Mehrdimensionale Zugriffspfade in Relationalen Datenbanksystemen

Theo Härder
Universität Kaiserslautern
FB Informatik, Postfach 3049
6750 Kaiserslautern
Tel.: 0631/205-4030
e-mail: haerder @informatik.uni-kl.de

Überblick

Relationale Datenbanksysteme gewähren ihren Benutzern ausschließlich wertabhängigen Zugriff auf die normalisierten Relationen der Datenbank. Dabei soll auch der Zugriff über mehrere Attribute effizient ausgeführt werden können, was die Bereitstellung von mehrdimensionalen Zugriffspfaden impliziert. Neben der Unterstützung von Anfragetypen wie Punktsuche und Bereichssuche und der Gewährleistung von wichtigen Abbildungseigenschaften wie Erhaltung der Topologie (mehrdimensionale Clusterbildung) und balancierte Zugriffsstruktur auch bei schiefen Werteverteilungen sollen solche Zugriffspfade auch zur Sicherung der Datenintegrität herangezogen werden. Insbesondere betrifft das die Erhaltung von Entitätsintegrität, von Referentieller Integrität und von UNIQUE-Optionen, bei denen jeweils $k \geq 2$ Attribute involviert sind. Ein wichtiger Aspekt unserer Untersuchungen ist auch das Leistungsverhalten der betrachteten Strukturen im Mehrbenutzerbetrieb unter Einhaltung des Transaktionsparadigmas, wozu es bisher noch keine Aussagen aus der Literatur gab.

In unserem Aufsatz betrachten wir zunächst den Fall, daß keine k-dimensionalen Zugriffspfade verfügbar sind und die entsprechenden mehrdimensionalen Zugriffe durch eindimensionale Indexstrukturen (B*-Bäume) zu simulieren sind. Eine Analyse der bedeutendsten mehrdimensionalen Zugriffspfade führt auf das Grid File als einer Struktur mit “offensichtlich” guten operationalen Eigenschaften und breiten Einsatzmöglichkeiten in DBS. Eine detailliertere Betrachtung dieser Strukturen sowie die Entwicklung von Synchronisationsprotokollen fördern jedoch einige Probleme zu Tage. Zwar können Grid Files auch zur effizienten Überprüfung der Relationalen Invarianten herangezogen werden, die bisher gefundenen Synchronisationsprotokolle scheinen jedoch bei großen Transaktionslasten eher die Eskalation von Konflikt- und Blockierungssituationen zu begünstigen und damit zu gewissen Leistungsproblemen zu führen.

1. Einführung

In Relationalen Datenbanksystemen kann der Benutzer ausschließlich über die Werte der verschiedenen Attribute der ihm zugänglichen Relationen auf deren Inhalt zugreifen. Dabei werden häufig mehrere Attribute zur Qualifikation herangezogen, so daß eine Mehrattributsuche (mehrdimensionale Suche) bei der Bereitstellung des Ergebnisses sehr vorteilhaft ist. Sogenannte mehrdimensionale Zugriffspfade sollen diese Art der Suche (Punktsuche, Bereichssuche, Best-Match-Suche) besonders effizient unterstützen. Die Anfrageoptimierung wird sie für alle Anfragen auswählen, für die sie in der konkreten Situation einen Leistungsgewinn verspricht. Die praktische Nützlichkeit solcher Strukturen wird aber auch durch ihre Unterstützung anderer Aufgaben wie z.B. der Integritätssicherung bestimmt. Deshalb muß ihre Tauglichkeit für den DBS-Einsatz von ihrem gesamten Operationsspektrum und von allen ihren Eigenschaften her beurteilt werden.

Im Relationenmodell können Schlüsselkandidaten aus einem Attribut oder einer minimalen Gruppe von Attributen bestehen. Sie werden nach dem SQL2-Standardisierungsvorschlag durch die UNIQUE-Option definiert. Für den ausgewählten Primärschlüssel (PRIMARY KEY) gilt automatisch die Attributklausel NOT NULL [SQL2, SQL3]. Wenn Schlüsselkandidaten aus mehreren Attributen bestehen, dann sind die zugehörigen Fremdschlüssel ebenso aus den entsprechenden Attributen zusammengesetzt.

Als zentrale und vom System erzwungene Integritätsbedingungen fordert der SQL2-Standard die Einhaltung der Relationalen Invarianten, nämlich der Entitätsintegrität als der Eindeutigkeit und Definiertheit der Primärschlüsselwerte sowie der Referentiellen Integrität als der Einhaltung der zwischen Primärschlüssel (Schlüsselkandidat) und Fremdschlüssel definierten Beziehung [Da81, Sh90].

In [HR91] wurde festgestellt, daß in DB-Anwendungen die Überprüfung der Integritätsforderungen für hinreichend große Relationen aus Leistungsaspekten nicht akzeptabel ist, wenn nicht ausreichende Zugriffspfad-Unterstützung geboten wird. In praktischen Fällen bedeutet das, daß für Primärschlüssel, für Fremdschlüssel und für jedes mit der Option UNIQUE versehene Attribut(-gruppe) jeweils eine Indexstruktur anzulegen und zu warten ist. Beim Einfügen eines neuen Tupels kann durch Aktualisierung der Indexstrukturen mit der Option UNIQUE automatisch die Eindeutigkeitseigenschaft des entsprechenden Attributs überprüft werden. Zur Wartung der Primärschlüssel/Fremdschlüssel-Beziehungen sind Zugriffe und ggf. Modifikationen in zwei Indexstrukturen (UNIQUE und NONUNIQUE) erforderlich. Da alle Lese- und Aktualisierungsoperationen innerhalb von Transaktionen abgewickelt werden, also die ACID-Eigenschaften [HR83] als Zusicherungen besitzen, muß für den Transaktionsbetrieb Konsistenzebene 3 gewährleistet werden. Weiterhin sind auf diesen Indexstrukturen häufig Schreib- und Leseoperationen verschiedener Transaktionen zu erwarten. Aus diesen Gründen kommt der Implementierung und Wartung der Indexstrukturen sowie der effizienten und effektiven Synchronisation der parallelen Transaktionen eine erhebliche Bedeutung für das gesamte DBMS-Verhalten zu.

In [HR91] wurden die Operationen, bei denen die Einhaltung der Referentiellen Integrität zu überprüfen bzw. herzustellen ist, im Detail betrachtet. Für einfache Attribute, was als Normalfall zu erwarten ist, wurden Indexstrukturen und zugehörige Sperrprotokolle entwickelt und untersucht. Diese Analyse zusammen mit einer Leistungsbe-

wertung gestattete die Auswahl der für die Integritätserhaltung am besten geeigneten (eindimensionalen) Indexstrukturen.

Bei zusammengesetzten, d.h. aus mehreren Attributen bestehenden Schlüsseln lassen sich diese Lösungen nicht ohne weiteres übernehmen. Für die Suche bei zusammengesetzten oder mehrdimensionalen Schlüsseln (Mehrattributsuche, mehrdimensionale Suche) wurden bereits viele Strukturen und Verfahren entwickelt [Be75, FB74, Gu84, NHS84, SRF87], die aber bisher noch nicht ihre praktische Tauglichkeit beweisen konnten. Auch wurden diese Verfahren noch nicht unter den Aspekten der Trans-aktionslogik und den Anforderungen des Mehrbenutzerbetriebs untersucht.

2. Operationen auf mehrdimensionalen Zugriffspfaden

Eine Relation ist eine Sammlung von N Sätzen (Tupeln) des Typs R = (A1, ..., An), wobei jeder Satz ein geordnetes n-Tupel t = (a1i, a2j, ..., anm) von Werten ist. Wir gehen davon aus, daß in einer Relation "punktförmige Ojekte" gespeichert werden sollen, d. h., jedes Objekt wird durch einen Satz repräsentiert. Wir schließen also den Fall aus, daß räumliche Objekte [Gu84, Gü89] in R gespeichert werden sollen und daß raumbezogener Zugriff auf diese geometrischen Objekte zu unterstützen ist.

Im allgemeinen Fall läßt sich ein mehrdimensionaler Zugriffspfad für k beliebige Attribute von R auslegen. Bei Aktualisierungsoperationen auf einer solchen Struktur sind dann k Attributwerte, bei Suchvorgängen k oder weniger Werte zu spezifizieren.

2.1 Spezielle Operationen

Als Operationen sollen auf einem k-dimensionalen Zugriffspfad unterstützt werden:

- Insert (a1i, a2j, ..., akn: TID)
  Es wird ein Verweis auf das Tupel mit dem internen Namen TID (Adresse) eingetragen.
- Delete (a1i,a2j, ..., akn: TID)
  Der entsprechende Verweis auf das Tupel TID wird ausgetragen.
- Fetch (A1=a1i AND A2=a2j AND ... AND Ak=akn)
  Das entspricht der Punktsuche (exact match query); alle Dimensionen des Schlüssels sind spezifiziert. Bei erfolgreicher Suche wird bei UNIQUE-Index das entsprechende TID und bei NONUNIQUE-Index eine TID-Liste zurückgeliefert.
- Fetch (Ai_1=ai_1i AND Ai_2=ai_2j AND ... AND Ai_s = ai_sl)
  mit $1 \le s < k$ und $1 \le i_1 < i_2 < ... < i_s \le k$
  Bei diesem Fragetyp werden einige der s < k Dimensionen spezifiziert und zwar mit einer '='-Relation (partial match query). Nach erfolgreicher Suche wird eine TID-Liste zurückgeliefert.

- Fetch ((A1≥ a1i AND A1 ≤ a1k) AND (A2 ≥ a2j AND A2 ≤ a2l) AND ...
  AND (Ak ≥ akn AND Ak ≤ akm))
  Mit diesem Fragetyp kann eine Bereichsfrage {>, ≥, <, ≤} spezifiziert werden (range query). Nach erfolgreicher Suche wird eine TID-Liste zurückgeliefert.
- Fetch (($Ai_1 \geq ai_1i$ AND $Ai_1 \leq ai_1h$) AND ($Ai_2 \geq ai_2j$ AND $Ai_2 \leq ai_2g$) AND ...
  AND ($Ai_s \geq ai_sl$ AND Ais ≤ ai_sf)
  mit $1 \leq s < k$ und $1 \leq i_1 < i_2 < ... < i_s \leq k$
  Wenn für nicht alle Indexdimensionen Bereiche als Suchbedingungen spezifiziert sind, hat man eine partielle Bereichsfrage (partial range query). Ein Spezialfall ergibt sich bei den Bereichsfragen, wenn untere oder obere Schranke weggelassen werden, d.h. die kleinsten bzw. größten Schlüsselwerte gewählt werden. Auch hier wird eine TID-Liste der qualifizierten Tupel zurückgeliefert.

Natürlich können alle obigen Anfragetypen als Spezialfälle der Bereichsfrage formuliert werden. Weiterhin ist es offensichtlich, daß man alle Operationen, die zur Überwachung und Einhaltung der Relationalen Invarianten benötigt werden, durch Folgen von Operationen auf den mehrdimensionalen Indexstrukturen nachbilden kann.

Ergänzend soll noch festgehalten werden, daß manche Anwendungen die Unterstützung der Ähnlichkeitssuche bei mehrdimensionalen Indexstrukturen fordern. Wenn beispielsweise eine Punktsuche kein Ergebnis liefert, möchte man ein oder mehrere Tupel finden, die der vorgegebenen Suchbedingung möglichst nahe kommen (nearest neighbor query, best match query). Ein wesentliches Problem ist dabei die Definition der Ähnlichkeit; unter Vorgabe einer Distanzfunktion sind solche Tupeln zu ermitteln, die eine nach dieser (anwendungsabhängigen) Funktion minimale Distanz zum spezifizierten Kriterium aufweisen. Die Unterstützung der Best-Match-Anfrage soll hier jedoch nicht im Mittelpunkt der Betrachtungen stehen.

Wir diskutieren hier meist den Fall k=2, weil graphische Darstellung und intuitives Verständnis hier am leichtesten fallen. Erweiterungen sind in den meisten Fällen offensichtlich. Wir beziehen uns also auf den Fall, bei dem in Relation R zwei Attribute A1 und A2 gemeinsam als Schlüssel auftreten. A1 habe die Schlüsselwerte a11, ... a1n und A2 die Schlüsselwerte a21, ..., a2m. Als Beispiel können wir für A1 und A2 ABTNR mit {K01, ... K99} und ALTER mit {20, ..., 65} heranziehen.

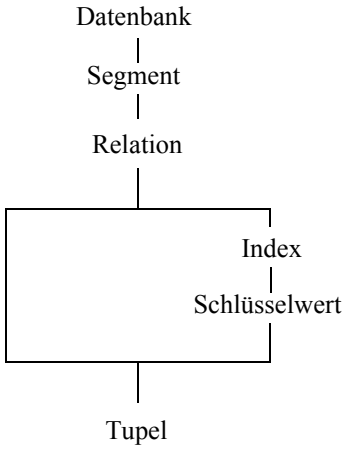
2.2 Anforderungen des Transaktionskonzeptes

Alle Datenbankoperationen werden innerhalb einer Transaktion als Kontrollstruktur abgewickelt. Dabei gewährleistet das DBMS die datenbankseitige Einhaltung der ACID-Eigenschaften, was besondere Anforderungen an die Sicherung der semantischen Integrität der Daten, an die Synchronisation der Zugriffe im Mehrbenutzerbetrieb sowie an die Logging- und Recovery-Funktionen stellt [HR83]. In unserem Zusammenhang interessiert neben der Einhaltung der Relationalen Invarianten vor allem die Serialisierbarkeit der parallelen Transaktionen. Diese Eigenschaft der Transaktionsverarbeitung erfordert bestimmte Synchronisationsmaßnahmen, die Konsistenzebene 3 oder die Wiederholbarkeit von beliebigen Leseoperationen innerhalb einer Transaktion einzuhalten erlauben.

In der Literatur wurden bisher sehr viele Synchronisationsprotokolle diskutiert, deren Tauglichkeit jedoch nicht nachgewiesen wurde. In praktischen Fällen wird deshalb immer das bewährte strikte Zwei-Phasen-Sperrprotokoll herangezogen. Wir benutzen es in seiner verallgemeinerten Form als hierarchisches Sperrprotokoll.

2.3 Allgemeines hierarchisches Sperrprotokoll

Als Ausgangspunkt der Diskussion von Synchronisationsverfahren auf mehrdimensionalen Zugriffspfaden führen wir die Sperrhierarchie ein, die für Datenzugriffe über Relationen-Scans oder Index-Scans bei eindimensionalen Zugriffsstrukturen vorgeschlagen wurde. Ihre wichtigste praktische Eigenschaft liegt in der Möglichkeit, Sperren in unterschiedlichen Granulaten anzufordern. Das Sperren von Objekten geschieht dann unter strikter Einhaltung einer Sperrhierarchie, z.B. nach folgender Halbordnung Datenbank, Segment, Relation sowie Index und Schlüsselwert oder Tupel:



Dieser azyklische Graph veranschaulicht, daß eine DB aus n Segmenten besteht, daß einem Segment m Relationen zugeordnet sein können und diese wiederum k Tupel besitzen können. Weiterhin können mehrere Indexstrukturen für eine Relation im selben Segment angelegt sein. Ein eindimensionaler Index besitzt j Schlüsselwerte, denen wiederum TIDs zugeordnet sind. Bei einem UNIQUE-Index ist pro Schlüsselwert ein TID vorhanden, das auf das zugehörige Tupel verweist; bei einem NONUNIQUE-Index können es mehrere (TID-Liste) pro Schlüsselwert sein.

Für jedes Objekt dieser Typen können beispielsweise die in der folgenden Kompatibilitätstabelle angegebenen Sperrmodi herangezogen werden [GLPT76,Gr78].

	IS	IX	S	SIX	X
IS	+	+	+	+	-
IX	+	+	-	-	-
S	+	-	+	-	-
SIX	+	-	-	-	-
X	-	-	-	-	-

Ein Objekt wird über seinen Namen gesperrt, beispielsweise ein Tupel mit Hilfe seines TID; wenn dies keinen DB-weit eindeutigen Namen ergibt, kann zusätzlich die RID der zugehörigen Relation benutzt werden (RID/TID).

Beim Sperren eines Objektes wird dynamisch ein Sperrkontrollblock (SKB) angelegt, der über eine Hashfunktion lokalisiert wird. Da ein sehr großer Namensraum zu verwalten und der zur Verfügung stehende Hauptspeicher für die Hashtabelle begrenzt ist, ist eine effiziente Kollisionsbehandlung erforderlich. Alle Synonyme in einer Hashklasse lassen sich z.B. verkettet speichern. Das Anlegen oder Freigeben eines SKB (Lock/Unlock) ist deshalb eine recht aufwendige Operation (> 100 Maschineninstruktionen).

Das hierarchische Sperrkonzept erzwingt die strikte Einhaltung der Sperrhierarchie. Der Zugriff auf ein Tupel kann, wie dargestellt, über die Objekte Datenbank, Segment und Relation oder eine der Indexstrukturen (wenn vorhanden) erfolgen, d.h., das Lokalisieren des Tupels kann mit Hilfe eines Relationen-Scans oder mit Hilfe eines Index-Scans über eine Indexstruktur bewerkstelligt werden. Eine Leseoperation erfordert zur Gewährleistung der Serialisierbarkeit nur das Sperren eines hierarchischen Pfades zum Tupel, während bei einer Einfüge- oder Lösch-Operation alle hierarchischen Pfade (alle Indexstrukturen und das Tupel oder die gesamte Relation) explizit oder implizit gesperrt werden müssen, da ja auch alle Indexstrukturen zu aktualisieren sind. Bei Änderung von Attributwerten eines Tupels genügt das Sperren des hierarchischen Pfades zum Tupel sowie das Sperren der betroffenen Indexstrukturen.

Folgendes hierarchische Sperrprotokoll der Transaktion T1 zum Lesen eines Tupels TID_1 der Relation RID1 über einen Index IID1 mit Schlüssel K1 (in Segment SID1) wäre denkbar:

```
T1:   Lock (DB, IS)
      Lock (SID1, IS)
      Lock (RID1, IS)
      Lock (IID1, IS)
      Lock (IID1K1, S)
      Lock (TID1, S)
      . . .
```

Diese Sperren bleiben bis Commit (T1) gesetzt. Solche langen Sperren verhindern, daß eine andere Transaktion Indexeintrag und Tupel verändert und gewährleisten die Wiederholbarkeit des Lesevorgangs. Deshalb muß die Sperre auf dem Schüsselwert (IID1K1) auch dann bis Commit (T1) gehalten werden, wenn es dafür kein Tupel gibt.

3. Mehrattributzugriff über eindimensionale Indexstrukturen

Bevor neue und zusätzliche Zugriffspfadtypen in einem DBMS implementiert und bereitgestellt werden, ist zu prüfen, wie gut die vorhandenen Hilfsmittel die gestellte Aufgabe zu lösen gestatten. In unserem Fall bedeutet das, den Einsatz von B*-Bäumen [Co79] bei der Realisierung von mehrdimensionalen Indexstrukturen für punktförmige Objekte zu untersuchen. Von seinem Konzept her erlaubt der B*-Baum die Indexierung (Invertierung) nach einer Dimension. Wie in Bild 1 für eine Indexstruktur I_{ABT}(ABTNR) für die Relation ABT veranschaulicht, erhält man durch den Einsatz eines B*-Baums eine Partitionierung des eindimensionalen Schlüsselraumes nach den verschiedenen Schlüsselwerten von ABTNR geordnet. In diesem Fall handelt es sich um eine Indexstruktur vom Typ

UNIQUE mit ≤ 2k Schlüsseleinträgen in inneren Baumseiten. Unterstellt man als Seitengröße 4KBytes und als Länge von ABTNR und Zeiger jeweils 4 Bytes, so erhält man für k den Wert 255 und als maximales Fan-out 510.

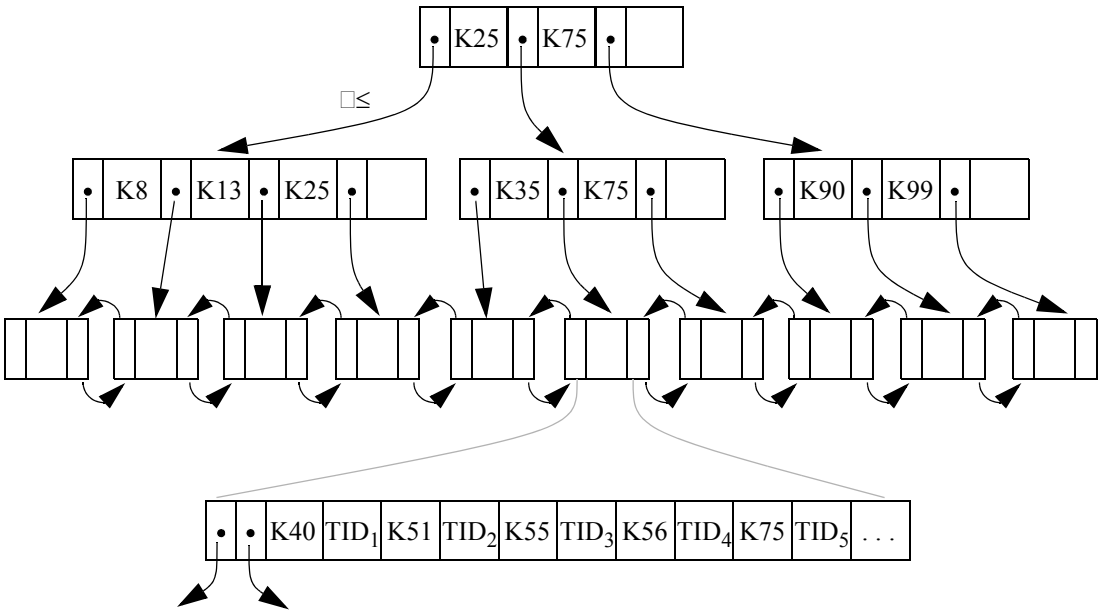
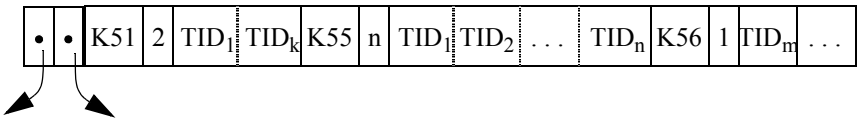


Bild 1: I_{ABT}(ABTNR) als B*-Baum

Für einen Index vom Typ NONUNIQUE kann dieselbe Baumstruktur herangezogen werden. Es ändert sich nur das Format der Blattknoten.



Blattknoten speichern variabel lange Einträge mit einem Schlüsselwert, einer Längenangabe und einer TID-Liste der entsprechenden Länge, wobei die TIDs auf alle Tupel der indexierten Relation, die den Schlüsselwert als Wert des indexierten Attributs besitzen, verweisen. Die obige Blattseite könnte beispielsweise zum Index I_{PERS}(ANR) gehören. Auch in diesem Fall würde der eindimensionale Schlüsselraum nach den Werten eines Schlüssels (ANR) partitioniert werden.

3.1 Separate Attribute als Schlüssel

Zur Partitionierung der zu speichernden punktförmigen Objekte (des Schlüsselraums) können Schlüsselwerte und Schlüsselbereiche verwendet werden. Die Kerneigenschaft bei B*-Bäumen ist in bezug auf unsere Fragestellung, daß Schlüsselwerte oder -bereiche zur Organisation bzw. Zerlegung des Schlüsselraumes herangezogen werden. Haben wir zwei Dimensionen im Schlüssel, so können wir mangels besserer Möglichkeiten pro Dimension einen B*-Baum verwenden und nach außen hin das gewünschte Verhalten simulieren. In Bild 2 ist skizziert, wie auf die-

se Weise der zweidimensionale Schlüsselraum von A1 und A2 nach den Werten der beiden Schlüsselattribute zerlegt werden kann.

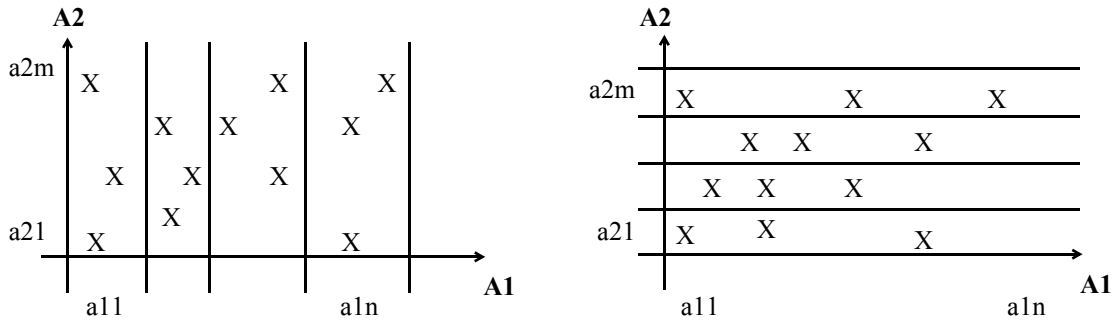


Bild 2: Getrennte Partitionierung des Schlüsselraumes nach A1 und A2

Der Zugriff nach einer Suchbedingung (A1=a1i AND A2=a2j) hat so zu erfolgen, daß nacheinander auf $I_R(A1)$ und $I_R(A2)$ die entsprechenden Suchbedingungen A1=a1i und A2=a2j überprüft werden. Bei erfolgreicher Suche werden die TID-Listen L(A1) und L(A2) zurückgeliefert. Die Ergebnisliste L(R) ergibt sich anschließend aus $L(A1) \cap L(A2)$. Das Ergebnis dieser Simulation eines zweidimensionalen Index wird in Bild 3 noch einmal zusammengefaßt.

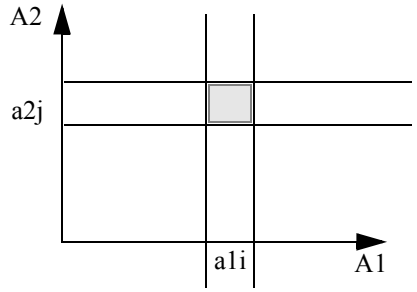


Bild 3: Ergebnis der getrennten (eindimensionalen) Suche

Da zwei getrennte Fetch-Operationen auf verschiedenen Indexstrukturen ausgeführt wurden, fallen als Zugriffskosten 2h Seitenzugriffe an (h = Höhe des B*-Baumes). Ferner ist für beide Indexstrukturen ein Sperrprotokoll zu befolgen (siehe Kap. 3.3). Durch die beiden Schlüsselsperren, die bis Commit zu halten sind, bleiben auf dem Schlüsselraum, wie in Bild 3 verdeutlicht, zwei sich überlappende Bereiche gesperrt (A1=a1i OR A2=a2j).

Weiterhin wird an diesem Beispiel klar, daß eine Transaktion, die nicht alle Sperren für k separate Indexstrukturen sofort erhält, ihre bis zum Wartezeitpunkt erworbenen Sperren wieder zurückgeben sollte, um Blockierungen und Deadlock-Gefahr in Grenzen zu halten.

Einfügen und Löschen eines Tupels mit (a1i,a2j,TID) erfordert offensichtlich einen Aufwand, der von der Anzahl der Dimensionen abhängt. Es sind also nacheinander ein Insert(a1i,TID) in $I_R(A1)$ und ein Insert(a2j,TID) in $I_R(A2)$ erforderlich. Entsprechend umständlich hat ein Löschvorgang zu erfolgen.

Ist für ein zusammengesetztes Attribut A1A2 die Option UNIQUE definiert, so läßt sich diese Eigenschaft bei Existenz von zwei getrennten Zugriffsstrukturen $I_R(A1)$ und $I_R(A2)$ nicht ohne weiteres beim Einfügen nachweisen, weil die UNIQUE-Option weder für $I_R(A1)$ noch für $I_R(A2)$ gilt. Prinzipiell müssen zusätzlich zwei Anfragen, wie in Bild 3 skizziert, gestellt werden, die als Ergebnis der Schnittmengenbildung nur ein TID liefern dürfen.

Die skizzierte Simulation von mehrdimensionalen Zugriffspfadstrukturen ist zwar gangbar, aber doch recht aufwendig und umständlich. Deshalb soll noch eine zweite Idee untersucht werden, die das Problem auch auf eindimensionale Indexstrukturen reduziert.

3.2 Konkatenierte Attribute als Schlüssel

Man kann die Schlüssel A1 und A2 konkatenieren und A1 | A2 als einen Gesamtschlüssel auffassen. Die Schlüsselwerte von A1 | A2 sind dann die Konkatenationen der einzelnen Werte. Sie ergeben folgende lexikographische Ordnung:

```
 A1 | A2
a11 | a21
a11 | a22
     :
a11 | a2m
a12 | a21
     :
a1n | a2m
```

Diese Ordnung entspricht der Reihenfolge, in der die Schlüsselwerte in der Indexstruktur (in den Blättern des B*-Baumes) auftreten. Ein Schlüsselwert besteht dann aus der Konkatenation der einzelnen Attributwerte, wobei die Index-ID noch als Präfix genommen wird. Dabei läßt sich der aus k Einzelwerten variabler Länge bestehende Schlüsselwert so codieren, daß durch einen Vergleich die Sortierfolge zweier Schlüsselwerte festgestellt werden kann [BCE77, Hä78]. Selbst wenn die Einzelwerte nach aufsteigender und/oder absteigender Sortierfolge eingeordnet werden sollen, kann das Codierschema angewendet werden.

Die Aufsuchoperation für (A1=a1i AND A2=a2j) wird umgesetzt auf die Suchbedingung (A1 | A2=a1i | a2j) für $I_R(A1 | A2)$. Sie liefert das entsprechende TID oder die TID-Liste zurück. Punktfragen sowie Einfügen und Löschen einzelner TIDs (wobei der Schlüsseleintrag jeweils genau spezifiziert ist) sind unproblematisch.

Unter der Annahme, daß für die Überprüfung der Relationalen Invarianten sowohl für die Vater-Relation V als auch für die Sohn-Relation S Indexstrukturen mit k konkatenierten Attributen als Schlüssel ($k<<n$) vorliegen, so kann jeweils mit einer Punktfrage der Vater überprüft oder alle betroffenen Sohn-Tupel lokalisiert werden, wofür typischerweise bei B*-Bäumen mit jeweils drei Seitenzugriffe zu rechnen ist. Die Überprüfung der UNIQUE-Eigenschaft irgendeiner Attributgruppe kann automatisch während des Einfügens oder Änderns von einer Indexstruktur mit der Option UNIQUE übernommen werden. Wird diese Indexstruktur nur dazu benutzt, so sind alle Aktualisierungskosten direkt der Integritätssicherung zuzuschlagen. Nähere Einzelheiten der Kostenberechnung können [HR91] entnommen werden.

Auch die Synchronisationsfrage ist bei dieser Indexstruktur mit konkatenierten Attributen effektiv gelöst. Zum Sperren des B*-Baumes kann die Technik der Latch-Kopplung [BS77] eingesetzt werden; die Operationen auf der Indexstruktur lassen sich durch die in Kap. 3.3 zusammengefaßten Sperrprotokolle in zufriedenstellender Weise synchronisieren.

Jedoch stellen sich Probleme bei allgemeinerer Nutzung einer solchen Indexstruktur mit konkatenierten Attributen ein. Der Zugriff über konkatenierte Schlüssel ist nämlich nicht symmetrisch; A2 | A1 würde eine weitere Indexstruktur erzwingen. Sie ist zugeschnitten auf Bedingungen des Typs (A1=a1i AND A2=A2j). (A1=a1i) erfordert mit Fetch(A1=a1i) und Fetch Next (A1=a1i) das Aufsuchen von bis zu m TID-Listen und ihre Verknüpfung: L(a1i | a21) ∪ L(a1i | a22) ∪ ... ∪ L(a1i | a2m). Bereichsfragen mit der Spezifikation von A1 werden noch aufwendiger. Fragen, in denen A1 nicht spezifiziert wird, also A2=a2j oder gar A2>a2j, sind nicht sinnvoll auszuwerten. Ebenso problematisch sind OR-Verknüpfungen, also (a1=a1i OR A2=a2j). Für die Unterstützung von Best-Match-Fragen fehlt jede Voraussetzung.

Es muß also festgestellt werden, daß bisher keine in allen Punkten befriedigende Lösung gefunden werden konnte. Deshalb müssen wir auch "echte" mehrdimensionale Zugriffspfade für die Lösung unseres Problems untersuchen. Falls sich jedoch dabei keine besseren Alternativen auftun, sollte die eindimensionale Indexstruktur mit k konkatenierten Attributen zur Sicherung der Relationalen Invarianten eingesetzt werden. Da hier nur Punktfragen erforderlich sind, kommen ihre schlechten Eigenschaften bei allgemeineren Zugriffsformen nicht so zum Tragen.

3.3 Sperrprotokoll für eindimensionale Indexstrukturen

In [Mo90, HR91] wurden Sperrprotokolle für eindimensionale Indexstrukturen vorgeschlagen, die eine flexible Bereichssuche, Wiederholbarkeit von Leseoperationen (Konsistenzebene 3) und einen möglichst behinderungsarmen Änderungsdienst unterstützen. Eine vorgeschlagene Optimierung verbessert die Parallelität beim Einfügen von Schlüsselwerten, wenn keine Lesefolgen zu schützen sind. Zusammenfassend lassen sich die Sperrprotokolle wie folgt darstellen (siehe [HR91]):

T1:Insert (K_i, TID)	
Wurzel und Zwischenknoten	Latch (P_i,S)
Blattknoten	Latch (P_j,X)
Nächster Schlüsselwert K_j	Lock (K_j, IX, I) Vergleich mit Sperren ohne T1-Sperre
Einzufügender Schlüsselwert K_i	Lock (K_i, IX) wenn K_j durch T1 nicht in S, SIX oder X gesperrt ist. Lock (K_i, X) wenn K_j durch T1 in S, SIX oder X gesperrt ist.
T1:Delete (K_i, TID)	
Wurzel und Zwischenknoten	Latch (P_i, S)
Blattknoten	Latch (P_j, S)
Nächster Schlüsselwert K_j	Lock (K_j, X)
zu löschender Schlüsselwert K_i	Lock (K_i, X, I)

In den folgenden Tabellen sind die Protokollanforderungen für einen UNIQUE-Index zusammengefaßt. K_i sei der aktuelle und K_j der nächste Schlüsselwert im Index. Die Protokolle für Fetch-Operationen

bleiben auch bei anderen Indexstrukturen (z.B. NONUNIQUE) erhalten. Da immer der Schlüsselwert gesperrt wird, kann mit demselben Protokoll auch auf die gesamte TID-Liste zugegriffen werden. Für

Fetch (= K_i)

Sperranforderung	Sperrmodus
für K_i	S
für K_j	-

Fetch Next (≤ K_l) oder Fetch Next (< K_l): Stoppbedingung

Sperranforderung	Sperrmodus
für K_i	S
für K_j	S*

* bei (≤), wenn K_j nächster Schlüsselwert > K_l ist und kein $K_i = K_l$ existiert oder
bei (<), wenn $K_i < K_l$ und $K_j \geq K_l$

Fetch Prior (≤ K_l) oder Fetch Next (< K_l): Startbedingung

Sperranforderung	Sperrmodus
für K_i	S
für K_j	S*

* bei (≤), wenn $K_j < K_l$ und kein $K_i = K_l$ existiert oder
bei (<), wenn $K_i < K_l$ und $K_j \geq K_l$

Tabelle 1: Sperrprotokolle für Leseoperationen

einen NONUNIQUE-Index können bei Aktualisierungsoperationen gewisse weitere Optimierungen vorgenommen werden, die jedoch hier nicht diskutiert werden [Mo90, HR91].

Insert (K_i, TID)

Sperr-anforderung	Sperrzustand von K_j: S, SIX, X	sonst
für K_i	X	IX
für K_j	IX, I	IX, I

Delete (K_i, TID)

Sperr-anforderung	Sperrzustand von K_j: S, SIX, X	sonst
für K_i	X, I	X, I
für K_j	X	X

Tabelle 2: Sperrprotokolle für Modifikationsoperationen

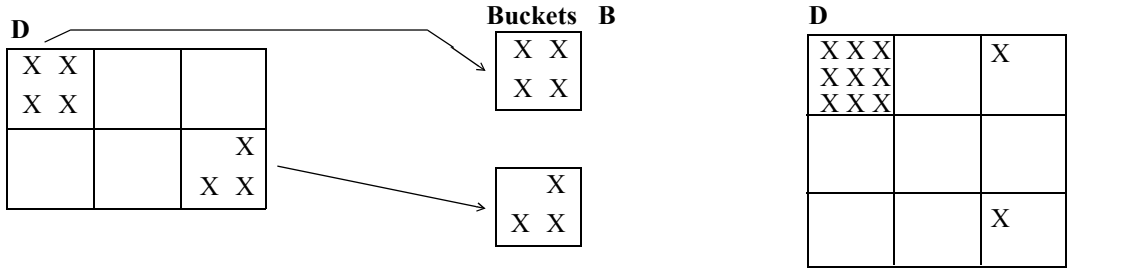
4. Einsatz von mehrdimensionalen Zugriffspfaden

4.1 Grundprobleme

Der B*-Baum gewährleistete durch sein Konstruktionsprinzip wichtige Eigenschaften einer Zugriffspfadstruktur. Das waren zum einen die stets balancierte Zugriffsstruktur, d.h. die gleiche Anzahl von Seitenzugriffen zu jedem Blatt, und zum anderen die Belegung des Baumes, die bei einer einfachen SplittingTechnik mindestens 50% betrug und durch allgemeinere Splitting-Verfahren beliebig nahe an 100% herangebracht werden konnte [Kü83]. Der B*-Baum, der typischerweise in seinen Blättern zusammen mit den Schlüsselwerten TIDs oder TID-Listen speichert, macht normalerweise keine Zusicherungen über die Speicherungsstruktur der Datensätze, auf die seine TIDs verweisen. Durch zusätzliche Maßnahmen kann jedoch ein Index mit Clusterbildung der Datensätze (clustered index) realisiert werden, wobei die Datensätze in der Reihenfolge der Indexschlüssel in einem Speicherbereich angeordnet werden. Diese Clusterbildung ist für das Leistungsverhalten bestimmter Operationen von großer Wichtigkeit.

Für mehrdimensionale Zugriffspfade müssen ähnliche Eigenschaften, die ihre praktische Tauglichkeit ganz wesentlich bestimmen, gefordert werden. Eine solche Zugriffsstruktur kann als eine Abbildungsfunktion aufgefaßt werden, die die Objekte eines mehrdimensionalen Datenraums D auf einen linearen Speicherbereich, der in Bukkets B (Seiten) aufgeteilt ist, abbildet. Bei dieser Abbildung ist es sehr wichtig, daß die topologische Struktur der Objekte in D auch in B erhalten bleibt, weil Zugriffsanforderungen häufig topologische Bezüge besitzen (Bereichssuche). Wie Bild 4 verdeutlicht, soll in der Speicherungsstruktur B eine Art mehrdimensionale Clusterbildung er-

zielt werden. Wenn die Objekte, die in D benachbart sind, auch benachbart gespeichert werden, sind offensichtlich Bereichsfragen oder Best-Match-Fragen sehr effizient abzuwickeln.



Erhaltung der topologischen Strukturen

Stark variierende Dichte der Objekte

Bild 4: Grundprobleme bei mehrdimensionalen Zugriffspfaden

In D ist typischerweise eine stark variierende Dichte der Objekte zu finden. Da durch die Speicherabbildung (Datei) eine gleiche Größe aller Buckets vorgegeben ist, ist es, wie in Bild 4 skizziert, keine gute Vorgehensweise, D für die Zuordnung zu B in ein regelmäßiges Raster aufzuteilen. Überdies ist noch mit einer starken Änderung der räumlichen Belegung über die Zeit zu rechnen, so daß das Raster ständig dynamisch geändert werden muß. Es ist also eine dynamische Reorganisation der Zuordnung von D und Speicherabbildung in B erforderlich, wie durch Bild 5 illustriert wird. Weiterhin sollen, wie beim B*-Baum, eine balancierte Zugriffsstruktur (mit höchstens 2 bis 3 Externspeicherzugriffen) sowie eine zufriedenstellende Belegung von B erzielt werden.

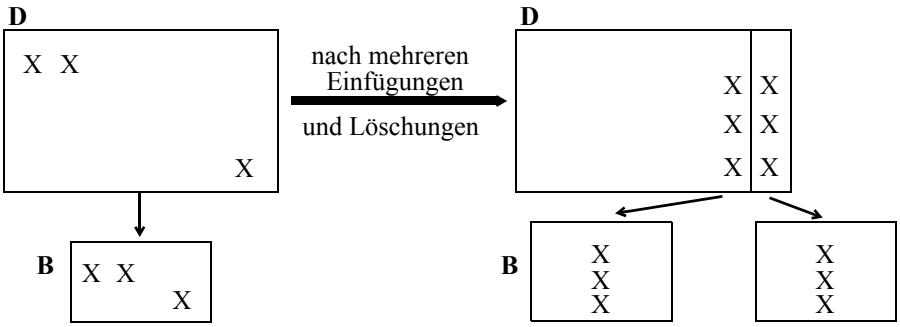


Bild 5: Dynamische Reorganisation bei mehrdimensionalen Zugriffspfaden

4.2 Überblick über mehrdimensionale Zugriffsstrukturen

Bisher wurden in der Literatur sehr viele Zugriffsstrukturen vorgeschlagen, die theoretisch das Problem der mehrdimensionalen Suche lösen. Jedoch erfüllen diese meist die praktischen Anforderungen nicht, da Topologieerhaltung oder balancierte Zugriffsstruktur usw. nicht garantiert werden können. Von den bekannteren Verfahren sollen hier einige nur skizziert werden. Verfahren, die auf die Speicherung und Suche räumlicher Objekte ausgelegt sind [Gu84, Gü89, SRF87], werden hier nicht betrachtet, da SQL2-Datenbanksysteme nicht für Non-Standard-Anwendungen ausgelegt sind.

Quadranten-Baum

Der Quadranten-Baum (quad-tree) unterstützt den zweidimensionalen Zugriff aufgrund eines zusammengesetzten Schlüssels A1A2 [FB74]. Er ist so aufgebaut, daß man sich die Sätze in einem zweidimensionalen Raum angeordnet denken kann, dessen Dimensionen (Koordinatenachsen) die einzelnen Schlüsselattribute A1 und A2 darstellen. Jeder Knoten ist durch einen Satz repräsentiert und kann bis zu vier Nachfolger haben. Die Wurzel teilt den zweidimensionalen Raum in 4 Quadranten auf. Die Quadranten lassen sich also durch die vier Teilbäume (Nachfolger) der Wurzel verkörpern. Dieses Prinzip wird nun rekursiv angewendet, d.h., jeder Quadrant wird wiederum in vier Quadranten aufgeteilt, wobei der erste in den Teilbaum eingespeicherte Satz die Wurzel des Teilbaums (Ausgangspunkt der Quadrantenaufteilung) ergibt. Auf diese Weise enthält der i-te Unterbaum eines Knotens alle Objekte (Sätze) im i-ten Quadranten.

Für unsere Analyse und Bewertung stellen folgende Eigenschaften KO-Kriterien dar:

- Einfügungen sind reihenfolgeabhängig; deshalb kann keine Balancierung gewährleistet werden. Es ist sogar eine Entartung zur linearen Liste denkbar.
- Löschen von Zwischenknoten ist sehr schwierig. Oft ist dabei das Neueinfügen des gesamten Teilbaumes erforderlich.
- Suchen ist nur bei der Punktfrage einfach, weil dadurch genau ein Pfad für den Baumdurchlauf festgelegt ist. Bereichssuche erfordert die rekursive Suche in Teilbäumen, wobei gleichzeitig das Suchfenster rekursiv zu zerlegen ist. Best-Match-Fragen sind schwierig durchzuführen.

Quadranten-Bäume und ihre Verallgemeinerungen (oct-tree, hex-tree) werden häufig in Ingenieuranwendungen genutzt, um mehrdimensionale Objekte darzustellen. Als Zugriffspfade zur Unterstützung referentieller Beziehungen im Relationenmodell erscheinen sie jedoch ungeeignet.

Mehrschlüssel-Hashing

Beim Mehrschlüssel-Hashing (multi-key hashing) [AU79] werden alle Schlüsselwerte eines zusammengesetzten Schlüssels ausgenutzt. Jedoch kann dabei nicht eine bloße Konkatenation aller Schlüsselwerte (als Superschlüssel) für das Hashing herangezogen werden, weil dadurch nur Punktfragen unterstützt werden würden. Wenn wir 2^B Buckets haben, benötigen wir zur Adressierung eine Folge von B Bits. Der Beitrag jedes der Schlüssel für die Hashadresse muß separat ermittelt werden, wenn wir einzelne Schlüsselattribute bei Anfragen ausnutzen wollen. Folgende Vorgehensweise stellt diese Forderung sicher: Die Bits der Bucketadresse werden im k-dimensionalen Fall in k Gruppen aufgeteilt, wobei dann der i-te Schlüssel die Bits für die i-te Gruppe zu liefern hat.

Ein Schlüsselwert $a = (a_1, a_2, ..., a_k)$ setzt sich aus den einzelnen am Schlüssel beteiligten Attributwerten (Teilschlüssel) a_i, $1 \le i \le k$, zusammen. B wird in geeigneter Weise so in b_i, $1 \le i \le k$, aufgeteilt, daß sich $B = \sum_{i=1}^{k} b_i$, $0 \le b_i \le B$, ergibt.

Die Hashfunktion h_i für den i-ten Teilschlüssel liefert eine Folge von b_i Bits mit

$$0 \leq h_i(a_i) \leq 2^{b_i} - 1$$

Die Adresse A des Bucket ergibt sich dann durch Konkatenation der Ergebnisse der k Hash-Funktionen:

$$A = h_1(a_1) \mid h_2(a_2) \mid ... \mid h_k(a_k)$$

Das Verfahren partitoniert den Suchraum offensichtlich in k Dimensionen. Es erlaubt sehr effiziente Punktfragen. Bei Partial-Match-Fragen werden eine oder mehr Bitgruppen in der Adresse nicht angegeben, wodurch dann alle Bitkombinationen der Gruppen ausgewertet werden, d.h., alle Buckets, die eine betreffende Dimension repräsentieren, werden zurückgeliefert.

Wesentliche Nachteile der Struktur sind folgende:

- Durch das Hash-Verfahren wird die topologische Anordnung der Daten in D bei der Abbildung auf B "zerschlagen".
- Relative Suchoperationen wie beispielsweise Fetch Next sind nicht möglich, d.h., Bereichsfragen oder Best-Match-Fragen sind prinzpiell nicht durchführbar.

Aus diesen Gründen scheidet dieses Verfahren für unsere Zwecke aus.

Mehrdimensionale binäre Suchbäume

Mehrdimensionale binäre Suchbäume oder k-d-Bäume [Be75] organisieren ihre Baumstruktur auf jeder Baumebene zyklisch nach den einzelnen Schlüsseln des Gesamtschlüssels. In jedem Baumknoten P sind alle k Schlüssel ($K_1(P)$, $K_2(P)$, ..., $K_k(P)$) und ein Diskriminator DISC(P) gespeichert. Außerdem besitzt ein Knoten P zwei Zeiger LOSON(P) und HISON(P) auf die beiden Unterbäume von P. Der Diskriminator erhält als Wert $d = DISC(P) \in \{1, ..,k\}$ und bezeichnet das Schlüsselattribut, nach dem die Nachfolgerknoten geordnet sind:

$$\forall\ Q \in LOSON(P) : K_d(Q) \leq K_d(P)$$
$$\forall\ R \in HISON(P) : K_d(R) > K_d(P)$$

Alle Knoten auf einer Baumebene i haben denselben Diskriminator, der sich beispielsweise durch die Formel $d = (i \bmod k) + 1$ festsetzen läßt. Als Optimierung läßt sich das Knotenformat auf ($K_d(P)$, DISC(P) reduzieren, da jeweils nur diese beiden Größen benötigt werden. Die Datensätze sind dann in Buckets (Seiten) abgespeichert, die die Blätter des Binärbaumes darstellen. Diese Baumstruktur mit Buckets als Blätter ist eigentlich eine Kombination von k-d-Baum und B*-Baum, weshalb sie auch als k-d-B-Baum [Ro81] bezeichnet wird.

Bei den Suchverfahren ist nur die Punktsuche einfach, weil der Suchweg durch die zyklische Betrachtung der einzelnen Schlüsselwerte auf einen Pfad führt. Die Suche nach einzelnen Schlüsselteilen oder die Bereichssuche führt auf die Anwendung rekursiver Techniken, deren Leistungsfähigkeit auch unter dem Gesichtspunkt von Synchronisationsprotokollen noch genauer zu bestimmen ist. Best-Match-Fragen müssen analog zu Bereichsfragen (Ähnlichkeitsbereich) abgewickelt werden. Problematisch ist auf alle Fälle der fehlende Mechanismus zur Gewährlei-

stung einer balancierten Zugriffsstruktur. Dadurch ergibt sich außerdem eine Abhängigkeit der Baumstruktur von der Einfügereihenfolge.

Aus dieser recht groben Diskussion ergeben sich schon eine Reihe von Aspekten, die genauer untersucht werden müssen, um einen Einsatz dieser Struktur in einem allgemeinen DBMS zu rechtfertigen. Deshalb detaillieren wir die Analyse dieser mehrdimensionalen Baumstruktur nicht weiter. Genauer wollen wir jedoch im folgenden das Grid File betrachten, dessen allgemeingültiger Einsatz und der seiner Weiterentwicklungen recht häufig empfohlen wird [Fr87,NHS84].

4.3 Das Grid File

Das Grid File ist eine k-dimensionale Zugriffsstruktur, die symmetrischen und gleichförmigen Zugriff über alle k Schlüssel eines Datensatzes bietet. Sie unterstützt ohne Unterschiede in der Vorgehensweise sowohl Punktfragen und Anfragen über Teilschlüssel als auch alle Varianten von Bereichsfragen sowie sogar Ähnlichkeitsfragen (best match queries).

4.3.1 Abbildungseigenschaften beim Grid File

Bei der Abbildung wird der Datenraum D durch ein orthogonales Grid (Raster) partitioniert, wobei das Raster nicht äquidistant gezogen wird, sondern sich in allen Dimensionen der Objektdichte eines Abschnitts anpaßt, wie es in Bild 6 illustriert ist. Dadurch entstehen sogenannte Gridblöcke GB, die rechteckig (oder bei höheren Dimensionen schachtelförmig) ausgeprägt sind.

Zur Abbildung eines k-dimensionalen Datenraumes D mit seinen Gridblöcken GB benötigen wir k Skalierungsvektoren S_i, ein Grid Directory GD sowie einen Bereich B mit Buckets B_j. Die Grideinteilung wird durch die Skalierungsvektoren erreicht, so daß zu jedem Zeitpunkt eine Menge von Gridblöcken existiert, die dem Produkt der durch die Skalierungsvektoren entstehenden Abschnitte in jeder Dimension entsprechen. Das Grid Directory verwaltet nun die einzelnen Gridblöcke, in dem es einen Eintrag pro Gridblock zur Verfügung stellt (1:1-Beziehung). Jeder Eintrag in GD ist ein Zeiger auf ein Bucket B_j, das alle im zugehörigen GB existierenden Objekte speichert. Die Objektdichte benachbarter GBs kann nun so gering sein, daß die Objekte mehrerer benachbarter GBs in einem Bucket gespeichert werden können. In diesem Fall verweisen mehrere benachbarte Elemente von GD auf das betreffende Bucket (n:1-Beziehung).

Im Gegensatz zum B*-Baum, wo der Datenraum nach einzelnen Schlüselwerten zerlegt wurde, organisiert das Grid File den die Schlüssel umgebenden Raum. Falls ein Bucket B_j die Objekte des zugehörigen Gridblocks GB_k nicht mehr aufnehmen kann, wird das Raster des Datenraums D in einer der Dimensionen i so verfeinert, daß GB_k (und andere GBs) zerlegt wird. Damit erhält der Skalierungsvektor S_i einen weiteren Eintrag, und als Folge davon ergibt sich eine Vergrößerung von GD in der Dimension i. Unter bestimmten Voraussetzungen können die Inhalte

von zwei benachbarten Buckets wieder zusammengelegt werden, so daß auch das Raster wieder vergröbert werden kann.

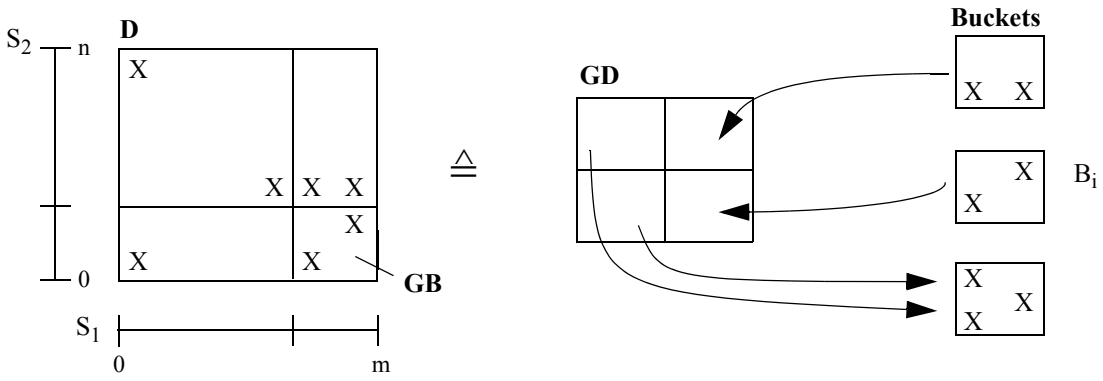


Bild 6: Abbildungsprinzip beim Grid File

Als Konsequenz der Verfeinerung und Vergröberung des Rasters werden bei der Implementierung (eindimensionale) dynamische Skalierungsvektoren S_i und ein k-dimensionaler dynamischer Vektor GD benötigt. Die Einzelheiten der komplexen Wartungsoperationen sind in [NHS84] dargestellt. Die Wartungsalgorithmen gewährleisten, daß bei Splitt- und Mischoperationen nur jeweils zwei Buckets betroffen sind und daß die durchschnittliche Belegung des Bucketbereichs B nicht beliebig klein werden kann. Schiefe Verteilungen der Objekte im Datenraum D führen höchstens zu feineren Rastereinteilungen und damit zu einerVergrößerung der S_i und von GD. Wegen der (n:1)-Beziehung zwischen den Elementen von GD und den Buckets B_j schlagen solche Verteilungen nicht auf die Belegung der B_j durch.

Als generelle Annahme des Leistungsverhaltens von Grid Files gilt das “Prinzip der zwei Plattenzugriffe”. Dabei wird unterstellt, daß sich alle Skalierungsvektoren im Hauptspeicher befinden und daß bei einer Anfrage durch Manipulation ihrer Werte die Elemente von GD bestimmt werden können, die die Zeiger auf die gesuchten Buckets enthalten. Bei einer Punktfrage ist dann ein Zugriff auf GD und ein Zugriff auf das betreffende Bucket erforderlich. Bei Bereichsfragen mit großen Treffermengen wird oft der erste Treffer nach zwei Plattenzugriffen gefunden.

In Bild 7 wird versucht, die Dynamik in einem Grid File zu illustrieren. Dabei wird von einem zweidimensionalen Datenraum ausgegangen. Der Zustand des Grid Files wird jeweils durch GD, die S_i und die Buckets repräsentiert. In Bild 7 werden zur Darstellung der Skalierungsvektoren keine linearen Listen (ARRAY) vorgeschlagen, sondern zweifach gekettete Listen mit expliziter Bezeichnung der referenzierten Zeile oder Spalte (allgemeiner: (k-1)-dim. Ebene). Zur Ermittlung des nächsten oder vorherigen Skalierungswertes (Next (S_i), Previous (S_i)) sind dann die Vorwärts- und Rückwärtsverzeigerungen zu verfolgen. Diese Indirektion erlaubt es, das Grid Directory GD an seinen Rändern wachsen zu lassen. Wenn eine (k-1)-dimensionale Ebene zu GD hinzugefügt oder von GD gelöscht wird, dann ist im zugehörigen Skalierungsvektor nur ein Eintrag und in GD nur die Erweiterung/Löschung betroffen. Diese explizite Indexierung hilft also die Änderungen in GD und S_i zu minimieren.

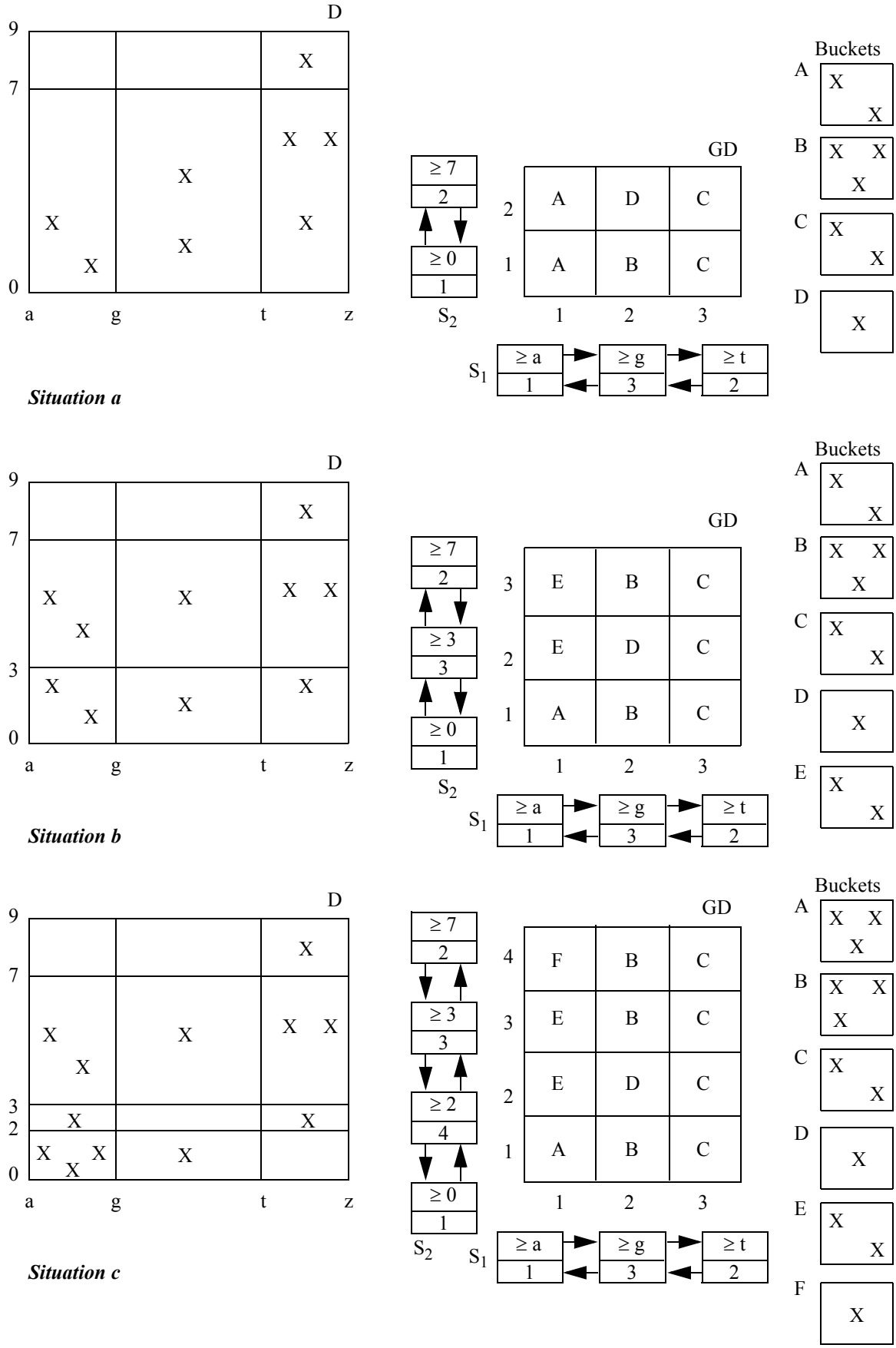
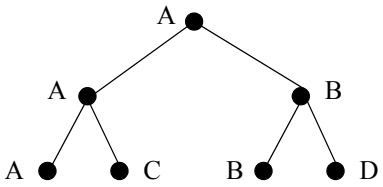


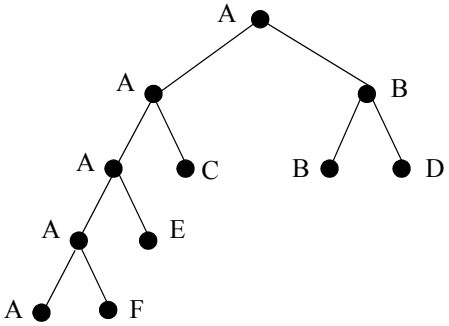
Bild 7: Beispiel für die Verfeinerung des Datenraums und der zugehörigen Entwicklung des Grid Files

Durch D wird veranschaulicht, wie durch Objekthäufungen ungleiche Rastergrößen entstehen. Wenn angenommen wird, daß in einem Bucket b=3 Datensätze Platz finden, dann könnte Situation a durch eine Verfeinerung der S_2-Dimension mit Rasterlinie 7 entstanden sein. Zuvor wurde die S_1-Dimension erst durch Linie t und dann durch Linie g verfeinert. Daraus resultierten 3 Erweiterungen von GD und die folgenden 3 Splitt-Vorgänge auf Buckets:



Situation b ist dadurch entstanden, daß zwei neue Objekte in den durch GD(1,1) repräsentierten Gridblock aufgenommen wurden. Dadurch wurde durch die Speicherung der entsprechenden Datensätze ein Splitt-Vorgang von Bucket A und eine Verfeinerung der S_2-Dimension (mit 3) ausgelöst. In Situation C wurde nochmals ein Wachstum von Gridblock GD(1,1) angenommen, wodurch Bucket A wiederum aufgeteilt und S_2 (bei 2) verfeinert wurde.

Diese abschließende Situation C hat nun bezogen auf die Buckets eine Entwicklungsgeschichte, die der folgende Binärbaum repräsentiert.



In Bild 7 wird mehrfach verdeutlicht, daß mehrere GD(i,j) die gleiche Bucketadresse besitzen können. Dadurch wird beispielsweise erreicht, daß leere Gridblöcke kein eigenes Bucket belegen oder mehrere schwach belegte Gridblöcke sich zusammen ein Bucket teilen können. Jedoch gibt es eine Einschränkung der Gridblock-Konfigurationen, die die gleiche Bucketadresse enthalten dürfen. Sie müssen konvex oder rechteckig sein. Beispielsweise führte diese Regel in Situation b auf die Belegung von GD(1,2) und GD(1,3) durch E. Bei der Ermittlung der Rechteckformen darf nicht von der physischen Repräsentation ausgegangen werden, sondern die logischen Reihenfolgen gemäß der Skalierungswerte sind maßgebend.

Auf der konzeptuellen Ebene ist die Anfragebearbeitung mit einem Grid File recht einfach. Ein Suchprädikat (A1=h AND A2=5) führt nach Bild 7c auf GD(3,3) und dann auf Bucket C. Eine Anfrage mit Teilschlüsseln, also zum Beispiel (A1=b), ermittelt die GD-Einträge GD(1,*) und die Buckets A, E, F, deren Datensätze dann genauer mit (A1=b) überprüft werden müssen. Entsprechend führt eine Bereichsfrage (A1≥h AND A1≤k AND A2≥1 AND A2≤7) auf GD(3,*) und auf das Bucket C. Diese Beispiele vermitteln den Eindruck eines vernünftigen, ja sogar effizienten Leistungsverhaltens.

4.3.2 Problemfälle bei der Abbildung

Das Grid-File-Konzept unterstellt zunächst, daß die Datensätze, auf die von GD aus verwiesen wird, in den Bukkets gespeichert sind. In solchen Fällen muß man typischerweise annehmen, daß b=10 bis b=40 vorliegt (Bucketgröße 4KB). Bei geringer Selektivität von Sekundärschlüsseln und ungünstiger Verteilung könnte dann zum Beispiel auftreten:

S_1 : ≥ a, ≥ d, ≥ d, ≥ d, ≥ f, ≥ s, ...

S_2 : ≥ 0, ≥ 5, ≥ 5,≥ 5, ≥ 7, ≥ 9, ...

Eine Punktanfrage (A1 = d AND A2 = 5) müßte dann auf den Skalierungsvektoren ganze Bereiche durchsuchen und würde 9 GD-Einträge zurückliefern.

Wenn hohe Selektivität der Attribute vorliegt, liefert eine Anfrage (A1=k AND A2=8) natürlich nur ein GD(i,j) und ein Bucket.

Solange das Grid File die Speicherung der Datensätze bestimmen kann, gibt es offensichtlich selbst bei extremen Datenverteilungen (sehr schiefe Schlüsselverteilungen) wenig Probleme, da solche Datenverteilungen zwar die Größe des GD erheblich beeinflussen, jedoch wenig Auswirkungen auf den Belegungsgrad der Buckets haben. In Datenbankanwendungen ist es jedoch oft wünschenswert, daß andere Kriterien die physische Speicherung der Datensätze bestimmen (Hashstruktur, Index mit Clusterbildung usw.). Dann müßte ein zusätzlich eingerichtetes Grid File mit einer Indirektion versehen werden, also anstelle der Datensätze müßten TIDs in den Buckets gespeichert werden. Dies kann jedoch sein Leistungsverhalten dramatisch ändern. Jetzt können in einem 4K-Bucket b ≈ 1000 TIDs gespeichert werden. Im Mittel wird dadurch GD auch erheblich kleiner. Eine Punktfrage führt auf ein GD(i,j) und ein Bucket; jedoch müßten jetzt 1000 TIDs in potentiell 1000 verschiedenen Seiten analysiert werden, was beispielsweise für einen Primärschlüsselzugriff (und andere) nicht tolerierbar ist. Auch bei Bereichsfragen wirkt sich die TID-Speicherungsform nicht positiv aus, weil hier die auf GD ausgewerteten Prädikate durch die qualifizierten GD(i,j) wesentlich größere Obermengen zurückliefern, die dann TID-weise auf Zugehörigkeit zum Suchprädikat getestet werden müssen.

Das Testen der verstreuten Datensätze kann man in einfacher Weise reduzieren, wenn man die in der Grid-Struktur verwendeten Schlüssel an jedes TID in den Buckets anhängt, in unserem Beispiel also (A1, A2, TID) als Einträge in den Buckets speichert. Wenn jeder Schlüssel z.B. 8 Bytes lang ist, also jeder Eintrag 20 Bytes umfaßt, reduziert sich das b ≈ 200; als Konsequenz wächst das Grid Directory entsprechend wieder. Die Punktfrage läßt sich dann wieder mit zwei (und einem) Externspeicherzugriffen behandeln, da das sich qualifizierende TID durch den Bukketzugriff ermittelt werden kann und der Datensatz daraufhin mit einem weiteren Plattenzugriff gefunden wird. Die skizzierte Idee führt zwar Schlüsselredundanz ein, was aber nicht so dramatisch zu sehen ist, da bei Schlüsselmodifikationen von A1 und A2 ohnehin das Grid File betroffen ist. Lediglich bei einer großen Anzahl k von Dimension ist es eine Frage der Speicherungsredundanz, da dann mit k zunehmend der ganze Satz zweimal gespeichert ist. Es bleibt jedoch festzuhalten, daß bei Indirektion der Satzspeicherung es zwingend erforderlich ist, in den Bukkets die Schlüssel zusammen mit der Satzadresse zu vermerken.

4.4 Leistungsaspekte bei der Überprüfung von Integritätsbedingungen

Eine UNIQUE-Option kann nicht direkt, wie bei der Indexstruktur, an ein Grid File gebunden werden; es bedeutet beim Einfügen und Ändern zusätzlichen Prüfungsaufwand. Die Buckets, die durch die bei der Modifikationsoperation qualifizierten GD-Einträge ermittelt werden, müssen vollständig durchsucht und auf Duplikate verglichen werden. Ist die Option nicht vorhanden, so muß sie von außen simuliert werden, indem eine Punktfrage mit den betreffenden Werten gestellt wird, bevor eine die UNIQUE-Option betreffende Modifikationsoperation angestoßen werden kann.

Unter der Annahme, daß für die Überprüfung der Relationalen Invarianten sowohl für die Vater-Relation V als auch für die Sohn-Relation S Grid Files für jeweils k Dimensionen vorliegen, so gestalten sich die entsprechenden Protokolle und Zugriffe konzeptionell recht übersichtlich. Jeweils mit einer Punktfrage können der Vater überprüft und alle betroffenen Sohn-Tupel lokalisiert werden. In der Regel bleiben dabei die Sperrprotokolle relativ einfach, da meist nur ein GD-Eintrag zu sperren ist (siehe Kap. 5). An Zugriffskosten zur Lokalisierung der beteiligten Tupel fallen normalerweise für jedes beteiligte Grid File zwei, höchstens drei Seitenzugriffe an (bei Clusterbildung). Nur bei gewissen pathologischen Fällen (siehe Kap. 4.3.2) können deutlich mehr Seitenzugriffe erforderlich sein. Da auch Einfügungen und Löschungen bei einem Grid File lokale Operationen sind, muß bei Aktualisierung von Vater-Tupeln jeweils nur mit ein bis zwei Bucketänderungen und in weniger häufigen Fällen mit Schreibaufwand bei Strukturmodifikationen im Grid Directory gerechnet werden. Bei Folgeänderungen an Sohn-Tupeln kann der Modifikationsaufwand jedoch drastisch größer werden (CASCADE- und SET-NULL-Optionen). Eine genaue Analyse des zu erwartenden Aufwandes erfolgt in einem getrennten Aufsatz.

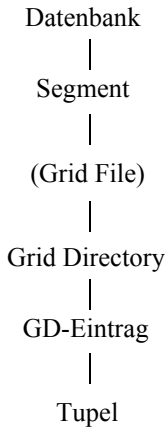
5. Synchronisation beim Grid File

In Datenbanksystemen erfolgen alle Operationen auf einem Grid File unter Kontrolle des Transaktionskonzeptes. Das bedeutet vor allem, daß alle Änderungen erst nach Commit der ändernden Transaktion für andere Transaktionen sichtbar werden. Weiterhin wird für jede Transaktion (auf Konsistenzebene 3) Wiederholbarkeit für ihre Lesevorgänge garantiert. Diese Aussagen gelten für die Benutzerdaten (Tupeln, Relationen), Systemdaten dagegen sind davon nicht betroffen. Deshalb müssen beispielsweise Strukturmodifikationen (Grid Directory, Skalierungsvektoren) des Grid Files bei einem Abort der auslösenden Transaktion nicht notwendigerweise rückgängig gemacht werden.

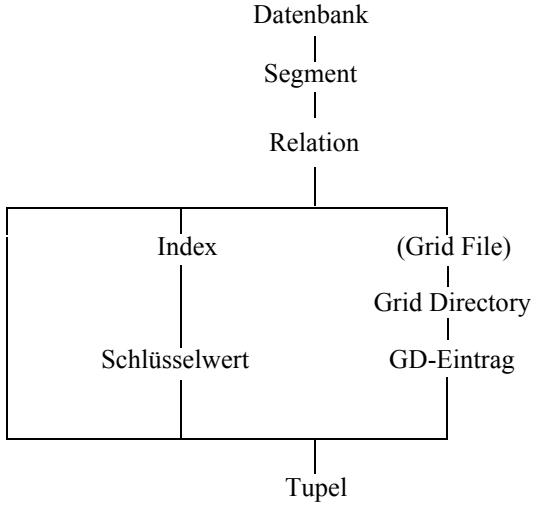
5.1 Speicherungsmodelle für Grid Files

In DBS dürfte es der Normalfall sein, daß man bei einem Grid File in den Buckets nur die Schlüssel mit dem TID des Satzes speichert und die Sätze der Relation in einer separaten Speicherungsstruktur (Segment) ablegt. Wir bezeichnen das als Modell 2 der Grid-File-Speicherung, bei dem die Relation auch noch über weitere Zugriffspfade erreicht werden kann. Als Modell 1 bezeichnen wir das Grid File, das in seinen Buckets die vollständigen Daten-

sätze speichert. In diesem Fall können die Sätze nur über das Grid File erreicht werden. Für diese Modelle hat ein hierarchisches Sperrprotokoll folgende (Halb-)Ordnung einzuhalten.



Modell 1



Modell 2

Das Grid File besteht aus Grid Directory, Skalierungsvektoren und Bucketbereich. Es braucht in der Sperrhierarchie nicht explizit geführt werden, da das Grid Directory seine Rolle übernehmen kann (1:1-Beziehung). Außerdem wird davon ausgegangen, daß für die Manipulation der einzelnen Grid-File-Komponenten Kurzzeitsperren (Latches) eingesetzt werden. Bei Modell 1 führt nur ein hierarchischer Pfad zum Tupel, der natürlich beim Lesen und beim Aktualisieren des Tupels zu sperren ist. Bei Modell 2 führen n Pfade zum Tupel; deshalb gilt die allgemeine Regel, daß beim Lesen ein, beim Aktualisieren aber alle Pfade gesperrt sein müssen.

Ganz wesentlich für die operationale Tauglichkeit eines Grid Files ist sein dynamisches Verhalten im Mehrbenutzerbetrieb. Beispielsweise kann es bei Operationen zur Sicherung der Relationalen Invarianten im DB-Betrieb ein hohes “Verkehrsaufkommen” und viele Blockierungssituationen auf den entsprechenden mehrdimensionalen Zugriffspfaden geben, zumal ja parallele Transaktionen auch komplexe Bereichsfragen mit großen Granulaten bearbeiten können.

5.2 Synchronisation über die Skalierungsvektoren

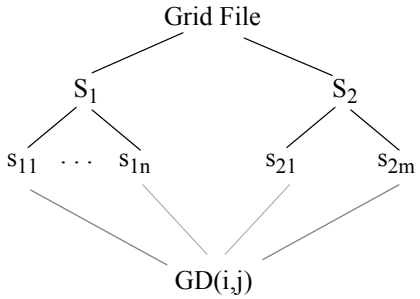
Unter Berücksichtigung unserer Randbemerkungen zur indirekten Speicherung der Datensätze können wir beim Grid File jedoch eine hohe Leistungsfähigkeit aller geforderten Operationen (Kap. 2) für mehrdimensionale Zugriffspfade erwarten. Bei einem Datenbank-Einsatz laufen diese Operationen im Mehrbenutzerbetrieb ab, wobei die ACID-Eigenschaften des Transaktionskonzeptes [HR83] zu beachten sind.

Dadurch, daß beim Grid File Teile oder Abschnitte eines k-dimensionalen Schlüssels als Suchbedingungen spezifiziert werden können, treten typischerweise bei Anfragen Prädikate in Form von Bereichsbedingungen sehr häufig auf. Mit dem Konzept der Prädikatssperren könnte man die Prädikate der Anfragen direkt zur Synchronisation nutzen. Jedoch wurde schon häufig festgestellt [EGLT76,HR91], daß Prädikatssperren nur sehr ineffektiv verwaltet werden können und deshalb auch noch in keinem DBMS ihren Platz gefunden haben. Aus diesen Gründen suchen

wir für unser Synchronisationsproblem eine Lösung beim hierarchischen Sperrkonzept basierend auf physischen Objektsperren.

Für Indexstrukturen, die mit B*-Bäumen realisiert wurden, konnten effiziente und effektive Sperrprotokolle entwickelt werden, wobei Bereichssperren durch das Konzept des "Sperren des nächsten Schlüsselwertes" simuliert wurden [Mo90,HR91]. Wesentlich für das einfache Funktionieren dieser Sperrprotokolle war die Einschränkung des Schlüsselraumes auf eine Dimension. Auch bei der Konkatenation von Schlüsseln (Kap. 4) wird der Schlüsselraum auf eine Dimension reduziert; die Symmetrie des Zugriffs geht dabei jedoch verloren. Beim Grid File dagegen muß seine fundamentale Eigenschaft des symmetrischen k-dimensionalen Zugriffs auch bei der Einführung von Sperrprotokollen erhalten bleiben, weil sonst alle seine operationalen Charakteristika verlorengehen würden.

Um das Problem besser zu verstehen, versuchen wir zunächst einen hierarchischen Ansatz für das Grid File über die Skalierungsvektoren S_i und deren Werte s_{ij}, die schließlich auf die Einträge GD(i,j) des Grid Directory verweisen.



Diese Struktur ist nicht als Typ, sondern als Ausprägungsstruktur für k=2 zu verstehen. Mit einem hierarchischen Sperrprotokoll soll nun versucht werden, die Vorteile von hierarchischen Indexsperren auf Grid Files zu übertragen. Das exklusive Sperren des gesamten Grid Files GF1 ist unproblematisch: Lock (GF1,X).

Die Punktsuche mit einer Bedingung (A1=h AND A2=5) müßte offensichtlich der Reihe nach S_1, S_2 und deren Nachfolger sperren:

T1: Fetch (A1=h AND A2=5)

. . .
Lock (GF1, IS)
Lock (S_1, IS)
Lock (s_{1i}, S) s_{1i} deckt A1=h ab
Lock (S_2,IS)
Lock (s_{2j},S) s_{2j} deckt A2=5 ab

Eine parallele Änderungstransaktion möchte folgenden zusammengesetzten Schlüsselwert einfügen:

T2: Insert (A1=t, A2=7:TID)

. . .
Lock (GF1, IX)
Lock (S_1, IX)
Lock (s_{1k}, X) s_{1k} deckt A1=t ab

Lock (S_2, IX)
Lock (s_{2l}, X) s_{2l} deckt A2=7 ab

Die Prädikate von T1 und T2 überlappen nicht. Mit den gezeigten Sperrprotokollen ist dieser einfache Fall zu beherrschen: beide Operationen können parallel erfolgen. Die Situation ist nochmals in Bild 8 veranschaulicht.

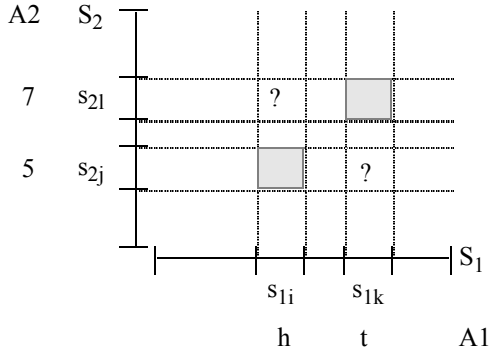


Bild 8: Überdeckung des Schlüsselraums durch zwei Prädikate (Punktfragen)

Eine kleine Veränderung eines der Prädikate zeigt jedoch, daß das gewählte Sperrprotokoll auch offensichtlich konfliktfreie Verarbeitungssituationen verhindert. Möchte T2 beispielsweise irgendein Tupel mit Teilschlüssel A2=5 oder mit A1=t einfügen, erkennt das Sperrprotokoll eine Wartesituation:

T2: Insert (A1=t, A2=5:TID)

. . .
Lock (GF1, IX)
Lock (S_1, IX)
Lock (s_{1k}, X)
Lock (S_2, IX)
Lock (s_{2j}, X) T2 muß auf Commit (T1) warten

Das Fragezeichen in Bild 8 zeigt, daß die eben skizzierte Sperranforderung unproblematisch ist. Durch die beiden unabhängigen hierarchischen Sperranforderungen werden nicht die gesamten Prädikate auf einmal auf ihre Konfliktfreiheit überprüft, sondern jedes Teilprädikat wird unabhängig mit dem entsprechenden Teilprädikat der momentanen Sperranforderung überprüft. Sobald also für eines von k Attributen die beiden Schlüsselwerte in irgendeiner Form überlappen und es sich um Leser/Schreiber- oder Schreiber/Schreiber-Transaktionen handelt, kann eine Sperranforderung nicht gewährt werden.

Noch wesentlich gravierender wirkt sich die unabhängige Auswertung von Teilprädikaten bei Partial-Match-Fragen oder Bereichsfragen aus. Ein Prädikat (A1=h) muß für die Zwecke der Sperranforderung um den gesamten Wertebereich von A2 ergänzt werden, woraus folgende konkrete Anforderung resultiert:

T1: Fetch (A1=h AND (A2 ≥ 0 AND A2 ≤ 9))

. . .
Lock (GF1, IS)
Lock (S_1, IS)
Lock (s_{1i}, S)
Lock (S_2, S)

Eine Schreibtransaktion muß jetzt in jedem Fall warten:

T2: Insert (A1=t, A2=5:TID)

. . .
Lock (GF1, IX)
Lock (S_1, IX)
Lock (s_{1k}, X)
Lock (S_2, IX) T2 muß auf Commit (T1) warten.

Weiterhin wird an diesem Beispiel klar, daß T2 eigentlich seine erworbenen Sperren freigeben müßte, um Blokkierungen und Deadlock-Gefahr nicht übermäßig steigen zu lassen. Eine Transaktion kann nämlich bereits auf k-1 Skalierungsvektoren und k-1 Skalierungswerten Sperren besitzen, bevor eine solche Wartesituation eintritt.

Wenn T1 mit dem Prädikat $P_1 = (P_{11} \wedge P_{12})$ und T2 mit dem Prädikat $P_2 = (P_{21} \wedge P_{22})$ zugreifen, so sollte ein Sperrprotokoll einen Konflikt melden, wenn P_1 und P_2 eine gemeinsame Schnittmenge besitzen, also $P_1 \wedge P_2$ sich erfüllen läßt. Durch die separate Prüfung der einzelnen Dimensionen wird als Ergebnis nicht $P_1 \wedge P_2$ überprüft, sondern $(P_{11} \wedge P_{21}) \vee (P_{12} \wedge P_{22})$ oder im k-dimensionalen Fall $(P_{11} \wedge P_{21}) \vee (P_{12} \wedge P_{22}) \vee ... \vee (P_{1k} \wedge P_{2k})$.

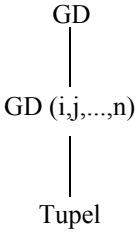
Wenn es sich um Bereichsanforderungen von Lesern und Schreibern handelt, tritt zwangsläufig eine Blockierung ein, sobald in einer der k Dimensionen eine Überlappung der Schlüsselwerte (nicht notwendigerweise der Bereiche) auftritt. Ein Leser mit einer Partial-Match-Frage blockiert alle Schreiber, obwohl gar kein Konflikt vorliegen muß. Es ist offensichtlich, daß ein derartig “pessimistisches” Verhalten bei DBMS nicht akzeptiert werden kann.

5.3 Ein hierarchisches Sperrprotokoll für das Grid Directory

Der Versuch, den symmetrischen Zugriff auf ein k-dimensionales Grid File durch k eindimensionale Zugriffe mit Hilfe der Skalierungsvektoren und -werte zu synchronisieren, führte auf ein denkbar schlechtes Sperrprotokoll, das in den meisten Fällen fiktive Konflikte zwischen Lesern und Schreibern oder Schreibern und Schreibern anzeigt. Der Grund dafür liegt darin, daß die Konfliktanalyse durch die Separierung in k hierarchische Sperrpfade de facto keine AND-Verknüpfung der Zugriffs-Prädikate auswertete, sondern eine OR-Verknüpfung aller durch AND verbundenen Teilprädikate.

5.3.1 Synchronisation für Modell 1

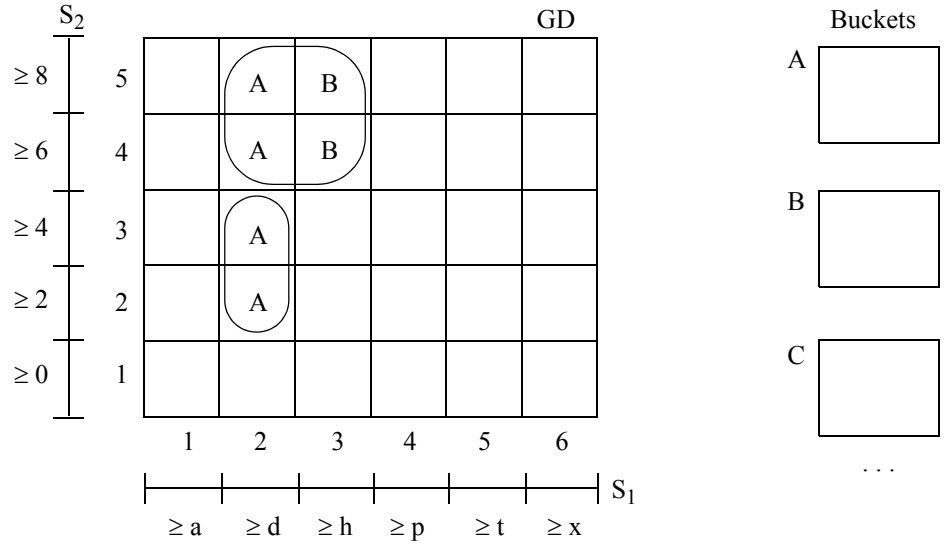
Es ist offensichtlich, daß innerhalb eines Grid Files keine Zugriffshierarchie existiert, die die Skalierungsvektoren und -werte einbezieht. Lediglich das Grid Directory und seine Einträge können für ein hierarchisches Protokoll genutzt werden. Im k-dimensionalen Fall ergibt sich folgende Typhierarchie:



Es können also nur das gesamte GD, jeder GD-Eintrag oder jedes Tupel einzeln gesperrt werden. Der für eine Sperranforderung benötigte GD-Eintrag muß aus dem Prädikat der betreffenden Operationen mittels der Skalierungsvektoren ermittelt werden. Da sich typischerweise bei Bereichsbedingungen Mengen von GD-Einträgen qualifizieren, sind offensichtlich in solchen Fällen Mengen von Sperren anzufordern, was sehr komplexe Sperrprotokolle impliziert. Sollen Tupelsperren als das feinste Sperrgranulat eingesetzt werden, so müssen die TIDs der qualifizierten Tupel in den Buckets ermittelt werden, bevor jedes Tupel einzeln gesperrt werden kann. Folgende Zwischenschritte sind also erforderlich:

- Abbildung der Teilprädikate $P_1 \wedge P_2 \wedge ... \wedge P_k = P$ auf die Skalierungsvektoren $S_1, ..., S_k$
- Ermittlung der GD-Indizes für k Dimensionen mit Hilfe der Skalierungswerte
- Berechnen der Menge der betroffenen GD-Einträge GD (i,j,...,n) aus den GD-Indizes
- Sperranforderungen für alle ermittelten GD-Einträge GD (i,j,...,n)
- Überprüfung der in den qualifizierten Buckets erreichbaren Tupel mit Hilfe von P
- Sperranforderungen für die qualifizierten Tupel über ihre TIDs.

Im zweidimensionalen Fall sei das folgende Grid File, wie in Bild 9 dargestellt, gegeben.



P_1 = (A1 = e AND (A2 ≥ 3 AND A2 ≤ 5)) liefert GD (2,2) und GD (2,3).

P_2 = ((A1 ≥ f AND A1 ≤ n) AND (A2 ≥ 7 AND A2 ≤ 9)) liefert GD (2,4), GD (3,4), GD (2,5) und GD (3,5).

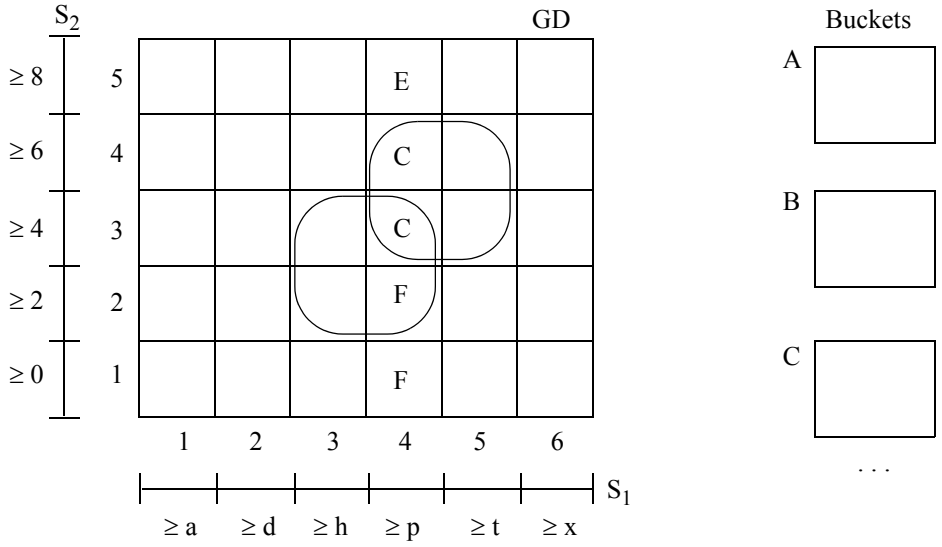
Bild 9: Beispiel zur Synchronisation auf einem Grid Directory

Betrachten wir zunächst Sperrprotokolle nach Modell 1. T1 fordere P_1 zum Lesen an, während T2 in den durch P_2 beschriebenen Sätzen ändern will. Die entsprechenden Ausschnitte des Sperrprotokolls ergeben sich wie folgt:

T1:	. . .	T2:	. . .
	Lock (GD, IS)		Lock (GD,IX)
	Lock (GD (2,2), S)		Lock (GD (2,4), X)
	Lock (GD (2,3), S)		Lock (GD (3,4), X)

. . .	Lock (GD (2,5), X)
	Lock (GD (3,5), X)
	. . .

Beide Anforderungen können parallel gewährt werden. Mit der auf einen GD(i,j) gewährten Sperre sind implizit alle Tupel gesperrt, die über GD(i,j) erreicht werden können und das Anforderungsprädikat P_i erfüllen. Alle Zugriffe auf die Buckets sind jedoch zusätzlich mit Hilfe von Kurzzeitsperren zu synchronisieren. Nach Bild 9 greifen T1 und T2 zwar auf disjunkte Satzmengen zu, beide benötigen jedoch Bucket A, was durch einen Latch-Erwerb zu regeln ist.



P_3 = ((A1 ≥ n AND A1 ≤ s) AND (A2 ≥ 3 AND A2 ≤ 4)) liefert GD (3,2), GD (4,2), GD (3,3) und GD (4,3).

P_4 = ((A1 ≥ p AND A1 ≤ w) AND (A2 ≥ 5 AND A2 ≤ 7)) liefert GD (4,3), GD (5,3), GD (4,4) und GD (5,4).

Bild 10: Beispiel zur Synchronisation auf einem Grid Directory

Mit dem folgenden Beispiel wollen wir eine Blockierungssituation erzeugen. T3 fordere P_3 zum Lesen an, während T4 die durch P_4 charakterisierten Sätze aktualisieren will. Es ergeben sich folgende Sperrprotokoll-Ausschnitte:

T3:		T4:	
	. . .		. . .
	Lock (GD, IS)		Lock (GD, IX)
	Lock (GD (3,2), S)		Lock (GD (5,4), X)
	Lock (GD (4,2), S)		Lock (GD (4,4), X)
	Lock (GD (3,3), S)		Lock (GD (5,3), X)
	Lock (GD (4,3), S)		Lock (GD (4,3), X)

Die beiden Sperranforderungen auf GD (4,3) sind unverträglich und erzeugen eine Wartesituation für die zuletzt anfordernde Transaktion. Die Veranschaulichung in Bild 10 zeigt, daß im allgemeinen Fall eine Transaktion für eine Grid-File-Operation eine Menge von GD-Einträgen sperren muß, so daß bereits zwei Grid-File-Operationen miteinander in einen Deadlock geraten können, der in geeigneter Weise aufzulösen ist. Falls es, wie oben, zu nur einer Blockierungssituation kommt, muß die wartende Transaktion ihre Sperren der letzten Anforderung trotzdem wieder freigeben, soll es nicht zu langen Wartesituationen mit kaskadierenden Granulaten kommen, die schließlich

die Arbeit auf dem Grid File serialisieren oder gar durch einen Deadlock vollständig blockieren. Da potentiell große Mengen von Sperren anzufordern und bei Mißerfolg wieder freizugeben sind, können solche Protokolle hochgradig ineffektiv werden. Bei der Neuanforderung der erforderlichen Sperren (nach Commit der blockierenden Transaktion) ist es nicht leicht, der Transaktion den Erfolg zu garantieren (Deadlocks, Blockierungen, Livelocks).

Ein weiterer Aspekt, der durch das Beispiel von Bild 10 sichtbar wird, soll hier erwähnt werden. Durch die Abbildung der Teilprädikate auf die Skalierungswerte wird im allgemeinen eine Vergröberung der Granulate erreicht. P3 und P4 stehen nämlich gar nicht in Konflikt miteinander, aber die abgeleiteten Mengen von GD-Einträgen. Das Beispiel zeigt also, daß unser Sperrprotokoll auch fiktive Konflikte erzeugen kann, wenn man zu grobe Sperrgranulate wählt.

Durch Verfeinerung der Sperrgranulate lassen sich solche fiktiven Konflikte beheben, wie die folgenden Protokoll-Ausschnitte für T3 und T4 belegen:

T3:	. . .
	Lock (GD, IS)
	Lock (GD (3,2), IS)
	Lock (GD (4,2), IS)
	Lock (GD (3,3), IS)
	Lock (GD (4,3), IS)
	Lock (TID_1, S)
	Lock (TID_2, S)
	. . .

T4:	. . .
	Lock (GD, IX)
	Lock (GD (5,4), IX)
	Lock (GD (4,4), IX)
	Lock (GD (5,3), IX)
	Lock (GD (4,3), IX)
	Lock (TID_3, X)
	Lock (TID_4, X)
	. . .

Durch die Anforderung von Anwartschaftssperren auf den GD-Einträgen kann auf dieser Ebene kein Konflikt auftreten. Allerdings müssen jetzt alle sich für die Prädikate P_3 und P_4 qualifizierenden Tupel explizit über ihre TIDs gesperrt werden, was die Menge der anzufordernden Sperren beträchtlich erhöhen kann.

5.3.2 Synchronisation für Modell 2

Bei der Synchronisation für Modell 1 ist nur eine Sperrhierarchie vorhanden, die Leser und Schreiber gleichzeitig benutzen. Als Sperrgranulate konnten das ganze Grid File (GD), GD-Einträge oder einzelne Tupel gewählt werden, was die Wahl eines Trade-offs zwischen Granulatgröße und Kosten des Sperrprotokolls erlaubte.

Die Synchronisation für Modell 2 gestattet im Prinzip die gleichen Freiheitsgrade, ist jedoch für Schreiber wesentlich komplexer, da diese alle existierenden Pfade zu einem Tupel sperren müssen, bevor mit der Modifikation begonnen werden kann. Auf Relationenebene ist dies noch sehr einfach: Lock (R, X). Bei feineren Granulaten kann sich zwischen einem Schreiber und einem Leser folgende Situation ergeben. Das Tupel mit TID_1 werde dabei von GD(i,j) überdeckt.

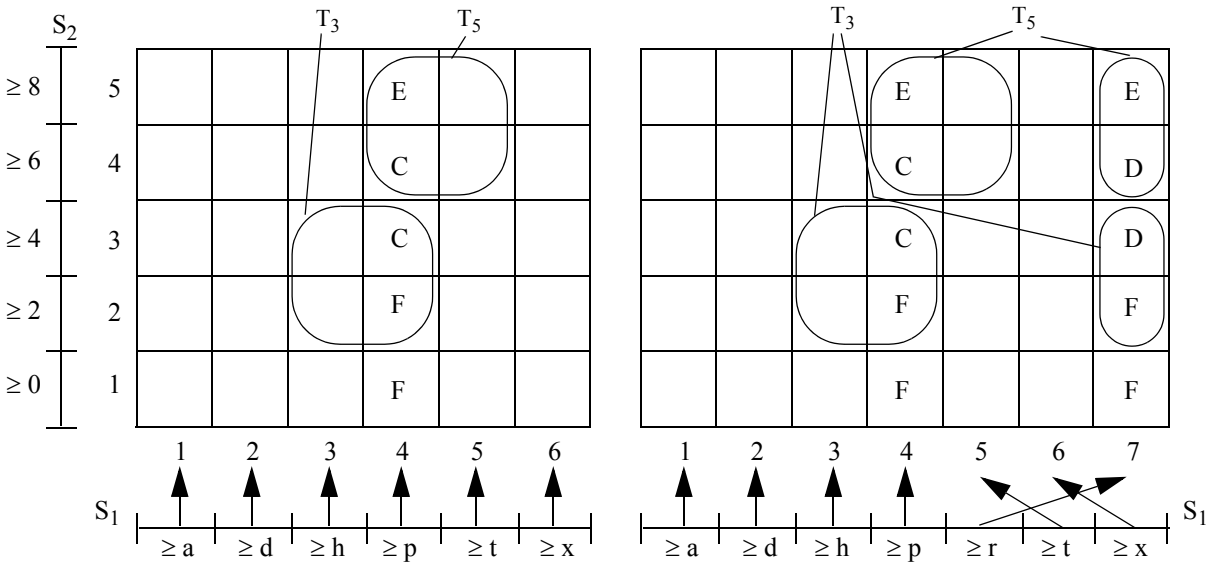
T1:	. . .
	Lock (R, IS)
	Lock (GD, IS)
	Lock (GD (i,j), S)
	. . .

T2:	. . .
	Lock (R, IX)
	Lock (TID_1, X)
	Lock (GD, IX)
	Lock (GD (i,j), IX)
	. . .

T1 habe die S-Sperre auf GD (i,j) erhalten, was T2 zum Warten zwingt. T2 besitzt bereits eine X-Sperre auf TID_1, was auf den ersten Blick auf eine Konfliktsituation hindeutet, da T1 das TID_1 implizit mit einer S-Sperre belegt hat. Da verabredungsgemäß T2 das betreffende Tupel erst ändern darf, wenn es alle Pfade zu ihm gesperrt hat, ist diese Situation nicht kritisch. T1 kommt entweder vor T2 in der Serialisierungsreihenfolge und liest den alten Wert von Tupel TID_1, oder es entsteht ein Deadlock durch weitere Sperranforderungen von T1.

5.4 Strukturmodifikationen im Grid Directory

Prinzipiell müssen alle Zugriffe auf das GD und auf die Skalierungsvektoren S_i bei Wertänderungen durch Kurzzeitsperren geschützt werden, damit beispielsweise T1 nicht einen GD-Eintrag liest, der gerade von T2 geändert wird. Dazu reicht ein Protokoll zum wechselseitigen Ausschluß von Lesern und Schreibern sowie von Schreibern und Schreibern aus, das nach verschiedenen Kriterien optimiert werden kann. Viel kritischer sind jedoch Strukturmodifikationen auf GD und den S_i, weil dadurch bei ungeeigneter Implementierung sich die Namen der GD-Einträge und damit die Namen von gewährten Sperren ändern können. Am folgenden Beispiel, das in Bild 11 illustriert ist, soll eine Lösung für dieses Problem aufgezeigt werden.



a) Ausgangssituation b) erweitertes GD

Bild 11: Splitt-Vorgang bei Bucket C

Die Änderungstransaktion T5 füge in Bucket C neue Sätze ein, die einen Splitt-Vorgang auslösen. Die Sätze von C werden auf die Buckets C und D neu aufgeteilt, so daß für C ≥ p und für D ≥ r gilt. Die Skalierungswerte von S_1 müssen um ≥ r ergänzt werden, d.h., die Spalten der S_1-Dimension von GD müssen um eine erweitert werden. Da die Abbildung der Skalierungswerte auf die GD-Einträge indirekt geschieht (siehe Bild 7), kann GD um die neue Spalte am Rand von S_1 erweitert werden. Eine Umbenennung der existierenden GD-Einträge ist nicht erforderlich, was wegen der gewährten Sperren auf GD-Einträgen auch katastrophal wäre. Lediglich die Einträge, die auf das durch den Splitt-Vorgang betroffene Bucket C verweisen (Bild 11a), müssen in der durch die Erweiterung entstan-

denen Spalte (7) angepaßt werden (auf Bucket D). Alle anderen Einträge behalten den Wert ihrer Ursprungsspalte. Wie in Bild 11b illustriert, müssen für die neu entstandenen Einträge ggf. Sperren angefordert werden (Erweiterung der gesperrten Mengen der GD-Einträge).

Entsprechend rückwirkungsfrei ist auch eine Schrumpfung von GD möglich. Wenn zwei Spalten zusammengelegt werden sollen, wird einfach ein Skalierungswert eines S_i entfernt, so daß eine Spalte nicht mehr angesprochen wird. Die ggf. vorhandenen Sperren auf den Spalteneinträgen können dann freigegeben werden, da niemand mehr die Einträge benutzen kann. Die tatsächliche Schrumpfung von GD kann dann einfach vorgenommen werden (z.B. durch ein Dienstprogramm), wenn keine Transaktionen auf dem GD aktiv sind.

Bei den zuletzt diskutierten Operationen zur Erweiterung oder Schrumpfung des GD und der S_i können parallele Operationen anderer Transaktionen nicht akzeptiert werden, da sich die Datenstruktur physisch verändert. Deshalb sind hierfür Kurzzeitsperren (Latches) zu erwerben, die in effizienter Weise den wechselseitigen Ausschluß gewährleisten. Eine mögliche Vorgehensweise wird in [Sa86] diskutiert.

5.5 Vergleich der Synchronisationsprotokolle

Um den Sperraufwand bei Nutzung der Skalierungsvektoren (Kap. 5.2) und der Zugriffshierarchie (Kap. 5.3) sowie das dabei erzeugte Sperrgranulat (Überdeckung) auf dem Grid Directory für den k-dimensionalen Fall zu ermitteln, führen wir noch folgende Größen ein:

- SI_i: zu sperrendes Skalierungsintervall von S_i
- n_i: Anzahl der Skalierungswerte s_{ij} in SI_i
- m_i: Anzahl der Skalierungswerte s_{ij} in S_i

Für eine Anfrage $P_t = P_{t1} \wedge P_{t2} \wedge ... \wedge P_{tk}$ von Transaktion T_t ist für das Sperren über die Skalierungsvektoren folgende Vorgehensweise erforderlich. Für jedes P_{ti} ist das kleinste SI_i zu ermitteln, das P_{ti} überdeckt. Damit sind die n_i Skalierungswerte s_{ij} festgelegt, die explizit zu sperren sind. Ist $SI_i \approx S_i$, so wird S_i explizit gesperrt. Als Sperraufwand für das Grid File erhalten wir also maximal:

$$C = 1 + k + \sum_{i=1}^{k} n_i$$

Die Sperrkosten wachsen somit additiv mit k und der Größe der Skalierungsintervalle. Das Sperrgranulat ergibt sich aus der Anzahl der überdeckten GD-Einträge. Jeder gesperrte Skalierungswert s_{ij} überdeckt m_i GD-Einträge, da dimensionsweise vorangegangen wird. Weiterhin ist zu beachten, daß das separate Sperren der einzelnen Dimensionen zu Mehrfachüberdeckungen führt (ODER-Verknüpfung):

$$Ü = \prod_{i=1}^{k} m_i - \prod_{i=1}^{k} (m_i - n_i)$$

Für k=3 ergibt sich damit

$$Ü = n_1m_2m_3 + n_2m_1m_3 + n_3m_1m_2 - n_1n_2m_3 - n_1n_3m_2 - n_2n_3m_1 + n_1n_2n_3$$

Anhand dieser Formeln wird noch einmal das große Sperrgranulat auf GD deutlich. Bereits bei einer Punktfrage mit k=3, $n_1 = n_2 = n_3 = 1$ und $m_1 = m_2 = m_3 = 10$ ergeben sich Ü = 271 überdeckte Einträge und C = 7 Sperranforderungen.

Das hierarchische Sperrprotokoll für das Grid Directory kann offensichtlich im Vergleich zum Sperren der Skalierungsvektoren bei Bereichsfragen zu einem höheren Aufwand beim Sperren führen, erzeugt aber wesentlich geringere Sperrgranulate (UND-Verknüpfung):

$$C = 1 + \prod_{i=1}^{k} n_i$$

Als Anzahl der überdeckten GD-Einträge erhalten wir:

$$Ü = \prod_{i=1}^{k} n_i$$

Mit obigen Zahlenwerten erhalten wir für die Punktfrage C = 2 und Ü = 1.

6. Zusammenfassung und Ausblick

Mehrdimensionale Zugriffspfadstrukturen mit ihren zentralen Eigenschaften wie symmetrischer Zugriff über k Attribute, k-dimensionale Clusterbildung, Adaption von schiefen Verteilungen durch erträgliche Belegungsgrade der Datenstruktur, balancierter Zugriff auf ≤ 3 Seiten usw. lassen sich unter den zusätzlichen Forderungen des Transaktionsparadigmas und der Effizienz auch im Mehrbenutzerbetrieb offensichtlich noch nicht für den Datenbank-Einsatz bereitstellen. Das wird auch dadurch bestätigt, daß existierende Datenbanksysteme bisher keine entsprechenden Strukturen anbieten. Nachfolgend sind noch einmal die wesentlichen Argument unserer Analyse zusammengefaßt.

Die bloße Simulation einer k-dimensionalen Zugriffspfadstruktur mit Hilfe von k eindimensionalen Indexstrukturen über separate B*-Bäume ist sicher die schlechteste Lösung, und zwar von den fehlenden operationalen Eigenschaften und auch vom Synchronisationsverhalten her.

Die Konkatenation von k Attributen zu einen Schlüssel (Superschlüssel) und ihre Realisierung durch eine B*-Baumstruktur zeigt bei der Bereichssuche kein zufriedenstellendes Verhalten. Für k=2 sind sicher Einsatzfälle denkbar, bei großem k ergeben sich jedoch sehr viele Schlüsselwerte mit den dazugehörigen TID-Listen, von denen ggf. viele fuer eine Partial-Match-Frage oder Bereichsfrage durch logische Mengenoperationen zusammenzufassen sind. Für die Indexsynchronisation sind jedoch effiziente Verfahren bekannt.

Das Grid File verkörpert als Struktur alle in Kap. 4.1 geforderten Eigenschaften und gestattet die Ausführung aller in Kap. 2 geforderten Operationen in effizienter Weise. Die bisher entwickelten Synchronisationsalgorithmen können vorerst in nicht allen Punkten überzeugen. Das in den Kap. 5.3 und 5.4 diskutierte hierarchische Sperrprotokoll

für Grid Directories erlaubt die maximal mögliche Parallelität auf GD, wie sie auf den von den Skalierungswerten bestimmten Granulaten GD (i,j) erzielt werden kann. Die Erweiterungs- und Schrumpfungsoperationen auf GD können aufgrund der vorgeschlagenen Indirektion bei der Abbildung der Skalierungswerte auf die GD-Dimensionen dynamisch vorgenommen werden, ohne den Transaktionsbetrieb wesentlich zu stören. Lediglich exklusive Kurzzeitsperren auf die physischen Datenstrukturen GD und S_i sind erforderlich.

Leider war es jedoch aus inhärenten Gründen der Strukturkomplexität nicht möglich, ebenso effiziente und effektive Sperrprotokolle für ein Grid File zu entwickeln, wie es für eindimensionale Indexstrukturen auf der Basis von B*-Bäumen [HR91] gelang. Die Anforderung von Mengen von Sperren auf der GD (i,j)-Ebene für eine Such- oder Änderungsoperation ist schwerfällig und möglicherweise sehr ineffizient. Das dynamische Verhalten solcher Sperrprotokolle ist schwer überschaubar. Über das Blockierungs- und Deadlock-Verhalten liegen keine Erfahrungen vor, es ist jedoch insgesamt mit einem ungünstigen Leistungsverhalten zu rechnen. Möglicherweise fehlen im Moment neue Ideen, wie symmetrischer Zugriff mit beliebigen Prädikaten mit einem hierarchischen Sperrprotokoll mit Granulaten variabler Größe miteinander zu vereinbaren sind.

7. Literaturverzeichnis

AU79 Aho, A. V., Ulman, J. D.: Optimal Partial Match Retrieval when Fields are Independently Specified, in: ACM TODS, Vol. 4, No. 2, 1979, pp. 168-179.

BCE77 Blasgen, M.W., Casey, R.G., Eswaran, K.P.: An Encoding Method for Multifield Sorting and Indexing, in: Comm. ACM, Vol. 20, No. 11, Nov. 1977, S. 874-876.

Be75 Bentley, J.L.: Multi-Dimensional Search Trees Used for Associative Searching, in: Comm. ACM, Vol. 18, No. 9, 1975, pp. 509-517.

BS77 Bayer, R., Schkolnick, M.: Concurrrency of Operations on B-Trees, in: Acta Informatica, Vol. 9, No. 1, p. 1-21, 1977.

Co70 Codd, E.F.: A Relational Model of Data for Large Shared Data Banks, in: Comm. ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

Co79 Comer, D. The Ubiquitous B-Tree, in: ACM Surveys, Vol. 12, No. 2, 1979, pp. 121-137.

Da81 Date, C.J.: Referential integrity, in: Proc. 7th Int. Conf. on VLDB, 1981, pp. 2-12.

EGLT76 Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database System, in: Comm. ACM, Vol. 19, No. 11, Nov. 1976, pp. 624-633.

FB74 Finkel, R.A., Bentley, J.L.: Quad Trees: A Data Structure for Retrieval on Composite Keys, in: Acta Informatica, Vol. 4, No. 1, 1974, pp. 1-9.

Fr87 Freeston, M. W. : The BANG File : a New Kind of Grid File, in: Proc. SIGMOD Conference 1987, ACM, San Francisco, pp. 260-269.

GLPT76 Gray, J.N., et al.: Granularity of Locks and Degrees of Consistency in a Large Shared Data Base, in: Modelling in Data Base Management Systems, North-Holland, 1976, pp. 365-394.

Gr78 Gray, J.N.: Notes on Data Base Operating Systems, in: Lecture Notes Computer Science, 60, Operating systems: An advanced course, Springer-Verlag, 1978, pp. 393-481.

Gu84 Guttman, A. : R-Trees: a Dynamic Index Structure for Spatial Searching, in: Proc. SIGMOD Conference 1984, ACM, Boston, pp. 47-57.

Gü89 Günther, O. : The Cell Tree: an Object-oriented Index Structure for Geometric Databases, in: Proc. IEEE 5th Int. Conference on Data Engineering, Los Angeles, Ca., Feb. 1989.

Hä78 Härder, T.: Implementierung von Datenbanksystemen, Hanser-Verlag, 1978.

HR83 Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, in: ACM Com-puting Surveys, Vol. 15, No. 4, 1983, pp. 287-317.

HR91 Härder, T., Rahm, E.: Zugriffspfad-Unterstützung zur Sicherung der Relationalen Invarianten, Interner Bericht, Universität Kaiserslautern, Dez. 1991.

Kü83 Küspert, K.: Storage Utilization in B*-Trees with a Generalized Overflow Technique, in: Acta Informatica, Vol. 19, No. 1, April 1983, pp. 35-55.

Mo90 Mohan, C.: ARIES/KVL: A Key-Values Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes, in: Proc. 16th VLDB Conference, Brisbane, Australia, August 1990.

NHS84 Nievergelt, J., Hinterberger, H., Sevcik, K.C.: The Grid File: An Adaptable, Symmetric Multikey File Structure, in: ACM TODS, Vol. 9, No. 1, 1984, pp. 38-71.

Ro81 Robinson, J. T.: The k-d-B-Tree: a Search Structure for Large Multidimensional Dynamic Indexes, in: Proc. SIGMOD Conference 1981, ACM, New York, pp. 10-18.

Sa86 Salzberg, B.: Grid File Concurrency, in: Information Systems, Vol. 11, No. 3, 1986, pp. 235-244.

Sh90 Shaw, P.: Database Language Standards: Past, Present, Future, in: Database Systems of the 90s, A. Blaser (ed.), LNCS 466, Springer-Verlag, 1990, pp. 50-88.

SQL2 ISO/IEC JTC1/SC21 Information Retrieval, Transfer and Management for OSI: International Standard IOS/IEC 9075:1992, Revised Text of CD 9075.2, Information Technology - Database Languages - SQL2, for DIS registration and letter ballot, 10.04.1991, 522 pp.

SQL3 X3H2-91-183 DBL-KAW-003 ISO/IEC JTC1/SC21/WG3 N1223: ISO/ANSI Working Draft - Database Language SQL3, 07.1991, 772 pp.

SRF87 Sellis, T., Roussopoulos, N. Faloutsos, C.: The R+-Tree: a Dynamic Index for Multi-dimensional Objects, in: Proc. 13th VLDB Conference, Brighton, U.K., August 1987, pp. 507-518..

.