ZRI-Bericht 2/92, University Kaiserslautern, 1992

Zugriffspfad-Unterstützung zur Sicherung der Relationalen Invarianten

Theo Härder
Erhard Rahm

2/92

Univ. Kaiserslautern
Fachbereich Informatik
Postfach 3049
6750 Kaiserslautern

Januar 1992

Zugriffspfad-Unterstützung zur Sicherung der Relationalen Invarianten

Theo Härder, Erhard Rahm
Universität Kaiserslautern
FB Informatik, Postfach 3049
6750 Kaiserslautern

E-mail: {haerder | rahm} @informatik.uni-kl.de

Januar 1992

ÜBERBLICK

Das Relationenmodell garantiert seinen Benutzern Integritätszusicherungen für Entitätsintegrität und Referentielle Integrität. Datenbanksysteme, die dem SQL2-Standard genügen wollen, müssen die Einhaltung dieser fundamentalen Integritätsregeln, der sogenannten Relationalen Invarianten, gewährleisten. Das bedeutet, daß bei den entsprechenden Modifikationsoperationen Primärschlüssel- und Fremdschlüsselbedingung explizit zu überprüfen sind. In diesem Aufsatz werden für Schlüssel, die aus einfachen Attributen bestehen, Verfahren untersucht, um diese Integritätsprüfungen effizient abwickeln zu können. Dabei werden Zugriffspfade betrachtet, die sich durch B*-Bäume und ihre Varianten implementieren lassen. Von besonderer Wichtigkeit wurde die Synchronisation auf solchen Indexstrukturen angesehen, da sich dort Behinderungen von parallelen Transaktionen äußerst drastisch auf das Systemverhalten auswirken. Eine Analyse des für die Integritätsüberprüfungen anfallenden Mehraufwands beschließt unsere Untersuchungen.

1. Einleitung

Der SQL2-Standard definiert für das Relationenmodell weitreichende Integritätszusicherungen [EGLT76] für Entitätsintegrität und Referentielle Integrität [Sh90], die vom DBMS zu gewährleisten sind (system-enforced integrity). Damit werden endlich Ankereigenschaften des Relationenmodells explizit spezifiziert und für DBMS verbindlich gemacht, ohne die das Relationenmodell eigentlich gar nicht denkbar ist und die schon in den ersten Veröffentlichungen der Relationalen Idee [Co70] zumindest implizit existierten. Seit 10 Jahren gibt es verstärkt Versuche, diese Eigenschaften zu formalisieren und damit auch in ihrer Semantik klarer festzulegen [Da81].

Durch die Standardisierung dieser Integritätszusicherungen werden Analysen ihrer Auswirkungen auf das Systemverhalten immer dringlicher. Noch wichtiger erscheint die optimale Unterstützung dieser häufig anfallenden Integritätskontrollen auf allen Systemebenen. Insbesondere betrifft das die Zugriffspfad-Unterstützung und die Synchronisation der betreffenden Operationen im Mehrbenutzerbetrieb.

Da für die Überprüfung von Schlüsselbedingungen (z.B. UNIQUE) wertabhängige Zugriffe auf große Datenmengen erforderlich sind, wurde bisher in verschiedenen Systemen, die solche Kontrollen vornahmen, das Anlegen eines Index für jeden in Frage kommenden Schlüssel erzwungen, oder umgekehrt ausgedrückt, das System akzeptierte nur Schlüsseldefinitionen, wenn vorher auf dem zugehörigen Attribut ein Index spezifiziert wurde. Diese Vorgehensweise zeigt, daß die verbindlich zu überprüfenden Integritätsbedingungen dafür sorgen werden, daß in allen Anwendungen große Mengen an Zugriffspfadstrukturen anzulegen und zu warten sind. Deshalb gilt es, solche Zugriffspfadstrukturen und ihre Operationen im Hinblick auf die zu erwartende Arbeitslast zu entwerfen und zu optimieren.

Nach der Einführung von Modellannahmen für die Überprüfung von Integritätsbedingungen (UNIQUE, Referentielle Integrität) diskutieren wir die dabei anfallenden kritischen Operationen. Das erlaubt uns, konkretere Aussagen über die Zugriffspfadunterstützung zu machen. Als wichtigste Struktur zur Indeximplementierung qualifiziert sich dabei der B*-Baum, der als verallgemeinerte Zugriffspfadstruktur mehrere logische Indexstrukturen auf einmal realisieren kann [Hä78]. Eine zentrale Rolle für die Leistungsfähigkeit von Indexoperationen im transaktionsgeschützten Mehrbenutzerbetrieb spielt offensichtlich die Synchronisation in Indexstrukturen. Deshalb werden hierfür Sperrprotokolle entwickelt und schrittweise optimiert, wobei Konsistenzebene 3 unterstellt wird. Vereinfachungen führen dann auf Konsistenzebene 2.

Schließlich helfen analytische Leistungsmodelle den zu erwartenden Aufwand an logischen und physischen Seitenzugriffen für verschiedene Zugriffspfadstrukturen abzuschätzen.

Die Ergebnisse dieser Untersuchungen zusammen mit den zugeschnittenen Sperrprotokollen für Indexstrukturen erlaubten die Auswahl und Empfehlung von bestimmten Zugriffspfadstrukturen für Aufgaben der Integritätssicherung.

2. Modellannahmen

SQL2 erlaubt für das Relationenmodell die Spezifikation von mehreren Schlüsselkandidaten mit Hilfe der UNIQUE-Option. Jeder Schlüsselkandidat besteht aus einem Attribut oder einer minimalen Gruppe von Attributen; er gewährleistet die Eindeutigkeit eines Tupels innerhalb einer Relation. Ein Schlüsselkandidat ist dann (aufgrund semantischer Kriterien) durch die Angabe in der PRIMARY KEY-Klausel als Primärschlüssel festzulegen, was gleichzeitig implizit bedeutet, daß für dieses Attribut oder diese Attributgruppe keine NULL-Werte erlaubt sind, was also der Attributklausel “NOT NULL” entspricht.

Zur Darstellung von Beziehungen zwischen Tupeln verschiedener Relationen oder innerhalb einer Relation dient das Konzept des Fremdschlüssels, der wie sein zugehöriger Schlüsselkandidat aus einem oder mehreren Attributen aufgebaut sein kann, die auf den entsprechenden Wertebereichen definiert sein müssen. Der Wert eines Fremdschlüssels wird dazu benutzt, um eine Beziehung zu einem Tupel in der referenzierten Relation darzustellen, d.h., er muß in mindestens einem Tupel der referenzierten Relation im zugehörigen Schlüsselkandidaten auch vorkommen.

SQL2 fordert, daß das Datenbanksystem die Integrität von Primär- und Sekundärschlüsseln, die sogenannten Relationalen Invarianten, gewährleistet. Das bedeutet, daß bei allen Aktualisierungsoperationen die Eindeutigkeitsbedingung beim Primärschlüsselwert (und bei allen mit UNIQUE definierten Attributwerten) sowie die Referentielle Integrität von Fremdschlüsselwerten überprüft werden muß.

In diesem Aufsatz nehmen wir an, daß alle Schlüsselkandidaten durch einfache Attribute dargestellt werden und definierte Werte besitzen (“NOT NULL”). Der Einsatz zusammengesetzter Attribute wird in [Hä92] untersucht. Weiterhin referenzieren alle Fremdschlüssel einen Primärschlüssel.

Als Basisbeispiel für die Diskussion der Kosten zur Integritätssicherung wählen wir zwei Relationen. Es können beispielsweise die Relationen ABT und PERS herangezogen werden:

ABT (ABTNR, MGR, ANAME, ...)
PERS (PNR, SVNR, ..., ANR, ...)

ABTNR und MGR sowie PNR und SVNR seien Schlüsselkandidaten. ANR sei Fremdschlüssel in Bezug auf ABTNR.

Verallgemeinert läßt sich das Beispiel formal folgendermaßen schreiben:

```
CREATE  DOMAIN    A1_typ AS ...
. . .
CREATE  DOMAIN    Bm_typ AS ...
CREATE  TABLE V (
    A1    A1_typ     PRIMARY KEY,
    A2    A2_typ    UNIQUE NOT NULL,
    ....
    Av    Av_typ    UNIQUE NOT NULL,
    ...
    An    An_typ)
```

```
CREATE  TABLE S (
    B1    B1_typ    PRIMARY KEY,
    B2    B2_typ    UNIQUE NOT NULL,
    ...
    Bs    Bs_typ    UNIQUE NOT NULL,
    ...
    A1    A1_typ,
    ...
    Bm    Bm_typ,
    CONSTRAINT  SFK FOREIGN KEY (A1)
                REFERENCES V (A1),
    ON UPDATE ...,
    ON DELETE ...)
```

Die Relationen V (Vater) und S (Sohn) haben folglich v und s Schlüsselkandidaten. S hat weiterhin einen Fremdschlüssel.

An diesem Basisbeispiel sollen die möglichen Aktualisierungsoperationen und ihr Aufwand zur Einhaltung der Relationalen Invarianten diskutiert werden. Komplexere Schemata mit mehreren Vater-Relationen für die Sohn-Relation S oder mehreren Sohn-Relationen für die Vater-Relation V lassen sich durch einfache Erweiterung des Basisbeispiels behandeln.

Das Aufsuchen von Tupeln einer Relation kann aufgrund von Werten eines Schlüsselkandidaten oder eines Fremdschlüssels erforderlich sein. Wir unterstellen hier die Existenz von Indexstrukturen für jeden Schlüsselkandidaten und Fremdschlüssel (z.B. I_V(A1), d.h. auf dem Attribut A1 der Relation V), so daß ein Index-Scan zur Lokalisierung des Zieltupels erfolgen kann. Das Fehlen einer Indexstruktur würde einen Relationen-Scan für das Aufsuchen erzwingen.

3. Diskussion der Operationen anhand des Basisbeispiels

Es sollen zunächst nur Reihenfolge und Art der Operationen bestimmt werden. Zur Ermittlung der Kosten muß das Betrachtungsmodell später verfeinert werden.

Die Relationalen Invarianten in Bezug auf die Relationen V und S können jeweils durch Einfügen, Ändern und Löschen eines Tupels in den betreffenden Relationen verletzt werden. Die Feststellung einer solchen Integritätsverletzung soll zusammen mit der auslösenden Operation bewerkstelligt werden (IMMEDIATE). Wird eine Integritätsverletzung erkannt, so ändert sich der DB-Zustand nicht, und es wird eine Fehlermeldung zurückgeliefert; sonst wird die Operation vollständig ausgeführt (Statement Atomicity).

Im Mehrbenutzerbetrieb müssen alle Operationen synchronisiert werden. Konsistenzebene 3 gewährleistet die Wiederholbarkeit aller Lesevorgänge einer Transaktion, d.h. einen logischen Einbenutzerbetrieb. Die dabei entstehenden Ablauffolgen von parallelen Transaktionen sind serialisierbar. Unterstellt man im konkreten Fall ein Sperrkonzept, so müssen alle Lese- und Schreibsperren auf den von der

Transaktion referenzierten Objekten bis zu Abort oder Commit (EOT) gehalten werden (lange S- oder X-Sperre). Da beim Überprüfen der Relationalen Invarianten zusätzliche (Hilfs-)Objekte referenziert werden müssen, ist zu fragen, ob dabei schwächere Synchronisationsmaßnahmen ausreichend sind. Wenn beispielsweise eine Sperre nur für die Dauer einer Operation oder eines Seitenzugriffs benötigt wird, so sprechen wir von einer kurzen Sperre. Solche kurzen Sperren sind vor allem auf Indexstrukturen und anderen Hot-Spot-Elementen höchst wünschenswert.

3.1 Operationen auf Tupeln in der Sohn-Relation S

3.1.1 Einfügen eines Tupels in S

Gemäß den Forderungen der Relationalen Invarianten und den definierten Optionen ist zu prüfen, ob die Werte von B1, ..., Bs eindeutig sind und einen zulässigen Wert besitzen und ob das durch S.A1 referenzierte Tupel in V existiert. Der interne Ablauf dieser Operation in Transaktion T1 hat folgende Struktur:

1. Setze für das S-Tupel eine lange X-Sperre und füge es in die S-Relation ein, was die Manipulation von Freispeicher- und Adressierungsinformation erfordert.
2. Greife auf $I_S(Bi)$, i = 1...s, zu, überprüfe die UNIQUE-Option und füge Index-Eintrag ein. Setze eine lange X-Sperre für jeden eingefügten Schlüsselwert.
3. Füge S.A1 in $I_S(A1)$ ein und setze lange X-Sperre für den eingefügten Schlüsselwert.
4. Falls weitere $I_S(Bi)$ für $s < i \leq m$ existieren, aktualisiere sie und halte lange X-Sperren auf den betreffenden Schlüsselwerten.
5. Überprüfe die Existenz des Wertes von S.A1 durch Zugriff auf $I_V(A1)$. Setze dabei nur eine kurze S-Sperre.
6. Sind weitere Vater-Relationen vorhanden, so ist Schritt 5 analog anzuwenden.

Bei der Überprüfung der Referentiellen Integrität auf $I_V(A1)$ ist eine kurze S-Sperre ausreichend. Die Referentielle Integrität könnte nur durch eine Transaktion T2, die den betreffenden Eintrag in $I_V(A1)$ löscht, gefährdet werden. T2 hat dann aber ihrerseits die Referentielle Integrität auf $I_S(A1)$ zu warten, was wegen der gesetzten langen X-Sperre zu einer Wartesituation bis zu EOT(T1) oder zu einem Deadlock führt. T2 ist also bestenfalls nach T1 serialisierbar.

Die Schritte 1-3 sind vor Schritt 4 auszuführen. Dadurch werden alle Wege, die Informationen über das noch nicht freigegebene S-Tupel bereithalten, bis EOT(T1) gesperrt.

Soll eine symmetrische Referentielle Integrität, beispielsweise in Form der PENDANT-Option [SQL3] oder von Kardinalitätsrestriktionen unterstützt werden, sind weitere Prüfungen, die ggf. verzögert ausgeführt werden müssen, erforderlich. Diese Aussage gilt auch für die folgenden Operationen.

3.1.2 Ändern eines Tupels in S

Lediglich das Ändern des Wertes eines Fremdschlüssels oder eines UNIQUE-Attributes führt zu zusätzlichen Prüfungen. Alle Pfade zum Tupel müssen vor Beginn der Änderung gesperrt sein.

1. Sperre alle Pfade zum betroffenen S-Tupel und das S-Tupel selbst mit langen X-Sperren und ändere die Attributwerte im Tupel.
2. Überprüfe die UNIQUE-Eigenschaft durch die betroffenen I_S(Bi) und modifiziere die Einträge. Aktualisiere sonstige I_S(Bi) .
3. Überprüfe die Existenz des neuen Wertes von S.A1 durch Zugriff auf I_V(A1). Setze dabei nur eine kurze S-Sperre.
4. Sind weitere Fremdschlüssel betroffen, verfahre analog zu 3.

3.1.3 Löschen eines Tupels aus S

Diese Operation ist unkritisch; sie kann auf die Relation S beschränkt werden. Die Reihenfolge beim Löschen von Index-Einträgen und S-Tupel ist hier wichtig. Das Tupel darf nicht gelöscht sein, bevor nicht alle Pfade zu ihm gesperrt sind.

1. Finde die Referenzen auf S in den I_S(Bi), i ≤ 1..m, setze lange X-Sperren und lösche die Einträge.
2. Setze lange X-Sperre für das S-Tupel und lösche es. Dazu sind Modifikationen in der Freispeicher- und Adressierungsinformation erforderlich.

3.2 Operationen auf Tupeln in V

3.2.1 Einfügen eines Tupels in der Vater-Relation V

Das Einfügen eines V-Tupels erfordert lediglich Operationen und Sperren auf der V-Relation und ihren Indexstrukturen I_V(Ai), i ≤ 1 .. n. Der interne Ablauf entspricht dem von 3.1.1, 1-3, in analoger Weise.

3.2.2 Ändern eines Tupels in V

In der Fremdschlüssel-Klausel einer TABLE-Definition (S-Relation) kann die Option

[ON UPDATE {SET NULL | CASCADE | SET DEFAULT}]

angegeben werden. Wenn in V der Wert von V.A1 von “a1alt” nach “a1neu” geändert wird, so können die Relationalen Invarianten davon betroffen sein. Die Option SET NULL würde bei allen S-Tupeln mit S.A1=a1alt eine Änderung nach S.A1=NULL veranlassen. CASCADE löst bei diesen referenzierenden Tupeln eine Folgeänderung nach S.A1=a1neu aus. Schließlich ändert SET DEFAULT den entsprechen-

den S.A1-Wert dieser Tupel auf einen DEFAULT-Wert, den der Benutzer für ein Attribut einer Tabelle und/oder zu einem Wertebereich (DOMAIN) für Attribute spezifiziert hat. Wird die Option ON UPDATE ganz weggelassen, so wird die Änderungsoperation auf dem V-Tupel zurückgewiesen, wenn in der S-Relation Tupel mit S.A1=a1alt vorhanden sind. Dies entspricht der Option RESTRICTED, die in anderen Sprachentwürfen verwendet wird.

Ist V.A1 von der Änderung nicht betroffen, so kann die Änderungsoperation analog zu 3.1.2, 1-2, durchgeführt werden. Wird V.A1 im entsprechenden Tupel geändert, so wird folgendermaßen verfahren:

1. Suche Tupel über I_V(A1), überprüfe UNIQUE-Option und lokalisiere V-Tupel. Sperre Tupel und Einträge in Index mit langen X-Sperren. Ändere die betroffenen Einträge nach a1neu.
2. Falls die Option RESTRICTED verletzt ist, setze gesamte Änderungsoperation zurück. RESTRICTED kann durch Zugriff auf I_S(A1) überprüft werden. Es ist dazu nur eine kurze S-Sperre erforderlich.
3. Bei der Option SET NULL oder SET DEFAULT werden alle S-Tupel mit S.A1=a1alt über I_S(A1) aufgesucht und auf S.A1=NULL oder S.A1=Default gesetzt. I_S(A1) ist entsprechend zu modifizieren, d.h. er muß einen NULL- oder einen Default-Wert besitzen. Alle geänderten Einträge erhalten lange X-Sperren.
4. Die Option CASCADE erzwingt einen Ablauf der Änderungsoperation ähnlich wie 3, wobei in den S-Tupeln und in I_S(A1) der Wert a1neu eingesetzt wird.

3.2.3 Löschen eines Tupels in V

Auch für diese Operation kann in der Fremdschlüssel-Klausel einer TABLE-Definition (S-Relation) eine spezielle Option angegeben werden:

[ON DELETE {SET NULL | CASCADE | SET DEFAULT}]

Analog zu 3.2.2 ergibt sich folgender Operationsablauf:

1. Suche Tupel in allen existierenden I_V(Ai) und lokalisiere Tupel. Sperre I_V-Einträge und Tupel mit langen X-Sperren und lösche sie.
2. Falls die Option RESTRICTED angewendet werden soll, überprüfe sie über I_S(A1) mit kurzer S-Sperre. Ist RESTRICTED verletzt, so ist Schritt 1 rückgängig zu machen.
3. SET NULL und SET DEFAULT entsprechen 3.2.2, Schritt 3, was einem Löschen der Beziehung gleichkommt.
4. Bei CASCADE werden die zugehörigen S-Tupel über I_S(A1) aufgesucht und gelöscht. Das impliziert auch das Entfernen aller entsprechenden Referenzen aus allen I_S(Bi). Dabei sind lange X-Sperren erforderlich.

Eine Relation kann mehrmals in einer Vaterrolle und/oder in einer Sohnrolle auftreten. Bei Aktualisierungsoperationen sind deshalb pro Vaterrolle die Operationsfolgen nach 3.4 - 3.6 und pro Sohnrolle die nach 3.1 - 3.3 entsprechend anzuwenden.

SQL erlaubt auch die Spezifikation mengenorientierter Aktualisierungsoperationen. Hierbei ist im Prinzip für jedes Tupel der spezifizierten Menge die entsprechende Ablaufstruktur nach 3.1 - 3.6 erforderlich; es können jedoch Optimierungen bei der Anfragebearbeitung erfolgen (Reihenfolge von Selektionen oder Join, Nutzung von Clusterbildungen oder Lokalität auf Indexstrukturen). Kann für ein Tupel der Menge die Operation nicht erfolgreich ausgeführt werden, so erfordert das Prinzip der Statement Atomicity das Rücksetzen der gesamten Mengenoperation.

4. Zugriffspfad-Unterstützung für die Einhaltung der Relationalen Invarianten

Die Betrachtung der Ablaufstruktur von Aktualisierungsoperationen nach SQL2 hat gezeigt, daß einer effizienten Zugriffspfad-Unterstützung ein ganz erhebliches Gewicht zukommt. Das betrifft die Überprüfung der UNIQUE-Option und die der verschiedenen Aspekte der Referentiellen Integrität in den S- und V-Relationen. Insbesondere müssen geeignete Indexstrukturen eingesetzt werden, die sowohl den schnellen, direkten Zugriff auf Schlüsselkandidaten und Fremdschlüssel unterstützen, als auch kostengünstige Änderungsoperationen (Strukturmodifikationen, Logging) erlauben.

4.1 Operationen auf Indexstrukturen

Wenn eine Indexstruktur in einem DBMS definiert wird, so soll sie sicher nicht ausschließlich zur Überprüfung der Relationalen Invarianten dienen, sondern auch für die Anfrageoptimierung bei verschiedenen Anfragetypen benutzt werden [Hä78a]. Ihre flexible Einsatzfähigkeit bei allen Suchvorgängen erfordert neben der Unterstützung tupelbezogener Operationen auch die der Bereichssuche, wobei der Bereich durch Schlüsselwerte (oder Präfixe) eingegrenzt werden kann und in beiden Richtungen (NEXT/ PRIOR) durchsucht werden kann. Weiterhin sind spezielle Synchronisationsprotokolle, die Sperrdauer und Sperrgranulat minimieren, von großer Wichtigkeit, da diese Indexstrukturen Betriebsmittel hoher Nutzungshäufigkeit sind (high-traffic data elements) [SG88,LY81,ESS91].

Unsere Analyse läßt in einfacher Weise erkennen, daß bestimmte Indexoperationen und -operationsfolgen besonders häufig zu erwarten sind. Danach sind die wichtigsten Indexoperationen

- das Aufsuchen eines Schlüssels in der Indexstruktur, z.B. in I_V(Ai) oder I_S(Bj), zur Prüfung der UNIQUE-Option, zur Ermittlung der Tupeladresse oder zum Ein-/Ausfügen von Einträgen.
- der sukzessive Zugriff auf alle Schlüssel, die den gleichen Wert besitzen oder in einem vorgegebenen Wertebereich liegen.

- das Aufsuchen eines Fremdschlüssels und seines dazugehörigen Primärschlüssels oder eines Primärschlüssels mit allen seinen Fremdschlüsseln; in diesen Fällen ist ein Schlüsselwert W_1 in $I_S(A1)$ und nachfolgend in $I_V(A1)$ zu lokalisieren oder umgekehrt erst in $I_V(A1)$ und danach in $I_S(A1)$.

Diese Indexoperationen lassen sich durch folgende Grundoperationen realisieren:

- Fetch: Es wird überprüft, ob sich ein gegebener Schlüsselwert im Index befindet. Im Erfolgsfall wird der Schlüssel mit zugehörigem TID zurückgeliefert.
- Fetch Next: Beim ersten Fetch-Next-Aufruf wird eine Startbedingung (=, >, ≥, ≠) mit Schlüsselwert oder Schlüssel-Präfix spezifiziert und ein Bereichs-Scan eröffnet. Jeder Aufruf liefert das nächste Schlüsselwert-TID-Paar, das die Bereichsbedingung erfüllt, zurück. Dabei läßt sich ein Stoppkriterium (Stoppschlüssel und Stoppbedingung (<, ≤, =, ≠)) spezifizieren.
- Fetch Prior: Beim ersten Fetch-Prior-Aufruf wird eine Startbedingung (=, <, ≤, ≠) mit Schlüsselwert oder Schlüssel-Präfix spezifiziert und ein Bereichs-Scan eröffnet. Jeder Aufruf liefert das vorherige Schlüsselwert-TID-Paar, das die Bereichsbedingung erfüllt, zurück. Dabei kann ein Stoppkriterium (>, ≥, =, ≠) spezifiziert werden.
- Insert: Ein gegebener Verweis (Schlüsselwert, TID) wird in den Index eingefügt. Bei der Option UNIQUE für den Index sind entsprechende Prüfungen vorgesehen.
- Delete: Ein gegebener Verweis (Schlüsselwert, TID) wird gelöscht.

Zur Verallgemeinerung der Suche lassen sich die Fetch-Operationen durch einfache Suchausdrücke (simple search arguments) erweitern. Sie werden nicht wie die Bereichsbedingungen an den Schlüsselwerten der Indexstruktur überprüft, sondern durch Zugriff auf die Attributwerte des referenzierten Tupels. Dabei werden nur Tupel, die die Bereichsbedingung sowie das Suchargument erfüllen, an den Benutzer zurückgeliefert.

Bei der Indexoption NONUNIQUE ist für die Operationen Fetch, Fetch Next und Fetch Prior die Variante “TID-Liste” sinnvoll, bei der nicht (Schlüsselwert, TID), sondern (Schlüsselwert, TID-Liste), also alle zum Schlüsselwert zugehörigen TIDs zurückgeliefert werden.

Änderungsoperationen (Update) veranlassen das Löschen und das Einfügen eines Verweises. Weitere Operationen wie Create Index und Destroy Index sind isoliert (Relationensperre) auszuführen; sie brauchen in unserem Zusammenhang nicht betrachtet zu werden.

Um eine Bewertung dieser Operationen im Zusammenhang mit der Integritätssicherung vornehmen zu können, ist für konkrete Implementierungen von Indexstrukturen eine Analyse des Leistungsverhaltens durchzuführen. Dabei spielen Fragen der Lokalität von Operationen, der Externspeicherzugriffe und der effizienten Nutzung im Mehrbenutzerbetrieb eine besondere Rolle. Es ist offensichtlich, daß eine balancierte Zugriffsstruktur mit dynamischer Reorganisation gefordert werden muß. Weiterhin sind der Synchronisationsalgorithmus bei Zugriffen, Strukturänderungen sowie Aktualisierungen der Indexstruktur dominierende Bewertungsgrößen bei der Auswahl der am besten geeigneten Indexstruktur.

4.2 B*-Baum als Zugriffsstruktur

Hash-Verfahren oder Erweiterbares Hashing [Fa79] können herangezogen werden, um schnell auf die Einträge einer Indexstruktur (Indextabelle) zuzugreifen. Die Operationen Fetch Next und Fetch Prior zum sortiert sequentiellen Aufsuchen aller Werte oder zur Bereichssuche lassen sich allerdings dann nicht unterstützen. Mehrwegbäume dagegen versprechen eine schnelle und effiziente Ausführung aller Grundoperationen.

Als wichtigstes Verfahren zur Implementierung der Zugriffsstruktur soll deshalb der bekannte B-Baum [BMc72] untersucht werden. B-Bäume sind in Bezug auf ihre Knotenstruktur (Seiten) ausgeglichene Mehrwegbäume. Ihr Knotenformat (2-4 KB) gewährleistet einen großen Verzweigungsgrad (fan-out) der Baumstruktur; ihre Modifikationsalgorithmen nehmen nur lokale Änderungen vor und garantieren eine Belegung $\beta \geq 50\%$ sowie eine balancierte Baumstruktur geringer Höhe.

Jeder Eintrag im B-Baum ist aus drei Elementen (K_i, D_i, P_i) zusammengesetzt (Schlüssel, (Verweis auf) Datensatz, Zeiger zum zugehörigen Unterbaum). Ein solcher Eintrag spielt in einem B-Baum zwei ganz verschiedene Rollen:

- Der zum Schlüssel K_i gehörende Datensatz D_i oder ein Verweis darauf werden gespeichert.
- Der Schlüssel K_i dient als Wegweiser im Baum.

Eine Indexstruktur soll neben dem direkten Zugriff auch den sortiert sequentiellen Zugriff (Image der Relation in Schlüsselreihenfolge [Hä78]) unterstützen. Diese sortierte Schlüsselfolge kann zwar durch einen Durchlauf in symmetrischer Ordnung (Verallgemeinerung des Durchlaufs in Zwischenordnung) erreicht werden, sie erfordert aber die Einbeziehung ganzer Baumpfade von der Wurzel bis zu einem Blatt während des gesamten Durchlaufs, so daß das Zugriffsverfahren und die notwendigen Sperrprotokolle sehr komplex werden und parallele Operationen erheblich einschränken. Außerdem wird das fan-out durch die D_i in Wurzel und Zwischenknoten eingeschränkt und dadurch die zu erwartende Baumhöhe oft größer als erforderlich.

Deshalb wollen wir den B-Baum nicht weiter detaillieren, sondern seine wichtigste Variante, den B*-Baum, betrachten. Der Hauptunterschied liegt in der Rolle der Einträge von inneren Knoten [Co79].

In B*-Bäumen wird in inneren Knoten nur die Wegweiser-Funktion ausgenutzt, d.h., es sind nur (K_i, P_i) als Einträge zu führen. Die zu speichernden Informationen (K_i, D_i) werden in den Blattknoten abgelegt. Dadurch ergibt sich für einige K_i eine redundante Speicherung. Die inneren Knoten bilden also einen Index (index part), der einen schnellen direkten Zugriff zu den Schlüsseln gestattet. Die Blätter enthalten alle Schlüssel mit ihren zugehörigen Daten in Sortierreihenfolge. Durch Verkettung aller Blattknoten (sequence set) läßt sich eine effiziente sequentielle Verarbeitung erreichen, die beim B-Baum einen umständlichen Durchlauf in symmetrischer Ordnung erforderte.

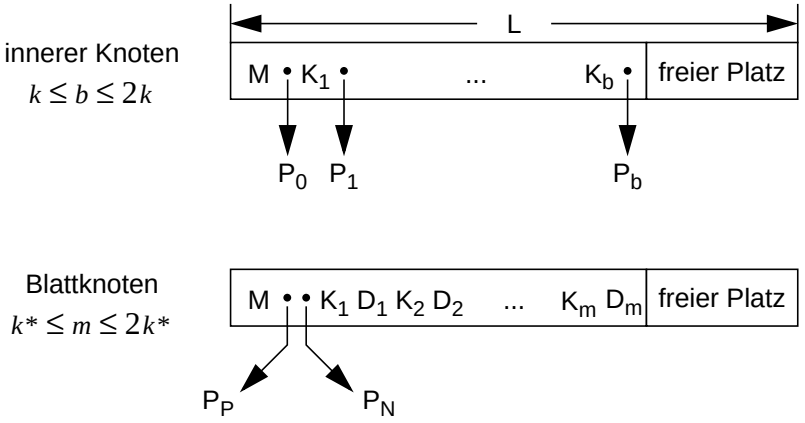
Definition: Seien k, k* und h* ganze Zahlen, $h^* \geq 0$, $k, k^* > 0$.
Ein B*-Baum B der Klasse τ (k, k*, h*) ist entweder ein leerer Baum oder ein geordneter Baum, für den gilt:

1. Jeder Pfad von der Wurzel zu einem Blatt besitzt die gleiche Länge.
2. Jeder Knoten außer der Wurzel und den Blättern hat mindestens k+1 Söhne, die Wurzel mindestens 2 Söhne, außer wenn sie ein Blatt ist.
3. Jeder innere Knoten hat höchstens 2k+1 Söhne.

4. Jeder Blattknoten mit Ausnahme der Wurzel als Blatt hat mindestens k* und höchstens 2k* Einträge.

Hinweis: Der so definierte Baum wird in der Literatur gelegentlich als B^+-Baum [Co79] bezeichnet.

Es sind also zwei Knotenformate zu unterscheiden:



Die Zeiger P_P und P_N (PRIOR und NEXT) dienen zur Verkettung der Blattknoten, damit alle Schlüssel direkt sequentiell aufsteigend und absteigend verarbeitet werden können. Das Feld M enthalte eine Kennung des Seitentyps sowie die Zahl der aktuellen Einträge. Da die Seiten eine feste Länge L besitzen, lassen sich aufgrund der obigen Formate k und k* bestimmen.

Soll in einem inneren Knoten der (2k+1)-te Eintrag gespeichert werden (beim Blattknoten der (2k*+1)-te Eintrag), entsteht ein Seitenüberlauf. Es wird ein Split-Vorgang ausgelöst, der ggf. auf der nächst höheren Ebene einen weiteren Split-Vorgang (usw. bis zur Wurzel) auslösen kann. Split-Faktoren größer 1 verbessern zwar die Speicherplatzbelegung, erhöhen aber auch den Reorganisationsaufwand [Kü83]; sie werden hier nicht betrachtet. Bei Seitenunterlauf erfordert der B*-Baum ein Mischen benachbarter Seiten, das sich ggf. auf dem Pfad bis zur Wurzel fortsetzen kann. In praktischen Systemen verzichtet man häufig auf die Realisierung der Mischoperation, wodurch man das Risiko einer Speicherplatzbelegung von $\beta < 50\%$ in Kauf nimmt. Eine Seite bleibt dann in der B*-Baumstruktur, solange sie noch mindestens einen Eintrag besitzt.

Ein Beispiel einer Indexstruktur I_{ABT}(ABTNR) mit dem Attribut UNIQUE ist in Bild 1 als B*-Baum veranschaulicht. Unterstellt man als Seitengröße 2KBytes und als Länge von ABTNR und P jeweils 4 Bytes, so erhält man für k den Wert 127 und als maximales Fan-out 255.

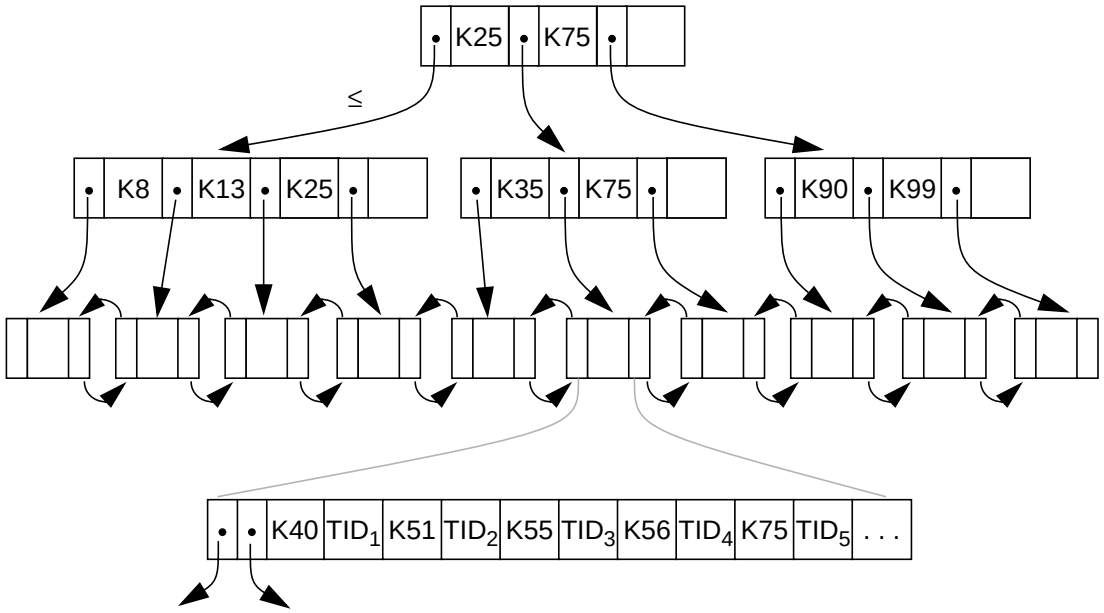
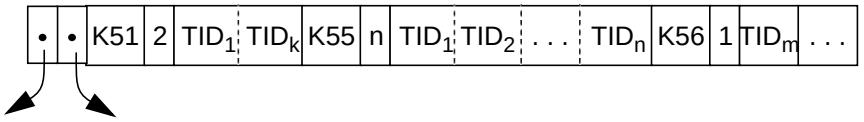


Bild 1: I_{ABT}(ABTNR) als B*-Baum der Klasse $\tau(2,3,3)$

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, eine 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. Die obige Blattseite könnte beispielsweise zum Index I_{PERS}(ANR) gehören.

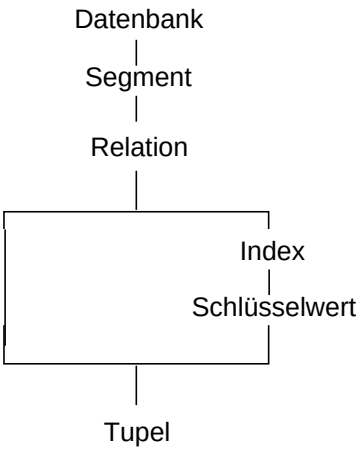
Nach dieser Skizzierung des B*-Baumes ist intuitiv klar, wie die oben eingeführten Basisoperationen einer Indexstruktur ablaufen. Die Lokalisierung des gesuchten Schlüssels (Fetch, Delete, Insert) erfordert ein Traversieren des Baumes von der Wurzel zum Blatt; es kostet also h* Seitenzugriffe (h*=3 in Bild 1 als typische Größe). Aktualisierungsoperationen bleiben immer lokal und auch bei Split- und Mischvorgängen immer beschränkt. Fetch Next verlangt eine Baumtraversierung zur Bestimmung der Startposition; es kann dann mit Hilfe eines Scans längs der verketteten Blattseiten fortgesetzt werden.

5. Synchronisation in Indexstrukturen

Die Serialisierbarkeit von Transaktionen erfordert bestimmte Synchronisationsmaßnahmen. In der Literatur wurden bisher sehr viele Synchronisationsprotokolle vorgeschlagen, deren Tauglichkeit jedoch nicht nachgewiesen wurde. Einen Überblick dazu gibt [ESS91]. In praktischen Fällen wird deshalb immer das bewährte strikte Zwei-Phasen-Sperrprotokoll herangezogen. Wir benutzen es in seiner verallgemeinerten Form als hierarchisches Sperrprotokoll.

5.1 Hierarchisches Sperrprotokoll

Zum Sperren von Objekten führen wir ein hierarchisches Sperrverfahren ein, das ein Sperren von Objekten mit unterschiedlichen Granulaten erlaubt, also z.B. Datenbank, Segment, Relation, oder Index und Tupel nach folgender Halbordnung:



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 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.

Generell nehmen wir an, daß bei Modifikationsoperationen das Tupel bereits längs der Hierarchie DB-Segment-Relation gesperrt ist, bevor ein Sperrprotokoll für Indexzugriffe angestoßen wird. Da bei Insert (K_i, TID) und Delete (K_i, TID) das TID des Tupels benötigt wird, muß es offensichtlich vorher lokalisiert sein.

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 (TID_1, 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.

Die Frage, wie die Indexstruktur (B*-Baum) beim Traversieren zu sperren ist, wurde bisher nicht diskutiert. Ebenso bleibt die Frage, wie ein Bereich im Index zu sperren ist, vorläufig offen. Die einfachste Lösung wäre das Setzen einer Indexsperre, z.B. durch

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

Aus Effizienzgründen kommt diese Lösung jedoch nur in Frage, wenn sich der Lesevorgang auf einen großen Teil des Indexbereichs bezieht und das Sperren der einzelnen Indexeinträge keine effektive Lösung mehr darstellt. Bei der Indexsperre bleiben natürlich alle Schlüsselwerte und alle Lücken dazwischen erhalten, so daß offensichtlich jeder Lesevorgang wiederholt werden kann.

5.2 Sperren in B*-Bäumen

Indexstrukturen werden typischerweise von verschiedenen Transaktionen gleichzeitig benutzt, wobei eine Transaktion T_i auch mehrere Operationen nacheinander ausführen kann. Konsistenzebene 3 (Serialisierbarkeit der T_i) erzwingt die Gewährleistung der Wiederholbarkeit von Lesevorgängen (RR = Repeatable Reads), was insbesondere auch für indexgestützte Leseoperationen gilt. Ein Synchronisationsprotokoll sollte deshalb folgendes ermöglichen [Gr81]:

- schneller Baumdurchlauf für T_i zur Lokalisierung von K_i
- Ausnutzung der Vorteile der Referenzlokalität auf Baumseiten
- Unterstützung hoher Transaktionsparallelität bei den eingeführten Grundoperationen während des Baumdurchlaufs und bei Strukturmodifikationen.

Diese Anforderungen implizieren

- möglichst kleine Sperrgranulate und geeignete Sperrmodi
- parallele Operationen mehrerer Transaktionen auf einer Indexstruktur und, wenn möglich, sogar auf einer Seite
- Einsatz von Sperren (lange Sperren bis EOT) auf Einträgen (K_i + TID-Liste) oder gar K_i-TID-Paaren
- Einsatz von kurzen Seitensperren (Latch, Sperrdauer bis zum Ende der Operation) für das Lesen und Ändern von Seiten
- Lesen und Aktualisierung von Objekten in einer Seite durch parallele Transaktionen T_i und T_j.

Protokolle mit Seitensperren würden auf Bäumen insbesondere bei Strukturmodifikationen oft zu einer Serialisierung der Transaktionen führen. Als Logging würden dann einfache Verfahren wie das Schreiben von BFIM- und AFIM-Seiten unter Beachtung der WAL- und COMMIT-Regel genügen [Gr81,HR83]. Als Konsequenz solcher Anforderungen sind deshalb ein geeignetes Sperrprotokoll für Einträge und damit zwangsläufig ein geeignetes Log-Verfahren auf Eintragsbasis zu entwickeln [ML89].

Hierbei ist zu beachten, daß bei einem ABORT von T_i ein bloßes physisches UNDO(T_i) nicht ausgeführt werden kann. Einträge und Sätze können in einer Seite ständig verschoben oder sogar in eine neue Seite migriert werden. Insbesondere können Strukturmodifikationen bei B*-Bäumen zum gleichzeitigen Verlagern mehrerer Einträge führen. Deshalb ist bei Recovery-Operationen eine Art Kompensation (logisches UNDO) oder eine Wiederholung (REDO) der Operation erforderlich, was wiederum einen für diese Operationen konsistenten DB-Zustand zum Recovery-Zeitpunkt voraussetzt.

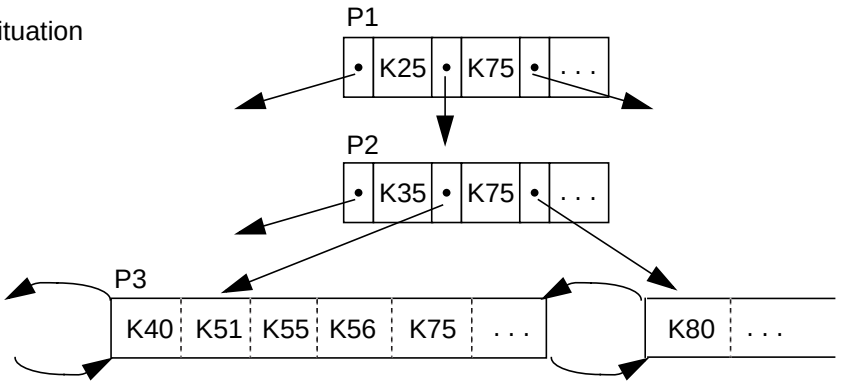
Folgendes Szenarium soll das Problem veranschaulichen. Die Operationen betreffen einen B*-Baum (Bild 1). Bild 2 illustriert Einfügeoperationen, von denen eine das Splitting einer Seite (P3) auslöst. K60 wurde dabei von T1 in P3 eingefügt und wird durch das Splitting nach P4 verschoben. Das nachfolgende Abort kann kein physisches UNDO für P3/P4 bewerkstelligen, weil inzwischen P3 durch Insert (K53) von T2 modifiziert wurde. Für unsere Zwecke verdeutlicht die skizzierte Operationsfolge, daß ein Logging/ Recovery-Verfahren auf Eintragsbasis (oder Operationsbasis) entworfen werden muß, wobei die Verschiebbarkeit protokollierter Einträge in einer Seite und zwischen Seiten gewährleistet sein muß.

Die Wiederholbarkeit von Lesevorgängen (RR) erfordert nur gleiche Reihenfolge von dem Benutzer sichtbaren Objekten. Das impliziert das lange Sperren dieser Objekte und, wenn notwendig, der Lücken zwischen diesen Objekten. Der Durchlauf eines B*-Baums braucht nicht wiederholbar zu sein, er muß lediglich zum gleichen Ergebnis, d.h. der gleichen Objektfolge führen.

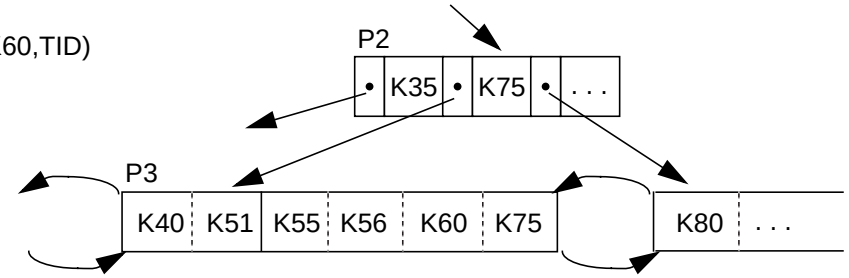
Deshalb schlagen wir für solche und andere Operationen Latches vor [Gr81], die in S- oder X-Modus erworben werden können. Da sie gegebenenfalls sehr häufig angefordert und freigegeben werden, sind spezielle Implementierungsmaßnahmen angebracht. Latch-Kontrollblöcke sind für Betriebsmittel, die häufig gemeinsam benutzt werden (Zugriffspfadseiten, Sperrtabelle, Systempuffer, Warteschlangen usw.), im virtuellen Speicher des DBS-Adreßraums “statisch” anzulegen, so daß sie direkt manipuliert werden können. Im Prinzip könnte man den Namen des Betriebsmittels B_i mit Hilfe einer Hashfunktion h auf eine Adresse h(B_i) abbilden, wo dann die Kontrollinformation für B_i gefunden werden kann. Durch die sorgfältige Wahl der Hashfunktion und durch eine ausreichende Größe der Hashtabelle kann sichergestellt werden, daß Hashkonflikte der Art h(B_i) = h(B_j) nur sehr selten auftreten. Eine Latch- und eine Unlatch-Operation sollte mit ~ 10 Maschineninstruktionen ausführbar sein. Unter der Voraussetzung, daß die Dauer der Verarbeitung zwischen Latch und Unlatch sehr kurz ist, kann das Warten auf einen Latch durch “busy wait” überbrückt werden; sonst durch explizite Deaktivierung des zugehörigen Prozesses.

B*-Bäume können mit Hilfe der Latch-Kopplung [BS77] durchquert werden. Von der Wurzel her werden längs des entsprechenden Pfades die Latches für die Seiten angefordert. Sobald für eine Seite P1 und

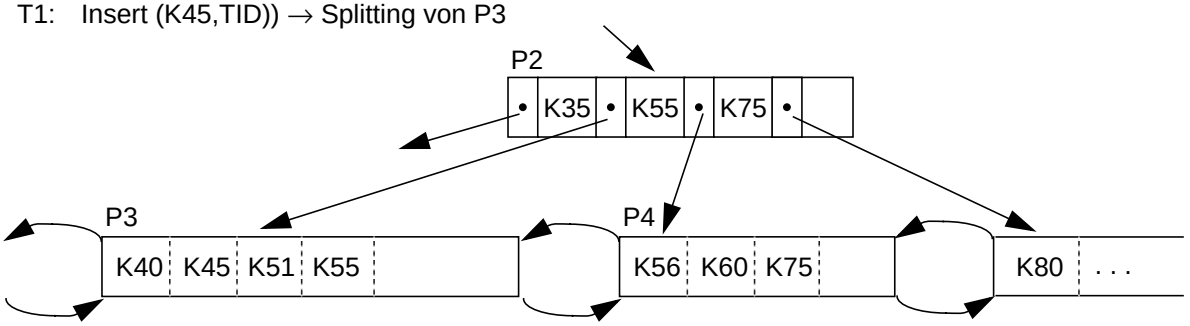
a) Ausgangssituation



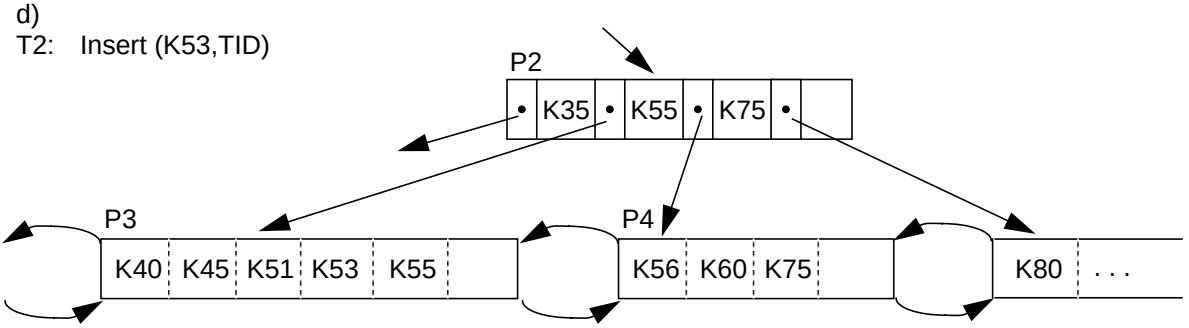
b)
T1: Insert (K60,TID)



c)
T1: Insert (K45,TID)) → Splitting von P3



d)
T2: Insert (K53,TID)



e)
T1: Abort

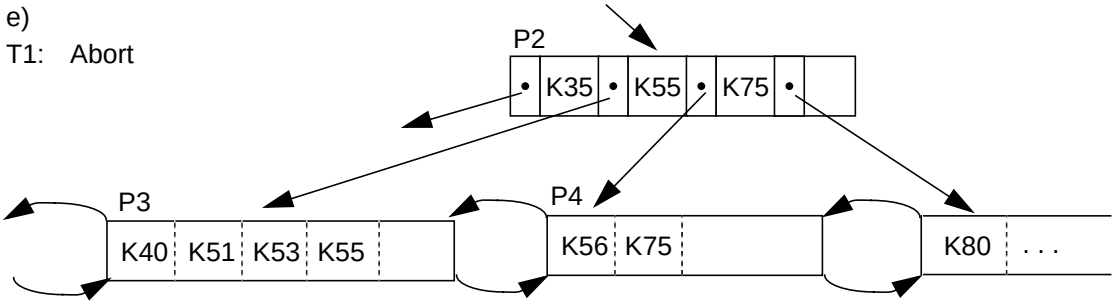


Bild 2: Modifikationen in einem B*-Baum

ihre Nachfolgeseite P2 Latches erworben wurden, kann der Latch für P1 wieder freigegeben werden. Das Insert (K60,TID) in Ausgangssituation a von Bild 2 erzeugt folgende Latch-Kopplung:

T1:	. . .	Anforderung der Sperren gemäß Sperrhierarchie bereits erfolgt
	Lock (IID1, IX)	
	Latch (P1, S)	P1 wird nur gelesen
	Latch (P2, S)	P2 wird nur gelesen
	Unlatch (P1)	
	Latch (P3, X)	P3 soll modifiziert werden; es ist genügend Platz vorhanden
	Unlatch (P2)	
	Lock (IID1K60,X)	lange Sperre für Indexeintrag
	. . .	Einfügen des Eintrags
	Unlatch (P3)	
	. . .	
	Unlock (. . .)	Freigabe aller Sperren bei Commit (T1)

Da ein Schlüsselwert in mehreren Indexstrukturen vorkommen kann, ist zusätzlich die Spezifikation des Indexnamens IID_k erforderlich. Da weiterhin der Name einer Sperre eine feste Länge (4 Bytes) haben muß, Schlüsselwerte jedoch variabel lang sein können, kann diese Restriktion mit Hilfe einer Hashfunktion erfüllt werden. Wenn beispielsweise I_{ABT}(ABTNR) den internen Namen IID1 besitzt, kann der Indexeintrag von Schlüsselwert K60 durch die Konkatenation (IID1K60) angesprochen werden.

Es ist offensichtlich, daß auf der Baumstruktur von Bild 2a auch parallele Transaktionen ohne gegenseitige Behinderungen Einfügungen vornehmen können. Beispielsweise können die beiden Aktionen T1: Insert(K57,TID_1) und T2: Insert(K81, TID_2) wie folgt ablaufen:

T1:			T2:	
	. . .			. . .
	Lock (TID_1, X)	Sperren der Tupel		Lock(TID_2, X)
	Latch (P1, S)			Latch (P1, S)
	Latch (P2, S)			Latch (P2, S)
	Unlatch (P1)			Unlatch (P1)
	Latch (P3, X)			Latch (P4, X)
	Unlatch (P2)			Unlatch (P2)
	Lock (IID1K57,X)			Lock (IID1K81,X)
	. . .			. . .
	Unlatch (P3)			Unlatch (P4)

Die beiden Einfügungen können ohne Strukturänderungen des B*-Baumes abgewickelt werden. Bei Strukturänderungen ist das Protokoll der Latch-Kopplung zu modifizieren. Ein einfaches Aussperrungsprotokoll könnte folgendermaßen ablaufen. Wird beim ersten Baumdurchlauf festgestellt, daß in der betreffenden Blattseite kein Platz mehr ist, wird der Latch wieder freigegeben, und der Baum wird von der Wurzel her im kritischen Pfad mit X-Latches belegt. Der Wurzel-Latch verhindert, daß neue Transaktionen in den Baum eintreten können. Über die anderen Latches kann verhindert werden, daß inkonsisten-

te Operationen ausgeführt werden. Es kann kein Deadlock entstehen, da alle Transaktionen zu den Blättern hin fortschreiten. Befinden sich noch Leser im Baum, so führt ein X-Latch höchstens zu einer Wartesituation. Vorhandene Latches werden in endlicher Zeit (sehr kurz) freigegeben. Wenn alle von einer Strukturmodifikation betroffenen Seiten mit X-Latches belegt sind, ist sichergestellt, daß andere Transaktionen nicht gefährdet werden. Die Situation von Bild "2b $\rightarrow$ 2c" läßt sich folgendermaßen bewältigen:

. . .	
Lock (IID1, IX)	
Latch (P1, S)	
Latch (P2, S)	
Unlatch (P1)	
Latch (P3, X)	P3 ist nicht sicher; es ist eine Strukturmodifikation erforderlich
Unlatch (P2)	
Unlatch (P3)	
Latch (P1, X)	
Latch (P2, X)	
Latch (P3, X)	
. . .	Anfordern einer neuen Seite P4
Latch (P4, X)	
Lock (IID1K45, X)	
. . .	Modifikation von P2, P3 und P4
Unlatch (P1, ... P4)	

Es ist offensichtlich, daß das strikte Aussperrungsprotokoll verbessert werden kann. Falls ein Knoten im Einfügungspfad sicher ist, d.h. noch Platz für einen Eintrag besitzt, kann der X-Latch auf dem Vorgängerknoten sofort wieder freigegeben werden. Das bedeutet, daß nur der "unsichere" Teilbaum während der gesamten Strukturmodifikation gesperrt bleibt.

Im Prinzip wird das Latch-Protokoll bei jeder Indexoperation von der Wurzel beginnend bis zum betreffenden Blatt durchlaufen. Beim Index-Scan ist eine Optimierung [Mo90] derart denkbar, daß im Scan-Kontrollblock die Blattseite vermerkt wird, die den Schlüsselwert mit der aktuellen Cursorposition enthält. Ein Fetch Next kann so versuchen, für die aktuelle Blattseite einen S-Latch direkt anzufordern, was mit den eben beschriebenen Baum-Protokollen verträglich ist. Jedoch muß dabei sorgfältig kontrolliert werden, ob die Blattseite in der Zwischenzeit modifiziert wurde. Bei einem Split- oder Mischvorgang kann der Schlüsselwert in eine ganz andere Seite verschoben worden sein; die fragliche Seite kann sogar vollständig aus dem B*-Baum entfernt worden sein. Wenn jede Seite mit einer Versionsnummer versehen und diese bei jeder Änderung in der Seite inkrementiert wird, läßt sich dieses Problem mit angemessenem Aufwand lösen.

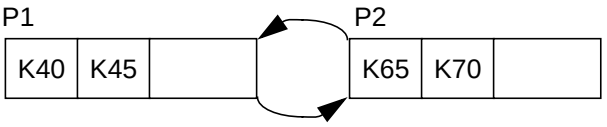
5.3 Sperrprobleme bei Indexzugriffen

Die bisher eingeführten Protokolle reichen für einfache Fetch-Operationen (= K_i) aus. Bei mächtigeren Fetch-Operationen (z.B. >K_i) und bei Fetch Next können jedoch Phantom-Probleme auftreten, die die Forderung nach Serialisierbarkeit der Transaktionen verletzen [Mo90]. Bei Suchbedingungen wie '>K45' oder '=K4?' als Präfix-Bedingung ist der erste diese Bedingung erfüllende Schlüsselwert im Index nicht unmittelbar bekannt. Deshalb kann der betreffende Indexeintrag auch nicht direkt gesperrt werden. Eine sequentielle Suche im Index führt zu dem zu diesem Zeitpunkt nächsten Schlüssel.

Im folgenden verzichten wir auf die Konkatenation (IID K_i) eines Index-Schlüssels und benutzen nur die Kurzform (K_i).

1. Problem

Der zugehörige Eintrag könnte von einer anderen Transaktion in einem unverträglichen Modus gesperrt sein.



T1:	Lock (K65, X)	
	. . .	
T2:	Fetch Next (>K45)	
	. . .	
	Latch (P1,S)	
	Latch (P2,S)	
	Unlatch (P1)	
	Lock (K65,S)	wartet auf Unlock (K65) durch T1

Damit ist die Seite P2 ggf. für lange Zeit blockiert, was die Parallelität der Operationen auf dem Index beeinträchtigen kann. Außerdem könnte ein Deadlock entstehen, wenn T1 noch einmal Latch (P2,X) anfordert. Eine Verbesserung bringt ein bedingter Lock-Aufruf, der bei Blockierung die Kontrolle sofort zurückgibt.

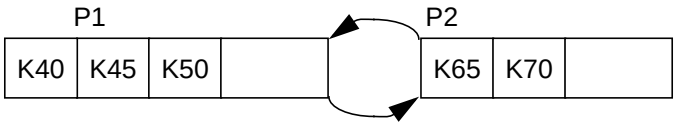
T2:	Condlock (K65, S)	Blockierung
	Unlatch (P2)	
	Lock (K65, S)	damit wird bis zur Lockgewährung gewartet

2. Problem

T1 könnte seine Operationen wie folgt fortsetzen:

T1:	Insert (K50, TID)	
	. . .	mit den entsprechenden Latch-Operationen
	Lock (K50, X)	

Damit ergibt sich folgende Situation:



T1: Commit (→ Unlock (K65), Unlock (K50))
T2: Fetch liefert jetzt K65 zurück

Bei diesem Suchvorgang sieht T2 K50 nicht; ein erneutes Fetch (>K45) würde jedoch K50 liefern. Um diese Inkonsistenz zu vermeiden, könnte man bei Sperrgewährung das zuletzt durchquerte Schlüsselintervall erneut durchsuchen. Dieses in [Mo90] vorgeschlagene Revalidierungsprinzip erkennt hier das Phantom K50; es hilft, wenn ein Suchvorgang durch die Änderungssperre einer anderen Transaktion blockiert wurde. Es ist jedoch keine allgemeine Lösung, die Serialisierbarkeit garantiert.

3. Problem

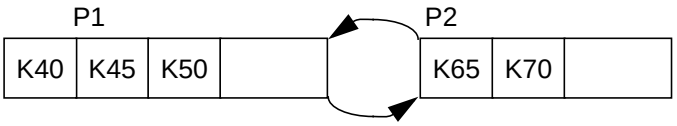
Wenn beispielsweise später die Operationen

```
T3: Insert (K48, TID)
        . . .
        Lock (K48, X)
        . . .
    Commit
T2: Fetch Next (>K45)
```

folgen, sieht T2 wiederum eine andere Schlüsselfolge. Das Revalidierungsprinzip hilft also bei der Wiederholung einer Lesefolge nicht. Das Problem tritt offensichtlich dadurch auf, daß T3 nicht mitgeteilt wurde, daß eine noch nicht beendete Transaktion (T2) die Lücke zwischen K45 und K65 bereits durchsucht hatte. Die Einfügung von T3 erzeugt also für T2 ein Phantom.

4. Problem

Wir beziehen uns auf folgende Situation:



Transaktion T1 löscht das Tupel (K50, TID), und Transaktion T2 startet zu zwei verschiedenen Zeitpunkten eine Suche, die das Indexintervall mit K50 betrifft.

T1: Delete (K50, TID)	
. . .	
Lock (TID, X)	Sperren des Tupels
Lock (K50, X)	Sperren des Indexeintrages
. . .	Löschen von Tupel und Eintrag
T2: Fetch Next (> K45)	
. . .	
Lock (K65, S)	Es wird Tupel mit Schlüssel K65 gefunden
. . .	
T1: Abort	Die Delete-Operation von T1 wird rückgängig gemacht
T2: Fetch Next (> K45)	
. . .	
Lock (K50, S)	Es wird Tupel mit Schlüssel K50 gefunden!
. . .	

Hier ergibt sich das Problem dadurch, daß die Löschung von T1 einem nachfolgenden Leser nicht bekannt gemacht wurde. Auch dadurch kann die RR-Eigenschaft gefährdet werden.

5.4 Sperrprotokolle für Indexzugriffe

Eine Lösung impliziert offensichtlich das Sperren aller gelesenen Objekte zusammen mit allen Lücken dazwischen. Das direkte Sperren von Lücken ist entweder extrem ineffizient, wenn alle möglichen (nicht existierenden) Schlüssel gesperrt werden, oder gar unmöglich z.B. bei Schlüsseln vom Typ REAL. Eine elegante Lösung wäre auf der Basis von Bereichssperren (Prädikate) möglich, die entsprechende Schlüssellücken abdecken. Jedoch würden solche Maßnahmen die Repräsentation von Prädikaten und ihre häufige Überprüfung verlangen, was aus verschiedenen Gründen ineffizient ist.

Eine akzeptable Lösung für das Problem, Schlüsselbereiche oder -lücken zu sperren, kann man erhalten, wenn man für alle Transaktionen ein verbindliches Sperrprotokoll vorschreibt. Eine Leseoperation von T1 muß einer in denselben Bereich nachfolgenden Einfügeoperation von T2 mitgeteilt werden können. Ebenso muß ein Löschvorgang einem nachfolgenden Leser bekannt gemacht werden. Diese Forderungen lassen sich durch folgendes Protokoll erfüllen. Eine Fetch-Operation setzt auf jeden aufgesuchten Schlüssel eines Suchbereiches eine S-Sperre. Eine Einfügeoperation muß überprüfen, ob der “nächste Schlüsselwert” verfügbar ist; dies läßt sich durch eine kurze Sperranforderung mit X-Modus erreichen. Bei einer Löschoperation wird der betreffende Eintrag aus dem Index entfernt. Folglich kann nur ein benachbarter Eintrag für die Kommunikation mit nachfolgenden Lesern benutzt werden. Eine Löschoperation muß deshalb eine lange Sperre auf dem nächsten Schlüsselwert in einem Modus erwerben, der einen Leser bis Commit der Löschtransaktion zu warten zwingt. Der zu löschende Schlüssel muß mit einer kurzen X-Sperre überprüft werden, ob er für die Löschoperation verfügbar ist. Als Beispiel kann folgendes Szenarium dienen:

T1: Fetch Next (>K50)

	Lock (K65, S)	erhält K65

T2: Insert (K55, TID)

	Lock (K65, X, I)	nächster Schlüsselwert muß überprüft werden
	. . .	

T1: Commit

	. . .	
T2:	Lock (K55, X)	eingefügter Schlüssel erhält eine lange Sperre

In die Lücke von K50 - K65 kann erst nach Commit(T1) eingefügt werden. Die Anweisung Lock (K65, X, I) besagt, daß die X-Sperre auf K65 erworben werden muß, aber sofort wieder freigegeben werden kann (I = instant).

In ähnlicher Weise muß bei Löschvorgängen verfahren werden.

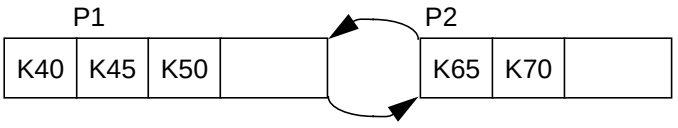
T1: Delete (K50, TID)

Lock (K65, X)	Beim Löschen muß auch der nächste Schlüsselwert gesperrt werden
Lock (K50, X, I)	Schlüsselwert K50 wird gesperrt, aber aus dem Index entfernt

T2: Fetch Next (>K45)

Lock (K65, S)	T2 sieht K50 nicht; es fordert den nächsten in der Indexstruktur befindlichen Schlüssel an
T1: Abort	Das Revalidierungsprinzip führt jetzt auf die Folge K50, K65, ...

Das Prinzip der Bereichssperren mit Hilfe des nächsten Schlüsselwertes fordert die Einhaltung von vorgegebenen Protokollen. Dabei soll noch einmal unterschieden werden, daß damit zwar die vier skizzierten Probleme gelöst werden können, jedoch nicht immer minimale Bereichssperren eingesetzt werden. Das liegt an folgender inhärenter Vorgehensweise: Das Konzept “Sperren des nächsten Schlüsselwertes” teilt nachfolgenden Operationen mit, daß in der Schlüssellücke noc*h nicht freigegebene* Operationen (Delete) erfolgt sind oder daß ein Leser die Lücke *vielleicht* durchsucht hat. Falls in der folgenden Situation



T1: Fetch (K65)

. . .
Lock (K65, S)
. . .

durchführt, bleibt das Schlüsselintervall K51-K64 für Einfügungen anderer Transaktionen unzugänglich.

T2: Insert (K53, TID)

. . .
Lock (K65, X)
. . .

In diesem Fall wird T2 unnötig blockiert, weil die Kommunikation über “Sperren des nächsten Schlüsselwertes” mit vorher dagewesenen Transaktionen eine zu geringe Information übermittelt. Insgesamt gesehen, erlaubt das gewählte Prinzip jedoch effiziente und effektive Lösungen unserer Problemstellung.

Die Diskussion der Sperrprobleme auf Indexstrukturen wurde bisher nur für die Suche mit Fetch und Fetch Next (>, ≥, =) geführt. Als Ergebnis kann folgendes festgehalten werden.

- Bevor eine Sperre auf Indexeinträge angefordert werden kann, ist die Baumstruktur mit Hilfe der Latch-Kopplung zu durchlaufen.
- Eine Fetch-Operation setzt eine S-Sperre auf jeden (intern) gelesenen Schlüsselwert des Index (auch auf solchen, deren referenzierte Tupel das Suchargument nicht erfüllen). Soll in die Lücke bis zum nächst höheren Schlüssel nichts eingefügt werden, so muß auch der nächst höhere Schlüsselwert mit S gesperrt werden.
- Das Konzept der Bereichssperren erfordert, daß Aktualisierungsoperationen (Insert und Delete) ein bestimmtes Sperrprotokoll einhalten. Sie müssen den nächst höheren Schlüsselwert in einem mit S unverträglichen Modus sperren.
- Beim Insert genügt ein Test, ob Zugriff auf den nächsten Schlüsselwert gewährt werden kann (instant lock duration).
- Bei Delete muß die Sperre auf den nächsten Schlüsselwert angefordert und bis Commit gehalten werden (long lock duration).
- Die Ungleichbehandlung von Insert und Delete liegt darin begründet, daß ein noch nicht freigegebener Schlüsselwert für Lesetransaktionen in der Indexstruktur sichtbar ist und einen Scan stoppt, während ein gelöschter Schlüsselwert bereits aus der Indexstruktur entfernt wurde, durch Abort der löschenden Transaktion jedoch jederzeit wieder “auftauchen” kann.

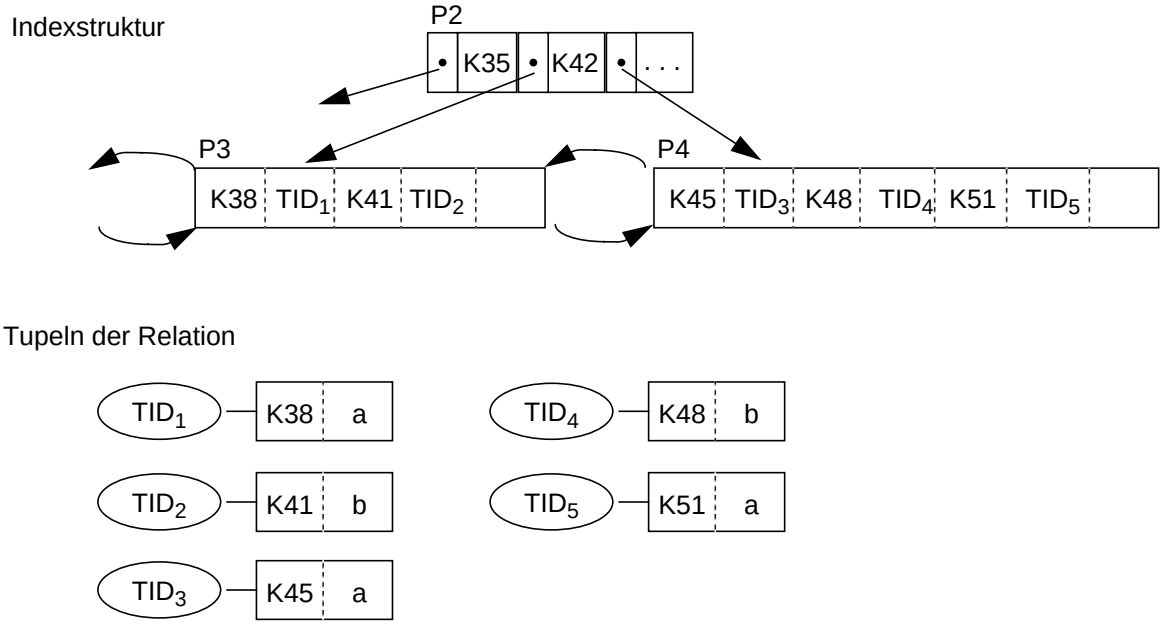


Bild 3: Ausgangssituation zu den Sperrprotokollen für Indexoperationen

An einem Beispiel soll jetzt die Synchronisation von Abläufen illustriert werden, die bei Erweiterung der Suchprädikate (<, ≤, ≠) und Durchlauf der Indexstruktur in beiden Richtungen (Next, Prior) auftreten. Dabei betrachten wir nur noch die Sperren auf die Schlüsselwerte und Tupel. Es sei folgende Ausgangssituation gegeben (Bild 3).

Transaktion T1 sucht die Tupel mit der Bedingung (A1 = 'K4?' AND A2 = 'a'), d.h. mit einer Präfix-Bereichsbedingung und einem einfachen Suchargument (Fetch Next). T2 sucht die Tupel, die (A1 ≤ 'K50') erfüllen (Fetch Prior). T3 und T4 fügen neue Tupel ein.

T1: Fetch Next (A1 = 'K4?' AND A2 = 'a')	
. . .	
Lock (K41, S)	lange Sperre, obwohl sich
Lock (TID_2, S)	(TID_2, K41, b) nicht qualifiziert
. . .	
Lock (K45, S)	
Lock (TID_3, S)	Tupel (TID_3, K45, a) qualifiziert sich
. . .	Tupel (TID_3, K45, a) wird verarbeitet
T3: Insert (K40, TID_6)	Tupel (TID_6, K40, a) soll eingefügt werden
Lock (TID_6, X)	
. . .	
Lock (K41, X, I)	nächster Schlüssel muß kurz gesperrt werden T3 wartet auf die Freigabe von K41 durch T1. Ohne die S-Sperre von T1 hätte hier ein “Phantom” erzeugt werden können.
T4: Insert (K47, TID_7)	Tupel (TID_7, K47, a) soll eingefügt werden
Lock (TID_7, X)	
. . .	
Lock (K48, X, I)	Sperre wird gewährt und sofort wieder freigegeben (instant duration)
Lock (K47, X)	
. . .	Einfügen des Eintrags (K47, TID_7)

Der Eintrag ist nun in Seite P4 des B*-Baumes eingefügt (Bild 4).

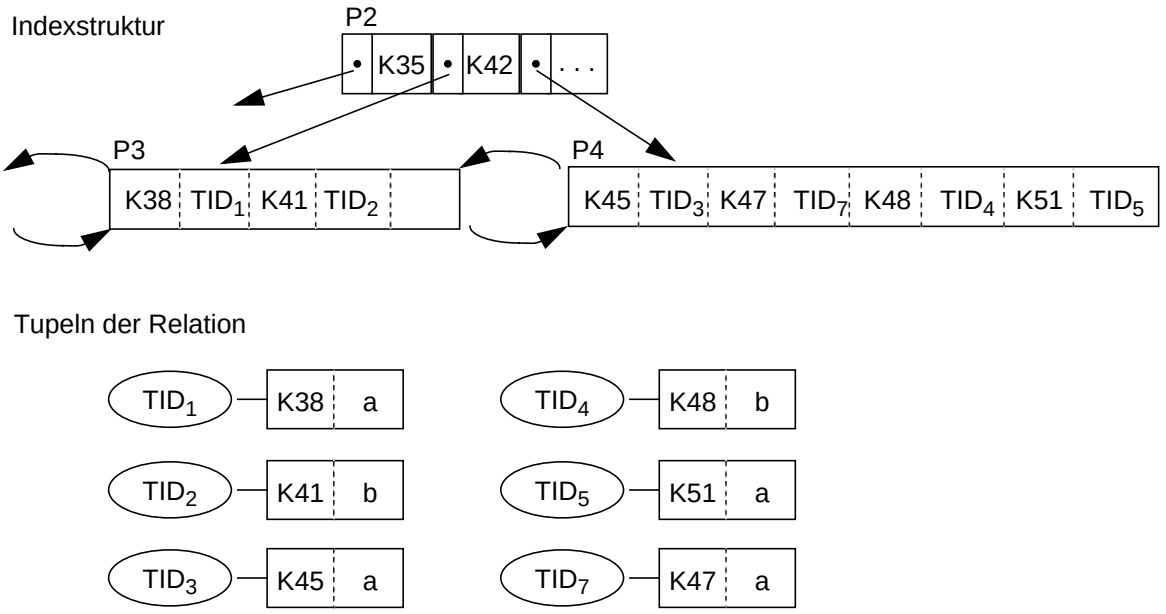


Bild 4: Fortsetzung des Beispiels zu den Sperrprotokollen für Indexoperationen

T2 beginnt jetzt mit der Suche nach (A1 ≤ 'K50')

T2: Fetch Prior (A1 ≤ 'K50')	
. . .	
Lock (K51, S)	T2 hat den Schlüsselwert des rechten Randes des Suchbereichs zu sperren. Jetzt kann keine Transaktion mehr den Schlüsselwert K48 löschen oder in die Lücke zwischen K48 und K51 einfügen.
Lock (K48, S)	
Lock (TID_4, S)	
. . .	Tupel (TID_4, K48, b) hat sich qualfiziert und wird verarbeitet.
T2: Fetch Prior (A1 ≤ 'K50')	
. . .	
Lock (K47, S)	
. . .	T2 muß hier auf die Freigabe von Tupel (TID_7, K47, a) durch T4 warten.
T4: Commit	
T2: Lock (TID_7, S)	
. . .	Tupel (TID_7, K47, a) wird verarbeitet
T1: Fetch Next (A1 = 'K4?' AND A2 = 'a')	
. . .	
Lock (K47, S)	wird sofort gewährt
Lock (TID_7, S)	
. . .	Tupel (TID_7, K47, a) qualifiziert sich und wird verarbeitet

T1 kann mit seiner Suche in der gezeigten Weise fortfahren. K48 wird gesperrt, qualifiziert sich aber nicht. Es wird festgestellt, daß K48 momentan der letzte Eintrag im Bereich K4? ist. Um die Wiederholung des Lesevorgangs jedoch zu gewährleisten, muß der Randschlüssel K51 gesperrt werden; das verhindert hier das Einfügen von K49.

T2 setzt seine Suche ebenfalls fort und erreicht K45 und K41. Es ändert dann das Suchargument auf (A1 ≥ 'K40' AND A1 ≤ 'K50') und beendet somit den Fetch-Prior-Scan. K41 ist gesperrt, was für unser Protokoll ausreichend ist, um Einfügungen und Löschungen im Bereich K38 - K41 zu vermeiden. Die vorherigen Lesesperren auf allen Schlüsseln, auf die zugegriffen wurde, verhindern diese Operationen auf dem gesamten Pfad, auf dem gelesen wurde, was also die Wiederholbarkeit des Lesens sicherstellt.

Bei der verallgemeinerten Suche können Leser die Blattseiten eines B*-Baumes in entgegengesetzten Richtungen durchlaufen. Dies kann jedoch nicht zu einem Deadlock führen, da für den Baum-Durchlauf nur S-Latches angefordert werden. Solange bei Änderungen und Strukturmodifikationen nur jeweils auf einem Blatt X-Latches gesetzt werden, kann es nur Wartesituationen geben, da solche Latches nur von der Wurzel zum Blatt hin erworben werden.

Unser Beispiel zeigte die Suche im Bereich (A1 ≥ K40 AND A1 ≤ K50) in NEXT- und PRIOR-Richtung. Daraus können wir in allgemeiner Weise die Regeln für das Sperren von Indexstrukturen ableiten. Die eingangs dieses Abschnitts aufgestellten Regeln müssen nur geringfügig ergänzt werden:

- Bei Insert wird der nächste Schlüsselwert K_i mit Lock (K_i, X, I) gesperrt (instant duration).
- Bei Delete wird der nächste Schlüsselwert K_i mit Lock (K_i, X) gesperrt.

- Bei Fetch Next sei die Stoppbedingung ($K_i \leq K_u$) oder ($K_i < K_u$). Alle Schlüsselwerte K_i, die im Suchbereich liegen, werden dann der Reihe nach mit Lock (K_i,S) gesperrt. Am Bereichsende ist folgendermaßen zu verfahren: Falls K_u Schlüsselwert des Index ist, genügt Lock (K_u,S) als letzte Sperranforderung. Ansonsten wird der nächst größere Schlüsselwert K_j im Index durch Lock (K_j,S) gesperrt.
- Bei Fetch Prior sei die Startbedingung ($K_i \leq K_u$) oder ($K_i < K_u$). Alle Schlüsselwerte K_i, die im Suchbereich liegen, werden der Reihe nach mit Lock (K_i <S) gesperrt. Am Bereichsanfang muß folgendermaßen verfahren werden: Falls K_u Schlüsselwert des Index ist, genügt Lock (K_u,S) als erste Sperranforderung. Ansonsten wird der nächst größere Schlüsselwert K_j im Index als erster durch Lock (K_j,S) gesperrt.

Der pathologische Fall ($\neq K_i$) impliziert, daß der gesamte Schlüsselbereich zu sperren ist und deshalb aus Effizienzgründen nicht mit Sperren für einzelne Schlüssel, sondern mit einer Sperre des gesamten Index behandelt werden sollte. Die gleiche Vorgehensweise ist zu empfehlen, sobald der zu sperrende Bereich viele Schlüssel enthält. (Dies ist eine relative Aussage, bei der auch die parallelen Aktivitäten auf der Indexstruktur durch andere Transaktionen zu berücksichtigen sind).

Bisher wurde angenommen, daß jeder Schlüsselwert nur einmal im Index vorkommt, daß also ein UNIQUE-Index vorliegt. Im Falle von TID-Listen (NON-UNIQUE-Index) lassen sich später Verbesserungen vornehmen. Unsere bisherige Lösung für Sperrprotokolle bei einem UNIQUE-Index kann wie folgt zusammengefaßt werden:

	Nächster Schlüsselwert	Betroffener Schlüsselwert
Fetch	-	S
Fetch Next	S*	S
Fetch Prior	S**	S
Insert	X, I	X
Delete	X	X

* nur wenn der nächste Schlüsselwert bei der Stoppbedingung außerhalb des Suchbereichs liegt

** nur wenn der vorherige Schlüsselwert bei der Startbedingung (<, ≤) außerhalb des Suchbereichs liegt.

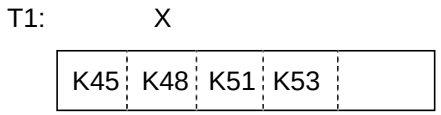
Bei Leseoperationen ist der betroffene Schlüsselwert (auch wenn er nicht im Index existiert) mit S und bei Aktualisierungsoperationen mit X zu sperren. Das Prinzip des “Sperren des nächsten Schlüsselwerts” erfordert weiterhin X-Sperren bei Insert/Delete und besondere Vorkehrungen beim Start (Fetch Prior) und bei Beendigung (Fetch Next) einer Bereichssuche.

5.5 Optimierung der Sperrprotokolle

Wenn der “nächste Schlüsselwert” bereits gesperrt ist, muß die Transaktion, die für ihn eine unverträgliche Sperre anfordert, warten. Durch diese Technik der Bereichssperren werden für die ändernde Transaktion größere Bereiche (relativ zum aktuellen Schlüssel vom Vorgänger bis zum Nachfolger) unzugänglich. Da das Einfügen und das Löschen eines Tupels jeden für die Relation definierten Index betreffen, können bei n Indexstrukturen n Bereiche blockiert sein, bei Tupeländerungen (Löschen gefolgt von einem Einfügen) sogar 2n. Sind bei solchen Aktualisierungsoperationen Systemkataloge mit ihren Indexstrukturen involviert, so können erhebliche Leistungsverluste durch Serialisierung von Transaktionen die Folge sein. Deshalb bleibt noch zu untersuchen ob immer lange Sperren in den strikten Modi S und X benötigt werden oder ob weniger restriktive Lösungen genügen.

Einfügen in einen Schlüsselbereich

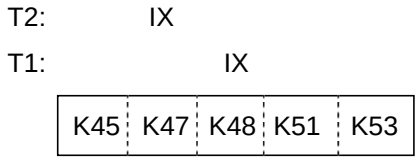
Nach dem bisherigen Sperrprotokoll würde Transaktion T1 durch Einfügen von K48 den Bereich von K45 bis K48 blockieren:



Würde für das Einfügen der Sperrmodus bei den aktuellen und nächsten Schlüsseln auf IX reduziert, so können mehrere Transaktionen in diesem Bereich einfügen:

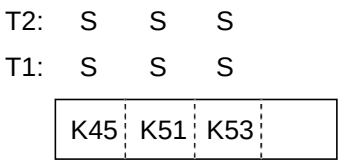
T1:	Lock (K51, IX, I)	wurde nächster Schlüsselwert gelesen?
	Lock (K48, IX)	
		Einfügen von (K48, TID)
	. . .	
T2:	Lock (K48, IX, I)	wurde nächster Schlüsselwert gelesen?
	Lock (K47, IX)	
		Einfügen von (K47, TID)
	. . .	

Als Ergebnis erhalten wir folgenden gültigen Zustand, wobei zwei Einträge noch nicht freigegeben sind:



Wenn also eine IX-Anforderung mit dem Sperrzustand des nächsten Schlüssels verträglich ist, kann eine durch IX abgesicherte Einfügung durchgeführt werden. Die optimierte Einfügeregel ist jedoch im allgemeinen Fall nicht so einfach, wie die nachfolgenden Beispiele zeigen.

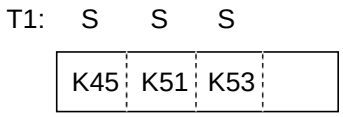
T1 und T2 haben den Bereich K45 - K53 gelesen, wodurch sich folgende Ausgangssituation ergibt:



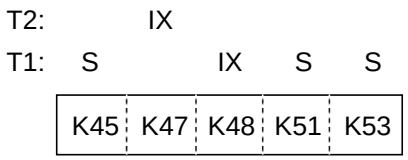
Das Einfügen von K48 durch T1 wird hier durch das Lock (K51, S) von T2 verhindert, da es für die Einfügung von T1 die Funktion einer Bereichssperre darstellt. Erst nach Commit von T2 ist diese Einfügung möglich.

Wiederholbarkeit von Lesevorgängen und Einfügen

T1 habe bereits den Bereich K45 - K53 gelesen.

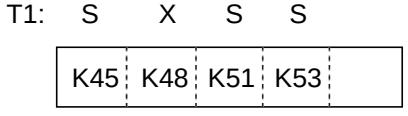


Es ist offensichtlich, daß T1 K48 einfügen darf. Wird K48, wie bisher angenommen, nur durch IX abgesichert, kann T2 K47 mit IX einfügen, was die Wiederholbarkeit des Lesevorgangs zunichte macht (nach Commit von T2).



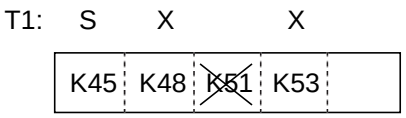
Die S-Sperre von T1 auf K51 hatte vor den Einfügungen die Funktion einer Bereichssperre für K45 - K51, was durch das vorgegebene Protokoll Einfügungen und Löschungen anderer Transaktionen in diesem Bereich verhinderte. Das Einfügen von K48 durch T1 teilte den ursprünglichen Bereich in zwei Bereiche K45 - K48 und K48 - K51, wobei allerdings das gewählte Sperrprotokoll aufgrund der S-Sperre auf K51 nur noch den zweiten Bereich absicherte. Im Bereich K45 - K48 können deshalb alle Operationen vorgenommen werden, deren Sperranforderungen mit Lock (K48, IX) verträglich sind. Um diese unerwünschte Situation zu vermeiden, muß offensichtlich die Funktion der Bereichssperre auf den eingefügten Schlüssel übertragen werden, damit die Absicherung des gesamten ursprünglichen Bereichs erhalten bleibt.

Wenn T1 (und keine andere Transaktion mehr) den nächsten Schlüssel mit S gesperrt hat, muß der neu eingefügte Schlüssel mit einem mit IX und S unverträglichen Modus gesperrt werden, also beispielsweise so:



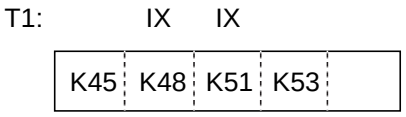
Löschen von Einträgen

Wird in der Ausgangssituation K51 durch T1 gelöscht, so ist der nächste Schlüssel mit X als Bereichssperre abzusichern. Beim Einfügen von K48 ist dann wiederum die Funktion der Bereichssperre zu übertragen.

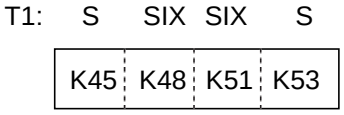


Einfügungen mit nachfolgenden Lesevorgängen

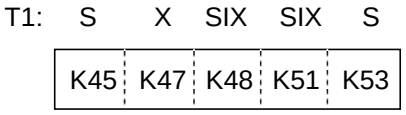
Ein dritter Fall kann in der folgenden Situation auftreten:



Nach dem Lesen des gesamten Bereichs durch T1 ergibt sich folgender Sperrzustand:



Das Einfügen von K47 erzwingt hier den Transfer der Funktion der Bereichssperre von K48, weil sonst eine andere Transaktion in die Lücke schreiben könnte:



Die Optimierung der Bereichssperrung beim Einfügen kann also folgendermaßen erfolgen:

- Zuerst wird der nächste Schlüsselwert mit Lock (K_i, IX, I) von T1 gesperrt. Dabei wird IX nur gegen Sperren anderer Transaktionen verglichen.
- Sobald dieser Test erfolgreich war, wird der Modus ermittelt, in dem T1 den nächsten Schlüssel gesperrt hält.
- Falls der Modus S, X oder SIX ist, wird der neu einzufügende Schlüssel mit X (oder SIX) bis Commit gesperrt.
- Sonst wird der Schlüssel mit IX bis Commit gesperrt.

Zusammenfassung

Durch die vorgeschlagene Optimierung wird also im wesentlichen das Einfügen von Schlüsselwerten verbessert, wenn keine Lesefolgen zu schützen sind. Zusammenfassend lassen sich die Sperrprotokolle wie folgt darstellen:

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 Ergebnisse der bisherigen Diskussion 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.

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

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

5.6 Sperrprotokolle bei Indexstrukturen für Sekundärschlüssel

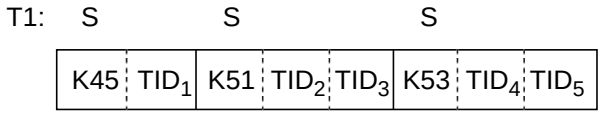
Die bisherige Diskussion bezog sich auf Indexstrukturen, bei denen jeder Schlüssel in einem Blattknoten auf genau ein Tupel verwies, also nur ein TID im Eintrag hatte. Sekundärschlüssel verweisen auf mehrere Tupel; die Einträge in der zugehörigen Indexstruktur besitzen also neben dem Schlüssel 1 bis n TIDs (Nonunique Index). Später entwickeln wir ein Sperrverfahren, das jedes Paar Schlüsselwert/TID separat zu sperren erlaubt.

Wir gehen jetzt davon aus, daß pro Schlüssel nur ein Eintrag existiert und daß die Einträge variabel lang sind, also aus einem Schlüsselwert und bis zu n TIDs bestehen. Aspekte der Abbildung, die die Verwaltung der Einträge komplex gestalten, wollen wir hier nicht betrachten. Wenn ein Eintrag sehr viele TIDs besitzt, so daß er nicht vollständig in eine Blattseite paßt, so könnte beispielsweise ein Teil in eine spezielle Überlaufseite ausgelagert werden oder in der Folgeseite auf Blattebene unter Wiederholung des Schlüssels fortgesetzt werden. Im letzten Fall gäbe es bei blattüberspannenden Operationen komplexere Latch-Protokolle [Mo90] (ggf. mit Deadlockgefahr); diese Detailebene der Betrachtung wollen wir hier jedoch nicht wählen.

Wir orientieren uns im wesentlichen an den Sperrprotokollen für UNIQUE-Indexstrukturen und diskutieren deren Probleme für den Fall variabel langer Schlüsselwert/TID-Listen.

Lesen eines Schlüsselbereichs

Der Schlüsselbereich (A1 ≥ K45 AND A1 ≤ K52) werde von T1 durch eine Folge von Fetch -Next-Operationen gelesen. Nach Abschluß des Lesevorgangs ergibt sich folgender Sperrzustand:



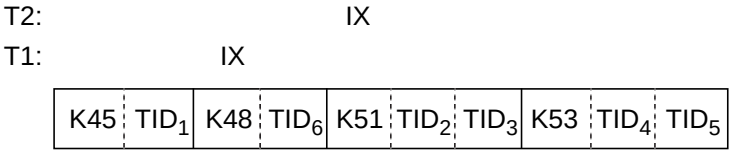
Das Protokoll der Bereichssperren verhindert hier, daß eine andere Transaktion neue Schlüsselwerte (z.B. K52) in die Lücken einfügt oder die vorhandenen Einträge (z.B. K51) erweitert.

Einfügen eines neuen Schlüsselwertes

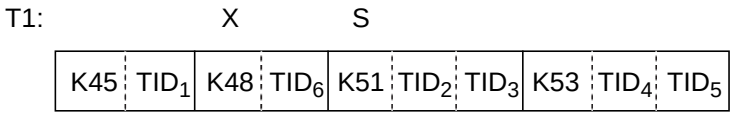
T1 fügt (K48, TID_6) in den Index ein. Dabei ist zunächst der nächste Schlüsselwert mit Lock (K51, IX, I) zu testen, ob eine unverträgliche Sperre vorliegt:

- S-Sperre: Eine andere Transaktion könnte gelesen haben
- SIX-Sperre: Eine andere Transaktion könnte eingefügt und nachfolgend gelesen haben
- X-Sperre: Eine andere Transaktion könnte einen Eintrag im kritischen Bereich (z.B. K48) vollständig gelöscht und noch nicht freigegeben haben.

Wenn eine verträgliche Sperrsituation vorliegt (z.B. keine Sperre oder (K51, IX)), dann wird die Einfügung durchgeführt mit Lock (K48, IX):



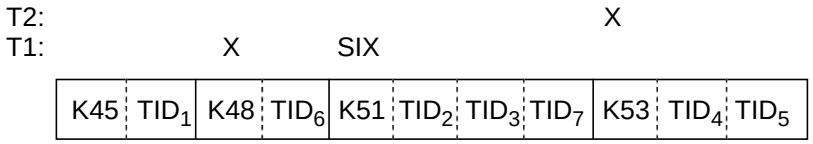
Wenn T1 selbst eine S-, X- oder SIX-Sperre auf dem nächsten Schlüsselwert besitzt, muß die Bereichssperrenfunktion transferiert werden. Der einzufügende neue Schlüsselwert erhält eine X-Sperre.



Erweitern eines vorhandenen Eintrags

Durch Einfügen von (K51, TID_7) durch T1 wird ein bereits existierender Eintrag erweitert. Irgendwelche Operationen laufender Transaktionen auf diesem Schlüsselwert sind durch den Sperrverwalter registriert, so daß bei Anforderung der neuen Sperre die Verträglichkeit der Operation überprüft werden

kann. Die Überprüfung des nächsten Schlüsselwertes kann also entfallen (Einfügen und Löschen brauchen die Lücke nicht zu beachten.).

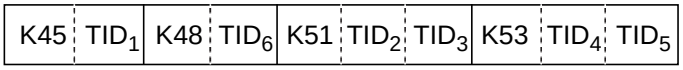


Wenn K51 durch T1 mit S belegt war, erfolgt eine Konversion in SIX.

Vollständiges Löschen eines Schlüsselwertes

Falls durch das Löschen ein Schlüsselwert vollständig verschwindet, ergibt sich durch diese Operation eine Lücke, die durch Abort beseitigt werden kann. Für andere Transaktionen muß deshalb auf dem nächsten Schlüssel eine geeignete Sperre hinterlassen werden, deren Beachtung Konsistenzprobleme verhindert. Lock (K_i, X) auf dem nächsten Schlüsselwert leistet das Gewünschte.

In der Situation



kann der Eintrag K48 wie folgt gelöscht werden. Dabei ist durch eine Instant-Sperranforderung zu prüfen, ob der Wert noch von einer anderen Transaktion bearbeitet wird.

Lock (K51, X)
Lock (K48, X, I)
Löschen von K48

Löschen in einem Eintrag

Falls durch das Löschen der Eintrag erhalten bleibt, braucht der nächste Eintrag durch die oben genannten Gründe nicht beachtet zu werden. Der Schlüsselwert erhält dann eine lange X-Sperre:

Lock (K51, X)

Zusammenfassung

Die Sperrprotokolle bei Indexstrukturen für Sekundärschlüssel (Nonunique Index) können für Leseoperationen vom Fall des eindeutigen Index (Unique Index) übernommen werden. Die Hauptunterscheidung bei Aktualisierungsoperationen ist darin zu treffen, ob ein Schlüsseleintrag neu in den Index aufgenommen wird oder ganz verschwindet oder der Eintrag nur modifiziert wird.

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) nur wenn K_i neu eingefügt wird, sonst nicht
Einzufügender Schlüsselwert K_i	Lock (K_i, IX) - wenn ein Eintrag für K_i bereits existiert oder - wenn ein Eintrag für K_i noch nicht existiert und der nächste Schlüsselwert nicht schon durch T_1 in X-, S- oder SIX-Modus gehalten wird Lock (K_i, X) - wenn ein Eintrag für K_i noch nicht existiert und der nächste Schlüsselwert schon durch T1 im X, S-, oder SIX-Modus gehalten wird

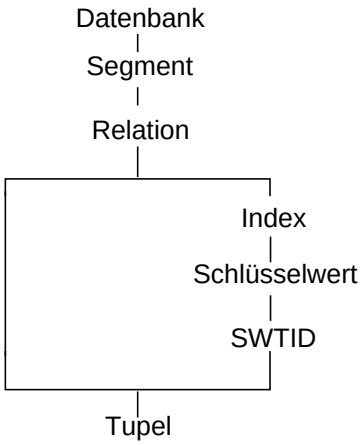
T1:Delete (K_i, TID)

Wurzel oder Zwischenknoten	Latch (P_i, S)
Blattknoten	Latch (P_j, X)
Nächster Schlüsselwert K_j	Lock (K_j, X) - nur wenn durch dieses Delete der Eintrag für K_i vollständig gelöscht wird
Zu löschender Schlüsselwert K_i	Lock (K_i, X, I) - wenn der Eintrag für K_i vollständig gelöscht wird Lock (K_i, X) - sonst

5.7 Verfeinerung des Sperrgranulates

Bisher wurde genommen, daß das kleinste Sperrgranulat beim Indexzugriff ein einzelner Schlüsselwert mit allen zugehörigen TIDs ist. Im Falle von NONUNIQUE können teilweise recht lange TID-Listen auftreten, auf denen parallele Operationen verschiedener Transaktionen durchaus wünschenswert sein können. Beim Sperrgranulat “Schlüsselwert” impliziert nämlich das Einfügen oder Löschen eines TIDs die Nichtverfügbarkeit des gesamten Eintrags bis zum Commit der modifizierenden Transaktion.

Die Verfeinerung der Index-Sperrhierarchie läßt sich durch ein selbständiges Granulat Schlüsselwert-TID (SWTID) erreichen, so daß einzelne Schlüsselwert-TID-Paare in einem Eintrag isoliert gesperrt werden können. Der folgende azyklische Sperrgraph veranschaulicht die Einordnung des vorgeschlagenen Sperrgranulates.



Damit kann nun nach Bedarf ein gesamter Index, ein Eintrag im Index oder nur ein Schlüsselwert mit einem TID gesperrt werden. Wenn nun im Index IID1 (K55, TID_1) gelöscht werden soll, so ist folgendes Sperrprotokoll (ohne Latch-Anforderung) denkbar.

```
T1:  . . .
     Lock (IID1, IX)
     Lock (K55, IX)
     Lock (K55TID1, X)
     . . .
```

Die Lock-Namen müßten genauer mit IID1K55 und IID1K55TID_1 angegeben werden. Wir verzichten wie bisher auf diese umständliche Darstellung.

Beim Lesen wird typischerweise ein Schlüssel mit einem Prädikat (Bereichsangabe) spezifiziert und nicht mit einer TID-Angabe. Deshalb werden Sperren auf der Schlüsselwertebene oder bei großen Bereichen gar auf Indexebene gesetzt. Die Regeln dazu wurden in den vorigen Abschnitten diskutiert.

Die Verfeinerung des Sperrgranulates kann deshalb offensichtlich nur bei Modifikationsoperationen wirksam werden. Zur Illustration gehen wir vom einfachsten Fall aus, daß durch die Operation der Eintrag mit dem betreffenden Schlüsselwert nicht neu eingefügt wird oder aus der Indexstruktur verschwindet. Die folgenden Operationen modifizieren den Eintrag K55. Sie beanspruchen nur während der Latch/Unlatch-Spanne exklusiven Zugriff auf die K55-TID-Liste, ansonsten können sie parallel abgewickelt werden.

```
T1: Insert (K55, TID1)        T2: Delete (K55, TID2)        T1: Insert (K55, TID3)
    . . .                         . . .                         . . .
    Lock (IID1, IX)               Lock (IID1, IX)               Lock (IID1, IX)
    Lock (K55, IX)                Lock (K55, IX)                Lock (K55, IX)
    Lock (K55TID1, X)             Lock (K55TID2, X)             Lock (K55TID3, X)
    Einfügen                      Löschen                       Einfügen
```

Zur Ableitung vollständiger Sperrprotokolle sind folgende Beobachtungen wichtig:

- Lesesperren werden auf der Schlüsselwert-Ebene (SW-Ebene) gesetzt und auf Verträglichkeit verglichen.
- Auf der SW-Ebene findet auch die Kommunikation von Sperrzuständen statt. Die Überprüfung des nächsten Schlüssels verhindert bei einem vorherigen Löschen in der Lücke ein Einfügen in der Lücke oder ein Lesen des nächsten Schlüssels. Andererseits wird bei vorherigem Lesen des nächsten Schlüssels eine Einfügung in der Lücke davor verhindert.
- Ein Transfer der Bereichssperrenfunktion findet nur statt, wenn ein Schlüssel (Eintrag) neu hinzukommt (Lücke wird geteilt) oder ganz verschwindet (Lücke wird vergrößert).
- Wenn ein Schlüssel neu hinzukommt oder ganz verschwindet, ist ohnehin keine Parallelität auf SW-TID-Ebene möglich. Deshalb können für diese Fälle die bekannten Protokolle übernommen werden.

Zusammengefaßt können die Einfüge- oder Löschprotokolle auf SW-Ebene folgendermaßen dargestellt werden. K_i sei der aktuelle und K_j der nächste Schlüsselwert:

Insert (K_i, TID)

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

Delete (K_i, TID)

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

Tabelle 3: Einfüge- und Löschprotokolle auf SW-Ebene

Durch diese Zusammenstellung wird deutlich, daß also nur die beiden markierten Fälle (Liste: sonst) für die weitere Optimierung herangezogen werden können. Der Sperrzustand des nächsten Schlüssels K_j braucht dabei nicht beachtet zu werden.

Wie bereits im obigen Beispiel angedeutet, können Einfügen und Löschen parallel ausgeführt werden, wenn die TID-Liste nicht neu erzeugt oder leer wird:

Insert (K_i,TID)

Sperranforderung	Liste ≥ 1 TID	Sperrmodus
für K_i	ja	IX
für K_iTID	ja	X

Delete (K_i,TID)

Sperranforderung	Liste > 1 TID	Sperrmodus
für K_i	ja	IX
für K_iTID	ja	X

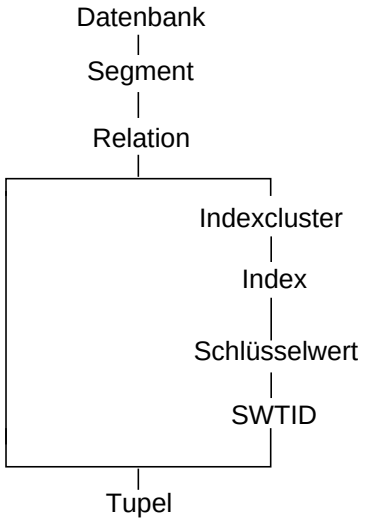
Tabelle 4: Optimiertes Einfüge- und Löschprotokoll für die SWTID-Ebene

Betrachtet man die Sperranforderung für K_i (auf SW-Ebene), so erkennt man, daß Einfügungen verschiedener Transaktionen in eine TID-Liste durch den IX-Modus nicht verhindert werden. Da TIDs jedoch nicht verglichen werden, könnten durch dieses Protokoll gleiche TIDs durch verschiedene Transaktionen eingefügt werden.

Nebenbei sei hier noch vermerkt, daß in Tabelle 3 die Zeilen (Liste:neu und Liste:letztes TID) den Fall eines UNIQUE-Index beschreiben (Tabelle 2).

5.8 Verallgemeinerte Sperrhierarchie für Indexzugriffe

Wir bezeichnen die Menge der Indexstrukturen, die für eine Relation existieren, auch als Indexcluster. Will man ein Indexcluster als Ganzes sperren, so kann man den Typ Indexcluster in die Sperrhierarchie aufnehmen. Dabei wird unterstellt, daß ein Indexcluster zusammen mit der Relation in einem Segment untergebracht ist.



Auf diese Weise können konkrete Indexcluster, Indexstrukturen, Schlüsselwerte oder nur Schlüsselwert-TID-Paare zum Lesen oder Schreiben gesperrt werden. Eine X-Sperre auf (K55, TID_1) in Index IID1 und im Indexcluster IC1 wird dabei wie folgt angefordert:

```
T1:   . . .
      Lock (IC1, IX)
      Lock (IID1, IX)
      Lock (K55, IX)
      Lock (K55TID1, X)        Das ist eine Sperre für einen SWTID-Eintrag
      . . .
```

Ob die Sperrhierarchie in voller Allgemeinheit realisiert werden sollte, hängt von den Bedürfnissen der Anwendungen ab. In jedem Fall muß der hierarchische Pfad Index - Schlüsselwert bereitgestellt werden.

5.9 Bemerkungen zur Konsistenzebene 2

Eine Transaktion T1, die auf Konsistenzebene 2 arbeitet, hält ihre Lesesperren nicht immer bis zu Commit(T1). Sie ist deshalb auch nicht serialisierbar. Andere Transaktionen können jedoch parallel auf Konsistenzebene 3 arbeiten, ohne gestört zu werden. Sperrprotokolle für Transaktionen auf Konsistenzebene 2 gewährleisten Stabilität von Cursorpositionen (cursor stability).

Wie können nun die Sperrprotokolle auf Indexstrukturen für solche Transaktionen modifiziert werden, damit der Grad an Parallelität für Leser und Schreiber so weit wie möglich gesteigert werden kann?

Die Protokolle für Aktualisierungsoperationen müssen offensichtlich erhalten bleiben, weil sonst für Transaktionen auf Konsistenzebene 3 keine Serialisierbarkeit erreicht werden kann. Sperrprotokolle für Leser vereinfachen sich jedoch; der nächste Schlüsselwert braucht dabei nicht mehr beachtet zu werden, da Wiederholbarkeit von Lesevorgängen nicht angestrebt wird. Die Protokolle können wie folgt skizziert werden:

T1:	Fetch (K_i)	
	Wurzel oder Zwischenknoten	Latch (P_i,S)
	Blattknoten	Latch (P_j,S)
	Schlüsselwert	Lock (K_i, S, I)

Die Protokolle für Fetch Next und Fetch Prior sehen im Prinzip gleich aus.

6. Verallgemeinerung der Zugriffspfadstrukturen

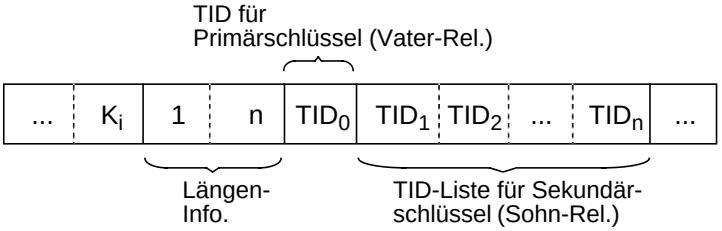
Werden Primärschlüssel- und zugehöriger Fremdschlüsselzugriff durch zwei Indexstrukturen (UNIQUE und NONUNIQUE), z.B. I_V(A1) und I_S(A1), unterstützt, so müssen bei entsprechenden Änderungsoperationen typischerweise beide Strukturen nacheinander durchlaufen werden, um die Relationalen Invarianten zu überprüfen. Als Zugriffskosten fallen dann stets die Seitenzugriffe in beiden B*-Baumstrukturen an; dabei sind die Bäume jeweils von der Wurzel zum betreffenden Blatt (h Seiten) zu durchqueren.

Die bisher eingeführten Sperrprotokolle minimieren unter der Nebenbedingung der Serialisierbarkeit das Blockierungs- und Konfliktpotential von parallelen Operationen auf einer Indexstruktur. Die Zugriffskosten selbst sind davon kaum berührt.

6.1 Kombinierte Zugriffspfadstruktur

Da Zugriffe auf Primär- und zugehörigen Fremdschlüssel sehr häufig als Teil von komplexeren Operationen erfolgen, bietet es sich an, die entsprechenden Indexstrukturen durch einen gemeinsamen B*-Baum zu implementieren [Hä78]. Dabei bleibt der Aufbau von Wurzel und Zwischenknoten erhalten, lediglich die Blattknoten erhalten eine andere Organisationsform. Ein Eintrag setzt sich aus einem Schlüs-

selwert, dem TID als Verweis auf die Vaterrelation und der TID-Liste mit den Verweisen auf die Sohnrelation zusammen. Blattknoten erhalten dadurch folgende Struktur:



Die Verweis-Information, die Primär- und Fremdschlüssel betrifft, kann jetzt mit Hilfe eines Baumdurchlaufs lokalisiert werden. Da die Höhe von B*-Bäumen sehr stabil gegenüber Einfügungen in den Blattknoten ist, kann in der Regel davon ausgegangen werden, daß sich durch das Zusammenlegen die Höhe im Vergleich zur separaten Struktur nicht ändert, daß also bei der kombinierten Zugriffspfadstruktur nur h Seitenzugriffe anfallen. Da jetzt noch mehr Durchläufe der B*-Baumstruktur erforderlich werden, erhöht sich damit auch die Lokalität der Referenzen auf die Baumseiten, was prinzipiell die Zugriffskosten weiter reduziert. Jedoch steigen damit auch die Anforderungen an die Synchronisationsprotokolle beim Indexzugriff. Nach Möglichkeit sollten sich die Konflikte oder Blockierungen im Vergleich zu zwei separaten Strukturen nur unwesentlich verschlechtern.

Im Prinzip lassen sich die optimierten Sperrprotokolle für Indexstrukturen (UNIQUE und NONUNIQUE) übernehmen. Obwohl nur eine physische Implementierung vorliegt, kann man in Bezug auf das Sperren der Schlüsselwerte von zwei (logischen) Indexstrukturen ausgehen. Baumdurchlauf und Strukturmodifikationen müssen allerdings durch einen gemeinsamen Latch-Mechanismus synchronisiert werden, da alle physischen Manipulationen auf einer Baumstruktur bewerkstelligt werden.

Im typischen Fall, wo zwei Indexstrukturen (IID1 und IID2) kombiniert werden, kann das Einfügen eines neuen Fremdschlüssels (K55, TID) in IID2 und das gleichzeitige Überprüfen des Primärschlüssels in IID1 durch folgendes Sperrprotokoll synchronisiert werden:

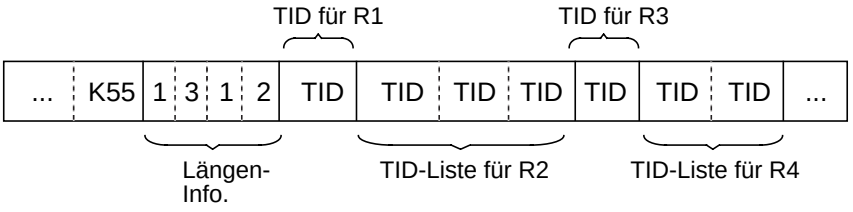
T1:	. . .	
	Lock (IID1, IS)	Sperren der Indexhierarchie
	Lock (IID2, IX)	
	Latch (P1, S)	Baumdurchlauf
	Latch (P2, S)	
	Unlatch (P1)	
	Latch (P3, X)	
	Unlatch (P2)	
	Lock (IID2K55,IX)	bis Commit (T1)
	Lock (IID1K55,S,I)	Existenztest für Vaterschlüssel
	. . .	Einfügen von K55, TID in IID2-Liste
	Unlatch (P3)	
	. . .	

Bei der Anforderung der Sperren für die Indexhierarchie wird deutlich, daß es sich um zwei logische Indexstrukturen IID1 und IID2 handelt. Der Baumdurchlauf zum gemeinsamen Ziel K55 geschieht mit Hilfe von speziellen Kurzzeitsperren (Latch-Kopplung). Für das Sperren der Indexeinträge kann wiederum so verfahren werden, als lägen separate Einträge mit verschiedenen Schlüsseln (IID1K55 und IID2K55) vor. Die bisher abgeleiteten Sperrprotokolle können direkt mit allen Optimierungen so übernommen werden, daß für Primärschlüssel die UNIQUE- und für Fremdschlüssel (Sekundärschlüssel) die NONUNIQUE-Option herangezogen wird (siehe Tabellen 2 und 3).

6.2 Verallgemeinerte Zugriffspfadstruktur

Das Konzept der kombinierten Zugriffspfadstruktur kann in der folgenden Weise zur verallgemeinerten Zugriffspfadstruktur erweitert werden: Im Relationenmodell können mehrere Attribute verschiedener Relationen auf einem Wertebereich (Domain) definiert sein. Für die invertierten Attribute können nun alle variabel langen TID-Listen zusammen mit dem Schlüsselwert in einem Eintrag zusammengefaßt werden. Es ändert sich dabei wiederum nur das Blattformat des zugehörigen B*-Baums.

Beispielsweise seien vier Attribute der Relationen R1 - R4 auf dem Wertebereich ANR definiert und invertiert. In zwei Relationen sei das entsprechende Attribut Schlüsselkandidat:



Im allgemeinen Fall können also n logische Indexstrukturen durch einen B*-Baum implementiert sein. Die eingeführten Sperrprotokolle sind dabei analog zur kombinierten Zugriffspfadstruktur einzusetzen. Latch-Kopplung wird zum Durchlauf und zur Manipulation des B*-Baumes herangezogen. Die Sperrprotokolle können dann so angewendet werden, als seien n Indexstrukturen separat verfügbar. Dabei können nebeneinander die Optionen UNIQUE und NONUNIQUE benutzt werden; die zugehörige Information ist der Beschreibung der Zugriffspfadstruktur in den Metadaten zu entnehmen.

Die bisherigen Überlegungen sind nicht beschränkt auf einfache Attribute und Wertebereiche. In gleicher Weise kann man das zugrundeliegende Konzept auf zusammengesetzte Attribute, die auf einer Folge von Wertebereichen definiert sind, anwenden. Ein Schlüsselwert besteht dann aus der Konkatenation der einzelnen Attributwerte, wobei die Index-ID noch als Präfix genommen wird. Dabei läßt sich der aus n Einzelwerten variabler Länge bestehende Schlüsselwert so codieren, daß durch einen Vergleich die Sortierfolge zweier Schlüsselwerte festgestellt werden kann [BCE77, Hä78a]. Selbst wenn die Einzelwerte nach aufsteigender und/oder absteigender Sortierfolge eingeordnet werden sollen, kann das Codierschema angewendet werden.

6.3 Join-Index

In [Va87] wurden sogenannte Join-Indizes als spezielle Zugriffspfade vorgeschlagen, insbesondere zur Optimierung von Zwei-Wege-Joins. Da die Überprüfung der Referentiellen Integrität den Zugriff auf dieselben Tupelmengen wie die Join-Berechnung über Fremdschlüssel erfordert, soll im folgenden untersucht werden, inwieweit Join-Indizes die Einhaltung der Referentiellen Integrität effizient unterstützen.

Der Join-Index zwischen zwei Relationen V und S ist folgendermaßen definiert:

$$JI = \{ (v.TID, s.TID) \mid f(v.A, s.B) \text{ ist TRUE}, v \in V, s \in S\}.$$

Dabei bezeichnet f ein beliebiges Join-Prädikat. Der Join-Index zu f entspricht also einer zweistelligen Relation, die zu jedem Element des Join-Ergebnisses die TIDs der beiden zugehörigen Tupel aus S und V enthält. Um einen schnellen Zugriff auf das Join-Ergebnis über S- und V-Werte zu gestatten, werden in der Regel zwei (geclusterte) B*-Bäume pro Join-Index als Einstiegsstruktur benötigt. Das Beispiel in Bild 5 zeigt den Join-Index für einen Fremdschlüssel-Join, der durch zwei geclusterte B*-Bäume repräsentiert ist, die den direkten Zugriff über die TIDs der Vater- bzw. Sohn-Relation gestatten.

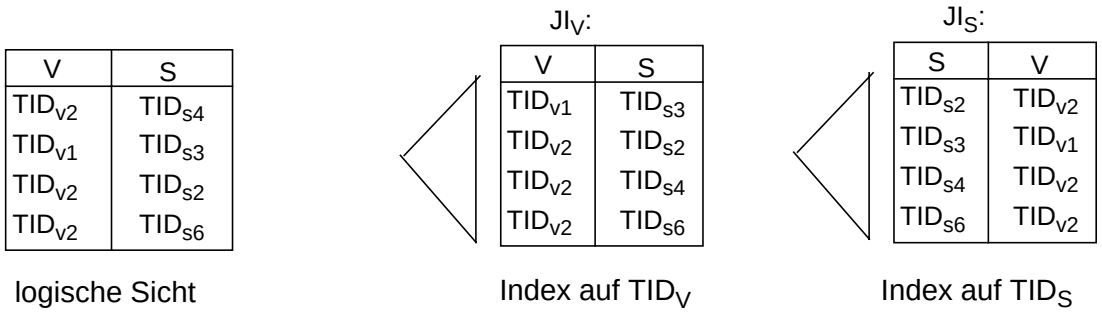


Bild 5. : Join-Index für Fremdschlüssel-Join (logische Sicht und physische Repräsentation)

Durch die Verwendung von TIDs anstatt von Schlüsselwerten kann typischerweise eine große Anzahl von Indexeinträgen (Join-Index-Tupel) pro Seite untergebracht werden, so daß die Baumhöhe meist auf 2 beschränkt ist. Andererseits werden i.d.R. zusätzliche Zugriffspfade benötigt, um zu einem Schlüsselwert die zugehörigen TIDs zu bestimmen. In Anfragen entstehen zur Ermittlung der TIDs jedoch meist keine Zusatzkosten, da sie im Zuge der Auswertung von Selektionsprädikaten festgestellt werden. Da Join-Indizes auf B*-Bäumen aufbauen, können die eingeführten Sperrprotokolle auch hierbei eingesetzt werden.

Der Join-Index kann nicht zur Überprüfung der Eindeutigkeit von Schlüsselkandidaten genutzt werden, da er keine Schlüsselwerte, sondern nur TIDs verwendet. Hierzu müssen daher herkömmliche Indexstrukturen zusätzlich eingesetzt werden. Zur Überwachung der Referentiellen Integrität ist für jeden Fremdschlüssel ein Join-Index anzulegen. Überprüfungen beim Einfügen/Ändern eines Sohn-Tupels werden damit jedoch nicht unterstützt, da hierbei zu überprüfen ist, ob zu einem bestimmten Fremdschlüssel*wert* ein zugehöriges Tupel in der Vater-Relation existiert. Beim Löschen bzw. Ändern eines

Vater-Tupels kann über den Join-Index jedoch schnell auf alle abhängigen Sohn-Tupel zugegriffen (CASCADE, SET NULL/DEFAULT) bzw. deren Existenz festgestellt werden (RESTRICTED).

Der Join-Index repräsentiert also einen Indextyp, der zusätzlich zu den Zugriffspfaden für Primär- und Sekundärschlüssel gewartet werden muß. Bezüglich der Referentiellen Integrität bietet er nur eine begrenzte Zugriffsbeschleunigung, insbesondere wenn man bedenkt, daß die ohnehin notwendigen Indexstrukturen auf Schlüsselwerten die notwendigen Integritätsprüfungen bereits weitgehend unterstützen (Kap. 3).

7. Abschätzung des Zusatzaufwandes

Zur Bewertung der Ausführungskosten zur Wahrung der Relationalen Invarianten sollen einfache Kostenformeln entwickelt werden. Wir bestimmen den Mehraufwand für die eingeführten Basisoperationen mittels zweier Kostenmaße:

1. *Anzahl zusätzlicher logischer Seitenzugriffe*
   Damit wird der zusätzliche E/A-Bedarf zumindest in Annäherung bestimmt, wenngleich nicht jede logische Seitenreferenz eine physische E/A zur Folge hat (Zugriffslokalität). Daneben ist der zusätzliche Instruktionsbedarf in der Regel proportional zur Anzahl der Seitenzugriffe.

2. *Anzahl zusätzlicher Sperranforderungen*
   Dieses Maß gestattet eine Abschätzung des zusätzlichen Instruktionsbedarfs zur Synchronisation. Zudem steigt die Konfliktgefahr i.d.R. mit der Anzahl zu setzender Sperren. Die Häufigkeit der Latch-Aufrufe wird nicht berücksichtigt, da sie vergleichsweise geringe Kosten und eine geringe Konfliktwahrscheinlichkeit verursachen.

Mit den Kostenformeln sollen zum einen die relativen Zusatzkosten der verschiedenen Änderungsoperationen verglichen sowie für die Leistungsfähigkeit kritische Parameter identifiziert werden. Dies erlaubt Rückschlüsse darauf, für welche Operationen/Parameter eine Optimierung besonders wichtig ist. Da die Zusatzkosten in Abhängigkeit der eingesetzten Zugriffspfadtypen bestimmt werden, ist es darüber hinaus möglich, verschiedene Zugriffspfadtypen hinsichtlich des bei ihnen entstehenden Mehraufwandes zur Wahrung der Referentiellen Integrität miteinander zu vergleichen.
Die Kostenformeln gestatten jedoch nicht, den Mehraufwand im Vergleich zur tatsächlichen DB-Verarbeitung abzuschätzen, da hierzu weitreichende Annahmen bezüglich des Lastprofils getroffen werden müßten. Aus ähnlichen Gründen ist es auch nicht möglich, quantitative Aussagen über das Sperrverhalten (Häufigkeiten und Dauer von Sperrkonflikten u.ä.) zu treffen. Hierzu wären detaillierte Implementierungen der Zugriffspfade und Sperrverfahren innerhalb von Simulationsmodellen bzw. innerhalb von Prototypen erforderlich sowie eine Leistungsbewertung anhand verschiedener Anwendungsprofile durchzuführen.

Die Aufwandsabschätzungen erfolgen für die in Kap. 3 eingeführten Änderungsoperationen auf der Sohn-Relation S sowie der Vater-Relation V. Die Basiskosten für diese Operationen sollen dabei unberücksichtigt bleiben, da sie ggf. die Auswertung beliebiger Prädikaten erfordern, um zu löschende bzw. zu ändernde Sätze zu lokalisieren. In jedem Fall fällt der Zugriff auf die Tupel selbst an, wobei gemäß der Höhe der Sperrhierarchie eine entsprechende Anzahl von Sperren zu setzen ist. Daneben ist auf gewisse Hilfsdaten zuzugreifen, z.B. zur Anpassung von Freispeicherinformationen bei Einfüge- und Löschoperationen.

Bei der Bestimmung der Zusatzkosten zur Wahrung der Relationalen Invarianten setzen wir voraus, daß für jeden Primärschlüssel, Schlüsselkandidaten sowie Fremdschlüssel ein Index (B^*-Baum) vorhanden ist. Wäre dies nicht der Fall, so müßte bereits zur Überprüfung der Eindeutigkeit von Schlüsselwerten ein Relationen-Scan in Kauf genommen werden, was einen prohibitiv hohen Aufwand verursachen würde. Ebenso würde die Überprüfung, ob ein Fremdschlüsselwert definiert ist, einen Relationen-Scan in der Vater-Relation erfordern. Weiterhin müßte bei Löschoperationen (sowie einem Teil der Änderungsoperationen) der Vater-Relation jeweils ein Relationen-Scan in der Sohn-Relation zur Bestimmung abhängiger Tupel durchgeführt werden, falls kein Fremdschlüsselindex vorliegt. Relationen-Scans implizieren ferner eine Sperre auf der gesamten Relation für eine lange Dauer, wodurch starke Sperrwartezeiten eingeführt werden.

Bei Indexnutzung kann für die Abschätzung des Mehraufwandes zur Einhaltung der Relationalen Invarianten zwischen direkten und indirekten Zusatzkosten unterschieden werden. *Direkte Zusatzkosten* betreffen die Überprüfung und Einhaltung der Eindeutigkeit von Primärschlüssel und Schlüsselkandidaten sowie der Referentiellen Integrität für Fremdschlüssel. *Indirekte Zusatzkosten* entstehen durch die notwendige Aktualisierung der Indexstrukturen auf Schlüsselkandidaten und Fremdschlüssel, welche zur Integritäts-überwachung verwendet werden. Werden diese Indexstrukturen (nahezu) ausschließlich zur schnelleren Überprüfung der Relationalen Invarianten geführt, so sind die Aktualisierungskosten für sie als “echter” Mehraufwand zu rechnen. Profitieren dagegen Leseoperationen auch verstärkt von diesen Indexstrukturen, z.B. zur Join-Berechnung, Auswertung von Built-In-Funktionen etc., stellen die Kosten zur Indexaktualisierung im Prinzip keinen Mehraufwand zur Wahrung der Relationalen Invarianten dar. Da eine klare Unterscheidung zwischen echten und unechten Zusatzkosten anwendungsunabhängig offenbar nicht möglich ist, weisen wir die direkten und (maximalen) indirekten Zusatzkosten getrennt aus. Weiterhin unterscheiden wir bei den Seitenzugriffen zwischen lesenden und schreibenden Referenzen. Damit werden für jede der sechs Änderungsoperationen sechs Kostenformeln entwickelt, die mit $IO_{<op>\text{-read-direkt}}$, $IO_{<op>\text{-read-indirekt}}$, $IO_{<op>\text{-write-direkt}}$, $IO_{<op>\text{-write-indirekt}}$,$LOCK_{<op>\text{-direkt}}$ und $LOCK_{<op>\text{-indirekt}}$ bezeichnet werden. IO kennzeichnet dabei die Anzahl zusätzlicher logischer Seitenzugriffe und LOCK die zusätzliche Anzahl von Sperraufrufen. Der Subskript spezifiziert die jeweilige Operation (S1, S2, S3, V1, V2, V3) sowie ob es sich um Lese- oder Schreibzugriffe bzw. um direkte oder indirekte Zusatzkosten handelt.

Für die Bestimmung der Zusatzkosten werden folgende allgemeine Parameter verwendet:

s	Anzahl der Schlüsselkandidaten in S (inklusive Primärschlüssel)
v	Anzahl der Schlüsselkandidaten in V (inklusive Primärschlüssel)
f	Anzahl der Fremdschlüssel in S
N_S, N_V	Anzahl der Tupel in Relation S bzw. V
$u_{S\text{-}SK}$, $u_{S\text{-}FS}$	mittl. Anzahl geänderter Schlüsselkandidaten bzw. Fremdschlüssel bei einer Tupeländerung in S ($u_{S\text{-}SK} \leq s$; $u_{S\text{-}FS} \leq f$)
$u_{V\text{-}SK}$, $u_{V\text{-}FS}$	mittl. Anzahl geänderter Schlüsselkandidaten in V ($u_{V\text{-}SK}$) bzw. geänderter Schlüsselkandidaten mit zugehörigem Fremdschlüssel in S ($u_{V\text{-}FS}$) bei einer Tupeländerung in V ($u_{V\text{-}SK} \leq v$; $u_{V\text{-}FS} \leq f$).

Die Zusatzkosten werden im folgenden für B*-Bäume (7.1) sowie kombinierte Zugriffspfade (7.2) bestimmt.

7.1 Zusatzkosten bei B*-Bäumen

Für jeden Index werden noch folgende Parameter benötigt:

$2k_I$, $2k^*_I$	max. Anzahl von Indexeinträgen pro Seite für Index I (innere bzw. Blattknoten).

Die Werte für k und k^* lassen sich in einfacher Weise aus der Schlüssellänge und der Seitengröße bestimmen. Mit k, k^* sowie der Tupelanzahl N kann für jeden Index die Baumhöhe h^* wie folgt ermittelt werden:

$$h^* \leq 2 + \log_{k+1} (N / (2k^*)).$$

Aus den Baumhöhen lassen sich folgende Mittelwerte ableiten, die in unseren Kostenformeln Anwendung finden werden:

H_S	mittlere Baumhöhe der s+f Indexstrukturen in S zu Schlüsselkandidaten und Fremdschlüsseln in S
H_V	mittlere Baumhöhe der v Indexstrukturen in V zu den Schlüsselkandidaten
$H_{S\text{-}FS}$	mittlere Baumhöhe der f Indexstrukturen in S zu Fremdschlüsseln in S
$H_{V\text{-}FS}$	mittlere Baumhöhe der f Indexstrukturen in V zu Fremdschlüsseln in S.

S1: Einfügen eines Tupels in S

Die Eindeutigkeitsprüfungen für die Schlüsselkandidaten von S erfordern keinen nennenswerten Mehraufwand, da sie im Rahmen der für das Einfügen ohnehin erforderlichen Indexzugriffe für die Schlüsselkandidaten vorgenommen werden können. Ein Mehraufwand entsteht jedoch für jeden Fremdschlüssel,

da die Existenz des zugehörigen Wertes in der Vater-Relation überprüft werden muß. Diese Prüfungen erfolgen ausschließlich auf den Indexstrukturen für die Schlüsselkandidaten von V.

Die Anzahl der dafür notwendigen lesenden Seitenzugriffe beträgt

$$IO_{S1\text{-}read\text{-}direkt} = f * H_{V\text{-}FS.}$$

Für jeden der f Fremdschlüssel ist also ein Baumdurchlauf auf dem zugehörigen Index von V (mittlere Baumhöhe $H_{V\text{-}FS}$) erforderlich. Die Anzahl der Seitenzugriffe entspricht der Summe der jeweiligen Baumhöhen. Schreibzugriffe fallen keine an, da nur die Existenz des Fremdschlüsselwertes zu überprüfen ist:

$$IO_{S1\text{-}write\text{-}direkt} = 0.$$

Für jeden Fremdschlüssel ist daneben eine Anwartschaftssperre auf dem jeweiligen Index von V sowie eine (kurze) Lesesperre auf dem betreffenden Indexeintrag zu setzen. Es ergibt sich also ingesamt folgende Anzahl von zusätzlichen Sperranforderungen:

$$LOCK_{S1\text{-}direkt} = 2\ f.$$

Indirekte Zusatzkosten können durch die Aktualisierung der vorhandenen Indexstrukturen entstehen. Als maximale Zusatzkosten hinsichtlich der Relationalen Invarianten ergeben sich:

$$IO_{S1\text{-}read\text{-}indirekt} = H_S\ (s + f)$$

$$IO_{S1\text{-}write\text{-}indirekt} = (s+f)\ (1 + 3/(2k^*))$$

$$LOCK_{S1\text{-}indirekt} = 3\ (s + f)$$

Für jede der s+f Indexstrukturen in S zu Schlüsselkandidaten und Fremdschlüsseln erfordert das Einfügen einen Baumdurchlauf, um die korrekte Blattseite zu lokalisieren (H_S Lesezugriffe pro Index). Kann die Einfügung des Indexeintrages ohne Split-Vorgang erfolgen, ist lediglich eine Seite zu ändern, bei Split-Vorgängen dagegen sind Änderungen in (wenigstens) drei Seiten erforderlich. Die Wahrscheinlichkeit eines Split-Vorganges beträgt $1/ (2k^*)$, da eine Blattseite $2k^*$ Einträge aufnehmen kann. Daneben müssen jeweils drei Sperren pro Indexstruktur gesetzt werden (Anwartschaftssperre auf Index sowie zwei Schlüsselwertsperren für den neuen Eintrag sowie den Nachfolgerschlüssel).

S2: Löschen eines Tupels in S

Es entsteht kein direkter Mehraufwand zur Wahrung der Relationalen Invarianten:

$$IO_{S2\text{-}read\text{-}direkt} = 0$$

$$IO_{S2\text{-}write\text{-}direkt} = 0$$

$$LOCK_{S2\text{-}direkt} = 0.$$

Die indirekten Kosten zur Anpassung der Indexstrukturen für Schlüsselkandidaten und Fremdschlüssel entsprechen weitgehend denen zum Einfügen eines Tupels (S1):

$$IO_{S2\text{-}read\text{-}indirekt} = H_S\ (s + f)$$

$$IO_{S2\text{-}write\text{-}indirekt} = s + f$$

$$LOCK_{S2\text{-}indirekt} = 3\ (s + f).$$

Zusätzliche Änderungskosten zum vollständigen Löschen von Blattseiten nach Entfernen des letzten Indexeintrages (Anpassung der Verweise in benachbarten Seiten) wurden ignoriert.

S3: Ändern eines Tupels in S

Änderungen des Primärschlüssels bzw. eines Schlüsselkandidaten erfordern die Überprüfung der Eindeutigkeit, die wie für S1 ohne signifikanten Mehraufwand mit den jeweiligen Indexstrukturen erfolgen kann. Für Änderungen von Fremdschlüsseln entstehen Kosten analog zu S1; statt allen f sind lediglich $u_{S\text{-}FS}$ Fremdschlüssel betroffen:

$$IO_{S3\text{-}read\text{-}direkt} = u_{S\text{-}FS} * H_{V\text{-}FS}$$

$$IO_{S3\text{-}write\text{-}direkt} = 0$$

$$LOCK_{S3\text{-}direkt} = 2\ u_{S\text{-}FS}.$$

Die Kosten zur Anpassung der Indexstrukturen entsprechen denen zum Löschen und Einfügen eines Indexeintrages; statt s (f) Schlüsselkandidaten (Fremdschlüssel) sind lediglich $u_{S\text{-}SK}$ ($u_{S\text{-}FS}$) betroffen:

$$IO_{S3\text{-}read\text{-}indirekt} = 2\ H_S\ (u_{S\text{-}SK} + u_{S\text{-}FS})$$

$$IO_{S3\text{-}write\text{-}indirekt} = 2\ (u_{S\text{-}SK} + u_{S\text{-}FS})$$

$$LOCK_{S3\text{-}indirekt} = 5\ (u_{S\text{-}SK} + u_{S\text{-}FS})$$

Split-Vorgänge sind bei Änderungsoperationen nicht zu erwarten. Pro Indexstruktur fallen maximal 5 Sperren an: eine Anwartschaftssperre für den Index sowie jeweils zwei Schlüsselwertsperren für den zu löschenden und neu einzufügenden Indexeintrag.

V1: Einfügen eines Tupels in V

Da die Eindeutigkeitsprüfungen für die Schlüsselkandidaten von V keinen nennenswerten Mehraufwand erfordern, gilt:

$$IO_{V1\text{-}read\text{-}direkt} = 0$$

$$IO_{V1\text{-}write\text{-}direkt} = 0$$

$$LOCK_{V1\text{-}direkt} = 0.$$

Für die Aktualisierung der Indexstrukturen entstehen folgende Zusatzkosten:

$IO_{V1\text{-}read\text{-}indirekt} = H_V * v$

$IO_{V1\text{-}write\text{-}indirekt} = v\ (1 + 3/(2k^*))$

$LOCK_{V1\text{-}indirekt} = 3\ v.$

V2: Löschen eines Tupels in V

Für diese Operation (wie auch bei V3) sind Indexaktualisierungen sowohl in der Vater- als in der Sohn-Relation erforderlich. Um die zur Wahrung der Referentiellen Integrität erforderlichen Kosten deutlicher zu machen, nehmen wir sämtliche Folgekosten bezüglich der Sohn-Relation in die direkten Zusatzkosten auf. In den indirekten Zusatzkosten geht daher nur der Aufwand zur Aktualisierung der Indexstrukturen in V ein. Diese Kosten entsprechen (bis auf die Split-Kosten) denen von Operation V1:

$IO_{V2\text{-}read\text{-}indirekt} = H_V * v$

$IO_{V2\text{-}write\text{-}indirekt} = v$

$LOCK_{V2\text{-}indirekt} = 3\ v.$

Die direkten Zusatzkosten zur Wahrung der Referentiellen Integrität hängen von der jeweiligen Löschregel (RESTRICTED, SET NULL/DEFAULT, CASCADE) ab.

a) RESTRICTED

In diesem Fall entstehen die geringsten Kosten, da in der Sohn-Relation S nur zu überprüfen ist, ob abhängige Tupel existieren. Dies erfordert lediglich einen Indexzugriff mit kurzer Lesesperre; die S-Tupel selbst werden nicht referenziert. Insbesondere sind keine Änderungen in der Sohn-Relation vorzunehmen. Liegt für jeden der f Fremdschlüssel die Löschregel RESTRICTED vor, gilt:

$IO_{V2\text{-}read\text{-}direkt} = f * H_{S\text{-}FS}$

$IO_{V2\text{-}write\text{-}direkt} = 0$

$LOCK_{V2\text{-}direkt} = 2\ f.$

b) SET NULL / DEFAULT

Hierbei ist für jedes abhängige Tupel in S der Fremdschlüssel auf NULL bzw. auf den Default-Wert zu setzen. Dies erfordert zunächst einen Baumdurchlauf im Fremdschlüsselindex von S zur Lokalisierung des ersten abhängigen Tupels. Danach ist für jedes abhängige Tupel der Indexeintrag sowie das Tupel selbst zu ändern. Die Kosten hierfür betragen bei f Fremdschlüsseln

$$IO_{V2\text{-}read\text{-}direkt} = f\,(2\,H_{S\text{-}FS} + N_S / N_V)$$

$$IO_{V2\text{-}write\text{-}direkt} = f\,(2 + N_S / N_V)$$

$$LOCK_{V2\text{-}direkt} = f\,(5 + 3 + N_S / N_V).$$

Der Quotient N_S / N_V entspricht der mittleren Anzahl abhängiger S-Tupel pro Schlüsselwert in V. Bei den Seitenzugriffen wurden zwei Baumdurchläufe angesetzt: einen zum Lokalisieren der abhängigen Tupel und einen zur Bestimmung der Blattseite, in die die geänderten Indexeinträge einzufügen sind. Daneben wurde pro abhängigem Tupel je einen lesenden und ändernden Seitenzugriff für die Tupeländerung selbst angesetzt. Auf dem Index sind pro Fremdschlüssel eine Anwartschaftssperre sowie vier Schlüsselwertsperren (Löschen des alten und Einfügen des neuen Wertes) erforderlich. Daneben wurden für die Zugriffe auf die abhängigen Tupel 3 Anwartschaftssperren (DB, Segment, Relation) sowie je eine Sperre pro Tupel angesetzt.

c) CASCADE

Diese Option verursacht die höchsten Kosten, da das Löschen der abhängigen Tupel die Anpassung sämtlicher Indexstrukturen von S verlangt. Daneben ist auf die abhängigen Tupel selbst zuzugreifen. Insgesamt ergeben sich folgende Kosten:

$$IO_{V2\text{-}read\text{-}direkt} = f\,(1 + (s + f)\,H_S\,)^*\,N_S / N_V$$

$$IO_{V2\text{-}write\text{-}direkt} = f\,(1 + s + f\,)^*\,N_S / N_V$$

$$LOCK_{V2\text{-}direkt} = f\,(3 + (1+ 3\,(s + f\,)\,)^*\,N_S / N_V).$$

Bei den Seitenzugriffen wurden pro abhängigem Tupel und Indexstruktur ein Baumdurchlauf sowie drei Sperren zum Löschen des Indexeintrages angesetzt. Für den Zugriff auf die abhängigen Tupel selbst wurden jeweils ein weiterer Seitenzugriff sowie vier Sperren berechnet. Nicht berücksichtigt sind weitere Kosten für zusätzliche Indexstrukturen von S (auf anderen Attributen als Schlüsselkandidaten und Fremdschlüssel) sowie zur Anpassung von Freispeicherinformationen u.ä.. Ist das CASCADE über mehrere Stufen fortzusetzen, kommt es natürlich zu einer entsprechenden Vervielfachung der Gesamtkosten.

Es ist klar, daß nicht für alle Fremdschlüssel die gleiche Löschregel gelten muß. Die Gesamtkosten lassen sich in solchen Fällen jedoch leicht aus den angegeben Kostenformeln bestimmen, indem für jede verwendete Löschregel die Teilkosten für die entsprechenden Anzahl von Fremdschlüsseln bestimmt und diese addiert werden.

V3: Ändern eines Tupels in V

Für die indirekten Zusatzkosten zur Anpassung der Indexstrukturen in V ergeben sich die analogen Formeln wie bei Operation S3:

$IO_{V3\text{-}read\text{-}indirekt} = 2\ H_V * u_{V\text{-}SK}$

$IO_{V3\text{-}write\text{-}indirekt} = 2\ u_{V\text{-}SK}$

$LOCK_{V3\text{-}indirekt} = 5\ u_{V\text{-}SK}$.

Direkte Zusatzkosten entstehen nur, wenn ein Schlüsselkandidat (Primärschlüssel) geändert wird, zu dem ein Fremdschlüssel in S definiert ist. Die Kosten hängen dabei, ähnlich wie bei der Löschoperation V2, von der verwendeten Änderungsregel ab (RESTRICTED, SET NULL/DEFAULT, CASCADE).

a) RESTRICTED

Die Kosten in diesem Fall entsprechen denen beim Löschen (V2), da die Operation abgewiesen werden muß, sofern abhängige Tupel in S existieren. Es sind jedoch i.d.R. nicht alle f, sondern nur eine Teilmenge von $u_{V\text{-}FS}$ Fremdschlüsseln betroffen. Es gilt:

$IO_{V3\text{-}read\text{-}direkt} = u_{V\text{-}FS} * H_{S\text{-}FS}$

$IO_{V3\text{-}write\text{-}direkt} = 0$

$LOCK_{V3\text{-}direkt} = 2\ u_{V\text{-}FS}$.

b) SET NULL / DEFAULT

Auch hierbei entsprechen die Zusatzkosten denen beim Löschen (Anpassung in S für $u_{V\text{-}FS}$ statt für f Fremdschlüssel):

$IO_{V3\text{-}read\text{-}direkt} = u_{V\text{-}FS}\ (2\ H_{S\text{-}FS} + N_S / N_V)$

$IO_{V3\text{-}write\text{-}direkt} = u_{V\text{-}FS}\ (2 + N_S / N_V)$

$LOCK_{V3\text{-}direkt} = u_{V\text{-}FS}\ (5 + 3 + N_S / N_V)$.

c) CASCADE

Im Gegensatz zum Löschen sind hier die Zusatzkosten äquivalent zu SET NULL/DEFAULT, da die Fremdschlüsselwerte in S lediglich auf den neuen Wert aus V anstatt auf NULL bzw. den Default-Wert gesetzt werden:

$IO_{V3\text{-}read\text{-}direkt} = u_{V\text{-}FS}\ (2\ H_{S\text{-}FS} + N_S / N_V)$

$IO_{V3\text{-}write\text{-}direkt} = u_{V\text{-}FS}\ (2 + N_S / N_V)$

$LOCK_{V3\text{-}direkt} = u_{V\text{-}FS}\ (5 + 3 + N_S / N_V)$.

Diskussion

Zur besseren Orientierung sind die *direkten* Zusatzkosten in Tabelle 5 zusammengefaßt. Es fällt auf, daß die Zusatzkosten für Sperren und Seitenzugriffe für die verschiedenen Operationen in etwa propor-

tional zueinander verlaufen. Die “billigsten” Operationen, die keinen direkten Mehraufwand zur Wahrung der Relationalen Invarianten einführen, sind das Einfügen von Sätzen in der Vater-Relation sowie das Löschen von Sätzen in der Sohn-Relation. Die Wahrung der Referentiellen Integrität beim Einfügen und Ändern von Tupeln in der Sohn-Relation führt geringe Zusatzkosten ein, da hierzu nur auf die entsprechende Indexstruktur(en) der Vater-Relation zugegriffen werden muß. Am “teuersten” ist das Löschen von Sätzen in der Vater-Relation, insbesondere bei CASCADE, da das Löschen der abhängigen Tupel die Aktualisierung aller Indexstrukturen der Sohn-Relation verlangt. Bei RESTRICTED kann das Löschen vergleichsweise effizient erfolgen, da bezüglich der Sohn-Relation nur auf den Index zugegriffen werden muß. Tupeländerungen in der Vater-Relation erfordern i.a. einen geringeren Zusatzaufwand für die Einhaltung der Referentiellen Integrität als das Löschen, da in der Regel nicht für alle Fremdschlüs-

	Einfügen	**Löschen**	**Ändern**
Vater	0	**R:** $f * H_{S-FS}$ **S:** $f (2 H_{S-FS} + N_S / N_V)$ **C:** $f (1 + (s + f) H_S)* N_S / N_V$	**R:** $u_{V-FS} * H_{S-FS}$ **S:** $u_{V-FS} (2 H_{S-FS} + N_S / N_V)$ **C:** $u_{V-FS} (2 H_{S-FS} + N_S / N_V)$
Sohn	$f * H_{V-FS}$	0	$u_{S-FS} * H_{V-FS}$

a) Zusätzliche lesende Seitenzugriffe

	Einfügen	**Löschen**	**Ändern**
Vater	0	**R:** 0 **S:** $f (2 + N_S / N_V)$ **C:** $f (1 + s + f)* N_S / N_V$	**R:** 0 **S:** $u_{V-FS} (2 + N_S / N_V)$ **C:** $u_{V-FS} (2 + N_S / N_V)$
Sohn	0	0	0

b) Zusätzliche Schreibzugriffe

	Einfügen	**Löschen**	**Ändern**
Vater	0	**R:** 2 f **S:** $f (8 + N_S / N_V)$ **C:** $f (3 + (1 + 3 (s + f))* N_S / N_V)$	**R:** $2 u_{V-FS}$ **S:** $u_{V-FS} (8 + N_S / N_V)$ **C:** $u_{V-FS} (8 + N_S / N_V)$
Sohn	2 f	0	$2 u_{S-FS}$

c) Zusätzliche Sperranforderungen

Tabelle 5: Direkte Zusatzkosten bei B*-Bäumen

sel der Sohn-Relation eine Überprüfung/Anpassung erforderlich ist. CASCADE verursacht hierbei die gleichen Kosten wie SET NULL/DEFAULT. Wie Tabelle 5b zeigt, sind zusätzliche Schreibzugriffe nur bei Lösch- und Änderungsvorgängen auf der Vater-Relation erforderlich, um für abhängige Tupel der Sohn-Relation Folgeänderungen durchzuführen (außer bei RESTRICTED).

Die Tabellen zeigen, daß die Zusatzkosten vor allem durch die Anzahl der Fremdschlüssel f bestimmt sind; für Änderungsbefehle (UPDATE) ist die durchschnittliche Anzahl geänderter Fremdschlüssel pro Operation maßgebend. Die Kosten steigen meist linear mit der Fremdschlüsselanzahl, bei der Löschregel CASCADE jedoch quadratisch. Daneben steigen die Zusatzkosten für das Löschen und Ändern von Sätzen der Vater-Relation linear mit der mittleren Anzahl von abhängigen Tupeln (außer bei RESTRICTED). Die Relationengrößen spielen dagegen eine untergeordnete Rolle, da sie nur über die Baumhö-

	Löschen eines V-Tupels			Ändern eines V-Tupels		
N_S / N_V	1	10	100	1	10	100
RESTRICTED	3	3	4	0.6	0.6	0.8
SET NULL/DEF.	7	16	108	1.4	3.2	22
CASCADE	10	100	1300	1.4	3.2	22

	Einfügen eines S-Tupels			Ändern eines S-Tupels		
N_S / N_V	1	10	100	1	10	100
	3	3	3	0.6	0.6	0.6

a) Anzahl zusätzlicher Lesezugriffe bei einem Fremdschlüssel (f=1, $u_{V\text{-}FS}$= $u_{S\text{-}FS}$= 0.2)

	Löschen eines V-Tupels			Ändern eines V-Tupels		
N_S / N_V	1	10	100	1	10	100
RESTRICTED	15	15	20	3	3	4
SET NULL/DEF.	35	80	540	7	16	108
CASCADE	110	1100	14500	7	16	108

	Einfügen eines S-Tupels			Ändern eines S-Tupels		
N_S / N_V	1	10	100	1	10	100
	15	15	15	3	3	3

b) Anzahl zusätzlicher Lesezugriffe bei fünf Fremdschlüsseln (f=5, $u_{V\text{-}FS}$= $u_{S\text{-}FS}$= 1)

Annahmen: s=2; H_V = 3; H_S = 3 für N_S / N_V < 100 bzw. H_S = 4 für N_S / N_V =100

Tabelle 6: Zusätzliche Seitenzugriffe bei B*-Bäumen (Beispiel)

hen in geringem Maße eingehen. Diese Beobachtungen werden in Tabelle 6 für einige Beispielwerte illustriert. Dabei wurde von relativ großen Relationen ausgegangen (z.B. $N_V=10^4$).

7.2 Zusatzkosten für kombinierte Zugriffspfadstruktur

Die für B*-Bäume ausführlich hergeleiteten Kostenformeln lassen sich in ähnlicher Weise auch für die kombinierte Zugriffspfadstruktur bestimmen. Dazu wird angenommen, daß für jeden der f Fremdschlüssel ein kombinierter Zugriffspfad für Vater- und Sohn-Relation existiert. Der Sperraufwand bleibt dabei gegenüber der Verwendung einfacher B*-Bäume unverändert, da die Synchronisation auf logischen Zugriffspfaden erfolgt, auch wenn diese physisch zusammengelegt werden (6.1). Die folgenden Überlegungen beschränken sich daher auf die Bestimmung der Seitenzugriffe. Hierbei ergeben sich für die indirekten Zusatzkosten im wesentlichen die gleichen Kostenformeln wie in 7.1. Die bei den Änderungsoperationen durchzuführenden Anpassungen der Indexstrukturen erfolgen lediglich zum Teil in den kombinierten anstatt auf dedizierten Zugriffspfaden.

	Einfügen	**Löschen**	**Ändern**
Vater	0	**R:** 0 **S:** $f (H_{FS} + N_S / N_V)$ **C:** $f (1 + s^* H_S)^* N_S / N_V$	**R:** 0 **S:** $u_{V\text{-}FS} (H_{FS} + N_S / N_V)$ **C:** $u_{V\text{-}FS} * N_S / N_V$
Sohn	0	0	0

a) Zusätzliche Lesezugriffe

	Einfügen	**Löschen**	**Ändern**
Vater	0	**R:** 0 **S:** $f (1 + N_S / N_V)$ **C:** $f (1 + s)^* N_S / N_V$	**R:** 0 **S:** $u_{V\text{-}FS} (1 + N_S / N_V)$ **C:** $u_{V\text{-}FS} * N_S / N_V$
Sohn	0	0	0

b) Zusätzliche Schreibzugriffe

Tabelle 7: Zusätzliche Seitenzugriffe bei kombinierten Zugriffspfaden (IO_{direkt})

Allerdings ergeben sich durch die Zusammenlegung der Indexstrukturen erhebliche Einsparungen bezüglich der direkten Zusatzkosten zur Wahrung der Referentiellen Integrität. Tabelle 7 zeigt zusammenfassend, wieviele zusätzliche logische Seitenzugriffe für jede der sechs Änderungsoperationen benötigt werden (der Parameter H_{FS} bezeichnet dabei die mittlere Höhe der f kombinierten Zugriffspfade). Ein Vergleich mit Tabelle 5a verdeutlicht die Einsparungen.

Es fällt auf, daß im günstigsten Fall (Lösch- bzw. Änderungsregel RESTRICTED) weder für Änderungen in der Sohn- noch für Änderungen in der Vater-Relation zusätzliche Seitenzugriffe zur Wahrung der Relationalen Invarianten anfallen! Das Einfügen und Ändern von Sohn-Tupeln läuft jetzt ohne Zusatzkosten ab, da die Überprüfung, ob zu einem Fremdschlüssel ein zugehöriger Wert in der Vater-Relation existiert, zusammen mit der ohnehin notwendigen Anpassung der kombinierten Zugriffspfadstruktur erfolgen kann. Die zugehörigen Indexeinträge der Vater-Relation befinden sich nämlich in derselben Blattseite, in die der neue Indexeintrag für die Sohn-Relation einzufügen ist. Somit kann also nicht nur die Eindeutigkeit von Schlüsselkandidaten, sondern auch die Referentielle Integrität während der Indexaktualisierung überwacht werden. In ähnlicher Weise lassen sich beim Löschen bzw. Ändern von Tupeln der Vater-Relation für RESTRICTED Zusatzkosten umgehen. Auch in diesen Fällen ist eine Aktualisierung der kombinierten Indexstruktur erforderlich, wobei in der entsprechenden Blattseite direkt überprüft werden kann, ob abhängige Tupel existieren. Auch für die restlichen Lösch- bzw. Änderungsregeln ergeben sich Einsparungen in der Anzahl von Seitenzugriffen, da für die Sohn-Relation die Indexstrukturen auf Fremdschlüsseln zum Teil zusammen mit der Indexaktualisierung für die Vater-Relation aktualisiert werden können. Für SET NULL/DEFAULT reduziert sich so pro Fremdschlüssel (DELETE) bzw. pro geändertem Fremdschlüssel (UPDATE) die Anzahl von zusätzlichen Baumdurchläufen von zwei auf eins (der Zugriff zur TID-Liste der abhängigen Tupel in der kombinierten Zugriffspfadstruktur sowie das Löschen erfolgt zusammen mit der Indexaktualisierung für die Vater-Relation; das Einfügen der TID-Liste mit geändertem Fremdschlüsselwert in eine neue Blattseite erfordert einen eigenen Baumdurchlauf). Noch weitergehende Einsparungen ergeben sich für CASCADE. Beim Löschen konnte die quadratische

	Löschen eines V-Tupels			Ändern eines V-Tupels		
N_S / N_V	1	10	100	1	10	100
RESTRICTED	0	0	0	0	0	0
SET NULL/DEF.	4	13	104	0.8	2.6	21
CASCADE	7	70	900	0.2	2	20

a) Anzahl zusätzlicher Lesezugriffe bei einem Fremdschlüssel (f=1, $u_{V\text{-}FS}$= $u_{S\text{-}FS}$= 0.2)

	Löschen eines V-Tupels			Ändern eines V-Tupels		
N_S / N_V	1	10	100	1	10	100
RESTRICTED	0	0	0	0	0	0
SET NULL/DEF.	20	65	520	4	13	104
CASCADE	35	350	4500	1	10	100

b) Anzahl zusätzlicher Lesezugriffe bei fünf Fremdschlüsseln (f=5, $u_{V\text{-}FS}$= $u_{S\text{-}FS}$= 1)

Annahmen: s=2; H_S = 3 für N_S / N_V < 100 bzw. H_S = 4 für N_S / N_V =100; H_{FS}=H_S

Tabelle 8: Zusätzliche Seitenzugriffe bei kombinierten Zugriffspfaden (Beispiel)

Abhängigkeit zur Parameter f beseitigt werden, da die Anpassung der Fremdschlüssel-Indexstrukturen keinerlei zusätzlichen Aufwand mehr verursacht. Das gilt auch beim Ändern mit CASCADE, so daß dabei in der Sohn-Relation nur noch für die Tupeländerungen selbst zusätzliche Seitenzugriffe anfallen. Die Indexanpassungen in der kombinierten Zugriffspfadstruktur erfolgen zusammen mit denen der Vater-Relation, da die Schlüsselwerte für Vater- und Sohn-Tupel auf denselben Wert geändert werden, so daß das Löschen und Einfügen der Indexeinträge jeweils dieselben Indexseiten betreffen.

Tabelle 8 zeigt die Zusatzkosten für die Beispielwerte aus Tabelle 6. Man erkennt, daß die größten Verbesserungen bei CASCADE zu verzeichnen sind, der Gesamtaufwand dabei jedoch noch immer vergleichsweise hoch liegt.

8. Zusammenfassung

Die Überprüfung der Relationalen Invarianten und von Attributen, die mit der UNIQUE-Option spezifiziert sind, verlangt bei hinreichend großen Relationen eine spezielle Zugriffspfad-Unterstützung, da die sonst anfallenden sequentiellen Suchvorgänge für das Leistungsverhalten des Systems nicht tolerierbar sind. Für die praktische Implementierung bedeutet deshalb diese zentrale Integritätsforderung das Anlegen einer Indexstruktur für jeden Schlüsselkandidaten und für jeden Fremdschlüssel.

Als beste Zugriffsstruktur erweist sich hier der B*-Baum. Für die Verweisstrukturen in den Blattseiten bieten sich verschiedene Erweiterungen an (UNIQUE und NONUNIQUE). Außerdem ist es durch die kombinierte und verallgemeinerte Zugriffspfadstruktur möglich, mehrere separate Indexstrukturen mittels eines B*-Baumes darzustellen.

Die Synchronisation in diesen Zugriffspfaden ist von besonderer Wichtigkeit für das Systemverhalten, da viele DB-Operationen diese Pfade benutzen müssen. Für den Baumdurchlauf wurde ein einfaches Protokoll der Latch-Kopplung skizziert, das sich in einer konkreten Implementierung weiter verfeinern und optimieren läßt. Für das Spektrum der Indexoperationen wurde ein Sperrprotokoll entwickelt, das Konflikte und Blockierungen von parallelen Transaktionen minimiert und trotzdem Konsistenzebene 3 gewährleistet. Wesentlich dabei war das Konzept des “Sperren des nächsten Schlüssels”, womit im Prinzip Bereichssperren simuliert wurden. Die Optimierung dieses Konzeptes erlaubte die parallele Modifikation in einer Indexstruktur, wobei selbst Einfügungen und Löschungen in einer TID-Liste eines Schlüsselwertes durch verschiedene Transaktionen ermöglicht wurden.

Anhand einfacher Kostenmodelle konnte gezeigt werden, daß der zur Überprüfung der Relationalen Invarianten erforderliche Zusatzaufwand durch die Verwendung von B*-Bäumen i.a. gering gehalten werden kann. Am teuersten sind Lösch- und Änderungsoperationen für Tupel, zu denen eine große Anzahl “abhängiger” Sätze existiert, insbesondere bei Verwendung der Optionen SET NULL, SET DEFAULT und CASCADE. Der Zusatzaufwand steigt dabei mit der mittleren Anzahl abhängiger Tupel sowie der Fremdschlüsselanzahl. Mit der kombinierten Zugriffspfadstruktur läßt sich der Zusatzaufwand zur Wah-

rung der Relationalen Invarianten merklich reduzieren. Im günstigsten Fall (RESTRICTED) entstehen keinerlei zusätzlichen Seitenzugriffe.

9. Literaturverzeichnis

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.

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.

ESS91 Eiter, T., Schrefl, M., Stumptner, M.: Sperrverfahren für B-Bäume im Vergeich, in: Informatik-Spektrum, Bd. 14, H. 4, Aug. 1991, S. 183-220.

Fa79 Fagin, R., et al.: Extendible Hashing - a Fast Access Method for Dynamic Files, in: ACM TODS, Vol. 4, No. 3, 1979, pp. 315-344.

GLPT76 Gray, J.N., et al.: Granularity of Locks and Degrees of Donsistency 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.

Gr81 Gray, J.N., et al.: The Recovery Manager of the System R Database Manager, in: ACM Computing Surveys, Vol. 13, No. 2, 1981, pp. 223-242.

Hä78 Härder, T.: Implementing a Generalized Access Path Structure for a Relational Data Base System, in: ACM Transactions on Database Systems, Vol. 3, No. 3, 1978, pp. 285-298.

Hä78a Härder, T.: Implementierung von Datenbanksystemen, Hanser-Verlag, 1978.

Hä92 Härder, T.: Mehrdimensionale Zugriffspfade in Relationalen Datenbanksystemen, Interner Bericht, Universität Kaiserslautern, Jan. 1992.

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

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.

LY81 Lehman, P.L., Yao, S.B.: Efficient Locking for Concurrent Operations on B-trees, in: ACM TODS, Vol. 6, No. 4, Dec. 1981, pp. 650-670.

ML89 Mohan, C., Levine, F.: ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging, IBM Research Report, RJ 5037, San Jose, Calif., August 1989.

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.

SG88 Shasha, D., Goodman, N.: Concurrent Search Structure Algorithms, in: ACM TODS, Vol. 13, No. 1, March 1988, pp. 53-90.

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.

Va87 Valduriez, P.: Join Indices, in: ACM TODS, Vol. 12, No. 2, 1987, pp. 218-246.