Entwurf und Implementierung eines hochperformanten Datenbanksystems für das Java Betriebssystem JX

Ivan Dedinski

Mai 2001

SA-I4-2002-01

Studienarbeit

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)

Entwurf und Implementierung eines hoch-performanten Datenbanksystems für das Java Betriebssystem JX

Studienarbeit im Fach Informatik

vorgelegt von

Ivan Dedinski

geb. am 12.07.1979 in Sofia

Angefertigt am 30.01.2002

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	***Prof. Dr. F. Hofmann***
	Dipl.-Inf. Michael Golm

Beginn der Arbeit:	01.05 2001
Abgabe der Arbeit:	30.01 2002

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den Januar 27, 2002 ______________________________

Abbildungsverzeichnis

Kurzfassung

Das Thema dieser Studienarbeit ist der Entwurf und Implementierung einer hochperformanten Datenbank für die JX Betriebssystem. Diese Datenbank sollte die Grundfunktionalität für die Entwicklung von komplexeren Datenbanksystemen mit verschiedenen Datenmodellen bereitstellen.

Die JX-Datenbank (JXDB) hat eine Schichtenarchitektur, die typisch für die meisten zur Zeit vorhandenen Datenbanksystemen ist. Die kommunikation zwischen den Schichten erfolgt über vereinfachte Schnittstellen, die die komplizierteren Internals verbergen.

Die komplette Datenbank wird in der Programmiersprache Java implementiert, wie auch der gößere Teil des JX Betriebsystems. Deswegen ist eine besondere Herausforderung gute Performanz zu erreichen. Es wurden viele Techniken eingesetzt, die die Ausführungszeiten optimieren. Die strenge Schichtenarchitektur wurde dadurch jedoch nicht benachteiligt. Als größte Performanzprobleme haben sich das Erzeugen von vielen Objekten und das Kopieren von großen Datenmengen über die Schichtengrenzen erwiesen. Das Erzeugen von vielen Java Objekten wurde durch den Einsatz von Objectpools in Grenzen gehalten. Das Kopieren von Daten über die Schichtengrenzen wurde grundsätzlich vermieden indem nur geschützte Objektreferenzen übergeben wurden.

Um die Leistung des Datenbanksystems zu untersuchen wurden Vergleiche mit MySQL durchgeführt. MySQL ist ein schnelles und frei erhältliches Datenbanksystem, das unter Linux läuft und komplett in der Programmiersprache C geschrieben ist. Es war nicht möglich, alle MySQL Benchmarks auf die JXDB zu übertragen, da die MySQL Funktionalität zum Teil in JXDB nicht vorhanden ist. Daher wurden nur einige Tests durchgeführt, die adequat für die beiden Systeme sind.

Die Ergebnisse der Benchmarks deuten darauf hin, daß es durchaus möglich ist, ein Datenbanksystem für JX zu entwickeln, die ungefähr die gleiche Leistung hat wie eine Linuxdatenbank geschrieben in C

Kapitel 1

Einführung

1.1 Zielsetzung und Rahmenbedingungen

Diese Studienarbeit ist Teil eines größeren Projekts, dessen Ziel es ist, ein Betriebssystem (JX) unter der Verwendung der Programmiersprache Java zu implementieren. Als Basis dient ein minimaler, in C geschriebener Betriebssystemkern, der unter besonderer Beachtung der Kriterien Stabilität und Sicherheit entwickelt wurde. Dieser Teil soll nur maschinennahe Operationen realisieren und eine klare Schnittstelle für die anderen JX-Bestandteile zur Verfügung stellen, d.h. Die darauf aufsetzenden Java-Klassen sind als isolierte, austauschbare Module (packages) konzipiert, die die eigentlichen Betriebssystemfunktionen implementieren (Speicherverwaltung, Gerätesteuerung, Scheduler, Dateisystem, Netzwerkprotokollstack usw.).

Das Ziel dieser Studienarbeit ist es, ein Package für JX zu entwickeln, der die Basisfunktionen einer Datenbank besitzt. Dabei wird nicht beabsichtigt, die Funktionalität von großen komerziellen Datenbanksystemen wie Oracle oder Informix vollständig zu implementieren, weil so ein Aufwand für eine Studienarbeit nicht angemessen ist. Vielmehr ist es Ziel, die Performanzeigenschaften von der JX-Betriebssystem anhand einer größeren Applikation zu untersuchen. Da fast das ganze JX in Java geschrieben ist ist kann man Vermuten, daß dadurch große Performanznachteile für die Applikationen enstehen. In JX wird aber eine Menge an Optimierungen vorgenommen (native code Compiler, direkre Hardwarezugriffe, etc.), die diese Aussage widerlwgwn. Weitere Aufgabe dieser Studienarbeit ist es, durch Messungen an das Datenbanksystem das zu überprüfen. Deswegen werden am Ende der Arbeit einige Benchmarks durchgeführt, welche die Leistung der JX-Datenbank (JXDB) mit der Leistung von MySQL (eine der schnellsten freien Datenbanken zur Zeit) verglichen.

Die JXDB hat zur Zeit nicht die volle Funktionalität von MySQL, manche Operationen wie SQL Unterstützung (siehe [CDHD97]) und Übersetzung müssen noch per Hand von den Applikationsentwickler gemacht werden, es gibt noch keine richtige Lockingmechanismen, etc. Deswegen sind die Vergleiche nur auf Grundfunktionen beschränkt, und man muß gestehen, die MySQL Datenbank führt einige Operationen mehr in den Benchmarks als die JXDB (automatische Anfrageübersetzung und Optimierung), was ihr natürlich zusätzliche Zeit kostet. Deswegen wurden solche Datenbankanfragen aus der Standardbenchmarks von MySQL gewählt, die fast keinen Aufwand für Übersetzung und Optimierung benötigen. Daher sind die Ergebniwerte der Tests schon vertrauenswürdig. Außerdem ist die Leistung der JXDB momentan so gut, daß man sich noch Performanzverminderung leisten kann, wenn neue Funktionalität eingefügt wird.

1.1Kapitelübersicht

Im Kapitel 2 findet man eine Einführung in den Konzepten der modernen DBMS. Es wird auf die Schichtenarchitektur eingegangen, die eine systematische Entwicklungstechnik für größere Softwareprojekte darstellt. Im Zusammenhang der Datenbaksysteme handelt es sich um eine 5 – Schichtenarchitektur[HR99].

In den Kapiteln 3 bis 6 werden die in der JXDB vier implementierten Schichten der 5-Schichtenarchitektur erläutert. Da jede höhere Schicht auf die unteren aufbaut, baut auch jedes weitere Kapitel auf die Vorherigen.

Im Anhang A wird die Funktionsweise des B* Baums erläutert.

Im Anhang B werden einige Beispiele für die Applikationsentwickler gegeben und die Benchmarks erläutert. Dort findet man auch die Ergebnisse von dem Vergleich mit MySQL.

Kapitel 2

Einführung in der Architektur von DBMS

In diesem Kapitel werden einige der zur Zeit existierenden Arten von Datenbanksystemen vorgestellt(Ausführliche Diskussion dieses Themas bietet [DAT95] an). Datenbanksysteme sind in der Regel sehr komplexe Softwaresysteme. In der bisherigen Entwicklung hat sich ein Schichtenmodell als nützlich für die Bewältigung dieser Komplexität erwiesen, deswegen wird es hier erläutert.

2.2 Arten von DBMS

Datenbanken werden vor allem nach ihrem Datenmodell klassifiziert.Ein Datenmodell bestimmt wie auf die Daten zugegriffen wird. So sind die Wesentlichen die Klassen der relationalen, objektorientierten, objektrelationalen, multidimensionalen und satzorientierten Datenbanksystemen.

2.2.1 Relationale Datenbanken

Die am weitesten verbreiteten sind die relationalen Datenbanken, die auf das relationale Datenbankmodell basieren. Obwohl das Modell formal beschrieben ist, gibt es heutzutage kaum ein komerzielles System, das es vollständig realisiert, hauptsächlich aus Performanzgründen.

Das relationale Datenmodell wird wie folgt definiert:

1. Alle Daten werden vom Anwender als Tabellen wahrgenommen.
2. Eine Relation ist eine Tabelle mit Spalten und Zeilen.
3. Ein Attribut ist der Name einer (von vielen) Spalte der Tabelle
4. Ein Relationenschema besteht aus dem Relationennamen und den Namen der Attribute der Relation
5. Ein Tupel entspricht einer Zeile einer Tabelle
6. Der Grad einer Relation entspricht der Anzahl seiner Spalten
7. Die Kardinalität einer Relation entspricht der Anzahl seiner Zeilen
8. Eine relationale Datenbank besteht aus einer Menge von Relationenschemata zusammen mit ihren Inhalten (Tupel).
9. Eine Domäne (Wertebereich) ist die Menge erlaubter Werte für ein oder mehrere Attribute
10. Ein Primärschlüssel ist eine eindeutige Identifikation eines Tupels.
11. Ein Fremdschlüssel ist ein Attribut, das sich auf den Primärschlüssel einer anderen Relation bezieht.
12. Eine Integritätsbedingung beschreibt eine Anforderung der Anwendungswelt, welche in der Datenbank berücksichtigt werden muß.

2.2.2 Objektorientierte Datenbanken

Das oben vorgestellte Modell ist simpel, einfach zu implementieren und daher auch performant(mm Vergleich zu anderen Modellen). In den modernen objektorientierten Programmiersprachen gibt es aber eine Menge von Mechanismen, die im Modell nicht vorhanden sind, wie die Vererbung und Datenkapselung zum Beispiel. Da diese Mechanismen für die Entwicklung von Softwaresystemen sehr wichtig sind, wurde das objektorientierte Datenbankmodell entwickelt. Die Daten werden nicht mehr als Tabellen repräsentiert, sondern in Form von Objekten und Objektbeziehungen. Jedes Objekt besitzt eine eindeutige ID, mit der auf ihm zugegriffen wird.

2.2.3 Objektrelationale Datenbanken

Es ist sehr schwer, eine performante logische Verwaltung der Daten in dem objektorientierten Modell zu realisieren, weil die Komplexität der möglichen Objektbeziehungen enorm mit der Anzahl der Objektklassen steigt. In den meisten Aufgaben wird nur ein Teil dieser Beziehungen wirklich benötigt. Deswegen ist es manchmal sinnvoll die Daten in dem Objektrelationalen Datenmodell darzustellen. Es ist nicht so komplex wie das objektorientierte Modell, erweitert aber das relationale Modell mit benutzerdefinierten Typen, Vererbung, und anderen objektorientierten Mechanismen. Die Daten werden als Objekte modelliert, werden aber in Tabellen gespeichert. Dadurch wird eine performantere Lösung ermöglicht.

2.2.4 Multidimensionale Datenbanken

Es gibt spezielle Anwendungsfälle, (Kaufmännische Datenbanken, Datawarehousing), wo ein multidimensionales Modell der Daten bequem ist, um die spezifische Anfragen zu realisieren. Beispielsweise kann eine solche Anfrage so aussehen:

> „Ermittle alle elektrische Geräte, die über 500 Euro gekostet haben und von 1.2.2000 bis 1.2.2001 verkauft wurden“

Es wäre sinnvoll die Daten in einem Würfel mit 3 Dimensionen(Geräteart, Preis, Verkaufszeitraum) zu modellieren, die obige Anfrage würde einen Teilwürfel als Ergebnis zurückliefern.

2.2.5 Satzorientierte Datenbanken

Beispiel dafür ist CODASYL([DAT95]). Die Daten werden in logische Datensätze organisiert, der Benutzer kann auf sie iterativ zugreifen. Die satzorientierte Datenbanken verlieren an Bedeutung, jedoch gibt es viele funktionierende Systeme die mit diesem Modell arbeiten.

2.3 Schichtenmodell

Die Einzelheiten der verschiedenen Datenmodellen sind für diese Arbeit nicht von Interesse. Wichtig ist aber, das die Modelle nicht bestimmen, wie die Daten physisch gespeichert und verwaltet werden müssen. Es ist möglich, ein unabhängiges Grundsystem für alle Datenmodelle zu entwickeln. Dieses System könnte dann entsprechend als Grundlage benutzt werden, um jede modellspezifische Funktionalität zu realizieren. Das ist ein Beispiel, wie man ein Datenbanksystem in getrennten Schichten modellieren kann, um die Komplexität zu reduzieren und die Wiederverwendbarkeit des Kodes zu verbessern. Zwischen den Schichten müssen klare Schnittstellen definiert werden. Im Idealfall sollte jede höhere Schicht nur auf die Funktionalität der direkt unterliegenden Schicht basieren und nur die entsprechende Schnittstelle benutzen, um die Funktionen ihrer eigenen Schnittstelle zu realisieren.

2.4 Schichten der heutigen DBMS

Normalerweise ist es sinnvoll ein System in 5 bis 10 Schichten zu unterteilen. Mehr als 10 Schichten wären unübersichtlich, weniger als 5 würden nicht ausreichen. In vielen Datenbanksystemen ist die 5 – Schichten Architektur vertreten:

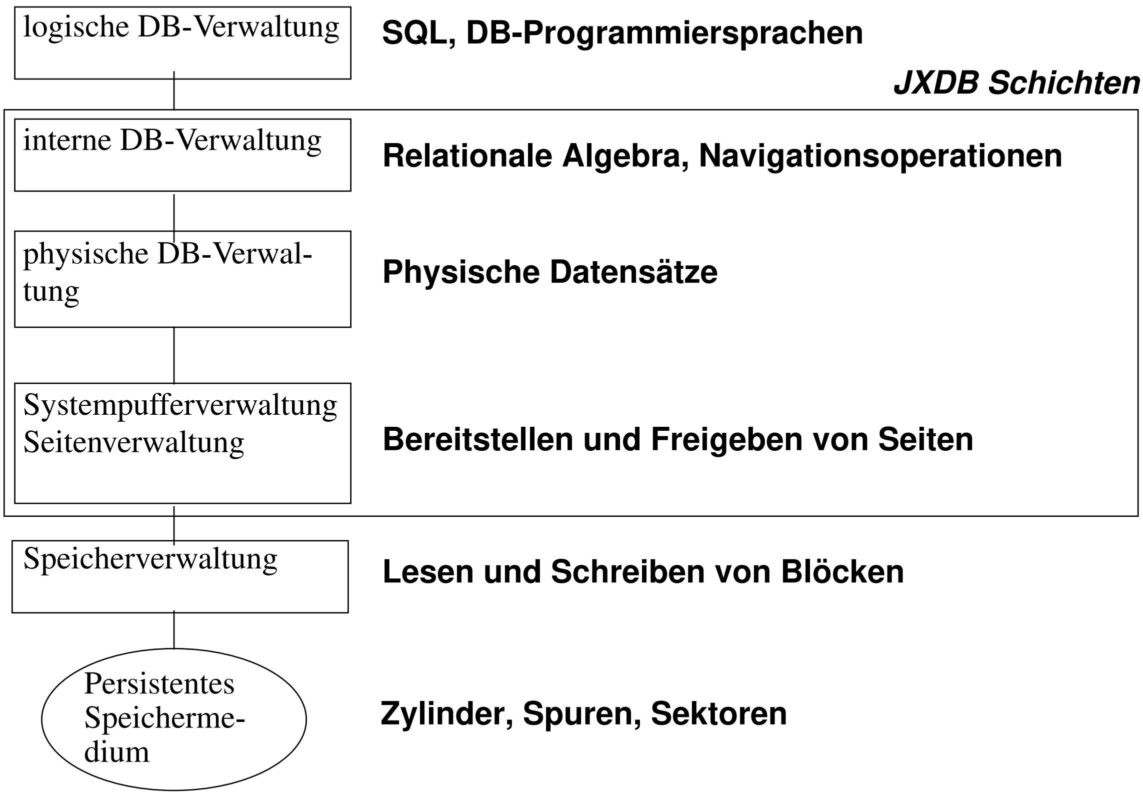


Abbildung 2.2 5-Schichtenarchitektur

2.4.1 Persistentes Speichermedium

Das persistente Speichermedium ist üblicherweise in Zylinder, Spuren, Sektoren u.ä. organisiert. Der Zugriff erfolgt über Gerätetreiber und Kanalprogramme.

2.4.2 Speicherverwaltung

Das Betriebssystem stellt eine Schnittstelle zur Verfügung, die von den Gerätecharakteristika abstrahiert. Diese Schnittstelle bietet Partitionen, Blöcke usw. an. Blöcke sind die Basiseinheiten für Zugriffe auf das persistente Speichermedium. Ein Block umfaßt üblicherweise zwischen 512 Bytes und 16 KB. Die Betriebssystemschnittstelle stellt Operationen für das Lesen und Schreiben von Blöcken bereit.

2.4.3 Seitenverwaltung

Die Seitenverwaltung abstrahiert von dem persistenten Speicher. Es wird ein virtueller Speicher verwaltet, der auf den persistenten Speicher abgebildet wird. Einheiten des virtuellen Speichers sind Seiten; eine Seite umfaßt zwischen 512 Bytes und 16 KB. Seiten werden auf Blöcke der Speicherverwaltung abgebildet, d.h. für die persistente Speicherung von Seiten wird die Speicherverwaltung benutzt. Entweder wird die Seitenverwaltung des Betriebssystems genutzt, oder das DBMS verfügt über eine eigene Seitenverwaltung.

2.4.4 Systempufferverwaltung

Der Systempuffer ist der Teil des Hauptspeichers, in dem Seiten für die Bearbeitung durch andere Module bereitsgestellt werden. Der Systempuffer ist in Pufferrahmen(Frames) eingeteilt, die jeweils genau eine Seite speichern können. Die Systempuffer- und Seitenverwaltung stellt eine Schnittstelle zur Verfügung, über die Seiten gelesen, geschrieben und persistent gespeichert werden können.

Bemerkung: Systempuffer- und Seitenverwaltung werden i.a. in einem gemeinsamen Modul realisiert.

2.4.5 Physische DB-Verwaltung

Die physische DB-Verwaltung bietet Operationen an, um physische Datenstrukturen zu erzeugen und persistent zu speichern. Die physischen Datenstrukturen werden auf die Seiten der Seitenverwaltung abgebildet. Es werden Zugriffs- und Änderungsoperationen für physische Datenstrukturen sowie Lese- und Schreiboperationen für die physischen Datensätze angeboten. Operationen auf den physischen Datenstrukturen werden in Operationen auf Seiten umgesetzt.

2.4.6 Interne DB-Verwaltung

Die interne DB-Verwaltung bietet Operationen an, mit denen Dateien (dies können auch Tabellen sein) erzeugt und mit Hilfe von physischen Datenstrukturen persistent gespeichert werden können. Es werden Operationen für den Zugriff auf Dateien und die Manipulation von Dateien angeboten. Als Sprache für die Spezifikation von Zugriffen kann bspw. die relationale Algebra genutzt werden. Interne Datensätze können interpretiert werden, d.h. die Felder der internen Datensätze sind sichtbar. Operationen auf internen Dateien und Datensätzen werden in Operationen auf den physischen Datenstrukturen umgesetzt.

2.4.7 Logische DB-Verwaltung

Die *logische DB-Verwaltung* bietet an ihrer Schnittstelle das Datenmodell des DBMS an: logische Datensätze und Kollektionen sowie deklarative Anfragesprachen (Bsp.: SQL) bzw. spezielle Datenbankprogrammiersprachen. Deklarative Anfragen werden in eine operationale Sprache (bspw. die relationale Algebra) übersetzt. Operationen auf den Kollektionen und logischen Datensätzen werden in Operationen auf den internen Dateien und Datensätzen transformiert. Für die Transformation einer Anfrage, die auf der logischen Ebene formuliert ist, in Operationen auf der internen Ebene existieren häufig viele Alternativen, die sich in ihren Ausführungskosten unterscheiden (Man spricht hierbei von *Anfrageoptimierung*).

2.5 Vertikale Schichten

Nicht alles in einem Datenbanksystem läßt sich in einer der genannten 5 Schichten realisieren. Die Transaktionsverwaltung müßte z.B. in fast allen Schichten implementiert werden, also kann sie als eine vertikale Schicht betrachtet werden. So könnte z.B. auf der Speicherverwaltungsebene Plattenredundanz eingeführt werden. In die Systempuffer- und Seitenverwaltungsschicht gehören Techniken wie das Schattenspeicherkonzept und Protokolierung (write ahead logging). In die interne DB-Verwaltung gehören Sperrkonzepte, welche die Benutzerthreads verhindern, Datenkonflikte zu erzeugen.

Da in JXDB noch kein Transaktionskonzept vorhanden ist, wird dieses Thema nicht näher betrachtet.

Kapitel 3

Speicherverwaltung

3.1 Aufgaben der Speicherverwaltung

Die Speicherverwaltung schafft eine Abstraktion von dem Aufbau des konkreten Speichermediums. Eine übliche Festplatte besteht aus mehrere Magnetplatten, die jeweils in Spuren unterteilt sind. Die Spuren ihrerseits bestehen aus Sektoren, die oft 512 Bytes enthalten. Die physikalische Adresse eines Bytes in einer solchen Platte würde daher aus der Magnetplattennummer, Spurnummer, Sektornummer und Bytenummer innerhalb des Sektors bestehen. Für Datenbanken, die komplett in Hauptspeicher gehalten werden (die übrigens immer mehr an Bedeutung gewinnen, weil das Hauptspeicher billiger wird), sieht die Adresse eines Bytes ganz anders, es ist nur eine einzige Zahl.

Die Speicherverwaltung stellt eine einheitliche Byteadressierung für alle Speichermedientypen zur Verfügung. Der gesamte Adressraum wird in Partitionen unterteilt, die Partitionen in Blöcke fester Größe(üblich sind 512 Bytes). Die Byteadresse besteht aus Partitionnummer, Blocknummer und Bytenummer innerhalb des Blocks.

3.2 Konzepte und Techniken

Bei manchen Speichermedien gibt es weitere Eigenschaften, die beachtet werden müssen, um effizienten Zugriff auf die Daten zu ermöglichen. Bei den Festplatten ist es sinnvoll die Bewegungen des Schreib- und Lesekopfs zu optimieren. Eine Möglichkeit dafür ist es Schreib- und Leseaufträge zu sortieren, um Plattenzugriffe zu sequenzialisieren. Dies könnte gute Ergebnisse liefern, wenn mehrere Aufträge gleichzeitig auf ihre Abarbeitung warten, d.h. entweder gibt es mehrere laufende Threads, die Aufträge produzieren, oder ein Thread produziert mehrere Aufträge, ohne daß er sich blockiert. Dies könnte man entsprechend bei der Implementierung der oberen Schichten berücksichtigen.

Eine andere Eigenschaft von Festplatten ist es, das zwischen der Abarbeitung von 2 Aufträge eine gewisse Verzögerung gibt. Das hat zur Folge, daß zwei benachbarte Sektoren nur auf Kosten einer vollen Umdrehung der Magnetplatte gelesen oder geschrieben werden können. Damit der sequentielle Zugriff nicht langsam wird, muß die Zuordnung von logischen Blöcken zu physikalischen Sektoren passend gewählt werden.

3.3 Implementierung

Die Speicherverwaltung der JXDB ist komplett im JX Betriebsystem realisiert. Das Interface BlockIO steht zur Verfügung und beinhaltet folgende Methoden:

int getCapacity();

int readSectors(int startSector, int numberOfSectors, Memory buf, boolean synchroneos);

int writeSectors(int startSector, int numberOfSectors, Memory buf, boolean synchroneos);

Es repräsentiert eine Partition im Speichermedium, getCapacity liefert die Anzahl der

Blöcke in der Partition, readSectors liest einige Sektoren von der Platte, writeSectors schreibt die Sektoren aus. (siehe source Doku). Objekte, die dieses Interface implementieren, werden bei einem systemweiten Nameserver angemeldet und können so von den höheren Schichten benutzt werden. Ob die Daten auf die Platte geschrieben werden, oder in einen virtuellen RAM Disk, ist transparent.

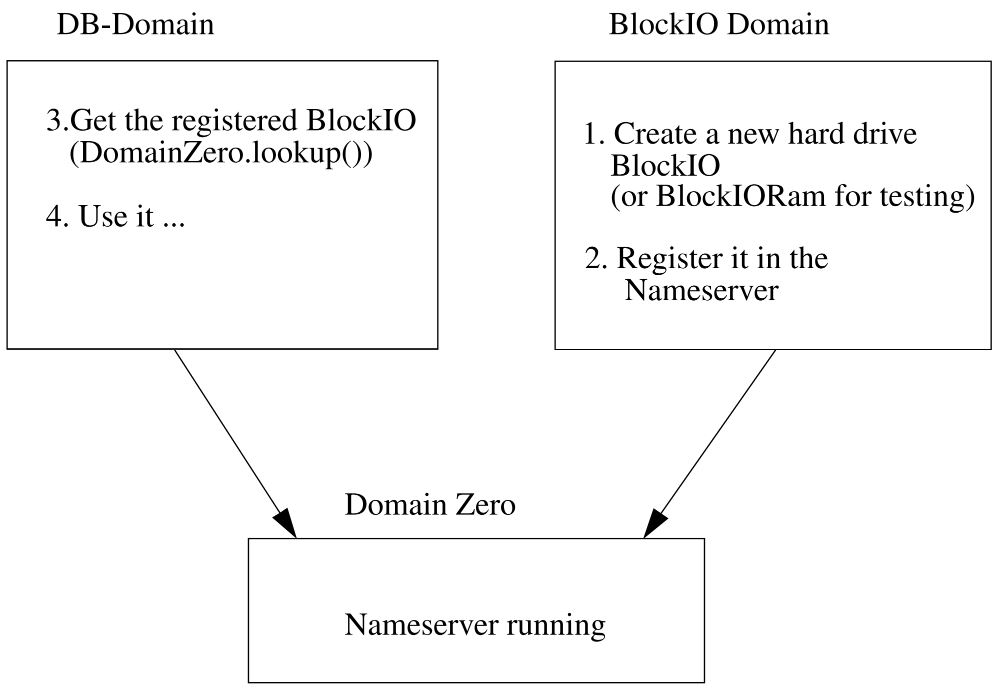


Abbildung 3.1 BlockIO Inizialisierung

(Für Domains und Memory Objekte siehe JX Dokumentation)

Kapitel 4

Systempuffer- und Seitenverwaltung

4.1 Aufgaben des Systempuffers und Seitenverwaltung

Eine nachteilige Eigenschaft der Plattenspeicher sind die sehr langen Lese- und Schreibzugriffszeiten. Der Zugriff auf die Platte kann um Faktor 1000 langsamer sein als der Zugriff auf den Hauptspeicher. Andererseits ist der Hauptspeicher in der Regel teurer als Festplattenspeicher. Deswegen ist meistens nicht genug Hauptspeicher im System um die Bedürfnisse der Datenbankapplikation zu befriedigen.(Allerdings ändert sich das schnell, heutzutage kann man ruhig für manche Anwendungen einige Gigabytes von billigem Hauptspeicher installieren). Die effiziente Ausnutzung der beiden Speicherarten ist die Aufgabe des Systempuffers.

Allgemein funktioniert diese Schicht so, daß die Daten, die gerade gebraucht werden, in den Hauptspeicher gepuffert werden. Die Daten, die momentan nicht gebraucht werden, werden auf die Festplatte ausgelagert. Ziel ist es die Anzahl der Festplattenzugriffe zu minimieren, also möglichst viele Datenbankoperationen hauptsächlich im Hauptspeicher zu erledigen.

Es ist nicht immer effizient, einzelne Blocks zwischen dem Hauptspeicher und der Festplatte zu transferieren. Deswegen werden mehrere Blöcke zu einer Seite zusammengefaßt. Irgendwo muß die Information verwaltet werden, welche Seite momentan besetzt ist und welche frei. Die Freispeicherverwaltung und die Zuordnung von Blöcken zu Seiten ist die Aufgabe der Seitenverwaltung.

4.2 Konzepte und Techniken des Systempuffer- und Seitenverwaltung.

4.2.1 Seiten, Frames und Blöcke

Wie schon erwähnt besteht eine Seite aus einem oder mehreren Blöcken. Die Anzahl der Blöcke ist gleich für alle Seiten. Bei direkter Blockzuordnung sind die zugeordneten Blöcke immer benachbart, die n-te Seite enthält im allgemeinen die Blöcke von n * k bis (n + 1) * k, wo k die Anzahl der Blöcke pro Seite ist. Bei indirekter Blockzuordnung brauchen die Blöcke einer Seite nicht unbedingt benachbart sein.Sie ist flexibler als die Direkte und ermöglicht die Implementierung von Konzepten wie Schattenspeicher, ist aber mit Zusatzaufwand verbunden und daher nicht so performant.

Wenn eine Seite benutzt wird, wird ihr Platz im Hauptspeicher zugeordnet. Dieser Platz wird als Frame bezeichnet. Ein Frame kann im Laufe der Zeit verschiedenen Seiten zugeordnet werden. Eine Seite besitzt ein Frame nur während sie im Hauptspeicher ist.

4.2.2 Zustände der Seiten

Eine Seite kann sich in den folgenden zuständen befinden:

paged out **–** die Seite befindet sich nur auf der Festplatte
paged in - die Seite befindet sich im Hauptspeicher und auf der Festplatte, die beiden Kopien sind konsistent
dirty – die Seite befindet sich im Hauptspeicher und auf der Festplatte, die Kopie im Hauptspeicher ist aktueller als die auf der Platte
fixed – die Seite befindet sich im Hauptspeicher und wird gerade gelesen oder modifiziert.

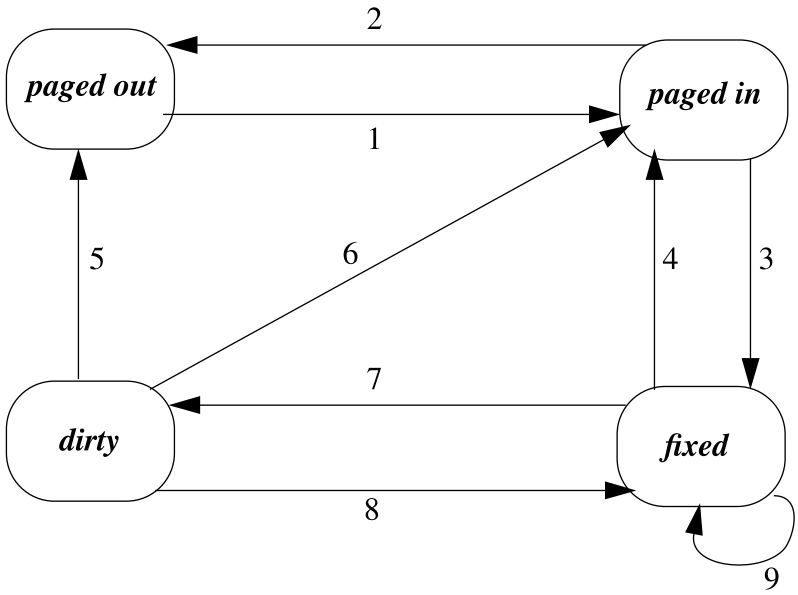


Abbildung 4.1 Seitenzustände und Zustandsübergänge

Das obige Diagramm veranschaulicht die Zustandsübergänge, die im System stattfinden können. Am Anfang (Start des Systems) sind alle Seiten in Zustand **paged out**. Wenn ein Benutzerthread die Daten einer Seite benötigt (egal ob zum Lesen oder zum Schreiben), wird dieser Seite ein Frame zugeordnet und die Daten der Seite werden in dieses Frame gelesen. Die Seite geht in Zustand **paged in**. (1) über.

Wenn sich eine seite in Zustand **paged in** befindet, kann die Seite wieder aus dem Hauptspeicher verdrängt werden und in den Zustand **paged out** gelangen**.** (2).Eine solche Situation entsteht, wenn der ganze dem Systempuffer zugeordnete Hauptspeicher voll ist und gerade eine Seite angefordert wird, die nicht im Hauptspeicher liegt. Dann entscheidet eine Seitenersetzungstrategie (siehe 4.2.3) welche von den gepufferten Seiten auszulagern ist, damit Platz für die neue Seite frei wird.

Wenn eine Seite im Zustand **paged in** ist, kann ein Benutzerthread eine Referenz ihrer Daten vom Systempuffer beantragen. Die Seite übergeht in Zustand **fixed.** (3) Fixed bedeutet, daß möglicherweise gerade ein Zugriff (Schreiben oder Lesen) auf die Daten der Seite stattfindet. Die Seite darf also in diesem Zustand nicht verdrängt werden, ein Zustandsübergang von **fixed** nach **paged out** ist nicht möglich.

Eine Seite kann mehrmals gefixed werden (9), der Systempufer verwaltet einen Referenzzähler für jede gepufferte Seite. Wenn ein Thread die fix Operation(siehe PageBuffer unten) aufruft, wird der Zähler inkrementiert. Die unfix Operation dekrementiert ihn entsprechend. Eine Seite ist in Zustand **fixed** wenn der Zähler größer Null ist und in **paged in** sonst (4).

Wenn ein Thread eine gefixte Seite modifiziert, muß er es dem Systempuffer entsprechend mitteilen, um sicherzustellen, daß die Seite bei der nächsten Verdrängung auf der Platte geschrieben wird und nicht einfach verworfen.(siehe PageBuffer). Von dem Zustand **fixed** kann eine Seite also entweder in **paged in** gelangen, falls es keine Änderungen gab, oder in **dirty** sonst. Der Zustand **dirty** kann nur über **fixed** erreicht werden (7). Eine modifizierte Seite kann selbstverständlich wieder gefixed werden (8).

Seiten im Zustand **dirty** können in den Zustand **paged in** (6) übergehen, wenn der Benutzerthread eine vorzeitige Sicherung des Seiteninhalts auf der Platte (flush) beantragt.

Diese Funktionalität ist für die Realisierung von Transaktionen und Checkpoints wichtig.

Beim Übergang von **dirty** in **paged out** (5) wird der Inhalt der Seite auf der Platte gesichert und den Platz im Hauptspeicher aufgeräumt.

4.2.3 Seitenersetzungsstrategien

Im Schritt 2 des obigen Diagramms muß eine von den im Hauptspeicher gepufferten Seiten zum Verdrängen ausgewählt werden. Für diese Entscheidung ist die Seitenersetzungsstrategie zuständig. Im wesentlichen versucht eine solche Strategie die Anzahl der Plattenzugriffe zu minimieren, weil die Hauptspeicherzugriffe sehr viel schneller sind. Fast alle effiziente Strategien versuchen vorauszusagen welche Seiten früher als andere gebraucht werden, um solche seiten im Hauptspeicher zu behalten. Eine bekannte und in JXDB realisierte Strategie ist die LRU (Least Recently Used). Die klassische Variante ist allerdings in JXDB ein wenig erweitert.

Alle gepufferten Seiten werden von der LRU Strategie in eine geordnete Liste verwaltet. Das Sortierkriterium ist der letzte Zugriff der Seite. Bei unfix einer Seite wird die Seite von der Liste herausgenommen und am Anfang der Liste hinzugefügt. Seiten zum auslagern werden vom Ende der Liste ausgewählt. Die Idee die dahiter steckt ist einfach: eine Seite, die lange Zeit nicht angesprochen wurde wird wahrscheinlich auch in Zukunft nicht mehr gebraucht, deswegen ist es sinnvoll solche Seiten vom Hauptspeicher zu verdrängen. Die kommen im Laufe der Zeit zum Ende der Liste und “fallen raus”.

LRU ist einfach zu realisieren und gibt durchschnittlich gute Ergebnisse, obwohl in manchen Fällen passieren kann, daß die Annahme (daß eine lange nicht gebrauchte Seite in Zukunft auch nicht gebraucht wird) nicht stimmt. Vorteilhaft ist, daß wichtige Seiten, die z.B. Metainformationen enthalten und auf denen ständig zugegriffen wird, im Puffer bleiben und den schnellen Zugriff beibehalten.

Eine kleine Erweiterung der Annahme ist jedoch möglich. Nicht modifizierten Seiten (die nicht in dem **dirty** Zustand sind) brauchen auch nicht auf die Platte geschrieben werden, weil der Platteninhalt aktuell ist. Die können einfach verworfen werden, ihre Verdrängung ist also viel billiger als die Verdrängung von modifizierten Seiten. Es ist deswegen sinnvoll, nicht modifizierte Seiten in der Auswahl vorzuziehen. Allerdings widerspricht das der anfänglichen Annahme. Es kann z.B. eine modifizierte Seite eine frühere letzte Zugriffszeit haben als eine nicht modifizierte. Die Kompromißlösung sieht so aus, daß von den letzten k Seiten in der Liste versucht wird, eine nicht modifizierte zu finden. Falls es keine nicht Modifizierte gibt, wird die letzte Modifizierte ausgewählt. Die Zahl k ist einstellbar.

4.2.4 Strategien zur Optimierung der Schreibzugriffe auf der Platte

Die Kommunikation zwischen dem Systempuffer und der Festplatte, bzw. Der Speicherverwaltung, besteht aus Lese- und Schreibaufträge. Eine mögliche Konfiguration könnte so aussehen:

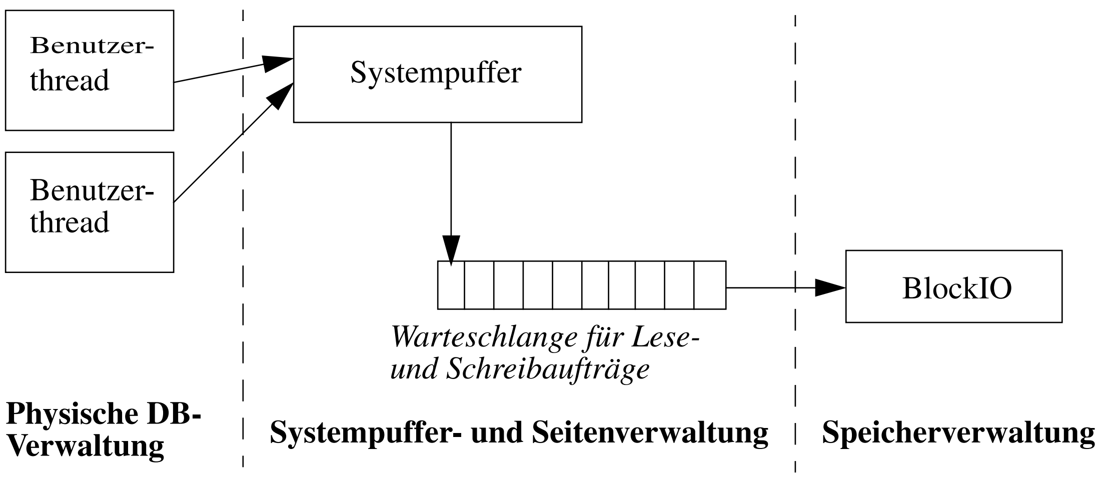


Abbildung 4.2 Grobstruktur des Systempuffers

Es gibt also mehrere Benutzerthreads, die gleichzeitig mit dem Systempuffer kommunizieren. Das führt zu Seitenverdrängungen, die als Schreib- und Leseaufträge an die Speicherverwaltung geschickt werden. Die Speicherverwaltung kann nur einen Auftrag gleichzeitig abarbeiten und die Aufträge können viel schneller produziert als abgearbeitet werden. Deswegen müssen sie in eine Warteschlange abwarten, bis sie an der Reihe kommen. Diese Warteschlange bietet einige Optimierungsmöglichkeiten.

Erstens können die Aufträge nach gewissen Kriterien sortiert werden, um z.B. sequenziellen Plattenzugriff für alte Platten zu realisieren. (dies wäre eigentlich ein Teil der Funktionalität der Speicherverwaltung, könnte aber auch hier flexibler realisiert werden).

Zweitens ist es offensichtlich, daß ein Leseauftrag für eine Seite einen vorherigen Lese- oder Schreibauftrag für dieselbe Seite überflüssig macht. So braucht eine Seite, die von mehreren Threads gleichzeitig angefordert wird nur einmal gelesen zu werden. Schreibaufträge, die in der Warteschlange stehen, können durch Leseaufträge für dieselbe Seite einfach gelöscht werden. Die Daten der Seite kommen in den Hauptspeicher zurück, die Seite geht in das Zustand **dirty** über. Anders gesagt darf pro Seite maximal ein einziger Lese- oder Schreibauftrag in der Warteschlange stehen.

4.2.4 Freispeicherverwaltung

Die Freispeicherverwaltung ist Teil der Seitenverwaltung. Den Benutzerthreads soll die Möglichkeit zur Verfügung stehen, neue, unbenutzte Seiten dynamisch anzufordern und später freizugeben. Um zu wissen, welche Seiten frei sind, muß zusätzliche Information verwaltet werden. Eine speichereffiziente Lösung ist es für jede Seite einen Bit zu setzen, wenn sie allokiert wird. Wenn die Seiten von Null bis n durchnumeriert sind, wird der nullte Bit der nullten Seite zu der ersten Seite zugeordnet, der erste Bit zur zweiten Seite usw. Falls die Bits der nullten Seite nicht ausreichen, könnten die bits der k + 1 - te Seite genommen werden (k sei die Anzahl von Bits pro Seite) usw. (Abbildung 4.3)

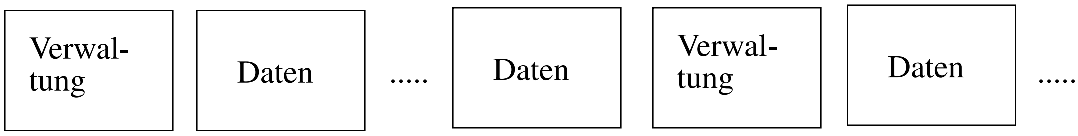


Abbildung 4.3. Struktur der Freispeicherverwaltungsinformationen auf der Platte

Eine wichtige Eigenschaft der Daten ist das Clustering. Das bedeutet, daß ähnliche Daten (oder Daten, die normalerweise zusammengehören und auf denen oft gleichzeitig zugegriffen wird), auch physisch benachbart auf der Platte abgespeichert werden. So kann man Plattenkopfbewegungen sparen. Die obige Lösung bietet eine einfache Strategie an, die zu dem Clustering beiträgt. Neue freie Seiten werden immer in der Nähe der zuletzt allokierten Seite gesucht. Angenommen alle zusammengehörende Seiten wurden zur selben Zeit allokiert, dann sind sie auch physisch benachbart. Daß muß nicht immer der Fall sein, ist aber oft so.

Am Anfang ist es Aufgabe der Freispeicherverwaltung das Speichermedium zu *formatieren*, d.h. alle Seiten als frei zu markieren.

4.2.5 Koordinierung

Ein Koordinierungsproblem entsteht zwischen dem Benutzerthread und der Speicherverwaltung. (die Koordinierungsprobleme zwischen mehrerer Benutzerthreads werden weiter unten diskutiert). Ein Thread, der eine Seite fixen will, generiert eine Leseanforderung, falls die Seite sich nicht im Hauptspeicher befindet. Er bekommt immer sofort eine Referenz auf die Seite (eigentlich eine Javareferenz auf dem Frame im Systempuffer, wo die Daten der Seite sich befinden sollen), in diesem konkreten Fall müssen die Daten der Seite aber gelesen werden und sind erst später gültig. Der Thread darf also die Referenz nicht benutzen, bis er vom Systempuffer benachrichtigt wird, daß die Daten angekommen sind. Die Zeit, die bis zur Gültigkeit der Referenz vergeht kann auch das Ausschreiben einer anderen Seite enthalten, um freien Speicherplatz für die gewünschte Seite zu verschaffen. Ein möglchies Szenario könnte so aussehen:

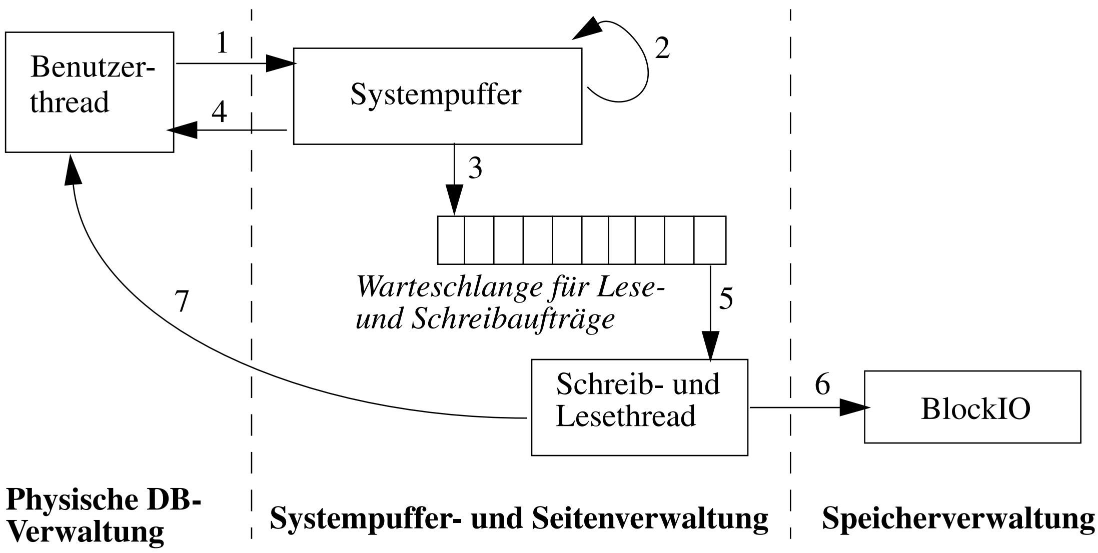


Abbildung 4.4. Ein mögliches Szenario für die fix-Operation

1. Der Benutzerthread ruft die fix Operation für eine Seite im Systempuffer auf.
2. Der Systempuffer sucht ein freies Frame, wo die Daten der Seite später untergebracht werden
3. Der Systembuffer erzeugt einen Leseauftrag für die Seite und fügt es am Anfang der Warteschlange ein.
4. Als Ergebnis der fix Operation liefert der Systempuffer die Referenz des Ausgewählten Frames zurück. Der Benutzerthread stellt jedoch fest, daß die Daten im Frame ungültig sind und blockiert sich(wartet auf Schritt 7).
5. Irgendwann später kommt der Leseauftrag am Ende der Warteschlange und wird zur Verarbeitung entnommen. Dies wird vom Schreib- und Lesethread getan, ein Thread, der nur dafür zuständig ist, Aufträge an die Speicherverwaltung und Ergebnisse nach oben weiterzuleiten. Falls es keine Aufträge gibt, blockiert er sich.
6. Der Schreib- und Lesethread ruft die readSectors Operation der Speicherverwaltung und empfängt die gelesenen Bytes und plaziert sie in das ausgewählte Frame.
7. Er schickt eine Nachricht direkt an dem entsprechenden Benutzerthread, daß das Frame gültig geworden ist. Der Benutzerthread kann jetzt mit der Seite weiterarbeiten.

Der Benutzerthread sollte also die Möglichkeit haben, sich zu blockieren, bis die Daten ankommen. Es ist auch sinnvoll, daß er mehrere Seiten gleichzeitig anfordern kann und sich erst nachher blockiert. Weil je mehr Aufträge in der Warteschlange stehen, desto bessere Optimierungschancen gibt es. (bessere Sortierung ist möglich und außerdem viele Aufträge können sich gegenseitig löschen). Deswegen muß der Thread für jede angeforderte Seite ein Koordinierungsobjekt bekommen, auf dem er sich blockieren kann (siehe db.com.EventSync). Diesem Objekt schickt später der Schreib- und Lesethread die Nachricht, daß die Daten angekommen sind.

4.3 Implementierung

Die Systempuffer- und Seitenverwaltungsschicht der JXDB enthält eine Menge von Interfaces, die ihr Gerüst bilden. Für jedes der Interfaces gibt es eine konkrete Implementierungsklasse, die bei Bedarf ausgewechselt werden kann.

4.3.1 Überblick der Interfaces

Die Interfaces, die die Schnittstelle für die oberen Schichten bilden, sind db.systembuffer.PageBuffer und db.systembuffer.PageManager. Die Schicht besteht jedoch aus mehreren internen Modulen, die auch java Interfaces besitzen, obwohl sie nicht nach außen zu verwenden sind. Diese Interfaces sind db.systembuffer.sortingbuf.SortingBuffer, db.systembuffer.strategy.PageStrategy und db.systembuffer.strategy.PageStrategyRecord.

4.3.2 Die Klasse FirstStageBuf

Das ist die Klasse, welche die Funktionen des Systembuffers implementiert. Sie ist von db.systembuffer.PageBuffer abgeleitet, liefert die fix und unfix Operationen, steuert die Warteschlange der Lese- und Schreibaufträge (SortingBuf) und benutzt die Seitenersetzungsstrategie (PageStrategy) um Seiten zum Verdrängen auszuwählen. In dieser Klasse ist einzustellen, wieviel Hauptspeicher maximal für das ganze Datenbanksystem zu reservieren ist. Erst wenn die eingestellte Grenze erreicht wird, beginnt FirstStageBuf

Seiten zu verdrängen. Die Klasse, zusammen mit db.systembuffer.sortingbuf.SortingBufferImpl, ist speziell für Plattenspeicher ausgelegt. Wenn anderes Speichermedium benutzt wird, etwa großes Hauptspeicher, sollte man auch eine andere PageBuffer Implementierung nehmen, in diesem Fall db.systembuffer.memorybuf.MemoryBuffer.

Momentan gibt es systemweit nur ein Objekt dieser Klasse. Es sind aber auch andere konfigurationen denkbar, besonders wenn mehrere Speichermediumtypen bentzt werden, oder verschiedene Seitenersetzungsstrategien für verschiedene Partitionen der Platte eingesetzt werden sollen.

4.3.3 Die Klasse FirstStagePageMgr

Diese Klasse implementiert die Seitenverwaltung in Zusammenhang mit db.systembuffer.firststage.FirstStageBuf. Sie ist von db.systembuffer.PageManager abgeleitet und bietet den oberen Schichten operationen zum Allokieren und Freigeben von Seiten an. Die Benutzerthreads können auch überprüfen, ob eine Seite frei ist. Es ist möglich das ganze Speichermedium zu formatieren (alle Seiten als frei zu markieren).

Obwohl die Klasse eigentlich nicht nur mit FirstStageBuf verwendbar ist, empfiehlt sich für jede PageBuffer Implementierung auch eine entsprechende PageManager Implementierung zu schreiben.

4.3.4 Die Klasse SortingBufferImpl

Diese Klasse implementiert die Warteschlange für Lese- und Schreibaufträge. Sie verwaltet auch den Lese- und Schreibthread. Ihre Basisinterface ist db.systembuffer.sortingbuf.SortingBuffer. Sie bietet dem FirstStageBuf Operationen zum hinzufügen von Aufträge am Anfang der Warteschlange. Die Aufträge werden momentan einfach nach aufsteigender Seitennummer sortiert. Um optimale Performanz zu erreichen, muß das Sortierkriterium mit der Plattengeometrie abgestimmt werden.

4.3.5 Die Klasse LruPageStrategyRecord

Diese Klasse enthält alle wichtige Verwaltungsinformationen über jede Seite, die im Hauptspeicher gepuffert ist. Ein Teil dieser Information ist spezifisch für die LRUPageStrategy, anderer Teil ist für alle von PageStrategyRecord abgeleitete Klassen gleich und wird von FirstStageBuf benötigt.

4.3.5 Die Klasse LRUPageStrategy

Diese Klasse realisiert die Seitenersetzungsstrategie. Sie ist von db.systembuffer.strategy.PageStrategy abgeleitet und funktioniert wie die schon oben vorgestellte LRU Strategie.

4.3.6 Die Klasse MemoryBuffer

Diese Klasse implementiert die PageBuffer Interface für Hauptspeicher als Speichermedium. Die Klasse ist sehr simpel, weil keine besondere Optimierungsmaßnahmen getroffen werden müssen. Es gibt keine Seitenersetzungsstrategie und keine Warteschlange für Lese- und Schreibaufträge. Sie dient hauptsächlich zu Testzwecken, kann aber auch für kleine Datenmengen benutzt werden und ist performanter als FirstStageBuf (nur die Logik der Klasse wird bei der Performanzaussage in Betracht genommen, nicht das Speichermedium, das sowieso schneller ist).

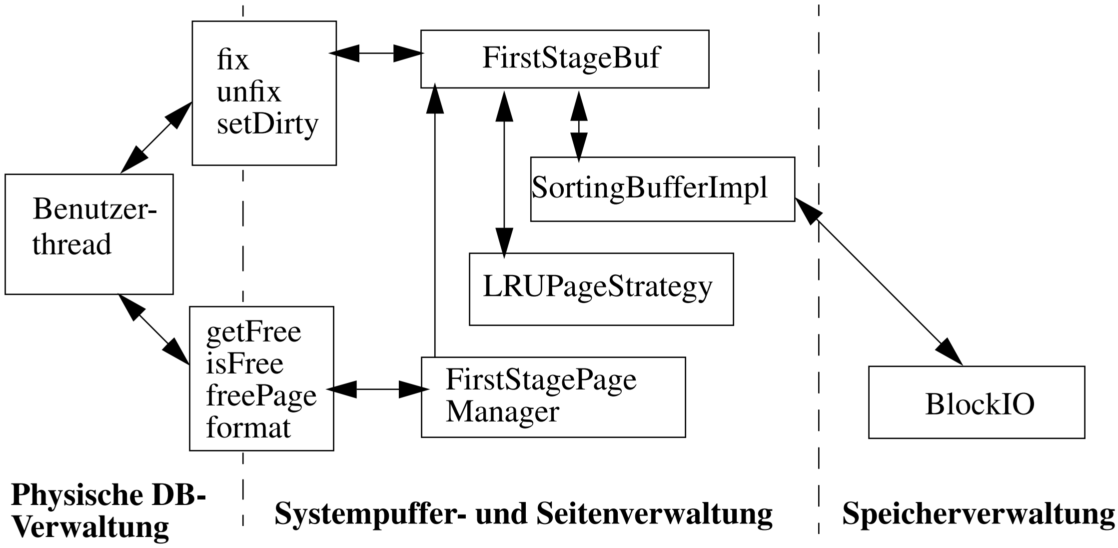


Abbildung 4.5 Systempufferkonfiguration mit Festplatte als Speichermedium

Die Abbildung 4.5 oben zeigt die Struktur einer Festplattenkonfiguration. Weiter unten ist eine Konfiguration mit Hauptspeicher als Speichermedium. Momentan unterstützt die Klasse MemoryBuffer kein Multithreading, kann aber ohne großen Umstand erweitert werden, falls das gebraucht wird. In der Abbildung 4.6 unten sieht man kein BlockIO Interface, weil auch die ganze Speicherverwaltung überflüssig wird. Das transparente Umschalten zwischen Platten BlockIO und BlockIORam ist also nicht sehr effizient, weil der Systempuffer unnötigt benutzt wird.

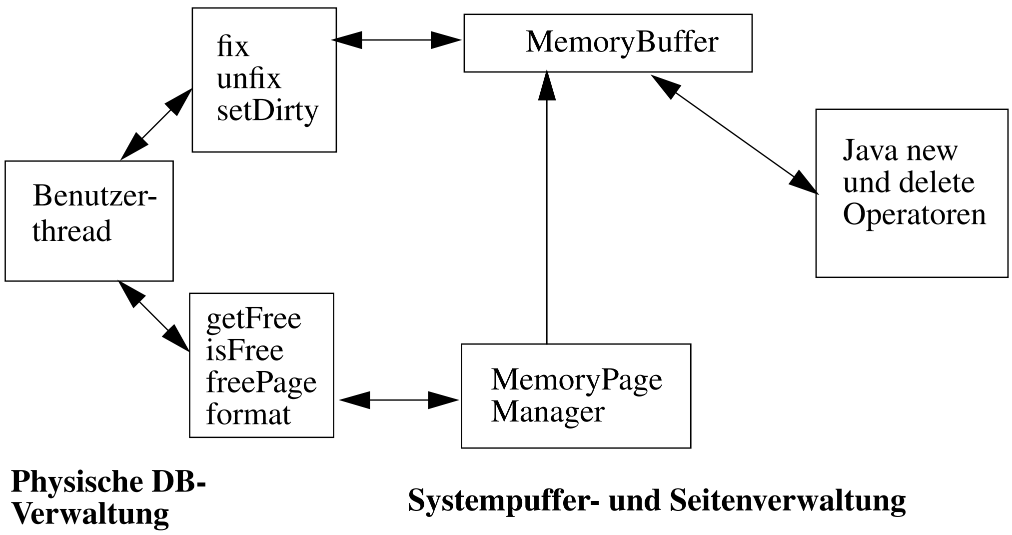


Abbildung 4.6 Systempufferkonfiguration mit Hauptspeicher als Speichermedium

4.3.7 Hilfsklassen und Zusatzklassen

Weitere Klassen, die von der Systempufferschicht benutzt werden sind db.com.Event-

Sync und db.treemap.TreeMap. EventSync ist die Klasse, mit derer Hilfe ein Benutzerthread feststellen kann ob die Daten einer Leseanforderung schon angekommen sind. Sie bietet die Funktion waitForEvent an, die den aufrufenden Thread blockiert, bis ein Event ankommt (d.h. die Funktion setEvent von einem anderen Thread aufgerufen wird). Da Blockieren von Threads in der Regel nicht performant ist, gibt es die Funktion getNeedToWait, die nur überprüft, ob Blockieren notwendig ist. Z.B. wenn eine Seite im Hauptspeicher schon liegt, ist keine Blockierung notwendig, der Benutzerthread kann mit der Seitenreferenz von der fix Operation sofort anfangen zu arbeiten.

Ein EventSync Objekt wird bei jedem Aufruf von der Fix Operation von dem Benutzerthread dem FirstStageBuf übergeben (Abb. 4.4, Schritt 1). Wenn die gesuchte Seite im Hauptspeicher liegt, schickt der FirstStageBuf dem Objekt ein Event noch bevor er die Seitenreferenz zurückgegeben hat(Abb. 4.4, Schritt 2). Die getNeedToWait Methode würde so im Benutzerthread false zurückliefern und der Thread würde sich nicht blockieren. Falls die Seite aber von der Platte gelesen werden soll, übergibt der FirstStageBuf das EventSync Objekt dem SortingBufferImpl (Abb. 4.4, Schritt 3) und liefert eine ungültige Seitenreferenz zurück (Abb. 4.4, Schritt 4). Der Benutzerthread blockiert sich auf dem EventSync Objekt und wartet, bis der SortingBufferImpl ihm ein Event schickt (Abb. 4.4, Schritt 7), nachdem die Referenz gültig wird.

Die TreeMap Klasse realisiert eine sortierte liste. Sie entspricht im Wesentlichen der JDK TreeMap Klasse und wurde nur deswegen implementiert, weil die JDK Klasse noch nicht in JX integriert war.

4.4 Zusammenfassung

Die zweite Schicht von unten nach oben von der 5-Schichtenarchitektur wurde vorgestellt. Zuerst wurden die Konzepte und Techniken der Implementierung erläutert, danach wurden die einzelnen Klassen genannt und ihre Verwendung durchdiskutiert. Ausführliche Klassendokumentation findet man in der Dokumentation im Sourcecode(javadoc)

Kapitel 5

Physische DB-Verwaltung

5.1 Aufgaben der Physischen DB-Verwaltung

Die Schichten der Speicherverwaltung und der Systempuffer- und Seitenverwaltung bilden zusammen eine Abstraktion, die es erlaubt die ganze Datenmenge der Datenbank in durchnummerierte Seiten fester größe zu repräsentieren. Die meisten Datenstücke in der Datenbank sind aber kleiner als einer Seite und schon gar nicht fester Größe. Die Aufgabe der physischen DB-Verwaltung besteht darin, den Benutzerthreads Funktionen zum Allokieren, Freigeben und Modifizieren von beliebig langen Byteketten anzubieten. Diese Byteketten werden als physische Sätze bezeichnet. Die sollen durch eine Adresse identifizierbar sein und sollen außerdem die Eigenschaft besitzen, daß diese Adresse im Laufe der Zeit konstant bleibt.

Diese Schicht benutzt nur die Schnittstelle der Systempuffer- und Seitenverwaltung, d.h. die Sätze werden in Seiten abgespeichert. Es sollte deswegen auch Möglichkeiten geben, größere Sätze als eine Seite abzuspeichern.

5.2 Konzepte und Techniken

5.2.1 Sätze

Wie schon erwähnt sind Sätze beliebig lange (aber kleiner als eine Seite) Byteketten. Sie haben eine eindeutige Adresse, die meistens aus Seitennummer und Offset in der Seite besteht. Unter Offset wird nicht die physische Position in der Seite verstanden, da sie sich ja ändern kann, wenn die Sätze verschoben werden, sondern eine logische Position, die konstant bleibt. Zusätzlich müssen auch Maßnahmen getroffen werden, falls ein Satz aus seiner Seite verschoben wird, daß seine Adresse sich trotzdem nicht ändert. Das wird durch Indirektionstechniken erreicht, die weiter unten erläutert sind.

5.2.2 Satztypen

Es ist manchmal sinnvoll, Sätze nach bestimmten Kriterien zu klassifizieren. Das wird wahrscheinlich im nächsten Kapitel ganz klar werden, hier findet man aber ein Überblick der wichtigen Gedanken, die zu dieser Idee führen. Wichtig ist, daß Sätze verschiedener Typen immer in verschiedenen Seiten abspeichert werden.

Erstens kann man alle Sätze eines Typs ganz einfach und effizient löschen, in dem man die seiten des Satztyps freigibt. Z.B. wenn alle Sätze, die mit einer Tabelle (siehe nächstes Kapitel) verbunden sind (Datensätze, Indexsätze, etc.), kann man die ganze Tabelle auf einen Schlag löschen.

Zweitens kann man so sicher sein, daß zwei verschiedene Threads, die jeweils mit Sätzen verschiedener Datentypen arbeiten, sich nie in die Quere kommen. Das ist also eine einfache Möglichkeit ganze Tabellen mit Schreib oder Lesesperren zu versehen.

Zuletzt sind Sätze eines Typs meistens gleichgroß, deswegen kann die Verwaltung effizienter gestaltet werden, wenn es viele Änderungsoperationen in der Datenbank gibt.

5.2.3 TID (Tuple Identifier) Konzept

Das in dem JXDB implementierte TID Konzept (Siehe [HR99] und [GUW00] für Details) ist eine effiziente Technik, den freien Speicherplatz innerhalb einer Seite zu verwalten. Es gibt auch andere Möglichkeiten, wie translation tables, etc., die hier aber nicht diskutiert werden um diese Dokumentation nicht unnötig zu überlasten.

Beim TID Konzept besteht eine Satzadresse aus eine Seitennummer und eine logische Nummer des Satzes innerhalb der Seite. Eine 4k große Seite sieht ungefähr so aus:

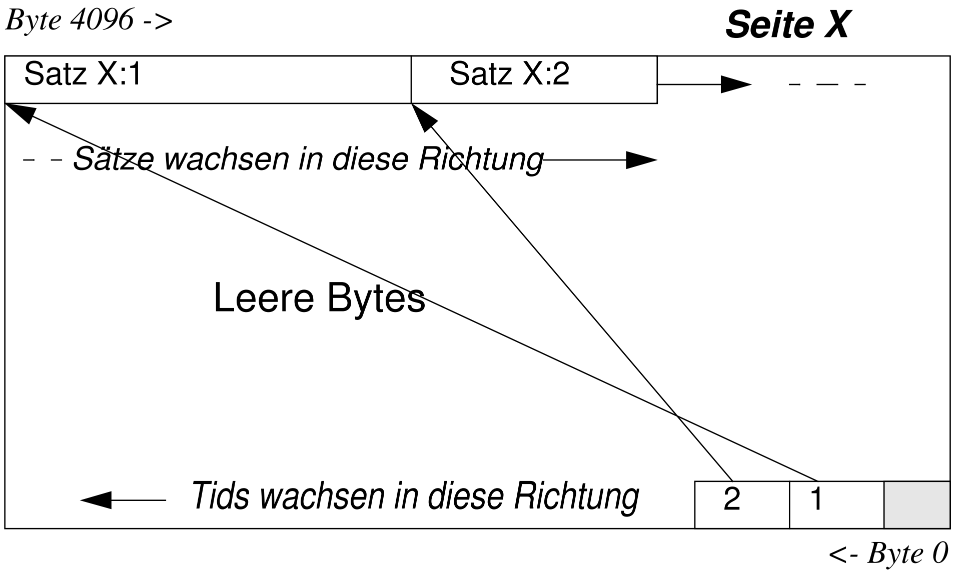


Abbildung 5.1. Aufbau einer Seite im TID Konzept

Die von 1 bis 2 durchnummerierten Kästchen in der unteren rechten Ecke der Seite sind je 2 Byte große integer Zahlen und enthalten die physischen Positionen der Sätze innerhalb der Seite. Ihre Nummern (1 – 2) sind die Logische Satznummern. Das erste leere Kästchen enthält die Anzahl der verbleibenden freien Bytes. Neue Sätze werden vom Byte 4096 aus in Richtung Byte 0 angelegt, die Satznummer dagegen werden in Richtung Byte 4096 angelegt.

Wenn ein Satz gelöscht wird, werden alle Sätze, die nach ihm in der Seite liegen, verschoben, um die Lücke zu schließen. Die Satznummern werden entsprechend aktualisiert, die Satznummer des gelöschten Satzes wird als ungültig markiert. Diese Satznummer darf aus Konsistenzgründen nie mehr verwendet werden. Das bedeutet, daß bei sehr häufigen Löschoperationen viele Bytes als ungültig markiert und daher unbrauchbar werden. Deswegen müssen die Satztypen evtl. periodisch aufgeräumt werden.

Wenn die Länge eines Satzes geändert wird und die neue Länge in die Seite paßt, werden wieder alle dahinterliegende Sätze verschoben, und alle Satznummer entsprechend modifiziert. Es werden keine Nummer als ungültig markiert.

Falls aber der modifizierte Satz nicht mehr in der Seite hineinpaßt, entsteht ein Problem, weil der Satz in eine andere Seite verschoben werden muß. D.h. Zugriffe auf die alte Adresse müssen entsprechend auf die Neue umgeleitet werden. (wie übrigens bei der Post der fall ist). Der vorhandene Satz wird verkürzt und als Indirektionssatz markiert, der neue Inhalt des Satzes wird in eine andere Seite abgespeichert und die neue Adresse wird in den Indirektionssatz gespeichert:

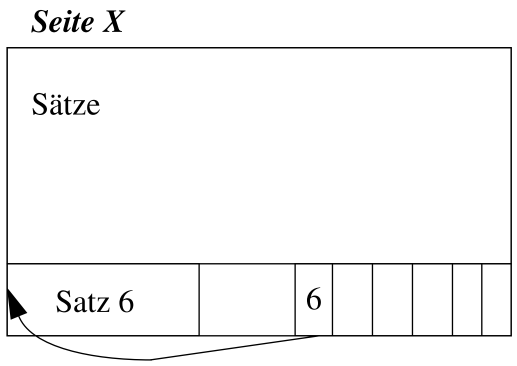


Abbildung 5.2 Redirektionsstufen. a) Satz 6 muß auf das Doppelte vergrößert werden

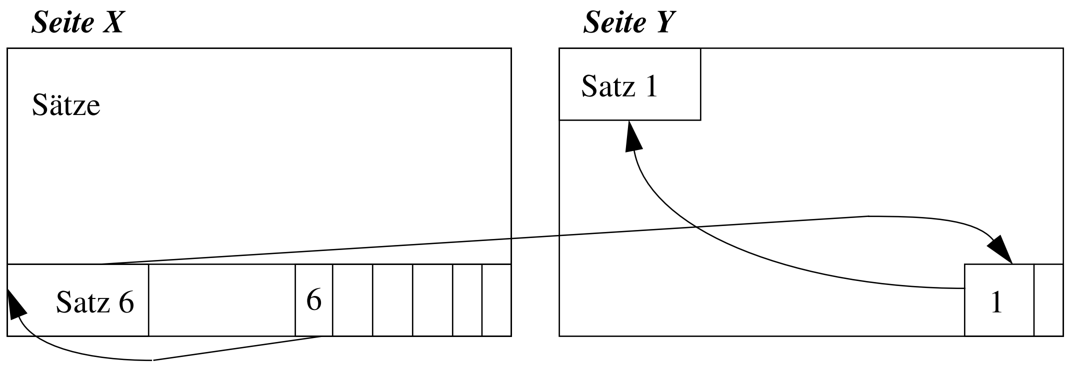


Abbildung 5.2 b) Satz 6 wird zu einem Indirektionssatz verkürzt, die Daten kommen in den Satz 1 in Seite Y

Wichtig ist, daß keine mehrstufige Indirektionen entstehen sollen. Wenn ein Satz wieder migrieren muß, wird keine neue Indirektion erstellt, sondern die alte modifiziert.

Der größte Vorteil des TID Konzepts ist, daß der lesende Zugriff auf einen Satz in der Regel 1 und höchstens 2 Seitenzugriffe benötigt (nur wenn eine Indirektion vorhanden ist). Das bedeutet selbstverständlich weniger Plattenzugriffe als bei anderen Strategien. Durch geschickte Optimiertechniken kann die Anzahl der Indirektionen klein gehalten werden. Z.B. können kleiner gewordene Sätze zurück in ihre ursprüngliche Seite verschoben werden.

Da Lesen normalerweise die häufigste Operation in einer Datenbank ist, ist in vielen Fällen das TID Konzept vor anderen vorzuziehen.

5.2.4 Freispeicherverwaltung

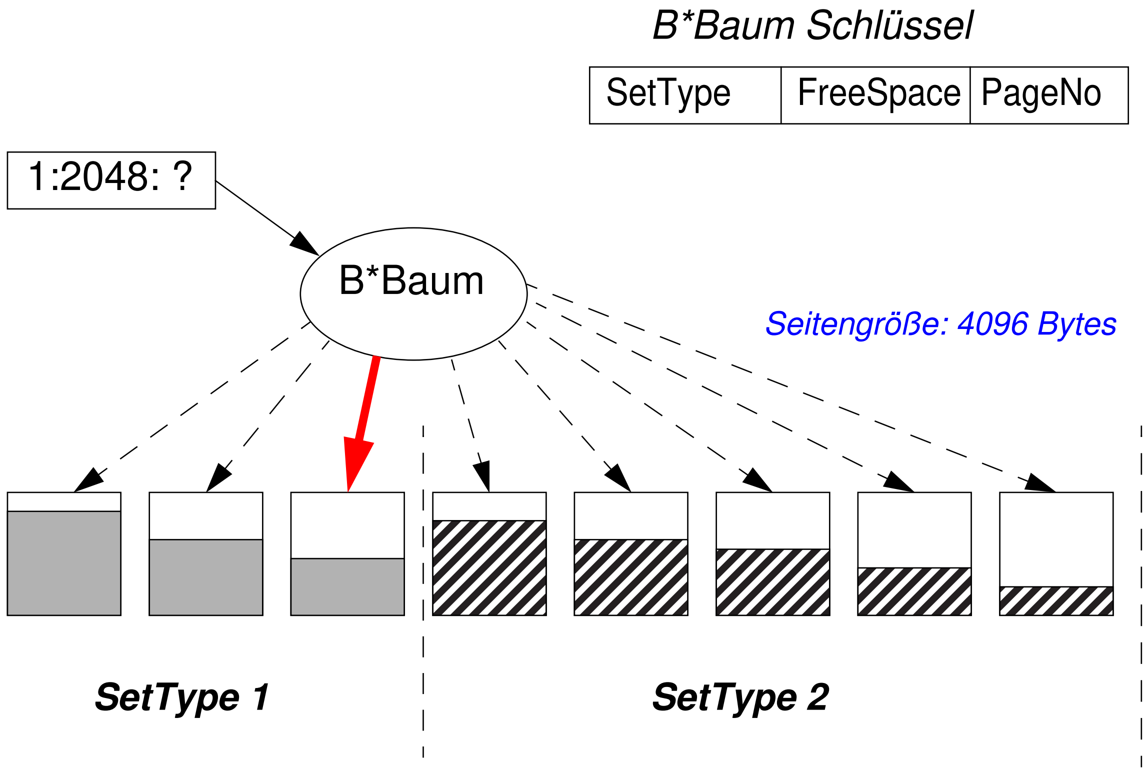


Abbildung 5.3 Freispeicherverwaltung der physischen DB-Verwaltung

Wenn ein neuer Satz angelegt werden muß, muß zuerst eine Seite ausgewählt werden, die genügend Speicherplatz enthält um den Satz unterzubringen. Das bedeutet, daß für alle Seiten, die zum Abspeichern von Sätzen benutzt werden, Freispeicherinformation verwaltet werden muß. Es können außerdem verschiedene Strategien zur Auswahl einer Seite eingesetzt werden, z.B *best fit* oder *first fit*, etc., die entsprechend auf irgendeine Weise auf diese Freispeicherinformation effizient zugreifen sollen. Z.B ist es für die first fit Strategie wichtig, schnell eine beliebige Seite zu finden, die den Satz unterbringen kann. Für best fit dagegen muß die Seite gefunden werden, in welcher der Satz am besten paßt (am wenigsten Verschnitt hinterläßt).

Die Freispeicherinformation wird in JXDB in einem B* Baum verwaltet (Abbildung 5.3). Die Methoden des B* Baums sind ausführlich in der Dokumentation im Sourcecode beschrieben(javadoc), die B* Baum Funktionsweise in Anhang A. Er dient zum persistenten Abspeichern von Datenpaaren < Schlüssel, Daten >, die nach dem Schlüssel sortiert werden. Der Schlüssel der Freispeicherverwaltung besteht aus Satztypnummer, Anzahl der freien Bytes in der Seite und Seitennummer in dieser Reihenfolge. Es werden dummy Daten im B* Baum gespeichert, die Schlüssel enthalten schon alle notwendige Informationen. Die in JXDB implementierte best fit Strategie sucht den ersten Schlüssel im B* Baum, dessen Satztypnummer dem des neuen Satzes entspricht und dessen Anzahl der freien Bytes größer ist als die Länge des Satzes. Da die Sätze sortiert sind ist das die kleinstmögliche Anzahl, also best fit. Aus diesem Schlüssel wird die Seitennummer genommen und die Seite zum einfügen des Satzes benutzt.

Die Schlüssel werden bei jeder Modifikation, welche die Anzahl der freien Bytes einer Seite verändert, aktualisiert. Diese Aktualisierung ist mit Löschen und Einfügen von Einträgen in den B* Baum verbunden, zusätzlich kommt auch der Aufwand zum evtl. Verschieben von anderen Sätzen in der Seite, um Lücken zu schließen. Deshalb ist in der Regel das Einfügen neuer Sätze und das Modifizieren von der Länge der Sätze viel langsamer als das Lesen.

Mit dem B* Baum sind auch andere Strategien zur Freispeicherverwaltung möglich. Es können z.B. Schwelwerte eingesetzt werden, d.h. in eine Seite darf nur eingefügt werden falls in der Seite mehr als n freien Bytes gibt.

5.3 Weitere Aspekte

5.3.1 Speicherung großer Objekte

Die Abspeicherung großer Objekte (größer als eine Seite) ist in den meisten Datenbanksysteme auf unterschiedliche Weisen realisiert. Die Sätze werden entweder in Seitengröße zerlegt und mittels datenbankinternen Methoden, oder extern, z.B. in Dateien des Betriebsystems, abgespeichert. Im zweiten Fall wird eine Referenz auf die externe Datei (beispielsweise URL) intern gespeichert.

In JXDB gibt es beide Möglichkeiten. Externes Abspeichern ist trivial. Da in JXDB nur die ersten 4 Schichten der 5-Schichtenarchitektur vorhanden sind, ist der Benutzer frei, das Format der Referenz auf das externe Objekt selber zu wählen.

Für die interne Speicherung gibt es wiederum zwei Möglichkeiten. Die erste ist das Objekt in einer Liste von Seiten zu speichern und eine Referenz auf diese Liste wie im externen Fall zu verwalten (Abbildung 5.4)

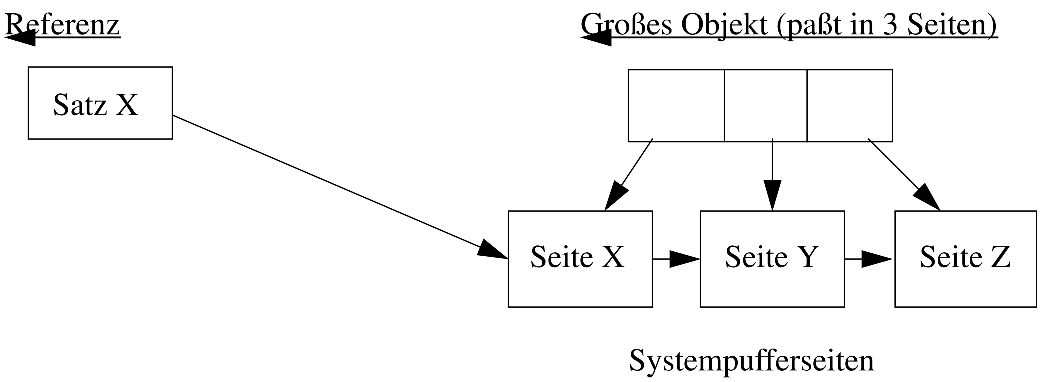


Abbildung 5.4 Interne Speicherung großer Objekte in Seiten des Systempuffers

Vorteilhaft für diese Methode ist, das sie einfach und daher performant ist. Es kann sich aber ein großer Verschnitt bilden, wenn viele Objekte verwaltet werden müssen, deren Größe nicht genaue Vielfache von der Seitengröße ist. Besonders schlecht kann das werden, wenn viele Mengen von kleinen Objekten gespeichert werden, die am Anfang fast leer sind (wichtigstes Beispiel: TID Listen, Nonunique Indizies, siehe nächstes Kapitel), später aber über die Seitengröße wachsen können. So wird beim Anlegen einer solchen Menge eine Seite allokiert, von der nur einige Bytes verwendet werden.

Die zweite Möglichkeit der internen Abspeicherung löst dieses Problem. Es können sowohl große Objekte gespeichert werden (vorher müssen sie in Stücke zerteilt werden), als auch Mengen von Objekten fester Größe, die kleiner als eine Seite sind (Abbildung 5.5).

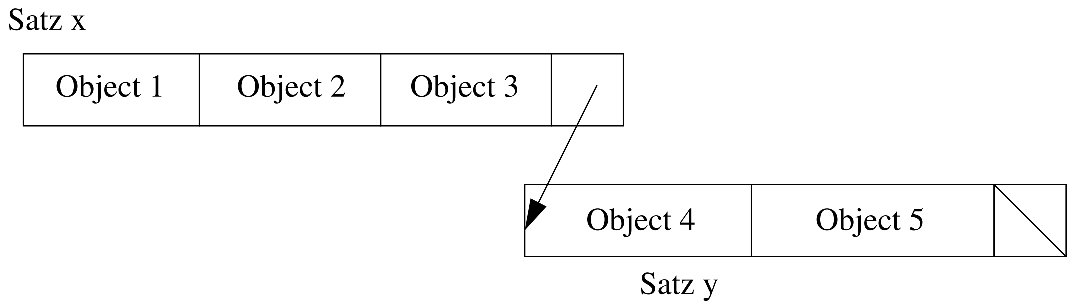


Abbildung 5.5. Abspeicherung einer Menge von gleich große Objekte mittels Sätze

Nehmen wir an, daß die Objekte (Byteketten) 1 bis 5 in dieser Reihenfolge abgespeichert werden. Dann enthält am Anfang Satz x nur das Objekt 1, Satz y existiert noch nicht, das Endelment von Satz x verweist auf Null. Beim Einfügen von den Objekten 2 und 3 wächst Satz x nach den Regeln in 5.2.3. Das Einfügen von Objekt 4 würde die maximale Satzlänge (die kleiner als die Seitenlänge ist !) überschreiten, deswegen wird Satz y erzeugt und dem Satz x ein Verweis auf Satz y hinzugefügt. Satz y verweist wieder auf Null, weil er der letzte Satz der Kette ist.

Es ist jedoch zu beachten, daß jedes Einfügen in einer solchen Menge mit einer Modifikation der Satzlänge verbunden ist. Wie in 5.2.3 schon erwähnt, ist das eine in der Regel teure Operation. Der Einsatz der ersten Möglichkeit ist also vorzuziehen, falls keine so große Flexibilität gebraucht wird.

5.3.2 Performanz vs. Speicherplatzeffizienz

Die in Abbildung 5.1 vorgestellte Struktur einer Seite, die zur Abspeicherung von Sätzen benutzt wird, dient nur um Überblick zu gewinnen. Viele der Details bleiben der konkreten Implementierung offen. Eine Frage ist z.B. wie die Längen der Sätze ermittelt werden Können, falls sie nicht explizit gespeichert werden. In JXDB werden die Satzlängen aus den physischen Positionen der Sätze innerhalb der Seite berechnet. Das setzt aber voraus, daß die Sätze physisch in der Reihenfolge liegen, in der ihre logische Nummer sind und daß es keine Lücken gibt. Damit es keine Lücken gibt, müssen Sätze innerhalb einer Seite bei jedem Einfügen, jedem Löschen und jeder Längenänderung möglicherweise verschoben (d.h. kopiert) werden. Es sind Strategien denkbar, bei denen Lücken nur on demand aufgeräumt werden und somit unnötiges Kopieren gespart wird. Sie sind aber nicht so effizient bezüglich des Speicherplatzes, zusätzliche Datenstrukturen wie die Seitznlänge und die Information über die Lücken müssen pro Satz verwaltet werden, auch die Implementierung ist nicht trivial.

5.4 Implementierung

Die wichtigen Klassen der physischen DB-Verwaltung sind wie bei der Systempuffer- und Seitenverwaltung in abstrakter und konkreter Teil zerlegt (Interface und Implementierung), damit der konkrete Teil ausgewechselt werden kann. Es gibt einige Hilfsklassen, für denen es keine solche Trennung gibt.

5.4.1 Überblick der Klassen

Die Klassen mit Interfaces sind db.tid.SetAccessImpl, db.tid.SetPageImpl und db.tid.SetArrayImpl. Ihre Interfaces sind entsprechend db.tid.SetAccess, db.tid.SetPage und db.tid.SetArray.

Die Hilfsklassen, die hier erläutert werden sind db.tid.SetNumber, db.tid.Set und db.systembuffer.collections.BufferList

5.4.2 SetAccess und SetAccessImpl

SetAccess und seine Implementierung SetAccessImpl sind Teil der Schnittstelle der Schicht für die oberen Schichten. SetAccess bietet den Benutzerthreads Funktionen zum Einfügen, Modifizieren und Löschen von Sätzen. Es können zusätzlich komplette Satztypen auf einmal gelöscht werden. SetAccess organisiert die Freispeicherverwaltung mit Hilfe vom B* Baum. Es ist auch für die Migration von Sätzen zuständig. Es weis aber nichts über den Aufbau der Seiten. Das ist Aufgabe der Klasse, die das SetPage Interface implementiert.

SetAccess benutzt die PageManager und PageBuffer Interfaces, um Seiten für die Satzverwaltung anzufordern und zu manipulieren.

5.4.3 SetPage und SetPageImpl

SetPageImpl verwaltet die Sätze innerhalb einer Seite. Das ist die Klasse, wo alle Informationen über den Aufbau der Seiten, die zur Satzspeicherung dienen, gekapselt ist. Sie wird nur vom SetAccess verwendet und bietet ihm Funktionen zum Einfügen, Löschen und Modifizieren von Sätzen an. SetAccess kann zusätzlich überprüfen, ob ein Satz Daten enthält, oder verweist auf einen anderen Satz (Indirektion). SetPageImpl kann auch die Anzahl der freien Bytes in der Seite liefern. Ein möglicher Ablauf beim Lesen könnte so aussehen:

1. Der Benutzerthread möchte einen neuen Satz hinzufügen. Er ruft die addSet Funktion vom SetAccess mit der gewünschten Satzlänge auf.
2. SetAccess sucht im B* Baum eine Seite aus, wo der neue Satz hineinpaßt. Falls keine passende Seite gefunden wird, wird eine neue Seite vom PageManager angefordert. Es wird in diesem Fall einen neuen Eintrag in den B* Baum eingefügt.
3. SetAccess konstruiert ein neues SetPage Objekt (oder reinitialisiert ein Altes, ist effizienter) und übergibt ihm eine Referenz auf die gefundenen Seite. Falls die Seite ganz neu ist, wird sie vom SetPage formatiert und als leer markiert.

4. SetAccess ruft die Funktion addSet vom SetPage auf. SetPage aktualisiert entsprechend die Seite und liefert ein Set Objekt, mit dem auf den neuen Satz zugegriffen werden kann (siehe die Beschreibung vom Set unten). Dieses Set Objekt wird vom SetAccess dem Benutzerthread zurückgeliefert.

5.4.4 Set

Ein wichtiges Konzept, das oben nicht erwähnt wurde, weil es auch in anderen Schichten vorhanden ist, ist das Vermeiden von Kopiervorgängen, indem SetAccess den oberen Schichten Referenzen auf die Seiten gibt, statt irgendwelche Zwischendaten. Z.B wenn ein Benutzerthread die Daten eines Satzes braucht, wird der Satz nicht in einem Zwischenpuffer gelesen, sondern dem Thread wird nur die Seitenreferenz, wo der Satz liegt, der physische Offset innerhalb der Seite und die Satzlänge übergeben. Der Thread kann mit diesen Informationen selber den Inhalt lesen und modifizieren.

Nachteilig ist natürlich, daß der Benutzerthread auf diese weise andere Sätze aus versehen beschädigen kann, z.B. wenn er die zurückgelieferte Satzlänge oder Offset nicht beachtet. Das wird aber in JXDB zugunsten einer erheblich besseren Performanz in Kauf genommen. Es ist außerdem für Datenbankapplikationen ungefährlich, weil sie nur die Schnittstelle von Schicht 4 benutzen und auf physische Datensätze direkt nicht zugreifen sollten.

Die Seitennummer, Satzoffset und Satzlänge werden in einem einzigen Set Objekt gekapselt, um einfacher und sicherer damit arbeiten zu können.

5.4.5 SetNumber

db.tid.SetNumber ist eine Hilfsklasse, die eine Satzadresse repräsentiert und bequeme Funktionen zum Manipulieren von Satzadressen zur Verfügung stellt.

5.4.6 SetArray

Die Klasse db.tid.SetArray wird für die interne Abspeicherung von Mengen von Objekten gleicher Größe benutzt , die kleiner als eine Seite sind (siehe 5.3.1, Abbildung 5.4). Sie stellt Funktionen zum Einfügen, Löschen und Modifizieren von Sätzen. Der Benutzer kann durch die Sätze von Anfang in Richtung Ende iterieren. Es ist möglich, die ganze Menge auf einmal zu löschen.

5.4.7 BufferList

Die Klasse db.systembuffer.collections.BufferList dient zur effizienten Abspeicherung von großen Objekten. Sie befindet sich nicht im tid package, weil sie auch von anderen Schichten (Schicht 4) benutzt wird. Die Klasse liefert ungefähr dieselbe Funktionalität wie SetArray, die Abspeicherungsmethode unterscheidet sich jedoch (siehe Abbildung 5.3)

BufferList wird momentan für den Update von Tabellen eingesetzt. In der Klasse db.test.SyscatTest gibt es ein Beispiel für diese Anwendung. (Siehe Anhang B, Benchmarks und Beispiele)

5.5 Zusammenfassung

Es wurden einige der Konzepte der physischen DB-Verwaltung und die Klassen, die sie implementieren, vorgestellt. In dieser Schicht gibt es eine Reihe von Optimierungsmöglichkeiten, welche die Performance der ganzen Datenbanksystem beeinflussen können. Deswegen ist es wichtig für jede konkrete Situation die geeignete Lösung zu verwenden. Ausführliche Klassendokumentation findet man in dem Sourcecode (javadoc).

Kapitel 6

Interne DB-Verwaltung

6.1 Aufgaben der Internen DB-Verwaltung

Die physische DB-Verwaltung bietet schon einfache Operationen zum Einfügen, Modifizieren und Löschen von Daten in das Datenbanksystem an. Zwischen diesen Operationen und einem Datenmodell besteht jedoch große Lücke, die durch die interne DB-Verwaltung gefüllt werden soll. Die Aufgabe dieser Schicht ist es, die Daten in einer solchen Form zu bringen, in der sie leicht und effizient in dem entsprechenden Datenmodell bearbeitet werden können. Beispielsweise sollten beim multidimensionalen Modell Hilfsstrukturen existieren, die das effiziente Ausschneiden eines Teilwürfels von der ganzen Datenmenge gestatten. Bei dem relationalen Datenmodell braucht man entsprechend Unterstützung für oft ausgeführten Operationen auf Tabellen, wie Join, Select, etc. Diese Hilfsstrukturen werden allgemein als Indizies bezeichnet.

Die interne DB-Verwaltung stellt der logischen DB-Verwaltung Operationen zur Manipulation der Hilfsstrukturen und der eigentlichen Daten zur Verfügung. Diese Operationen können in eine *operationale Sprache* zusammengefaßt werden. Eine solche Sprache, die um Teil in JXDB realisiert ist, ist die relationale Algebra. Wenn man eine Anfrage in einer operationalen Sprache formulieren will, muß man angeben, wie genau die Anfrage ablaufen soll. Bei einer deskriptiven Sprache dagegen braucht man nur anzugeben was selektiert werden soll.

Eine andere Aufgabe dieser Schicht ist es, Informationen über die Benutzerdaten zu verwalten, die sogenannten *Metadaten.* Wenn es sich um eine relationale Datenbank handelt, könnten zum Beispiel für jede vorhandene Relation der Name der Relation, die Anzahl und die Namen der Attribute, sowie ihre IDs, die IDs ihrer Indizes usw. abgespeichert werden. Für JXDB ist die Struktur dieser Metadaten in 5.2 Beschrieben. Es ist sinnvoll die Metadaten auf exakt die gleiche Art und Weise zu repräsentieren, wie die anderen Benutzerdaten. So könnte der Benutzer die bequemen Operatoren der Schicht 5, mit denen er schon vertraut ist, benutzen um sie abzufragen.

6.2 Konzepte und Techniken der internen DB-Verwaltung

6.2.1 Tabellen

Alle Benutzerdaten werden in JXDB in Form von internen Tabellen gespeichert. Diese Tabellen sind den Tabellen des relationalen Datenmodells ähnlich, können aber indiziert werden (siehe 6.2.2). Es gibt auch keine primary und foreign keys, sowie einige andere Unterschiede. Eine interne Tabelle besteht aus Zeilen und Spalten. Jede Spalte hat einen eindeutigen Namen (eindeutig innerhalb der Tabelle). Die Tabelle selbst hat einen systemweit eindeutigen Namen.

Wichtig zu diesem Zeitpunkt ist es, daß jede Zeile in einem Satz der physischen DB-Verwaltung gespeichert wird. Um auf allen Daten der Tabelle zugreifen zu können, muß mindestens ein Zugriffspfad (Index) existieren, der die Adressen aller Sätze der Tabelle enthält.

6.2.2 Indizes allgemein

Allgemein sind Indizes redundante Informationen, die auf irgendeine Weise nützlich für die schnelle Abarbeitung der Datenbankanfragen sind.

Als Indizes können Baumstrukturen, Hashtabellen, Bitmaps, verkettete Listen, etc. Eingesetzt werden. Sie können die Leseoperationen beträchtlich beschleunigen, müssen aber bei Änderungsoperationen aktualisiert werden, was Aufwand bedeutet. Durch geschicktes Anlegen von Indizes, wie z.B. Indizieren on demand kann dieser Aufwand minimiert werden. Die Bereitstellung von passenden Indizes ist Aufgabe der internen DB-Verwaltung, der geschickte Gebrauch dagegen gehört zur logischen DB-Verwaltung.
Die Indizes der JXDB basieren alle auf dem B*Baum. Es ist aber möglich auch andere Typen von Indizes zu unterstützen, wie multidimensionale Indizes.
Ein Beispiel für Indizierung mit B*Baum ist in Abbildung 6.1 dargestellt.

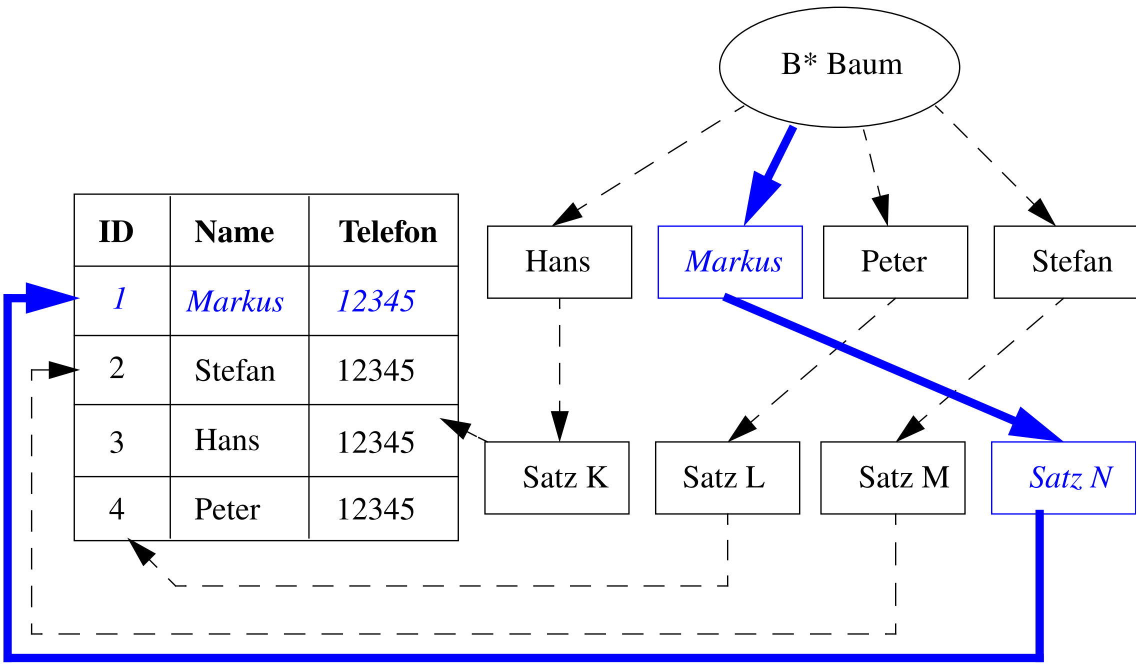


Abbildung 6.1. Indizierung von einem unique Attribut mit B* Baum

Die Tabelle Phones besteht aus 4 Zeilen. Jede Zeile ist intern in einem Satz der physischen DB-Verwaltung gespeichert. Das sind die Sätze mit Nummern K, L, M, N, die für die interne DB-Verwaltung transparent ausgewählt werden. Die Spalte Name der Tabelle ist indiziert. Jeder der Namen in der Spalte ist als Schlüssel in dem B*Baum gespeichert. (zur Erinnerung: der B*Baum speichert Paare < Schlüssel, Data > in sortierte Reihenfolge). Die Daten sind die Adressen der Sätze, wo die Zeilen der Tabelle gespeichert werden. Angenommen man hat den Schlüsselwert „Markus“. Über den B* Baum kann man in O(logN) Schritten die Adresse des Satzes bekommen, wo die erste Zeile der Tabelle (1, Markus, 12345) gespeichert ist. Jetzt kann man den Satz lesen und an die Telefonnummer von Markus gelangen.

6.2.3 Unique Indizes

In dem obigen Beispiel wiederholen sich die Schlüsselwerte nicht. Man spricht von einem unique Index. Ein Versuch ein Schlüssel zwei mal in den Index einzufügen wird in JXDB aus Gründen der Performanz nicht abgefangen. In dem B* Baum wird einfach das alte <Schlüssel, Daten> Paar durch das Neue überschrieben.

6.2.4 Non unique Indizes

Wenn sich die Schlüsselwerte einer Spalte wiederholen, kann diese Spalte nur mit einem non unique Index indiziert werden. Die in JXDB implementierte Variante von dem B* Baum kann keine mehrfachen Schlüssel speichern. Um einen non unique Index mittels B* Baum zu realisieren, muß man eine zusätzliche Indirektionsstufe einführen, wie in der Abbildung 6.2 zu sehen ist.

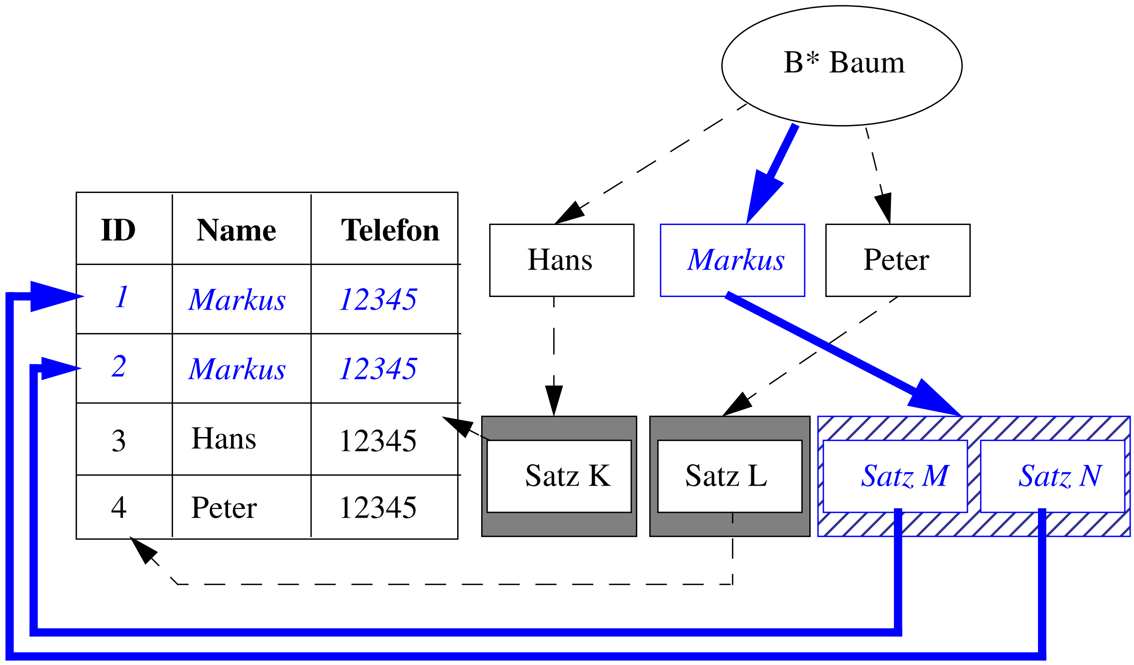


Abbildung 6.2. Indizierung von einem non unique Attribut mit B* Baum

Statt einzelneSatzadressen sind jetzt die Daten in den Paaren <Schlüssel, Daten> Verweise auf Mengen von Satzadressen. Diese Mengen werden mit der Klasse SetArray bearbeitet.

Einen Zugriff auf die Tabelle über diesen Index könnte so aussehen:

1. Der Benutzer sucht im B* Baum den Eintrag mit dem Schlüssel „Markus“
2. Ein Paar <“Markus“, X > wird gefunden und zurückgeliefert, wo X die Adresse von einem SetArray ist.
3. Der Benutzer konstruiert ein neues SetArray Objekt mit X als Anfangsadresse (siehe Kapitel 5, sowie Dokumentation im Sourcecode).
4. Der Benutzer liest die erste Bytekette im SetArray, N, und konstruiert ein SetNumber Objekt damit.
5. Der Benutzer liest den Satz N.
6. Der Benutzer liest die zweite Bytekette im SetArray, L (siehe SetArray.moveToNext)
7. Satz L wird gelesen.

Man sieht, daß beim non unique Zugriff fast doppelt so viele Operationen ausgeführt werden müssen, als beim unique Zugriff. Daher ist der nonunique Index wesentlich langsamer. Die Aktualisierung von nonunique Indizes ist auch langsamer, weil SetArray Objekte in der Regel langsam funktionieren. Deswegen sollte man unique Indizes verwenden, überall wo es möglich ist.

Wenn man ein Satz von der Tabelle löschen will, müssen die Indizes entsprechend aktualisiert werden. Bei dem unique index ist das relativ einfach, der Schlüssel wird einfach vom B*Baum entfernt. Bei dem non unique Index dagegen ist es viel komplizierter. Zuerst muß überprüft werden, wieviele Elemente in dem SetArray zu diesem Schlüssel existieren. Wenn der zu löschende Satz der einzige mit diesem Schlüssel ist (Sätze K und M in der Abbildung 6.2), können das SetArray und das Paar im B*Baum gelöscht werden. Wenn es aber mehrere Satzadressen im SetArray gibt (Sätze N und L in der Abbildung 6.2), darf nur die entsprechnde Satzadresse vom SetArray entfernt werden. Dazu muß unter Umständen das ganze SetArray durchgelaufen werden, was bei vielen gleichen Schlüsseln ineffizient werden kann. Das sollte man beim Entwurf der Datenbank beachten.

6.2.5 Eindimensionale Indizes für ein Attribut

Das sind die gerade vorgestellten Indizes. Bei denen wird nur eine einzige Spalte indiziert. Mit solchen Indizes kann man gut exact match Anfragen (Anfragen, die eine einzige Zeile, Tupel als Ergebnis liefern), die sich auf diese Spalte beziehen, ausführen. Bereichsanfragen, die wiederum begrenzt auf diese eine Spalte sind, können auch leicht ausgeführt werden. Problematisch wird es, wenn mehrere Spalten in der Anfrage einbezogen werden. Man könnte in diesem Fall z.B. eine exact match Anfrage über 2 Attribute in zwei Bereichsanfragen über 1 Attribut aufspalten und dann die Schnittmenge der Ergebnisse der beiden Bereichsanfragen bilden, was in der Regel ineffizient ist.

6.2.6 Eindimensionale Indizes für mehrere Attribute

Die oben genannte exact match Anfrage über 2 und mehrere Attribute kann effizient mit Hilfe von eindimensionalen Indizes für mehrere Attribute ausgeführt werden. Diese Indizes in JXDB basieren wieder auf dem B*Baum, ihre Schlüssel sind aber aus den Werten aller beteiligten Spalten zusammengesetzt(Abbildung 6.3)

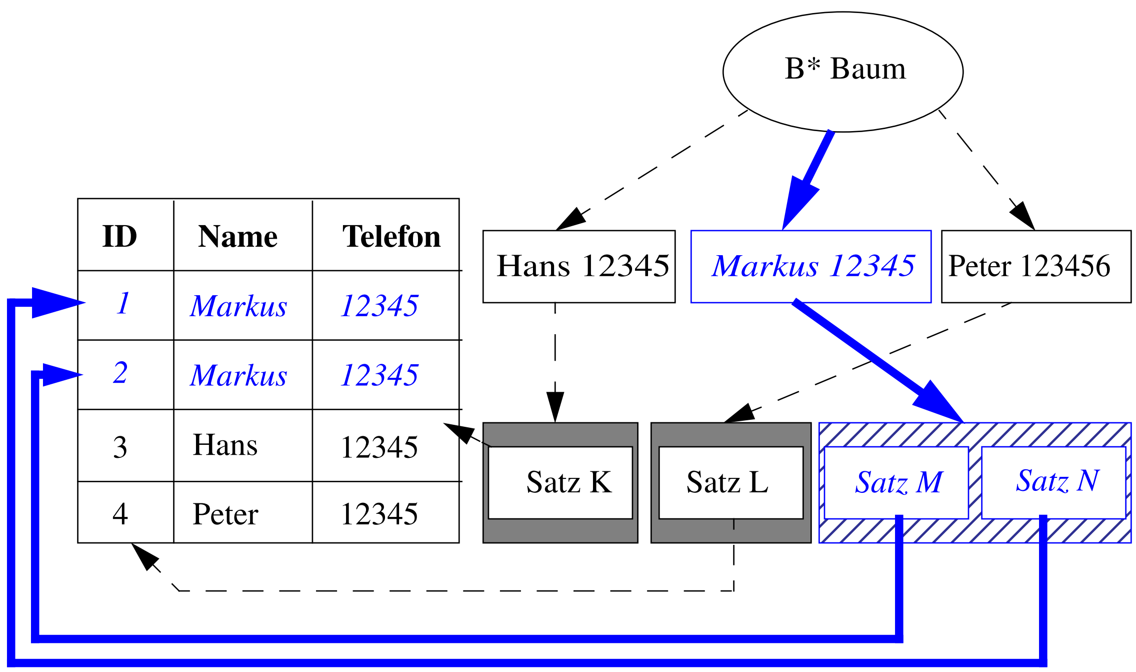


Abbildung 6.2. Indizierung von zwei non unique Attributen mit B* Baum

In der obigen Abbildung werden die non unique Spalten Name und Phone indiziert. Der resultierende Index ist auch non unique (der Schlüsselwert „Markus“ tritt 2 mal auf). Es ist aber möglich, daß von zwei non unique Spalten ein unique Index entsteht. Wenn mindestens eine der Spalten unique ist, ist der Index mit sicherheit auch unique. Das kann manchmal praktisch sein, weil so kann man leicht für non unique Indizes eine effizientere äquivalente unique Variante konstruieren. Beispielsweise möchte man die Spalte Name indizieren. Man könnte einen ineffizienten non unique Index mit einem Attribut bauen, oder einen unique Index mit 2 Attribute über die Spalten Name und ID (Reihenfolge ist wichtig!). Eine Anfrage für den non unique Index würde so aussehen:

Suche alle Sätze mit Schlüssel im B*Baum gleich „Peter“.

und die entsprechende Anfrage für den unique Index:

Suche alle Sätze mit Schlüssel im B*Baum von „Peter0“ bis „Peter65535“

falls ID 2 Byte lange unsigned integer Zahl ist.

Noch ein Beispiel für trickreiche Anwendung von zusammengesetzten Schlüssel ist die Freispeicherverwaltung der physischen DB-Verwaltung (siehe 5.2.4.).

6.2.7 Mehrdimensionale Indizes

Wie gesehen, kann man mit eindimensionalen Indizes mit einem oder mehreren Attributen ziemlich vieles erreichen. Es gibt aber Fälle, wo echte multidimensionale Indexstrukturen die effizienteste Lösung wären. Angenommen man hat die 3D Koordinaten einer Menge von räumlich verteilten Punkten in der Datenbank gespeichert. Wie könnte man z.B. die Anfrage „gib mir alle Punkte mit X zwischen 2 und 5, Y zwischen 3 und 4 und Z zwischen 2 und 2.5“? Solche anfragen findet man z.B. in CAD Systemen. Erstens sollte man sowieso die Daten in eine Tabelle speichern, z.B. mit 3 Spalten, eine für jede Koordinate. Man könnte für jede der Spalten einen eindimensionalen Index erzeugen. So könnte man für die konkrete Anfrage die 3 Teilmengen (X zwischen 2 und 5, Y zwischen 3 und 4 und Z zwischen 2 und 2.5) bilden. Die gesuchte Ergebnismenge wäre dann die Schnittmenge der 3 Teilmengen. Das funktioniert sicher, ist aber ineffizient, weil Schnittmenge zu bilden in der Regel eine Teure Operation ist.

Mit einem eindimensionalen Index mit mehreren Attributen kann man auch nicht sehr viel mehr erreichen. Was man braucht, ist ein Index, der die Schnittmengen bildende Operation intern effizient realisiert. Das ist möglich, indem die Daten intern nach multidimensionalen Muster abgespeichert werden. Solche Indizes sind multidimensionale Bäume, Gridfiles und viele andere. In JXDB ist keiner davon implementiert worden, neue Implementierungen können aber problemlos integriert werden.

6.2.8 Datentypen, Komparatoren und Konverters

Im Zusammenhang mit den Indizes sollte auch etwas über Datentypen erwähnt werden. Wie gesagt, speichern Indizes Schlüsseln in sortierte Reihenfolge ab. Diese Schlüssel sind entweder Elemente einer Spalte einer Tabelle, oder sind aus solchen Elementen zusammengesetzt. Diese Elemente sind Byteketten, die aber für verschiedene Spalten verschiedene Längen und Bedeutung haben. Beispielsweise sind die Elemente in der Spalte ID der tabelle Phones immer 4 Byte lang und werden als Integers interpretiert. Die Elemente der Spalte Name haben z.B. eine Länge von 20 Bytes, werden aber im Unterschied zu den Ids als Strings interpretiert, d.h. ihre Sortierfunktion ist unterschiedlich. Das sollte ein B*Baum beachten, er könnte aber im Voraus nicht wissen, wiviele verschiedene Datentypen es überhaupt gibt. Momentan gibt es in JXDB nur Integers und beliebig lange (aber kleiner als die maximale Satzlänge) Zeichenketten (Strings). Man könnte aber ohne viel Mühe neue Datentypen dem System hinzufügen.

Man könnte das Problem lösen indem man die Sortierungsfunktion von der Implementierung des B*Baums trennt. In JXDB gibt es für jedes Datentyp eine Komparator Klasse, die von dem Interface Comparator abgeleitet ist. Sie stellt dem B*Baum (und auch anderen Klassen) die Funktion compare zu Verfügung, die zwei Byteketten von dem jeweiligen Datentyp richtig vergleicht. Der B*Baum bekommt ein Objekt von einer solchen Klasse als Parameter, um seine Schlüssel sortieren zu können.

Oft haben die Datentypen in JXDB auch Java Äquivalenten, die bequemer zu benutzen sind, als Byteketten. Deswegen gibt es in JXDB eine menge von DbConverter implementierenden Klassen, die zur Umwandlung von Byteketten in Java Objekten und umgekehrt dienen.

6.2.9 Systemkatalog

Der Systemkatalog enthält Metadaten über die Benutzerdaten, die zur Zeit in der Datenbank gespeichert sind. Diese Metadaten werden von der internen DB-Verwaltung gewartet und benutzt um die ganze Datenbank in einem Konsistenten zustand zu halten. Es werden zur Zeit folgende Informationen verwaltet:

1. Tabelleninfo: Für jede Tabelle wird der Name der Tabelle und die Tabellen ID gespeichert. Beide sind eindeutig, die meisten Funktionen der internen DB-Verwaltung nehmen Tabellen IDs als Parameter, deswegen bietet das Systemkatalog die Möglichkeit an, nach dem ID mittels des Tabellennamens zu suchen.
2. Attributinfo: Für jedes Attribut wird eine eindeutige Attribut ID, die ID der Tabelle, zu der das Attribut gehört, die Position des Attributs innerhalb der Tabelle (wichtig für die Indizes), das Typ, der Name und die Bytelänge gespeichert.
3. Datentypinfo: Für jedes der Datentypen wird die Datentyp ID und der Name gespeichert
4. Indexinfo: Für jeden der existierenden Indizes wird die Index ID, die Nummer der ersten Seite, die der Index belegt, ein Flag, der anzeigt, ob der Index unique ist, und das Indextyp (ob eindimensional oder mehrdimensional) gespeichert.
5. Weitere Indexinfos: für jeden Index werden alle IDs der Attributen und ihre Positionen gespeichert, aus denen der Schlüssel der Index zusammengesetzt ist.

Diese ganze Informationen kommen entsprechend in 5 Tabellen, die Relation, Attribute, Index, Datatype und IndexMap. Ihre Headers sind hier aufgelistet:

Relation Table

RID: Integer	Rname:String

Datatype Table

DID: Integer	Dname:String

Index Table

IID:Integer	INDNO(Page Number):Integer	ITYPE(unique or non unique):Integer	CARD(single attribute, multiattribute, multidim):Integer

IndexMap Table

IID:Integer	AID: Integer	APOS(Attribute position):Integer

Attrubute Table

AID:Integer	Aname:String	RID:Integer	APOS:Integer	DID:Integer	Size:Integer

Die Relation Tabelle hat zwei unique Indizes, einen über RID und einen über Rname. Die Datatype Tabelle hat einen unique Index über DID, wird auch tatsächlich nur dazu benutzt zu überprüfen, ob eine DID gültig ist, wenn der Benutzer versucht, sie zu verwenden.

Die Index Tabelle hat einen unique Index über die IID, weil alle Zugriffe auf ihre Daten übder die IID erfolgen

Die IndexMap Tabelle hat zwei non unique Indizes über IID und AID. Ein Index kann mehrere Attribute indizieren und ein Attribut kann von mehreren Indizes gleichzeitig indiziert werden, deswegen sind die Spalten IID und AID in der IndexMap Tabelle non unique. Es gibt zwei Anfragen, die auf diese Tabelle zugreifen, die erste sucht alle IIDs für eine AID, und die zweite sucht alle AIDs für eine IID. Deswegen existieren auch die zwei Indizes.

Die Attribute Tabelle hat einen unique Index über AID und einen non unique Index über RID. Der erste beschleunigt den Zugriff auf Attributsätze über AID, der zweite gestattet die effiziente Suche nach allen Attributen einer Relation.

Jedes mal wenn eine Tabelle oder ein Index erzeugt oder gelöscht wird, müssen die entsprechenden Tabellen vom Systemkatalog aktualisiert werden. Damit sie vollständig sind, enthalten sie auch die Informationen über sich selber. Z.B. sind in der Index Tabelle Daten über die Indizes der Relation, Datatype, IndexMap, Index und Attribute Tabelle enthalten. Das bedeutet, daß sogar bei leerer Datenbank die Tabellen des Systemkatalogs einiges an Daten enthalten. Der Systemkatalog ist dafür zuständig sie beim Erzeugen einer neuen Datenbank in einen konsistenten Zustand zu bringen.

Die Tabellen des Systemkatalogs sollten von dem Normalbenutzer nicht modifiziert, sondern nur gelesen werden. In JXDB gibt es noch keine Kontrolle darüber.

Das ER-Diagramm auf der Abbildung 6.3 veranschaulicht noch einmal die Zusamehänge zwischen der Tabellen.

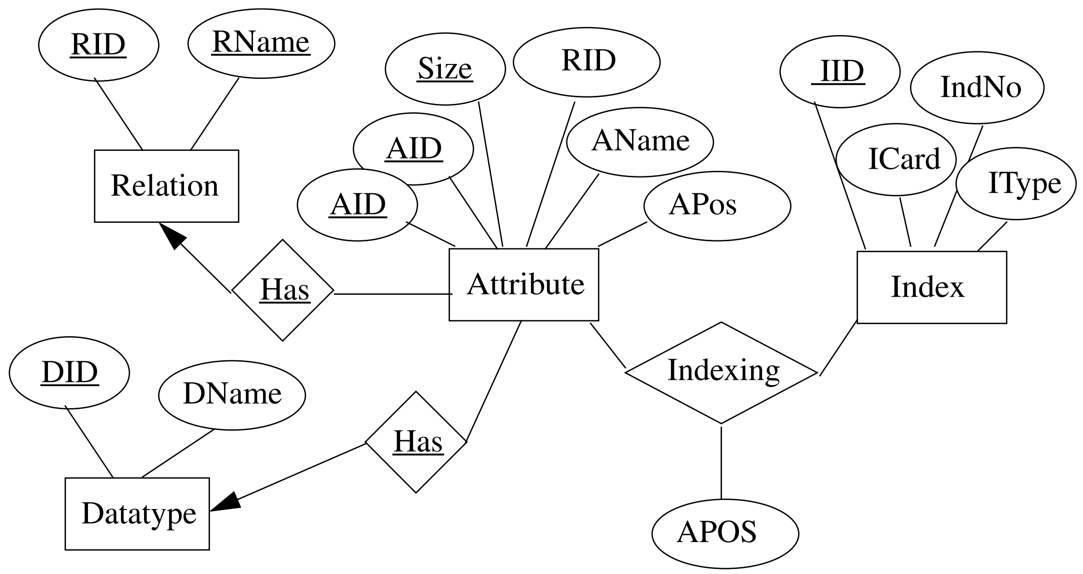


Abbildung 6.3. ER-Diagramm der Tabellen des Systemkatalogs

6.2.10 Relationale Operatoren

Die interne DB-Verwaltung bietet der logischen DB-Verwaltung eine Menge von Operatoren zur Datenmanipulation. In JXDB ist ein Teil der relationalen Algebra implementiert worden. Es können neue Operatoren jederzeit in das System integriert werden.

Jeder Operator bekommt als Eingabe eine Menge von Tabellenzeilen (Tupeln, Sätze). Seine Ausgabe ist auch eine Menge von Tabellenzeilen. Wichtig ist, daß die Eingabemenge eines Operators die Ausgabemengen von einem oder mehreren anderen Operatoren sein kann. So kann man Operatorenketten bilden, um komplexe Anfragen darzustellen.

Jeder Operator erlaubt iterativen Zugriff auf seine Ausgabemenge. Es gibt die Standardfunktionen eines Iterators - moveToFirst, moveToNext, die den Zugriff auf jedes Element der Menge erlauben. In den Operatorenketten benutzen die Operatoren diese Funktionen bei ihren untergeordneten Operatoren, um die Eingabe einzulesen.

Die vorhandenen Operatoren werden im jetzt vorgestellt, nachher folgen einige Beispiele.

6.2.11 IndexScan

Der IndexScan Operator erlaubt Iteration durch die Daten einer Tabelle. Der Zugriff wird über einen Index gemacht (jede Tabelle besitzt mindestens einen Index, sonst ist sie nutzlos). Die Tupelmenge kann durch Start- und/oder Endbedingung begrenzt werden, falls der Index das erlaubt.

Der IndexScan ist der einzige Operator in der JXDB, der direkt auf die Datenbank zugreift und keine untergeordnete relationale Operatoren braucht. In den Operatorenketten (Operatorenbäume) können die Blätter nur IndexScans sein.

6.2.12 Selektion

Der Selektionsoperator filtert Tupel von der Eingabemenge nach bestimmten Bedingungen raus. Die Bedingungen werden von dem Benutzer als Parameter angegeben. In der JXDB ist die Bedingung ein Objekt, das entscheiden kann, ob ein Tupel sich für die Ausgabemenge qualifiziert oder nicht.

6.2.13 Projektion

Der Projektionsoperator formt die eingehenden Tupeln um, indem er nur die vom Benutzer angegebenen Tabellenspalten beibehält. Beispielsweise wird das eingehende Tupel (1, Markus, 12345) in das Tupel (Markus, 12345) transformiert, weil der Benutzer nur die Spalten Name und Phone projeziert haben will.

Bei der Projektion können Duplikate entstehen, etwa wenn z.B. eine unique Spalte weggenommen wird. In dem strikten relationalen Datenmodell dürfen keine duplizierte Zeilen in eine Tabelle existieren (Die Ausgabemengen der Operatoren können eigentlich auch als Tabellen gesehen werden, weil alle Tupel vom gleichen Typ sind). Da aber die Duplikateliminierung eine teure, mit Sortieren verbundene Operation ist, wird sie nicht implizit in der Projektion, sondern explizit vom Benutzer ausgeführt.

6.2.14 Verbund (Join)

Der Verbundoperator faßt Daten von mehreren Tabellen in eine Tabelle zusammen.

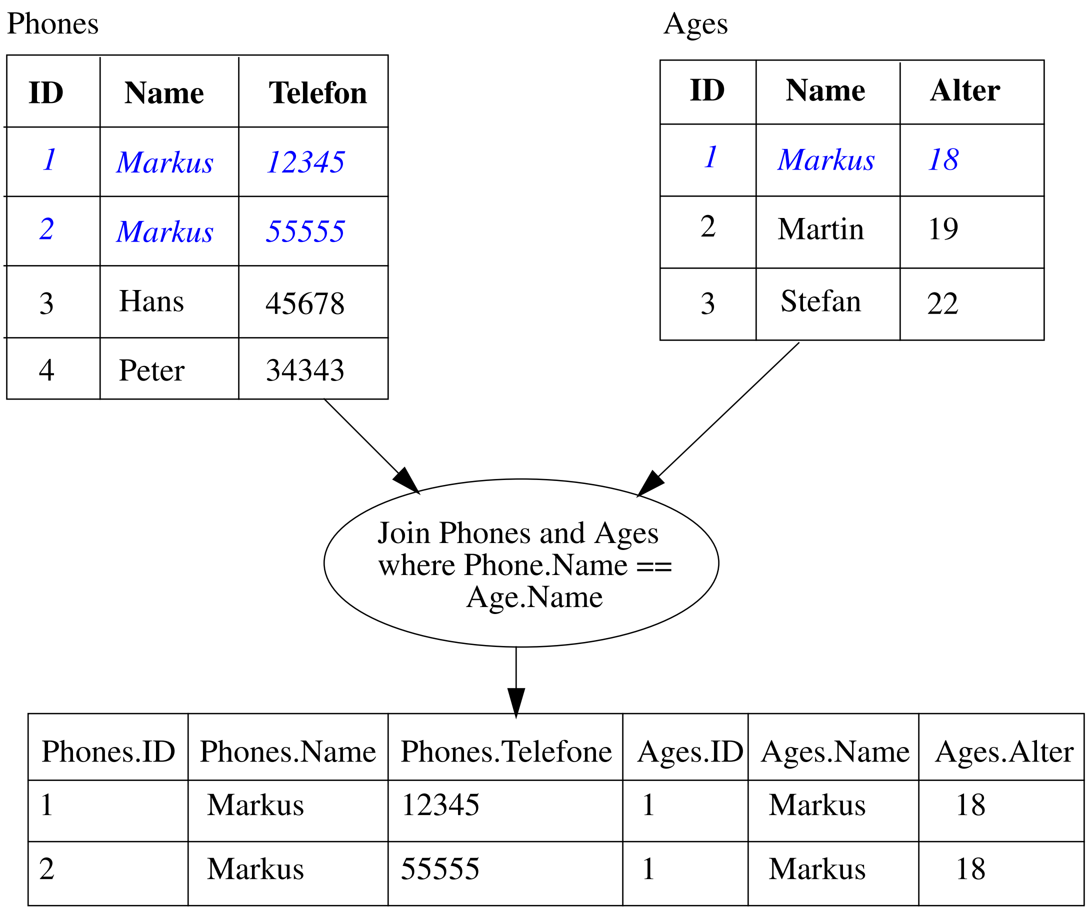


Abbildung 6.4. Einfaches Verbundbeispiel

Alle Verbundoperationen auf mehr als 2 Tabellen können aus einfachen Verbunden auf 2 Tabellen zusammengebaut werden (Beispiele folgen). Die Verbundbedingung auf der Abbildung 6.4 ist Gleichheit, allgemein können alle mögliche logische Operatoren benutzt werden. Die Gleichheit erlaubt aber eine viel effizientere Implementierung des Verbund-operators, als eine beliebige Bedingung. Für allgemeine Bedingungen ist immer der Nested Loop Join anwendbar. Bei diesem Join werden die beteiligten Tabellen einfach geschachtelt iteriert, bis alle Kandidattupel verbunden sind. Er ist einfach zu implementieren, bei großen Tabellen aber sehr langsam

Nur bei Equijoins (wo die Bedingung die Gleichheit ist) kann man den HashJoin anwenden. Dabei wird eine Hashfunktion benutzt um schnell Tupelpaare zum Verbinden zu finden. In der JXDB ist noch keinen HashJoin implementiert, ist aber möglich, eine effiziente Lösung zu konstruieren, die auf das bisher Gebaute basiert.

6.2.15 Sortieren und Eliminieren von Duplikaten

Der Sortieroperator in der JXDB benutzt den B*Baum um die Eingabemenge über einige Attribute zu sortieren. Der Benutzer kann angeben, ob bei der Sortierung die Duplikaten eliminiert werden sollen, wobei die Sortierung mit Duplikateliminierung effizienter ist als ohne. Es ist so, weil ohne Duplikateliminierung sollte der B*Baum als non unique Index verwendet werden. (siehe 6.2.4).

Es gibt effizientere Sortierverfahren, als alle Tupel in einen B*Baun einzufügen und dann sie in sortierter Reihenfolge auszulesen, z.B. Quicksort. Der Quicksort muß aber komplett im Hauptspeicher ablaufen, was bei großen Tabellen nicht möglich ist. Es gibt auch Sortierverfahren, die mit der Festplatte arbeiten können, sind aber nicht sehr viel schneller als der B*Baum.

6.2.16 Beispiele

1. Zur Selektion: Die Quelle sei die Tabelle Phones auf der Abbildung 6.4. Eine Selektion wäre die Anfrage „Finde alle Zeilen aus, wo die ID ungleich 3 und größer 1 ist“. Des würde die folgenden Zeilen liefern:

ID	**Name**	**Phone**
2	Stefan	12345
4	Peter	12345

2. Zur Projektion: Wieder auf die Tabelle Phones bezogen, ist eine Anfrage der Art „Gib mir die ID und den Namen von Phones“ eine Projektion und würde Folgendes liefern:

ID	**Name**
1	*Markus*
2	Stefan
3	*Markus*
4	Peter

3. Zum Verbund: Angenommen hat man die Tabellen Phones und Ages von der Abbildung 6.4. und noch eine Tabelle Adresses, die den folgenden Inhalt hat:

Addresses

ID	Name	Address
1	Markus	XXXX
2	Stefan	YYYY
3	Peter	ZZZZ

Die Anfrage „Verbinde alle Zeilen in Phones, Ages und Addresses, wo Phones.ID = Ages.ID und Phones.ID = Adresses.ID “ ist ein Verbund über 3 Tabellen, der in zwei Verbunde über 2 Tabellen zerlegt werden kann:
. „Verbinde alle Zeilen in Phones und Ages, wo Phones.ID = Ages.ID und benenne das Ergebnis X“
. „Verbinde alle Zeilen in X und Addresses, wo X.Phones.ID = Addresses.ID “

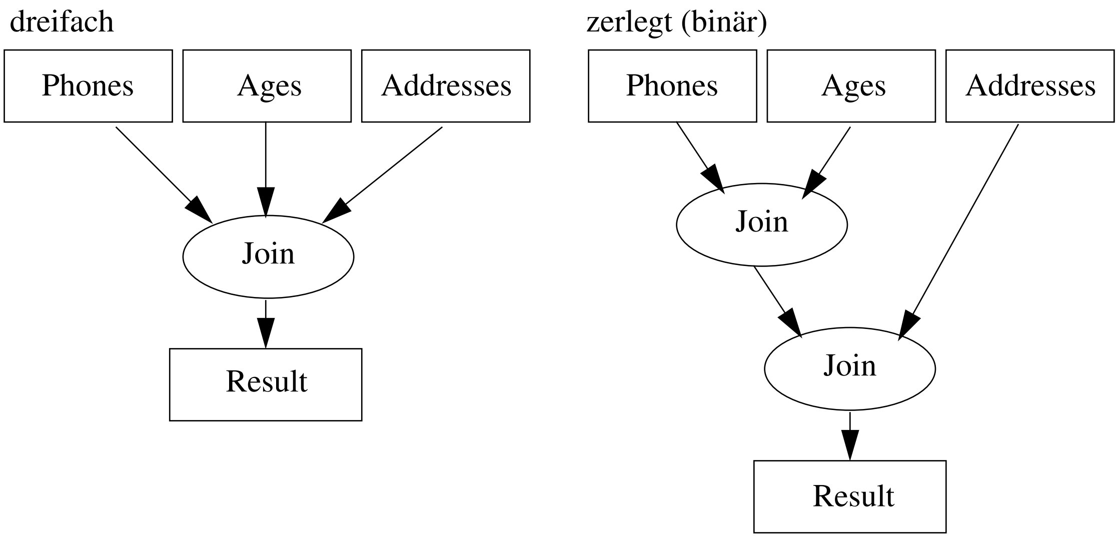


Abbildung 6.5. Join Bäume

6.2.17 Cursorkonzept und Paralellität

Wie schon erwähnt, implementiert jeder Operator die Funktionalität eines Iterators (moveToFirst, moveToNext). Interessant ist aber das bei den verschiedenen Operatoren verschiedene Aktionen dahinter stecken. Der Selektions-, Projektions- und Verbundoperator arbeiten on demand, d.h. die Tupel in ihrer Ergebnismengen werden erst dann erzeugt, wenn der übergeordnete Operator oder der Benutzer sie anfordert. Das erlaubt dem System gleichzeitig Anfragen von vielen Threads zu akzeptieren (sobald sie sich nicht in die Quere kommen) und sie parallel abzuwickeln. Die parallele Abarbeitung kann die mittlere Bearbeitungszeit verkürzen, die Zwischenergebnisse werden auch kleiner.

Leider kann das on demand Mechanismus nicht bei allen Operatoren funktionieren, beim Sort z.B. muß die ganze Eingabemenge zuerst sortiert werden um den ersten Tupel von der Ausgabemenge produzieren zu können. Es ist Aufgabe der Anfrageoptimierung in der logischen DB-Verwaltung einen optimalen Operatorenbaum zu konstruieren, der möglichst hohes Maß an Parallelität bietet..

6.3 Implementierung

6.3.1 Überblick der Klassen

Die ganze JXDB besteht aus 4 getrennten JX-Bibliotheken – db, db_types_impl, db_user und db_operators.

In der Bibliothek db befinden sich alle bisher beschriebenen klassen der Schichten 1 bis 3, sowie alle Klassen der internen DB-Verwaltung (Schicht 4), die direkt von den Benutzerthreads nicht zugreifbar sein sollen, weil sie die Möglichkeit des Benutzers die Datenbank zu verändern nicht einschränken können. Die wichtigste Klasse der internen DB-Verwaltung ist die Klasse db.syscat.SystemCatalogImpl. Sie wird zur Zeit direkt vom Benutzer verwendet, weil es in JXDB noch keine logische DB-Verwaltung gibt. SystemCatalogImpl benutzt die Klassen der Indexhierarchie, die tuplebeschreibenden Klassen sowie die Schnittstelle der physischen DB-Verwaltung um ihre Aufgaben zu realisieren. Die Indexhierarchie besteht aus den Interfaces db.dbindex.DbIndex und db.dbindex.DbIndexSingleDim und die implementierenden Klassen db.dbindex.DbSingleAttrIndex und db.dbindex.DbSingleDimIndex. Die tuplebeschreibende Klassen sind db.syscat.Tuple und db.syscat.TupleDescriptorImpl.

In der Bibliothek db_user kommen alle Interfaces, die von den Benutzerthreads direkt ansprechbar sind. Sie bilden eine Schnittstelle, die einen effizienten und sicheren Zugriff auf die Datenbank gestatten. Das sind die Interfaces jx.db.Database, jx.db.Table, jx.db.Index, jx.db.IndexInfo, jx.db.RelationScan, jx.db.RelationalOperator, jx.db.TupleDescriptor, jx.db.TupleReader, jx.db.TupleWriter, jx.db.Key, jx.db.CodedException, jx.db.SetNumberReader, jx.db.types.TypeManager, jx.db.types.DbConverter, und jx.db.types.DbComparator.

Die Klassen, die die Datentypen der JXDB implementieren sind in die db_types_impl Bibliothek abgetrennt. Das sind jx.db.mytypes.TypeManagerImpl und alle Klassen in den Packages jx.db.mytypes.converters und jx.db.mytypes.comparators.

Die relationalen Operatoren, welche einen Teil der relationalen Algebra realisieren, befinden sich in der db_operators Bibliothek. Das Basisinterface für alle Operatoren ist jx.db.RelationalOperator. Die implementierenden Klassen sind jx.db.RelationScan(in db_user Bibliothek!), db.operators.RelationSelect, db.operators.RelationSort und db.operators.RelationProject.

Der einzige relationale Operator, der den SystemCatalogImpl benutzt, ist RelationScan. Alle andere basieren auf diesen Operator. Die Klasse db.operators.SelectCondition stellt die Selektionsbedingung für den RelationSelect dar.

6.3.2Indexhieachie

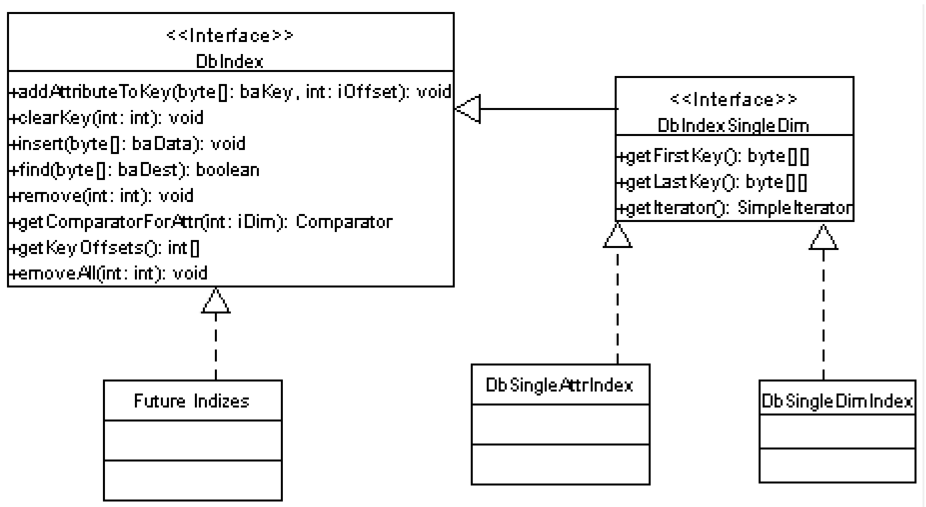


Abbildung 6.6. Indexhierarchie

6.3.3 DbIndex

DbIdex ist das Basisinterface für alle Indexklassen. Es bietet Methoden zum Suchen, Einfügen und Löschen von <Schlüssel, Daten> Paare in den Index. Da der Schlüssel zusammengesetzt sein kann, gibt die Methode addAttributeToKey die Möglichkeit vor der eigentlichen Indexoperation den Schlüssel aufzubauen.

Man kann alle Daten in dem Index mit der Methode removeAll auf einmal löschen. Informationen über den Schlüssel sind mittels getKeyOffsets und getComparatorForAttr zu bekommen (genauere Spezifikation befindet sich in der Dokumentation im Sourcecode).

6.3.4 DbIndexSingleDim

DbIndexSingleDim beinhaltet Funktionalität, die für die eindimensionalen Indizes, die auf B* Baum basieren, spezifisch ist. Bei solchen Indizes besitzen die Schlüssel eine totale Ordnung über welche ein Iterator definiert ist. Bei mehrdimensionalen Indizes muß das nicht unbedingt der Fall sein (2D Punkt 1,5 ist normalerweise mit dem 2D Punkt 3, 2 nicht vergleichbar). Deswegen befinden sich die Funktionen getFirstKey, getLastKey, getIterator hier und nicht in DbIndex.

6.3.5 DbSingleAttrIndex

Die Klasse DbSingleAttrIndex implementiert das Interface DbIndexSingleDim und bietet die Funktionalität eines eindimensionalen Index über ein Attribut. Sie benutzt B* Baum um die Daten abzuspeichern und ist sowohl für unique als auch für non unique Indizes geeignet. Ein gültiges Objekt von dieser Klasse kann nur über der getIndex Methode des SystemCatalogs bekommen werden.

6.3.6 DbSingleDimIndex

Die Klasse DbSingleDimIndex bietet die Funktionalität eines eindimensionalen Index über mehreren Attributen. Wie bei den Konzepten schon erwähnt, sind die Schlüssel eines solchen Index aus mehreren Attributwerten zusammengesetzt. Die Klasse benutzt B*

Baum um die Daten abzuspeichern und ist sowohl für unique als auch für non unique Indizes geeignet. Ein gültiges Objekt von dieser Klasse kann nur über der getIndex Methode des SystemCatalogs bekommen werden.

6.3.7 Tuple

Die Klasse Tuple stellt eine Zeile einer Tabelle dar. Jede interne Tabelle ist für den Benutzer eine Menge von Tupel. Die Klasse bietet Methoden zum Lesen und Schreiben einzelner Attributwerte innerhalb der Zeile. Jedes Objekt der Klasse besitzt eine Referenz auf ein TupleDescriptor Objekt, das Informationen über die Zeilenstruktur beinhaltet.

Jeder relationale Operator liefert als Ausgabe eine Menge von Tuple (genauer TupleReader, siehe unten) Objekte zurück. Der Benutzer kann durch diese Menge mittels der Iteratorfunktionalität des Operators navigieren.

6.3.8 TupleDescriptorImpl

Die Klasse TupleDeskriptorImpl beschreibt eine Tabellenzeile. Sie beinhaltet Informationen über die Länge, die Position innerhalb der Zeile und das Typ der Attribute. Die Klasse Tuple benutzt diese Information um richtig auf die Daten zugreifen zu können.

6.3.9 Hierarchie der relationalen Operatoren

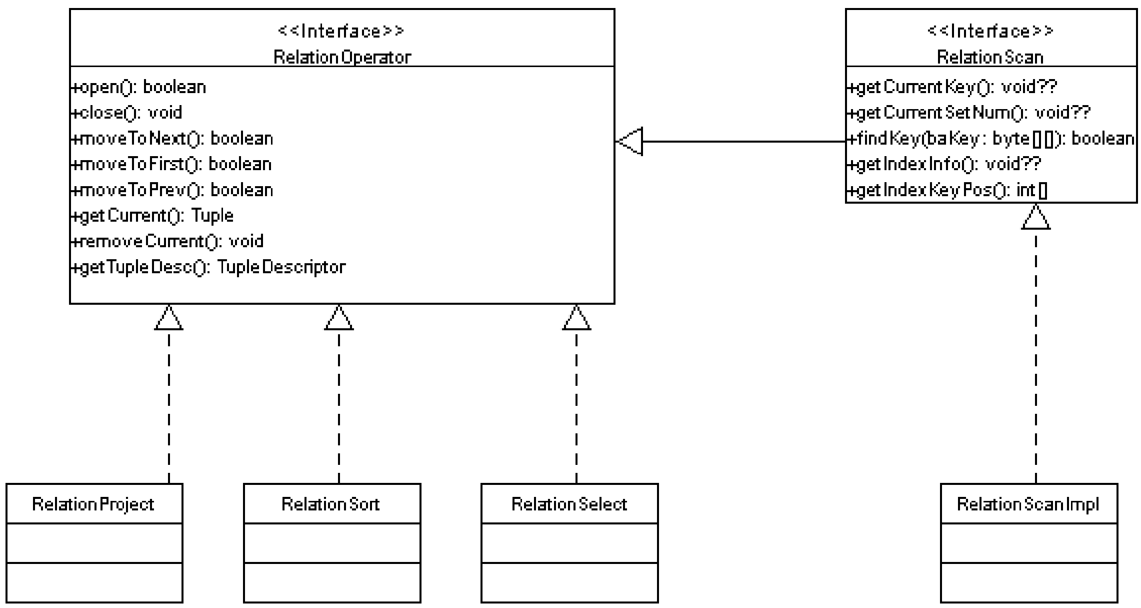


Abbildung 6.7. Hierarchie der relationalen Operatoren

Ein typisches Verwendungszyklus von den relationalen Operatoren sieht wie folgt aus:

1. Der Benutzer erzeugt und inizialisiert die Operator Objekte. Es werden Operatorenbäume gebildet.
2. Der Benutzer ruft die open() Methode des Wurzels auf. Die open() Methoden von allen Operatoren rekursiv aufgerufen.
3. Der Benutzer ruft die Methoden moveToFirst, moveToNext, und moveToPrev auf, um durch die Tabelle zu navigieren , getCurrent, getCurrentSet-

Num und getCurrentKey um auf die Daten zuzugreifen.

4. Der Benutzer ruft beim Wurzel die Methode close() auf, um alle gefixte Seiten freizugeben. Die close() Methoden werden bei allen Operatoren rekursiv aufgerufen.

6.3.10 RelationScanImpl

Die Klasse RelationScanImpl bietet den Benutzern (andere relationale Operatoren oder Endbenutzer) iterativen Zugriff auf ganze interne Tabellen über einen eindimensionalen Index. Die Ausgabemenge kann zusätzlich durch Start- und Endbedingungen begrenzt werden.

Wie im obigen Bild zu sehen ist, unterscheidet sich die Klasse RelationScanImpl von den anderen relationalen Operatoren. Erstens ist das der einzige Operator, der zu den Methoden vom SystemCatalog Zugriff hat. Alle andere Operatoren benutzen RelationScans um auf interne Tabellen zuzugreifen. Zweitens besitzt RelationScanImpl zusätzliche Funktionalität, die in das Interface RelationScan zusammengefaßt ist. Diese Funktionalität kann für performante Scans benutzt werden. Das Lesen von einem ganzen Tupel ist z.B. viel ineffizienter als das Lesen von dem Schlüssel des Tupels im Index, und in vielen Fällen reichen die Schlüssel aus. Deswegen bietet RelationScan die Funktion getCurrentKey, die von der Klasse RelationSelect genutzt wird.

Der Benutzer kann den Index des RelationScans benutzen, um an beliebigen Stellen in der Ausgabemenge springen zu können (findKey).

Während ein Benutzer mit dem RelationScan Operator durch den Tupeln einer Tabelle navigiert, dürfen keine Tupel der Tabelle und insbesondere keine Indizes modifiziert werden. Um ein Update durchzuführen, braucht man ähnliche Funktionalität. RelationScan bietet die Methode getCurrentSetNum an, die den SetNumber des aktuellen Tupels zurückgibt. So können die Nummer aller Tupel, die upgedated werden sollen, gesammelt und zwischengespeichert werden. Anschliesend kann der Operator geschlossen und die Modifikation der Daten ausgeführt werden.

6.3.10 RelationProject

Die Klasse RelationProject dient dazu, Spalten in den Tupel einer Tupelmenge auszublenden. Der Benutzer gibt bei dem Erzeugen eines RelationProject Objekts an, welche Spalten beibehalten werden sollen. Es wird einen entsprechenden TupleDeskriptor generiert, der die neuentstandenen Tupel beschreibt und von der getTupleDesc() Methode zu bekommen ist.

Die Tupelmenge muß die Ausgabemenge eines anderen Operators sein. Die möglichen Duplikaten werden nicht eliminiert, das muß explizit mittels RelationSort Operator gemacht werden.

6.3.11 RelationSelect

Die Klasse RelationSelect dient dazu, Zeilen einer Tabelle auszublenden. Der Benutzer soll eine Bedingung angeben, welche die Eingabetupel für die Ausgabemenge des Operators qualifiziert. Die Klasse SelectCondition repräsentiert diese Bedingung.

Wie schon erwähnt, bietet das RelationScan Möglichkeit effiziente Selects zu realisieren. RelationSelect kann diese Möglichkeit automatisch ausnutzen, wenn die an der SelectCondition beteiligten Attribute teile des Schlüssels des Index sind, der in dem relationScan verwendet wird. Der RelationSelect Operator kann mit Hilfe von der Methode SelectCondition.getAttributeMap abfragen, welche Attribute in der Bedingung berücksichtigt werden. Andererseits liefert die Methode RelationScan.getIndexInfo alle nötige Informationen über den Schlüssel des Index. So kann der RelationSelect Operator entscheiden, ob eine effiziente Variante möglich ist.

6.3.12 RelationSort

Die Klasse RelationSort sortiert die Eingabemenge nach einer angegebenen Attributmenge. Es benutzt ein B* Baum für diese Aufgabe und kann Duplikate eliminieren, falls diese Option im Konstruktor angegeben ist.

6.3.13 Datentypen und die db_types_impl Bibliothek

Momentan sind in JXDB Integers und Strings als Datentypen vorhanden. Für eine komplizierte Anwendung ist das selbstverständlich nicht genügend, deswegen ist die komplette Datentypfunktionalität in einer eigenen Bibliothek abgetrennt, um erweitert werden zu können, ohne den Rest des Systems zu ändern.

Jedes Objekt von einem JXDB-Datentyp muß die folgenden Eigenschaften besitzen:

1. Es muß in einen byte array konvertiert werden können und umgekehrt. Dazu muß der Applikationsentwickler eine Konverterklasse (abgeleitet von jx.db.types.DbConverter) schreiben und sie in den Package jx.db.mytypes.converters plazieren. Die Klasse muß die convert und die revert Methoden implementieren. Convert konvertiert das Objekt in einen byte array, revert dagegen erzeugt ein Objekt aus einem byte array.
2. Es muß mit anderen Objekten des gleichen Datentyps verglichen werden können. Dazu muß der Applikationsentwickler ein Komparatorobjekt (abgeleitet von jx.db.types.DbComparator) schreiben und es in den jx.db.mytypes.comparators Package plazieren. Diese klasse muß dir compare Methode implementieren, die zwei byte array Repräsentationen von zwei Objekte des Datentyps vergleicht. Die compare Methode liefert –1, falls das erste Objekt kleiner als das zweite ist, 0, falls die Beiden gleich sind und 1 sonst.

Um ein neues Datentyp dem Rest des Systems bekanntzumachen muß der Entwickler die Klasse jx.db.mytypes.TypeManagerImpl modifizieren. Sie liefert Informationen über alle zur Zeit im System vorhandenen Datentypen.

6.3.14 SystemCatalogImpl

Die Klasse SystemCatalogImpl stellt Methoden zum Erzeugen und Verwalten von Tabellen, Indizes, Tupel und temporäre Listen. Sie ist der Kern der internen DB-Verwaltung. Fast ohne Ausnahmen bilden alle Interfaces in 6.3.15 auf diese Klasse ab.

6.3.15 Die db_user Bibliothek

Hier sind die Interfaces zur „Außenwelt", in einer JXDB-Aplikation sollten nur diese Interfaces dem Benutzer zugänglich sein. Sie sind so konzipiert, das die Realisierung eines Sicherheitkonzepts möglich ist. Es ist möglich das manche Benutzerthreads Schreib- und Lesezugriff, manche nur Lesezugriff und manche keinen Zugriff auf Tabellen der Datenbank bekommen.

Das jx.db.Database Interface stellt Methoden zum erzeugen, löschen und finden einer Tabelle zur Verfügung. Noch kann der Benutzer einen RelationScan Objekt über eine Tabelle anfordern, oder auf den TypeManager des Datenbanks zugreifen. Dieses Interface wird vom SystemCatalogImpl implementiert.

Das jx.db.Table Interface stellt Methoden zum Erzeugen, Modifizieren und Löschen von Tupel in einer Tabelle. Es können auch Indizes in der Tabelle erzeugt, können aber nur über das jx.db.Index Interface gelöscht werden. Die implementierende Klasse ist db.syscat.TableImpl. Sie macht nichts Weiteres, als die Methodenaufrufe an das System-

CatalogImpl zu delegieren.

Das jx.db.Index Interface bietet Methoden zum Verwalten von Indizes an. Man kann den Index von der Datenbank löschen und Informationen über den Index bekommen (Anzahl und Positionen der Attribute, Typ des Index, unique oder nicht, etc., siehe IndexInfo). Die implementierende Klasse ist db.syscat.IndexImpl. Sie macht nichts Weiteres, als die Methodenaufrufe an das SystemCatalogImpl zu delegieren.

Das jx.db.IndexInfo Interface kapselt Informationen über ein Index. (Welche Attribute der Tabelle sind in dem Index beteiligt, von welchem Typ ist der Index, etc.) Ein IndexInfo Objekt wird als Rückgabewert der Index.getIndexInfo Methode bekommen. Die implementierende Klasse ist db.syscat.IndexInfoImpl.

Das jx.db.TupleReader Interface erlaubt lesender Zugriff auf die Daten eines Tupels. Der Benutzer kann einzelne Felder des Tupels über die Methode getField lesen. Ein TupleReader Objekt wird entweder von dem Table Interface (readTuple), oder vom RelationScan (getCurrent) bereitgestellt. Wichtig ist es, daß der Benutzer das Tupel nicht modifizieren kann, sogar durch einem cast zu der implemetierenden Klasse db.syscat.Tuple$TupleReaderImpl. Das wird von den JX Domains sichergestellt.

Das jx.db.TupleWriter Interface dagegen kann ein Tuple modifizieren. (setField Methode). Ein TupleWriter Objekt kann man nur über die Table.modifyTuple oder Table.createTuple Methoden bekommen. Die relationalen Operatoren dürfen keine TupleWriter erzeugen, weil die Modifikation eines Indexes während der Suche zu unkonsistente Ergebnisse führen kann (siehe Konzepte und Techniken oben.).

Wichtig für TupleWriter und TupleReader ist, das nach dem Gebrauch immer die close Methode aufgerufen werden muß. Ansonsten verbleiben Inkonsistenzen in der Datenbank.

Das jx.db.TupleDescriptor ist das Interface der TupleDescriptorImpl Klasse (siehe 6.3.8)

Das jx.db.Key Interface wird benötigt, um bequem auf die Attribute eines Indexschlüssels zuzugreifen. Der Schlüssel kann aus mehreren Attributen bestehen, und das Key Interface bietet Methoden zum Lesen und Schreiben von einzelnen Attributen an. Die implementierende Klasse ist db.syscat.KeyImpl.

Das jx.db.SetNumberReader Interface ist dem TupleReader ähnlich, bezieht sich aber auf die SetNumber Klasse. Der Benutzer kann dieses Interface z.B. benutzen, um die Adressen mehrerer Tupel abzuspeichern (bei Updates ist das nötig).

Das Interface jx.db.BufferList dient zum effizienten Abspeichern von großen Mengen von gleich langen byte arrays, kleiner als eine Pufferseite (z.B. SetNumbers). Es bietet Methoden zum Iterieren in der abgespeicherten Menge. Ein BufferList Objekt kann über die Database.getTmpStorage Methode bekommen werden.

Die CodedException Klasse ist eine veralgemeinerte Exception, die von allen JXDB Klassen geworfen werden kann. Sie enthält eine Referenz auf das werfende Objekt, den errror code und einen String, der den Fehler bescreibt.

Alle Interfaces in der db_user Bibliothek (Ausnahme ist nur jx.db.Key) erben von dem jx.zero.Portal Interface. So ist es möglich, daß das SystemCatalogImpl Objekt in einem JX-Domain läuft und die userthreads in einem anderen und die ganze Kommunikation nur über JX-Portale stattfindet. Es können selektiv Rechte für die Benutzer an den Portalen ein- und ausgeschaltet werden.

6.4 Zusammenfassung

In diesem Kapitel wurden die Konzepte und Techniken der Internen DB-Verwaltung vorgestellt, sowie die konkrete Implementierung. Da die Interne DB-Verwaltung momentan die letzte Schicht der JXDB ist, wurde auch die Benutzerschnittstelle erläutert. Weitere

Entwicklungen können weiterer Schichten einführen (logische DB-Verwaltung), und damit eventuell auch kleine Änderungen in der Schnittstelle.

Kapitel 7

Benchmarks

7.1 Konzept

Um die Leistung der JXDB abzuschätzen sind Vergleiche mit existierenden Datenbanken notwendig. Die besten Kandidaten sind natürlich Datenbanken, die ähnliche Funktionalität wie JXDB besitzen. Hier wird ein Vergleich mit MySQL vorgestellt. MySQL ist ein freies Datenbanksystem das hauptsächlich auf LINUX Systeme verwendet wird, hat aber auch Portierungen für MS Windows.

Es gibt viele Benchmarks für MySQL, die mit der Standarddistribution mitgeliefert werden. Es genügt also äquivalente Benchmarkprogramme für JXDB zu entwickeln und die Ergebnisse zu vergleichen. Es ist aber sehr wichtig, daß die Tests die beiden Datenbanken möglichst gleichberechtigt behandeln.Daher ist Folgendes zu beachten:

1. JXDB hat nur 4 von den 5 Schichten in der 5 Schichtenarchitektur. SQL Anweisungen müssen manuell in Bäumen von relationalen Operatoren übersetzt werden. MySQL macht das dagegen automatisch und investiert relativ viel Zeit in der Anfrageoptimierung, besonders bei komplexeren SQL Befehlen. Deswegen bestehen alle der hier vorgestellten Tests aus wenigen relationalen Operatoren, die aber viele einfache Datenbankoperationen durchführen.

2. MySQL ist ein Client-Server System, der Client und der Server kommunizieren über eine TCP Verbindung. Die Transferzeiten sind bei großen Datenmengen, die vom Server zum Client fließen, nicht zu vernachlässigen. Bei JXDB bestehen keine TCP Verbindungen. Um fair zu sein dürfen also nur Tests durchgefürt werden, die wenig Datentransfer verursachen.

Gemäß der obigen Überlegungen wurden die folgenden 3 Tests aus der MySQL-Distribution ausgewählt:

7.1.1 Einfaches Insert

In eine Tabelle werden 10000 Zeilen eingefügt. Jede Zeile besteht aus 4 Integerfelder. Zusätzlich werden drei Indizes erzeugt. Die Operation verläuft komplett im Hauptspeicher bei den beiden Systemen (wie übrigens die andere zwei Tests auch). Vorteilhaft ist, daß fast keine Anfrageoptimierung nötig ist, weil der Operatorenbaum nur aus einem Element besteht. Nachteilig ist aber der große Datentransfer (alle 10000 Zeilen müssen bei MySQL vom Client zum Server transferiert werden).

7.1.2 Einfache Selects

Auf der obigen Tabelle werden einige simple Select-Operatoren durchgeführt. Das ist ein rechenintensiver Test, der vernachlässigbare Anfrageoptimierung bei MySQL verursacht und kein Datentransfer zwischen dem Client und dem Server erzeugt (die Daten werden beim Server vorbereitet, zum Client wird nur einen Cursor geschickt, der wird im Test aber nicht benutzt).Die Select-Operation ist häufig die Wichtigste in einem Datenbanksystem, was diesen Test besonders relevant macht.

7.1.3 Einfaches Update

Zuletzt wird ein Feld in jeder der 10000 Zeilen modifiziert. Dieser Test besitzt die guten Eigenschaften des Selecttests (keine Anfrageoptimierung, rechenintensiv, kein Datentransfer)

7.2 Ergebnisse

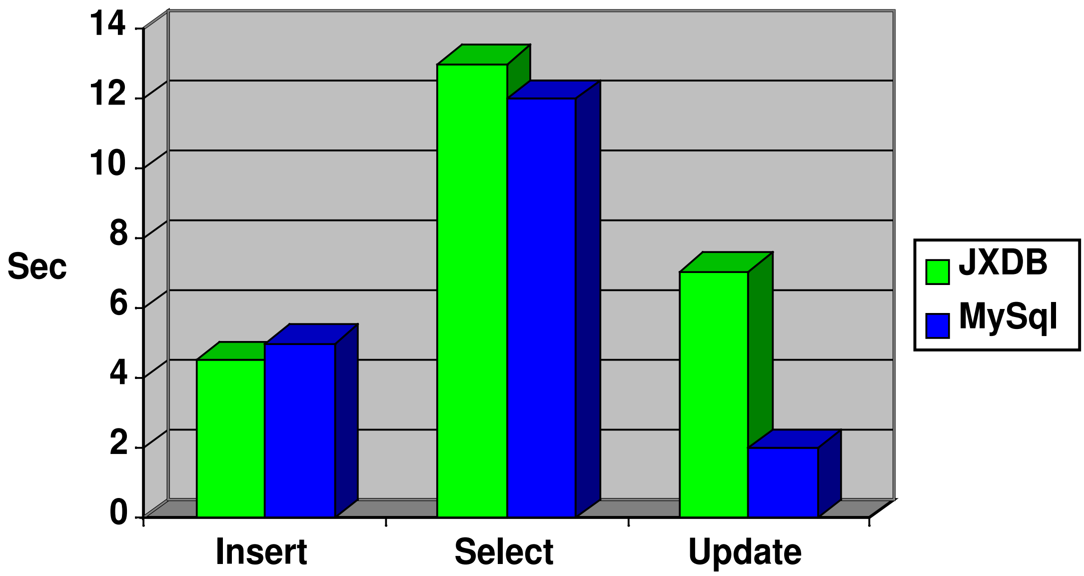


Abbildung 7.1. Benchmark Ergebnisse

Die Ergebnisse beim Update unterscheiden sich beträchtlich, weil JXDB 7 Sekunden für den Wiederaufbau von den Indexstrukturen braucht (0.05 Sekunden ist die reine Updatezeit). Bei MySQL ist das nicht unbedingt der Fall. Bei JXDB könnte der Wiederaufbau der Indizes auf dem Zeitpunkt verschoben werden, wo sie wirklich gebraucht werden.

Zusammenfassung und Ausblick

In den vorigen 7 Kapitel wurden die wichtigsten Eigenschaften des JXDB Datenbanksystems und die Ergebnisse von einigen Leistungsmessungen vorgestellt. Die Studienarbeit zeigt, daß es möglich ist, komplexe und rechenintensive Systeme in die Sprache Java zu implementieren, ohne zu große Performanznachteile. Es hat sich erwiesen, daß die richtige Konzipierung des Systems wichtiger für die Leistung ist, als die Wahl einer konkreten Programmiersprache. Für eine Datenbank ist aber nicht nur die Performanz von Bedeutung, sondern auch die Sicherheit und die Verläßlichkeit, die von der Sprache Java und dem JX-Betriebsystem unterstützt werden.

JXDB bietet zu dieser Zeit nur Grundfunktionalität an, sie kann aber zu einem richtigen Datenbanksystem erweitert werden. Dazu muß die fünfte Schicht der 5-Schichtenarchitektur implementiert werden. Für den realen Betrieb, z.B. als Verwaltungssystem der Administrationsdaten innerhalb von JX, sollte in JXDB auch Transaktionsverwaltung eingebaut werden.

Es sind noch viele Performanzoptimierungen möglich, einige davon:

1) Automatische Clusterung - Seiten könnten Segmenten(Partitionen) zugeordnet werden und die Segmente könnten dynamisch defragmentiert werden

2) Automatische B*Baum Optimierung - ohne Optimierung ist im Durchschnitt 1/3 des einem B*Baum zugeordneten Speicherplatzes leer, Optimierungsalgorithmen können die B*Baum Daten dynamisch “verdichten“ und so die Anzahl der Plattenoperationen vermindern.

3) RAM-Komprimierung - schnelle Komprimierungsalgorithmen können die Daten im Hauptspeicher verdichten und so die Anzahl der Plattenoperationen vermindern. Das ist bei einem Datenbank relevant, da die Plattenzugriffe ungefähr 1000 mal langsamer sind als die RAM-Zugriffe.

Anhang A

B* Baum

Der B*Baum ist eine in der JXDB sehr häufig benutzte Datenstruktur, die auch noch dazu sehr komplex ist. Es lohnt sich daher sie in anderen Modulen von JX wiederzuverwenden. Dafür muß man den Package db.bxtree importieren. Die komplette B*Baum Funktionalität befindet sich entsprechend in der Klasse db.bxtree.BxTree. Zur Zeit muß man die ganze db-Bibliothek importieren, da die BxTree Klasse Klassen von anderen Packages benutzt. Die Trennung in eine eigene Bibliothek ist aber mit wenig Aufwand möglich.

Ein B*-Baum mit Parameter k wird charakterisiert durch folgende Eigenschaften:

- Jeder Weg von der Wurzel zu einem Blatt hat dieselbe Länge.
- Jeder Knoten außer der Wurzel und den Blättern hat mindestens k Nachfolger.
- Jeder Knoten hat höchstens $2 \cdot k$ Nachfolger.
- Die Wurzel hat keinen oder mindestens 2 Nachfolger.

Der Baum T befindet sich im Hintergrundspeicher, und zwar nimmt jeder Knoten einen Block ein. Ein Knoten mit j Nachfolgern speichert j Paare von Schlüsseln und Adressen $(s_1, a_1),...,(s_j, a_j)$. Es gilt $s_1 <= s_2 <= ... <= s_j$. Eine Adresse in einem Blattknoten bezeichnet den Datenblock mit den restlichen Informationen zum zugehörigen Schlüssel, sonst bezeichnet sie den Block zu einem Baumknoten: Enthalte der Block für Knoten p die Einträge $(s_1, a_1),...,(s_j, a_j)$. Dann ist der erste Schlüssel im i-ten Sohn von p gleich s_i, alle weiteren Schlüssel in diesem Sohn (sofern vorhanden) sind größer als s_i und kleiner als s_{i+1}.

Man betrachte nur die Operationen auf den Knoten des Baumes und nicht auf den eigentlichen Datenblöcken. Gegeben sei der Schlüssel s.

LOOKUP:

Beginnend bei der Wurzel steigt man den Baum hinab in Richtung des Blattes, welches den Schlüssel s enthalten müßte. Hierzu wird bei einem Knoten mit Schlüsseln $s_1, s_2,..., s_j$ als nächstes der i-te Sohn besucht, wenn gilt $s_i <= s < s_{i+1}$.

MODIFY:

Wenn das Schlüsselfeld verändert wird, muß ein DELETE mit nachfolgendem INSERT erfolgen. Wenn das Schlüsselfeld nicht verändert wird, kann der Datensatz nach einem LOOKUP überschrieben werden.

INSERT:

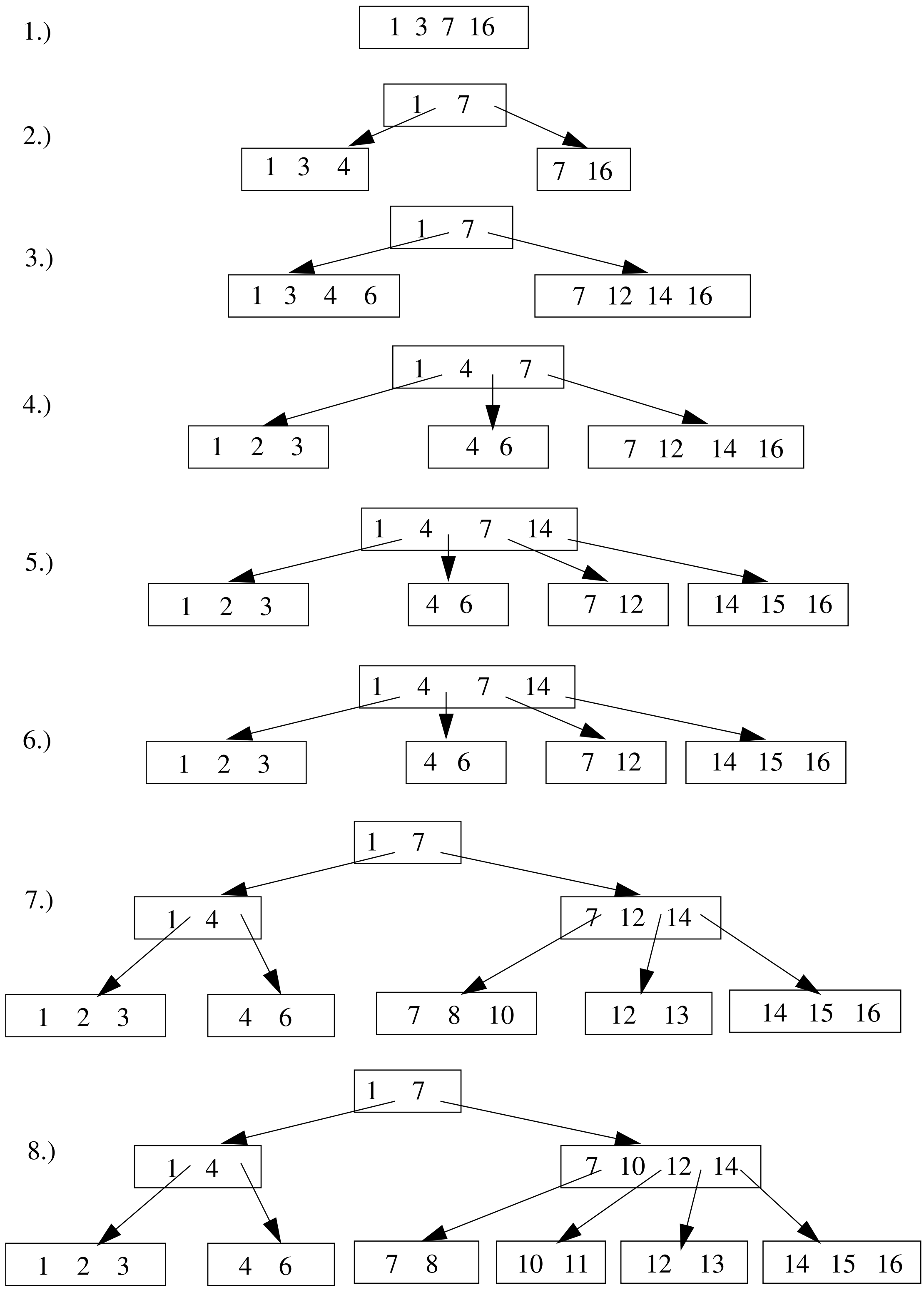
Nach LOOKUP sei Blatt B gefunden, welches den Schlüssel s enthalten soll. Wenn B weniger als $2k$ Einträge hat, so wird s eingefügt, und es werden die Vorgängerknoten berichtigt, sofern s kleinster Schlüssel im Baum ist. Wenn B $2 \cdot k$ Einträge hat, wird ein neues Blatt B' generiert, mit den größeren k Einträgen von B gefüllt und dann der Schlüssel s eingetragen. Der Vorgänger von B und B' wird um einen weiteren Schlüssel s' (kleinster Eintrag in B') erweitert. Falls dabei Überlauf eintritt, pflanzt sich dieser nach oben fort.

DELETE:

Nach LOOKUP sei Blatt B gefunden, welches den Schlüssel s enthält. Das Paar (s, a) wird entfernt und ggf. der Schlüsseleintrag der Vorgänger korrigiert. Falls B jetzt k - 1 Einträge hat, wird der unmittelbare Bruder B' mit den meisten Einträgen bestimmt. Haben beide Brüder gleich viel Einträge, so wird der linke genommen.

Hat B' mehr als k Einträge, so werden die Einträge von B und B' auf diese beiden Knoten gleichmäßig verteilt. Haben B und B' zusammen eine ungerade Anzahl, so erhält der linke einen Eintrag mehr. Hat B' genau k Einträge, so werden B und B' verschmolzen. Die Vorgängerknoten müssen korrigiert werden.

Abbildung A.1 zeigt das dynamische Verhalten eines B*-Baums mit dem Parameter $k = 2$. Es werden nacheinander die Schlüssel `3, 7, 1, 16, 4, 14, 12, 6, 2, 15, 13, 8,` 10, 5, 11, 9 eingefügt und insgesamt 8 Schnappschüsse gezeichnet.



Abblidung A.1 dynamisches Verhalten eines B*-Baums

Der Parameter *k* ergibt sich aus dem Platzbedarf für die Schlüssel/Adreßpaare und der Blockgröße. Die Höhe des Baumes ergibt sich aus der benötigten Anzahl von Verzweigungen, um in den Blättern genügend Zeiger auf die Datenblöcke zu haben. Die Klasse sb.bxtree.BxTree realisiert die Eigenschaften, die in der obigen allgemeinen Beschreibung erwähnt wurden. Die Methoden find, insert und remove stehen entsprechend für LOOKUP, INSERT und DELETE. MODIFY kann mit remove und insert realisiert werden. Die Methode getIterator liefert ein Iterator Objekt zurück, mit dem man durch die sortierte Schlüssel iterieren kann. Die Methode removeAll löscht den ganzen Baum schlagartig(viel schneller als für jeden Schlüssel remove aufzurufen). Wenn der Speicher zu Ende ist, wirft die insert Methode ein Exception, ohne den B*Baum inkonsistent zu machen, obwohl vor der insert Operation der Speicherbedarf nicht bekannt ist..

Anhang B

Beispiele und Benchmarks

In diesem Kapitel wird ein kleines Benchmark erläutert, der die Leistung der JXDB mit der Leistung der MySQL Datenbank vergleicht. Dieses Benchmark dient auch als Beispiel für die meisten Operationen, die zur Zeit in der JXDB ausgeführt werden können. Der Code vom Benchmark befindet sich in der Klasse test.Benchmark in der test_db Bibliothek.

B.2 Inizialisierung

Um auf die JXDB Funktionen Zugriff zu bekommen, braucht jeder Benutzerthread eine jx.db.Database Referenz zuerst:

```
public static void dbtest(jx.zero.Naming naming) {
   jx.db.Database db;
   while((db  = (Database)naming.lookup("DB")) == null){
        Thread.yield();
   }
...
```

Die Variable naming wird als Parameter der dbtest Methode vom JX-System übergeben. Sie ist ein Nameserver, wo alle JX-Module ihre öffentliche Dienste anmelden. Die while Schleife wartet ab, bis die JXDB in ihrem Domain gestartet und initialisiert wird. Bis dahin liefert die lookup Methode null zurück.

B.2 Erzeugen von Tabellen

Es wird jetzt eine Tabelle mit vier Attribute erzeugt. Diese Tabelle wird später benutzt um verschiedene Scans durchzuführen. Zuerst werden arrays angelegt, wo die Informationen über die Attribute gespeichert werden:

```
Debug.out.println("Creating table bench");

String[] aszNames = new String[4];
int[]    aiTypes  = new int[4];
int[]    aiSizes  = new int[4];
```

Hier werden die einzelne Felder inizialisiert. Der aiTypes Array wird mit werten aus dem TypeManager gefüllt.

```
Debug.out.println("Attribute 1 is id of int");
aszNames[0] = "id";
aiSizes [0] = 4;
aiTypes [0] = TypeManager.DATP_DID_INT;

Debug.out.println("Attribute 2 is region of int");
aszNames[1] = "region";
aiSizes [1] = 4;
aiTypes [1] = TypeManager.DATP_DID_INT;
```

```
Debug.out.println("Attribute 3 is grp of int");
aszNames[2] = "grp";
aiSizes [2] = 4;
aiTypes [2] = TypeManager.DATP_DID_INT;

Debug.out.println("Attribute 4 is rev_id of int");
aszNames[3] = "rev_id";
aiSizes [3] = 4;
aiTypes [3] = TypeManager.DATP_DID_INT;
```

Das ist die Zeile, welche die Tabelle mit Name „bench“ erzeugt. Es wird eine Table Referenz zurückgeliefert, die ein JX-Portal ist:

```
Table cTable = db.createTable("bench", aszNames, aiSizes,
aiTypes);
```

B.3 Erzeugen von Indizes

Bisher existiert nur ein Eintrag in der JXDB für die Tabelle „bench“. Die Tabelle selber enthält noch keine Daten. Um etwas einfügen zu können, muß in der Tabelle mindestens ein Index definiert sein. Der folgende Codeabschnitt definiert 3 Inizes über veschiedene Attribute:

```
int[] aiImap = new int[2];
Debug.out.println("Creating indexes ...");
Debug.out.println("Create unique index < region, id >");
aiImap[0] = 1;
aiImap[1] = 0;
Index        cIndex1        =        cTable.createIn-
dex(Index.INDX_TYPE_SINGLE_DIM_MULTI_ATTR, true, aiImap);

Debug.out.println("Create  unique  index  <  region,  rev_id
>");
aiImap[0] = 1;
aiImap[1] = 3;
Index        cIndex2        =        cTable.createIn-
dex(Index.INDX_TYPE_SINGLE_DIM_MULTI_ATTR, true, aiImap);

Debug.out.println("Create  unique  index  <  region,  grp,  id
>");
aiImap = new int[3];
aiImap[0] = 1;
aiImap[1] = 2;
aiImap[2] = 0;
Index        cIndex3        =        cTable.createIn-
dex(Index.INDX_TYPE_SINGLE_DIM_MULTI_ATTR, true, aiImap);
```

Der aiImap array (index map) enthält die Positionen der Attribute, über welche der Index definiert ist. Der Index wird jeweils mit dem Aufruf cTable.createIndex erzeugt, die ersten zwei Parameter geben den Typ und ob der Index unique ist an. Es werden also 3 unique Indizes vom typ `INDX_TYPE_SINGLE_DIM_MULTI_ATTR` erzeugt und in der JXDB angemeldet. Ab jetzt sorgt die JXDB automatisch für ihre Konsistenz. Indizes sind immer mit einer einzigen Tabelle verbunden, deswegen können sie nur über ein konkretes Table Objekt erzeugt werden.

B.4 Einfügen in der Tabelle

Nachdem die Indizes erzeugt sind kann die Tabelle mit Daten gefüllt werden. In der folgenden Schleife werden 10000 Tupel initialisiert und der Tabelle hinzugefügt.

```
for (int iCnter = 0; iCnter < 10000; iCnter++) {
TupleWriter cTupleWriter = cTable.createTuple();
cTupleWriter.setField(0, iCnter);
cTupleWriter.setField(1, 65 + iCnter % 6);
cTupleWriter.setField(2, (iCnter * 3) % 100);
cTupleWriter.setField(3, 9999 - iCnter);
//free resources!
cTupleWriter.close();
}
```

Mit dem Aufruf cTable.createTuple() wird ein leeres Tupleobjekt in dem JXDB erzeugt und eine TupleWriter Referenz dem Benutzer zurückgeliefert. Der Benutzer modifiziert die Attribute der Tuple mittels TupleWriter.setField(...) Methoden. Am Ende der Modifikation muß die Methode TupleWriter.close() aufgerufen werden, um der JXDB mitzuteilen, daß die Indizes jetzt aktualisiert werden sollen.

B.5 Scan über ein Index mit Start- und Endbedingung

Zuerst muß die Start- und Endbedingung spezifiziert werden. Das sind zwei jx.db.Key Objekte, die von der Methode `Index.getKey()` erzeugt werden:

```
jx.db.Key cStart = cIndex2.getKey();
jx.db.Key cEnd   = cIndex2.getKey();
```

Hier werden die Key Objekte initialisiert. Die Ausgabemenge soll alle Tupel enthalten, dessen „region“ Attribut gleich 66 ist. Da die Bedingung sich nur auf ein Attribut bezieht, und der Index 2 Attribute hat, muß die Start- und Endbedingung für das zweite Attribut von min bis max sein, also von 0 bis 10000.

```
cStart.setField(66,0);
cStart.setField(0,1);
cEnd.setField(66,0);
cEnd.setField(10000,1);
```

Die folgende Zeile erzeugt ein RelationScan Objekt, ohne Start- und Endbedingungen sollte man null anstelle von cStart und cEnd angeben.

```
cOp = db.getRelationScan(cTable, cIndex2, cStart, cEnd);
```

Die nächste Schleife liest alle Tupel, die sich für den Scan qualifizieren, und druckt ihren Inhalt auf den Debug Ausgabekanal aus:

```
 if (cOp.moveToFirst())
       do {
          TupleReader cTupleReader = cOp.getCurrent();
          cTupleReader.dump();
          cTupleReader.close();
       } while (cOp.moveToNext());
```

```
cOp.close();
```

Die RelationScan.getCurrent() Methode liefert ein TupleReader Objekt zurück, der das aktuelle Tupel referenziert. Nach der Verwendung muß die Methode TupleReader.close() aufgerufen werden, sowie auch die Methode RelationalOperator.close(), um Ressourcen freizugeben. Die Methoden RelationalOperator.moveToFirst() und RelationalOperator.moveToNext() dienen zum Iterieren in der Tupelmenge.

B.6 Unfolständiges IndexScan

Für kompliziertere Selects müsste entweder der RelationSelect Operator verwendet, oder die obige Schleife so modifiziert werden, daß die gewünschten Tupel ausgefiltert werden. Der folgende Codestück findet alle Tupel, die „region" zwischen 66 und 67 haben und „grp" gleich 100. Dieser Selekt kann nicht vollständig mit dem cIndex3 realisiert werden, deswegen wird die Schleife erweitert:

```
//"select idn from bench1 where region>=66 and region<=67
// and grp=100"
cStart.setField(65 + 1,  0);
cEnd.setField(65 + 2,  0);
cOp = db.getRelationScan(cTable, cIndex3, cStart, cEnd);
if (cOp.moveToFirst())
   do {
     Key cKey = cOp.getCurrentKey();
     if (100 == cKey.getField(1)) {
         TupleReader cTupleReader = cOp.getCurrent();
         cTupleReader.dump();
         cTupleReader.close();
     }
   }while (cOp.moveToNext());
cOp.close();
```

cIndex3 sucht alle Tupel raus, die „region" zwischen 66 und 67 haben. Die Zeile

```
if (100 == cKey.getField(1))
```

findet zusätzlich die Tupel, dir "grp" gleich 100 haben. Wichtig ist die Methode RelationScan.getCurrentKey(), die den Indexschlüssel des aktuellen Tupels zurückliefert. Sie ist viel effizienter als getCurrent(), weil getCurrent sie intern aufruft und noch zusätzliche Operationen ausführt.

B.7 Update

Der Update besteht aus einem Select und eine Modifikation der Ergebnismenge. In JXDB können diese beide Operationen nicht gleichzeitig stattfinden. Deswegen müssen die Adressen der Tupel in der Ergebnismenge zwischengespeichert werden. Dazu holt man sich zuerst ein temporäres Puffer von JXDB:

```
BufferList cBufList =
db.getTmpStorage(SetNumber.geByteLen());
```

Der Parameter in der Methode gibt die länge der Objekte an, die in dem BufferList gespeichert werden. Jetzt kann man ein Select specifizieren, in diesem Fall kommen alle Tupel der Tabelle in die Ergebnismenge:

```
cOp = db.getRelationScan(cTable, cIndex3, null, null);

//first fetch all tuples
byte[] baSetNum = new byte[ SetNumber.getByteLen() ];
if (cOp.moveToFirst())
   do {
         SetNumberReader cSetNum = cOp.getCurrentSetNum();
         cSetNum.getSetNumber(baSetNum, 0);
         cBufList.addAtEnd(baSetNum, 0);
   }while (cOp.moveToNext());

cOp.close();
```

Die obige Schleife speichert die SetNumbers aller Tupel der Tabelle in die BufferList. Danach wird des RelationOperator geschlossen und danach kan der eigentliche Update stattfinden:

```
//do the update
cBufList.moveToFirst();
SetNumber cSetNum = new SetNumber();

do {
     baSetNum = cBufList.getCurrent();
     cSetNum.setSetNumber(baSetNum, 0);
     TupleWriter cTupleWriter =
     cTable.modifyTuple(cSetNum.getReader());
     cTupleWriter.setField(0, opt_loop_count +
     cTupleWriter.getReader().getField(0));
     cTupleWriter.close();
}while(cBufList.moveToNext());
```

Nach dem Update werden die zwischengespeicherten SetNumbers nicht mehr gebraucht und werden gelöscht:

```
//delete temporary storage
cBufList.delete();
```

B.8 Der Benchmark

In dem Benchmark wird also zuerst eine Tabelle erzeugt und mit 10000 Tupel gefüllt. Dann werden die folgenden Select-Aufrufe 100 mal ausgeführt:

```
"\nselect idn from bench1 where region=region'";
"select idn from bench1 where region=region and idn=i"
"select idn from bench1 where region=region and rev_idn=i"
"select idn from bench1 where region=region and grp=grp"
"select idn from bench1 where region>=66 and region<=67
and grp=grp"
 "select idn from bench1 where region>=66 and region<=69
and grp=grp"
 "select idn from bench1 where grp=grp"
```

Die Variablen grp, region und rev_idn werden mit schleifenabhängige Werte belegt (sihe Sourcecode). Am ende wird ein Update durchgeführt, das idn Attribut aller Tupel wird um 10000 erhöht. Es werden die Zeiten für das Insert, Select und Update getrennt gemessen.

B.9 Das MySQL Bechmarkskript

Hier folgt der komplette Code des MySQL Benchmarks:

```
#########################################################################
#!/usr/bin/perl
# Copyright (C) 2000 MySQL AB & MySQL Finland AB & TCX DataKonsult AB
#
# This library is free software; you can redistribute it
#and/or modify it under the terms of the GNU Library General Public
# License as published by the Free Software Foundation; either
# version 2 of the License, or (at your option) any later version.
#
# This library is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# Library General Public License for more details.
#
# You should have received a copy of the GNU Library General Public
# License along with this library; if not, write to the Free
# Software Foundation, Inc., 59 Temple Place - Suite 330, Boston,
# MA 02111-1307, USA
#
# Test of selecting on keys that consist of many parts
#
##################### Standard benchmark inits #########################

use DBI;
use Getopt::Long;
use Benchmark;

$opt_loop_count=10000;
$opt_medium_loop_count=1000;
$opt_small_loop_count=100;
$opt_regions=6;
$opt_groups=100;

chomp($pwd = `pwd`); $pwd = "." if ($pwd eq '');
require  "$pwd/bench-init.pl"  ||  die  "Can't  read  Configuration  file:
$!\n";

$columns=min($limits->{'max_columns'},500,($limits->{'query_size'}-50)/
24,
             $limits->{'max_conditions'}/2-3);

if ($opt_small_test)
{
  $opt_loop_count/=10;
  $opt_medium_loop_count/=10;
  $opt_small_loop_count/=10;
  $opt_groups/=10;
}

print  "Testing  the  speed  of  selecting  on  keys  that  consist  of  many
parts\n";
print "The test-table has $opt_loop_count rows and the test is done with
$columns ranges.\n\n";

####
####  Connect and start timeing
####

$dbh = $server->connect();
$start_time=new Benchmark;
```

```
####
#### Create needed tables
####

goto select_test if ($opt_skip_create);

print "Creating table\n";
$dbh->do("drop table bench1" . $server->{'drop_attr'});

do_many($dbh,$server->create("bench1",
                             ["region char(1) NOT NULL",
                              "idn integer(6) NOT NULL",
                              "rev_idn integer(6) NOT NULL",
                              "grp integer(6) NOT NULL"],
                             ["primary key (region,idn)",
                              "unique (region,rev_idn)",
                              "unique (region,grp,idn)"]));
if ($opt_lock_tables)
{
  do_query($dbh,"LOCK TABLES bench1 WRITE");
}

if ($opt_fast && defined($server->{vacuum}))
{
  $server->vacuum(1,\$dbh);
}

####
#### Insert $opt_loop_count records with
#### region:    "A" -> "E"
#### idn:       0 -> count
#### rev_idn:   count -> 0,
#### grp:       distributed values 0 - > count/100
####

print "Inserting $opt_loop_count rows\n";

$loop_time=new Benchmark;
$query="insert into bench1 values (";
$half_done=$opt_loop_count/2;
for   ($id=0,$rev_id=$opt_loop_count-1   ;   $id   <   $opt_loop_count   ;
$id++,$rev_id--)
{
  $grp=$id*3 % $opt_groups;
  $region=chr(65+$id%$opt_regions);
  do_query($dbh,"$query'$region',$id,$rev_id,$grp)");
  if ($id == $half_done)
  {                             # Test with different insert
    $query="insert into bench1 (region,idn,rev_idn,grp) values (";
  }
}

$end_time=new Benchmark;
print "Time to insert ($opt_loop_count): " .
    timestr(timediff($end_time, $loop_time),"all") . "\n\n";

if ($opt_lock_tables)
{
  do_query($dbh,"UNLOCK TABLES");
}

if ($opt_fast && defined($server->{vacuum}))
{
  $server->vacuum(0,\$dbh,"bench1");
}

if ($opt_lock_tables)
```

```
{
  do_query($dbh,"LOCK TABLES bench1 WRITE");
}

####
#### Do some selects on the table
####

select_test:

print "Testing big selects on the table\n";
$loop_time=new Benchmark;
$rows=0;
for ($i=0 ; $i < $opt_small_loop_count ; $i++)
{
  $grp=$i*11 % $opt_groups;
  $region=chr(65+$i%($opt_regions+1));  # One larger to test misses
       $rows+=fetch_all_rows($dbh,"select   idn   from   bench1   where
region='$region'");
       $rows+=fetch_all_rows($dbh,"select   idn   from   bench1   where
region='$region' and idn=$i");
       $rows+=fetch_all_rows($dbh,"select   idn   from   bench1   where
region='$region' and rev_idn=$i");
       $rows+=fetch_all_rows($dbh,"select   idn   from   bench1   where
region='$region' and grp=$grp");
   $rows+=fetch_all_rows($dbh,"select idn from bench1 where region>='B'
and region<='C' and grp=$grp");
   $rows+=fetch_all_rows($dbh,"select idn from bench1 where region>='B'
and region<='E' and grp=$grp");
  $rows+=fetch_all_rows($dbh,"select idn from bench1 where grp=$grp"); #
This is hard
}
$count=$opt_small_loop_count*7;

$end_time=new Benchmark;
print "Time for select_big ($count:$rows): " .
    timestr(timediff($end_time, $loop_time),"all") . "\n";

###
###Testing update
###

$loop_time=new Benchmark;

$sth=$dbh->do("update  bench1  set  idn=$opt_loop_count  +  idn")  or  die
$DBI::errstr;

$end_time=new Benchmark;

print "Time for update_big: " .
    timestr(timediff($end_time, $loop_time),"all") . "\n";
```

Literaturverzeichnis

HR99 Härder, T.;Rahm, E: Datenbanksysteme. Konzepte und Techniken der Implementierung. Springer-Verlag, 1999

DAT95 Date, C.J.: An Introduction to Database Systems, Sixt Edition, Addison-Wesley Publishing Company, New York 1995.

CDHD97 C. J. Date and Hugh Darwen: A Guide to the SQL Standard (4th Edition). Reading, Mass.: Addison-Wesley (1997).

GUW00 Garcia-Molina, H., Ullman, J.D. Widom, J.: Database System Implementation. Prentice Hall, 2000