erfüllt das OR-Simulationsmodul alle Voraussetzungen des Mehrbenutzerbetriebes. Die Parameter "*interarrival time*" und "*synchronous execution*" können in einem Anfragegenerator die Werte annehmen, mit denen die Ausführungsreihenfolge der Anfragen bestimmt wird.

Falls die Ausführung synchron erfolgen sollte, wartet eine Anfrage immer darauf, wann die Bearbeitung ihres Vorgängers beendet wird. Erst dann wird nach der "*interarrival time*"-Zeitspanne eine neue Anfrage generiert und gestartet. Bei dem asynchronen Verfahren beginnt die Ausführung einer neuer Anfrage sofort, sobald die "*interarrival time*" seit dem Start ihres Vorgängers vergeht (Simulation des Mehrbenutzerbetriebes). So lassen sich die Queries im System besser kontrollieren, und die Ergebnisse ihrer Ausführung einfacher auswerten. Der Parameter "max. simulatoin. time" bestimmt den Zeitpunkt, ab wann keine neuen Anfragen mehr generiert werden.

Unter dem Koordinator **Bild 17** ist ein Prozessorknoten gemeint, der die Koordinationsarbeit während der Bearbeitung von Indizes und Daten übernimmt und am Ende der Anfrageverarbeitung alle Ergebnisobjekte sammelt. Der Koordinatorknoten kann sowohl vom Nutzer eindeutig bestimmt, als auch durch den Zufall vom System selbst festgelegt werden. Der zweite Fall tritt dann auf, wenn anstelle eines bestimmten Wertes ein Knotennummerinterval angegeben wird. Der Koordinator bekommt in der Regel mehr Arbeit zugewiesen, als die anderen Prozessorknoten im System. Das spiegelt sich natürlich auch in seiner Auslastungsstatistik wieder, wenn man sie mit den Statistiken der anderen Prozessoren vergleicht.

4.2.3 OR-Anfragebaum

Die Gestalt einer Anfrage, die von einem Anfragegenerator produziert wird, kann vom Nutzer mittels *SimPaD*-Managers bequem konfiguriert werden und stellt eine baumartige Struktur dar, **Absch. 4.2.2**. Der Nutzer hat die Option, zwischen verschiedenen Verzweigungsknoten zu wählen. Die Typen der Knoten, die in einem Anfragebaum vorkommen können, sind *UnaryNode, BinaryNode, TertiaryNode und QuartaryNode*. Als Blätter des Baumes können nur die *UnaryNodes* verwendet werden.

Unter einem Knotentyp können ein oder mehrere bestimmte OR-Operatoren verstanden werden, die im Feld "*execution method*" angegeben werden können. In der OR-Simulation sind *Simple-Scan*, *Reference-Scan-*, *Reference-Join-*, *Merge-Join-* und *Hash-Join*-Operatoren implementiert. Sie sind mit einigen wenigen Einschränkungen sinnvoll kombinierbar.

Die Einschränkungen werden durch ein spezielles Regelsystem von den Methoden des Anfragegeneratoren kontrolliert. Bei der falschen Parameterangabe wird die Simulation sofort abgebrochen und die entsprechenden Meldungen ausgegeben. Die folgende Zuordnung von den Knotentypen zu den Anfragemethoden bildet dieses Regelsystem und muß bei der Parameterangabe zu den OR-Operatoren im *SimPaD*-Manager beachtet werden:

- *Simple Scan und Reference Scan sind stets einstellig;*
- *Hash Join und Reference Join sind stets zweistellig;*
- *Merge Join ist stets zwei- oder mehrstellig;*
- *der obere Sohn eines Reference Joins ist ein Reference Scan;*
- *Merge Joins und Reference Joins können beliebig geschachtelt sein;*
- *Unterhalb eines Merge Joins dürfen sich keine Hash Joins befinden.*

Ein *Simple Scan* und ein *Referenz Scan* haben die gleiche Parameterliste im Fenster des *SimPaD*-Managers, **Bild 18.** Die Bedeutung der Parameter ist in beiden Fällen jedoch verschieden. Für einen *Simple Scan* wird beispielsweise ein Feld namens "*selectivity*" für die Wahrscheinlichkeitsberechnung der Trefferrate eines Scan vorgesehen, die aber für den *Referenz Scan* keine Rolle mehr spielt. Das gleiche gilt für den Parameter "*attribute for selection*". Das läßt sich durch die Natur des Referenz-Join erklären. Ein *Referenz Scan* läßt sich lediglich als Teil des komplexeren Referenz-Join-Verfahrens, der für die Zugriffe auf die Seiten des Zielobjekts verantwortlich ist **Absch. 3.3.3**, parametrisieren und stellt damit keinen vollständigen *Scan*-Operator dar.

Also wird die Trefferrate eines *Referenz Scans* aus der Angabe zum *Referenz Join* ermittelt, dessen Parameterliste eine Ebene höher liegt. Sie wird im Feld "result probability after join" angegeben, welches auch für die anderen Join-Verfahren für die Ermittlung der Treffezahl eingesetzt wird. Im Gegensatz zum *Simple Scan* kann die Trefferrate mehr als hundert Prozent betragen, wenn die Zahl der Objekte nach einer Join-Operation sich erhöht. Als Basis für die Berechnung des Resultats bei einem Join-Verfahren wird die Zahl der Objekte genommen, die aus dem unteren Zweig des Hierarchiebaums in den Join hineinfließen.

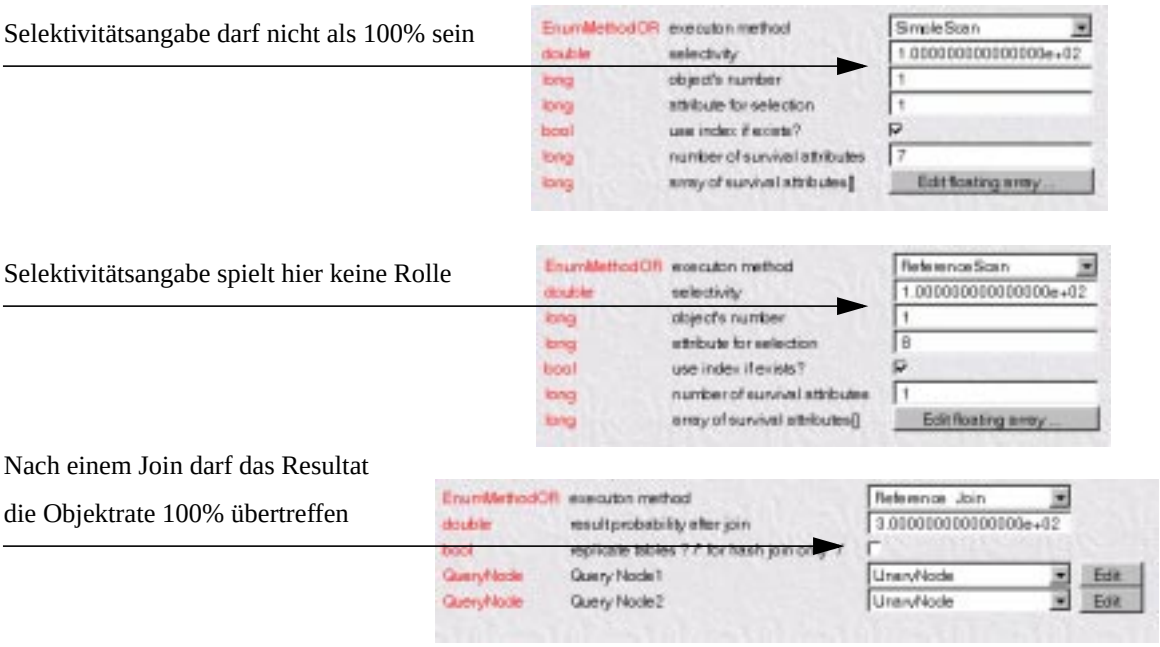


Bild 18. *Angabe der Resultatswahrscheinlichkeit für OR-Operatoren*

Die Abarbeitung von Operatoren im Anfragebaum wird parallel von mehreren Prozessoren durchgeführt. Die Anfrageverarbeitung beginnt mit den Scan-Operatoren und endet mit dem Operator, der sich an der Spitze des Baumes befindet. Somit beeinflussen die Operatoren, die tiefer in dem Baum liegen, die anderen.

Der Einfluß auf die höherliegenden Operatoren kann von zweierlei Art sein. Die erste Einflußart trägt einen numerischen Charakter und hängt direkt von der zu bearbeitenden Objektzahl in jedem Operator des Anfragebaumes stark ab. Die zweite Einflußart macht sich durch die Verfahren, nach denen die höherliegenden Operatoren durchgeführt werden, bemerkbar. Ein Merge-Join darf beispielsweise nicht nach einem Hash-Join plaziert werden, da die Sortierungsreihenfolge für die Eingabedaten des Merge-Join durch den Hash-Join im allgemeinen zerstört wird.

Einem Referenz-Join werden auch zwei verschiedene Algorithmen zugrunde gelegt. Werden beispielsweise die Objekte einer abgeleiteten Klasse nach dem Vorbild ihrer Basisklasse gespeichert und dann die beiden Klassen in einem Referenz-Join verknüpft, so sollen auch die Zugriffe auf die zweite Klasse in der selben Reihenfolge wie in der ersten auf den hintereinanderliegenden Seiten folgen. Für die Klassen, die unsortiert bzw. verschieden sortiert sind, soll diese Regel nicht mehr beachtet werden.

Die Abkopplung zwischen den Datenstrukturen, die einerseits für das Gestalten eines Anfragebaumes auf der *SimPaD*-Manager-Oberfläche und andererseits für die innere Simulationsverarbeitung benötigt werden, wird wie bei der Übertragung des OR-Schemas **Absch. 4.2.1** wieder erforderlich. An dem eigentlichen Simulationsvorgang nehmen nur die Duplikate der *LoadGeneratorOR*- und abstrakten *QueryNode*-Klassen sog. *Clones* teil. Der Kopiervorgang wird gleich nach der Konsistenzprüfung der Parameter des *SimPaD*-Manager beim Start der Simulation durchgeführt. Hierfür werden die Methoden namens *create_clone* sowie in die *LoadGeneratorOR* als auch in die einzelnen Knotenklassen *UnaryNode*, *BinaryNode* etc., die von der *QueryNode*-Klasse abgeleitet sind, integriert.

4.2.4 Index-Verarbeitung und Datenzugriff

Die Konzepte der Implementierung eines Scan-Operators sowie die Methoden für die Indexverarbeitung in der OR-Simulation unterscheiden sich nicht so sehr von denen, die bereits für die R-Simulation entwickelt wurden. Es waren aber einige Eingriffe erforderlich, um die R-Scan-Verarbeitung neuen Zwecken anzupassen.

Die Methode eines Anfragegenerators, welche die einmalige Bearbeitung seines ganzen Querybaumes durchführt, heißt *one_query.* Sie bekommt als Parameter eine Querynummer aus der Anfrageliste sowie einen Verweis auf eine CSIM-*Mailbox* [Me97], womit sie das Terminieren der Query allen anderen Methoden mitteilt. Unmittelbar nach dem Aufruf bestimmt die *one_query*-Methode den Koordinatorknoten **Absch. 4.2.2** für die ganze Anfrageverarbeitung.
Danach werden alle Scan-Blätter des Baumes gezählt.

Alle Eigenschaften von Scan-Operatoren einer Anfrage, die vom Nutzer im *SimPaD*-Manager angegeben werden, werden in die *scanOR_information*-Struktur übertragen. Mit Hilfe von Informationen aus dieser Struktur wird für jeden der Scan-Operatoren ein sog. *annotated_scanOR*-Objekt erzeugt, das dann an die Scan-Verarbeitungsroutine weitergeleitet wird. Während der *Prepare*-Phase einer Anfrage s.u. werden die *annotated_scanOR*-Objekte mit zusätzlichen Informationen vervollständigt. Die *annotated_scanOR*-Klasse hat alle wichtigsten Eigenschaften der *annotated_scan*-Struktur aus der R-Simulation geerbt.

```
class annotated_scanOR { // extended information relating to a single scan query node
    public:
        scan_queryOR*   qnode;          // original query node
        long            coord;          // coordinator node (processor)
        long            max_index_par;  // max. degrees of parallelism for
        long            max_data_par;   //   index and data access, resp.
        long            real_index_par; // actual degrees of parellelism for
        long            real_data_par;  //    index and data access, resp.
        dbindex*        idx;            // index used (0 if none)
        long long       index_hits;     // total hits in the index
        long long       data_hits;      // total hits in the data
        long            first;          // first and last attribute value for
        long            last;           // range scans, respectively
        long            num_index_jobs; // [size of] array of index scan jobs
        index_job*      index_jobs;     //   in descending order of #pages
        long            num_data_jobs;  // [size of] array of data scan jobs
        data_job*       data_jobs;      //   in descending order of #pages
};
```

Unter anderem wird mit einem Parameter einem Scan-Operator die Nummer des Selektionsattributes übergeben, **Bild 18**. Wie es bei der R-Scan-Verarbeitung bereits der Fall war, hat man in der OR-Simulation die Möglichkeit vorgesehen, die Schemaattribute mit einem Typ der Indizes zu versorgen. Bei dem Konfigurieren eines Schemaattributes auf der *SimPaD*-Oberfläche wird zwischen “*clustered*”-, “*non-clustered*”- und “*none*”-Typen eines Indexes unterschieden. Die Parameter “*clustered*” und “*non-clustered*” entsprechen den Kategorien aus dem **Absch. 3.3.1**. Der Wert “*none*” ist dann anzugeben, wenn für das Attribut gar keinen Index erzeugt werden soll. Sollte im letzten Fall das betroffene Attribut einer Klasse das Selektionsattribut einer Anfrage für das vorgegebene OR-Schema werden, unterliegen die Objekte der Klasse unabhängig von der Selektionsangabe des Nutzers einem totalen Scan.

Die ersten Attribute einer Schemaklasse, die sich innerhalb eines Hierarchiebaumes befindet, sind Hierarchiereferenzen. Sie können immer einen *Clustered Index* besitzen, da die Sortierungsreihenfolge, welche für die Datenallokation **Absch. 3.2.3** gewählt wird, ihrer Anordnung entspricht. Somit arbeiten die Operatoren wie Referenz-Join, welche die Bestandteile einer Hierarchie bei den selektiven Anfragen zusammenführen, schneller, da die benötigten Daten nur teilweise gelesen werden und die Anzahl der Seitenzugriffe wesentlich reduziert wird. Damit man jedoch beim Bedarf auf den Indexeinsatz auf dem Selektionsattribut verzichten kann[1], sieht man für jedes Blatt des Querybaumes die Möglichkeit vor, durch das explizite Setzen des Parameters "*use index if exists?*" auf "false" die Indexverarbeitung zu unterdrücken.

Die Verarbeitung aller Indizes wird bei dem Simulationsstart einer Query initiiert. Der Koordinator sammelt die Informationen über die im System vorhandenen Zugriffspfade und verteilt möglichst gleichmäßig die Aufträge zwischen den einzelnen Prozessorknoten. Um das Gleichgewicht zwischen den einzelnen Platten zu erreichen, wird die Menge der Aufträge so gemischt, daß die Aufgaben, die sich auf die gleichen Plattenmengen beziehen, in ihrer Liste möglichst weit auseinander gebracht werden. Das eigentliche Scannen von den Daten des Allokationsschemas beginnt erst dann, wenn das Lesen, Verteilen und Verarbeiten der Indizes komplett abgeschlossen ist.

Für die gesamte Scan-Verarbeitung werden zwei Schritte nötig. Man unterscheidet zwischen ihren *Prepare*- und *Execute*-Phasen, wobei die *Prepare*-Phase die Verarbeitung der Anfrageindizes für alle Scan-Operatoren im System beinhaltet. Die *Prepare*-Phase ist dadurch gekennzeichnet, daß ihre Ausführung noch keine CSIM-Prozesse für Hardware-Module startet, die für die Bearbeitung der Datenfragmente benötigt werden. Während dieser Phase werden für jeden Scan-Operator ein Array von *Job*-Strukturen vorbereitet.

1. Aus Effizienzgründen ist es nötig, wenn die Selektivität einer Anfrage sich dem Wert 100% nähert.

```
class data_job {
    public:
        partition* part;    // pointer to partition
        long       pages;  // number of pages to be scanned there
        long       first;   // first page to be scanned
        long       last;    // last page to be scanned
        long long  hits;    // number of hits in partition
        long       node;   // number of CPU to process partition
        long       begin_of_part;// first attribute value in partition
};
```

Die OR-Jobs unterscheiden sich nicht von denen, die bereits in der R-Simulation eingeführt wurden. Sie tragen die entscheidende Information über die Seiten, die während eines Scan zu lesen sind, sowie ihre Zugehörigkeit zu den Fragmenten bzw. Platten. Die Anzahl der Seiten wird in Abhängigkeit von dem Scan-Typ ausgerechnet. Die Index-Strukturen, die Verteilung des Selektionsattributes und die Zufallsfaktoren beeinflussen diese Berechnung. Außerdem wird der Wahrscheinlichkeitsangabe des Nutzers Rechnung getragen.

Die Zuordnung von Seiten zu den Platten wird durch die Allokation eindeutig bestimmt. Der Parameter *begin_of_part* wird in die *Job*-Struktur eingeführt, um die Suche nach dem entsprechenden Wertebereich für die Seiten eines Jobs zu vereinfachen. Die Seiten verschiedener Jobs, die wegen ihrer abhängigen Allokation gemeinsam bearbeitet werden, werden mittels des *begin_of_part*-Parameters schnell gefunden, was vor allem für die Merge- und Referenz-Join-Bearbeitung relevant ist.

Die zweite *Execute*-Phase wird nach der Analyse des ganzen Anfragebaumes für alle Scan-Operatoren unter der Kontrolle eines *Scheduling*-Verwalters **Absch. 4.2.6** durchgeführt. Die Kenntnisse über die einzelnen *Jobs*, die während der *Prepare*-Phase gesammelt werden, werden hier für die Hardware-Module eingesetzt. Dabei wird der für die einzelnen Systemkomponenten entstehenden Aufwand in Instruktionen für CPUs, Nachrichten für das Netzwerk und Seiten für Puffer berechnet. Das Resultat wird den entsprechenden Systemmodulen eingespeist.

4.2.5 Analyse des Querybaumes

Nachdem die Indizes abgearbeitet sind und die *Prepare*-Phase für alle Scan-Operatoren abgeschlossen wird, beginnt man mit der Analyse der gesamten Anfragebaumstruktur. Der ganze Querybaum wird ausgehend von dem Wurzelknoten auf die Verzweigungen untersucht. Die rekursive Methode, die diese Aufgabe in der *LoadGeneratorOR*-Klasse übernimmt, wird *construct_query_tree* genannt und bekommt als einen der Parameter den Baumknoten, von dem sie aufgerufen wird.

Der zweite Übergabeparameter der Methode *construct_query_tree* ist eine *estimated_result*-Struktur (passive C++-Klasse). Diese Struktur verfügt über die Information über die Größe und Zahl der Objekte, die am Ende jedes Verarbeitungsschrittes herauskommen. Die *estimated_result*-Struktur wird auch dazu benötigt, die Kenntnisse über die Verweise auf die *Mailbox* und das *Event* [Me97], womit der *Scheduling*-Verwalter **Absch. 4.2.6** mit den anderen CSIM-Prozessen für gleichzeitige Scan-Operatoren kommuniziert, zwischen den einzelnen Operatorknoten zu verbreiten. Darüber hinaus werden dort die Statistiken über die möglichen Merge-Jobs im System gesammelt. Darunter findet man auch Information über die Gesamtgröße, den Gesamtspeicherbedarf und die Plattenverteilung eines Jobs, welche für das gemeinsame *Scheduling* der Scan-Operatoren unentbehrlich sind.

```
class estimated_result {
    public:
        long        tuple_size;// object size
        long long   tuples_number;// number of objects
        long        coordinator_node;// cpu-number for coordinator
        bool        merge_exists;// at least one merge has been initiated
        mailbox*    mb;     //  distributed scan managament mailbox
        event*      final_event;// the last scan final event
        long        real_jobs_numb;// number of real jobs to process
        long        all_possible;// max. possible jobs can be created
        declusteringOR*    decl;// main declustering for merge und ref. join
        merge_job**merge_array;// array of komplex merge jobs
};
```

Die *estimated_result*-Struktur wird mit den Werten über die Gestalt des gesamten Anfragebaumes von allen Knoten versorgt. Eine Referenz auf diese Struktur wird vor dem ersten Methodenaufruf erzeugt. Sie wird bei jedem folgenden Aufruf zu den unterliegenden Knoten immer weitergeleitet, solange alle Blätterknoten noch nicht erreicht worden sind. Nachdem die Blätterknoten die *estimated_result*-Struktur mit den Daten, die von den Scan-Operatoren bereitgestellt werden, erstmal versorgt haben, werden auch die höheren Operatorknoten sich an diesem Prozeß beteiligen, indem sie diese Struktur bei der Parameterrückgabe immer weiter vervollständigen und präzisieren. Am Ende dieses Vorgangs werden die Kenntnisse über die ganze Baumstruktur und ihre einzelnen Komponenten gesammelt.

Für jede Verzweigung ist ein Identifizierungsvorgang notwendig, der die Objektinstanzen für alle Kollektoren **Absch. 4.2.7** erzeugt. Nach der Identifikation folgt eine ganze Reihe von Schritten, die spezifisch für jeden Typ der Verzweigung sind. Dann setzt man den Untersuchungsvorgang in jedem unteren Zweig des Querybaumes sukzessiv solange fort, bis die Blätterknoten des Baumes erreicht werden. Die einzige Ausnahme hierfür stellt die Verzweigung eines Hash-Join-Types dar. Dort untersucht man den zweiten Unterbaum erst dann, wenn die Bearbeitung des ersteren unteren Zweigs zu Ende ist.

Nach dem Erreichen des Bodens des Anfragebaumes werden alle Scan-Operatoren initiiert und als unabhängige CSIM-Prozesse abgelegt. Sie beginnen aber noch nicht, die Daten von den Platten zu lesen (die Kontrolle wird an sie erst später übergeben [Me97]). Als Anfang der *Execution*-Phase wird für alle Scan-Operatoren der Moment bezeichnet, wenn der für ihren Start benötigte Befehl seitens des *Scheduling*-Verwalters **Absch. 4.2.6** erteilt wird.

Wie bereits erwähnt wurde, unterbricht ein Hash-Operatorknoten die Baumstrukturanalyse. Laut seiner Natur besitzt ein Hash-Join-Operator zwei Phasen **Absch. 3.3.2,** die getrennt durchgeführt werden, **Bild 10**. Der erste Zweig eines Hash-Joins-Knoten entspricht einer *Build*-Phase und wird als erste komplett abgearbeitet, bevor man sich mit dem zweiten Zweig des Unterbaumes auseinandersetzt.

Um den Einfluß der *Build*-Phase auf die *Probe*-Phase in der Simulation zu vermeiden, wird für den ersten Zweig des Hash-Join-Knotens eine flache Kopie der *estimated_result*-Struktur erzeugt. Sobald das Duplikat seine Daten mit neuen Kenntnissen, die in dem ersten Zweig des Baumes erworben wurden, erneuert, wird sofort eine verteilte Hash-Tabelle im Pufferbereichen aller Prozessoren angelegt und eine von dem gesamten Anfragebaum unabhängige *Scheduling*-Routine gestartet. Erst dann, wenn die ganze *Build*-Phase zu Ende ist, wird die Analyse des zweiten Zweiges des Hash-Joins mit dem Original der *estimated_result*-Struktur fortgesetzt. Der Datenfluß wird an die höherliegenden Kollektoren **Absch. 4.2.7** weitergeleitet, als ob der erste Zweig für die *Build*-Phase des Joins überhaupt nicht existiert hätte, **Bild 19**.

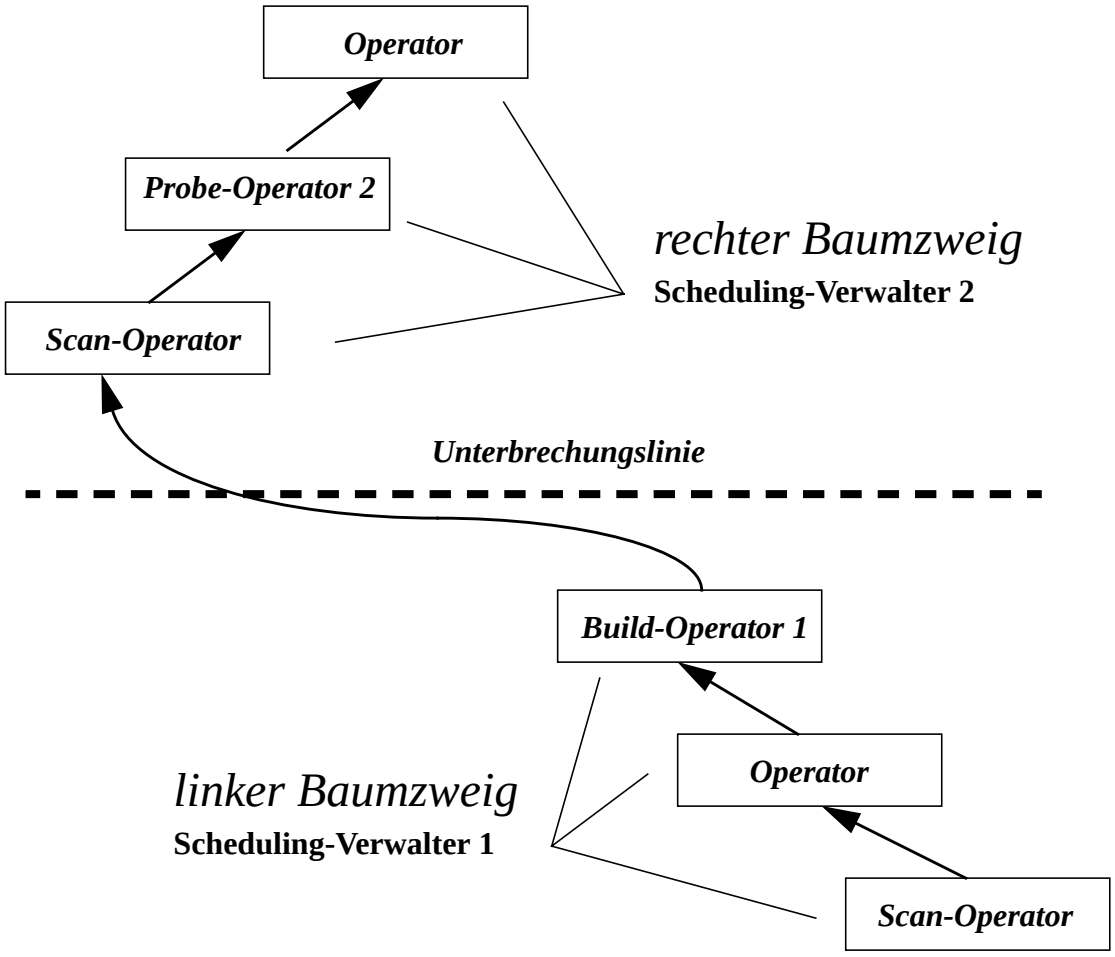


Bild 19. *Simulation eines Hash-Operatoren*

4.2.6 Scheduling der Scan-Operatoren

Ein *Scheduling*-Verwalter wird entweder von der *construct_query_tree*-Methode eines Anfragegeneratoren oder einem Hash-Join-Knoten initiiert. Das Kreieren erfolgt direkt nach der Anfragebaumanalyse, die im **Absch. 4.2.5** beschrieben wurde. Die Methode der *QueryProcessingOR*-Klasse, worunter sich der *Scheduling*-Verwalter verbirgt, heißt *scheduling_jobs.* Sie läuft immer parallel zu dem Hauptsimulationsverfahren (getrennter CSIM-Prozeß). Der Verwalter bekommt als Parameter die bereits bekannte *estimated_result*-Struktur mit allen Ergebnissen der Anfragebaumanalyse.

Als erstes wird vom Verwalter der aktuelle Stand der Belastung aller Hardware-Komponenten in dem ganzen Simulationssystem abgefragt. Die geeigneten Prozessorknoten, die noch genügend Ressourcen für die Verarbeitung mindestens eines Scans besitzen, werden von ihm entdeckt und nach ihrer laufenden Auslastung angeordnet. Somit wird der Parallelitätsgrad der Bearbeitung ermittelt, der im besten Fall der Anzahl aller Prozessoren im System äquivalent ist.

Mit dem zweiten Schritt soll das Sortieren der komplexen Scan-*Jobs* **Absch. 4.2.4** erfolgen. Dies geschieht aufgrund einer der im **Absch. 3.4** aufgelisteten *Scheduling*-Strategien. Für jeden Scan-Job wird die Kostenfunktion nach dem α-Parameter des Nutzers laut der aktuellen Simulationskonfiguration ausgerechnet und durch den Einsatz des *qsort*-Algorithmus angeordnet.

Bei dem nächsten Schritt werden die beiden sortierten Listen der Prozessorknoten und Jobs gegenübergestellt, so daß ein CPU mindestens einen Job zugewiesen bekommt. Es besteht die Möglichkeit, einem freien Prozessor mehr als einen Job zuzuordnen, indem man den Parameter "*max. number of complex jobs per CPU node*" (Modul *QueryProcessingOR*) erhöht. In dem Fall wird die Verteilung der Jobs solange fortgesetzt, bis entweder alle CPUs mit der entsprechenden Zahl der Jobs belegt werden, oder die Liste aller Jobs geleert wird.

Die Zuordnung der Jobs an die Prozessorknoten wird so durchgeführt, daß zunächst jeder CPU einen Job zugewiesen bekommt, erst dann einen zweiten u.s.w. Als weitere Vorkehrung zu der effizienten Anfrageverarbeitung wird das Eliminieren von leeren Scan-Operatoren angesehen. Diese Maßnahme ist vor allem für den *Merge*-Join relevant, da die Fragmente verschiedener Klassen, die bei dieser Operation zusammengesetzt werden, nicht unbedingt mit den anderen übereinstimmen mußten, die während der *Prepare*-Phase **Absch. 4.2.4** initiiert wurden.

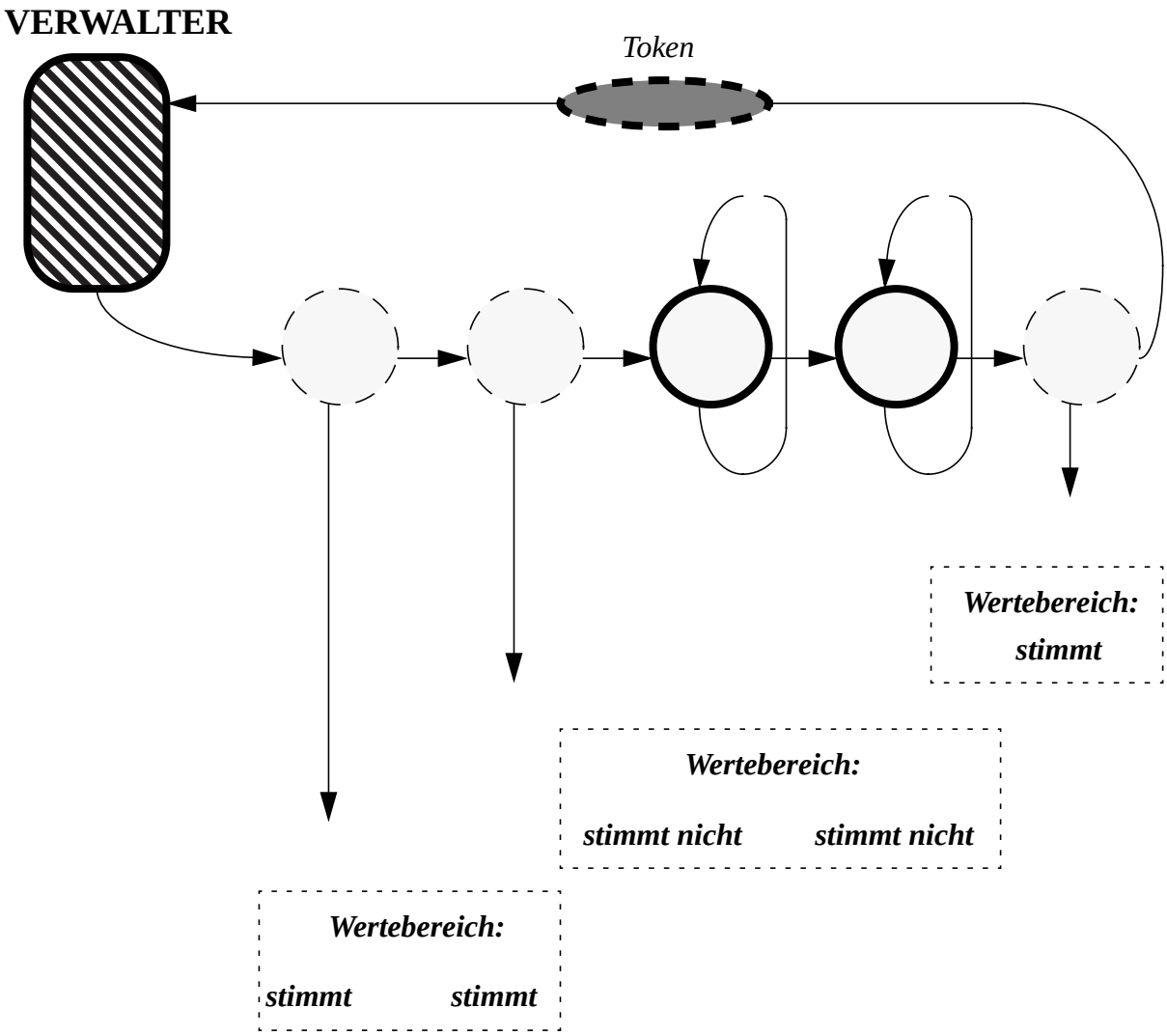


Bild 20. *Verwaltung der Scan-Prozessen*

Erst dann wird den gewählten CSIM-Prozessen für Scan-Operatoren die Erlaubnis zum Starten vom Verwalter erteilt. Die Benachrichtigung erfolgt mittels eines Tokens, der die Information über die Wertebereich des Verteilungsattributes mit sich trägt. Der Token wird von jedem Scan-Prozeß abgefangen, nach dem Übereinstimmen mit dem Wertebereich seines Fragments geprüft und unabhängig davon, ob ein Treffer vorliegt oder nicht, dem nächsten CSIM-Prozeß übergeben. Bei der Übereinstimmung wird die Scan-Routine sofort durchgeführt. Anderesfalls kehrt der Scan-Prozeß in die Warteposition zurück (endlose Schleife), **Bild 20**.

Der Verwalter, der das Token kreiert, hat auch die Aufgabe, den Token beim Rückkehr zu zerstören. Sobald den Token abgeschickt wird, wird der Verwalter selbst in die Warteposition durch eine *mailbox->receive*-Funktion versetzt. Dank innerer CSIM-Organisation wird gesichert, daß der Verwalter erst dann den Token wiederbekommt, wenn alle CSIM-Prozesse, die auf das *Mailbox* gewartet haben, die von dem Verwalter geschickte Nachricht ausgepackt, angesehen, eingepackt und weitergesendet haben. Durch diese CSIM-Eigenschaft geht die Kontrolle über die Ablaufreihenfolge und Ausführungszeiten der einzelnen Scan-Operatoren nicht verloren.

Das Verschicken des Tokens kostet aber keine Simulationszeit und ist somit momentan. Als Resultat werden im System mehrere Scan-Prozessen parallel gestartet. Der Aufwand, der durch Ausführung der Operatoren entsteht, wird dann auf den Hardware-Komponenten des Systems registriert. Bei dem Zugriff auf die gleichen Hardware-Komponenten stören sich die Scan-Operatoren wie in einem realen DBS gegenseitig. Das hat Verzögerung des Simulationszeit zufolge, was man nach dem Ablauf der Simulation aus der Resultatsdatei entnimmt. Die gesamte Auslastung einzelner Hardware-Komponenten wird durch die verursachten Störungen seitens Scan-Operatoren ebenfalls erhöht.

Während eines Lesevorganges wird die ganze Menge von Objekten produziert. Sie werden als Datenstrom den höherliegenden Kollektoren **Absch. 4.2.7** zwecks weiterer Verarbeitung sofort übergeben. Somit können die CPU-Konflikte zwischen den Operatoren entstehen, die einerseits einen direkten Lesevorgang durchführen, und die anderen, die sich mit der Verarbeitung des Datenstromes befassen und auf den höherliegenden Ebenen im Anfragebaum agieren. In solchen Fällen sind dann zusätzliche CPU-Kosten fällig.

Die Übergabe der Objekte kann abhängig von den Scan-Operationen von dem Typ des Kollektoren synchron oder asynchron erfolgen. Im ersten Fall wird der Lesevorgang solange verzögert, bis der höher liegende Kollektor die Bearbeitung der gegebenen Objekte komplett abgeschlossen hat. Als Beispiel hierfür ist der Reference-Join-Kollektor zu nennen. Dem Scannen einer Seite des Ausgangsobjekts **Absch. 3.3.3** folgen unmittelbar die Zugriffe auf die Instanzen des Zielobjekts. Erst dann, wenn die Seite des Zielobjekts in den Puffer eines Prozessors geholt und dort komplett abgearbeitet wird, wird der Lesevorgang der Seiten des Ausgangsobjekts fortgesetzt. Im Gegensatz dazu erfolgt die Bearbeitung eines Hash-Kollektoren asynchron, d.h. die zu lesenden Objekte werden über das Netzwerk an das entsprechende CPU abgeschickt, ohne die Kontrolle über sie im scannenden Prozeß beizubehalten. Der Lesevorgang wird ohne Zeitverzögerung weiter fortgesetzt.

Sobald einer der Scan-Operatoren seine Aufgabe erfüllt hat, wird eine Terminate-Nachricht an den Verwalter, der sich bereits in der Warteposition befinden soll, geschickt. Für die dynamische Lastbalancierung wird dann eine weitere Voraussetzung geschaffen, indem der Verwalter einen neuen nach den *Scheduling*-Prinzipien gewählten Job dem Prozessorknoten, der mit dem vorigen Job gerade erst fertig geworden ist, zuordnet. Dieser Vorgang wird solange fortgesetzt, bis keine Jobs mehr existieren.

Falls gar keine Jobs mehr zu vergeben sind, beginnt der Verwalter die Zahl der verbliebenden Scan-Operatoren herunterzuzählen. Sobald diese Zahl den Wert Null erreicht hat, wird vom Verwalter das Ende aller Lesevorgänge angekündigt. Diese Benachrichtigung wird dem Hauptsimulationsmodul sowie allen Scan-Kollektoren mittels eines CSIM-Events sofort mitgeteilt. Das Terminieren des Verwalterprozesses ist die allererste Bedingung für das Termenieren der gesamten Simulation einer Anfrage. Die zweite benötigte Bedingung geht erst dann in Erfüllung, wenn der letzte der Kollektoren, der diese Anfrage bearbeitet, seine Aufträge erfolgreich beendet hat.

4.2.7 Datenkollektoren

Datenkollektoren sind die Klassen, deren Instanzen anstelle aller Knoten in dem Operatorbaum direkt eingesetzt werden. Sie ermöglichen, jeden Typ der in dem Querybaum vertretenen Operatoren gemäß ihrer Natur spezifisch zu behandeln. Der Datenstrom, der nach dem Lesevorgang aus Objekten einer Klasse entsteht, fließt durch die Kollektoren solange hindurch, bis die Wurzel des Anfragebaumes erreicht wird. Somit wird der Datenaustausch in der Simulation unterstützt.

Die Aufgabe des Wurzelkollektoren besteht darin, alle Ergebnistupel zu sammeln und dem Koordinatorknoten zu übergeben. Sobald das letzte Objekt bei dem Koordinator ankommt, endet der Bearbeitungsprozeß der Anfrage, was in Form eines Events und einer Mailbox dem Hauptsimulationsmodul sofort mitgeteilt wird.

Die neuen OR-Kollektoren wurden von den einfachen Kollektortypen des R-Simulationsmoduls wie *simple_scan_collector* oder *event_scan_collector* abgeleitet. Die beiden Basisklassen werden ihrerseits auf der CSIM-Event-Klasse aufgebaut. Dieses Event wird von einem Kollektor erst dann gefeuert, wenn sein letzter Ergebnistupel abgeschickt wird. Dadurch wird die Konsistenz des Datenflusses gewährleistet.

Einige komplexere Klassen aus dem *R-Star-Join*-Simulationsmodul kommen bei der OR-Simulation zum Einsatz. So werden die Kollektoren für die Hash-Operatoren mit oder ohne Replikation der Instanzen erster Klasse (partitionierte vs. replizierte Hash-Tabelle) während der *Build*-Phase **Absch. 3.3.2** bereitgestellt. Sie sollen lediglich an wenigen Stellen für die OR-Simulation spezifisch ergänzt und mit wenigen neuen Methoden ausgestattet werden. Die anderen Kollektor-Klassen wie Merge- und Referenz-Join-Kollektoren werden aber komplett neu gestaltet.

Im Gegensatz zu der R-Simulation reguliert sich der Datenfluß in dem OR-Prototyp selbst. Die OR-Kollektoren benötigen keinen zentralen Verwalter-Prozeß, der ihre Arbeit dirigiert. Das Initialisieren und Terminieren einzelner Kollektor-Instanzen wird von den Kollektor-Klassen selbst herauf- und heruntergezählt. Darüber hinaus wird das Pipelining so organisiert, daß einem beliebigen Kollektoren aus den oberen Schichten nicht gestattet wird, die allerletzte Terminate-Funktion zu beenden, bevor alle seinen Vorgänger-Kollektoren auf den tiefer liegenden Knoten

des Kollektorbaumes das bereits getan haben. Um diesen Effekt zu erreichen, werden die Verweise auf die Kollektoren der unteren Schichten immer weiter nach oben propagiert.

In dem OR-Simulationssystem sind die folgenden Kollektortypen zu unterscheiden:

- *simple_scan_collector_OR* ist die Basiskollektorklasse für fast alle OR-Kollektortypen; außerdem wird sie für das Abschicken der Resultatsobjekte am Ende einer Anfrage an den Koordinator und die Benachrichtigung des Anfragegenerators eingesetzt;
- *or_scan_collector* ist die Kollektor-Klasse für das Sammeln von Tupeln, die als Ergebnis eines Scan-Operatoren erhalten werden;
- *multi_scan_collectorOR* und *replicating_multi_scan_collectorOR* sind die Kollektoren, die für die Build-Phase eines Hash-Joins ohne und mit Replikation der Hash-Tabelle verwendet werden;
- *pipeline_join_collectorOR* und *replicated_pipeline_join_collectorOR* sind die entsprechenden Kollektor-Klassen für die Probe-Phase eines Hash-Joins;
- *reference_join_collector* ist der Kollektortyp für einen Referenz-Join;
- *sub_merge_collector* und *major_merge_collector* bilden einen Merge-Join-Operator gemeinsam.

Die Beschreibung der Kollektoren findet man entweder in dem Verzeichnis *DataCollectors* oder in dem Verzeichnis *LoadGeneratorOR*. Einzelheiten zu den Algorithmen der einzelnen Kollektoren werden in dieser Arbeit nicht weiter erläutert. Man findet die Grundlagen für ihren Aufbau in der Theorie **Absch. 3.3.2** wieder. Die Menge der Tricks, die während der Implementierung einzelner Kollektoren angewendet werden, stellen eine gewöhnliche Programmierungsarbeit dar, und haben nicht so viele Besonderheiten, um hier diskutiert zu werden.

5. Simulationsergebnisse

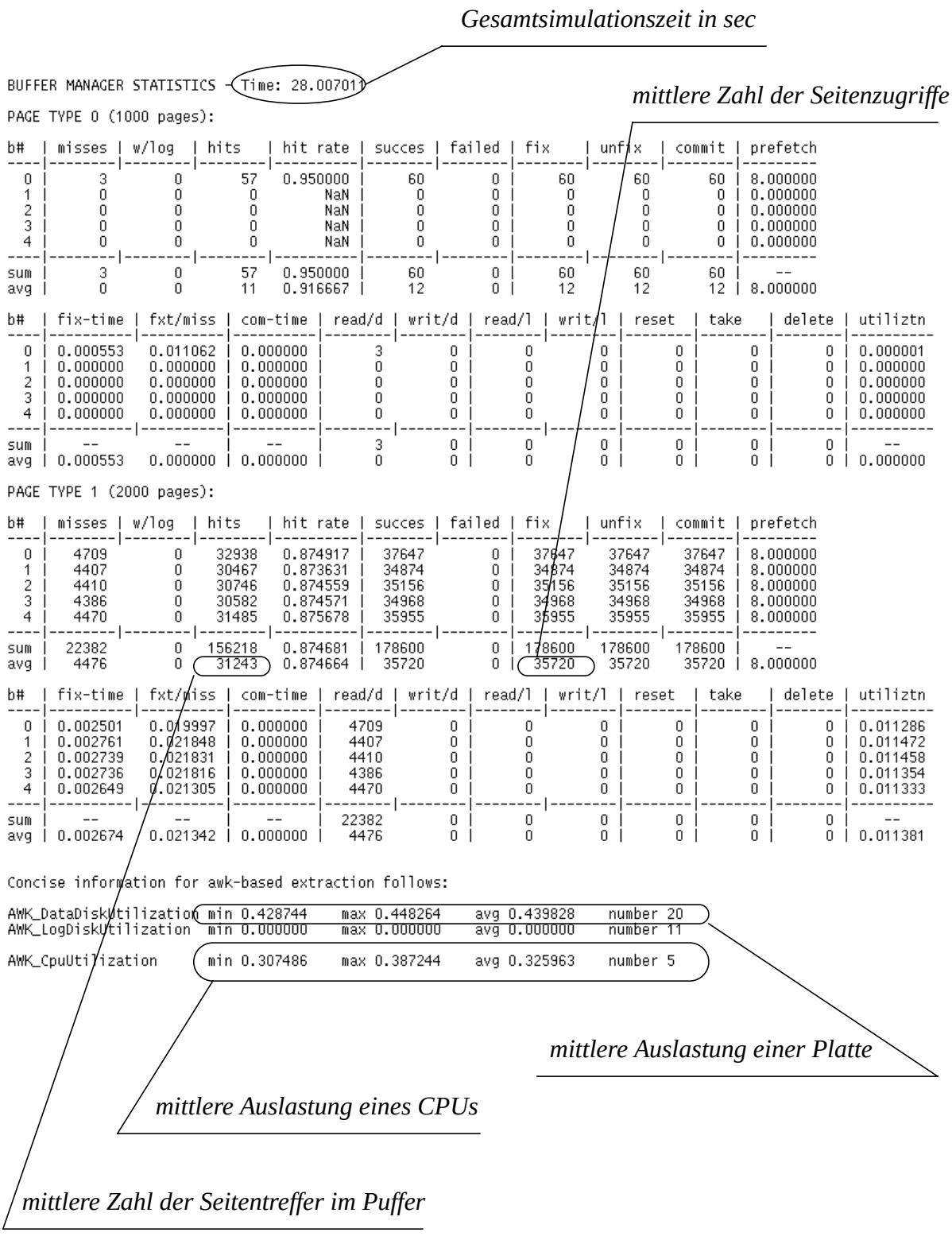
Nachdem die OR-Erweiterung zu dem Simulationssystem fertiggestellt wurde, wird sie vielseitig untersucht und getestet. Wie bei dem Umgang mit einem empfindlichen Meßinstrument bestand unsere weitere Aufgabe darin, die OR-Module solange zu tunen, bis sie endlich die korrekten und abschätzbaren Ergebnissen liefern werden. Erst dann, wenn die endgültige Systemkonfiguration festgelegt wird und man sich überzeugen läßt, daß bei den Simulationsläufen gar nichts schief geht, kann die Arbeit mit den Effizienzmessungen verschiedener Join-Verfahrend erst richtig beginnen.

Auf **Bild 21** ist ein Fragment der Statistikausgabe zu sehen, die am Ende jeder Simulation in die Resultatsdatei geschrieben wird. Aufgrund dieser Information kann das Gesamtsimulationssystem für OR-Daten auf ihre Korrektheit geprüft und den neuen spezifischen Anforderungen angepaßt werden. Einige Werte, die bei der Bewertung der Ergebnisse ein höheres Maß an Aufmerksamkeit bekommen, sind auf dem Bild eingekreist. Diese Werte spiegeln die Auslastung der wichtigsten Hardware-Komponenten des Systems während der Simulation sowie die spezifischen Charakteristika der Anfrageausführung wieder.

Damit die Simulationsergebnisse durch die Zufallsfaktoren nicht zu stark beeinflußt werden, werden die Simulationen meistens im Weiderholungsmodus ohne Verzögerung der Simulationszeit zwischen den einzelnen Abläufen ausgeführt. Die Resultate für die einzelnen Statistiken werden dann durch die Anzahl der Simulationsläufe geteilt und ihre Durchschnitte gebildet.

5.1 Einstellung der Parameter

Eine ganze Reihe der Parameter soll für die anstehenden Experimente mit vernünftigen Werten belegt werden. Hier hat uns wiederum die Erfahrung aus den früheren R-Simulationen geholfen. Die Parameter für Hardware-Komponenten des Systems werden mit einigen wenigen Änderungen in die OR-Simulationserweiterung übernommen. Solche Parameter wie beispielsweise die Geschwindigkeit eines Prozessorknotens in MIPS sollten jedoch aktualisiert und mit den Werten aus der realen Welt in Einklang gebracht werden. So entsprach die Geschwindigkeit eines Prozessors bei allen OR-Simulationen dem Wert 50.




```
BUFFER MANAGER STATISTICS - Time: 28.007011

PAGE TYPE 0 (1000 pages):

b#  | misses | w/log  | hits   | hit rate | succes | failed | fix    | unfix  | commit | prefetch
----|--------|--------|--------|----------|--------|--------|--------|--------|--------|---------
  0 |      3 |      0 |     57 | 0.950000 |     60 |      0 |     60 |     60 |     60 | 8.000000
  1 |      0 |      0 |      0 |      NaN |      0 |      0 |      0 |      0 |      0 | 0.000000
  2 |      0 |      0 |      0 |      NaN |      0 |      0 |      0 |      0 |      0 | 0.000000
  3 |      0 |      0 |      0 |      NaN |      0 |      0 |      0 |      0 |      0 | 0.000000
  4 |      0 |      0 |      0 |      NaN |      0 |      0 |      0 |      0 |      0 | 0.000000
----|--------|--------|--------|----------|--------|--------|--------|--------|--------|---------
sum |      3 |      0 |     57 | 0.950000 |     60 |      0 |     60 |     60 |     60 |    --
avg |      0 |      0 |     11 | 0.916667 |     12 |      0 |     12 |     12 |     12 | 8.000000

b#  | fix-time | fxt/miss | com-time | read/d | writ/d | read/l | writ/l | reset  | take   | delete | utiliztn
----|----------|----------|----------|--------|--------|--------|--------|--------|--------|--------|----------
  0 | 0.000553 | 0.011062 | 0.000000 |      3 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000001
  1 | 0.000000 | 0.000000 | 0.000000 |      0 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000000
  2 | 0.000000 | 0.000000 | 0.000000 |      0 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000000
  3 | 0.000000 | 0.000000 | 0.000000 |      0 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000000
  4 | 0.000000 | 0.000000 | 0.000000 |      0 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000000
----|----------|----------|----------|--------|--------|--------|--------|--------|--------|--------|----------
sum |    --    |    --    |    --    |      3 |      0 |      0 |      0 |      0 |      0 |      0 |    --
avg | 0.000553 | 0.000000 | 0.000000 |      0 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000000

PAGE TYPE 1 (2000 pages):

b#  | misses | w/log  | hits   | hit rate | succes | failed | fix    | unfix  | commit | prefetch
----|--------|--------|--------|----------|--------|--------|--------|--------|--------|---------
  0 |   4709 |      0 |  32938 | 0.874917 |  37647 |      0 |  37647 |  37647 |  37647 | 8.000000
  1 |   4407 |      0 |  30467 | 0.873631 |  34874 |      0 |  34874 |  34874 |  34874 | 8.000000
  2 |   4410 |      0 |  30746 | 0.874559 |  35156 |      0 |  35156 |  35156 |  35156 | 8.000000
  3 |   4386 |      0 |  30582 | 0.874571 |  34968 |      0 |  34968 |  34968 |  34968 | 8.000000
  4 |   4470 |      0 |  31485 | 0.875678 |  35955 |      0 |  35955 |  35955 |  35955 | 8.000000
----|--------|--------|--------|----------|--------|--------|--------|--------|--------|---------
sum |  22382 |      0 | 156218 | 0.874681 | 178600 |      0 | 178600 | 178600 | 178600 |    --
avg |   4476 |      0 |  31243 | 0.874664 |  35720 |      0 |  35720 |  35720 |  35720 | 8.000000

b#  | fix-time | fxt/miss | com-time | read/d | writ/d | read/l | writ/l | reset  | take   | delete | utiliztn
----|----------|----------|----------|--------|--------|--------|--------|--------|--------|--------|----------
  0 | 0.002501 | 0.019997 | 0.000000 |   4709 |      0 |      0 |      0 |      0 |      0 |      0 | 0.011286
  1 | 0.002761 | 0.021848 | 0.000000 |   4407 |      0 |      0 |      0 |      0 |      0 |      0 | 0.011472
  2 | 0.002739 | 0.021831 | 0.000000 |   4410 |      0 |      0 |      0 |      0 |      0 |      0 | 0.011458
  3 | 0.002736 | 0.021816 | 0.000000 |   4386 |      0 |      0 |      0 |      0 |      0 |      0 | 0.011354
  4 | 0.002649 | 0.021305 | 0.000000 |   4470 |      0 |      0 |      0 |      0 |      0 |      0 | 0.011333
----|----------|----------|----------|--------|--------|--------|--------|--------|--------|--------|----------
sum |    --    |    --    |    --    |  22382 |      0 |      0 |      0 |      0 |      0 |      0 |    --
avg | 0.002674 | 0.021342 | 0.000000 |   4476 |      0 |      0 |      0 |      0 |      0 |      0 | 0.011381


Concise information for awk-based extraction follows:

AWK_DataDiskUtilization min 0.428744    max 0.448264    avg 0.439828    number 20
AWK_LogDiskUtilization  min 0.000000    max 0.000000    avg 0.000000    number 11

AWK_CpuUtilization      min 0.307486    max 0.387244    avg 0.325963    number 5
```

Bild 21. *Fragment einer Statistikausgabe*

Die andere Gruppe der Parameter soll so gewählt werden, daß sie die Besonderheiten einer parallelen SD-Architektur **Absch. 2.3.1** in vollem Maße reflektiert. Hier spricht man vor allem vom Verhältnis zwischen der Plattenzahl im System, wo die Datenbestände und Indizes untergebracht werden, und der Anzahl der Prozessorknoten, welche die Anfrageverarbeitung übernehmen. Es ist nun anzumerken, daß den OR-Simulationen nur eine Art der Allokation zugrunde gelegt wird, nämlich die ohne Rücksicht auf die Anzahl der Prozessorknoten im System **Absch. 3.2.3.** Die zweite Allokationsvariante wird während des Parameterbestimmungsvorganges schnell untersucht und hat gezeigt, daß sie bei allen gleichen sonstigen Bedingungen schlechter als die erste ist. Für alle OR-Queries wird das Verhältnis 20 zu 5 zwischen Platten und Prozessorknoten festgelegt.

Das Bucky-Teilschema **Absch. 3.1.1** wurde als Objekt für die Effizienzuntersuchungen der verschiedenen Verarbeitungsverfahren s.u. genommen. Alle Klassen und ihre Attribute werden dementsprechend konfiguriert. Für jedes Attribut wird entschieden, welche der Indexstrukturen auf ihm eingesetzt werden. Es wird angenommen, daß ein *Clustered-Index* **Absch. 3.3.1** nur auf einem beliebigen gewählten Attribut der Klasse *Person* wie z.B. *PersonID* angewendet wird, so daß seine Anordnung mit der Plazierung der Objekte im Rahmen der Hierarchieallokation übereinstimmt. Da ein *Non-Clustered-Index* sich als nicht besonders effizient für die kleineren Objektmengen einer Klasse erwiesen hat und nur bei einer kleiner Selektivität einen Vorteil gegenüber den totalen Scan bringt, ist er bei den anstehenden Untersuchungen weniger interessant und wird nicht verwendet.

Der nächste Parameter, der während der Untersuchungen viel Anpassungsarbeit erfordert, ist die Größe eines Puffers. Es ist wichtig, daß die Seiten, die während eines Simulationslaufes in den Puffer gelangen, nicht immer für die ganze Zeit dort bleiben. Das hätte nicht den Prozessen in einem wirklichen DBS (realistische Trefferraten) entsprochen und zu gewaltigen Verzerrungen bei den Wiederholungsanfragen geführt. Also wird die allgemeine Puffergröße pro CPU folgendermaßen gewählt, daß ein Puffer ungefähr ein zehntel aller Objekte der Klasse *Person* (22.000 Seiten) beinhaltet, ohne sie verdrängen zu müssen (die Größe einer Seite beträgt 4096 bytes).

Der nächste Parameter für unsere Untersuchungen, der Simulationsergebnisse stark beeinflußt und eine direkte Beziehung zu der gewählten *Scheduling*-Strategie hat, wird "*max number of complex jobs per CPU node*" **Absch. 4.2.6** genannt. Damit wird die allgemeine Auslastung der CPU-Knoten im System reguliert. Nach mehreren Experimenten wird dieser Parameter für unser System spezifisch mit dem Wert 4 belegt, was genau dem Platten / Prozessoren Verhältnis entspricht. Dementsprechend werden während der Simulationen ca. zwei Datenfragmente pro Platte in die Verarbeitungsroutine einbezogen.

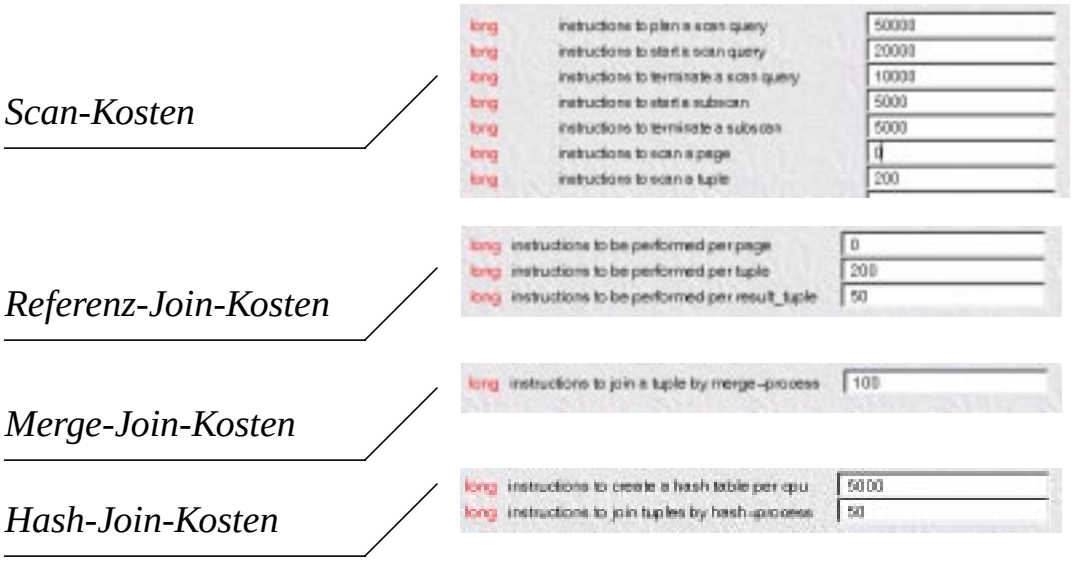


Bild 22. *Kostenparameter für Query-Operatoren*

Die Scheduling-Strategie, die bei der Anfrageanalyse stets verwendet wird, ist "*das Scheduling nach Konflikten mit laufenden Jobs"* **Absch. 3.4**. Dafür soll der α_l-Parameter den Wert Eins annehmen, alle anderen α den Wert Null. Diese Strategie hat sich als die effizienteste erwiesen. Alle Leseoperationen im System werden mit dem *Prefetching* **Absch. 2.2.2, 3.3.3** gleich acht Seiten pro Zugriff durchgeführt. Diese Technik hat einige besondere Vorteile für die gute Pufferstrategie geschaffen.

Abschließend soll ein ausgeglichenes Verhältnis für den CPU-Aufwand bei der Verarbeitung der verteilten Anfrageoperatoren gefunden werden. Auf **Bild 22** werden die Parameter aufgelistet, die unserer Meinung nach für die Tests s.u. geeignet sind und grob der korrekten Operatoren-gewichtung entsprechen. Sie lassen sich durch das Wissen über den inneren Aufbau des Simulationssystems sowie durch die allgemeinen Überlegungen über die Operatoreneffizienz erklären.

Beispielsweise sind wir davon ausgegangen, daß die CPU-Kosten für die Zusammensetzung von zwei Objekten in einem Merge-Join ungefähr gleich mit denen in einem Hash-Join sind. In beiden Fällen wird eine ganze Reihe Tupelvergleiche auf den Prozessorknoten durchgeführt. Die CPU-Kosten in einem Referenz-Join werden deswegen mindestens zwei mal kleiner.

5.2 Untersuchungsschwerpunkt

Nachdem alle Parameter im System festgelegt werden, beginnt man mit den eigentlichen Simulationstests, welche als Bestandteil der Studien zum Thema dieser Arbeit aufgefaßt werden können. Bei der Themenauswahl hätte man zwischen mehreren Untersuchungsaspekten entscheiden sollen. Einige davon waren die folgenden:

- Untersuchung der verschiedenen Allokations- und Speicherungsverfahren für Mengenwertige Attribute;
- Vergleich zwischen diversen Allokationsstrategien für Hierarchieklassen im Bezug auf die CPU-Platten-Zugehörigkeit;
- Untersuchung der verschiedenen Scheduling-Strategien sowie ihres Einflusses auf die Gesamtausführungszeit einer Anfrage;
- Beobachtung der Verarbeitungsverfahren unter den verschiedenen komplexen Bedingungen wie z.B. das CPU-Platten-Verhältnis, Eins- und Mehrbenutzerbetrieb, Variieren des Parallelitätsgrades;
- Untersuchung der Indexzuteilung zu den OR-Attributen etc.

Am Ende wird entschieden, die Bemühungen dem Untersuchungsschwerpunkt zu widmen, der eine Antwort auf folgende Frage liefern sollte:

- Mit welchem Join-Verfahren und unter welchen Bedingungen eine Zusammensetzung der Objekte in einem OR-Schema am besten erfolgt, wenn eine geeignete Hierarchie-Allokation des ORDB-Schemas dem Verfahren zugrunde gelegt wird.

Die folgenden Überlegungen begründeten unsere Entscheidung, dieses Thema vor den anderen zu bevorzugen. Der Untersuchungsschwerpunk umfaßt die allerwichtigsten Aspekte der Speicherung eines OR-Schemas und ihrer Anfrageverarbeitung. Es werden z.B. unsere Allokationsstrategie und das Referenz-Join-Verfahren im Rahmen der Arbeit getestet, die für ein ORDBS spezifisch sind und keine Analogien in der R-Welt finden. Die Besonderheiten einer *SD*-Architektur bleiben dabei nicht vergessen. Andererseits bekommt man aber aufgrund der Schlüsse aus der Literatur **Absch. 3.3.2** eine gute Untersuchungsbasis, worauf man sich während der eigenen Experimente ausgezeichnet stützen kann.

5.3 Vergleich zwischen Join-Methoden

Als Kategorien zu der Untersuchungsarbeit haben wir die folgenden drei Fälle unterschieden: Vereinigung durch *Vererbung*, *Mengenwertige Attribute* und unabhängige *Referenzen*. In allen drei Fällen geht es um Datenmengen, die logisch zusammenhängen, aber getrennt allokiert sind und zur Laufzeit zusammengesetzt werden müssen (impliziter Join).

Zu jeder der Kategorien werden jeweils zwei (im ersten Fall) oder eine (in den übrigen Fällen) OR-Anfrage ohne Angabe des Selektivitätsgrades aus dem Bucky-Schema vorbereitet, die beim Einsatz der verschiedenen Join-Verfahren (jeweils zwei) getestet werden. Die erste Anfrage wird außerdem in zwei Richtungen durchgeführt um zu sehen, inwieweit sich die Ergebnisse der Simulation in Abhängigkeit von der Richtung der Anfrage unterscheiden. Die Selektivität wird durch das Einsetzen der entsprechenden Parameter für die jeweils erste Hierarchieklasse bzw. Hierarchiekomponente stets variiert.

I. Vererbung

Finde Studenten und die ihnen entsprechenden Personen ***(erste Richtung)***

Finde Personen und die ihnen entsprechenden Studenten ***(zweite Richtung)***

II. Mengewertige Attribute

Finde Personen und alle ihren Kinder

III. Unabhängige Referenzen

Finde Studenten und die sie betreuenden Professoren

5.3.1 Hierarchie-Join

Die ersten Anfragen rücken die Effizienz eines Hierarchie-Join in den Vordergrund. Ein Hierarchie-Join wird benötigt, wenn man die OR-Klassen, die logisch zusammengehören aber separat allokiert werden, in einer Anfrage zusammenführt. Die wichtigste Bedingung, die einen Hierarchie-Join sehr effizient macht, ist die gleiche Sortierungsreihenfolge für alle an dem Join beteiligten Objekte. Somit erlaubt man hier den Einsatz eines Merge-Joins ohne einer expliziten Sortierung. Ein Referenz-Join profitiert von der gleichen Anordnung der Objekte **Absch. 3.3.3** und wird zum Hierarchie-Join-Vergleich ebenfalls angeboten.

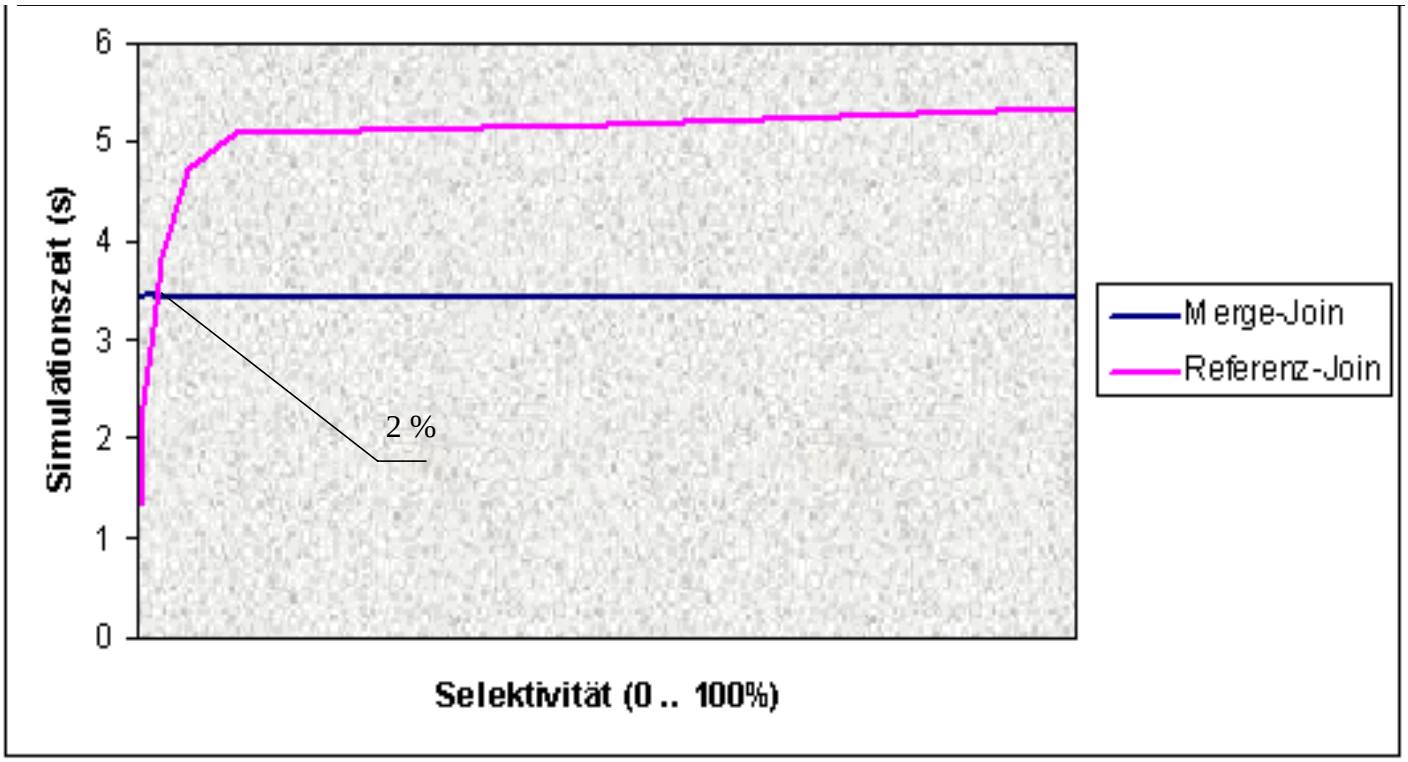
Auf **Bild 23** werden die Graphiken abgebildet, die nach der Untersuchung der Ergebnisse der ersten Hierarchieanfragen entstanden sind. Das obere Teilbild kommt durch die Änderungen des Selektivitätsparameters für die Klasse *Student* zustande, das untere entsteht durch die Änderungen der Angaben zur Klasse Person. Die Rahmenbedingungen, unter denen die Hierarchieanfragen durchgeführt werden, sind die folgenden:

- der Klasse *Person* wird ein *Clustered-Index* auf dem Verteilattribut zugrunde gelegt;
- die Klasse *Student* hat keinen verwendbaren Index und soll komplett von den Platten gelesen und bearbeitet werden;
- laut dem Bucky-Schema ist die Anzahl der Objekte der Klasse *Person* **Absch. 3.1.1** zwei mal größer als die der Klasse *Student*;

- in Seiten übertrifft die Gesamtgröße aller *Person*-Objekte die der Objekte aus der Klasse *Student* sechsfach;
- wenn man eine Selektion auf den Objekten der Klasse *Student* vornimmt, wird die Größe des Resultats nach dem Join als 100% angenommen, da für jeden Student genau ein Partner-Objekt aus der *Personen*-Klasse ermittelt wird;
- umgekehrt, wird die Wahrscheinlichkeit der Ergebnisobjekte für die zweite Anfrage 50% betragen (nur die Hälfte aller *Personen* sind *Studenten*), **Absch. 3.1.1.**

Das erste, was man aus dem Kurvenvergleich (oberes Teilbild) notiert, ist die totale Überlegenheit eines Merge-Joins gegenüber seinen Referenz-Join-Konkurrenten. Die Erklärung dafür steckt wahrscheinlich in den Ausführungsmethoden der beiden Operatoren selbst. Weil bei einem Referenz-Join mit dem weiteren Scan des Ausgangsobjekts **Absch. 3.3.3** darauf gewartet wird, bis die entsprechenden Seiten aus dem Zielobjekt gelesen werden, passiert das bei einem Merge-Join gleichzeitig - also ohne Verzögerung der Simulationszeit. Es besteht auch die Möglichkeit, einen solchen Algorithmus zu implementieren, dem ein progressives Scan-Verfahren zugrunde gelegt wird. Das Scannen der beiden beteiligten Klassen (Merge-Join) bzw. der Seiten des Ausgangsobjekts (Referenz-Join) wird in dem Fall durchgeführt, ohne das Ende der Join-Verarbeitungsroutine für die vorangehenden Seiten abwarten zu müssen. Wir haben aber auf diese Implementierungsart verzichtet, da auch in der Realität solche Ansätze kaum zu finden sind.

Lediglich bei einer geringeren Selektivität sollte man einen Referenz-Join vorziehen. Bei unseren Versuchen fand die Überschneidung der Kurven für die beiden Join-Techniken etwa bei den Wert gleich 2% statt. Dieses Ergebnis ist darauf zurückzuführen, daß die Anzahl der Seiten aus der Klasse *Person*, die beim Referenz-Join-Verfahren gelesen werden, wesentlich geringer ist, als beim vollen Scan während des Merge-Joins. Außerdem verwandelt sich die Kurve für den Merge-Join in eine horizontale Linie. Das war auch zu erwarten, weil die beiden an der Anfrage beteiligten Klassen bei dem Join komplett gelesen werden (es gibt keinen Index auf der Klasse *Student*).



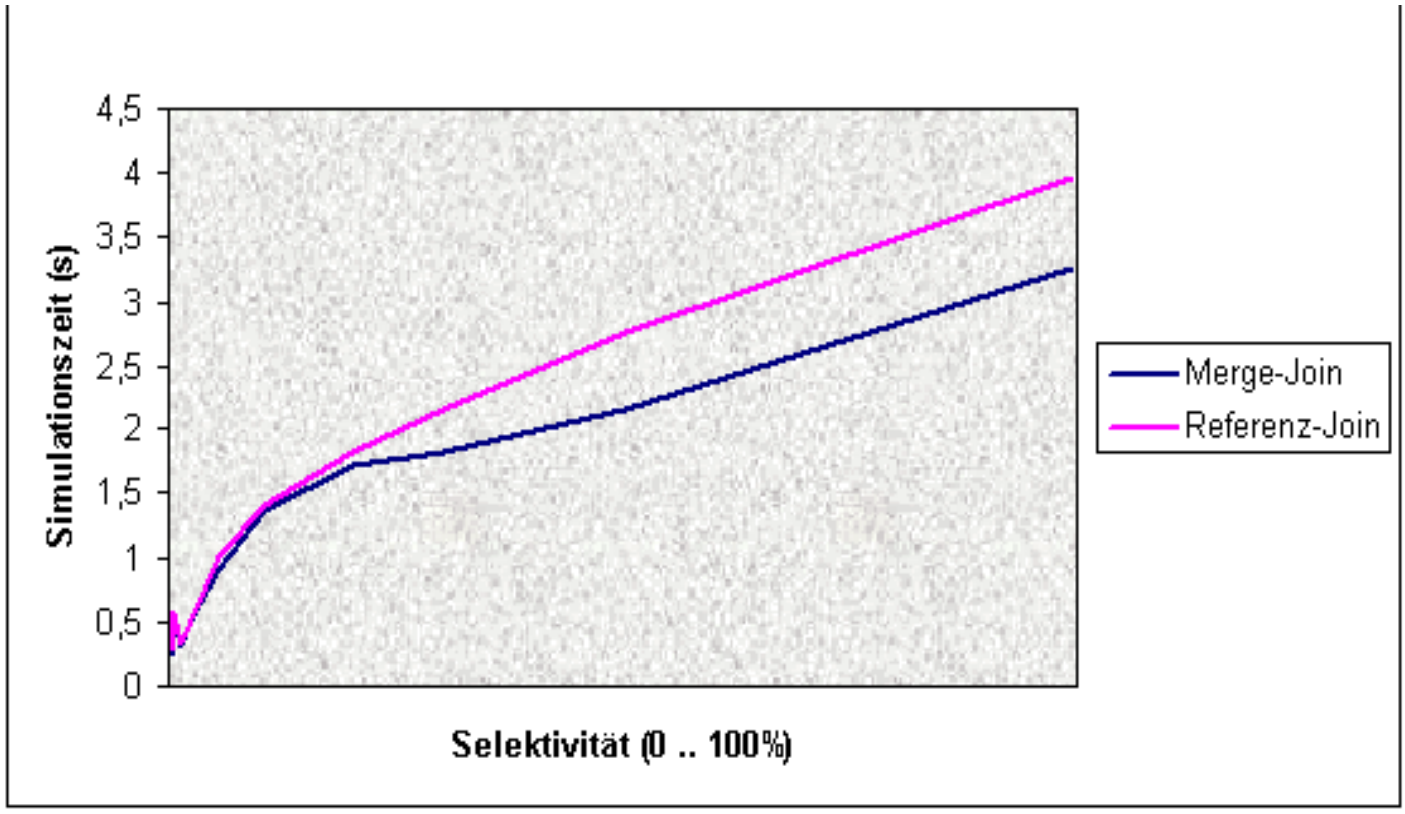


Bild 23. *Zwei Richtungen der Hierarchie-Join-Ausführung*

Bei einer geringeren Selektivität verhalten sich die Kurven auf dem unteren Teilbild chaotischer, da die Zufallsfaktoren die Ergebnisse immer stärker beeinflussen. Außerdem liegen die Kurven niedriger als die auf dem oberen Teilbild. Erst gegen die 100%-Selektivität erreichen die Kurven des unteren Diagramms ungefähr das gleiche Niveau wie die des oberen. Einerseits bestätigt das noch ein mal die Korrektheit der erhaltenen Ergebnisse, andererseits läßt es sich aber als allgemeiner Hinweis für die Planung einer Hierarchiequery zusammenfassen. Wenn eine ähnliche Hierarchieanfrage unter den gleichen Bedingungen (wenn es möglich ist, sie zu schaffen) ausgeführt wird, hat man dann die zweite Ausführungsreihenfolge - also die Selektion der Sub-Klasse nach der Basis-Klasse - zu bevorzugen. Dieses Ergebnis ist durch den Einsatz eines *Clustered-Indexes* auf dem Verteilattribut der Basis-Klasse zu erklären, der die Seitenzugriffe auf die Objekte der beiden beteiligten Klassen stark reduziert.

5.3.2 Join für Klassenkomponente

Die zweite Anfrage **Absch. 5.3** ist dafür gedacht, die Join-Verfahren für die Zusammensetzung der Komponenten innerhalb einer Hierarchieklasse zu untersuchen. Bei solchen Anfragen kann man davon ausgehen, daß die Sortierungsreihenfolge für die unterschiedlichen Klassenkomponenten gleich ist, was wiederum den Einsatz aller Join-Verfahren wie früher ermöglicht.

Als Basis für die zweite Anfrage wird die Klasse *Person* genommen. Obwohl eine *Person* im Durchschnitt drei *Kinder* hat, sind einem *Person*-Objekt nur zwei *Kinder* im Rahmen eines Set-Attributes zuzuordnen. Über das Verwenden eines Indexes wird genauso entschieden wie bei der ersten Anfrage. Die Anfrage wird nur in eine Richtung (Selektion der esten Komponente) durchgeführt, da die Größen der beiden Klassenkomponenten in Seiten ungefähr gleich sind und ein *Clustered-Index* nur auf den Ausprägungen eines Kopfteilattributes eingesetzt werden kann.

Dementsprechend werden auch bei der zweiten Anfrage ähnliche Ergebnisse erwartet wie bei der ersten. Diese Annahme hat sich bestätigt. Die Kurven auf **Bild 24** ähneln denen des Hierarchie-Joins (**Bild 23** unten). Der Vergleich zwischen den Verarbeitungsmethoden wird wiederum mit Hilfe des Merge- und Referenz-Joins ausgeführt. Die allgemeine Annahme, daß unter gleichen Bedingungen ein Merge-Join fast immer schneller ist, gilt auch hier.

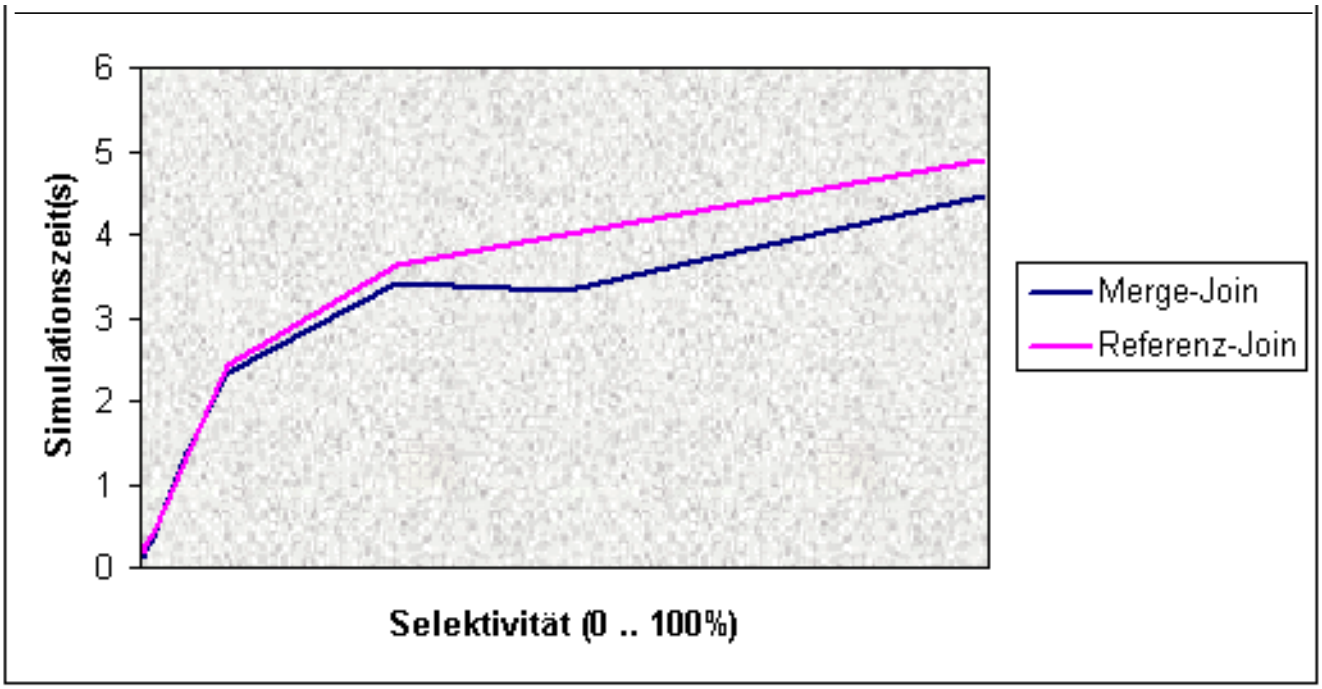


Bild 24. *Join-Verfahren für Klassenkomponenten*

5.3.3 Join für unabhängige Referenzen

Der dritten Anfrage wird eine ganz andere Art des Verbindungsaufbaus zwischen zwei Schemaklassen zugrunde gelegt. Jeder *Student* hat genau eine Referenz auf die *Professor*-Objekte, womit sein Betreuer eindeutig identifiziert wird. Die beiden Klassen besitzen keine Index-Strukturen und stehen nicht in einer Hierarchiebeziehung zueinander. Ihre Größen können nicht mit der Größe der *Personen*-Klasse in Seiten verglichen werden (die *Professor*-Klasse ist zwei mal kleiner als die Klasse *Student*). Also kann nicht erwartet werden, das die absoluten Meßwerte für die Simulationszeiten der Anfrage im selben Bereich wie bei den vorigen Queries liegen. Bei jedem Simulationsversuch werden die Objekte der Klasse Student zunächst selektiert und dann ihre Verbundpartner in der Professorenreihe ermittelt. Die Richtung der Anfrage wird so gewählt, weil die Zugriffe auf die *Studenten* von den *Professor*-Objekten mit dem Einsatz von Mengenwertigen Attributen (ein Professor hat mehrere Studenten zu betreuen) zusammenhängt, was die Ergebnisse unserer Query beeinflußt hätte.

Das Hash-Join-Verfahren wurde absichtlich aus dem Wettbewerb der Join-Methoden für die ersten zwei Queries ausgeschlossen. Dort, wo die Sortierung von Relationen bereits vorliegt, hat der Hash-Join keine Chance, die Bearbeitung einer Anfrage effizienter als die anderen Methoden durchzuführen und sich damit gegen seine Join-Rivalen zu behaupten. Um diese Aussage zu bestätigen und zu zeigen, daß sogar ohne eine passende Allokationssortierung ein Hash- im Vergleich zu dem Referenz-Join langsamer ist, wird die dritte Anfrage mit den beiden Join-Techniken getestet. Der Einsatz eines Merge-Joins ist bei dieser Anfrage nicht möglich.

Die Ergebnisse der dritten Anfrageausführung sieht man auf **Bild 25**. Der Referenz-Join wird hier ohne explizite Sortierung eingesetzt. Bei den Zugriffen auf die sortierten Seiten des Zielobjekts in den ersten beiden Anfragen spielt das *Prefetching* **Absch. 2.2.2** eine wichtige Rolle. Bei einer größeren Selektivität wird es immer wahrscheinlicher, daß die Seiten, die während eines *Prefetching* von den Platten gelesen und in den Puffer geholt werden, auch für die weitere Anfrageverarbeitung gleich benötigt werden. Damit erhöht sich die Trefferrate des Puffers stark, was folglich zu einer Reduktion des E/A-Aufwandes und somit der Anfrageverarbeitungszeit führt.

Die fehlende Sortierung führt bei der dritten Anfrage dazu, daß die Trefferrate des Puffers sich verringert und die Anzahl der benötigten Seitenzugriffe auf die Objekte der zweiten Klasse (des Zielobjektes) steigt. Das *Prefetching* wird zwar hier erneut eingesetzt, bringt aber keine entscheidenden Vorteile mit sich. Da die Zugriffe sehr verstreut sind, werden die Seiten, die während einer Leseoperation mit *Prefetching* in den Puffer gelangen, weiter aus dem Puffer verdrängt und beim Bedarf wieder gelesen. Das kostet Verarbeitungszeit.

Die beiden Kurven auf **Bild 25** sind fast linear. Für den Hash-Join führt die Verringerung des Nachrichtenaufwandes bei der niedrigen Selektivität zu einer kleinen Reduktion der Gesamtsimulationszeit, weil die Objekte der beiden beteiligten Klassen immer im vollen Umfang nacheinander gelesen werden sollen. Die gleichermaßen sinkende Kurve des Referenz-Joins spiegelt die sich stets reduzierende Zahl der Seitenzugriffe auf die zweite Relation und als Folge die Verkleinerung des somit entstehenden Leseaufwandes wieder.

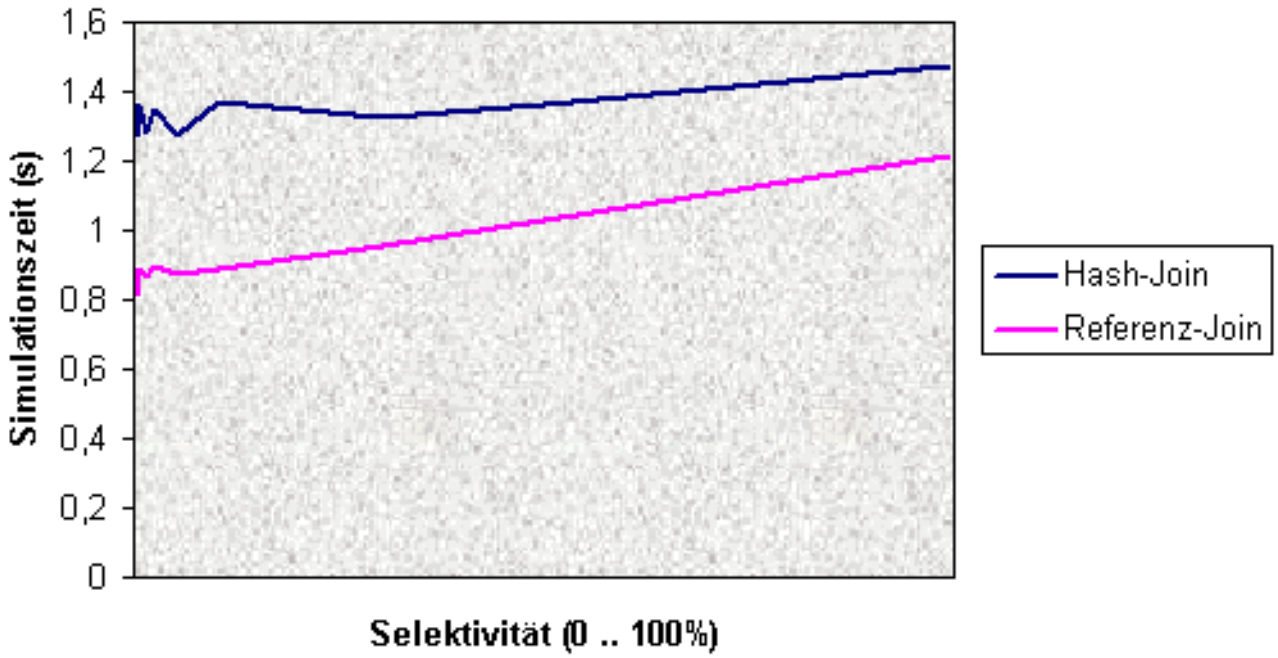


Bild 25. *Vergleich zwischen Referenz- und Hash-Join*

Die Phase des scharfen Abfalls der Referenz-Join-Kurve bei einer geringeren Selektivität tritt für die dritte Anfrage wegen der kleinen Relationengrößen wesentlich später auf, und ist wegen der Zufallsfaktoren in der Simulation auf dem Bild nicht mehr zu sehen. Außerdem wird vermutet, daß ein Hash-Join nicht immer schlechter abschneidet als ein Referenz-Join.

Das könnte passieren, wenn die Größe des Puffers wesentlich kleiner gewählt worden wäre, so daß die Trefferrate des Puffers sich bei dem Referenz-Join sehr verschlechtert. Somit könnte die Überschneidung der Kurven für die beiden Verfahren irgendwann wieder stattfinden. Im Rahmen dieser Arbeit trat dieser Fall jedoch nicht auf, wobei man ziemlich viel Zeit für das Drehen der Parameter aufgewendet hat, um diesen Effekt zu erreichen.

5.4 Resultierendes Regelsystem

Die Vorgehensweise bei den Untersuchungen, die in diesem Kapitel beschrieben wurden, möchten wir hier noch ein mal ganz kurz skizzieren. Die hier vorgestellten Simulationen erforderten eine komplexere Anpassungsarbeit für die Parameter, bevor sie getestet wurden. Der Untersuchungsschwerpunkt lag auf dem Vergleich der parallelen Join-Verfahren für eine geeignete OR-Allokation. Dafür wurden vier Queries für drei Arten der Anfragen, die für ein OR-Schema typisch sind, spezifiziert und zum Laufen gebracht. Am Ende wurden die Ergebnisse analysiert und bewertet.

Hier wird ein kleines Regelsystem aufgestellt, das als Resultat unserer Untersuchungen aufgefaßt werden kann und als einfacher Hinweis zum Einsatz der Join-Algorithmen unter ähnlichen Bedingungen angesehen wird:

- eine geeignete Allokation sollte Anfragen über die Hierarchie in einem OR-Schema generisch unterstützen;
- eine gute Auswahl der Lastbalancierungstrategie ist immer erforderlich, um die Effizienz eines Join zu steigern;
- alle möglichen Techniken zur Beschleunigung von Scan-Operatoren sollten bei einer Join-Anfrage eingesetzt werden;
- der Einsatz eines Clustered-Indexes lohnt sich immer;
- ein Clustered-Index-Scan kommt in einer Hierarchie-Anfrage zuallererst vor;
- der Einsatz eines Non-Clustered-Indexes lohnt sich vor allem dann, wenn die Selektivität der Anfrage niedrig ist und die zu lesende Klasse groß ist;
- nach Möglichkeit einen Merge-Join verwenden;
- für die Anfragen mit geringer Selektivität oder ohne geeigneter Sortierung einen Referenz-Join verwenden;
- ein Hash-Join nur dann anwenden, wenn die anderen Techniken sich als schlecht oder untauglich erwiesen haben.

Wir behaupten, daß man die besseren Ausführungszeiten für die Join-Anfragen in einem parallelen ORDBS erzielen kann, wenn man nach unseren Regeln vorgeht.

6. Fazit

Am Ende möchten wir einen Blick über den gesamten Verlauf der Arbeit werfen und die während dieser Arbeit erhaltenen Ergebnisse ganz kurz zusammenfassen. Im **Kap. 2** wurden die wichtigsten theoretischen Grundlagen, auf die wir bei dem Erstellen unseres Simulationsmodells zurückgreifen mußten, erläutert. Im **Kap. 3** hat man das Pro und Kontra für die verschiedenen Aufbaukonzepte eines parallelen ORDBS analysiert und verglichen. Die daraus entstandenen Vorschläge zur Implementierung des OR-Simulationsprototypes wurden dabei begründet. Im **Kap. 4** haben wir uns bemüht, die wichtigsten Eigenschaften unseres Modells durch die Beschreibung der implementierten Algorithmen darzustellen. **Kap. 5** hat dann die ersten Simulationsergebnisse geliefert und interpretiert.

Während dieser Arbeit wurde das neue Konzept zur verteilten Allokation von OR-Daten im Rahmen der *SD*-Architektur mit dem Schwerpunk auf Vererbungshierarchien entwickelt. Darüber hinaus hat man die Möglichkeit bekommen, die diversen Strategien zur Lastbalancierung einander gegenüberzustellen. Diese und auch mehrere andere Ansätze dieser Arbeit stellen ein umfassendes Gebiet für die Untersuchungsarbeit dar und werden zur weiteren Forschung angeboten.

Zu den wichtigsten Ergebnissen dieser Arbeit gehört die Anordnung der parallelen Join-Operatoren nach ihrer Effizienz. Eine passende OR-Umgebung ist die notwendige Voraussetzung für das im **Kap. 5** erstellte Regelsystem. Der Merge-Join ohne eine vorangehende Sortierung hat den Platz an der Spitze der Anordnung der Join-Methoden bekommen. Dieses Verfahren hat sich sowohl in der Literatur **Absch. 3.3.2**, als auch in der Praxis (die Simulationsergebnisse, **Absch. 5.2**) als bestes erwiesen. Die totale Überlegenheit des Merge-Join-Verfahrens läßt uns behaupten, daß die in dieser Arbeit vorgestellten Ansätze zur Datenallokation und Lastbalancierung gut sind und das Recht bekommen, für die weiteren Untersuchungen erforscht und verwendet zu werden.

Die Simulationen, die im **Kap. 5** dem Leser präsentiert wurden, stellen nur eine mögliche Variante dar, wie das von uns entwickelte OR-Simulationssystem für das Nachbilden der Abläufe in einem realen ORDBS verwendet werden kann. Mit dem gleichen Erfolg hätte ein anderer Schwerpunk als Vergleich zwischen verschiedenen OR-Join-Strategien gewählt werden können, **Absch. 5.2**. Das wird aber den weiteren Nutzern des *SimPaD*-Systems überlassen.

Bei der Erarbeitung der Konzepte zum Erstellen des Simulationssystems waren wir oft gezwungen, auf mehrere alternative Ansätze zu verzichten. Es ist sicher so, daß die endgültige Variante nicht alle möglichen Anforderungen an das OR-Simulationssystem abdeckt. Man hätte beispielsweise noch andere Varianten zur Datenallokation und Lastbalancierung implementieren können. Erweiterungen zu den Speicherungsstrategien **Absch. 3.1.5** sind auch möglich, u.s.w. Das Grundkonzept der Arbeit sowie die hier getroffenen Annahmen sollten aber dem Leser klar und deutlich sein, so daß die weiteren Versuche, unsere Simulation zu erweitern bzw. eine ähnliche Simulation zu erstellen, dadurch wesentlich erleichtert werden sollten.

Wir wünschen den ORDBS-Forschern eine produktive Arbeit mit unserem System und wollen hoffen, daß man mit seiner Hilfe weitere interessante Ergebnisse in näherer Zukunft meldet, auf die wir uns sehr freuen würden.

7. Anhang

7.1 Algorithmus zum Erstellen einer OR-Datenallokation

// main funktion for creating an allocation for an or-allocation class

return values: the number of new or-fragments in the class allocated

parameters:

long maxdom - max domain size of attribute allocated;
mix_element* mix_result - collection of element buckets
with all necessary allocation information
(should be created before this procedure);
bool mix_perform - wenn true, perform the buckets exchange
according to the shift parameter;
long shift - disk's offset , which is necessary for shifting the buckets
due to allocatition strategy

```
long relationOR::get_value_area(long maxdom, mix_element* mix_result,
                        bool mix_perform, long  shift)
{
        long cpu_numb = dbor->cpu_numb;                         // cpus' number
        bool disk_groupped = dbor->group;                       // groupped by cpus or not
        long disk_numb = lastdisk - firstdisk + 1 ;             // number of disks
        long disk_begin = shift % disk_numb + firstdisk;        // first disk for allocation
                                                                //                  begin
        printf("\n\n\n");
        long i, j, k, l;
        long numparts = 0;
        if( disk_groupped ) {                                   // divide all disks into groups
            long maxdom_from[cpu_numb];                         // according to cpu number
            long maxdom_to[cpu_numb];
            long disk_from[cpu_numb];
            long disk_to[cpu_numb];
            for( i = 0; i < cpu_numb; i++ ) {                   // calculate the scope for each
                maxdom_from[i] = maxdom/cpu_numb*i+             // disk group separately
                        ((maxdom%cpu_numb)*i)/cpu_numb;
                maxdom_to[i] = maxdom/cpu_numb*(i+1) +
                        ((maxdom%cpu_numb)*(i+1))/cpu_numb - 1;
                disk_from[i] = disk_numb/cpu_numb*i+
                        ((disk_numb%cpu_numb)*i)/cpu_numb;
                disk_to[i] = disk_numb/cpu_numb*(i+1) +
                        ((disk_numb%cpu_numb)*(i+1))/cpu_numb - 1;
                if( ! (j =(disk_to[i] - disk_from[i])*
                        (disk_to[i] - disk_from[i] +1))) j=1;

                numparts +=j;
            }
```

```
    mix_element mix_ele[numparts];                  // create an empty array of all
    numparts = 0;                                   // class fragments
    for( i = 0; i < cpu_numb; i++ ) {
        if( ! (j =(disk_to[i] - disk_from[i])*      // calculate disk_offset
                (disk_to[i] - disk_from[i] +1))) j=1;   // for a disk group
        fill_mix_str( mix_ele, numparts,  j, disk_from[i],
                disk_to[i] - disk_from[i] + 1, maxdom_from[i],
                maxdom_to[i] - maxdom_from[i] + 1);  // create an initial allocation
                                                    // for each disk group
        if(mix_perform)
                mix_ele_function(mix_ele, mix_result, numparts, j,
                disk_from[i], disk_to[i] - disk_from[i] + 1,
                 disk_begin);                       // mix the fragmets for each
                                                    // group due to disk's offset
        else {
                for( k=numparts; k<numparts+j; k++) {
                mix_result[k].disk = (mix_ele[k].disk + shift)%
                (disk_to[i] - disk_from[i] + 1) + firstdisk;
                mix_result[k].from = mix_ele[k].from;
                mix_result[k].to = mix_ele[k].to;
                }
                qsort(mix_result + numparts, j, sizeof(mix_element), mix_str_comp );
        }               // allign all fragments according to the disks they belong to
        numparts += j;
    }
}

else {
    numparts = (disk_numb) * ( disk_numb - 1);      // calculate number of disk
    if(!numparts) numparts = 1;                     //                  fragments
    mix_element mix_ele[numparts];
    fill_mix_str( mix_ele, 0,  numparts, 0, disk_numb, 0, maxdom);
    if(mix_perform)                                 // mix fragmets if necessary
        mix_ele_function(mix_ele, mix_result,0,  numparts,
                firstdisk, disk_numb, disk_begin);
    else {
        for( k=0; k<numparts; k++) {
                mix_result[k].disk = (mix_ele[k].disk + shift)%disk_numb + firstdisk;
                mix_result[k].from = mix_ele[k].from;
                mix_result[k].to = mix_ele[k].to;
        }
        qsort(mix_result, numparts, sizeof(mix_element), mix_str_comp );
    }           // allign all fragments according to the disks they belong to
}
```

```
        for( k=0; k<numparts; k++)
            mix_result[k].part = k;

        printf(" RESULTS \n\n");                                    // print out result allocation
        for(  k=0; k<disk_numb; k++)
            printf("___________ disk %ld _____", (firstdisk + k));
            printf("\n\n");
        for( l=0; l< numparts/ disk_numb ; l++) {
            for( k=0; k< disk_numb  ; k++)
                printf( "%ld - %ld", mix_result[k*numparts/ disk_numb + l].from,
                        mix_result[k*numparts/ disk_numb + l].to);
                printf("\n");}
        printf("\n\n");
        return numparts;
};
```

// function for buckets distribution and mixing

```
void relationOR::mix_ele_function(mix_element* mix_ele, mix_element* mix_result,
                        long first_part, long parts_numb, long first_disk,
                        long disk_number, long disk_begin)
 {
        long i, j;
        long part_disk = mix_ele[0].disk;
        for( i = first_part, j = 0; i<  first_part + parts_numb; i++, j++) {
            if(part_disk != mix_ele[i].disk) {
                part_disk = mix_ele[i].disk;                        // j is a temporary disk's offset
                j = 0;}
            if( (j + disk_begin - first_disk) % disk_number == 0 ) j++;
            mix_result[i].part = i;
            mix_result[i].disk =first_disk +
                (disk_begin + mix_ele[i].disk - first_disk + j)
                        % disk_number;
            mix_result[i].from = mix_ele[i].from;                   // assign the fragment scope
            mix_result[i].to = mix_ele[i].to;                       //    to the paramters of
                                                                    //       mix_elements structure
        }

        qsort(mix_result + first_part, parts_numb,
                        sizeof(mix_element), mix_str_comp );        // allign the fragments due to
                                                                    //      disk number they have
        for( i = first_part; i< parts_numb + first_part; i++)
            mix_result[i].part = i;
};
```

// function for creating an initinal allocation without any disk's offset

```
void relationOR::fill_mix_str(mix_element* mix_ele, long first_part, long parts_numb,
                              long first_disk, long disk_numb, long first_boader,
                               long maxdom)
{
long i, j, x;
x = parts_numb + first_part;                                    // calculate first fragment
                                                                //                  place

for( i = first_part; i < x; i++)                                // determine scope of each
{                                                               //          fragment
        mix_ele[i].part = i;
        mix_ele[i].from = (first_boader + maxdom)/ x *i+
                (((first_boader + maxdom)% x )*i)/x;
        mix_ele[i].to = (first_boader + maxdom)/ x *(i+1) +
                (((first_boader + maxdom)
                        % x)*(i+1))/ x - 1;
        mix_ele[i].disk =  (first_disk + disk_numb)/ x *i+
                (((first_disk + disk_numb)% x )*i)/x;
}

        printf(" INITIAL ALLOCATION \n\n");                     // print out inital allocation
        for(  i=first_disk; i< (first_disk+disk_numb); i++)
            printf("_____ disk %ld _____",i);
        printf("\n\n");

        for( i= 0; i<  disk_numb -1; i++) {
            for( j=0; j< disk_numb; j++)
                printf( "%ld - %ld",
                         mix_ele[j*(disk_numb -1) + i + first_part].from,
                        mix_ele[j*(disk_numb -1) + i + first_part].to);
        printf("\n");
        }
        printf("\n\n\n");
};
```

// function for comparisons between two mix_element structures

```
static int mix_str_comp( mix_element* ele1, mix_element* ele2)
{
        if( ele1->disk > ele2->disk ) return 1;
        if( ele2->disk > ele1->disk ) return -1;
        if(ele1->from > ele2->from) return 1;
      if(ele1->from < ele2->from) return -1;
        return 0;
};
```

7.2 Koeffizienten zur Berechnung von Plattenzugriffskonflikten

Meßgröße	Koeffizient/Definition	Beschreibung	Eigenschaft
Jobgröße	$p_d(j)$	Anzahl Seiten, die Job j auf Platte d liest	
	$p(j) = \sum_{d \in D} p_d(j)$	Anzahl Seiten, die Job j auf allen Platte liest	
Jobintensität	$i_d(j) = \frac{p_d(j)}{p(j)}$	relative Arbeitsintensität von Job j auf Platte d	$\sum_{d \in D} i_d(j) = 1$
Plattenlast	$P_d(J) = \sum_{j \in J} p_d(j)$	Anzahl Seiten, die von allen Jobs aus J auf Platte d gelesen wird; Gesamtbelastung der Platte d	
	$P(j) = \sum_{d \in D} P_d(j)$		
Plattenintensität	$I_d(J) = \frac{P_d(J)}{P(J)}$	relative Arbeitsintensität der Jobmenge J auf Platte d	$\sum_{d \in D} I_d(j) = 1$
Konkurrenz	$k_d(j, J) = i_d(j) \cdot I_d(J)$	relative Konkurrenz zwischen Job j und Jobmenge J auf Platte d	
	$k(j, J) = \sum_{d \in D} k_d(j, J)$	gesamte Konkurrenz zwischen Job j und Jobmenge J	$k(j, J) \leq 1$
spezielle Jobmengen	A	alle Jobs	
	E	alle bereits beendeten Jobs	$E \cup L \cup R = A$ (disjunkt)
	L	alle zur Zeit laufenden Jobs	
	R	alle noch nicht gestarteten Jobs	

Quelle: [Mä00]

7.3 Gewinnung von Spezialfällen aus der verallgemeinerten Maßfunktion

Zielfunktion	α_p	α_l	α_r	α_c	**Kommentar**
M_1	1	0	0	1	Die Parameter α_p und α_c können auch beliebige andere Werte größer Null annehmen, ohne daß sich die Planungsreihenfolge ändert.
M_2	0	1	0	0	Der Parameter α_l kann auch beliebige andere Werte größer Null annehmen, ohne daß sich die Planungsreihenfolge ändert.
M_2'	1				Mit anderen Werten für α_p kann der Einfluß des Faktors $p(j)$ variiert werden.
M_3	0	0	1	0	Der Parameter α_r kann auch beliebige andere Werte größer Null annehmen (analog zu M_2).
M_3'	1				Mit anderen Werten für α_p kann der Einfluß des Faktors $p(j)$ variiert werden (analog zu M_2).
M_4	0	1	1	0	
M_4'		> 0	> 0		Solange $\alpha_l = \alpha_r$ gilt, ist dies noch äquivalent zu M_4.
M_4"	1				Mit anderen Werten für α_p kann der Einfluß des Faktors $p(j)$ variiert werden (analog zu M_2, M_3).

7.4 Leiteraturverzeichnis

[ADHW99] Anastassia Ailamaki, David J.DeWitt, Mark D.Hill, David A.Wood, *DBMSs on A Modern Processor: Where Does Time Go?*, 25th VLDB Conference, Edinburgh,1999

[CD96] Michael J.Carey, David J.DeWitt, *Of Objects and Databases, A Decade of Turmoil*, the 22nd VLDB Conference, Bombay,1996

[CD96a] Michael J.Carey, David J.DeWitt, *The BUCKY Object-Relational Benchmark*, technical report, 1996

[EM99] Andrew Eisenberg, Jim Melton, *SQL:1999, formerly known as SQL3*, Sybase,1999

[GLS94] G. Graefe, A. Linville, L. Shapiro, *Sort vs. Hash Revisted*, IEEE transactions on knowledge and data engineering, vol.6, no.6, December 1994

[Go00] Vivekanand Gopalkrishnan, *Efficient Query Processing with Associated Horizontal Class Partitioning in an Object Relational Data Warehousing Environment*, International Workshop on Design and Management of Data Warehouses, Stockholm, Juni 2000

[Gr94] Goetz Graefe, *Sort-Merge-Join: An Idea Whose Time Has(h) Passed?*, Portland State University, Februar 1994

[Gr99] Goetz Graefe, *The Value of Merge-Join and Hash-Join in SQL Server*, 25th VLDB Conference, Edinburgh,1999

[Hä90] Härder, Theo, *An Approach to Implement Dynamically Defined Complex Object*, Proc. PRISMA Workshop, Noordwijk, 1990

[HR96] Evan P.Harris, Kotagirl Ramamohanarao, *Join algorithm costs revisited*, the VLDB Journal 5(1), Seiten 64-84, Springer-Verlag 1996

[HR99] Theo Härder, Erhard Rahm, *Datenbanksysteme, Konzepte und Techniken der Implementierung*, Springer Verlag, 1999

[Ja98a] Jens Jahny, *csim-TopologieManager und modulare Datenstrukturen*, internes Papier, Universität Leipzig, Abteilung DBS, December 1998

[Ja98b] Jens Jahny, *SimPaDManager*, internes Papier, Universität Leipzig, Abteilung DBS, December 1998

[JM98] Michael Jaedicke, Bernhard Mitschang, *Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS,* Technische Universität München, Computer Science Department, 1998

[KG90] Tom Keller, Goetz Graefe, *Efficient Assembly of Complex Objects*, Portland State University, 1990

[KL00] K. Karlapalem and Q.Li, *A Framework for Class Partitioning in Object-Oriented Databases*, Distributed und Parallel Databases 8 (3), Seiten 333 - 366, Kluwer Academic Publishers, 2000

[Mä00] Märtens, Holger, *Scheduling von Scan-Operatoren in komplexen objektrelationalen Anfragen für parallele DBMS*, internes Papier, Universität Leipzig, Abteilung DBS, 2000

[ME92] Mishra, Eich, *Join Processing in Relational Databases*, ACM Computing Surveys 24(1), 1992

[Me97] Mesquite Software, Inc, *CSIM18 Simulation Engine, User's Guide*, Austin, 1997

[Me97a] Mesquite Software, Inc, *CSIM18 Simulation Engine, Getting Started*, Austin, 1997

[OHE99] Robert Orfali, Dan Harkey, Teri Edwards, *Client/Server survival guide*, third edition, John wiley & Sons, Inc., 1999

[Ra94] Erhard Rahm, *Mehrrechner-Datenbanksysteme*, Addison-Wesley 1994

[Ra97] Erhard Rahm, *Objektrrelationale DBS/ SQL3*, Skript zur Vorlesung DBS2, Universität Leipzig, Abteilung DBS

[SD00] *SimPaD-Dokumentation*, innere HTML-Dokumente zum Erstellen der SimPaD-Modulen, URL: file:/u/psimadm/InfoPages/Dokumentation.html, Universität Leipzig, Abteilung DBS, 2000

[SLR97] Praveen Seshadri, Miron Livny, Raghu Ramakrishnan, *The Case for Enhanced Abstract Data Types*, 23rd VLDB Conference, Athens, 1997

[St93] Thomas Stöhr, *Simulation eines Shared-Disk-Systems zur parallelen Query-Bearbeitung*, Diplomarbeit, Universität Kaiserslautern, März 1993

[Sto96] Michael Stonebraker, *Object-Relational DBMSs, The next great wave*, Morgan Kaufmann, 1996

[TB94] Wouter B.Teeuw, Henk M.Blanken, *Complex Object Joins in a Distributed Database*, University of Twente, Department of Computer Science, 1994