Institut für Informatik
der Technischen Universität München

Datenbanksysteme für Rechenanlagen mit intelligenten Subsystemen: Architektur, Algorithmen, Optimierung

Werner Kießling

Vollständiger Abdruck der von der Fakultät für Mathematik und Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors rer. nat.

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. K.-H. Helwig
1. Prüfer:	Prof. ~~Dr.~~ R. Bayer, Ph. D.
2. Prüfer:	Prof. Dr. U. Güntzer

Die Dissertation wurde am ~~8.~~ 7. 6. 1983 bei der Technischen Universität München eingereicht und durch die Fakultät für Mathematik und Informatik am ~~29.~~ 25. 7. 1983 angenommen.

Vorwort

Ich danke Herrn Prof. Dr. R. Bayer für das Thema dieser Arbeit und für seine Anregungen und Verbesserungsvorschläge bei der Abfassung. Mein Dank gilt gleichfalls Herrn Prof. Dr. U. Güntzer für seine sorgfältigen Anmerkungen beim Durchlesen des Manuskripts.
Für die stets freundliche Atmosphäre möchte ich mich bei meinen Kolleginnen und Kollegen bedanken, insbesondere bei Elli Prochaska, die mich beim Tippen des Manuskripts sehr unterstützte.
Schließlich danke ich meiner Frau Elfriede und meinen Kindern Michael und Julia für die Geduld, die sie gegen Abschluß der Arbeit mit mir hatten.

Kurzfassung.

Die ständig steigenden Leistungsanforderungen von Datenbank-Anwendungen können mit den heutigen Datenbanksystemen nur noch unzureichend bewältigt werden. Trotz intensiver Forschungsaktivitäten in jüngster Zeit, insbesondere auf dem Sektor der Datenbank-Maschinen, ist bis jetzt noch kein richtungsweisender Durchbruch auf dem Weg zu allgemeinen, kommerziell einsetzbaren Hochleistungssystemen erkennbar.
Durch den technologischen Fortschritt im Hardware-Bereich werden nun Rechenanlagen mit intelligenten Subsystemen verfügbar und wirtschaftlich. Sie sind der Anstoß, das Design einer leistungsfähigen DB-Architektur neu zu überdenken.
In der vorliegenden Arbeit wird auf der Basis moderner Rechner-Architektur der Entwurf eines funktional vollständigen Datenbanksystems vorgestellt, welches den heutigen und zukünftigen Leistungsanforderungen besser gerecht wird.
Die herausragenden Merkmale dieses Entwurfs sind:

- der geschickte Einsatz von intelligenten Subsystemen für die Verwaltung der physischen Datenbank,
- das Prinzip der Mengenverarbeitung bei der Konstruktion der internen Schnittstellen des Datenbanksystems,
- eine möglichst frühzeitige Datenfilterung mit dem Ziel, die Datentransporte zwischen Peripheriespeicher und Arbeitsspeicher stark zu reduzieren.

Diese Architekturmerkmale und der DBS-Entwurf führten zur Entwicklung einer Reihe von völlig neuen, sehr effizienten Algorithmen zur Queryauswertung, die auch für konventionelle Datenbanksysteme von Interesse sind. Die Kernidee dabei ist die Berechnung von dynamischen Filtern, welche auf die Leistungsfähigkeit der intelligenten Subsysteme abgestimmt sind. Schließlich wird ein analytisches Modell entwickelt, welches erste Einsichten in die veränderten Eigenschaften dieser Datenbank-Architektur im Hinblick auf Verwendung/Nutzen von Indexen gibt.

Inhaltsverzeichnis.

1. Leistungsmerkmale existierender DB-Systeme.

1.1. Zielsetzung und Grundbegriffe.

Eine der wichtigsten kommerziellen Anwendungen für Computersysteme ist die Verwaltung von großen Datenbanken (DBs). Dabei werden ständig steigende Anforderungen an das Leistungsvermögen , die Verfügbarkeit und die Sicherheit solcher DB-Systeme gestellt. Heutige DB-Systeme auf konventionellen Rechenanlagen können diesen erhöhten Ansprüchen nur noch schwerlich genügen. Diese Sachlage führte in den vergangenen zehn Jahren zu einer intensiven Forschungsaktivität auf dem Gebiet der Datenbankmaschinen. Dabei versuchte man zumeist, eine Leistungssteigerung durch den Einsatz von spezialisierter Hardware zu erzielen, von der man vermutete, daß diese für die funktionalen Erfordernisse von DB-Systemen besser geeignet seien als die konventionelle, ursprünglich für wissenschaftlich/technische Aufgabenlösungen konzipierte von Neumann-Architektur. Jedoch haben diese Bemühungen bis heute keinen entscheidenden Durchbruch auf dem Weg zu wirtschaftlichen Hochleistungssystemen gebracht. Die Gründe hierfür sind einmal in den sehr hohen Kosten für die Spezialhardware zu suchen, aber auch in einer nicht optimalen Konstruktion der internen Schnittstellen des DB-Systems.
Neue Perspektiven eröffnen sich indes durch zukunftsweisende Tendenzen auf dem Hardware-Sektor. Die jüngsten Fortschritte in der Entwicklung von preisgünstiger, leistungsfähiger konventioneller Hardware geben Anlaß dazu, die Probleme beim Design einer leistungsfähigen DB-Architektur neu zu überdenken.

Forschungsziel dieser Arbeit:
Es soll untersucht werden, wie DB-Systeme aufgebaut sein müssen, um mit modernen Rechenanlagen optimal einsetzbar zu sein. Dabei sollen die neuen Eigenschaften moderner Rechenanlagen gezielt ausgenutzt und die geänderten Kosten- sowie Leistungsfaktoren berücksichtigt werden. Zugleich sollen die neuesten Erkenntnisse auf dem Gebiet der Ablaufsteuerung/Verfügbarkeit für

DB-Systeme in wirksamer Weise integriert werden.
Das angestrebte Entwicklungsziel ist eine Hochleistungsarchitektur für transaktionsorientierte DB-Systeme, welche den folgenden Randbedingungen *benügt:*

* Sie soll ein vollständiges Datenmodell (wie z.B. das Relationenmodell) unterstützen.
* Sie soll auch sehr große DBs (z.B. > 10 GBytes) effizient verwalten können.
* Sie soll auch für sehr komplexe (interaktive) DB-Anfragen ein hohes Leistungsvermögen aufweisen.
* Sie soll in naher Zukunft für kommerzielle Anwendungen realisierbar sein.

Anmerkung: Für spezielle, stark eingeschränkte DB-Anwendungen existieren heute schon Speziallösungen mit sehr hoher Leistungsfähigkeit. Als Beispiele hierfür sind IMS Fastpath[1] sowie ACP/PARS[2] zu nennen.
Der Inhalt dieser Arbeit ist die Grundlagenentwicklung für eine Hochleistungsarchitektur, welche den gestellten Anforderungen genügt. Dabei sollen Architekturprinzipien, Schnittstellenbeschreibungen, neue Auswertungsalgorithmen und Optimierungskonzepte soweit entwickelt werden, daß auf dieser Grundlage die Erstellung eines Prototyps einfach möglich sein sollte; der Einbau einzelner Optimierungen lokaler Natur bleibt späteren Weiterentwicklungen vorbehalten.

Grundbegriffe bei DB-Systemen.

Als Schlüsselkonzept für die korrekte Verwaltung einer DB, welche von vielen Benutzern gemeinsam und parallel bearbeitet wird, hat sich der Begriff der Transaktion erwiesen. Eine Transaktion T besteht aus einer endlichen Folge von Aktionen (z.B. Lesen, Schreiben) auf den Objekten in einer DB und hat folgende fundamentale Eigenschaften:

* Erhaltung der Datenintegrität:
  Wenn T allein auf der DB arbeitet, so transformiert T einen korrekten

[1] IMS Fastpath schränkt die Anzahl der Hierarchiestufen auf 1-2 ein ([IMSPF]).

[2] ACP/PARS erzielt eine Transaktionsrate von etwa 180 Trans/sec für sehr kurze Transaktionen ([SIWI77]).

DB-Zustand in einen neuen korrekten Zustand.

* Atomares Verhalten:
  Entweder alle Aktionen von T werden erfolgreich durchgeführt (Commit), oder - im Fehlerfalle[3] - alle bisherigen Änderungen des DB-Zustands durch T werden rückgängig gemacht (Back up).
* Dauerhaftigkeit:
  Die Änderungen von T im Falle eines erfolgreichen Commits sind immun gegen nachfolgende Fehler.

Eine ausführliche Diskussion des Transaktionskonzepts ist z.B. in [GRAY77] zu finden.

Die DB-Benutzer bearbeiten die DB mittels Transaktionen. Das Datenbankverwaltungssystem (DBMS) ist verantwortlich für die Realisierung der DB-Retrievalanfragen sowie DB-Änderungsaufträge unter Einhaltung der genannten Transaktionseigenschaften. Dabei hat das DBMS - neben vielen weiteren Aufgaben wie Zugriffsschutz, Optimierung von Anfragen, Realisierung externer Benutzersichten, etc. - zwei wesentliche Mechanismen bereitzustellen:

(a) Recovery-Mechanismen, um die Eigenschaften des atomaren Verhaltens und der Dauerhaftigkeit zu garantieren.
(b) Synchronisationsmechanismen (Concurrency Control), um Inkonsistenzen bei der Parallelausführung von Transaktionen vermeiden zu können.

Für eine detaillierte Beschreibung verweisen wir wieder auf [GRAY77], desgleichen setzen wir eine Vertrautheit mit den drei wichtigsten Datenmodellen für DBs - hierarchisch, netzwerkartig (Codasyl), relational - voraus (siehe z.B. [ULLM80]).

1.2. Konventionelle DB-Systeme.

Eine ausgezeichnete Übersicht über den prinzipiellen Aufbau von konventionellen DB-Systemen und von DB-Systemen, welche eine sogenannte DB-Maschine enthalten, ist in [DATE82] zu finden. Beginnen wir die Diskussion mit den kommerziell im Augenblick fast ausschließlich eingesetzten

[3] z.B., wenn der Benutzer die Transaktion abbricht.

sogenannten konventionellen DB-Systemen.

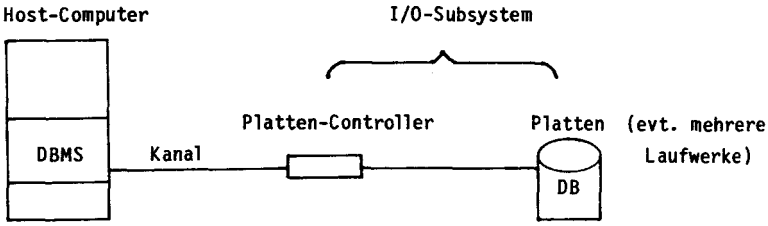


Abb. 1.1: Struktur eines konventionellen DB-Systems.

Die charakteristischen Merkmale eines solchen DB-Systems sind dabei wie folgt:

* Der Host-Computer ist ein "General-Purpose"-Rechner.
* Das DBMS liegt vollständig im Host.
* Die Schnittstelle zwischen DBMS und I/O-Subsystem ist relativ niedrig, typischerweise auf der Ebene von 'Lies-Block', 'Schreibe-Block'.
* Die physische DB ist auf den Platten abgespeichert, bei denen es sich i.a. um Magnetplattenstapel mit einem beweglichen Lese/Schreibkamm sowie maximal einem aktiven Lese/Schreibkopf handelt (konventionelle Plattenlaufwerke).

Bekanntermaßen besteht eine große semantische Lücke zwischen der Funktionalität solcher konventioneller I/O-Subsysteme und den Erfordernissen von DB-Aufträgen[4] , welche komplexe Suchanfragen beinhalten. Diese Diskrepanz hatte notwendigerweise Auswirkungen auf die Datenzugriffskomponenten des DBMS in der Form einer extensiven Verwendung von Zeigern, z.B. Indexbäume, Hashing, invertierte Listen, Adreßketten, usw.. Die Folge davon ist ein navigierendes Zugriffsverhalten (bei Codasyl-DBs und hierarchischen DBs bereits an der Benutzerschnittstelle, bei relationalen DBs durch das DBMS), welches ungeachtet des betreffenden verwendeten Mechanismus i.a. mehrere Plattenzugriffe zur Erledigung eines Satz/Tupel-Zugriffs bewirkt. Da der Aufbau von Codasyl- sowie hierarchischen DB-Systemen gut bekannt ist,

[4] Man vergleiche etwa DB-Anfragen wie "Gib mir alle Sätze mit Schlüssel = 'ABC'" mit der typischen Lies-Block Schnittstelle konventioneller I/O-Subsysteme.

wollen wir darauf nicht näher eingehen und nur den DBMS-Aufbau des relationalen SystemR ([ASTR76]) kurz skizzieren.

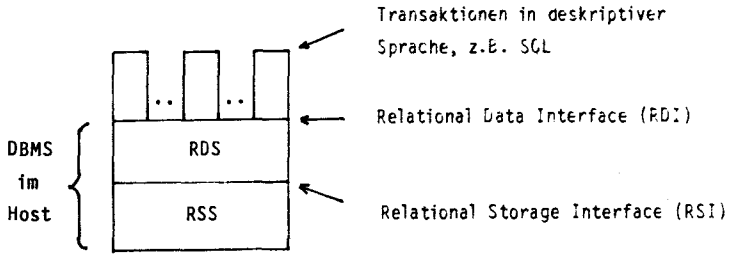


Abb. 1.2: DBMS-Struktur von SystemR.

Die für uns hier wesentlichen globalen Eigenschaften von SystemR sind die folgenden:

* RDI bildet eine Schnittstelle für deskriptive DB-Anfragen, das RDS (Relational Data System) bildet diese auf RSI-Operatoren ab.
* RSI ist eine 1-Tupel-Schnittstelle, das RSS (Relational Storage System) realisiert den Zugriff auf ein gewünschtes Tupel unter Verwendung von Zeigern (Indexes, Links, Segmentscans).

Eine Strukturierung des DBMS in analoger Weise , Datenzugriffssystem hat eine 1-Tupel-Schnittstelle, ist bei einem weiteren bekannten Vertreter dieser Klasse, nämlich INGRES ([STON76]) zu finden. Auf die Leistungsfähigkeit existierender konventioneller DB-Systeme wird in Kap. 1.4) eingegangen werden.

1.3. Backend-Systeme.

Um die gegenwärtigen und zukünftigen Leistungsanforderungen an 'General-Purpose'-DB-Systeme besser erfüllen zu können, sind in den letzten Jahren verschiedene Vorschläge für leistungsfähigere Architekturen gemacht worden. Für die Kommunikation Benutzer-Host ist der Begriff des Front-

end-Prozessors eingeführt worden mit der Absicht, Antwortzeiten und Durchsatz zu verbessern, indem man terminalorientierte Operationen aus dem Host auslagert und einem speziellen Terminal-Controller überträgt. Bei DB-Anwendungen jedoch, insbesondere bei interaktiven Querysystemen mit komplexen Suchanfragen, liegt der Systemengpaß oft auch in den langsamen Zugriffen zu den DB-Platten[5] . Eine weitere Möglichkeit, die konventionellen Rechenanlagen - welche ursprünglich für rechenintensive numerische Anwendungen konzipiert wurden - für DB-Anwendungen leistungsfähiger zu machen, ist somit offenbar die Einrichtung von DB-Pufferbereichen (DB-Cache) im Host-Arbeitsspeicher, um im Falle einer vorhandenen Lokalität bei Datenzugriffen Plattenzugriffe einsparen zu können. Als Sekundäreffekt führt die niedrige I/O-Schnittstelle zu einer hohen Prozessorbelastung im Host, sodaß man die CPU-Leistung des Hosts durch die Einführung von Multiprozessorsystemen zu erhöhen versuchte.
All diese Maßnahmen sind zwar Schritte in die richtige Richtung, ändern jedoch nichts an der für DB-Anwendungen zu niedrigen Schnittstelle zum I/O-Subsystem. Um diese angesprochene semantische Lücke und die damit verbundenen potentiellen Systemengpässe prinzipiell zu verringern, sind als Gegenstück zur Frontend-Lösung verschiedene sogenannte Backend-Systeme vorgeschlagen und zum Teil auch implementiert worden.
Anmerkung: Backend-Systeme für DB-Anwendungen werden häufig auch als Datenbankmaschinen bezeichnet.

Die grundlegende Idee dabei ist die Auslagerung des ganzen DBMS oder eines Teils davon auf einen intelligenten gewidmeten Rechner, den Backend, welcher das Durchsuchen der Platten effizienter ausführen kann. Eine allen Backend-Prozessoren gemeinsame Eigenschaft ist dabei, ein gewünschtes Datenobjekt zu lokalisieren (eine Funktion, welche vorher von der CPU durchzuführen war) und dann nur noch das gewünschte Datenobjekt an den Host-Computer zu übertragen. Dieses Vorgehen zur Überbrückung der semantischen Lücke zieht nun einige prinzipielle Auswirkungen nach sich.
Als potentelle Vorteile sind zu nennen:

* Parallele Verarbeitung im Host und Backend ist möglich.

5) Das Verhältnis Hauptspeicherzugriffszeit : Plattenzugriffszeit beträgt etwa $1 : 10^5$.

* Effizienteres Arbeiten durch einen spezialisierten Backend.

Um Nachteile zu vermeiden, ist folgendes zu beachten:

* Gefahr eines unbalanzierten Gesamtsystems, da Systeme mit gewidmeten Prozessoren weniger flexibel in der Fähigkeit sind, die Arbeitslast zu verteilen, als es bei einem zentralen Host-Computer der Fall ist.
* "Offloading Theorem":
  Um die Auslagerung von DBMS-Funktionen auf einen Backend kosteneffektiv zu machen, sollte der ausgelagerte Arbeitsaufwand signifikant größer sein als der erforderliche Kommunikationsmehraufwand.

Die nachfolgend vorgestellten Backend-Systeme unterscheiden sich in der Backend-Hardware sowie in der Funktionalität des Backends (Anteil des ausgelagerten DBMS, Schnittstelle zum Backend).

Spezielle Backend-Systeme.

(1) Gewidmeter konventioneller Backend.

Diese Architektur zeichnet sich dadurch aus, daß das gesamte DBMS in den Backend ausgelagert ist, welcher selbst im wesentlichen ein konventioneller Rechner ist (mit für DB-Anwendungen besser geeignetem Betriebssystem). Der Host-Computer übernimmt hierbei nur noch Frontend-Funktionen. Die Beschreibung von ersten implementierten Prototypen dieser Art ist z.B. in [MARY80] zu finden. Ein gemeinsamer Nachteil all dieser Backend-Systeme, wie z.B. XDMS von Bell, war die für heutige Codasyl-Systeme typische niedrige get_next-Schnittstelle zum Backend. Darin dürfte auch der Grund zu suchen sein, daß keines dieser frühen Backend-Systeme zur kommerziellen Anwendung gelangte. Der gewidmete konventionelle Backend IDM500 mit einer relationalen Schnittstelle (ganze Query in deskriptiver Sprache) wurde 1982 von Britton Lee auf den Markt gebracht. Diese DB-Maschine IDM500 verwendet in ihrer Standardversion Zeiger zum Lokalisieren von Daten (konkret

B-Baume), optional kann assoziative Plattenhardware[6] dazugeschaltet werden. Mit dieser Ausbaustufe soll die IDM500 eine Transaktionsrate von 20-30 Trans/sec bewältigen (siehe [DATE82]).

(2) Assoziative Platten als Backend.

Bei dieser DB-Architektur wird nicht das gesamte DBMS aus dem Host ausgelagert, sondern zumeist nur die Funktionen, welche für das assoziative Suchen verantwortlich sind. Die Grundidee ist dabei wie folgt:

* Zwischen Host und DB-Platten wird Hardware-Suchlogik eingebaut, um Daten filtern zu können während sie von der Platte[7] gelesen werden.
* Die Verwendung von Zeigern (Indexen) soll überflüssig werden (oder zumindest reduziert werden). Dazu ist es notwendig, daß das physisch-sequentielle Auslesen der DB von den Platten beschleunigt wird.

Dementsprechend lassen sich assoziative Platten klassifizieren hinsichtlich der Auslesegeschwindigkeit und der entsprechenden Funktionalität der Suchlogik:

(a) Processor-Per-Track (PPT)

Diese Klasse zeichnet sich dadurch aus, daß jede Plattenspur über ihren eigenen gewidmeten (Mikro-)Prozessor verfügt. All diese Prozessoren können diesselbe Suchoperation parallel ausführen, sodaß ein gesamter Plattenstapel in einer Umdrehung durchsucht werden kann. Damit kann auf die Verwendung von Indexen gänzlich verzichtet werden.

Als typischer Vertreter dieser Klasse ist das DB-System RAP (Relational Associative Processor, [OZKA77]) von der Universität Toronto zu nennen. Für die Implementierung der assoziativen PPT-Platten wurde eine entsprechend ausgebaute Variante von Festkopfplatten verwendet. Die Suchlogik umfaßt die komplette Relationenalgebra, d.h. auch komplexe, nicht lineare Operationen wie Join sind hardwaremäßig implementiert.

6) welche einen Spezialprozessor - den sogenannten Accelerator- mit 10 MIPS enthalten soll.

7) Platte ist dabei als Sammelbegriff für permanente Speichermedien wie Magnetplatten,Blasenspeicher,CCDs,... zu verstehen.

(b) Processor-Per-Surface (PPS)
Die PPS-Klasse kann als ein ausgebautes Plattenlaufwerk angesehen werden,welches einen beweglichen Lese/Schreibarm besitzt, welcher aber für jede Plattenoberfläche einen aktiven Lese/Schreibkopf besitzt, sodaß ein ganzer Zylinder in einer Umdrehung ausgelesen werden kann. Ein bekannter Vertreter hierfür ist die Datenbankmaschine DBC ([BANE78]) von der Universität Columbo/Ohio, welche - neben weiterer Spezialhardware - ein solches PPS-Gerät enthält. Um die Anzahl der auszulesenden Zylinder zu reduzieren, verwendet DBC Indexe auf Zylinderbasis. Ebenso wie bei RAP umfaßt die Schnittstelle zum DBC die komplette Relationenalgebra.

(c) Processor-Per-Disk (PPD)
PPD-Geräte verwenden konventionelle Platten mit einem Prozessor zwischen Host-Computer und Platten, welcher nicht relevante Daten eliminiert. Als Beispiel hierfür ist die IDM500 zu nennen, welche intern den Spezialprozessor 'Accelerator' optional als Filter vor alle Plattenlaufwerke schalten kann. Vergleichend wollen wir festhalten, daß physisch-sequentielles Lesen langsamer ist als bei PPS und PPT; ebenso muß der einzige Filterprozessor über eine enorm hohe MIPS-Rate verfügen,um mit allen Platten Schritt halten zu können.

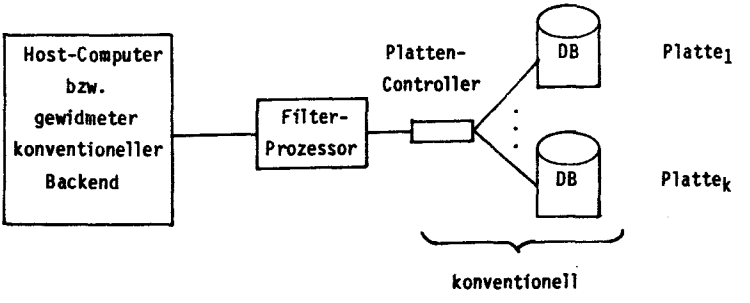


Abb.1.3: Struktur eines PPD-Backends.

(d) Multi-Processor-Cache (MPC)
Diese Klasse unterscheidet sich beträchtlich in puncto Zielsetzung. Bei den

PPT-, PPS-, und PPD-Architekturen handelt es sich im wesentlichen um SIMD-Architekturen (SIMD = Single Instruction Multiple Data). MPC-Systeme verfolgen das Ziel einer MIMD-Architektur (MIMD = Multiple Instruction Multiple Data), um die Parallelität bei der Queryauswertung steigern zu können.

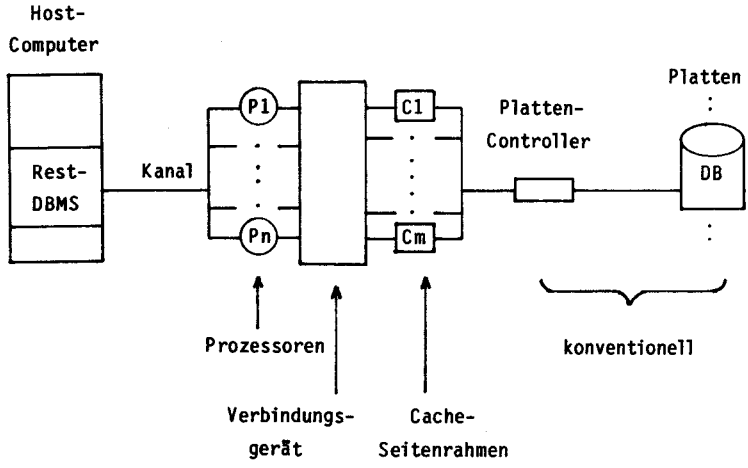


Abb.1.4: Struktur eines DB-Systems mit MPC-Backend.

Die Eigenschaften dieser Architektur wollen wir kurz anhand des Systems DIRECT ([DEWI79]) von der Universität Wisconsin erläutern. Jeder der Prozessoren Pi soll Zugriff auf jede Cache-Seite Cj haben. Deshalb muß das Verbindungsgerät eine Art Kreuzschienenschalter (crossbar switch) sein. Außerdem soll jedes Cj von mehreren Prozessoren gleichzeitig gelesen werden können. Diese Anforderungen bedingen jedoch die Konstruktion von teurer Spezialhardware, will man Engpässe beim Cachezugriff vermeiden. In DIRECT wurde aus Kostengründen anstelle eines allgemeinen Crossbar Switch ein spezieller Multiport-Speicher (für bis zu 8 Prozessoren) gebaut; die Cache-Seiten senden dabei ständig ihren Inhalt an die angeschlossenen Prozessoren. DIRECT verwendet keine Indexe, sondern liest Relationen

sequentiell von den Platten (wenn benötigte Teile nicht im Cache sind). Ein weiteres schwieriges Problem bei MPC-Architekturen ist das Problem der Synchronisation der einzelnen Prozessoren (Botschaften-Overhead!). Schließlich ist noch zu erwähnen, daß DIRECT über eine hohe Schnittstelle zum MPC-Backend verfügt, die sogenannten Querypakete, jedoch über die Standard-Blockschnittstelle zu den DB-Platten.

1.4. Zusammenfassung des heutigen Stands der Technologie für DB-Systeme.

In diesem Abschnitt soll nur in komprimierter Form auf diejenigen Architekturaspekte eingegangen werden, die für die weiteren Überlegungen von entscheidender Bedeutung sind[8] .

Konventionelle DB-Systeme:

Diese DB-Systeme simulieren den assoziativen Datenzugriff durch einen adressierten Zugriff und legen dazu aus Effizienzgründen umfangreiche und komplizierte Hilfsstrukturen (z.B. Indexe) an. Das läuft darauf hinaus, daß, trotz der niedrigen I/O-Schnittstelle, diese Systeme gelegentlich zu einem CPU-Engpaß tendieren. Daß die angesprochene semantische Lücke durch die Verwendung von Zeigern nur notdürftig überbrückt wird, sieht man z.B. auch daran, daß selbst hochgezüchtete Codasyl- oder hierarchische DB-Systeme (wie IMS) über relativ bescheidene Transaktionsraten nicht hinauskommen[9] . Dies ist zum Teil auch dadurch bedingt, daß diese benötigten Hilfsstrukturen eine aufwendige Steuerung, Optimierung und dynamische Wartung erfordern. Darüber hinaus müssen die existierenden DB-Systeme mit vermutlich viel zu kleinem Arbeitsspeicherbereichen auskommen, was zu einer komplizierten Pufferverwaltung und zu intensivem Datenaustausch zwischen DB-Pufferbereich und DB-Platten (bei zu kleinen Transporteinheiten) führt. Ein weiterer typischer Engpaß liegt somit beim I/O-Subsystem, da sehr viele,

8) Welche Implementierungsprobleme bei Prototypen wie z.B. INGRES auftreten können, kann man in [STON80] nachlesen.

9) IMS auf einem großen IBM-Rechner der 3033-Klasse schafft etwa 10-15 Trans/sec bei einer Transaktionspfadlänge von 10^5 Instruktionen und 7-10 I/Os pro Transaktion ([KING80]).

einzeln angestoßene Transporte einzelner Seiten auftreten. Dies führt, neben der erwähnten komplizierten DBMS-Steuerung, auch zu einer sehr schlechten Ausnutzung der eigentlichen Transportleistung heutiger Peripheriespeicher[10] . Als Sekundärphänomen, welches den I/O-Engpaß oft überdeckt, ist meist auch eine hohe Prozessorbelastung der Host-CPU(s) zu beobachten. Dies ist eine Folge der für DB-Anwendungen ungeeigneten Betriebssystemschnittstelle heutiger Rechner, da I/O-Unterbrechungsbehandlungen sehr teuer sind. Ein weiterer Ineffizienzfaktor ist in den verwendeten Synchronisationsverfahren zu suchen. Heutige Systeme verwenden unnötig restriktive Sychronisationsverfahren, welche als veraltet angesehen werden müssen[11] . Dasgleiche gilt für Verfahren zur Sicherung (Logging) und Wiederanlauf (Recovery, Restart). Die heute verwendeten Verfahren für Sicherung und Wiederanlauf wirken sich störend auf die Transaktionsverarbeitung aus und müssen ebenso als veraltet beurteilt werden[12] .

Backend-Systeme:

Die jüngsten Fortschritte in der Mikroelektronik haben den praktischen Einsatz von assoziativen Platten für PPT- sowie PPS-Systeme ermöglicht, jedoch auf Kosten eines 6-7 mal so hohen Preises wie für konventionelle Platten ([DATE82])[13] . Auch für MPC-Systeme ergeben sich ziemlich hohe Kosten aufgrund der benötigten Spezialhardware. Ein weiterer Einwand gegen MPC-Systeme besteht darin , daß sie ein großes Cache vermutlich auch nicht besser ausnutzen können als konventionelle DB-Systeme dazu in der Lage wären.

Zusammenfassende Bewertung:

Trotz intensiver Anstrengungen ist mit den bis jetzt vorgeschlagenen Backend-Systemen für DB-Anwendungen noch kein entscheidender Durchbruch in

[10] Bei komplexen DB-Suchanfragen auf sehr großen DBs wird sich dieser Engpaß somit noch weiter verschärfen.

[11] Für bessere Verfahren siehe [BAHR80].

[12] Für bessere Verfahren siehe [ELHA82].

[13] Aus diesem Grund sind PPT-Systeme für große DBs nicht geeignet; z.B. ist RAP in seiner Basisversion auf 10 MBytes beschränkt.

Richtung wirtschaftlicher Hochleistungssysteme gelungen. Was die effiziente Implementierbarkeit der drei Datenmodelle (hierarchisch, Codasyl, relational) betrifft, so lassen sich folgende Schlußfolgerungen ziehen:

(a) Aufgrund der satzweisen navigierenden Verarbeitungsform von hierarchischen sowie Codasyl-DBs ist auch in Zukunft hier keine weitere wesentliche Leistungssteigerung durch den Einsatz spezieller DB-Maschinen zu erwarten.

(b) Die Effizienz existierender Implementierungen des Relationenmodells mittels eines konventionellen DB-Systems ist zwar im Augenblick noch geringer als die für Codasyl- bzw. hierarchische Systeme erzielbare Leistungsfähigkeit. Aber die in [KING80] präsentierten Studien deuten darauf hin, daß relationale Systeme mittels einer konventionellen DB-Architektur mit einer vernünftigen Leistungsfähigkeit implementierbar sind und daß 'exotische' Hardware keine Voraussetzung für relationale Systeme ist. Diese Einschätzung in [KING80] wollen wir aufgreifen und darüberhinaus die Behauptung aufstellen, daß Spezialhardware auch keine Voraussetzung für relationale Hochleistungssysteme ist.

2. Basisentwurf einer leistungsfähigen DB-Architektur.

2.1. Allgemeine Entwurfskriterien für Hochleistungs-DB-Architekturen.

Transaktionsorientierte DB-Systeme mit sehr hoher Leistung[14] sind prinzipiell durch den Einsatz von besserer Hardware (schnellere Prozessoren, große und schnelle Speicher, leistungsfähige Kanäle, etc.) sowie durch eine möglichst hohe Parallelität bei der Transaktionsverarbeitung erreichbar. Die Verwendung von schnellerer und spezialisierter Hardware alleine gibt jedoch noch keine Garantie für einen erhöhten Transaktionsdurchsatz oder für kürzere Antwortzeiten, wie die Diskussion über existierende DB-Maschinen angedeutet hat. Die Ursache dafür ist in Engpässen zu suchen, die durch eine vorgegebene Systemarchitektur erzeugt werden (z.B. zu niedrige, satzorientierte Schnittstellen zwischen Host und Backend).
Folglich sind bei der Konstruktion eines DBMS folgende Probleme im Zusammenhang zu lösen:

(Prob11) Zerlegung des komplexen DBMS in funktionale Teilsysteme mit geeigneten Schnittstellen.

(Prob12) Auswahl geeigneter Hardware und eine geeignete Abbildung der Teilsysteme auf diese Hardware.

Bei diesen Bestrebungen nach einer ganzheitlichen Systemkonzeption im Sinne eines optimalen Zusammenspiels von Hardware- und Softwarekomponenten müssen im einzelnen folgende Aspekte berücksichtigt werden:

(A) Geeignete Wahl von Schnittstellen zwischen Teilsystemen:
Wie die Diskussion über die Nachteile einiger existierender DBMS offengelegt hat, kann sich die navigierende Einzelverarbeitung von Sätzen/Tupeln nachteilig auf das Gesamtleistungsvermögen auswirken. Deshalb sollte ein Ziel einer Neuentwicklung eines DBMS sein, daß

[14] Wir wollen nochmals darauf hinweisen, daß wir ein DBMS für ein vollständiges, nicht eingeschränktes Datenmodell konstruieren wollen.

effizient implementierbare mengenorientierte[15] Schnittstellen zwischen Teilsystemen vorhanden sind, wo es sinnvoll ist. Diese wünschenswerte Eigenschaft ist insbesondere auch im Hinblick auf Backend-Architekturen zu sehen, wo es wichtig ist, daß der Anteil der Arbeitslast, welcher an einen Backend deligiert werden kann, signifikant größer ist als der verursachte Kommunikationsmehraufwand.

(B) Zerlegungsprinzipien zur Definition von geeigneten Teilsystemen:

* Das DBMS sollte so konzipiert werden, daß es in einfacher und billiger Weise möglich ist, separate und autonome Rechenleistung zwischen die physische DB und der Benutzerumgebung zu legen.
* Das permanente Speichermedium für die physische DB sollte ein günstiges Preis/Leistungsverhältnis aufweisen und sollte über eine für die meisten praktischen Anwendungen ausreichende Speicherkapazität verfügen[16] .
* Reduktion von zu transportierenden Datenmengen (Datenfilterung): Zwischen den einzelnen Teilsystemen sollten nur relevante Daten übertragen werden.
* Das Teilsystem, welches für die permanente Speicherung der physischen DB verantwortlich ist, sollte möglichst autonom arbeiten können und eine eigenständige Kontrolle über die effiziente Belegung des DB-Speichermediums ausüben.
* Das Zusammenspiel der Teilsysteme sollte so geregelt sein, daß die Zuverlässigkeit und Verfügbarkeit des Gesamtsystems nicht vermindert wird.

(C) Abbildung auf die Hardware:

Sowohl bei der Definition der Schnittstellenoperatoren von Teilsystemen als auch bei der Auswahl der Hardware, auf der die betreffenden Teilsysteme ablaufen, ist auf eine effiziente Realisierbarkeit zu achten. Insbesondere sollte sichergestellt sein, daß die zu verwendende Hardware eine effiziente Implementierung von komplexen DB-Basisoperationen gestattet. Ferner ist zu berücksichtigen, daß der

---

[15] Mengenorientiert ist dabei als Gegensatz zu Satz/Tupel-Schnittstellen zu verstehen.

[16] Diese Forderung schließt z.B. Festkopfplatten aus.

potentielle Parallelismus mehrerer Prozessoren für eine echte[17] Parallelverarbeitung bei der Transaktionsabarbeitung ausgenutzt werden kann.

Die Fähigkeit, einzelne Standard-DB-Operationen[18] effizient ausführen zu können, ist jedoch in keiner Weise ausreichend für eine höhere Leistung noch für ein kommerziell lebensfähiges DB-System. Das Ziel einer Neuentwicklung muß vielmehr ein funktional vollständiges Design für ein DBMS sein, in dem alle Systemkomponenten (Software und Hardware) harmonisch (synergistisch) zusammenarbeiten. Die Beurteilung eines Systementwurfs muß sich dann mit allgemeinen Problemkreisen wie Verfügbarkeit, Zuverlässigkeit, Erweiterbarkeit und Leistungsfähigkeit bei der Transaktionsabarbeitung beschäftigen. Bei der zentralen Frage des effizienten Zusammenspiels der einzelnen DBMS-Teilsysteme zur Realisierung des Transaktionskonzepts darf man sich jedoch auf keinen Fall darauf beschränken, das DBMS einzig und allein auf eine hohe Retrieval-Leistungsfähigkeit zu konzipieren und andere wichtige Aspekte wie Backup und Recovery beim Systementwurf in der ersten Phase erst einmal auszuklammern. Vielmehr ist ein vollständiger Systementwurf zu erarbeiten, der es erlaubt, spezielle DBMS-Aufgaben wie Retrieval, Updates, Backup und Recovery, Synchronisation paralleler Transaktionen, Queryoptimierung, DB-Restrukturierung, Systemtuning u.a. gleichzeitig effizient zu lösen.

Neben diesen allgemeinen Erwägungen sind für die Beurteilung eines vorliegenden Systementwurfs folgende konkreten Aspekte primär von Interesse:

- Reduziert die Systemarchitektur die Gefahr für die beiden häufigsten Engpässe in bestehenden DBMS, nämlich I/O-Engpaß und CPU-Engpaß?
- Wie hoch sind die (zusätzlichen) Hardwarekosten für die vorliegende Architektur, und wie aufwendig ist die Konvertierung von einem bestehenden konventionellen DBMS in die neue Architektur (evolutionär oder revolutionär)?

---

[17] Im Sinne von gleichzeitiger Bearbeitung (im Gegensatz zum Multiplexing von Transaktionen bei 1-Prozessorsystemen).

[18] wie z.B. Restriktion oder Projektion in der Relationenalgebra

2.2. Grobstruktur der neuen DB-Architektur.

Der Neuentwurf eines leistungsfähigen 'General-Purpose'-DB-Systems sollte sich stark am kommerziellen Hardware-Angebot orientieren, und zwar sowohl an der Funktionalität als auch am Preis/Leistungsverhältnis der momentan oder in naher Zukunft verfügbaren Hardware. Als relevante technologische Entwicklungen bei Rechenanlagen, welche für die Konzipierung einer zukünftigen, kommerziell einsetzbaren Hochleistungs-DB-Architektur von Bedeutung sind, sind die folgenden Bereiche zu nennen:

(HW1) Große Arbeitsspeicher (AS):
Rechner werden in nächster Zukunft mit wesentlich größerem AS ausgestattet werden können. Angekündigt sind zur Zeit bereits Rechner von IBM und Fujitsu mit 32 MBytes bzw. 128 MBytes AS-Kapazität.

(HW2) Schnelle Magnetplatten mit großer Kapazität:
Die Speicherdichte von konventionellen Magnetplattenspeichern hat sich in den letzten Jahren enorm erhöht, sowohl bzgl. der Speicherdichte als auch bzgl. der Anzahl von Spuren pro Plattenoberfläche. Derzeit verfügbar sind Plattenlaufwerke mit ca. 40 MBytes pro Oberfläche und ca. 700-800 MBytes pro Laufwerk (etwa die IBM 3380). Das erlaubt, den Hintergrundspeicher für die langfristige Datenhaltung einer modernen Rechenanlage im Umfang von z.B. 10 GBytes mit 15 Plattenlaufwerken zu realisieren. Solche Standardplatten werden somit auf lange Zeit das dominierende Speichermedium für die physische DB sein. (Blasenspeicher und CCDs werden nur vereinzelt zum Einsatz kommen.)
Weitere Eigenschaften der schnellen Platten:
Um Information von einer Platte in den AS transportieren zu können, müssen in einem konventionellen DBMS[19] sehr häufig Bewegungen des Lese/Schreibarms vorgenommen werden. Diese Verzögerungen und Tot-

[19] Die Transporteinheit ist i.a. ein Block von 1-4 KBytes.

zeiten machen den bei weitem größten Zeitaufwand für I/Os von der Platte in den AS aus; typische Werte liegen bei 30-50 msec. Die hohen Positionierungszeiten sind durch die Eigenschaften mechanischer Teile bedingt, und es ist nicht absehbar, daß auf diesem Gebiet wesentliche technologische Fortschritte erzielbar sind. Aufgrund der sich ständig erhöhenden Speicherdichte pro Spur zeichnen sich somit zwei Eigenschaften ab, nämlich

* der Zugriffsengpaß für Random-Plattenzugriffe wird immer akuter,
* physisch-sequentielles Plattenlesen wird ständig schneller werden.

(HW3) Hohe Übertragungsraten:

Infolge der höheren Speicherdichte der neuen Platten ergibt sich beim physisch-sequentiellen Lesen eine höhere Datenrate, welche vom Platten-Controller über den Kanal in den AS übertragen werden muß. Fortschritte in der Kanaltechnologie ermöglichen es, daß die Transportleistung zwischen Controller und AS mit der Datenrate der neuen Platten Schritt hält (IBM 3380: ca. 3MBytes/sec).

(HW4) Preisgünstige, leistungsstarke Mikros:

Heutzutage werden billige und leistungsstarke Mikro-Computer von einer Vielzahl von Herstellern auf dem DV-Markt angeboten. Dabei ist auch hier ein Ende der Tendenz 'mehr Leistung für weniger Geld' nicht abzusehen. Beispielsweise kostet ein Motorola 68000 mit einem 1 MIPS General-Purpose Prozessor samt RAM-Speicher momentan ca. 4000$.[20] Solche oder ähnliche Mikros können zu einer funktionalen Aufrüstung von Platten ('intelligente Platten') eingesetzt werden.

(HW5) Optische Platten:

Es ist zu erwarten, daß diese Billigspeicher mit Kapazitäten von mehr als 1 GBytes pro Plattenoberfläche in den nächsten Jahren verfügbar werden. Ihre Verwendung wird dann hauptsächlich im Bereich der Datensicherung/Datenarchivierung zu finden sein.

[20] Diese MIPS-Rate eines Mikros ist nicht mit der MIPS-Rate einer Jumbo-CPU vergleichbar, da der Befehlssatz eines Mikros i.a. weniger komplexe Instruktionen enthält.

Vor diesem Hintergrund einer stürmischen technologischen Fortentwicklung von konventioneller Hardware ist der folgende Grobentwurf für eine Hardware-Architektur zukünftiger Hochleistungs-DB-Systeme zu sehen.

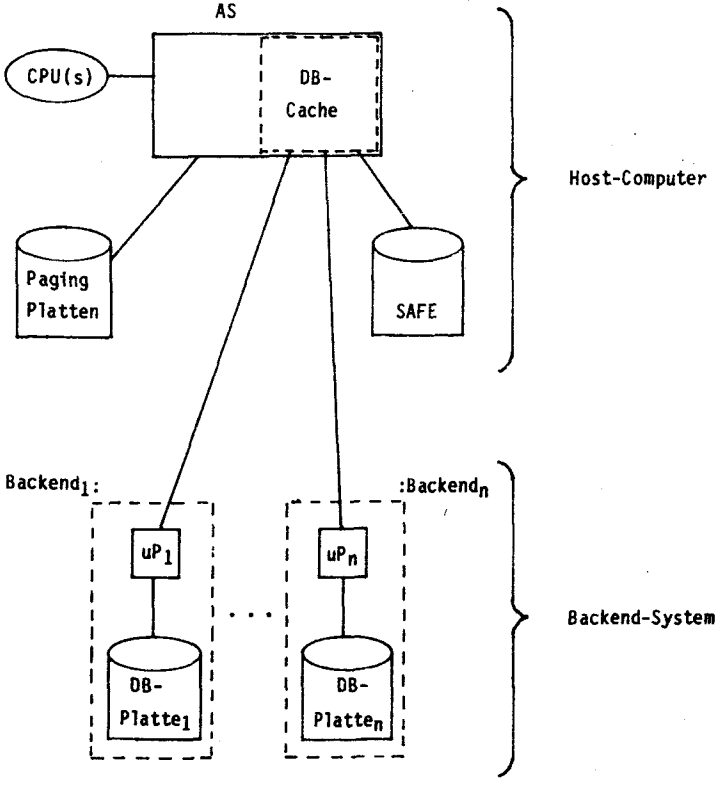


Abb.2.1: Hardware-Konfiguration mit zentralem DB-Cache/SAFE und intelligenten DB-Platten.

Der Host-Computer ist ein konventioneller 'Mainframe' (Ein- oder Mehrprozessoranlage), auf dem neben DB-Transaktionen i.a. noch andere

Anwendungen laufen[21] .
Das Backend-System ist verantwortlich für Zugriffe auf benötigte Daten der physischen DB, welche auf den Standard-Plattenlaufwerken DB-Platte$_1$,..., DB-Platte$_n$ abgespeichert ist. Zwischen jeder DB-Platte$_j$ und dem DB-Cache ist ein konventioneller Mikrocomputer uP$_j$ mit eigenem lokalen Speicher eingebaut[22] ; die Verbindung von uP$_j$ mit DB-Platte$_j$ ist durch einen schnellen Kanal hergestellt.
Insgesamt betrachtet stellt die vorgeschlagene Hardware-Konfiguration also eine evolutionäre Fortentwicklung der Hardware-Konzeption für ein konventionelles DB-System dar. Im Einklang mit den vorher postulierten Prinzipien zur Konstruktion von Hochleistungs-DB-Systemen geben wir nun eine grobe Zerlegung des komplexen DBMS in funktionale Teilsysteme sowie deren Abbildung auf die angegebene Hardware-Konfiguration an.

Grundzüge einer Architektur für zukünftige DB-Systeme.

(1) Benutzerschnittstellen:
Der DB-Benutzer soll über eine hohe Schnittstelle zum DBMS verfügen. Es scheint derzeit nicht notwendig oder opportun zu sein, neue Benutzerschnittstellen zu entwickeln. Heutige Benutzerschnittstellen wie SQL in SystemR ([ASTR76]) oder Quel in INGRES ([STON76]) sollen beibehalten werden. Die herausragende Eigenschaft dieser Sprachen ist dabei ihr deskriptiver Charakter, d.h. der Benutzer muß nur angeben, was er haben will und nicht, wie das DBMS es ihm beschaffen soll. Fragen der Realisierung und Effizienz werden somit an das DBMS deligiert. Mit der Wahl einer deskriptiven Benutzerschnittstelle liegt quasi auch die Wahl des Relationenmodells als zugrundeliegendes Datenmodell fest. Neben den bekannten konzeptionellen Vorteilen auf logischer Ebene (vgl. z.B. [KING80]), wollen wir eine Eigenschaft herausheben, welche von vielen bisher als nachteilig für eine effiziente Implementierbarkeit empfunden wurde, nämlich seine mengenorientierten Operatoren. Gerade diese Eigenschaft wird in unserer

[21] Die Funktionen von DB-Cache und SAFE werden anschließend erläutert.

[22] Es liegt somit eine verallgemeinerte PPD-Architektur vor.

Architektur als Vorteil angesehen, welche eine effiziente Implementierung erst ermöglicht.

(2) DB-Cache/SAFE:

Das DB-Cache ist ein Teilbereich des AS im Host. Es dient als Arbeitsspeicher zur Abarbeitung von Transaktionen. Zugriffe auf das DB-Cache werden somit für folgende Zwecke benötigt:

- Für die Realisierung der funktionellen Erfordernisse des unterstützten Datenmodells für die Berechnung und Zwischenspeicherung von Zwischen- und Endergebnissen.
- Für DBMS-Funktionen, welche für Logging/Commit/Recovery verantwortlich sind. Der SAFE ist ein Speichermedium (idealerweise optische Platten), auf den Log-Information zur Recovery von Systemfehlern geschrieben wird.
  In [ELHA82] sind bereits schnelle Commit- und Wiederanlaufverfahren für eine konventionelle DB-Architektur entwickelt worden. Diese Algorithmen weisen die von uns geforderte Eigenschaft einer harmonischen Wechselwirkung zwischen DBMS-Teilsystemen insofern auf, als sie eine integrierte Lösung für das Synchronisations-, Commit- und Recoveryproblem darstellen. Dieses DB-Cache/SAFE-Verfahren soll somit auch auf unsere Architektur übertragen werden (mit den evt. notwendigen Modifikationen).

(3) Geschickte Verteilung wichtiger DBMS-Funktionen:

Im Sinne einer effizienten Abbildung von DBMS-Funktionen auf die verfügbare Hardware, nehmen wir folgende Aufteilung von Operationen der Relationenalgebra vor:

- Jeder uP_j soll einstellige relationale Operationen - wie z.B. Restriktion oder Projektion ohne Duplikatelimination - , welche aufgrund tupellokaler Bedingungen auswertbar sind, ausführen können. Die Realisierung solcher Operationen ist mithilfe kleiner uP_j-Arbeitsspeicher erreichbar und die Auswertungsgeschwindigkeit sollte nach Möglichkeit mit voller Plattengeschwindigkeit

geschehen (Im-Flug-Filterung) [23] .

- Alle anderen relationalen Operationen, welche obige Bedingung nicht erfüllen (wie z.B. der Join), sollen im DB-Cache abgearbeitet werden. Dabei gehen wir davon aus, daß im Normalfall volle Zwischenergebnisoperanden zur Ausdrucksauswertung im DB-Cache zur Verfügung stehen.

Diese Auslagerung komplexer Operationen auf das Backend-System verspricht beträchtliche Leistungsgewinne aufgrund paralleler Operationen von den uP_{js} und den Host-CPU(s). Zusätzlich wird die Gefahr eines CPU-Engpasses im Host durch eine starke Einsparung an CPU-Zyklen verringert.

(4) Transporteinheit zwischen $Backend_j$ und DB-Cache:

In den meisten Fällen - z.B. zur Beantwortung komplexer relationaler Anfragen - soll die Transporteinheit zwischen $Backend_j$ und dem DB-Cache eine ganze Zwischenergebnisrelation (ZER) sein. Im Vergleich zu einem konventionellen DB-System führt dies zu einer starken Reduktion der teuren I/O-Unterbrechungsbehandlungen im Host. Zusätzlich führt dies aufgrund der Datenfilterung durch die uP_{js} zu einer wesentlich reduzierten Informationsübertragung in das DB-Cache, sodaß die (wesentlich vergrößerte) DB-Cache-Kapazität ausreicht, um volle ZERs aufnehmen zu können. Die Datenfilterung bewirkt überdies eine Einsparung an AS-Zyklen im Host. Nur in denjenigen Fällen, wo navigierender Zugriff auf einzelne Datenobjekte noch sinnvoll ist[24] , soll die Transporteinheit einer Seite noch beibehalten werden.

---

[23] Der Einbau eines Mikros als Datenfilter zwischen Host-AS und einem DB-Plattenstapel wurde bereits in [LANG77] vorgeschlagen; die Mächtigkeit dieses Filters ist jedoch geringer (nur eingeschränkte Restriktionen auf Seitenbasis). In [BANC80] hingegen wird ein funktional etwas mächtigerer Filter (Restriktion sowie Projektion im Flug) vorgeschlagen; zur Realisierung dieses Filters scheint aber die Konstruktion von Spezialhardware notwendig zu sein. Typischerweise wird bei beiden Vorschlägen auf die Spezifikation eines darauf aufbauenden kompletten DBMS verzichtet.

[24] Das ist z.B. beim Zugriff auf einen Knoten eines B-Baums der Fall.

(5) Weitere Aufgaben der Backend$_{js}$:

Ene weitere Entlastung des Hosts von störenden Verwaltungsaufgaben soll dadurch erreicht werden, daß die uP$_{js}$ weitestgehend autonom für die Verwaltung der DB-Platten zuständig sind. (z.B. Einbringen von Updates auf die DB-Platten, oder effiziente Abspeicherung von Relationen mit dem Ziel, die Fähigkeit der neuen DB-Platten zu schnellem physisch-sequentiellen Lesen konsequent auszunutzen.)

Der von uns erhobene Anspruch ist nun, daß die Konstruktion eines DB-Systems unter Verwendung der betrachteten Hardware-Konfiguration und unter Einhaltung der aufgezählten Architektureigenschaften des DBMS ein erfolgversprechender Weg ist für zukünftige Hochleistungs-DB-Systeme, und zwar insbesondere im Hinblick auf kommerzielle Anwendungen aufgrund der ausschließlichen Verwendung von konventioneller Hardware. Um diesen Anspruch zu untermauern, beschäftigt sich die weitere Arbeit mit folgenden Aspekten:

* Kap.3 bringt einen funktional vollständigen Entwurf des DBMS mit einer geeigneten Abbildung der einzelnen DMBS-Komponenten auf die zugrundeliegende Hardware. Dabei werden insbesondere Fragen der effizienten Implementierung von Schnittstellenoperatoren behandelt.
* Kap.4 beschäftigt sich mit allgemeinen Fragen der Queryoptimierung, sowie im speziellen mit der Konstruktion effizienter Auswertungalgorithmen. Ein Leitmotiv bei diesen Überlegungen wird dabei sein, daß der postulierte Normalfall , Information für parallele Transaktionen paßt so lange wie für eine effiziente Verarbeitung benötigt in das DB-Cache, für eine sehr große Klasse von Transaktionen erfüllt ist[25] .
* Kap.5 schließlich befaßt sich mit konkreten Kostenfunktionen, insbesondere im Hinblick auf die Frage des Nutzens von Indexen in der betrachteten Architektur.

[25] Der Ausnahmefall muß zwar behandelbar sein, darf aber, da er nur selten vorkommt, aufwendig werden und sollte mit möglichst einfacher Software gelöst werden.

3. Funktionale Beschreibung der neuen Architektur.

3.1. Allgemeines zu DB-Schichtenmodellen und DBMS-Spezifikationsmethoden.

Allgemein bezeichnet der Begriff 'Architektur eines Systems' den Gesamtaufbau eines Systems, d.h. die Strukturierung seiner Komponenten und deren Wechselbeziehungen untereinander. Da DB-Verwaltungssysteme äußerst umfangreiche Software-Pakete sind, besteht in der Literatur Einigkeit darüber, daß beim Entwurf eines DBMS eine Modellbildung in mehrere Schichten vorgenommen werden muß. Eine solche hierarchische Strukturierung erfüllt dabei zwei wesentliche Forderungen an die Qualität eines DB-Designs:

- Strukturierte, zuverlässige, robuste und verifizierbare Software-Technologie (DBMS software engineering) wird ermöglicht.
- Jede Ebene der hierachischen Strukturierung bietet die Möglichkeit zur Erhöhung der Datenunabhängigkeit, wenn organisatorische Details der darunterliegenden Ebenen verborgen bleiben (need-to-know Prinzip, Geheimhaltungsprinzip [PARN72]).

Somit ist das Ziel eines hierarchischen DB-Systementwurfs die Realisierung mehrerer möglichst voneinander unabhängiger Entwurfsebenen. Die Systemstruktur wird dabei durch eine Menge von abstrakten (virtuellen) Maschinen modelliert. Jede Maschine in einer solchen Hierarchie charakterisiert das DB-System auf einem spezifischen Abstraktionsniveau und verbirgt somit einige Aspekte der darunterliegenden Maschine (vgl. auch [HAER78]).
Eine virtuelle Maschine besitzt die Funktion einer formalen Schnittstelle, sie stellt jeweils eine Menge von Objekten und Operatoren zur Verfügung und dient als Basismaschine zur Implementierung der nächst höheren Maschine.
Ein implementierungsunabhängiges DBMS-Schichtenmodell ist in [SENK72] und [DITT77] beschrieben worden:

level 5 ≡ Ebene der logischen Sicht des Benutzers
level 4 ≡ Ebene der logischen Sicht des Systems
level 3 ≡ Ebene der logischen Zugriffspfade
level 2 ≡ Ebene der Speicherstrukturen
level 1 ≡ Ebene der Speicherzuordnungsstrukturen

Neben dieser horizontalen Systemstrukturierung (funktionale Abstraktion) ist in der Literatur über den Problemkreis der Anwendung formaler Spezifikationsmethoden auf DBMS auch die Notwendigkeit einer vertikalen Strukturierung (Datenabstraktion) in Moduln betont worden ([YEH77], [WEBE78]). Die Zerlegung eines komplexen Systems in Moduln geschieht dabei datengetrieben, d.h. ein Modul wird entworfen, um aufgerufen zu werden zur Erzeugung und Manipulation von Datenobjekten eines bestimmten Typs. Ein Modul besteht dabei aus zwei Teilen, einer Schnittstelle und einem Rumpf. Die Spezifikation eines Moduls ist eine implementierungsunabhängige, funktionale Beschreibung seiner Schnittstellenoperatoren. Die Implementierung dieser Operatoren wird im Modulrumpf angegeben und bleibt somit seinem Aufrufer verborgen. Soweit betrachtet ähnelt das Modulkonzept dem Konzept des abstrakten Datentyps. Unterschiede treten jedoch im Verhalten der Interaktionen von mehreren Moduln zutage. Im Idealfall sollte bei der Konstruktion von Moduln folgendes Grundprinzip eingehalten werden: Moduln sollten nur eigene Datenobjekte manipulieren (Lokalitätsprinzip), und somit nach Möglichkeit keine Seiteneffekte bewirken. Aufgrund von Leistungserfordernissen kann dieses wünschenswerte Ziel in der Praxis nicht immer vollkommen erreicht werden. Insgesamt gesehen bildet jedoch das Modulkonzept in Verbindung mit einem Schichtenmodell ein geeignetes Mittel zur Spezifikation (und Implementierung) eines DBMS. Die Beziehungen zwischen Moduln sollten dabei hierachischer[26] Natur sein, und können somit durch Auftragsbeziehungen zwischen Moduln charakterisiert werden.
Bei der sich nun anschließenden funktionalen Beschreibung unserer neuen DBMS-Architektur werden wir die eben erläuterte Entwurfsmethodologie soweit

[26] Bei der Konstruktion von Betriebssystemen streng nach der hierachischen Vorgehensweise sind in der Praxis oft viele Probleme aufgetreten. Es ist jedoch die Überzeugung des Autors, daß eine hierachische Strukturierung eines DBMS, welche den praktischen Leistungsanforderungen gerecht wird, möglich ist.

sinnvoll anwenden. Aus naheliegenden Gründen wird die Spezifikation der Systemebenen und Systemmoduln in informeller Weise erfolgen. Desweiteren geben wir nicht nur eine Beschreibung des Systems als solches, sondern es werden bei wichtigen Schnittstellenentscheidungen auch die entsprechenden Motivationen und Entscheidungskriterien dokumentiert. Damit soll der Entscheidungsprozeß für die Auswahl von Schnittstellenoperatoren und Datenstrukturen offengelegt werden, um nachweisen zu können, daß die getroffenen Entscheidungen sehr gut auf die vorgegebene Systemkonfiguration abgestimmt sind.

3.2. Entwurfskriterien, Modularisierung und Schnittstellenbeschreibungen.

3.2.1. Schichtenmodell und Modularisierung.

Bei dem Entwurf eines DBMS-Schichtenmodells für unsere Architektur wurde von folgenden Überlegungen ausgegangen:

(a) Ist die strukturelle Zergliederung des DBMS in fünf hierachische Systemebenen auch in unserer Systemumgebung in dieser Weise sinnvoll?
(b) In welcher Weise sollen funktionale Teilsysteme zwischen Host und Backend aufgeteilt werden?

Das Ergebnis dieser Überlegung ist in dem nachfolgendem Diagramm festgehalten.

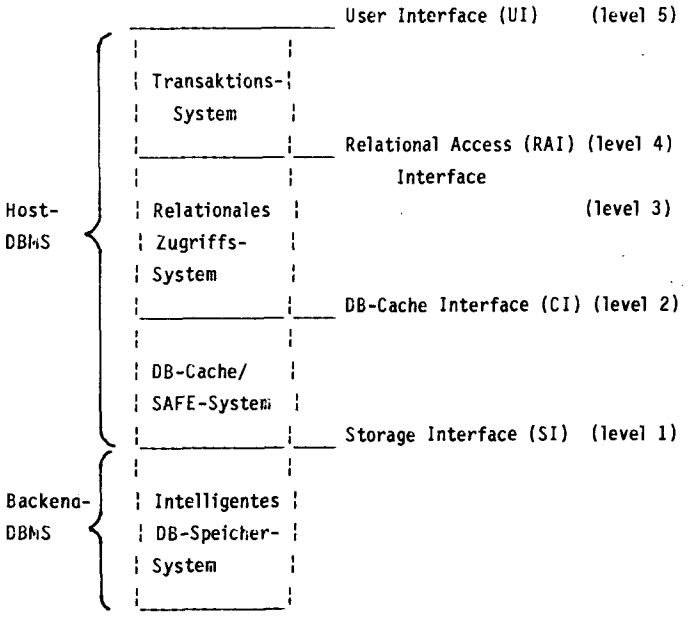


Abb.3.1: DBMS-Schichtenmodell für die neue Architektur.

Die Aufteilung der einzelnen funktionalen Teilsysteme auf Host oder Backend würde gemäß des folgenden Prinzips getroffen:
Logische Aspekte sind vom Host zu übernehmen, physisch-orientierte Aspekte dagegen vom intelligenten Backend.

Den einzelnen Hauptkomponenten dieser funktionalen Aufteilung fallen dabei folgende Aufgaben zu:

- Das Transaktions-System bildet deskriptive Anweisungen der logischen Sicht des Benutzers (Schnittstelle UI, level 5) auf die logische Sicht des DBMS ab (Schnittstelle RAI, level 4). Dabei sind Funktionen wie Integritäts- und Zugriffskontrolle und Queryoptimierung wahrzunehmen. Die Frage des Bindungszeitpunktes zwischen Transaktionsprogrammen und DB-Daten (interpretative oder compilierte Ausführung) ist dabei für die weiteren Architekturüberlegungen irrelevant und wird somit nicht eigens diskutiert.
- Das Relationale Zugriffs-System mit seiner Schnittstelle RAI bildet die virtuelle Maschine, auf der Transaktionen ausgeführt werden. Intern werden dabei Aufgaben wie die Realisierung logischer Zugriffspfade und Systemkatalogzugriffe (level 3) als auch die korrekte Synchronisation paralleler Transaktionen erfüllt.
- Das DB-Cache/SAFE-System ist verantwortlich für die Verwaltung des DB-Cache, welcher Teil des Host-Arbeitsspeichers ist. Eine wesentliche Entwurfsvorgabe stellte dabei die Forderung, daß die in [ELHA82] entwickelten Algorithmen für leistungsfähige DB-Pufferverwaltung /Commit- und Recoveryverfahren auch in die veränderte Systemumgebung integriert werden sollen. Das DB-Cache bietet dem Relationalem Zugriffs-System einen integrierten Arbeitsbereich zur Erfüllung seiner Aufgaben an; somit entspricht die Schnittstelle CI dem level 2 der Speicherstrukturen.
- Das Intelligente Speicher-System nimmt die Speicherzuordnung (Schnittstelle SI, level 1) von permanenten DB-Objekten auf physischen Langzeitspeicher, den DB-Platten, vor. Aufgrund der eigenständigen lokalen Intelligenz des Backends stellt die SI-Schnittstelle eine wesentlich höhere Schnittstelle als in konventionellen DB-Systemen dar, ebenso ist die bisherige passive Rolle dieses Systemteils durch komplexe Filter-Retrievaloperationen wesentlich verändert.

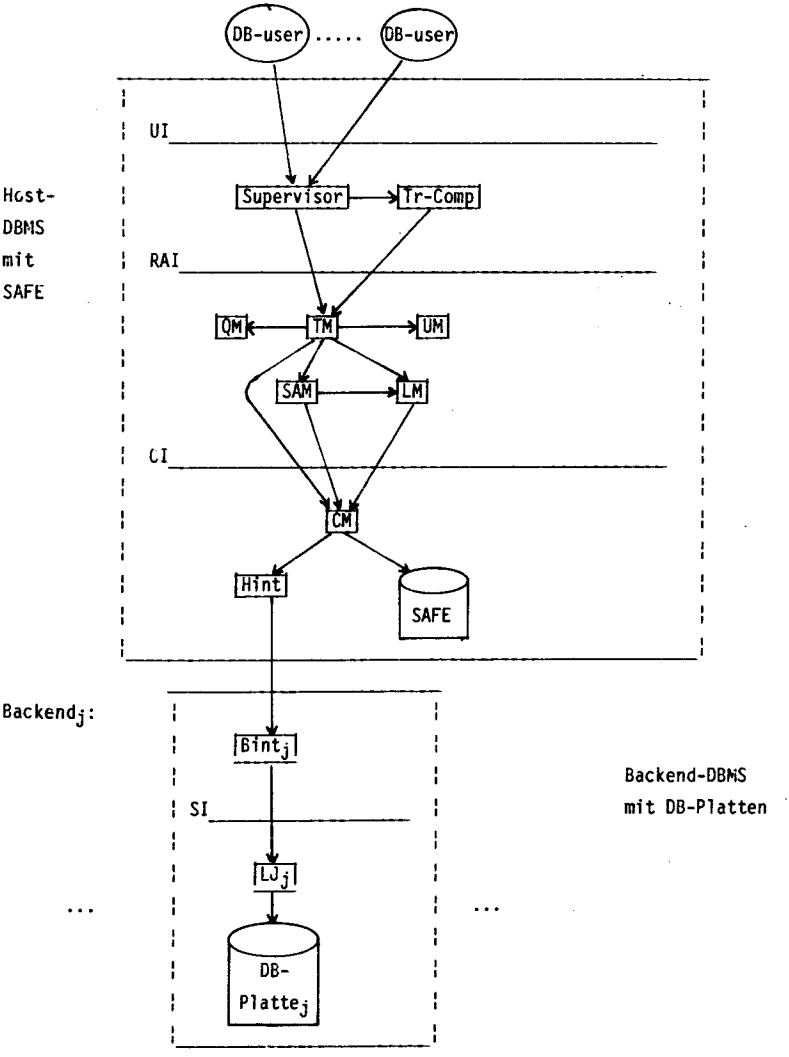


Abb.3.2: Modularisierung und Verteilung des DBMS.

Nach dieser Schichtenbildung geben wir nun eine funktionale Verfeinerung der Systemarchitektur durch eine Internstrukturierung der angegebenen Systemhauptkomponenten in diverse Moduln an. Die angegebene Zergliederung in einzelne Moduln erfolgt dabei sowohl aus funktionalen Aspekten als auch datengetrieben.[27)]

In Abb.3.2 ist die statische DBMS-Struktur mittels einer Auftragshierarchie zwischen den einzelnen Moduln festgehalten (die Pfeile sind somit als Kontrollfluß zu deuten).

Erläuterungen zur Wahl der Modul-Bezeichnungen:

* Tr-Comp: Transaction (Pre-) Compiler
* TM : Transaction Module
* QM : Query Module
* UM : Update Module
* SAM : System Access Module
* LM : Lock Module
* CM : DB-Cache Module
* Hint : Host Interface
* $Bint_j$: Backend Interface to $Backend_j$
* LI_j : Local Intelligence in $Backend_j$

Weitere Anmerkungen zu Abb.3.2:

(a) Das DB-Cache ist in Abb.3.2 nicht abgebildet; die Verwendung des DB-Cache durch das Host-DBMS wird in Kap.3.2.3 diskutiert.

(b) Die Moduln QM und UM setzen keine weiteren Aufträge ab, da die für ihre Aufgaben benötigten Objekte im DB-Cache zur Verfügung stehen (siehe nachfolgendes Kap.3.2.2).

(c) Das Intelligente Speichersystem wird von allen $Backend_j$ (j=1,..., #DB-Plattenlaufwerke) gebildet. Die Funktionalität der Moduln $Bint_j$ und LI_j ist dabei identisch für alle betrachteten j. Die Funktionen von $Bint_j$ und LI_j werden entsprechend Abb.2.1 von einem Mikroprozessor uP_j

[27)] Die Architekturbeschreibung wird top-down erfolgen, obwohl die angegebene Systemarchitektur aus mehreren top-down und bottom-up Entwurfsphasen synthetisiert wurde.

mit eigenem lokalen Speicher realisiert.

Weitere Detaillierung der Aufgaben des Transaktions-Systems:

Die Auswahl einer geeigneten DML für die Benutzerschnittstelle UI ist nicht Gegenstand dieser Arbeit. Um jedoch einen konkreten Bezugspunkt bei späteren Diskussioen zu haben, orientieren wir uns an der deskriptiven Sprache SQL von SystemR mit Einbettung in eine Hostprogrammiersprache ([CHAM76]). Eine Benutzertransaktion mit SQL-Anweisungen wird durch den (Pre)-Compiler Tr-Comp auf Operatoren der Schnittstelle RAI abgebildet. Bevor wir in die Diskussion der RAI-Schnittstelle einsteigen, sollen kurz noch die Aufgaben des Transaktions-Systems detailliert werden.

Der Tr-Comp ist (im Falle einer vollständigen Bindung zum Übersetzungszeitpunkt, [BLAS79a]) verantwortlich für:

- Precompilation
  - Parsing
  - Überprüfung von Zugriffsberechtigungen und Integritätsbedingungen
- Query-Optimierung
  - Transformation von SQL-Anweisungen in einen internen, 'optimalen' Operatorbaum.
  - Optimale Wahl der Sperrgranularität bei mehrstufigen Sperrprotokollen: Dieser Schritt sollte hier erfolgen, da für die Erstellung eines optimalen Operatorbaumes i.a. Schätzungen über die zu erwartende Tupeltrefferanzahl gemacht werden müssen, welche ebenso für die Wahl einer geeigneten Sperrgranularität wesentlich sind.
  - Codegenerierung

Der Supervisor übernimmt die Aufgabe des globalen Schedulings von Benutzertransaktionen, z.B.

- Gültigkeitsüberprüfungen für vorübersetzte Bibliothekstransaktionen.[28)]
- Einrichtung und Auflösung von lauffähigen Transaktionsprozessen.
- Globale Überwachung des DB-Multiprogrammings.

[28)] Falls verwendete Zugriffspfade zwischenzeitlich gelöscht wurden, so muß dies erkannt und eine Neuübersetzung durchgeführt werden.

3.2.2. Spezifikation des Relationalen Zugriffssystems.

Die RAI-Schnittstelle entspricht der logischen Sicht des DBMS und stellt somit die virtuelle Maschine zur Ausführung von Transaktionen dar. Die RAI-Operatoren lassen sich in vier Gruppen einteilen:

(i) Operatoren zur DML-Auswertung (Queryauswertung)
(ii) Operatoren zur DDL-Auswertung
(iii) Operatoren zur Implementierung von Features der Einbettungen in eine Host-Sprache.
(iv) Operatoren zur Realisierung des Transaktionsbegriffs.

Eine grundlegende Entscheidung, welche sehr starken Einfluß auf die Effizienz des gesamten DBMS hat, betrifft die Wahl geeigneter Operatoren zur Queryauswertung.

Basisentscheidung:

Der Teil der Schnittstelle RAI, welcher die Queryauswertung übernimmt, besteht aus

- Operatoren der relationalen Algebra (RelA), sowie
- expliziten mengenorientierten Zugriffspfadoperatoren.

Die Wahl dieser mengenorientierten Schnittstelle (Mehrtupel-Schnittstelle) kann durch folgende Aspekte motiviert werden.

(1) Eine RelA-Schnittstelle behandelt und manipuliert ganze Relationen als einzelne Objekte, wohingegen Schnittstellen vom Typ des relationalen Calculus sich mit Relationen auf einer Tupel-für-Tupel-Basis beschäftigen (vgl. z.B. RSI in SystemR). Eine RelA-Schnittstelle kann daher als eine höhere Abstraktionsebene angesehen werden als diese 1-Tupel-Zugriffssysteme, und sie kann folglich mehr Möglichkeiten bieten für eine High-level Optimierung.
(2) In verteilten Datenbanken mit heterogenen deskriptiven Benutzerschnittstellen ist es manchmal billiger, Queries anstatt Daten zwischen den einzelnen Knoten auszutauschen. Dazu ist es notwendig, über eine (mathematisch) klar definierte Zwischensprache für Queries zu verfügen.

Die RelA ist ein sehr guter Kandidat für eine solche Zwischensprache, worauf die verschiedenartigsten deskriptiven Abfragesprachen der einzelnen Knoten abgebildet werden können.

Diese Vorteile einer RelA-Schnittstelle zum Zugriffssystem sind bereits auch in [SMIT75] und [TODD76] aufgezeigt worden. Für unsere spezielle Systemumgebung kommen jedoch noch zwei weitere Faktoren hinzu, welche die konzeptionellen Vorteile einer RelA-Schnittstelle ergänzen um Effizienzgesichtspunkte bei der Implementierung von RelA-Operatoren.

(3) Die neue Hardware-Umgebung (großes DB-Cache, autonome lokale Intelligenz an den DB-Platten) ist sehr gut geeignet, um RelA-Operatoren effizient zu realisieren, z.B.:
   - Jeder LI_j-Modul ermöglicht effiziente Restriktionen und Projektionen (ohne Duplikatelimination) von ganzen Relationen auf den DB-Platten. (MIMD-Architektur!)
   - Die große DB-Cache-Kapazität sollte für die allermeisten praktischen Anwendungen ermöglichen, daß die für die Operatoren der RelA benötigten (Zwischenergebnis-)Relationen solange wie benötigt vollständig im DB-Cache gehalten werden können. RelA-Operatoren wie z.B. der Join-Operator sollten somit im Normalfall vollkommen cache-intern implementierbar sein.[29)]

(4) Die RelA gestattet die Repräsentation einer übersetzten Query in Form eines Operatorbaumes, was folgende Vorteile bietet:
   - Optimierende Compiler-Techniken können bei der Codegenerierung für solche Operatorbäume eingesetzt werden.
   - Der in der Auswertung von Operatorbäumen inherente Parallelismus kann mit unseren Hardware-Resourcen gut ausgenützt werden.(Host-CPU(s), LI-Prozessoren).

Die Entscheidung, explizite Zugriffspfadoperatoren in die RAI-Schnittstelle mit aufzunehmen, resultiert aus folgender Überlegung:
Globale Query-Optimierung, welche auch die Existenz spezieller logischer Zugriffspfade (z.B. Sekundärindexe) mit einbezieht, soll auf alle Fälle möglich sein.

29) Optimierungsaspekte für die RelA werden ausführlich in Kap.4 diskutiert.

Spezifikation des Moduls TM:

Wie aus Abb.3.2 ersichtlich ist, wird die RAI-Schnittstelle durch die Schnittstellenoperatoren des Moduls TM (Transaction Module) realisiert. Der TM führt Verwaltungsinformation über laufende Transaktionen.[30] Er ist zuständig für die dynamische Kreation, Manipulation und Löschung von Datenobjekten des Typs 'Transaktion'. Ein solches Objekt vom Typ 'Transaktion' beinhaltet in etwa transaktionsspezifische Daten wie:

- Benutzerspezifikation.
- Interner Identifikator dieser Transaktion.
- Transaktionsdeskriptor: dieser enthält Information zur Interpretation von Objekten, welche von den Moduln QM und UM manipuliert werden, sowie deren Abbildung auf das DB-Cache; ferner sind diejenigen Indexe vermerkt, welche vom UM gegebenenfalls dynamisch gewartet werden müssen (des weiteren Integritäts- sowie Privacy-Information).

Die Aufgaben des TM lassen sich in zwei Kategorien zerlegen, nämlich

(A) Einrichtung und Verwaltung von Transaktionsdeskriptoren für Transaktionen, welche aktuell vom DBMS bearbeitet werden, sowie

(B) Scheduling aktiver Transaktionen durch Verteilung gewisser RAI-Aufträge einzelner Transaktionen zur Bearbeitung an andere Moduln.

Bevor wir nun zu genauen Angaben von Operatoren übergehen, müssen wir eine allgemeine Bemerkung vorausstellen:

Die Spezifikation konkreter Parametertypen ist i.a. nur unter Berücksichtigung des Bindezeitpunktes von Transaktionsprogrammen möglich. Zur Erleichterung der Beschreibung gehen wir im folgenden von einer vollständigen Bindung zum Übersetzungszeitpunkt aus, d.h. die Information für den Transaktionsdeskriptor einer Transaktion ist vollständig beim Start der Transaktion bekannt.

[30] Der Begriff 'laufende' Transaktion soll den Unterschied zum statischen Programmtext herausstellen.

TM-Operatoren:

Gruppe A:

- BEGIN_TRANS(<user_spec>,<trans_descr_info>)
- END_TRANS(<user_spec>) RETURNS(<'OK'|'aborted'>)[31]

Der Operator BEGIN_TRANS bewirkt die Kreation eines Objekts vom Typ 'Transaktion', wobei auch ein eindeutiger, invarianter und nichtwiederverwendbarer Transaktionsidentifikator transid zur internen Identifizierung vergeben wird. Damit ist ein Kommunikationspfad zwischen dem DB-Benutzer und dem DBMS geschaffen.[32] END_TRANS schließt die Transaktion atomar ab; dazu erfolgt im Rumpf von END_TRANS ein Aufruf an den Synchronisationsmodul LM, und von dort weiter an den Cache-Modul CM.

Gruppe B:

Diese Gruppe umfaßt RAI-Operatoren, welche an andere Moduln zur Bearbeitung weitergeleitet werden. Entsprechend dem Verteilungsziel treffen wir folgenden Unterteilungen:

(B1) QM-bezogene Operatoren

(B2) UM-bezogene Operatoren

(B3) SAM-bezogene Operatoren

Für Operatoren dieser drei Grupppen, welche zu ihrer Ausführung DB-Cache-Speicherplatz benötigen, fordert der TM vor der Weiterleitung an die entsprechenden Moduln vom Cache-Modul CM den notwendigen Speicherplatz im DB-Cache an.(vgl.auch Beschreibung von QM und CM).

(B4) CM-bezogene Operatoren

Eine Diskussion dieser Operatoren erfolgt zweckmäßigerweise erst nach der Spezifikation des Moduls QM.

---

31) Für interaktive Transaktionen ergibt sich die Notwendigkeit eines weiteren Operators CANCEL_TRANS(<user_spec>,<trans_id>)

32) Der Parameter <trans_descr_info> kann auch Sperrwünsche auf Relationen enthalten, welche an den Modul LM weitergeleitet werden.

Spezifikation des Moduls QM:

Der Modul QM (Query Module) erzeugt, manipuliert und vernichtet Datenobjekte des Typs 'Zwischenergebnisrelation' (ZER) sowie des Typs 'Tupelidliste'. QM führt relationale Operatoren auf ZERs aus und manipuliert Indextrefferlisten vom Typ Tupelidliste (welche bei der Diskussion des SAM noch näher erläutert wird). Alle QM-Operatoren sind somit mengenorientiert.

Die nachstehend aufgeführten QM-Operatoren sind unter den folgenden Voraussetzungen konzipiert:

(a) Die Interpretationsinformation für ZERs sowie Tupelidlisten ist im entsprechenden Transaktionsdeskriptor enthalten und steht dem QM zum Lesen zur Verfügung (d.h. QM-Operatoren sind 'generic procedures').

(b) Speicherplatz für Operanden- oder Resultat-ZERs (analog für Tupelidlisten) ist im DB-Cache bereits reserviert worden,[33] die entsprechenden Cache-Adressen sind dem betreffenden Transaktionsdeskriptor zu entnehmen. Dabei wird vorausgesetzt, daß diese ZERs bzw. Tupelidlisten im Normalfall vollständig in das DB-Cache passen.

(c) Die für eine Transaktion reservierten ZERs bzw. Tupelidlisten stehen -wenn sie nicht explizit freigegeben werden- bis zum Transaktionsende für eine direkte Adressierung zur Verfügung.

QM-Operatoren:

- Standard-RelA-Operatoren auf ZERs:
  - RESTRICT . PROJECT
  - JOIN
  - UNION
  - INTERSEC
  - MINUS
  - CONTAINS

Im Interesse einer Reduktion des benötigten DB-Cache-Platzes ist eine

[33] Diese a priori Reservierung basiert auf einer Schätzung des benötigten DB-Cache-Platzes. Durch die in Kap.3.2.3 angegebene Organisation des DB-Cache ist die Möglichkeit zur Nachforderung von Speicherplatz ebenfalls gegeben.

kombinierte Ausführung mehrerer der obigen Operatoren (pipelining of operations) denkbar und wünschenswert.

- Eingebaute Aggregatfunktionen auf ZERs:
  - z.B. MIN, MAX, SUM, AVG, COUNT
- Verallgemeinerter Sortieroperator (Sortieren und Duplikatelimination)
  - SORT&ELIM
- Manipulation von Tupelidlisten:
  - UNION
  - INTERSEC
  - SORT&ELIM

Es ist nochmals zu betonen, daß alle QM-Operatoren vollkommen DB-Cache-intern implementierbar vorausgesetzt werden. Insbesondere für die Operatoren JOIN und SORT&ELIM bedeutet das, daß keine Externspeicherzugriffe auf benötigte Daten notwendig sind. Die Semantik der angegebenen QM-Operatoren wird als selbsterklärend angenommen, deshalb ist auf die Angabe von Parametern verzichtet worden. Als effiziente Einsatzmöglichkeiten des (internen!) Sortieroperators SORT&ELIM ist zu nennen:

(i) Sortieren von ZERs zwecks nachfolgender effizienter Elimination von Duplikattupeln (vgl auch Diskussion der LI-Filteroperationen).

(ii) Effiziente Vorverarbeitung für eine nachfolgende Berechnung von Aggregatfunktionen (vgl. GROUP BY-Klausel in SQL).

(iii) Geeignete Sortierung von Tupelidlisten zwecks nachfolgender Materialisierung dieser Tupel durch die LI (vgl. dazu spätere Abschnitte 3.2.4.1 und 3.3.2.1).

(iv) Als vorbereitender Schritt für schnelle JOINS, UNIONs und INTERSECs.

Damit ist die informelle Spezifikation des QM abgeschlossen. Als Nachtrag zur TM-Spezifikation bringen wir die Spezifikation der CM-bezogenen Operatoren des TM:

Diese Gruppe von TM-Operatoren betrifft die Materialisierung von Tupelmengen im DB-Cache.

- COMPLETE_RELSCAN(<Relid>,<restr_predicate>,<attr_list>,<exp.size>)

RETURNS(<DB-Cache_addr>)

. SELECTIVE_RELSCAN(<Tupelidlist>,<restr_predicate>,<attr_list>,
<exp.size>) RETURNS(<DB-Cache_addr>)

Das Ergebnis dieser Operatoren ist eine ZER im DB-Cache, welche dem QM dann zur weiteren Bearbeitung zur Verfügung gestellt wird. Der TM notiert in den Transaktionsdeskriptoren die DB-Cache-Adressen dieser ZERs. Der Parameter <exp.size> gibt eine Schätzung für den von der betreffenden ZER benötigten DB-Cache-Speicherplatz an. Die Erstellung der verlangten ZER erfolgt letztendlich durch eine LI (vom CM beauftragt).

Spezifikation des Moduls UM:

Der Modul UM (Update Module) manipuliert Objekte vom Typ ZER, welche ihm vom QM bereitgestellt wurden.
Die Aufgaben des UM zerfallen in folgende Teilbereiche:

(i) Implementierung von Mengenupdates.
Der UM stellt Operatoren zur Implementierung von mengenorientierten Änderungsaufträgen zur Verfügung. Zur Illustration sei kurz die SQL-Benutzerschnittstelle für solche mengenorientierten Änderungsaufträge skizziert (vgl. [CHAM78]).
Betrachten wir dazu die Relationen
R1(r1,r2,r3)
R2(r2,r3)

```
Löschungen:     DELETE FROM R1
                WHERE  Queryprädikat              (*Query1*)
Einfügungen:    INSERT INTO R2
                       SELECT r3                  (*Query2*)
                       FROM   R1
                       WHERE  Queryprädikat
Modifikationen: UPDATE R2
                    SET    r2 = 1.1*r2            (*Query3*)
                    WHERE  Queryprädikat
```

Aufgrund der Semantik solcher SQL-Änderungsaufträge wollen wir festhalten: Bei der Auswahl der zu ändernden (DELETE,INSERT,UPDATE) Tupelwerte durch das Queryprädikat sind Projektionen zulässig, jedoch ist durch SQL garantiert, daß dadurch keine Mehrdeutigkeiten entstehen.

Für obige drei Beispiele heißt das:

- Bei Query1 braucht die vom QM bereitgestellte ZER nur die betreffenden Tupelids zu enthalten.
- Bei Query2 braucht die ZER nur Attributwerte von r3 enthalten; r3 muß jedoch Schlüssel von R2 sein (r2 wird mit Defaultwerten aufgefüllt).
- Bei Query3 braucht die ZER nur Attributwerte von r2 sowie die dazugehörigen Tupelids enthalten.

(ii) Realisierung der Einbettung von SQL in eine Wirtssprache durch Bereitstellung von Operatoren zur Implementierung des Cursorkonzepts (vgl. [CHAM78], [ASTR76]).
Hierbei werden Tupel einer vorliegenden ZER im 1-Tupel-Modus in den Benutzeradreßraum übertragen. Für diesen Vorgang sind gegebenenfalls Transformationen von Tupeldarstellungen in Systemform (internes Schema) in die Tupeldarstellung des Benutzers (konzeptuelles bzw. externes Schema bei Views) vorzunehmen.

(iii) Überwachung von Integritätsbedingungen, eventuell auch (inhaltsabhängiger) Zugriffsschutz.

(iv) Überwachung der automatischen Maintenance von Indexen.
Bei Tupel-Einfügungen/Löschungen/Updates hat der UM anhand der Transaktionsdeskriptoren zu überprüfen, welche Indexe eventuell geändert werden müssen. Entsprechende Index-Updateaufträge werden an den SAM weitergeleitet.

(v) Generierung von Tupelidentifikatoren bei Tupeleinfügungen (vgl. auch Kap.3.3.3).

(vi) Sammlung von Statistikinformation aufgrund von Updateaktivitäten (diese Statistikdaten werden zur Queryoptimierung benötigt, vgl. auch Kap 5.4.3.1)
Anmerkung: Die Sammlung von Statistiken über ausgewählte

Zugriffspfade könnte dem Tr-Comp übertragen werden.

UM-Operatoren:

- Cursoroperatoren (für Wirtssprachencinbettung):
  - INIT_CURS(ZERid) (*Cursor auf Anfang positionieren*)
  - GET_CURS(ZERid) (*Aktuelles Tupel lesen*)
  - PUT_CURS(ZERid) (*Tupel an aktuelle Position schreiben*)
  - CLOSE_CURS(ZERid) (*Cursor zerstören*)
- Mengenupdates:
  - UPD_ZER (ZERid,upd-spec) (*Modifikation aller Tupel von ZERid gemäß upd-spec*)
  - INS_ZER (ZERid,Relid) (*Einfügen aller Tupel von ZERid in Relation Relid*)
  - DEL_ZER (ZERid) (*Löschen aller Tupel von ZERid*)

Anmerkung: Für alle UM-Operatoren gelten dieselben Voraussetzungen wie für QM-Operatoren, d.h. sie sind DB-Cache-intern implementierbar.

Spezifikation des Moduls SAM:

Der SAM (System Access Module) realisiert die Ebene der logischen Zugriffspfade. Der SAM kreiert, manipuliert und zerstört Objekte vom Typ 'Index' sowie 'Systemkatalog'.[34] Unter einem Index ist dabei eine invertierte Liste für ein oder mehrere Attribute einer Relation zu verstehen.[35] Ein Eintrag in einer invertierten Liste hat die Form

[34] Der Grund, warum wir die scheinbar inhomogenen Aufgaben der Index- und Katalogverwaltung in einem Modul konzentrieren, liegt im Zugriffsverhalten auf das DB-Cache (Kap.3.2.3).

[35] Aufgrund der Diskussion über existierende DB-Systeme sind wir der Meinung, daß eine Mischung aus assoziativem Suchen und adressiertem Zugriff über Indexe die flexibelste Lösung ist (vgl. auch Kap. 5).

<attribute_value(s),list of tupelids>.[36] Die Systemkataloge (oft auch Data Dictionary genannt) enthalten die Meta-DB-Information und sind selbst als Relationen organisiert.

Folgende Aufgaben fallen dem SAM zu:

(i) Lesezugriffe auf Systemkataloge.
Die Information in den Systemkatalogen wird vom Tr-Comp bei der Queryübersetzung benötigt, ebenso vom Modul TM zum Aufbau von Transaktionsdeskriptoren zur ZER/Tupelidliste-Interpretation (für QM, UM) im Falle einer interpretativen Transaktionsausführung (bei interaktiven Transaktionen).

(ii) Implementierung von DDL-Updates.
SQL verfügt über eigene Anweisungen zur Manipulation von DB-Meta-Information, welche in den Systemkatalogen verwaltet wird. Ein solcher DDL-Update[37] äußert sich im Einfügen/Löschen/Ändern von Tupeln in Systemrelationen.

(iii) Erzeugung von Tupelidlisten durch Indexscans.
Restriktionen auf invertierten Attributen können ohne Tupelzugriff alleine mittels Indexscans ausgewertet werden; ein Parameter eines solchen Indexscan-Operators gibt dabei das aktuelle Restriktionsprädikat an. Die resultierende Tupelidliste wird dem QM zur weiteren Manipulation zur Verfügung gestellt.

(iv) Wartung der Indexe (aufgrund von Aufträgen des UM).

Anmerkung: Bei Einfügungen neuer Systemtupel oder Indexknoten vergibt der SAM analog zum UM neue Tupelidentifikatoren bzw. Knotenidentifikatoren (siehe Kap.3.3.3).

SAM-Operatoren:

- Indexoperatoren (*Operationen auf einem Index*)
  - IND_SCAN
  - IND_UPD

[36] Wir unterscheiden nicht zwischen Primär/Cluster-Index und Sekundär/Nichtcluster-Indexes (vgl. dazu Kap.3.2.4.2).

[37] Beispiele: . CREATE INDEX I13 ON EMP(SAL)
. GRANT READ, UPDATE(JOB) ON EMP TO User1

- Systemkatalog-Operatoren (*Operationen auf einem Systemkatalog*)
  - SYS_READ
  - SYS_UPD

Spezifikation des Moduls LM:

Der Modul LM ist eine zentrale Synchronisationsinstanz für (quasi-)parallele Transaktionen. Der LM kreiert, manipuliert und zerstört Objekte vom Typ 'Sperrinformation'.
Die Synchronisationsaufgaben des LM beziehen sich auf
- Concurrency-Control für Zugriffe auf DB-Objekte wie Relation und Tupel (evt. zweistufiges Sperrprotokoll, vgl. [GRAY77]),
- Concurrency-Control bei Indexzugriffen (vgl. z.B.[BAYE77])

Ferner verwaltet der LM transaktionsspezifisch solche Commit/Back-up-Information, welche vom Cache Module CM zur korrekten Erfüllung seiner DB-Cache-Verwaltungsoperationen benötigt wird.

LM-Operatoren:

- START_TRANS
- FINISH_TRANS
- ABORT_TRANS
- LOCK_DB_OBJ
- UNLOCK_DB_OBJ
- LOCK_IND_OBJ
- UNLOCK_IND_OBJ

Zur Erzielung einer hohen Parallelität sollten selbstverständlich moderne Synchronisationsverfahren wie in [BAHR80] realisiert werden.

3.2.3. Konstruktion eines Objekt-DB-Cache.

Eine wichtige Entwurfsvorgabe für die neue Architektur war die Forderung einer Integration des DB-Cache/SAFE-Verfahrens ([ELHA82]). Das DB-Cache ermöglicht eine Integration der Transaktions-Sicherung in die Cache-Verwaltung und eine sorgfältige Abstimmung von Transaktions-Sicherung mit Transaktions-Synchronisation, wodurch eine erhebliche Leistungssteigerung im Normalbetrieb zu erwarten ist und ein extrem schneller

Wiederanlauf nach Systemzusammenbruch garantiert wird.
Eine wesentliche Einschränkung in [ELHA82] ist die Forderung, daß die Synchronisation sowie die Pufferverwaltung im DB-Cache auf der Basis (logischer) Seiten vorausgesetzt wird.
Die für einen seitenorientierten DB-Cache charakteristischen Eigenschaften eines Transaktionsablaufes einer Transaktion T sind wie folgt: Benötigte Seiten werden von den DB-Platten geholt, falls sie nicht bereits im DB-Cache stehen. Seiten, die von T geändert werden, bleiben im DB-Cache und werden nicht vor Ende der Transaktion (EOT) auf die DB-Platten verdrängt (DB bleibt sauber). Seiten, die von T gelesen werden, werden nicht vor EOT aus dem DB-Cache verdrängt (um Platz für benötigte Seiten anderer Transaktionen zu schaffen). Zur Beendigung von T werden die geänderten Seiten sequentiell in den SAFE geschrieben. Geänderte Seiten können dann später vom DB-Cache auf die DB-Platten durchgeschrieben werden oder ohne Durchschreiben erneut geändert werden.
Ziel dieses Kapitels ist es nun nachzuweisen, daß das DB-Cache/SAFE-Konzept auch in unserer Systemumgebung effizient, d.h. ohne wesentliche Änderungen der in [ELHA82] für ein Seiten-Cache entwickelten Algorithmen, verwendet werden kann.

Funktionale Anforderungen an das DB-Cache in der neuen Architektur (N.A.)

Die Schnittstelle CI zum Cache-System bildet die virtuelle Maschine auf der Ebene der Speicherstrukturen (level 2). Das DB-Cache (als Teil des Host-Arbeitsspeichers) ist somit das konkrete Speichermedium, auf dem die Datenstrukturen des Relationalen Zugriffssystems abgebildet werden.

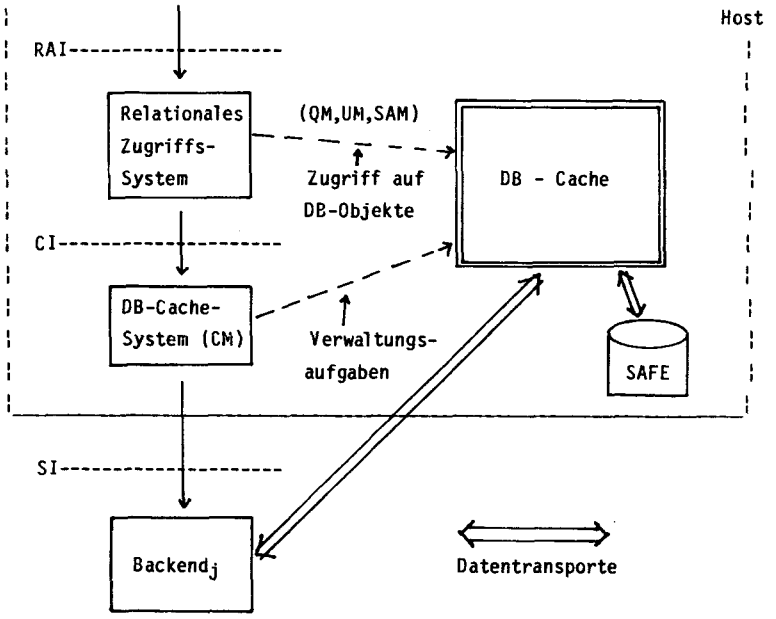


Abb.3.3: Stellung des DB-Cache in der N.A.

Wie aus Abb.3.3 ersichtlich ist, dient das DB-Cache dem Relationalen Zugriffssystem als integrierter DBMS-Arbeitsspeicher zur Transaktionsabarbeitung.
Der Cache-Modul CM ist somit verantwortlich für

- DB-Cache-Speicherplatzverwaltung: Bereitstellung benötigten Speichers,
- Commit & Recovery mittels SAFE,
- Durchschreiben gültiger Objekte aus dem DB-Cache auf die DB-Platten (via LIs),
- Beauftragung von LIs zur Materialisierung von DB-Objekten im DB-Cache.

Dabei lassen sich die funktionalen und leistungsmäßigen Anforderungen an das

DB-Cache wie folgt detaillieren:

(V1) Verwendung zur Retrieval-Query-Auswertung

Eine effiziente Implementierung der mengenorientierten RAI-Schnittstelle verlangt, das benötigte Operanden zur schnellen Verarbeitung sich vollständig im DB-Cache befinden.

Das DB-Cache dient somit als

(V1a) Speicher für ZERs, welche von einem Backend geliefert werden.

(V1b) Ergebnisspeicher bei Ausdrucksauswertung mittels der RelA-Operatoren des QM.

(V1c) Als Indexknoten-Speicher zur Realisierung eines schnellen Zugriffs auf invertierte Listen.

(V1d) 'Schmierzettel' für Ablage (durch SAM) und Manipulation (durch QM) von Tupelidlisten aufgrund von Restriktionsauswertungen mittels Indexen.

(V2) Verwendung zur Vorbereitung von DB-Updates

Hierfür soll das DB-Cache verwendet werden als Speichermedium für die Vorbereitung von

(V2a) Updates einer ZER durch den UM, welche vorher vom QM bereitgestellt wurde,

(V2b) Indexknoten-Updates durch den SAM,

(V2c) Systemkatalog-Updates (d.h. Einfügen/Löschen/Ändern eines Tupels in eine Systemrelation) durch den SAM.

(V3) Verwendung zum Lesezugriff auf Systemkatalog-Information

Lesezugriffe auf Kataloginformation werden von SAM ausgeführt. Dabei werden einzelne Tupel aus den Systemkatalogen gelesen, welche zum Aufbau von Transaktionsdeskriptoren oder zur Query-Optimierung gebraucht werden.

Aus diesem Anforderungskatalog an das DB-Cache ergeben sich zusammengefaßt folgende erste Konsequenzen für die virtuelle Maschine CI und DB-Cache:

(1) Mengenverarbeitung:

- ZERs und Tupelidlisten müssen effizient implementierbar sein für Lesezwecke (durch QM)
- ZERs müssen effizient implementierbar sein für Update-Zwecke (durch

QM).

(2) Einzelverarbeitung:

- Indexknoten- und Systemtupelzugriffe (sowohl Lesen als auch Ändern) müssen effizient implementierbar sein (durch SAM).

Als nächster Aspekt soll nun der Einfluß unserer spezifischen Hardware/Software-Architektur auf das DB-Cache aufgeklärt werden.

Grundtatsache:

Als Ergebnis von Filter-Retrievalaufträgen wird nur relevante Information in das DB-Cache transportiert.

=====>

(a) Die (bisherige) Seitenstruktur im DB-Cache geht verloren.

(b) Die ursprüngliche Tupelstruktur geht verloren (bei Wegfiltern von Attributen durch eine LI).

Somit müssen die in [ELHA82] entwickelten Konzepte eines seitenorientierten Cache an die neue Systemumgebung angepaßt werden.

Konsequenz:

Die Konstruktion eines Objekt-DB-Cache ist erforderlich.

Ein Objekt-DB-Cache soll als Verwaltungseinheit beliebige Objekte enthalten können, und nicht etwa mittels Simulation auf einem Seitencache realisiert werden. Als nächster Komplex sind nun das Lokalitätsverhalten sowie Verwaltungsprobleme eines Objekt-DB-Caches, welcher den genannten heterogenen Anforderungen einer effizienten Mengen- als auch Einzelverarbeitung gerecht wird, zu diskutieren.
Einleitend dazu geben wir zuerst einige Lokalitätsbeobachtungen auf seitenorganisierten DB-Puffern in herkömmlichen DBMS (ohne Mengenverarbeitung durch das Zugriffssystem) an.

Das Lokalitätsprinzip in seitenorganisierten DB-Puffern basiert auf zwei Effekten:

(1) Aktuell benutzte Information wird wahrscheinlich in der nahen Zukunft

wieder benutzt ('locality by time').

(2) Information, welche in derselben Seite liegt wie aktuell benutzte Information, wird wahrscheinlich in der nahen Zukunft benutzt ('locality by prefetching').

Dabei ist noch zwischen Intertransaktions-Lokalität und Intratransaktions-Lokalität zu differenzieren.

Vorgenommene Lokalitätsbeobachtungen durch Analysen von Seitenreferenzstrings von DB-Systemen mit satzweiser (tupelweiser) Verarbeitung zeigten in vielen Fällen folgendes Verhalten unter parallelen Transaktionen (vgl. z.B. [RODR76]):

- Lokalität auf Benutzerdaten ist relativ gering.
- Lokalität auf Systemdaten ist wesentlich stärker ausgeprägt.

Ausgehend von diesen empirischen Erkenntnissen soll nun das zu erwartende Lokalitätsverhalten eines Objekt-DB-Cache in der N.A. prognostiziert werden.

Erwartetes Lokalitätsverhalten eines Objekt-DB-Cache.

(E1) Auswirkungen auf die Lokalität bei Benutzerrelationen.
Das Lokalitätsverhalten wird durch zwei Faktoren bestimmt:

(a) Bei LI-Filterung wird nur relevante Information (die ZERs) für die weitere Ausdrucksauswertung in das DB-Cache transportiert; diese ZERs werden aufgrund der mengenorientierten Filterung im Ganzen im DB-Cache zur Verfügung gestellt.

=====>

- Der Begriff der 'locality by prefetching' im Bezug auf Intratransaktions-Lokalität ist nicht mehr relevant, da sich für QM-Operatoren benötigte Operanden vollständig im DB-Cache befinden.
- Da für jede Transaktion nur relevante Information in das DB-Cache transportiert wird, verringert sich die 'locality by prefetching' im Bezug auf Intertransaktions-Lokalität, da i.a. nur Teiltupel im DB-Cache stehen.

(b) Die DB-Cache/SAFE-Eigenschaft, daß die von einer Transaktion benötigten ZERs nicht vor EOT aus dem DB-Cache verdrängt werden, bringt folgende Auswirkung mit sich: Die Ausnutzung von Intratransaktions-Lokalität auf ZER-Basis, falls ein RelA-Operatorbaum identische Teilausdrücke aufweist, ist garantiert (dies entspricht einer 'locality by time').

Insgesamt leiten wir aus diesen Überlegungen folgende Prognose ab: Intertransaktions-Lokalität auf gefilterten Benutzerrelationen kann i.a. als vernachlässigbar angesehen werden.

(E2) Für Indexknoten (z.B. B-Baum-Wurzel) ist Lokalität ein wichtiger Faktor, der auf keinen Fall zerstört werden soll. Somit darf keine LI-Filterung auf Indexknoten angewendet werden. Es sind volle Indexknoten im DB-Cache dem SAM zur Verfügung zu stellen. Dabei ist gemeinsamer Lesezugriff für parallele Transaktionen zu gewährleisten.

(E3) Insbesondere für relationale DB-Systeme stellen die Systemkataloge Objekte mit sehr hoher Zugriffsfrequenz dar, d.h. die zu erwartende Lokalität ist hier am größten, und darf deshalb durch LI-Filterung nicht zerstört werden.

Im DB-Cache müssen deshalb volle Systemtupel zur Verfügung gestellt werden mit der Möglichkeit gemeinsamen Lesezugriffs. Aufgrund der geringen Größe von Systemrelationen ist als Optimierung anstelle eines tupelweisen Transportes von den DB-Platten in das DB-Cache ein einmaliges Laden (gezieltes Objekt-Prefetching) aller Systemrelationen in das DB-Cache bei DB-Sessionbeginn denkbar und wünschenswert.

Konstruktionsprinzipien für ein effizientes Objekt-DB-Cache.

Die geschilderten Lokalitätsüberlegungen geben Anlaß zu folgender funktionalen Aufteilung des DB-Cache:

Jede Transaktion läßt sich benötigte ZERs, welche von permanenten Benutzerrelationen abzuleiten sind, von einer LI über Filter-Retrievalaufträge besorgen (vgl. auch 3.2.4.1). Das bedeutet aber, daß wir keinen

Versuch unternehmen, die Existenz von ZERs anderer Transaktionen im DB-Cache auszunützen.[38] Somit können ZERs als privat für gewisse Transaktionen betrachtet werden.
In ähnlicher Weise werden wir Tupelidlisten behandeln. Sie sind privat für gewisse Transaktionen und werden jeweils neu vom SAM erzeugt ohne Ausnützung evt. schon vorhandener anderer Tupelidlisten.
Für Indexknoten und Systemtupel hingegen wird Lokalität ausgenützt.
Insgesamt erhalten wir also folgende Unterteilung des DB-Cache:

- Private Bereiche zur Mengenverarbeitung : Private-C
- Gemeinsame Bereiche zur Einzelverarbeitung: Public-C

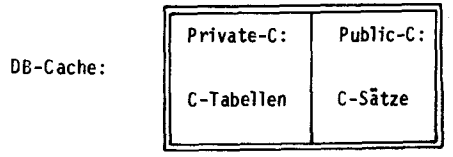


Abb.3.4: Funktionale Aufteilung des DB-Cache.

Spezifikation des Objekt-DB-Cache.

(DB-C1) Logische disjunkte Aufteilung:

$$\text{DB-Cache} = \text{Private-C} \cup \text{Public-C}$$

(DB-C2) Spezifikation von Private-C.
Der Private-C ist partioniert unter parallelen Transaktionen. Speicherobjekte in Private-C sind die sogenannten C-Tabellen (siehe Abb. 3.4). Eine solche C-Tabelle besteht einfach aus einer Folge von Bytes (z.B. realisiert als Array), welche durch die Cache-Verwaltung CM nicht interpretiert wird. Der Verwendungszweck von C-Tabellen ist die Implementierung von

38) Man wird i.a. auch nicht das Gewünschte finden.

- ZERs
- Tupelidlisten

Operationen auf C-Tabellen: sequentieller und/oder random Zugriff auf einzelne Bytes.

Die Interpretation von C-Tabellen-Inhalten erfolgt oberhalb der Cache-Schnittstelle CI durch die Moduln QM und UM.

(DB-C3) Spezifikation von Public-C.

Der Public-C ist gemeinsam benutzbar für parallele Transaktionen. Speicherobjekte sind die sogenannten C-Sätze, welche ebenfalls eine durch den CM uninterpretierte Bytefolge darstellen.

Der Verwendungszweck von C-Sätzen besteht in der Implementierung von

- vollen Indexknoten
- vollen Systemtupeln

Die Interpretation von C-Sätzen erfolgt oberhalb von CI durch den SAM.

Eigenschaften dieser Organisation.

(1) Die Bedürfnisse einer effizienten mengenmäßigen Implementierung der QM- und UM-Mengenoperatoren sind durch die Private-C-Organisation erfüllt.

(2) Die Aufteilung in Private-C und Public-C steht im Einklang mit der erwarteten Lokalität für DB-Cache-Zugriffe durch das Relationale Zugriffssystem.[39]

(3) Die Cache-Speicherverwaltung ist durch den Modul CM einfach und effizient möglich.

Der CM benötigt dazu den sogenannten Cache-Katalog C-cat, in dem die Abbildungen von C-Tabellen und C-Sätzen auf DB-Cache-Adressen festgehalten sind.[40] Dazu benötigt man eindeutige Identifikatoren für die

[39] Für Spezialanwendungen mit hoher zeitlicher Lokalität -wie etwa bei Flugbuchungssystemen: die Nachfragehäufigkeit zu den Flügen steigt mit nahendem Abflugtermin- ist folgende, nachträglich einbaubare Zusatzoptimierung möglich: Jede LI wird mit einem seitenorganisierten I/O-Cache ausgerüstet. Damit können DB-Plattenzugriffe für hochaktuelle Benutzertupel vermieden werden.

[40] Weitere Zustandsinformationen für Commit/Backup/Recovery-Zwecke werden hier nicht eigens erwähnt (vgl. dazu [ELHA82]).

Cache-Objekte, welche wir wie folgt bezeichnen:

- C-id für eine C-Tabelle
- lid für einen C-Satz[41)]

Der C-cat enthält dann folgende Information, falls alle Cache-Objekte physisch-sequentiell abgelegt sind:

(a) Private-C-Abbildung: {C-id} ⟶ {C_addr,byte_length}

(b) Public-C-Abbildung: {lid} ⟶ {C_addr,byte_length}

(c) Freispeicherliste FS für freien DB-Cache-Platz.

Die Private-C-Abbildung ist sehr kurz, da nur komplette C-Tabellen, und nicht (Teil-)Tupel einzeln verwaltet werden. Der Speicherbedarf für die Public-C-Abbildung ist auch nicht übermäßig, da die Kardinalität von Systemrelationen relativ gering ist und Indexknoten gewöhnlich Seitengröße aufweisen. Wesentlich ist ferner, daß die angegebene Private-C-Abbildung eine sehr effiziente Implementierung der QM-Operatoren gestattet. Insgesamt betrachtet, ist das Identifizierungsproblem eines DB-Cache mit beliebig großen und beliebig strukturierten Objekten zufriedenstellend gelöst. Ein kleiner unvermeidbarer Nachteil eines Objekt-DB-Cache bleibt natürlich erhalten, nämlich das Fragmentierungsproblem von Cache-Speicherplatz. Zu diesem Komplex existieren jedoch bekannte Standardlösungen, so daß wir diesen Aspekt nicht weiter verfolgen brauchen.

(4) Bei der Reservierung einer C-Tabelle ist es wichtig, daß dem CM eine zuverlässige Schätzung der Tabellengröße mitgeteilt wird. Dies gilt insbesondere bei der Füllung einer C-Tabelle mit gefilterten Daten durch eine LI. Da Kardinalitäts-Abschätzungen für ZERs bei der Query-Optimierung sowieso vorgenommen werden müssen (vgl. Kap.5.4.3.1), stehen diese somit 'gratis' zur Verfügung.[42)]

Filteranomalien beim Cache/SAFE-Konzept.

---

41) Eine ausgiebige Diskussion der Eigenschaften von lids erfolgt in Kap.3.3.2.1.

42) Eine alternative Lösungsmöglichkeit für dieses Problem besteht in einer Kontingentierung von Teilen (Kacheln) der Freispeicherliste FS an einzelne LIs. Der Philosophie von strenger funktionaler Aufteilung folgend, ziehen wir jedoch obige Variante vor.

Bei der Ausführung von LI-Filteroperationen muß eine spezifische Cache/SAFE-Eigenschaft sorgfältig berücksichtigt werden:
Neue gültige Tupelwerte werden nicht vor oder bei EOT auf die DB-Platten geschrieben, sondern erst später bei Bedarf (z.B. DB-Cache-Platzmangel) eigenständig durch den CM. Somit ist auf den DB-Platten nicht notwendigerweise immer die neueste gültige Version einer Relation gespeichert.
Falls Filter-Aufträge 'bedenkenlos' an die LI geschickt werden, können deshalb folgende Filteranomalien auftreten:

(1) Fehlerhafte Filterung eines Tupels:
Ein Tupel wird als Treffer durch die LI erkannt, obwohl es inzwischen gelöscht wurde oder bereits eine neue gültige Tupelversion im DB-Cache existiert, welche das vorliegende Filterprädikat nicht mehr erfüllt.

(2) Unterlassene Filterung eines Tupels:
Ein Tupel wird nicht als Treffer durch die LI erkannt, obwohl bereits eine neue gültige Tupelversion im DB-Cache existiert, welche das vorliegende Filterprädikat nunmehr erfüllt, oder weil es neu eingefügt, aber noch nicht auf die DB-Platten durchgeschrieben wurde.

Ein möglicher Weg aus diesem Dilemma bestünde darin, nach erfolgter LI-Filterung im DB-Cache alle Treffertupel zu überprüfen, ob eine ungültige Filterung vorliegt und gegebenfalls diese nachträglich zu eliminieren. Außerdem müßte noch auf unterlassene Filterung getestet werden. Durch dieses aufwendige Verfahren würde jedoch ein Teil der Effizienzvorteile einer schnellen LI-Filterung in Frage gestellt werden, auch würde die Systemkomplexität dadurch enorm erhöht werden.

Lösungsvorschlag:

Wir betrachten folgendes Scenario:
Ein Filterauftrag für Relation R wird an den CM erteilt (vom TM als Folge der CM-bezogenen Operatoren COMPLETE_RELSCAN oder SELECTIVE_RELSCAN). Dann geht der CM wie folgt vor:

if <nicht ausdrücklich die alte Version von R verlangt wird[43]) >
and

[43]) Das ist möglich bei einer Transaktions-Synchronisation nach dem Schattenkonzept ([BAHR80]).

```
        <es befinden sich neue gültige Tupelversionen von R im DB-Cache,
         welche noch nicht auf DB-Platte durchgeschrieben sind>
    then     (*forciertes Durchschreiben*)
         <CM erteilt Durchschreibe-Auftrag für diese Tupel an LI>;44)
         <Nach erfolgter Bestätigung durch die LI leitet CM den
          Filterauftrag an die LI weiter>
    else
         <CM leitet den Filterauftrag sofort an die LI weiter>
fi;
```

Beurteilung dieses Verfahrens:

Das forcierte Durchschreiben von geänderten gültigen Objekten aus dem DB-Cache auf die DB-Platten bringt offensichtlich nur dann keinen Effizienzverlust mit sich, wenn die Lokalität auf den durchzuschreibenden Objekten relativ gering ist. Nach unserer Einschätzung heißt das, daß LI-Filteroperationen auf Systemrelationen nicht zulässig sein dürfen. Sie sind auch nicht notwendig in unserer Architektur, da wir annehmen, daß sich die Systemrelationen stets vollständig im DB-Cache befinden.

Forciertes Durchschreiben von Benutzertupeln hingegen bedeutet keinen Effizienzverlust gegenüber einem seitenorientierten DB-Cache (wo dies nicht notwendig ist), da die erwartete Lokalität hierbei sehr gering ist und somit Seiten mit solchen Tupeln auch bei einem Seiten-Cache irgendwann durchgeschrieben werden, ohne vorher wiederbenutzt worden zu sein.

Diese Notwendigkeit des forcierten Durchschreibens von Benutzertupeln wird durch die bisher getroffene Aufteilung des Objekt-DB-Cache in Private-C und Public-C in wirksamer Weise unterstützt.

Verfeinerte funktionale Aufteilung des Objekt-DB-Cache.

Wir nehmen eine weitere Untergliederung des Objekt-DB-Cache hinsichtlich des Verwendungszwecks vor.

(1) Logische Aufteilung des Private-C:

44) Diese Initiative muß vom CM ausgehen (und nicht etwa von der betreffenden LI), da, wie in Kap.3.3.1 diskutiert wird, die LIs nicht in das Transaktionskonzept eingebunden sind.

Private-C = Workspace-C ∪ Update-C ∪ Committed-C

(2) Physische Aufteilung des Public-C:

Public-C = Read-C ∪ Copy-C

Die diesen Partitionen zugrundeliegende Verwendungsabsicht ist dabei wie folgt:

ad (1): - Workspace-C ist partitioniert unter den parallelen Transaktionen, er wird verwendet zur mengenmäßigen Retrieval-Query-Auswertung.
C-Tabellen enthalten: . ZERs
. Tupelid-Listen

- Update-C ist auch partioniert unter parallelen Transaktionen, er wird verwendet zur Vorbereitung von Änderungen von Benutzerrelationen.
C-Tabellen werden aus dem Workspace-C geliefert (nur logischer Übergang, keine Kopiervorgänge),
C-Tabellen enthalten: . ZERs

- Committed-C ist unzugänglich für Transaktionen, er wird vom CM forciert bei entsprechenden LI-Filteraufträgen auf die DB-Platten durchgeschrieben (über SI-Schreibaufträge). Beim erfolgreichen Commit von ZERs im Update-C erfolgt ein logischer Übergang in den Committed-C.

ad (2): - Read-C ist gemeinsam benutzbar für Lesezugriffe paralleler Transaktionen.
C-Sätze enthalten: . volle Systemtupel
. volle Indexknoten

- Copy-C ist partitioniert unter parallelen Transaktionen, er dient zur Vorbereitung von Änderungen von Systemtupeln oder Indexknoten. Dabei wird eine physische Kopie aus dem Read-C gezogen und im Copy-C zur Verfügung gestellt.

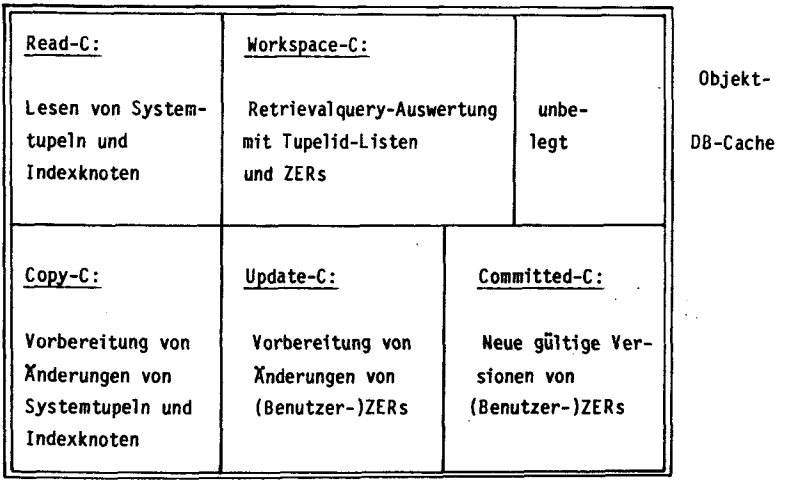


Abb. 3.5: Das Objekt-DB-Cache als integrierter DBMS-Arbeitsspeicher für Mengen- und Einzelverarbeitung.

Als letzter Punkt verbleibt nun noch eine explizite Angabe der externen CM-Operatoren (welche der CI-Schnittstelle entsprechen).

Spezifikation der CI-Schnittstelle:

(CI1) Operatoren auf Public-C

Diese Gruppe von Operatoren entspricht der CM-Schnittstelle eines Seitencaches, jedoch mit Aufträgen für C-Sätze anstelle von Seiten. Als Antwort auf Aufträge zur Verfügungstellung von C-Sätzen wird die DB-Cacheadresse zurückgegeben (an den SAM).

- gR_REC (<lid>,<byte length>) RETURNS (<C_addr>)
- gW_REC (<lid>,<byte length>) RETURNS (<C_addr>)
- noR_REC (<lid>)
- noW_REC (<lid>)

Anmerkungen:

- gR_REC und gW_REC werden vom SAM aufgerufen, noR_REC und noW_REC vom LM.
- Die Satzlänge ist dem SAM bekannt, da gR_REC bzw. gW_REC nur für Systemdaten mit bekannter Länge verwendet werden.
- Die Semantik von noR_REC bzw. noW_REC ist identisch zum seitenorganisierten DB-Cache: das angesprochene Objekt wird von keiner Transaktion zum Lesen im Read-C bzw. zum Ändern im Copy-C benötigt und kann somit vom CM in die Liste der ersetzbaren C-Objekte aufgenommen werden.

(CI2) Operatoren auf Private-C

Aufgrund der 'Schmierzettel'-Verwendung von Workspace-C durch den QM sind zusätzlich zum Seitencache explizite Reservierungsaufträge für benötigte C-Tabellen notwendig.[45] Dies kann insofern losgelöst von Sperren gesehen werden, als ein Weiterverarbeiten von DB-Information stattfindet, welche vorher schon entsprechend gesperrt wurde. Da außerdem C-Tabellen exklusiv für Transaktionen sind, kann eine explizite Freigabe für C-Tabellen aus Workspace-C vor EOT erfolgen. Diese Möglichkeit ist besonders für unsere mengenmäßige Queryauswertungsstrategie unbedingt wünschenswert.

- RESERVE_TAB (<C-id>,{<Relid>},<exp byte length>)
  RETURNS(<C_addr>,<byte length>)
- RELEASE_TAB (<C-id>)

Diese beiden Operatoren werden vom TM aufgerufen, welcher die eindeutigen C-ids vergibt und zusammen mit den Cacheadressen in die jeweiligen Transaktionsdeskriptoren einträgt. Der Parameter <Relid> ist nur für solche C-Tabellen anzugeben, welche zur Vorbereitung von ZER-Updates verwendet werden sollen.
Des weiteren benötigen wir einen Operator, welcher die Füllung einer C-Tabelle mittels LI-Filterung bewerkstelligt; dieser Operator wird

[45] Diese bleiben dann auch im DB-Cache bei EOT oder bis zur expliziten Freigabe!

vom TN aufgerufen.

. gTAB (<C_addr,byte length>,<Relid>,<op_spec>)

Die mitgelieferte Cacheadresse identifiziert die (bereits vorher reservierte C-Tabelle), in die die betreffende LI die gefilterten Daten ablegen soll. Der konkrete LI-Auftrag (vgl. auch SI-Schnittstelle, 3.2.4.1) ist im Parameter <op_spec> enthalten. Bei der Implementierung dieses Operators ist der Lösungsvorschlag zur Vermeidung von Filteranomalien zu berücksichtigen.

(CI3) Commit-Operator auf Public-C und Private-C

Dieser Operator wird vom LM aufgrund einer erfolgreichen Prüfung eines FINISH_TRANS-Wunsches aufgerufen.

. COMMIT (<C-id_specs>,<lids>)

Der Parameter <C-id_specs> liefert folgende Information:

- Liste der C-Tabellen für das Commit.
- Aus dem Transaktionsdeskriptor entnommene Beschreibung, welche Attribute (in welcher Reihenfolge) vorhanden sind (für alle beteiligten C-Tabellen aus Update-C).

Die Ausführung dieses Auftrags durch den CM bewirkt das Logging der angegebenen Objekte auf den SAFE. Für Systemtupel oder Indexknoten wird dabei der lid mit vermerkt, bei ZERs die entsprechende (Teil-)Tupelspezifikation. Daß diese objekt-orientierte SAFE-Information auch tatsächlich ausreicht, um das Transaktionskonzept im Fehlerfall zu garantieren, wird im nächsten Abschnitt diskutiert.

Abschließende Bewertung des Objekt-DB-Cache:

Die angegebene Konstruktion des Objekt-DB-Cache stellt eine effiziente Verallgemeinerung des seitenorganisierten DB-Cache/SAFE-Verfahrens dar, und zwar aus folgenden Gründen:

(a) Das Problem "Was ist eine vernünftige Verwaltungseinheit für den CM?"

ist zufriedenstellend gelöst: C-Tabellen für Mengenverarbeitung, C-Sätze für Einzelverarbeitung.

(b) Die Aufteilung des DB-Cache in Private-C und Public-C steht im Einklang mit der prognostizierten Lokalität.

(c) Die CI-Operatoren lassen sich, wie beim Seiten-DB-Cache, in sinnvoller Weise an die Sperrebene koppeln:

- Die Semantik der CI-Operatoren auf Public-C ist völlig analog zu den entsprechenden Operatoren beim Seiten-DB-Cache (vgl.[ELHA82]).
- Die CI-Operatoren auf Private-C sind eine homogene Erweiterung des DB-Cache/SAFE-Verfahrens für die Erfordernisse der Mengenverarbeitung.

(d) Die schnellen Restart-Zeiten des seitenorganisierten DB-Cache/-SAFE-Verfahrens sind auch beim Objekt-DB-Cache garantiert:
Beim Logging auf den SAFE muß für ZERs zwar etwas Verwaltungsinformation geschrieben werden; die auf den SAFE zu sichernden ZERs enthalten aber aufgrund der Datenfilterung nur relevante Information. Es ist deshalb bei unserer Architektur zu erwarten, daß die SAFE-Länge kürzer ist als beim Seitencache und somit die Restart-Zeiten im Fehlerfalle mindestens genau so schnell sind.

3.2.4. Spezifikation des Intelligenten DB-Speichersystems.

Zur Vergegenwärtigung der Systemarchitektur illustrieren wir nochmals einen Ausschnitt aus der Gesamt-DBMS-Architektur (Abb.3.2).

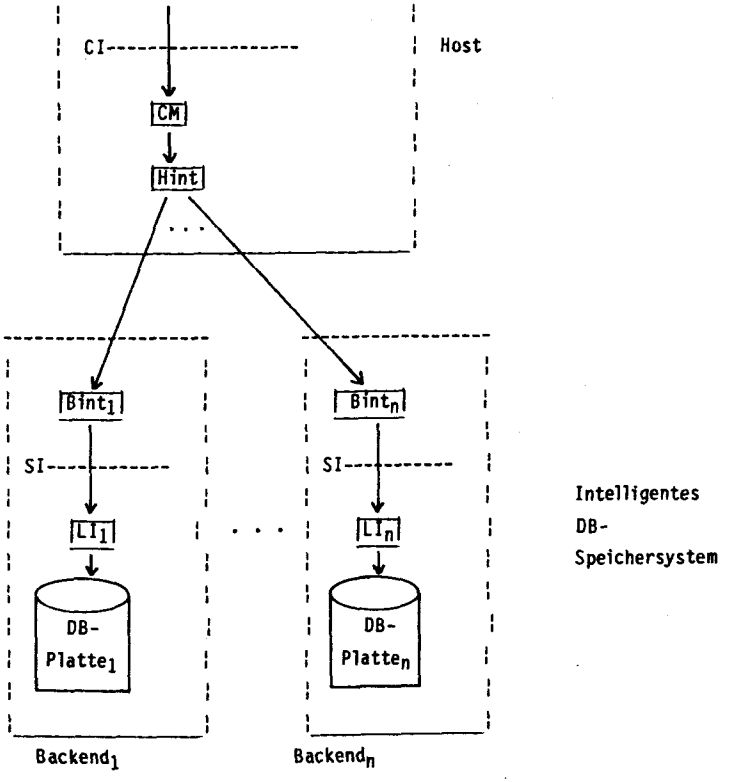


Abb.3.6: Ausschnitt aus der Modulhierarchie.

Wir wollen noch einmal darauf hinweisen, daß das Backendinterface $Bint_j$ identisch ist für alle j=1,....,n. Das gleiche gilt für die Lokale Intelligenz LI_j. Dementsprechend besitzen natürlich auch alle $Backend_j$ dieselbe Schnittstelle SI.

Kommunikation zwischen Host und Backends:

Jeder $Backend_j$ soll vollkommen asynchron zum Host arbeiten können. Die Kommunikation wird dabei durch Hint und $Bint_j$ realisiert. Das Hostinterface Hint ist ein Bestandteil oder eine Erweiterung des Host-Betriebssystems. Die Kommunikation kann sowohl über eine I/O-Schnittstelle mit intelligenten Kanälen (Kanalprogramm) als auch über einen Bus mit Botschaften-Mechanismen laufen. Diese Feststellung soll insbesondere zum Ausdruck bringen, daß kein gemeinsamer Speicher für schnelle Kommunikationszwecke verwendet zu werden braucht (und soll). Das läßt sich damit begründen, daß die SI-Schnittstelle zu den LIs eine hohe, teilweise mengenorientierte Retrieval/Store-Schnittstelle bildet und somit zu erwarten ist, daß der Kommunikations-Overhead zwischen einem Backend und Host relativ gering ist im Vergleich zur Bearbeitungszeit für SI-Operatoren durch eine LI.

3.2.4.1. Beschreibung der Schnittstellen-Operatoren und Aspekte ihrer Implementierung.

Eine LI bildet zusammen mit einem DB-Plattenstapel eine Backend-Maschine mit folgenden Aufgaben:

(LI1) Komplexe Filter-Retrievaloperationen für Mengenverarbeitung.

(LI2) Einfache, nicht filternde Retrievaloperationen für Einzelverarbeitung.

(LI3) Sichere Durchschreibe-Operationen von gültigen DB-Cache-Objekten auf die DB-Platten.

(LI4) Autonome DB-Platten-Verwaltung, Media-Failure Recovery.

Die DB-Platten sind das sichere Langzeitspeichermedium für permanente

- Benutzerrelationen
- Systemrelationen
- Indexe

Die LI-Operatoren (LI1), (LI2) und (LI3) realisieren die Speicherschnittstelle SI.
Die von uns vorgenommene funktionale Aufteilung des relationalen DBMS ist charakterisiert durch die Auslagerung von linearen Operationen auf Tupelmengen in das Intelligente DB-Speichersystem.

Definition:
Als lineare Operation bezeichnen wir eine Operation Op auf einer Menge M von Tupeln mit folgender Eigenschaft: Zur Berechnung des Ergebnisses von Op muß jedes Tupel von M höchstens einmal betrachtet werden.
Für eine mengenmäßige Queryauswertung über RelA-Operatoren kommen folgende lineare Operationen in Frage:

(1) lineare Filteroperationen:
   (1.1) Restriktion einer Relation, welche tupellokal ausgewertet werden kann.
   (1.1) 'Wegschneiden' von Attributwerten von Tupeln einer Relation ("partielle" Projektion, d.h. ohne Duplikatelimination).

(2) lineare Aggregatfunktionen:
   z.B. MIN, MAX, SUM, AVG, COUNT

Eine Kombination solcher linearer Operationen ergibt wiederum eine lineare Operation, so daß also Sequenzen von Operationen (1.1), (1.2) und (2) für ein und dieselbe Relation in einem einzigen LI-Auftrag zusammengefaßt werden können. Eine Kombination von Restriktions-Operationen bezeichnen wir als LI-Filterformel.
Eine LI-Filterformel F_R ist wie folgt definiert:

$F_R = \bigwedge_{i=1}^{l} C_i$, wobei jedes C_i von der Form

$C_i = \bigvee_{j=1}^{m_j} C_{ij}$ ist (konjunktive Normalform).

Jeder Term C_{ij} bezieht sich dabei auf dieselbe Relation R und besitzt die elementare Form

C_{ij} = <R.attr1> <comp_op> <const_val> oder

C_{ij} = <R.attr1> <comp_op> <R.attr2>

wobei:. <R.attr1> und <R.attr2> sind Attribute von R

. <comp_op> e {=, ≠, >, ≥, <, ≤}

. <const_val> ist ein Element des Wertebereichs von <R.attr1>

Anmerkung: In SQL ist es möglich, tupellokale Restriktionen mit Hilfe von arithmetischen Ausdrücken zu formulieren, z.B.:

```
SELECT r1, r2
FROM   R
WHERE  r1-r2 < 50
```

Solche (etwas aufwendigere) tupellokal auswertbare Restriktionen sollten natürlich auch zum Filterrepertoire jeder LI gehören. Wir verzichten jedoch der Einfachheit halber auf eine dementsprechende Erweiterung der Definition für eine LI-Filterformel.

Spezifikation der SI-Schnittstelle:

- EXH_FILTER (<Relid>,<LI-filterformula>,<aggr_fcts>,<atttr_list>, <tupel_layout>,<C-table_addr>,<id_flag>)
- SEL_FILTER(<Relid>,<tupelidlist>,<LI-filterformula>,<aggr_fcts>, <attr_list>,<tupel_layout>,<C-table_addr>,<id_flag>)
- GET_REC (<lid>,<C-record_addr>)
- PUT_TAB (<Relid>,<C-table>,<C-table_spec>,<id_flag>)
- PUT_RECS (<lid_list>,<C-records>,<id_flag>)
- CREATE_SEG (<Relid/Indid>,<space_spec>)
- DROP_SEG (<Relid/Indid>)

Im Vergleich zu anderen DB-Systemen lassen sich obige drei Leseoperationen EXH_FILTER, SEL_FILTER und GET_REC in etwa wie folgt einordnen:

(a) EXH_FILTER ist charakteristisch für assoziative Platten (wie z.B. in DBC): es durchsucht sequentiell eine komplette Relation und filtert die gesuchten Tupel heraus; als Ergebnis wird eine C-Tabelle im DB-Cache gefüllt.

(b) SEL_FILTER ist eine Mischung aus adressiertem und assoziativem Zugriff: es arbeitet auf einer Teilmenge der Blöcke, in denen eine Relation gespeichert ist, und filtert die gesuchten Tupel aus diesen Blöcken heraus; als Ergebnis wird ebenfalls eine C-Tabelle im DB-Cache gefüllt.

(c) GET_REC entspricht der Standard-Leseoperation in konventionellen DB-Systemen. (Hier jedoch auf C-Satz-Basis anstelle von einer Seite). Als Ergebnis wird ein C-Satz im DB-Cache gefüllt.

PUT_TAB und PUT_RECS sind mengenorientierte Durchschreibe-Operatoren für neue gültige Objektversionen aus dem DB-Cache auf die DB-Platten. Auf die eben genannten Operatoren werden wir später noch detaillierter eingehen.
Die Operatoren CREATE_SEG und DROP_SEG werden zur Realisierung (hinsichtlich Reservierung/Freigabe von DB-Plattenspeicher) von DDL-Anweisungen der Art 'CREATE RELATION...', 'CREATE INDEX...', 'DROP RELATION...' und 'DROP INDEX...' benötigt. Genaueres dazu enthalten die Kap. 3.2.4.2 und 3.3.2.

Die Bedeutung der Parameter der drei Leseoperationen ist wie folgt:

<Relid> : eindeutiger Identifikator für eine Benutzerrelation.

<lid> : eindeutiger Identifikator für ein Systemtupel oder Indexknoten.

<LI-filterformula> : ist eine LI-Filterformel.

<aggr_fcts> : zusätzlich auszuwertende Aggregatsfunktion(en).

<attr_list> : diejenigen Attribute, welche nicht wegprojeziert werden dürfen.

<tupelidlist> : ist eine Tupelidliste für Tupel von Relid, welche vorher vom SAM bzw. QM berechnet wurde. Nach Möglichkeit ist diese bereits so sortiert, daß die Plattenarmbewegung auf die entsprechenden Tupel minimiert sind (vgl. dazu

Kap.3.3.2.1).

<tupel_layout> : hier bekommt eine LI Information über die Internstruktur von Tupeln mitgeliefert (z.B. Attributlängen,-positionen)

<C-table_addr> : bezeichnet die DB-Cache-Adresse (einschließlich Länge) einer bereits reservierten C-Tabelle.

<C-record_addr> : analog für einen C-Satz.

<id_flag> : boolesche Größe, welche angibt, ob der Tupelid als Teil eines gesuchten Tupels mitgeliefert werden soll (Begründung erfolgt in Kap.3.3.2.1).

Einige Anmerkungen:

(a) ad <tupel_layout>:

Die Versorgung einer LI mit der Strukturinformation für Tupel hat den Vorteil, daß die LI nicht auf die Systemrelationen zugreifen muß, um Tupel richtig analysieren zu können. Jede LI muß somit nur Tupelgrenzen auf den Blöcken der DB-Platten kennen. Dieses Vorgehen entspricht wiederum dem Prinzip der strikten Trennung von logischen und physischen Aspekten, d.h. im vorliegenden Fall, daß eine LI nur mit Details der DB-Platten-Speicherorganisation betraut werden darf. Eine alternative Lösung wird beim DBC-Datenbank-Rechner ([BANE78]) beschritten: selbstbeschreibende Abspeicherung von Tupeln auf den DB-Platten. Der Nachteil dieser Organisation ist ein bis zum Doppelten erhöhter Plattenspeicherbedarf für Relationen, weswegen wir uns für die angegebene Lösung entscheiden (vgl. auch Diskussion in Kap.3.2.4.2).

(b) ad <C-table_addr>:

Wie schon erwähnt ist es bei diesem Verfahren der a priori Reservierung wichtig, gute Größenschätzungen für C-Tabellen zur Hand zu haben. Der Hauptgrund liegt nicht etwa in einer potentiellen DB-Cache-Speicherverschwendung bei zu hoch angesetzten Schätzungen, sondern genau der umgekehrte Fall soll vermieden werden. Der Transfer von gefilterten Daten in das DB-Cache ist genau dann am effizientesten, wenn die gesamten Ergebnisdaten 'in einem Schwung' übertragbar sind mit nur einem

Unterbrechungssignal vom Backend an den Host nach erfolgreichem Ende der gesamten Übertragung. Diesen Vorgang wollen wir als C-Table-Swapping bezeichnen. Stellt sich hingegen die reservierte C-Tabelle während der Datenübertragung als zu klein heraus, dann muß zum einen der Filterungsprozeß vorerst gestoppt werden, und dann vom CM eine Verlängerung der C-Tabelle beantragt werden. (Die denkbare Alternative einer a posteriori Reservierung durch die LI scheitert vermutlich am zu kleinen LI-Arbeitsspeicher).

Effizienzfragen bei der Implementierung linearer Operationen.

Der anzustrebende Idealfall bei der Implementierung der Operationen EXH_FILTER und SEL_FILTER ist folgender:
Die betroffenen Tupel können von der DB-Platte mit voller Plattengeschwindigkeit - d.h. ohne den Verlust von unnötigen Plattenumdrehungen - in den LI-Arbeitsspeicher gelesen und dort von der LI 'im Flug' entsprechend der anzuwendenden linearen Operation analysiert werden. Die Auswertungszeit für EXH_FILTER bzw. SEL_FILTER entspricht dann der Übertragungszeit von der DB-Platte, welche wir als Realzeit bezeichnen wollen. Falls EXH_FILTER bzw. SEL_FILTER in Realzeit ausgeführt werden kann, so sprechen wir von einer Im-Flug-Filterung.

Prinzip zur Realisierung der Im-Flug-Filterung:
Die Auswertung linearer Operationen wird 'off-the-disk' unter Zuhilfenahme der Wechselpuffertechnik vorgenommen.

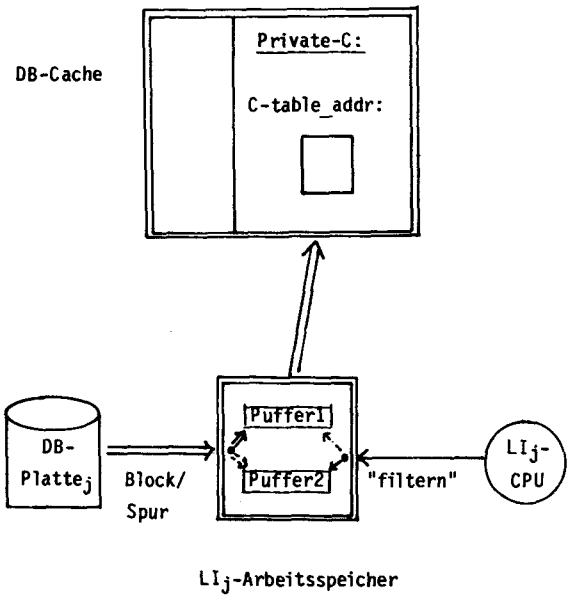


Abb.3.7.: Prinzip der Im-Flug-Filterung bei 'off-the-disk'-Verarbeitung.

Dabei wird ein Puffer von der DB-Platte mit 'rohen' Daten gefüllt, während parallel dazu die Daten im zweiten Puffer von der LI gemäß der anzuwendenden linearen Operation (Restriktion, partielle Projektion, Aggregatsfunktion bzw. Kombination aus diesen) verarbeitet werden. Die Eigenschaft einer linearen Operation - jedes Tupel muß höchstens einmal betrachtet werden - sollte nun garantieren, daß die LI mit der Verarbeitung eines Puffers fertig wird, bevor der Datentransfer in den anderen Puffer beendet ist, wonach sofort die Rollen der beiden Puffer für die weitere Verarbeitung vertauscht werden können (d.h. für diesen Fall liegt tatsächlich eine Im-Flug-Filterung in Realzeit vor). Falls jedoch die LI bei der Ausführung der linearen Operation zu langsam ist, so muß mit dem weiteren Datentransfer von der DB-Platte gewartet werden, was den Verlust von einer oder mehreren Plattenumdrehungen zur Folge hat.

Die Verarbeitungszeit einer linearen Operation ist somit abhängig von ihrer Komplexität, d.h. z.B. von der Anzahl von Konjunktionen und Disjunktionen in einer LI-Filterformel und der Anzahl der Aggregatsfunktionen. Je nach der Zahl der verlorenen Plattenumdrehungen ergeben sich somit für die Verarbeitungszeit einer linearen Operation Treppenfunktionseffekte in Abhängigkeit ihrer Komplexität.
Für jeden konkreten LI-Prozessor wird es somit eine Im-Flug-Komplexitätsgrenze für lineare Operationen geben, unterhalb derer das Wunschziel einer echten Im-Flug-Filterung erfüllt ist.
Diese interessante Frage der Bestimmung der Im-Flug-Komplexitätsgrenze (in Abhängigkeit der Geschwindigkeit der DB-Platte und des LI-Mikroprozessors) kann im Rahmen dieser Arbeit nicht umfassend weiterverfolgt werden.

Stattdessen begnügen wir uns mit einer groben Abschätzung der Anforderungen an die Prozessorleistung eines LI-Prozessors, wenn als DB-Platte die momentan schnellste Platte IBM 3380 eingesetzt wird.
Die maximale Datentransferrate beträgt für die 3380 etwa 3MBytes/sec.
Angenommen, bei der Filterung

- muß jedes Byte angeschaut werden (ungünstigster Fall),
- der Test eines Bytes erfordert drei Instruktionen (günstigster Fall).

Daraus ergibt sich als untere Schranke eine Anforderung von 9 MIPS für einfache Vergleichsoperationen an die Prozessorleistung des Filterprozessors. (Anm.: Der Filterprozessor 'Accelerator' in der IDM 500 leistet 10 MIPS) Eine solche hohe MIPS-Rate ist natürlich mit konventionellen Mikros nicht erzielbar. Aufgrund der Information in <tupel_layout> ist diese allerdings im Normalfall auch nicht notwendig, da die LI gezielt immer nur einen Teil eines Tupels testen muß. Deshalb ist zu vermuten, daß im Normalfall mit 1-2 MIPS Mikroprozessoren das Ziel einer Im-Flug-Filterung für eine sehr große Klasse von Filterprädikaten verwirklicht werden kann. Abschließend zum Komplex der Im-Flug-Filterung sei noch vermerkt, daß bei Verwendung von firmware-programmierbaren LI-Prozessoren weitere kostengünstige Leistungssteigerungen zu erzielen

sind.[46]

Die Wahl einer vernünftigen Puffergröße wird von folgenden Faktoren beeinflußt:

(1) Was ist eine 'vernünftige' Filtereinheit:
Tupellokale Filterung auf Blockbasis ist nur möglich, wenn die zu prüfenden Attribute in einem Block liegen; anderenfalls ist zu größeren Einheiten überzugehen (mehrere Blöcke bis maximal eine Spur). Unter der Annahme, daß die obige Voraussetzung erfüllt ist, ist für die Operation SEL_FILTER Filterung auf Blockbasis vorteilhaft. Für EXH_FILTER hingegen kann Filterung auf Spurbasis besser sein.

(2) Kapazität des verfügbaren LI-Arbeitsspeichers:
Da wir als LI-Prozessoren billige Standard-Mikros einsetzen wollen, ist die Kapazität des LI-Arbeitsspeicher relativ gering.
Für die IBM-Platte 3380 beträgt die Speicherdichte etwa 47.5 KBytes/Spur. Somit benötigen wir bei spurweiser Filterung 95 KBytes für die beiden Wechselpuffer.

Für den Nachweis, daß die Filterung eines Puffers parallel zur Füllung des anderen mittels heute schon im Handel verfügbaren Mikros realisierbar ist, wollen wir uns mit einem Hinweis auf die Mikroprozessoren 8086 und 8089 von INTEL beschränken. Der 8089 Input/Output Prozessor ist ein hochleistungsfähiges General-Purpose I/O-System, welches auf einem einzigen Chip implementiert ist, und ist mit dem 8086-Prozessor kompatibel. Der Befehlssatz umfaßt ca. 50 verschiedene Befehlstypen, welche speziell in Hinblick auf eine effiziente I/O-Verarbeitung konstruiert wurden. Innerhalb eines 8089 gibt es zwei unabhängige I/O-Kanäle, von denen jeder die Eigenschaften einer CPU mit denen eines sehr flexiblen DMA-Kontrollers kombiniert. Jeder Kanal kann mit hoher Geschwindigkeit DMA-Übertragungen ausführen; optionale Operationen gestatten es, Daten während der Übertragung zu manipulieren (codieren/decodieren, Suchvergleiche). Die Zusammenarbeit eines 8086 und 8089 geht konzeptionell wie folgt vor sich:

[46] Erfahrungen auf diesem Gebiet liegen vom GDS-Backend-Rechner [MAKI82] vor. Firmware-Programmierung häufig benötigter DB-Funktionen brachte hier eine Leistungssteigerung um den Faktor 4-10.

Der 8086 soll dabei die Filterung der Daten vornehmen, während der 8089 die Aufgabe eines DMA-Kontrollers übernimmt. Wichtig ist dabei, daß der 8086 und 8089 im Prinzip vollkommen parallel arbeiten können (vgl. [INTE79]).
Die Verfügbarkeit in der nächsten Zukunft von immer intelligenteren Kontrollern wird auch in [SMIT81] angedeutet. Neben der Fähigkeit zur Ausführung komplexer Operationen wird dabei ein wesentliches Merkmal dieser neuen Generation von Kontrollern sein, daß sie Daten zwischenpuffern können, wenn I/O-Pfade nicht frei sind. Diese Eigenschaft ist gerade für die Realisierung der Im-Flug-Filterung mittels der Wechselpuffertechnik interessant. Die Kombination 8086/8089 ist bereits als ein Schritt in diese Entwicklungsrichtung zu sehen.

Aspekte der Implementierung der Durchschreibe-Operatoren.

Die relevanten SI-Operatoren sind
- PUT_TAB: Durchschreiben aller (Teil-)Tupel in einer C-Tabelle aus Committed-C.
- PUT_RECS: Durchschreiben eines oder mehrerer Objekte aus Read-C, das sind Indexknoten oder Systemtupel.

Beim Durchschreiben solcher neuer gültiger Objektversionen ergeben sich einige Recovery-Probleme, die aus der unterschiedlichen Granularität von logischen und physischen Objekten resultieren. Ausschlaggebend dafür ist eine Eigenschaft konventioneller Platteninterfaces:
Information ist nur in Einheiten fester Blöcke schreibbar![47)]
Diese Einschränkung zieht folgende Konsequenzen nach sich:
Um eine neue gültige Objektversion (geändertes Benutzertupel oder Systemtupel) auf die DB-Platte schreiben zu können, muß erst der ganze Block, welcher das betreffende Tupel enthält, im Arbeitsspeicher der LI zur Verfügung stehen.
Daraufhin muß die neue zu schreibende Blockversion konstruiert werden (dieser Vorgang kann durch Kanalprogramme optimiert werden).

[47)] Standard-Schnittstelle zwischen LI und DB-Platte:
Lies Block, Schreibe Block bzw. Lies Spur, Schreibe Spur.

Im nächsten Schritt jedoch darf diese neue Blockversion nicht sofort auf die DB-Platte geschrieben werden!
Die Gründe dafür sind folgende:

(1) Die von der LI angewandte Blockupdate-Methode ist die Update-in-Place-Technik, d.h. geänderte Blockversionen werden zwecks Erhaltung physisch-sequentieller Clustereigenschaften auf dieselbe DB-Plattenstelle zurückgeschrieben.

(2) Die Recovery-Information des DB-Cache/SAFE-Systems ist objektorientiert (≠ Blockgröße).

Somit können beim Schreibvorgang direkt auf der DB-Platte gültige andere Objektversionen in dem betreffenden Block im Fehlerfalle zerstört werden, ohne daß für diese Objekte im SAFE Recovery-Information gesichert ist.
Fazit: Eine Zusatzsicherung ist notwendig.

Als einfache, jedoch effiziente Lösungsmethode schlagen wir die sogenannte MINISAFE-Methode vor:

MINISAFE-Methode für ein sicheres Durchschreiben eines DB-Objektes vom LI-Arbeitsspeicher auf die DB-Platte:

(MS1) Lese zu schreibenden Block in den LI-Arbeitsspeicher.

(MS2) Baue neue Tupelversion ein (kann durch Kanalprogramme beschleunigt werden).

(MS3) After-Image-Protokollierung: Schreibe diese neue Blockversion auf einen Sicherungsblock.

(MS4) Sobald (MS3) erfolgreich ist, schreibe neue Blockversion in-place auf die DB-Platte.

Bewertung dieser MINISAFE-Methode:

(1) Schritt (MS1) ist evt. vermeidbar, falls die LI über ein zusätzliches blockorganisiertes I/O-Cache verfügt.

(2) Schritt (MS2) könnte für Indexknoten entfallen, falls in üblicher Weise ein Knoten auf einen Block abgebildet wird.

(3) Die Zusatzsicherung unter (MS3) ist bei Plattengeräten mit einem extra Festkopfzylinder (wie bei der IBM 3380) effizient realisierbar. Die

Schritte (MS3) und (MS4) sind dann innerhalb einer Plattenumdrehung ausführbar. Weitere Optimierungen sind möglich, da PUT_TAB und PUT_RECS Mengenupdates sind (Minimierung der Plattenarmbewegungen, geschicktes Interleaving mit dem Schreiben von Sicherungsblöcken).

(4) (MS3) und (MS4) erfordern ein effizientes read-after-write Feature, z.B. Aufbau von CRCs während des Schreibvorgangs. Die Schreibvorgänge sind gegebenfalls solange zu wiederholen, bis sie erfolgreich sind (falls ein Block als defekt betrachtet wird nach einigen erfolglosen Schreibversuchen, muß ein Ersatzblock besorgt werden).

Schließlich wollen wir noch eine wichtige Eigenschaft der Durchschreibeoperatoren bei Verwendung der MINISAFE-Methode festhalten: PUT_TAB und PUT_RECS sind atomare und idempotente (wiederholbare, restartable) Operatoren.

Schlußbemerkung zu diesem Komplex:
DB-Plattenfehler-Recovery ist die autonome Aufgabe jeder LI.

3.2.4.2 Effiziente DB-Plattenorganisation.

In einem Im-Flug-Filterungssystem ist die Verarbeitungszeit von linearen Filteroperationen auf einer Relation R in erster Linie abhängig von

(1) der Anzahl von Blöcken (Spuren) auf den DB-Platten, in denen Tupel von R gespeichert sind,

(2) von der Verteilung dieser Blöcke (Spuren) auf den DB-Platten (physisch-sequentiell oder 'wild' verstreut über die ganze Platte).

Da die von uns verwendeten DB-Platten über nur einen aktiven Lesekopf verfügen, ergeben sich aus (1) und (2) die folgenden Anforderungen an eine geeignete Ablage von Relationen auf den DB-Platten:

(a) Relationen sind streng physisch-sequentiell abzuspeichern.

(b) Der unter (a) belegte Speicherplatz ist zu minimieren.

Eine solche Ablageform reduziert offensichtlich die Ausführungszeit der Filteroperationen, da zum einen die Anzahl langsamer Plattenarmbewegungen minimiert ist, zum anderen bei vollständigen Relationenfilterungen die Anzahl

der zu lesenden Blöcke (Spuren) reduziert wird.[48] Diese anzustrebende kompakte physisch-sequentielle Relationenspeicherung ist möglich aufgrund des Cache/SAFE-Konzepts, welches einen In-Place-Update auf den DB-Platten zuläßt (im Gegensatz zum Schattenspeicherkonzept in SystemR [LORI77]). Die genannten Anforderungen schließen auch aus, daß Tupel verschiedener Relationen auf einem Block (Spur) abgelegt werden dürfen, wie etwa in SystemR. Diese Form der Clusterung wird von uns als nicht nützlich angesehen, da sie nur bei Verwendung von Zugriffspfaden des Typs 'link' ausgenützt werden kann (solche links sind typische Zugriffspfade in hierarchischen oder Codasyl-Datenbanken).

Im Einklang mit den Leistungsmerkmalen der modernen Plattentechnologie (schnelles sequentielles Lesen, langsame Armpositionierungszeiten) und den funktionalen Erfordernissen der LIs, schlagen wir folgende Db-Plattenorganisation vor:

Definition:

- Ein Bereich (extent) ist eine Folge streng physisch-sequentieller Blöcke auf einer DB-Platte.[49]
- Ein Segment wird auf einen oder mehrere Bereiche abgebildet.

(D1) Ablage einer Benutzerrelation auf einer DB-Platte:[50]
  (a) Eine Benutzerrelation wird in einem Segment abgelegt.
  (b) Dieses Segment enthält nur diese eine Benutzerrelation.

(D2) Ablage eines Index auf einer DB-Platte:
  (a) Ein Index wird in einem Segment abgelegt.
  (b) Dieses Segment enthält nur diesen einen Index.

---

[48] EXH FILTER liest nur die Blöcke der betroffenen Relation (analog zur Datenbankmaschine DBC), und nicht die ganze DB-Platte wie etwa RAP.

[49] und sollte sich über möglichst wenige Zylinder erstrecken.

[50] Die DBC-Maschine verwendet hier in etwa dieselbe Methode, welche als 'clustering by relation' bezeichnet wird.

(D3) Ablage von Systemrelationen auf einer DB-Platte:

(a) Alle Systemrelationen werden in einem Segment abgelegt.

(b) Dieses Segment enthält nur diese Systemrelationen.

Diese enorm simple Speicherorganisation der DB-Platte ist genau auf die funktionalen Bedürfnisse unserer Backends zugeschnitten:

- (D3) ermöglicht ein effizientes Laden von Systemkatalogen bei DB-Sessionbeginn.
- (D1) und (D2) ermöglichen eine effiziente Implementierung der SI-Operatoren CREATE_SEG und DROP_SEG.
- (D1) soll die Geschwindigkeit der Filteroperationen erhöhen.[51] Um dieses Ziel zu erreichen, insbesondere bei vollständigen Relationenfilterungen mittels EXH_FILTER, muß die Anzahl von Bereiche eines Segments minimiert werden.

Somit hat jede LI folgende Speicherverwaltungs-Strategie für seine DB-Platte zu verfolgen:
Es wird grundsätzlich versucht, eine Benutzerrelation auf ein Segment abzubilden, welches genau einen Bereich umfaßt.
Offensichtlich erreichen wir mit dieser Strategie, daß die Anzahl der Plattenarmbewegungen bei beiden Filteroperationen reduziert wird. Falls ein Segment im Lauf der Zeit aufgrund von Tupeleinfügungen überläuft, so muß dieses Segment verlängert werden. Entgegengesetzt, falls ein Bereich durch Tupellöschungen leer werden sollte, so ist das betreffende Segment zu verkürzen.

Konsequenz dieser Strategie:
Beim Einrichten eines Segments muß eine gute a priori Schätzung des voraussichtlich benötigten Platten-Speicherumfangs an die LI mitgeteilt werden. Für diesen Zweck wurde der zweite Parameter im Operator CREATE_SEG (<Relid/Indid>,<space spec>) vorgesehen.

Für die Behandlung von Einfügungen bzw. Löschungen von DB-Objekten in

[51] Ein weiterer wichtiger Vorteil bzgl. der effizienten Abb. von DB-Objekten auf die physischen Plattenadressen wird in Kap.3.3.2.1 diskutiert.

Segmenten benötigen wir ferner die folgenden LI-internen Operationen:

- LENGTHEN_SEG: "Verlängere existierendes Segment um einen neuen Bereich."
- SHORTEN_SEG : "Verkürze existierendes Segment um einen gegebenen Bereich."

Die Größe eines aufgrund von LENGTHEN_SEG neu zu reservierenden Bereichs wird dabei von der LI selbst bestimmt (z.B. in Abhängigkeit von noch vorhandenen freien Plattenspeicher unter Berücksichtigung von bisherigen Insert-Raten).

Abschließend zu diesem Komplex der DB-Plattenverwaltung auf Segment/Bereichs-Basis sei noch vermerkt:

- Ein Nachteil dieser vorgeschlagenen Organisation ist eine Speicherfragmentierung bzw. Speicherplatzverschwendung bei schlechter Schätzung für CREATE_SEG (extern durch Benutzer) oder LENGTHEN_SEG (intern durch LI).
- Der für die N.A. unerwünschte Fall, daß ein Bereich einer Seite (Block) entspricht, ergibt die Speicherzuordnungs-Schnittstelle im Projekt LEO ([BITT82]), welche dort als Gebiet bezeichnet wird.

Internorganisation eines Segments:

Der zweite Aspekt zur Minimierung der Ausführungszeiten unserer beiden Filteroperatoren betrifft die Frage, inwieweit die Anzahl der Blöcke eines Segments, in denen Tupel gespeichert sind, reduziert werden kann.
In diesem Zusammenhang ist zu untersuchen, ob eine Clusterbildung innerhalb eines Segments nach gewissen Attributen für die N.A. sinnvoll ist. Eine solche Clusterbildung ist stets verbunden mit einer sortierten bzw. teilsortierten Abspeicherung der Tupel nach einem Attributwert.
Für die Plattenorganisation der bekannten DB-Systeme läßt sich folgendes feststellen:

- DBMS ohne Backends (wie SystemR, INGRES) verwenden sortierte Abspeicherung.
- Spezialisierte DB-Maschinen, welche als Zugriffspfad nur das voll-

ständige Durchsuchen der gesamten physischen DB kennen (wie z.B. RAP), bevorzugen eine unsortierte Abspeicherung.

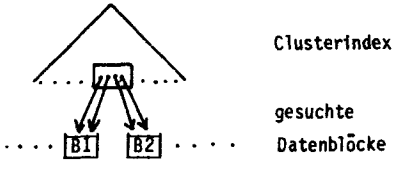
Eine Zwischenstellung nimmt der DBC-Rechner ein, welcher eine Clusterbildung innerhalb einer Relation auf Zylinderbasis vornimmt. Diese Lösung ist speziell auf das parallele Auslesevermögen der verwendeten DB-Plattengeräte zugeschnitten (ein Zylinder ist in einer Plattenumdrehung auslesbar).

Vorteil sortierter Speicherung:

Sortierte Abspeicherung ermöglicht die Einrichtung genau eines Clusterindexes (auch Primärindex genannt).

In DB-Systemen mit tupelweisem Zugriffssystem (Einzeltupelverarbeitung), ohne Datenfilterung und kleinem DB-Pufferbereich wird der Zugriff über einen Clusterindex als wesentlichste Möglichkeit der Leistungssteigerung bei der Queryauswertung betrachtet, und zwar aufgrund folgender Eigenschaft: Jeder (Daten-)Block, welcher gesuchte Daten enthält, muß höchstens einmal von den DB-Platten geholt werden, und die Anzahl betroffener Blöcke ist minimal.

Zur Illustration:

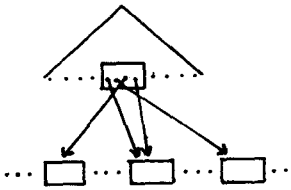


Aufgrund der hohen zeitlichen Lokalität beim Zugriff auf B1 und B2 ist es auch in solchen DB-Systemen (Einzeltupelverarbeitung, kleiner DB-Puffer, fehlende Datenfilterung und somit vorzeitiges Verdrängen von Information) sehr wahrscheinlich, daß B1 bzw. B2 solange wie benötigt im DB-Puffer bleiben.

Anmerkung:

Nichtcluster-Indexe (Sekundärindexe) zeigen i.a. in DB-Systemen wie SystemR nicht obiges Verhalten.

Zur Illustration:



Neben der Einrichtung von Clusterindexen erlaubt eine sortierte Abspeicherung auch das Anlegen von nicht-dichten Indexen.

Nachteil sortierter Speicherung:
Für Einfügungen muß 'genug' Platz[52] auf den Blöcken reserviert werden (vgl. z.B. die ISAM-Organisation). Die Tupel einer Relation sind somit auf mehr Blöcken als unbedingt nötig abgespeichert.

Beurteilung für die N.A.:
(1) Der eine Vorteil eines Clusterindexes, daß jedes Tupel nur einmal von den DB-Platten geholt werden muß, wird in der N.A. wie folgt erzielt:
- Aufgrund der Mengenverarbeitung steht bei SEL_FILTER-Aufträgen immer eine komplette Tupelid-Liste zur Verfügung, welche somit nach physisch-sequentieller Ordnung sortiert werden kann. Deshalb müssen betroffene Blöcke nur einmal von den DB-Platten in den LI-Arbeitsspeicher zur Filterung geholt werden. Zu beachten ist dabei auch, daß die Ausführungszeiten für SEL_FILTER-Aufträge geringer sind als die entsprechenden random Blockzugriffe in konventionellen DB-Systemen (vgl. auch Kap. 5.1.3).
- Eine durch SEL_FILTER in das DB-Cache transportierte ZER bleibt dort infolge der DB-Cache-Verwaltungsstrategien im Normalfall solange wie benötigt. Daß dieser Normalfall für eine sehr große Klasse von Transaktionen zutrifft, wird durch spezielle Techniken zur frühzeitigen Datenfilterung gewährleistet (vgl. dazu Kap.4). Somit wird durch die Architektureigenschaften der N.A. ermöglicht,

[52] Typische Freiplatz-Reservierungen liegen im Bereich von 15-30%.

daß bei Restriktionen für alle Attribute die gesuchten Tupel nur einmal von den DB-Platten geholt werden müssen.

(2) Der genannte Nachteil schlägt voll auf die Ausführungszeiten insbesondere der EXH_FILTER-Operationen für Restriktionen auf Sekundärattributen durch. (Filteroperationen auf dem Primärattribut werden jedoch länger dauern, wenn keine Sortierung nach dem Primärattribut vorliegt.)

Aufgrund dieser Überlegungen entscheiden wir uns für folgende Organisationsform von Relationen in Segmenten:

- Keine Sortierordnung in Segmenten.
- Füllung der Blöcke mit Tupeln soweit wie möglich.
- Falls bei Einfügungen kein Platz in schon belegten Blöcken ist, dann Abspeicherung im nächsten physisch-benachbarten Block des Bereichs.

Diese Segment-Organisation wird in der Literatur bisweilen als 'entry-sequenced organization' bezeichnet; sie entspricht i.a. einer chronologischen Organisation und unterstützt damit in wirksamer Weise das Löschen veralteter Daten.
Um eine EXH_FILTER-Operation möglichst frühzeitig terminieren zu können, ist bei dieser Organisationsform die Einrichtung von Endemarken der Form eotr (end of occupied track) und eo-ext (end of occupied extent) möglich.

Anmerkung:
Mit der gewählten unsortierten Abspeicherung entfällt die Unterscheidung in Primärattribut und Sekundärattribute. Jeder Index ist nach herkömmlicher Terminologie Sekundärindex, jedoch mit veränderten Eigenschaften.

3.3. Zuverlässigkeits- und Effizienzaspekte der Koppelung zwischen Backend und Host.

Unsere bisherigen Untersuchungen haben sich auf mehr oder weniger statische Eigenschaften der Systemarchitektur konzentriert. Diese statischen Systemeigenschaften sind durch das modularisierte DBMS-Schichtenmodell festgelegt. Als zweiter großer Komplex bei einer Systembeschreibung ergibt sich die Notwendigkeit einer Spezifikation der operationalen Charakteristika des DB-Systems. Eine solche operationale Charakterisierung hat dabei über die pure operationale Interpretation von Systemabläufen hinauszugehen, welche durch die Aufrufstruktur der einzelnen Moduln festgelegt ist. Der wichtigste Aspekt dynamischen Verhaltens bei Backend-Systemen ist die Frage, ob gekoppelte Maschinen zuverlässiger sind als eine zentrale Maschine.

3.3.1. Das Problem der losen Koppelung zwischen Backend und Host.

Bei der Verteilung der DBMS-Aufgaben auf den Host und die Backends orientierten wir uns stets an dem zentralen Leitmotiv der 'Konzentration von logischen und physischen Aspekten auf separate Teilsysteme'.
Diese Entwurfsforderung kann als Teil des umfassenden Begriffs der Datenunabhängigkeit gesehen werden. Mit der Entwicklung deskriptiver DB-Sprachen wurde die Abkoppelung der externen Benutzersicht von der konzeptuellen Sicht des DBMS erreicht. Jedoch scheint in keinem existierenden DBMS eine ähnliche Qualität zwischen dem konzeptuellen Modell und dem physischen Modell (DB-Speichersubsystem) bisher realisiert worden zu sein.
Das genannte Leitmotiv führte bei uns zu einer funktionalen Aufteilung des DBMS in zwei relative unabhängige Teilsysteme mit eigenen Betriebsmitteln:

(1) Logische Aspekte übernimmt das Host-DBMS, eigene Betriebsmittel sind dabei DB-Cache-Platz (objektorientiert!) und Sperren auf DB-Objekten.
(2) Physische Aspekte übernimmt das Intelligente DB-Speichersystem auf den Backends, bei den Betriebsmitteln handelt es sich um die DB-Platten und

LI-Verwaltungsinformation.

Die zentrale Frage nach der Qualität und Zuverlässigkeit der Koppelung dieser beiden Teilsysteme soll nun anhand des Zusammenspiels beider Teilsysteme bei der Realisierung des Transaktionsbegriffs analysiert werden. Um die Problematik besser verdeutlichen zu können, sollen einige strukturelle Probleme in existierenden DBMS aufgezeigt werden.

Einige strukturelle Probleme in existierenden DBMS:

Problem 1: DB-Objektgranularität ≠ DB-Pufferverwaltungseinheit
Alle konventionellen DB-Systeme verwenden seitenorganisierte DB-Puffer
=====>

(1a) Sperr-Dilemma: Falls logische Objektsperren gewählt werden, so sind zusätzliche physische Kurzzeitsperren auf Seiten notwendig (vgl. SystemR [ASTR76]).

(1b) Commit&Recovery-Dilemma, falls die separat schreibbaren physischen Einheiten auf den DB-Platten verschieden von den DB-Objektgrößen sind (z.B. mehrere einzeln sperrbare Tupel befinden sich auf einer Seite, welche nur als ganzes auf die DB-Platte geschrieben werden kann): aufwendige Logging- und Recoveryverfahren sind die Folge.[53]

Problem 2: Verwaltung von DB-Plattenbelegungsinformation.
Die Identifizierung von Tupeln geschieht in SystemR (und vielen anderen DBMS) mittels des TID-Mechanismus. Ein TID besteht aus einer Seitennummer und einer Relativadresse, und vermischt somit logische und physische Sachverhalte.

=====> Recovery- und Concurrency-Probleme[54] mit Speicherabbildungstabellen, wenn TIDs vor Transaktions-Commit vergeben oder gelöscht

[53] Die Einschränkung auf ein Seiten-DB-Cache in [ELHA82] entspringt ebenfalls dieser Problematik.

[54] Sehr feine Sperr-Granularitäten (<<Seitengröße) erforderlich, andernfalls serialisieren sich die Transaktionen an diesen Systemdaten; dazu werden oft separate Spezial-Algorithmen verwendet.

werden.

Die strikte Einhaltung des funktionalen Isolierungsprinzips in logische und physische Aspekte bei der Konstruktion des DBMS für die N.A. bewirkt, daß obiges Problem1 in der N.A. sehr zufriedenstellend gelöst ist. Aufgrund der Systemspezifikation im Kap.3.2 können wir folgende Architekturmerkmale festhalten:

(1) Es müssen nur Sperren auf logischen DB-Objekten gesetzt werden.

(2) Es besteht eine funktionale Aufteilung der Recovery in

(2a) logische Recovery durch Objekt-DB-Cache/SAFE

(2b) physische Recovery durch die LIs und DB-Platten.

Die Eigenschaften (1) und (2a) wurden erreicht durch die Konstruktion des Objekt-DB-Cache, Eigenschaft (2b) durch die Realisierung der Backends als sicheres Speichersubsystem mit atomaren und wiederholbaren Schreiboperatoren.

Zu Problem 2 wurde bisher noch keine Aussage getroffen. Da die Verwaltung der DB-Plattenbelegungsinformation von den einzelnen LIs vorgenommen wird, werden hierbei Interaktionen zwischen Backends und Host erforderlich und somit stellt sich die Frage dem Grad der operationalen Kopplung (und der daraus resultierenden Zuverlässigkeit und Effizienz) zwischen Backends und Host.

Unter dem Begriff der operationalen Kopplung wollen wir nicht hardware-orientierte Aspekte verstehen, sondern die Art und Weise, wie die Interaktion zwischen Host und Backends zur Realisierung des Transaktionsbegriffs abläuft. Diese operationale Kopplung ist auch zu unterscheiden von der logischen Kopplung, welche durch die Notwendigkeit der Abbildung von logischen auf physische Sachverhalte entsteht.

Probleme der operationalen Kopplung lassen sich am besten anhand des Einfüge-Problems verdeutlichen.

Beim Einfügen eines neuen DB-Objektes sind grundsätzlich zwei verschiedene Aufgaben zu bewältigen:

(I1) Es muß ein neuer Objektidentifikator generiert werden.

(I2) Es muß freier Speicherplatz auf den DB-Platten besorgt werden; dazu müssen Speicherbelegungstabellen geändert werden.

Mittels dieser Abstraktion werden die Auswirkungen der Vermischung von logischen und physischen Aspekten beim TID-Mechanismus sofort klar: Mit einer TID-Vergabe ist untrennbar die Tupelidentifizierung mit der Speicherplatzreservierung auf einer Seite verbunden.
Konsequenz: Die Verwaltung dieser Speicherbelegungstabellen muß transaktionsorientiert sein, d.h. im Fehlerfalle sind Änderungen von zurückzusetzenden Transaktionen auch für diese Systeminformation rückgängig zu machen (vgl. Problem 2).
Ein solches Verfahren hätte für die N.A. eine sehr enge operationale Kopplung zwischen Host und Backends zur Folge, da

- TID-Vergabe nur unter Mitwirkung einer LI möglich ist,
- aufwendige Interaktionen bei Transaktionsfehler zwischen zwei ansonsten relativ unabhängigen Teilsystemen stattfinden müßten.

Die Ursache für die geschilderten Unannehmlichkeiten ist die Vermischung von logischen und physischen Aspekten beim TID-Konzept. Bei Codasyl-DBs ist eine Entflechtung der Punkte (I1) und (I2) durch die Einführung der 'database keys' als Objektidentifikatoren gegeben. Jedoch ergeben sich hier ähnliche operationalen Schwierigkeiten, wenn Punkt (I2) vor EOT vorgenommen wird.
Aus den geschilderten Beobachtungen beim Einfüge-Problem ziehen wir folgende Erkenntnis:
Tatsache 1: Es ist nicht wünschenswert, die LIs in das Transaktionskonzept einzubinden.
Unter Berücksichtigung unseres bisher strikt eingehaltenen horizontalen Aufteilungsprinzips - logische Aspekte im Host-DBMS, d.h. insbesondere Sperren und Konsistenzprüfungen oberhalb der LIs ausschließlich im Relationalen Zugriffssystem - halten wir fest:
Tatsache 2: Es ist nicht notwendig, die LIs in das Transaktionskonzept einzubinden.

Diese beiden Sachverhalte veranlassen uns nun zu der Anforderung, daß zwischen Host und Backends eine lose Koppelung bestehen muß. Dabei definieren wir, daß die lose Koppelung gewährleistet ist, wenn folgendes

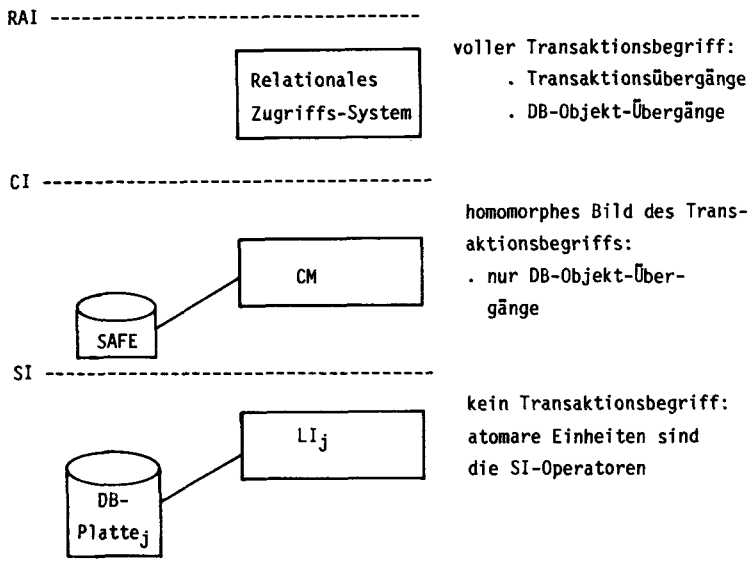


Abb.3.8: Realisierung des Transaktionsbegriffs in der N.A.

erfüllt ist:

(1) Konzentration logischer Aspekte in das Host-DBMS.
(2) Konzentration physischer Aspekte in die LIs.
(3) Die Abbildung logischer auf physische Aspekte geschieht in den LIs.
(4) LI-spezifische Daten - das sind z.B. Speicherabbildungsinformation wie Segmenttabellen - werden nicht transaktionsspezifisch geändert.

Die von uns beabsichtigte Realisierung des Transaktionsbegriffs durch das DBMS ist in Abb.3.8 beschrieben.

Die gestellte Anforderung läßt sich dahingehend interpretieren, daß 'Transaktion' ein rein logischer Begriff ist, für den der CM die logischen Recoveryverfahren realisiert. Das Intelligente DB-Speichersystem hingegen

ist ein sicheres Speichersubsystem, welches keine Transaktionen kennt und somit auch nicht mit den Rücksetzungsproblemen bei Transaktionsfehlern belastet ist. Bei der Implementierung der atomaren Schreiboperationen können sich somit beträchtliche Effizienzvorteile ergeben. Wichtig zu vermerken ist dabei auch, daß ein SI-Schreiboperator eine sehr viel kürzere atomare Operation ist als eine aus vielen Aktionen bestehende Transaktion.

Konsequenzen der Forderung nach loser Kopplung :

Forderung an einen Identifizierungsmechanismus für permanente DB-Objekte:

(LK1) Permanente DB-Objekte besitzen rein logische Identifikatoren. Solche Identifikatoren sind völlig entkoppelt von Speicheraspekten.

Forderungen an die Durchschreibe-Strategie vom DB-Cache zu den DB-Platten:

(LK2) Nur Durchschreibe-Aufträge für gültige Änderungen werden an die LIs geschickt.

Forderung an die Behandlung von Einfügungen:

(LK3) Die Speicherbelegung auf den DB-Platten für Objekteinfügungen erfolgt durch die LIs erst bei Ausführung des Durchschreibeauftrags.

In den folgenden Kapiteln 3.3.2 und 3.3.3 werden wir nun effiziente Lösungsmethoden präsentieren, welche die Eigenschaft der losen Kopplung unter Berücksichtigung obiger Forderungen garantieren.

3.3.2. Effiziente Adressierungsmechanismen für permanente DB-Objekte.

3.3.2.1. Identifikations- und Abbildungsmechanismen.

Um auf permanente DB-Objekte assoziativ über Indexe zugreifen zu können, benötigen wir einen Adressierungsmechanismus. Prinzipiell setzt sich ein solcher Adressierungsmechanismus aus zwei Komponenten zusammen:

- einem Objekt-Identifikator,
- einer Abbildung dieses Objekt-Identifikators auf physischen

Speicherplatz auf den DB-Platten.

Die Wahl eines geeigneten Adressierungsmechanismus für DB-objekte muß sich dabei an folgenden Kriterien orientieren:

- Welches Maß an Datenunabhängigkeit ist gewährleistet?
- Wie effizient ist die Abbildung {Obj_id} ⟶ {DB-Platte}?
- Entstehen Probleme in Zusammenhang mit Concurrency und Recovery?

Das DBMS der N.A. kennt die folgenden permanenten DB-Objekte:

(1) - Relation
    - Tupel

(2) - Index
    - Indexknoten

Adressierung der Objekte 'Relation' und 'Index'.

Die Adressierung einer Relation (Index) mit der freigewählten Benutzerbezeichnung R (I) geschieht folgendermaßen:

(1) Vergabe von systeminternen Identifikatoren Relid (Indid) durch den SAM:[55]

Eintrag von {R} ⟶ {Relid} bzw. {I} ⟶ {Indid} in die Systemkataloge.

Eigenschaften: - Relids und Indids sind
. eindeutig
. nicht wiederverwendbar[56]
. rein logisch
- Die Abbildung {R} ⟶ {Relid} bzw. {I} ⟶ {Indid}ist invariant über die Lebensdauer von R (I).

Verwendung: als Parameter bei

[55] Aufgrund von DDL-Anweisungen der Form
CREATE RELATION R...bzw. CREATE INDEX I...

[56] Diese Eigenschaft ist z.B. dadurch erreichbar, indem man in den Systemkatalogen einen Zähler für den bisher vergebenen maximalen Relid bzw. Indid verwaltet (falls Relids bzw. Indids natürliche Zahlen sind).
Natürlich kann über eine lange Lebensdauer des DBMS 'nicht wiederverwendbar' nicht strikt eingehalten werden, sondern ist modulo N für ein großes N zu verstehen.

. Sperraufträgen an den LM
. Filteraufträgen (nur Relids) an eine LI
. Einrichtung/Löschung von Segmenten

(2) Vergabe von LI-internen Identifikatoren Segid für Segmente infolge eines CREATE_SEG (<Relid/Indid>,<space_spec>)-Auftrags:
Eintrag von {Relid} ⟶ {Segid} bzw. {Indid} ⟶ {Segid} in die LI-privateDB-Platten-Verwaltungstabellen. (In unserem speziellen Fall kann hierfür in beiden Fällen die Identität gewählt werden.)

Eigenschaften: - Segids sind
. eindeutig
. nicht wiederverwendbar
. rein logisch
- Die Abb. {Relid} ⟶ {Segid} bzw. {Indid} ⟶ {Segid} ist invariant über die Lebensdauer von Relid bzw. Indid.

Verwendung: - zum Zugriff auf die (noch zu beschreibende) Segmentabbildungstabelle.

Adressierung der Objekte 'Tupel' und 'Indexknoten'.

Die Erfordernisse der losen Kopplung bedingen, daß Identifikatoren für Tupel und Indexknoten rein logischer Natur sind, die sogenannten logischen Identifikatoren lids.
Die Vergabe von lids geschieht wie folgt:
- durch den UM bei Einfügungen von Tupeln in Benutzerrelationen,
- durch den SAM bei Einfügungen von Systemtupeln oder Indexknoten.

Eigenschaften: lids sind
. eindeutig
. nicht wiederverwendbar[57)]
. rein logisch (z.B. Relid.nat#)

[57)] TIDs sind wiederverwendbar, was in System R zu einigen Ärgernissen führte.

. Zuordnung Tupel/Indexknoten ⟷ lid ist invariant über die Lebensdauer des Tupels/Indexknoten.

Verwendung: als Parameter bei

. Sperranfragen an den LM
. DB-Cache-Aufträgen für Public-C-Objekte
. als Einträge in Indexen (invertierte Liste)
. als Indexverweise auf andere Knoten
. PUT_TAB, PUT_RECS, SEL_FILTER-Aufträgen an die LIs.

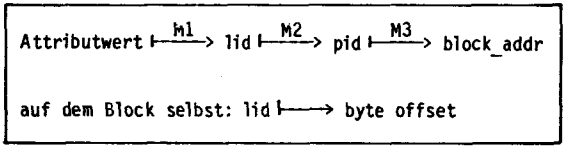
Das Problem der Abbildung {lids} ⟶ {DB-Platte}

Zur Implementierung von SI-Aufträgen der Form SEL_FILTER, GET_REC, PUT_TAB oder PUT_RECS ist es notwendig, daß die LIs die Abbildung von lids auf die entsprechende DB-Plattenaddressen kennen. Um jedoch bei DB-Plattenreorganisationen diese Abbildung nicht immer ändern zu müssen, führen wir als zusätzliche Abstraktionsebene einen physisch-orientierten Identifikator, genannt pid, ein.
Mit pids wollen wir (logische) Seiten identifizieren. Seiten müssen somit noch auf (physische) Blöcke abgebildet werden. Blöcke sind die kleinsten Einheiten des Lese-/Schreibzugriffs auf die DB-Platten.

Eigenschaften: pids sind

. eindeutig für eine Seite, jedoch nicht notwendigerweise eindeutig für ein Tupel (falls mehrere Tupel auf eine Seite passen).
. wiederverwendbar

Die Wiederverwendbarkeitseigenschaft bei pids ist natürlich unverzichtbar, da wir ansonsten bei Löschungen (logischen) Speicherplatz nicht wiederverwenden könnten. Die pids werden von den LIs vergeben; genaueres dazu wird in Kap. 3.3.3 gegeben.
Die Abbildung eines Tupels auf seine DB-Plattenadresse sieht bei Verwendung eines Indexes nun wie folgt aus:



Eigenschaften der Abbildungen M1, M2, M3:

- M1 ist invariant, Implementierung erfolgt mittels dichtem Index.
- Invarianz von M2 ist wünschenswert, wenn wir in den invertierten Listen Einträge der Form (attributvalue, (lid,pid)) wählen.
- M3 muß selbstverständlich variant sein, um DB-Plattenreorganisationen zu ermöglichen.

Die Abbildung lid ⟼ byte offset im Block selbst ermöglicht der LI:

- Kompaktifizierungen
- Aufsammeln von lids bei EXH_FILTER-Operationen.

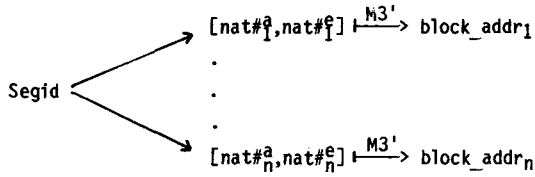
Effizienzbetrachtungen für diese 3-stufige Abbildung

(1) Die Kosten für die Abb. M1 hat man in jedem DB-System, welches als Zugriffspfade invertierte Listen anbietet.

(2) Die Abbildung M3: {pid} ⟶ {block_addr} ist wesentlich billiger als in DB-Systemen, welche als Verwaltungseinheit zur Speicherzuordnung eine einzelne Seite verwenden. Der Grund für diese Effizienz ist die bereichsmäßige Organisation von Segmenten.
Die Abbildung M3 kann wie folgt realisiert werden:
(a) Konstruktion von pids:

pid = Segid.nat#

(b) Jede LI verwaltet für ihre Segmente jeweils eine Segmentabbildungstabelle. Die Abbildung eines Segments Segid, welches n Bereiche umfaßt, auf seinen zugeordneten DB-Plattenspeicherplatz sieht dann folgendermaßen aus:



Dabei gilt:

- $[nat\#^a_i,nat\#^e_i]$ ist ein abgeschlossenes Intervall natürlicher Zahlen, für alle i=1,...n.
- $[nat\#^a_i,nat\#^e_i]$ ist disjunkt zu $[nat\#^a_j,nat\#^e_j]$, für $i \neq j$, i,j e {1,...,n}.
- block_addr$_i$ ist die Adresse des ersten Blocks in Bereich$_i$, Bereich$_i$ enthält $nat\#^e_i - nat\#^a_i + 1$ Blöcke; i e {1,...,n}.
- Die Abbildung M3' ist nicht invariant.

Bei dieser Organisation ist sofort ersichtlich:

(a) Diese Segmentabbildungen sind sehr klein, da wir stets die Anzahl der Bereiche eines Segments minimieren wollen. Sie können somit resistent im (kleinen) LI-Arbeitsspeicher gehalten werden.

(b) Die Abbildung {pid} ⟶ {block_addr} ist effizient berechenbar: Für pid$_1$ = Segid.1 bestimmt die LI dasjenige Intervall aus der Segmentabbildungstabelle für Segid, für das 1 e $[nat\#^a_i,nat\#^e_i]$ gilt. Der gesuchte Block ist dann der $1 - nat\#^a_i + 1$.te Block im Bereich$_i$, welcher bei block_addr$_i$ beginnt.

(3) Der einzig wirklich teure Overhead bei der vorgeschlagenen dreistufigen Tupeladressierung stellt die Abb.M2:{lid} ⟶ {pid} dar. Diese Abbildung muß von den LIs verwaltet werden, da die pid-Vergabe Aufgabe der LIs ist. Abbildung M2 ist sehr umfangreich und kann somit nicht vollständig im LI-Arbeitsspeicher gehalten werden. Selbst wenn wir die Abbildung als ('löchrige'[58]) Tabelle organisieren, ist zur Ermittlung eines pids für einen vorgegebenen lid manchmal ein Plattenzugriff nötig.

[58] aufgrund der Nichtwiederverwendbarkeitseigenschaft der lids.

Dieser Overhead kann minimiert werden, wenn die DB-Platten über Festkopfzylinder verfügen.[59] Falls man jedoch nicht bereit ist, diesen Overhead in Kauf zu nehmen, so kann man ihn auf folgende Weise 'wegoptimieren':

Optimierung der Abb. M2: {lid} ⟶ {pid}.

Ausgehend von der Überlegung, daß pids von der LI benötigt werden zur Ausführung der Operatoren SEL_FILTER, GET_REC, PUT_TAB sowie PUT_RECS, erhält man folgendes Optimierungsverfahren:

(i) (lid,pid)-Einträge in invertierten Listen.
(ii) (lid,pid) als Knotenverweise in Indexen.
(iii) (lid,pid) als Teil jedes Tupels in Update-C oder Public-C.

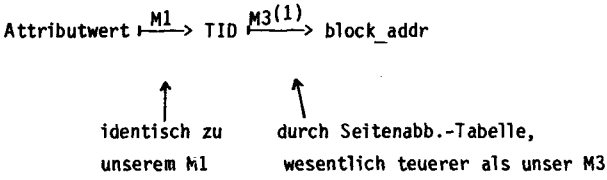
Mit dieser Technik des 'Umherschickens' von Teilen der Abbildung {lid} ⟶ {pid} vermeiden wir offensichtlich den Overhead des Zugriffs auf diese Abbildung durch die LI. Ein weiterer Vorteil besteht darin, daß Tupelidlisten für SEL_FILTER-Aufträge bereits vom QM nach pids sortiert werden können. Anzumerken in diesem Zusammenhang ist noch die wichtige Eigenschaft der Invarianz von Abb. M2. Falls dies gewährleistet ist[60] , repräsentieren (lid,pid)-Teile immer die richtige Abbildung und sind nicht nur 'probable position pointers' (wie im System UDS [HAER77]).
Insgesamt betrachtet können wir somit feststellen, daß unser Tupeladressierungsmechanismus den Ansprüchen der losen Kopplung und einer hohen Effizienz genügt. Dieser Effizienzanspruch läßt sich auf mit bekannten Lösungen aus der Praxis untermauern:

---

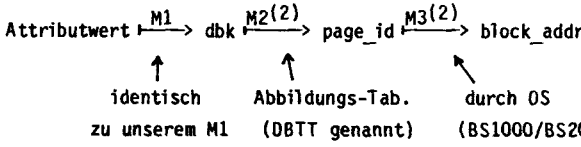
[59] Noch besser wäre ein spezieller Hardware-Adreßtranslationsmodul für {lid} ⟶ {pid}.

[60] Wie, wird im anschließenden Kapitel über DB-Plattenreorganisation erläutert.

SystemR:



System UDS: (Siemens Codasyl-DBMS)
Jeder DB-record erhält einen sogenannten database key (dbk). Ein dbk hat dieselben Eigenschaften wie ein lid, die Motivation für die Einführung von dbks war aber mit anderen Absichten verknüpft[61] als die Einführung von lids.



Ein page_id entspricht in etwa einem pid.

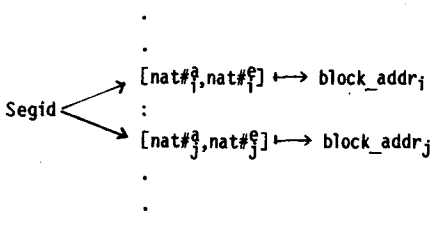
3.3.2.2 DB-Plattenreorganisation.

Ein weiterer wichtiger Gesichtspunkt für die Einführung der pids als Abstraktion von physischen Blöcken ist die Fähigkeit der LIs zu autonomen, vom Host unabhängigen DB-Plattenreorganisationen.
Der vorgeschlagene Adressierungsmechanismus läßt folgende solche DB-Plattenreorganisationen zu:
(1) Verschieben von Bereichen: (zur Beschleunigung von Filteroperationen)

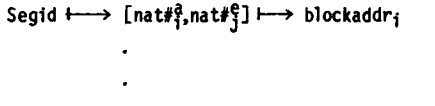
[61] Dbks in Codasyl-Systemen sind an der Benutzerschnittstelle sichtbar (lids nicht) und können vom Benutzer bei späteren DB-Aufträgen direkt verwendet werden (zur Optimierung der Abbildung von DB-records auf die Blockadressen).

Da die Abbildungen M3' in den Segmentabb.-Tabellen nicht invariant sind, können Bereiche auf den DB-Platten verschoben werden.

(2) Vereinigung von Bereichen: (zur Verkürzung von Segmenttabellen)
Betrachten wir folgenden Ausschnitt aus einer Segmentabb.-Tabelle:



Nehmen wir weiter an, daß Bereich$_i$ und Bereich$_j$ physisch-sequentiell zusammenhängen, d.h. der letzte Block von Bereich$_i$ ist der direkt physisch-sequentielle Vorgänger von Block block_addr$_j$.
Falls nun zusätzlich nat#$_i^e$ + 1 = nat#$_j^a$ gilt, so kann man obige Abbildung vereinfachen zu:



Mit diesen beiden LI-lokalen Operationen läßt sich nun eine einfache Reorganisations-Strategie zur Minimierung der Anzahl von Bereichen angeben. Die Absicht hinter einer solchen DB-Reorganisation ist natürlich, eine Reduzierung der Ausführungszeiten für Filteroperationen zu erreichen.

Reorganisations-Verfahren:

Betrachten wir ein Segment Segid, welches u.a. die zwei Bereiche extent$_i$ und extent$_j$ enthält.
Periodisch kann die LI überprüfen, ob folgendes Verfahren durchführbar ist:

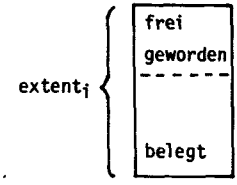
(1) Falls extent$_i$ und extent$_j$ nicht physisch-sequentiell zusammenhängen, verschiebe extent$_i$ physisch-sequentiell zusammenhängend an extent$_j$,

falls dies möglich ist.

(2) Wenn (1) erfolgreich ist, prüfe ob $extent_i$ und $extent_j$ vereinigt werden können.

Diese einfache Reorganisationsmöglichkeit[62] sollte in der Praxis im Normalfall ausreichend sein, um zu gewährleisten, daß jedes Segment aus sehr wenigen Bereichen besteht (vorausgesetzt einer guten a priori Schätzung bei der Segmentkreierung). Weitergehende Reorganisationen würden nur bei solchen exzessiven Löschaktivitäten notwendig werden, wo sich diese Löschungen gleichförmig über alle Bereiche verteilen würden und diese entstehenden Lücken nicht durch neue Einfügungen aufgefüllt werden könnten (schrumpfender Datenbestand). Einerseits sind jedoch schrumpfende Datenbestände in praktischen DBs sehr unwahrscheinlich. Andererseits ist eine Gleichverteilung der Löschungen bei unserer 'entry-sequenced'-Organisation i.a. auch nicht zu befürchten, da diese der chronologischen Organisation Rechnung trägt; d.h. Löschungen sind im Bereich mit den ältesten Daten am wahrscheinlichsten.
Um jedoch noch über weitere Reorganisationsmöglichkeiten zu verfügen ohne die Invarianz der Abbildung M2: {lid} ⟶ {pid} durch Umspeicherungen aufgeben zu müssen, bieten sich bei Bedarf folgende Maßnahmen an:

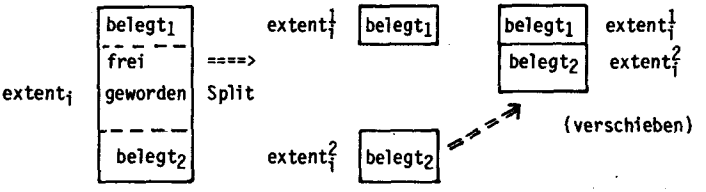
(1)



In diesem Fall kann ein Bereich am Anfang (analog am Ende) verkürzt werden. Die danach nichtbenutzten pids muß die LI für eine Wiederverwendung merken.

62) - für die bestimmt interessante Optimierungsalgorithmen bzgl. der Wahl von i und j existieren -

(2) Splitting eines Bereichs bei 'großer Lücke' in der Mitte, mit evt. nachfolgender Verschiebung



3.3.3. Lösungen für das Einfüge-Problem.

Die Realisierung einer losen Kopplung zwischen LIs und CM wollen wir zuerst einmal ohne Berücksichtigung der Optimierungsvorschläge für Abb. M2 demonstrieren.
Sei T eine Transaktion, welche ein neues Tupel in eine Benutzerrelation einfügen möchte.

Einfüge-Scenario für die N.A.:

(E1) UM vergibt neuen logischen Identifikator lid_t für das einzufügende Tupel. Aufgrund automatischer Indexwartung vergibt der SAM neuen logischen Identifikator lid_n für einen evt. einzufügenden Indexknoten.
(E2) CM vollzieht Commit-Vorgang.
(E3) Bei Bedarf schickt CM PUT_TAB/PUT_RECS-Aufträge an die betreffende LI zum Durchschreiben von lid_t und lid_n.
(E4) Die betreffende LI fügt neues Tupel/Indexknoten sicher und wiederholbar ein.[63] Dabei werden jetzt erst die entsprechenden pids (und damit auch Plattenspeicher) vergeben.

Wie man leicht erkennt, sind mit diesem Verfahren die gestellten Forderungen zur Erreichung einer losen Kopplung erfüllt:

[63] Mögliches Verfahren für sicheren Update von LI-Verwaltungs-/Abbildungs-Information: 2 Versionen auf DB-Platte, Verwendung der Ping-Pong-Methode. (Für Benutzerdaten wird MINISAFE-Verfahren verwendet).

- (LK1) durch lids.
- (LK2) durch (E2), (E3) unter Berücksichtigung der wichtigen DB-Cache/-SAFE-Eigenschaft, daß nur gültige Objektversionen nach den LIs durchgeschrieben werden.
- (LK3) durch pid-Vergabe erst in (E4).

Kopplungseigenschaften dieses Verfahrens

(1) Einfügungen können im Host vorbereitet und gültig gemacht werden ohne Kommunikation mit den LIs.
(2) Transaktions- oder Systemfehler des Host betreffen weder die DB-Platten-Konsistenz noch die Korrektheit der LI-privaten Systemdaten (Segmentabb.-Tabellen mit Freispeicherinfo, {lid} → {pid}-Tabelle).
(3) Ein LI-Crash ist auch unproblematisch, da die LI-Operatoren atomar und wiederholbar sind und die LIs nicht ins Transaktionskonzept eingebunden sind. Stürzt etwa eine LI während der Bearbeitung eines Durchschreibe-Auftrags ab, so kann der CM nach erfolgreichem Hochfahren dieser LIs die noch nicht quittierten Aufträge wiederholen.
(4) Ein gleichzeitiger Zusammenbruch von Backends und Host ist auch unproblematisch, die Recoverymaßnahmen sind eine Kombination aus (2) und (3).

Somit ist eine lose Kopplung zwischen LIs und CM in dem diskutierten Fall (Einfügung von Benutzertupeln) erreicht. Als Spezialfall für die N.A. müssen wir noch Einfüge-Transaktionen untersuchen, welche 'CREATE RELATION...'/'CREATE INDEX...' -DDL-Anweisungen enthalten.
Solche Transaktionen sind Einfüge-Transaktionen in zweierlei Hinsicht:
(a) Neue Systemtupel müssen in Systemrelationen eingefügt werden.
Das ist analog dem geschildertem Einfüge-Scenario durchzuführen.
(b) DB-Plattenspeicherplatz wird reserviert infolge eines 'CREATE_SEG(<Relid/Indid>,<space_spec>)-Auftrags an eine LI.

Das neue Problem aufgrund (b) ist nun, daß ein CREATE_SEG-Auftrag ausgeführt werden muß, bevor die ersten Durchschreibe-Aufträge für neu eingefügte Objekte (bzgl. der betreffenden Relation/Index) vom CM an die LI erteilt werden. Andererseits darf natürlich CREATE_SEG auch nicht vor Transaktions-Commit vom CM an die LI erteilt werden, weil dadurch das Prinzip der losen Kopplung verletzt werden würde.

Konsequenz:

(1) In COMMIT-Aufträgen an den CM sind sogenannte Post-Commit-Aufträge für LIs mitzuliefern. Diese Post-Commit-Aufträge enthalten CREATE_SEG oder DROP_SEG-Kommandos für LIs, sie sind im Zuge des COMMIT-Operators ebenfalls auf dem SAFE zu sichern.
(2) Nach erfolgreicher Sicherung auf dem SAFE (Commit-Punkt!) erteilt der CM vorhandene Post-Commit-Aufträge an LIs.
(3) Nach Erhalt positiver Bestätigungen durch die betr. LIs ist der COMMIT-Auftrag beendet und kann an den LM bestätigt werden.

Dadurch ist auch hier das Prinzip der losen Kopplung gewahrt. Da solche Transaktionen sehr selten sind, wird die Effizienz des DB-Cache/SAFE-Verfahrens durch die Post-Commit-Aufträge auch nicht berührt.

Abschließend zu diesem Problemkreis wollen wir die Auswirkungen diskutieren, welche die vorgeschlagenen Optimierungen für Abb. M2 haben.
Durch die Aufnahme von pids in Indexen ((lid_s,pid_t) als Einträge in invertierten Listen, (lid_n,pid_n) als Knotenverweise) ergibt sich folgende Unannehmlichkeit bei Einfügungen: Aufgrund unseres Einfüge-Scenarios kann im Schritt (E1) nur $(lid_t,pid_t$='unknown') bzw. $(lid_n,pid_n$='unknown') im Indexknoten eingetragen werden. Diese 'unknown'-pids müssen später durch die tatsächlichen pids ausgetauscht werden, will man den expliziten Zugriff auf die {lid} ⟶ {pid}-Tabelle bei nachfolgenden Zugriffen auf solche neueingefügten Objekte vermeiden. Bei einem solchen Austauschverfahren (z.B. 'lazy evaluation': Austausch erst bei erstmaliger Wiederbenutzung) sind zwar die Backends weiterhin nicht in das Transaktionskonzept eingebunden, jedoch

wird dadurch die Software-Komplexität des DBMS erhöht, sodaß man die Effizienzgewinne durch die Optimierung für die Abb. M2: {lid} ⟶ {pid} sorgfältig gegen diese erhöhte Systemkomplexität abwägen sollte.

Schlußbemerkung:
Nachdem wir nun die Recovery- und Concurrency-Probleme auf den LI-Systemdaten zufriedenstellend gelöst haben, bleibt uns nur noch das Recovery- und Concurrency-Problem bei der lid-Vergabe. Dafür lassen sich sehr effiziente Lösungen angeben, da für lids nicht das Bedürfnis einer Wiederverwendbarkeit nach Transaktions/Hostsystemfehler besteht, und ebenfalls nicht so strenge Konsistenzbedingungen wie bei Synchronisation von Zugriffen auf andere DB-Objekte angelegt werden brauchen.

3.4. Überlegungen zur Prozeßorganisation.

Um unsere DBMS-Architektur auf einer konkreten Rechenanlage implementieren zu können, müssen die Benutzer- und DBMS-Programme in die Betriebssystemumgebung des Host eingebettet werden. Wie aus der diskutierten Modularisierung ersichtlich ist, haben wir das DBMS als vom Betriebssystem unabhängiges Anwendungsprogramm konzipiert, welches spezielle Funktionen wie Parallelitätskontrolle für DB-Objekte (LM), DB-Pufferverwaltung/Recovery (CM) und DB-Plattenverwaltung (LIs) selbst erledigt.
Bei einer Realisierung unseres Host-DBMS werden nur noch eine Reihe einfacher Funktionen des Betriebssystems ausgenützt, wie z.B.

(OS1) Verwaltung des virtuellen Arbeitsspeichers (DB-Cache wird jedoch von CM verwaltet!),

(OS2) Kontrolle der Terminal-Ein/Ausgabe (falls nicht ein eigener Frontend-Prozessor zur Verfügung steht),

(OS3) Synchronisations-Primitive zur Prozeß-Kommunikation,

(OS4) Prozeßverwaltung und -Scheduling (Parallelarbeit mehrerer Anwenderprogramme durch geeignete Betriebsmittelvergabe und Rechnerkernzuteilung).

Unter einem Prozeß verstehen wir dabei eine Programmausführung in einem Adreßraum; Prozesse sind Einheiten der Rechnerkernzuteilung. Als nächstes wollen wir untersuchen, wie die Funktionen (OS3) und (OS4) existierender ('General Purpose'-)Betriebssysteme verwendet werden können, um eine möglichst große Leistungsfähigkeit erreichen zu können.

3.4.1. Einfluß der Systemarchitektur auf Probleme der Prozeßstrukturierung.

Unter einer Prozeßstrukturierung verstehen wir eine Abbildung von Transaktionsprogrammen und DBMS-Programmen auf Betriebssystem-Prozesse. Eine solche Prozeßstrukturierung hat sich dabei an folgenden Zielsetzungen zu orientieren:

(1) Ein hohes Maß an Parallelarbeit ist bei der Bearbeitung von DB-Transaktionen anzustreben.

(2) Kommunikations- und Synchronisationsprobleme von selbständigen und gleichberechtigten Prozessen sind in befriedigender Weise zu lösen.

Zur Lösung von Prozeß-Kommunikation/Synchronisation existieren zwei grundsätzliche Philosophien:

- Botschaften-Mechanismus
- Prozedur-Mechanismus

Wie in [LAUE78] nachgewiesen wird, sind beide Lösungen konzeptuell gleichwertig, und lassen sich ineinander überführen. Die weitergehende Aussage, daß beide Verfahren auch leistungsmäßig identisch sind, wird durch die Praxis nicht bestätigt:

Die Prozeßschnittstelle vieler 'General-Purpose'-Betriebssysteme ist für typische Anwendungen (hohe Transaktionsraten, kurze Antwortzeitanforderungen) untauglich, da diese Betriebssysteme 'große' Prozesse verwalten, d.h. mit einer großen Anzahl von Zustandsinformation für Abrechnungszwecke, und komplizierte Rechnerkern-Zuteilungsstrategien (pre-emptive scheduling, time slicing) verwenden. Prozeßumschaltungen sind also sehr teuer, sie liegen in der Größenordnung von 5000-10000 Befehlen, und sollten somit soweit wie möglich minimiert werden, um einen CPU-engpaß

zu vermeiden (vgl. dazu auch [HAER78], [GRAY77], [STON81]).
Somit scheidet eine konzeptionell attraktive Prozeßstrukturierung der N.A. für solche Systemumgebungen von vornherein aus, nämlich daß wir jeden Modul auf einen separaten Prozeß abbilden (Botschaften-System, Server-Modell)[64]
Andererseits besteht auch keine vollkommene Freiheit bei der Auswahl einer geeigneten Prozeßstrukturierung aus folgendem Grund:
Die vorgenommene horizontale Zerlegung des Gesamt-DBMS in Verbindung mit dem Einsatz gewidmeter Prozessoren (Host-DBMS auf ein oder mehreren eigenen Prozessoren, jede LI auf einem eigenen Prozessor) bewirkt, daß zwischen dem Host und den LIs eine Botschaftenschnittstelle bestehen muß und somit jede LI als Server-Prozeß realisiert werden muß. Somit verbleibt uns die Entscheidung über die Prozeßstrukturierung der im Host laufenden Teile des DB-Systems.
In der Praxis werden bei der Implementierung eines DBMS auf einem General-Purpose-Host fast ausschließlich folgende zwei Prozeßstrukturierungsmethoden verwendet:

Methode 1: Virtuelle Datenbankmaschinen-Architektur im Host.

Die Prozeßorganisation geschieht dabei folgendermaßen:
Jedes Benutzertransaktionsprogramm und die entsprechende Aktivierung des Host-DBMS werden auf einem Prozeß abgebildet. Bei der parallelen Ausführung von N Transaktionen existieren N Betriebssystemprozesse im Host. Jeder dieser Prozesse enthält den gesamten Host-DBMS-Funktionsumfang, d.h. pro Transaktion steht eine virtuelle DB-Maschine zur Verfügung.

Ein bekannter Vertreter für diese Prozeßorganisation ist SystemR auf dem Betriebssystem VM370 ([ASTR76]).

Methode 2: Das Host-DBMS als ein Service-Prozeß.

Bei dieser Organisation läuft das gesamte Host-DBMS in einem separaten Adreßraum ab. Bei der Parallelausführung von N Transaktionen existieren somit N+1 Betriebssystemprozesse im Host.

[64] Das entspräche der im Betriebssystem BSM ([LAGA75]) verfolgten Strategie.

Als Beispiel hierfür ist System UDS auf dem Betriebssystem BS1000 zu nennen.

Anmerkung:

Eine Variation dieser Methode besteht darin, anstelle eines zentralen Host-DBMS-Servers einen Pool von solchen Servers zur Verfügung zu stellen, wie es z.B. bei System UDS auf BS2000 realisiert wird. Diese Organisation unterscheidet sich jedoch nur noch geringfügig von Methode 1.

Eigenschaften dieser Prozeßorganisationsformen:

ad Methode 1: (virtuelles DB-Maschinenkonzept im Host)

Für eine effiziente Realisierung dieser Methode ist die Verfügbarkeit von 'reentrant programs' nötig, um gemeinsame Adreßräume für verschiedene Prozesse zu erlauben. Da die Aktivierung des DBMS über Prozeduraufrufe erfolgt, sind wirksame Speicherschutzmechanismen notwendig. Die Synchronisation der verschiedenen unabhängigen DBMS-Aktivierungen beim Zugriff auf kritische Programmteile (critial sections) und Tabellen hat mittels eines Serialisierungsmechanismus wie z.B. Monitore (oder critical regions) zu erfolgen; diese Monitor-Implementierung ist mittels Synchronisations-Primitiven des Host-Betriebssystems wie etwa Semaphore oder unteilbaren Hardwareinstruktionen (z.B. Compare&Swap) zu realisieren.

Wesentlichste Vorteile:

(M1-V1) Da Benutzertransaktion und Host-DBMS als ein Prozeß ablaufen, werden teuere Prozeßumschaltungen zwischen Benutzertransaktion und Host-DBMS vermieden. Dabei setzen wir ferner voraus, daß der Overhead zur Implementierung von gemeinsamen Host-DBMS-Code billiger ist als Prozeßumschaltungen.[65]

(M1-V2) Aufgrund der virtuellen DB-Maschinen-Eigenschaft sind DBMS-Dienstleistungen nicht zentralisiert. Dieser Vorteil kommt dann zum Tragen, wenn der Hostrechner eine Multiprozessoranlage ist.

[65] Gemeinsamer Code ist z.B. realisierbar über gemeinsam benutzbare Programmsegmente. Dabei entstehen zwar einmalige (teure) Kosten für das Einbinden eines solchen Programmsegments in einen Adreßraum, nachfolgende Zugriffe über Programmsegmenttabellen sind jedoch wesentlich billiger als Prozeßumschaltungen.

(M1-V3) Durch die Prozeßverwaltung des Betriebssystems ist es auf einfache Weise möglich, Parallelität unter den Transaktionen zu realisieren (Inter-Transaktionsparallelität).

Wesentliche Nachteile:

(M1-N1) Falls das Host-Betriebssystem vorzeitigen Rechnerkernentzug (pre-emptive scheduling) vornimmt, entsteht das Phänomen der Konvoys ([BLAS79b]), wenn einem Prozeß, der sich in einem Monitor mit sehr hoher Zugriffsfrequenz befindet, der Rechnerkern entzogen wird. Konvoys können einen verheerenden Einfluß auf die Leistungsfähigkeit des gesamten Systems ausüben.
Anmerkung: Ein interessanter Vorschlag zur Behebung dieses Dilemmas wird in [STON81] an die Adresse der BS-Konstrukteure gemacht: Einrichtung spezieller Scheduling-Klassen, denen der Rechnerkern nicht vorzeitig entzogen werden darf. Eine ähnlich wirksame Lösung bestünde z.B. darin, daß mit Eintritt in einen Monitor der Wechsel in einen besonderen Systemmodus mit Rechnerkernentzugssperre verbunden ist.

ad Methode 2: (Host-DBMS als Serverprozeß)
Hier geschieht die Kommunikation zwischen Benutzertransaktion und Host-DBMS über einen Botschaftenmechanismus. Die Synchronisation der Prozesse ist problemlos, da nur ein zentraler Host-DBMS-Prozeß existiert.

Wesentlichste Vorteile:

(M2-V1) Das Konvoy-Problem entsteht nicht.

Wesentlichste Nachteile:

(M2-N1) Falls man viele Interaktionen zwischen Benutzertransaktion und Host-DBMS hat, entsteht die Gefahr eines CPU-Engpasses aufgrund der vielen teuren Prozeßumschaltungen.

(M2-N2) Bei der reinen Server-Methode (d.h. kein Server-Pool) kann die Rechenleistung mehrerer Host-Prozessoren nicht parallel für Host-DBMS-Funktionen ausgenützt werden.

(M2-N3) Um N parallele Transaktionen effizient bedienen zu können, muß das Host-DBMS als Mehrbenutzerprozeß organisiert werden, um ein Multiplexing von Aufträgen zwecks Verringerung von E/A-Verzögerungen durchführen zu können. Das Host-DBMS muß somit sein eigenes Multitasking und Scheduling vornehmen.

Im zusammenfassenden Vergleich der beiden genannten Methoden läßt sich feststellen:
(1) Die aufgezählten Vorteile und Nachteile beider Methoden sind in etwa invers zu einander.
(2) Die Anzahl der Prozeßumschaltungen aufgrund von I/Os zu den DB-Platten ist in beiden Methoden gleich.

Für die weitere Diskussion, welchen Einfluß unsere DBMS-Architektur auf die Prozeßstrukturierung hat, wollen wir weiterhin davon ausgehen, daß Prozeßumschaltungen sehr teuer sind, d.h. wesentlich länger dauern als Prozeduraufrufe.
Vor diesem Hintergrund sind zwei Kriterien für die Beurteilung einer Prozeßstrukturierung wesentlich:
(1) Anzahl der Interaktionen zwischen Benutzertransaktion und Host-DBMS.
(2) Anzahl der Interaktionen zwischen Host-DBMS und DB-Platten.

ad (1): (Interaktion Benutzertransaktion <===> Host-DBMS)

- Das virtuelle DB-Maschinen-Konzept vermeidet jeglichen Prozeßwechsel auf dieser Ebene.
- Die Server-Methode kann zu CPU-Engpässen führen, wenn diese Schnittstelle sehr niedrig ist (z.B. 1-Tupel(record)-at-a-time wie etwa in SystemR oder UDS).

1. Haupteigenschaft der N.A.:

hohe RAI-Schnittstelle ===> diese Interaktionen sind enorm reduziert.

Diese Tatsache trifft im wesentlichen auf die mengenorientierten QM- und UM-bezogenen RAI-Operatoren zu. Für tupelorientierte RAI-Operatoren wie die Cursor-Extensions des UM bleibt der oben genannte Nachteil natürlich bestehen. Insgesamt gesehen, dürfte aufgrund der Verwendung der Server-Methode für die N.A. kein CPU-Engpaß zu befürchten sein.

ad (2): (Interaktionen Host-DBMS <===> DB-Platten)

- Bei herkömmlichen DB-Systemen mit einer sehr niedrigen Blockschnittstelle zwischen Host-DBMS und den DB-Platten ergeben sich bei beiden Methoden sehr hohe Prozeßwechselraten.

2.Haupteigenschaft der N.A.:

hohe SI-Schnittstelle
in Verbindung mit dem C-Table-Swapping-Mechanismus ======> diese Interaktionen sind enorm reduziert.

Diese Überlegungen führen uns zu folgender Einschätzung für die N.A.:

> Was den Overhead an Prozeßwechseln anbetrifft, so kommen sowohl das virtuelle DB-Maschinen-Konzept als auch die Server-Methode als Kandidaten für die Prozeßstrukturierung im Host in Frage.[66] Alles in allem schneidet das virtuelle DB-Maschinen-Konzept in diesem Punkt doch noch etwas besser ab.

Bei der Betrachtung der Möglichkeiten zur Parallelausführung von Transaktionen fällt auf, daß keine der beiden Methoden die Ausnützung von Intra-Transaktionsparallelität ohne Zusatzmaßnahmen zuläßt. Dies ist nicht verwunderlich, da sie bisher nur für DB-Systeme mit tupel-(record)weisem Zugriffssystem eingesetzt wurden, wo Parallelität innerhalb einer

[66] Eine weitere interessante Prozeßorganisation, welche wir in dieser Arbeit nicht weiterverfolgen wollen, wäre wie folgt: Host ist Mehrprozessoranlage ==> Ein Prozessor könnte für DB-Cache/SAFE-System (CM) gewidmet werden.

Transaktion uninteressant ist. Das jedoch ändert sich für die N.A. mit ihren mengenmäßigen RAI-Operatoren und Filteroperatoren grundlegend.

3.4.2. Ausnutzung von Inter- und Intra-Transaktionsparallelität.

Zunächst wollen wir die speziellen Anforderungen in der N.A. bezüglich Inter-Transaktionsparallelität diskutieren.
Dazu sei noch einmal der Vorteil (M1-V2) und der Nachteil (M2-N2) gegenübergestellt, was zu folgendem Schluß berechtigt:

Host ist M-Prozessoranlage, $M \geq 2$ $\Rightarrow$ das Konzept der virtuellen DB-Maschinen ist vorteilhafter auf der Host-Ebene.

Dieser Vorteil einer höheren potentiellen Rechenleistung für die Ausführung von Host-DBMS-Funktionen manifestiert sich jedoch erst dann, wenn der Host-DBMS-Code relativ wenige 'critical sections' enthält, an denen sich die einzelnen Prozesse (Prozessoren) mittels eines Mutual Exclusion Mechanismus synchronisieren müssen. Die Identifikation von critical sections ist für die N.A. aufgrund unserer sorgfältigen Systemstrukturierung mithilfe von Moduln (Prinzip der funktionalen sowie Datenabstraktion, keine oder sehr wenig globale Daten zwischen den Moduln) leicht möglich:
Ausgehend von der Tatsache, daß die Einrichtung von critical sections nur dann notwendig ist, wenn durch das betreffende Programmstück Betriebsmittel vergeben werden, ergibt sich aufgrund unserer Systemarchitektur:

- TM manipuliert Transaktionsdeskriptoren
- LM verwaltet Sperrinformation
- CM verwaltet DB-Cache (C-cat)

$\Rightarrow$ Teile von TM, LM und CM sind critical sections.

Für die Ausführung des Codes von QM, UM und SAM hingegen dürfte keinerlei Synchronisation notwendig sein!
Besonders wichtig erscheint uns dieses Ergebnis für die QM-Operatoren,

welche sehr rechenintensiv sind und in DB-Anwendungen mit einem sehr hohen Prozentsatz von komplexen Retrieval-Anfragen sehr oft ausgeführt werden müssen.

Fazit: Die Rechenleistung mehrerer Host-Prozessoren kann in der N.A. ausgezeichnet für ein hohes Maß an echter Inter-Transaktionsparallelität ausgenützt werden. Deshalb ist das virtuelle DB-Maschinenkonzept in einer solchen Hardware-Umgebung dem Server-Modell auf der Host-Ebene vorzuziehen.

Bedeutung der Intra-Transaktionsparallelität für die N.A.:

Schon bei der Entscheidungsbegründung für die Wahl der RAI-Schnittstelle (in Kap. 3.2.2) haben wir den Aspekt der Parallelverarbeitung innerhalb einer Transaktion betont:
Der in der Auswertung von Operatorbäumen inherente Parallelismus kann mit unseren Hardware-Resourcen gut ausgenützt werden.
Theoretisch könnte bei M_H Host-Prozessoren und $M_{\#LI}$ LI-Prozessoren eine Transaktion gleichzeitig durch $M_H+M_{\#LI}$ Prozessoren bearbeitet werden. Bei der Prozeßorganisation mittels virtueller DB-Maschinen wird jedoch eine Transaktion mit ihrem virtuellen Host-DBMS auf einen Prozeß abgebildet, welcher auf einem Host-Prozessor abläuft.

Grundsätzlich stehen für die Parallelarbeit innerhalb einer Transaktion bei den genannten Voraussetzungen zur Verfügung:

- 1 Host-Prozessor[67)]
- bis zu $M_{\#LI}$ LI-Prozessoren

67) Das gilt genau dann, wenn der Host-Prozeß im Host selbst keine Unterprozesse kreiert, wie es für die Implementierung von asynchronen Prozeduraufrufen (fork...join, vgl. z.B. [LAUE78]) nötig ist.

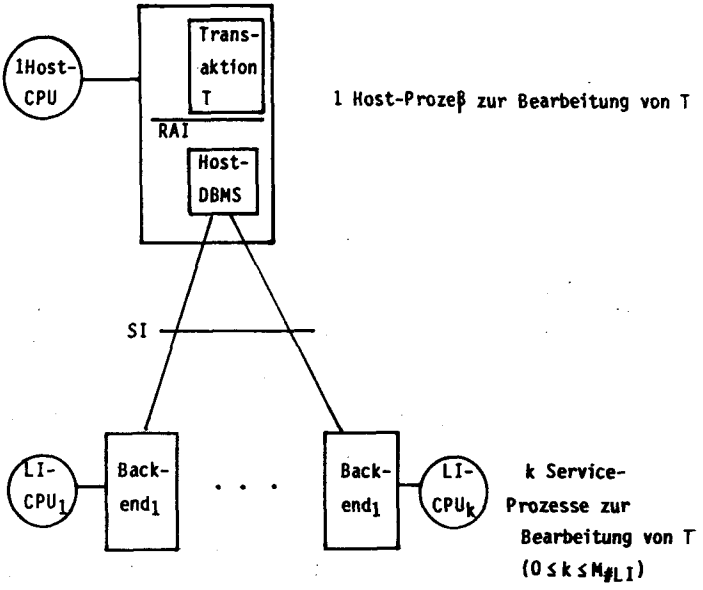


Abb. 3.9.: Prozessorhierarchie zur Parallelverarbeitung innerhalb einer Transaktion.

Als nächstes soll der Frage nachgegangen werden, in welchen Fällen die Ausnutzung von Intra-Transaktionsparallelität wirklich sinnvoll ist. Prinzipiell kann Parallelverarbeitung im Host innerhalb einer Transaktion T immer dann stattfinden, wenn der Host-Prozeß von T die I/O-Zeiten der aufgrund von SI-Aufträgen für T tätigen LI-Prozesse auszunutzen vermag. In anderen Worten, der Host-Prozeß von T muß in der Lage sein weitere Berechnungen für T ausführen zu können, ohne auf die Antworten der betreffenden LI-Prozesse warten zu müssen. Unsere Aufgabe besteht somit darin, den Arbeitsumfang zu ermitteln, welcher für T von den einzelnen Host-DBMS-Moduln parallel zur Bearbeitung von LI-Operatoren für T erledigt werden kann. Diese Frage läßt sich jedoch aufgrund der Modularisierung der N.A. sehr schnell und leicht beantworten.

Ausnutzung von Bearbeitungszeiten für SI-Operatoren:

(1) GET_REC-Aufträge:
Diese Aufträge zur Bereitstellung eines Systemtupel oder Indexknoten im Public-C werden vom CM aufgrund entsprechende Anfragen vom SAM erteilt (bei Public-C Faults). Die Arbeitsweise des SAM ist jedoch nicht mengenorientiert, sondern tupel/knotenweise.
====> Der SAM kann wenig tun bis der betreffende GET_REC-Auftrag erledigt ist.

(2) Filteraufträge EXH_FILTER, SEL_FILTER:
Diese Aufräge resultieren aus den Aufträgen COMPLETE_RELSCAN bzw. SELECTIVE_RELSCAN an den TM. Der TM selbst kann die LI-Bearbeitungszeiten dieser Filteraufträge in vielen Fällen sehr gut für die Verteilung von weiteren TM-Operatoren an solche Moduln (wie z.B. den QM) ausnutzen, welche alle benötigten Operanden bereits im DB-Cache zur Verfügung haben.

Beispiel:

Gegeben seien drei Relationen Ri, i=1,2,3.
Ri sei auf den DB-Platten von $Backend_i$ abgespeichert, i=1,2,3.
Betrachten wir nun folgenden RAI-Operatorbaum (Kurzbezeichnung: CRELS für COMPLETE_RELSCAN):

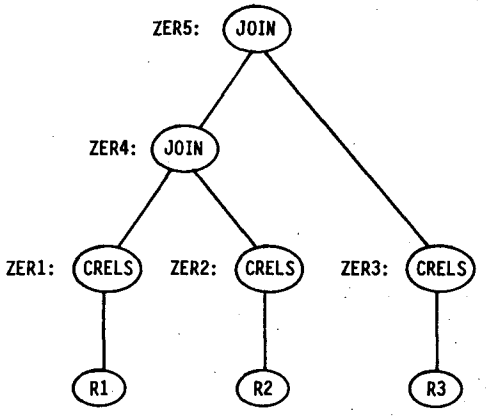


Abb. 3.10: Beispiel eines RAI-Operatorbaumes.

Die Knotenmarkierungen bezeichnen dabei die resultierenden ZERs.
Für dieses Beispiel ist folgende Parallelauswertung erstrebenswert:

1: ZER1, ZER2 und ZER3 können parallel von LI_1, LI_2 bzw. LI_3 berechnet werden (CRELS bewirkt einen Aufruf von EXH_FILTER an LI_i).
2: Wenn ZER1 und ZER2 Im DB-Cache vollständig zur Verfügung stehen, kann ZER4 durch QM im Host berechnet werden.
3: Wenn ZER4 und ZER5 im DB-Cache vollständig zur Verfügung stehen, kann ZER5 durch QM im Host berechnet werden.

Die Parallelauswertung eines RAI-Operatorbaumes geschieht somit nach dem Datenfluß-Prinzip: Operatoren sind dann ausführbereit, wenn die benötigten Operanden zur Verfügung stehen. Einheiten des Datenflusses sind dabei die ZERs.

Anmerkung: Die erhöhte Intra-Transaktionsparallelität kann zu einem erhöhten DB-Cache-Platzbedarf führen, im angegeben Beispiel für ZER1, ZER2 und ZER3 gleichzeitig. Die daraus resultierenden interessanten Optimierungsprobleme (Bestimmung eines optimalen Grads der Intra-Transaktionsparallelität bei gegebenem vorhandenem DB-Cache-Platz) werden jedoch in dieser Arbeit nicht analysiert.

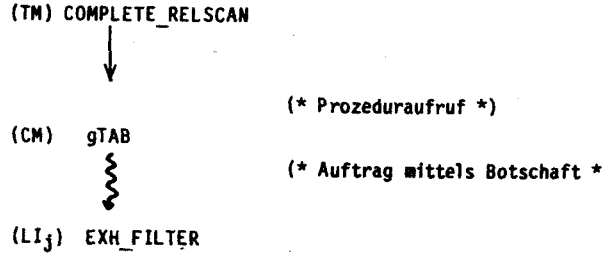
Vorschlag zur Prozeßsynchronisation Host-Prozeß ⟷ LI-Prozesse.

Bei der Prozeßsychronisation zwischen einem Host-Prozeß und ein oder mehreren LI-Prozessoren, welche alle dieselbe Transaktion T bearbeiten, müssen wir folglich differenzieren, ob die LI-Bearbeitungszeiten vom Host-Prozeß ausnutzbar sind oder nicht.

(1) Host-Prozeß gibt GET_REC-Aufrag an eine LI.
    ====> dieser I/O-Wunsch wird als synchroner DB-I/O vom Hint behandelt, d.h. Transaktion T wird die Host-CPU entzogen und an einen anderen rechenwilligen Host-Prozeß zugeteilt.
(2) Host-Prozeß gibt EXH_FILTER/SEL_FILTER-Auftrag an eine LI
    ====> dieser I/O-Wunsch wird als asynchroner DB-I/O vom Hint behandelt, d.h. T wird nicht unterbrochen und kann weiterrechnen.

Bei der Implementierung eines solchen asynchronen Verhaltens zwischen einem Host-Prozeß und LI-Prozessoren muß man dafür Sorge tragen, daß man nicht mit der Prozedur-Struktur innerhalb des Host-DBMS in Konflikt gerät.
Zur Verdeutlichung betrachten wir dazu den Ablauf einer Ausführung des COMPLETE_RELSCAN-Operators:

Aufrufstruktur:



Da der EXH_FILTER-Auftrag über eine Botschaft an eine LI_j erteilt werden muß, muß -um ein asynchrones Verhalten zu erzielen- der Aufruf der Prozedur gTAB sofort nach Absenden dieser Botschaft über den Hint terminieren (wonach die Prozedur COMPLETE_RELSCAN auch terminiert), damit gleich anschließend durch den Host-Prozeß weitere Operationen für dieselbe Transaktion ausgeführt werden können. Das Absetzen von asynchronen Aufträgen an die LIs verhält sich somit wie ein sofort beendeter schlichter Prozeduraufruf.

Nach Beendigung des C-Table-Swappings durch eine LI darf dieser Host-Prozeß jedoch nicht sofort über dieses Ereignis informiert werden, da er sich an beliebiger Stelle im Host-DBMS befinden kann. Die Modul-Struktur des Host-DBMS in Verbindung verlangt vielmehr folgendes Vorgehen:

- Bei Beendigung einer EXH_FILTER/SEL_FILTER-Operation wird dieses Ereignis dem Hint gemeldet und dort vermerkt (dazu ist i.a. eine gemeinsame Ergebnisvariable mit unteilbarer Lese- und Schreiboperation, z.B. spezielles Register, im Hint erforderlich).
- Das Host-DBMS erkundigt sich selbsttätig beim Hint über beendigte EXH_FILTER/SEL_FILTER-Operationen, wenn die entsprechenden C-Tabellen benötigt werden.

Die Implementierung dieser asynchronen DB-I/Os kann mit Standardmethoden der Systemprogrammierung (START/WAIT) realisiert werden.

Dazu ist eine Aufspaltung der bisherigen CM-bezogenen Operatoren des Moduls TM erforderlich:

COMPLETE_RELSCAN wird aufgeteilt in

. START_COMPLETE_RELSCAN (<Relid>,<restr_predicate>,<attr_list>,
<exp.size>,<ZER-name>)

. WAIT (<ZER-name>)

Analog wird mit SELECTIVE_RELSCAN verfahren.

Der Operator WAIT ist als zusätzlicher Operator an der RAI-Schnittstelle zur Verfügung zu stellen.

Mit dieser Erweiterung zur Realisierung von Intra-Transaktions-parallelität könnte das vom Transaktions-Compiler Tr-Comp erzeugte

Programm zur Auswertung des RAI-Operatorbaumes aus Abb. 3.10 wie folgt aussehen:

```
.
.
.
START_COMPLETE_RELSCAN (R1,...,...,...,ZER1);
START_COMPLETE_RELSCAN (R2,...,...,...,ZER2);
START_COMPLETE_RELSCAN (R3,...,...,...,ZER3);
WAIT (ZER1,ZER2);              (*verallgemeinertes konjunktives WAIT*)
ZER4:= JOIN (ZER1,ZER2,....);
WAIT (ZER3);
ZER5:= JOIN (ZER4,ZER3,....);
 .
 .
 .
```

Abschlußbemerkung:

Dieses Kapitel 3.4 hatte nicht die Absicht einer erschöpfenden Diskussion dieses komplexen und noch weitgehend unerforschten Gebiets der Wahl einer optimalen Prozeßstrukturierung für eine bestimmte Systemkonfiguration und DBMS-Architektur, sondern sollte erste Anregungen in diese Richtung geben. Mehr formale und quantitative Analysen (z.B. exakte Identifizierung von critical sections, Zählung von Prozeßwechseln/Botschaften, Ermittlung aller möglichen Parallelabläufe beim Dataflow-Approach,...) hätten den zeitlichen Rahmen dieser Arbeit überschritten.

4. Optimale Queryauswertung für unsere Architektur.

4.1. Auswahl geeigneter Queryoptimierungs-Strategien.

High-level Querysprachen wie SQL erlauben es dem Benutzer, Queries zu formulieren, welche bei einer unüberlegten Implementierung einen sehr großen Zeitaufwand (mehrere Stunden, sogar Tage) für die Ausführung benötigen würden. Deshalb ist der Einsatz eines Optimierers vor der Ausführung unverzichtbar, um die Ausführungszeiten mindestens auf die Größenordnungen zu reduzieren, welche für prozedurale Querysprachen in hierarchischen oder Codasyl-DBs erzielbar sind.
Als erstes müssen wir uns die Frage beantworten, wo wir Optimierungen vornehmen können. Dazu betrachten wir den Übersetzungsvorgang von der High-level Querysprache in die Operatoren der RAI-Schnittstelle zum Relationalen Zugriffssystem, welches letztendlich die Übersetzte Query ausführen soll. Der Übersetzungsvorgang wird vom Modul Tr-Comp des Transaktions-Systems erbracht, welcher in die Komponenten Pre-Compiler und Optimizer strukturiert ist.

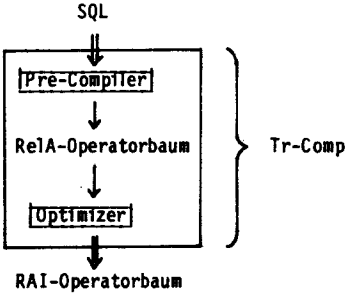


Abb.4.1: Grobablauf einer Queryübersetzung.

Als nächstes ist zu klären, was der Gegenstand der Optimierung sein soll.

Dem Optimizer fallen zwei Aufgaben zu, nämlich

(1) Algebraische Optimierung:
Eingabe: RelA-Operatorbaum T
Ausgabe: 'verbesserter' RelA-Operatorbaum T'

(2) Zugriffspfad-/Algorithmenauswahl:
Eingabe: T'
Ausgabe: 'optimierter' RAI-Operatorbaum T"

Dieser Aufteilung liegen folgende Prinzipien zugrunde:

ad(1) Die algebraische Optimierung ist eine High-level Optimierungstechnik, welche meist auf der Verwendung von Heuristiken basiert.

ad(2) Die Zugriffspfad-/Algorithmenauswahl ist eine Medium-level Optimierungstechnik, welche vornehmlich auf der Verwendung konkreter Kostenfunktionen basiert.

Ergänzend sei vermerkt, daß sich Low-level Optimierungstechniken etwa mit Organisationsstrukturen für Relationen und Indexe, mit physikalischen Größen wie z.B. Blockgröße, etc. befassen. Selbstverständlicherweise sollen solche Kenngrößen als Parameter in die Kostenfunktionen für die Medium-level Optimierung eingehen (vgl. dazu Kap.5).

4.1.1. Allgemeines über Queryoptimierungs-Strategien.

Als Grundlage für die sich anschließenden Diskussionen über gängige Optimierungs-Strategien geben wir einige bekannte Eigenschaften im Überblick an. Elementare Begriffe wie Relation und Tupel werden als bekannt vorausgesetzt (siehe z.B. [ULLM80]). Der Wertebereich eines Attributs r einer Relation R sei mit dom(R,r) bezeichnet.
Als Bezeichnungen für RelA-Operatoren verwenden wir:

- Projektion $\pi_{r_1 \ldots r_m}(R)$
  $r_1,\ldots,r_m$ sind dabei Attribute der Relation R.

- Restriktion $\sigma_F(R)$

  Dabei ist F eine Restriktionsformel, welche aufgebaut ist aus

  (i) Operanden, welche Konstanten oder Attribute von R sind,

  (ii) den arithmetischen Vergleichsoperatoren $<,=,>,\leq,\neq,\geq$,

  (iii) den logischen Operatoren $\wedge, \vee$, 'NOT'.

  Beispiel: $dom(R,r_i) = \mathbb{N}_0$, $i = 1,2,3,4$,

  $F = ((r_1<6) \vee (r_2>r_3)) \wedge (r_4=9)$

- θ-Join $R \,|X|_{r\theta s}\, S$

  Dabei ist θ ein arithmetischer Vergleichsoperator, r ist Attribut von R, s Attribut von S.

  Der θ-Join kann wie folgt definiert werden (RxS bezeichnet das cartesische von R mit S):

  $$R \,|X|_{r\theta s}\, S := \sigma_{r\theta s}(RxS)$$

  Im Falle von $\theta \equiv =$ sprechen wir von einem Equi-Join. Falls bei einem Equi-Join ein redundantes Joinattribut gestrichen wird, so ergibt sich der Natural Join.

(Anm.: Die restlichen RelA-Operatoren wie die Standard-Mengenoperatoren oder die Division werden in unseren weiteren Untersuchungen nicht benötigt.)

Algebraische Optimierungstechniken.

Grundlage für die gängigsten Optimierungsheuristiken auf diesem Gebiet bilden einige einfache Gesetze der RelA, welche wir jetzt auflisten wollen (vgl. dazu [ULLM80]). Zwei Ausdrücke E_1 und E_2 der RelA bezeichnen wir als äquivalent ($E_1 \equiv E_2$), wenn die beiden Auswertungen von E_1 bzw. E_2 dasselbe Ergebnis liefern.

Bem.:Die nachfolgenden Gesetze sind als Transformationsregeln von links nach rechts zu lesen.

(G1a) $\pi_{r_1}(\pi_{r_2}(\ldots(\pi_{r_m}(R))\ldots)) \equiv \pi_{r_1\ldots r_m}(R)$

(G1b) $\sigma_{F_1}(\sigma_{F_2}(R)) \equiv \sigma_{F_1 \wedge F_2}(R)$

(G1c) $\sigma_F(\pi_{r_1\ldots r_m}(R) \equiv \pi_{r_1\ldots r_m}(\sigma_F(R))$

(G2a) Sei $F = F_R \wedge F_S \wedge F_{RS}$ eine Restriktionsformel auf R und S, wobei gilt:
- F_R (F_S) betrifft nur Attribute von R (S).
- F_{RS} betrifft Attribute von R und S.

$$\sigma_F(\ R \underset{r\theta s}{|X|}\ S\ ) \equiv \sigma_{F_{RS}}(\sigma_{F_R}(R) \underset{r\theta s}{|X|}\ \sigma_{F_S}(S))$$

(G2b) Seien r, $r_1,\ldots,r_n$ Attribute von R; s, $s_1,\ldots,s_m$ Attribute von S.

$$\pi_{r_1\ldots r_n r s_1\ldots s_m s}(\ R \underset{r\theta s}{|X|}\ S\ ) \equiv \pi_{r_1\ldots r_n r}(R) \underset{r\theta s}{|X|}\ \pi_{s_1\ldots s_m s}(S)$$

Natürlich existieren noch viele andere solcher einfachen Äquivalenzen, jedoch genügen als Motivation für unsere Zwecke die soeben aufgeführten Regeln. Aus diesen Gleichungen lassen sich zwei Optimierungsheuristiken ableiten:

- Pipelining von Projektionen und Restriktionen.
  Relevante Gleichungen: (G1a)-(G1c)
  Zweck:
  Durch die Kombination von Sequenzen von Projektionen und Restriktionen soll eine Reduktion der Anzahl und Größe von Zwischenergebnisrelationen erzielt werden; gleichfalls wird eine Reduzierung der Sortiervorgänge für Duplikateliminationen (bei Projektionen) oder für eine Vorbereitung nachfolgender Joins bewirkt.
- Restriktionen und Projektionen so früh wie möglich.
  Relevante Gleichungen: (G2a)-(G2b)

Zweck:
Diese Verschiebung von Restriktionen und Projektionen zu den Blättern des Operatorbaums (so weit wie möglich) beabsichtigt eine Reduktion der Größe der Joinoperanden.

In allen Arbeiten über Queryoptimierung in der Literatur werden diese beiden Heuristiken vorgeschlagen, vgl. etwa [SELI79], [SMIT75], [WONG76][68] . Neben diesen offensichtlichen Heuristiken werden in der Literatur noch kompliziertere algebraische Optimierungstechniken vorgeschlagen, die auf nicht-trivialen Äquivalenzen zwischen RelA-Ausdrücken beruhen, vgl. z.B. [HALL76], [ULLM80], [WONG76]. Erwähnenswert sind ferner Techniken zur Erkennung identischer Teilbäume ([HALL76],[FINK82]). Diese Techniken sind jedoch nicht Gegenstand dieser Arbeit.

Optimierungen auf der Zugriffspfadebene.
Die üblicherweise in der Literatur vorgeschlagenen Medium-level Optimierungen, basierend auf der Minimierung konkreter Kostenfunktionen, umfassen in etwa folgende Aspekte:

- Zugriffspfadauswahl.
  Existieren mehrere Zugriffspfade zur Auswertung eines RelA-Operators (z.B. Segment-Scan, clustered Index-Scan und nonclustered Index-Scan für $\sigma_F(R)$ in SystemR), dann ist der billigste zu bestimmen.
- Beste Sortierordnung.
  Im Zusammenhang mit der Zugriffspfadauswahl steht auch die Frage nach der besten Sortierordnung bei der Auswertung einer Teilquery (siehe z.B. [SMIT75]). Geeignete Sortierordnungen von Zwischenergebnissen können vorteilhaft sein für eine effiziente Auswertung einiger RelA-Operatoren, z.B. für Joins, Projektionen oder die Implementierung der GROUP BY Anweisungen in SQL. Auch können existierende Sortierordnungen Einfluß auf die Frage haben, wie weit Projektionen im Operatorbaum nach unten verschoben werden sollen. Da eine effiziente Auswertung von Projektionen i.a. eine Sortierung nach einem Projektionsattribut voraussetzt, kann es in manchen DB-Architekturen sinnvoll

[68] Ausnahmefälle werden in [YA078] erwähnt.

sein, Projektionen, welche erst eine externe Sortierung erfordern, möglichst spät auszuführen.

- Algorithmenauswahl.
  Für die Implementierung eines Zugriffsoperators stehen manchmal mehrere Möglichkeiten zur Auswahl, z.B. für den Join der Nested-Loops Algorithmus und der Sort-Merge Algorithmus. Unter Berücksichtigung existierender Zugriffspfade ist dann der billigste zu wählen (vgl. z.B. die verschiedenen Algorithmen für Restriktion-Projektion-Join Queries in [BLAS76], [YA079]).
- Existenz identischer Teilbäume.
  Falls ein Operatorbaum identische Teilbäume aufweist, dann kann es vorteilhaft sein, das betreffende Ergebnis nur einmal auszuwerten.
- Ausnutzung von Kommutativität und Distributivität.
  Die Auswertungsreihenfolge bei Sequenzen von Joins oder Mengenoperationen kann unter Zuhilfenahme von Kenngrößen wie etwa der Kardinalität der Operanden optimiert werden (siehe z.B. [SELI79]).
- Vorverarbeitung von Relationen.
  Die effiziente Auswertung von Queries hängt natürlich von einer geeigneten Plattenorganisation (sortiert,clustered,...) der Relationen und von existierenden schnellen Zugriffspfaden (Indexe,Links,...) ab. In manchen DB-Architekturen kann es vorteilhaft sein, bei Bedarf Relationen extern zu sortieren oder temporäre Indexe einzurichten.

4.1.2. Sinnvolle Strategien in unserer Architektur.

Die für die Queryoptimierung wesentlichsten Aspekte unserer DB-Architektur sollen noch einmal aufgezählt werden.

- Eigenständige Intelligenz LI_j in jedem Backend-Subsystem zur Auswertung 'linearer' Operationen.
- Die große DB-Cache-Kapazität ermöglicht eine effiziente mengenorientierte Verarbeitungsweise von RelA-Operatoren.
- Die LIs entlasten die Host-CPU(s) so sehr, daß wir voraussetzen, daß der Host nicht CPU-bound ist.

Aufgrund dieser Eigenschaften erscheinen folgende Optimierungs-Strategien

als sinnvoll:

(1) High-level Optimierung:
Die frühzeitige Auswertung von Projektionen ist offensichtlich eine gute Strategie besonders in unserer Architektur, da die LIs partielle Projektionen i.a. mit voller Plattengeschwindigkeit ausführen können; dasselbe Argument gilt für Restriktionen. Aufgrund der LI-Datenfilterung erreicht man damit eine frühzeitige Größenreduktion der in das DB-Cache zu transportierenden ZERs. Um die Anzahl von ZERs zu reduzieren, ist es natürlich auch wichtig, Pipelining von Restriktionen und Projektionen vorzunehmen (vgl. dazu auch die Spezifikation der SI-Schnittstelle in Kap. 3.2.4.1).
Somit halten wir an zwei Heuristiken fest:
* Restriktionen und Projektionen so früh wie möglich.
* Pipelining von Restriktionen und Projektionen.
Infolge dieser frühzeitigen Datenfilterung und im Hinblick auf spezielle Auswertungsalgorthmen in Kap.4.2, welche eine weitere frühzeitige Größenreduktion von ZERs erzielen, sowie eingedenk einer sehr großen DB-Cache-Kapazität (im MByte-Bereich) können wir davon ausgehen, daß im Normalfall von den LIs gelieferte ZERs vollständig in das DB-Cache passen. Da ferner n.V. unsere Architektur keinen Host-CPU-Engpaß aufweist ('Offloading'-Effekt der LIs), können wir es uns im postulierten Normalfall immer leisten, Duplikateliminationen in ZERs mittels internem Sortieren vorzunehmen. Hier zeigt sich ein wesentlicher Vorteil gegenüber einem konventionellen DB-System, wo infolge fehlender Datenfilterung meist ein sehr teures externes Sortieren notwendig wird und man deswegen Sortiervorgänge nach Möglichkeit vermeidet.

(2) Medium-level Optimierung:
Das Problem einer besten Sortierordnung ist für uns nicht mehr so wichtig, da im Normalfall immer internes Sortieren vorausgesetzt wird. Deshalb wenden wir folgende Heuristik an:
* DB-Cache-Objekte werden sortiert, wenn es für die nachfolgende Operation vorteilhaft ist (z.B. für nachfolgende Mengenoperationen

(Durchschnitt, Vereinigung,...) sowie Joins auf ZERs).
Die Algorithmenauswahl für die Implementierung des (cache-internen) Joinoperators fällt uns demnach auch leicht.
* Der JOIN-Operator wird als Merge-Join implementiert.
Die beste Lösung für das Problem der Auswertung identischer Teilbäume ist jedoch in unserer Architektur auch nicht offensichtlich. Es kann nämlich der Fall eintreten, daß zwar die für eine Operation benötigten ZERs (2 Eingabe-ZERs, 1 Ergebnis-ZER) solange wie für diese Operation benötigt im DB-Cache gehalten werden können, daß jedoch die längere Aufbewahrung einer ZER (welche das Ergebnis eines identischen Teilbaums enthält) im DB-Cache zu Platzproblemen führt. Falls eine solche ZER aus dem DB-Cache verdrängt werden muß, ist abzuwägen, ob sie irgendwo zwischengespeichert werden oder bei der nächsten Referenzierung neu berechnet werden soll.

4.2. Effiziente Queryauswertungsalgorithmen.

4.2.1. Dynamische Filter.

Wie schon wiederholt erläutert, ist eine sehr wesentliche Voraussetzung für eine hohe Leistungsfähigkeit unserer DB-Architektur, daß die für RelA- (bzw. RAI-) Operatoren benötigten Operanden so lange wie benötigt vollständig in das DB-Cache passen, um eine effiziente Mengenverarbeitung ohne Zuhilfenahme von temporären Plattendateien möglich zu machen. Folglich muß unser Ziel beim Entwurf von Auswertungsstrategien und -algorithmen darin liegen, den Datenumfang zu minimieren, welcher von den LIs in das DB-Cache transportiert werden muß, d.h. es ist eine bestmögliche Reduktion der Kardinalität von ZERs so früh wie möglich anzustreben. Einen Schritt in diese Richtung stellt die gewählte High-level Strategie 'Projektionen und Restriktionen so früh wie möglich' dar. Damit sind die Möglichkeiten in dieser Hinsicht noch nicht erschöpft. Vielmehr ist die Auswertung gewisser Querytypen in unserer DB-Architektur sehr viel effizienter durchführbar als in

den bekannten DBMS.
Vorbereitend für die in diesem Abschnitt zu entwickelnden Konzepte führen wir erst einige abkürzende Bezeichnungen ein.

Auswertung von RelA-Ausdrücken der Form $\pi_{r_1 \ldots r_k}\sigma_F(R)$:

In unserer Architektur existieren für die Auswertung dieses Ausdrucks zwei Typen von Zugriffspfaden, nämlich

(i) Indexverarbeitung mit anschließender selektiver Segmentfilterung (vgl. SEL_FILTER-Operator),

(ii) vollständige Segmentfilterung (vgl. EXH_FILTER-Operator).

Wir geben nun die zugehörigen Algorithmen ALG1 und ALG2 zur Auswertung von $\pi_{r_1 \ldots r_k}\sigma_F(R)$ im Detail an und legen eine Kurznotation für spätere Zwecke fest. ALG1 und ALG2 sind äqivalente Algorithmen zur Berechnung der Projektion-Restriktion Query $\pi_{r_1 \ldots r_k}\sigma_F(R)$.

ALG1: (Indexverarbeitung & selektive Segmentfilterung)

Voraussetzung:

F ist zerlegbar in $F = F^+ \wedge F^-$. F^+ betrifft die Attribute $r_{i_1},\ldots,r_{i_m}$ von R, alle r_{i_j} $(1 \leq j \leq m)$ sind invertiert und F^+ ist über diese Indexe auswertbar[69].

Die Berechnung von $\pi_{r_1 \ldots r_k}\sigma_F(R)$ erfolgt in zwei Schritten:

(a) Die Auswertung der Restriktion $\sigma_{F^+}(R)$ ist alleine durch Indexverarbeitung möglich. Dabei liefert der Modul SAM die entsprechenden Tupelidlisten L_j für Restriktionen auf den einzelnen Attributen r_{i_j}, $1 \leq j \leq m$. Der QM manipuliert diese Trefferlisten entspr. weiter mittels Durchschnitts- oder Vereinigungsoperationen. Das erarbeitete Endergebnis Id_R ist jedoch keine materialisierte ZER, sondern eine Tupelidliste der Tupel von R, welche die Restriktion F^+ erfüllen.

Notation für all diese Schritte unter (a):

[69] D.h., F^+ enthält nur Terme der Form '$r_{i_j}\theta const_{i_j}$'. Außerdem muß der Index eine sequentielle Verarbeitungsmöglichkeit bieten (wie das z.B. bei B-Bäumen der Fall ist).

(Op1) $Id_R := IND_{F^+}(R)$ (*INDexauswertung*)

(b) Die Materialisierung der durch Id_R qualifizierten Treffertupel unter Anwendung der verbleibenden Restriktion F^- geschieht wie folgt:
- Sortiere Id_R in geeigneter Weise nach pids[70] .
- Eine selektive Segmentfilterung von R gegen F^- mit gleichzeitiger partieller Projektion durch eine LI (vgl. LI-Operation SEL_FILTER) liefert eine ZER im DB-Cache.
- Diese ZER wird vom QM in geeigneter Weise sortiert (wenn nötig); dabei werden vorhandene Duplikate eliminiert. Das Endergebnis heiße ZER_R.

Notation für all diese Schritte unter (b):

(Op2) $ZER_R := SEL_{\pi_{r_1 \ldots r_k} \sigma_{F^-}}(Id_R)$

ENDALG1

ALG2: (Vollständige Segmentfilterung)

Voraussetzungen: keine.

Die Auswertung von $\pi_{r_1 \ldots r_k} \sigma_F(R)$ geschieht wie folgt:

(a) Eine vollständige Segmentfilterung von R gegen F mt gleichzeitiger partieller Projektion nach $r_1, \ldots, r_k$ mittels des LI-Operators EXH_FILTER liefert eine ZER_R im DB-Cache.

(b) Diese ZER_R wird dann vom QM (wenn nötig) in geeigneter Weise sortiert; dabei werden gegebenenfalls vorhandene Duplikate eliminiert. Das Endergebnis heiße ZER_R.

Notation für alle Schritte (a) und (b):

(Op3) $ZER_R := EXH_{\pi_{r_1 \ldots r_k} \sigma_F}(R)$

ENDALG2

[70] Die Sortierung erfolgt durch den QM im DB-Cache, falls wir die vorgeschlagene Optimierung von pids in Indexen verwenden; andernfalls erfolgt die Sortierung durch die betreffende LI.

Vergleichend seien in Kurznotation die beiden prinzipiellen Zugriffspfadtypen zur Berechnung von $\pi_{r_1 \ldots r_k}\sigma_F(R)$ gegenübergestellt:

ALG1: $Id_R := IND_{F^+}(R)$; (* $F = F^+ \wedge F^-$ *)
$ZER_R := SEL_{\pi_{r_1 \ldots r_k}\sigma_{F^-}}(Id_R)$

ALG2: $ZER_R := EXH_{\pi_{r_1 \ldots r_k}\sigma_F}(R)$

Alternativ schreiben wir für ALG1 auch:
ALG1: $ZER_R := SEL_{\pi_{r_1 \ldots r_k}\sigma_{F^-}}(IND_{F^+}(R))$

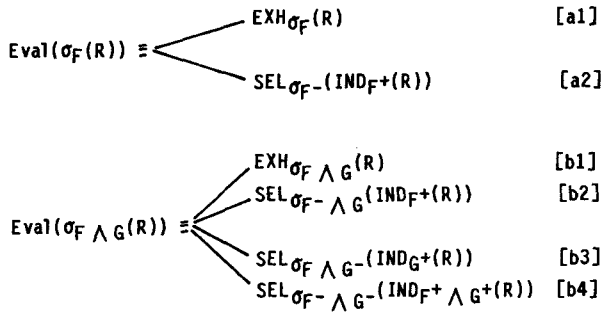
Weitere Bezeichnungen:
- Mit $Eval(\pi_{r_1 \ldots r_k}\sigma_F(R))$ bezeichnen wir die Auswertung von $\pi_{r_1 \ldots r_k}\sigma_F(R)$ mittels ALG1 oder ALG2.
- mincost(RelA-Ausdruck) sei eine Funktion, welche die minimalen Kosten für die Auswertung des RelA-Ausdrucks angibt.
  cost(AP) sei eine Funktion, welche die LI-Kosten der Auswertung für einen konkreten Zugriffspfad AP angibt.

Die Grundlage für die im folgenden vorzustellenden Konzepte bildet das anschließenden Lemma.

Lemma 4.1:
Seien F und G Restriktionsformeln auf einer permanenten Relation R derart, daß die betreffende LI das Prädikat $F \wedge G$ in dieser Form gegen R im Fluge filtern kann. Dann gilt:

$$mincost(\sigma_{F \wedge G}(R)) \leq mincost(\sigma_F(R))$$

Beweis:
Seien $F = F^+ \wedge F^-$, $G = G^+ \wedge G^-$ Zerlegungen von F bzw. G derart, daß F^+ bzw. G^+ über Indexe auswertbar sind. Unsere Architektur bietet folgende Auswertungsmöglichkeiten für $\sigma_F(R)$ und $\sigma_{F \wedge G}(R)$ an:



Weiterhin gilt:
F ∧ G in dieser Form durch LI im Fluge berechenbar ⟶
F, F⁻ ∧ G, F ∧ G⁻, F⁻ ∧ G⁻ sind auch im Flug durch LI auswertbar, da diese Restriktionsformeln eine geringere Komplexität als F ∧ G aufweisen.
Somit ergibt sich: cost([a1]) = cost([b1]),
cost([a2]) = cost([b2])
D.h. aber: $mincost(\sigma_{F \wedge G}) = \min\{mincost(\sigma_F(R)), cost([b3]), cost([b4])\} \leq mincost(\sigma_F(R))$.

Interpretation:
Querymodifikation von Restriktionsformeln durch Hinzufügen von weiteren Konjunktionen unter Bewahrung der Im-Flug-Verarbeitung durch die LIs bewirkt, daß
(1) die optimalen Auswertungskosten einer solchen Restriktion reduziert (oder höchstens nicht erhöht) werden, und
(2) die Kardinalität der aus der Auswertung resultierenden ZER im DB-Cache weiter reduziert wird.

Der wichtige Punkt bei Lemma 4.1 ist die Bewahrung der Im-Flug-Verarbeitung durch die LIs. Andernfalls könnte der unter (2) genannte erstrebenswerte Vorteil durch zu hohe LI-Kosten zunichte gemacht werden. Die nun zu beantwortende Frage ist, wie man zu geeigneten Prädikaten für die Querymodifikation kommt.

Dynamische Filter.

Informell stellen dynamische Filter Prädikate dar, welche während der Auswertung einer Query von schon berechneten Teilresultaten gewonnen werden; sie werden zur dynamischen Querymodifikation für die Berechnung weiterer Zwischenergebnisresultate verwendet. Mittels dyn. Filter soll die in das DB-Cache zu transportierende Informationsmenge weitestgehend reduziert werden, jedoch ist dabei auf die Im-Flug-Verarbeitung durch die LIs zu achten.

Definition 4.1:

(a) X sei Teilmenge von dom(R,r). Ein Filter für X ist ein Prädikat $F_X[r]$ in der Variablen r mit folgender Eigenschaft:

$$\forall\ x \in dom(R,r) \text{ gilt: } x \in X \longrightarrow F_X[r](x)$$

(b) Ein Filter $F_X[r]$ für X heißt dynamischer Filter, wenn X Teilmenge von $\pi_r(R)$ ist.

Ein dyn. Filter charakterisiert somit die aktuell in der DB vorhandenen Werte eines Attributs einer Relation.
(Ein nicht-dyn. Filter hingegen ist nicht an die aktuell existierenden Werte gebunden, er ist somit statisch angebbar ('statischer Filter')[71] .

Beispiel 4.1:

[71] Ein statischer Filter für X mit $\pi_r(R) \subseteq X$ kann auch zur Formulierung und Überprüfung von Integritätsbedingungen auf r herangezogen werden; ebenso kann er bei der Queryoptimierung vor Beginn der Abarbeitung einer Query zur Querymodifikation verwendet werden (z.B. 'Schrankenfilter' aus Kataloginformation).

Angenommen, im DB-Cache steht folgende ZER_R zur Verfügung:

ZER_R	r_1	r_2
	1	10
	2	17
	3	3
	3	19
	800	6
	816	8
	1500	5
	1502	6

Sei $X = \pi_{r_1}(ZER_R) = \{1,2,3,800,816,1500,1502\}$.

Beispiele dyn. Filter für X sind nun:

$F_X^1[r_1] = '1 \leq r_1 \leq 1502'$

$F_X^2[r_1] = '(1 \leq r_1 \leq 3) \vee (800 \leq r_1 \leq 816) \vee (1500 \leq r_1 \leq 1502)'$

$F_X^3[r_1] = '(r_1=1) \vee (r_1=2) \vee (r_1=3) \vee (r_1=800) \vee (r_1=816) \vee (r_1=1500) \vee (r_1=1502)'$

Fundamentale Eigenschaft von Filtern:

Sei $x \in dom(R,r)$, $F_X[r]$ ein Filter für X. Dann ist unmittelbar aus Def. 4.1(a) ersichtlich:

* $F_X[r](x) =$ <u>false</u> $\longrightarrow x \notin X$
* Für $F_X[r](x) =$ <u>true</u> ist i.a. keine definitive Aussage möglich, ob $x \in X$ oder $x \notin X$ gilt.

Im Licht der Queryauswertung kann man diese beiden Eigenschaften wie folgt interpretieren:

Angenommen, es ist bereits ein dyn. Filter $F_X[r]$ ermittelt worden und es muß im Zuge einer in Bearbeitung befindlichen Query folgende weitere Aufgabe gelöst werden:

"Bestimme alle Tupel einer Relation S, für deren Wert vs eines Attributs s von S mit $dom(R,r)=dom(S,s)$ gilt, daß $vs \in X$."

Die Verwendung von $F_X[r]$ zur Lösung dieses Problems bewirkt nun:

(1) Im Falle $F_X[r](vs)=$<u>false</u> erhält man eine definitive <u>Nein-Antwort</u> auf die Frage $vs \in X$?, d.h. die betreffenden Tupel von S können

weggefiltert werden.

(2) Im Falle $F_X[r](vs)$=true erhält man nur eine Vielleicht-Antwort auf die Frage $vs \in X$?, d.h. die betreffenden Tupel von S können nicht weggefiltert werden und es ist zusätzliche Arbeit für eine definitive Beantwortung erforderlich.

Durch den Einsatz eines dyn. Filters kann somit ein gewisser Prozentsatz von nicht benötigter Information (Nein-Antworten) rechtzeitig weggefiltert werden. Bei einem zu groben Filter jedoch könnte möglicherweise viel Information in das DB-Cache aufgrund von Vielleicht-Antworten transportiert werden, für die sich bei einer nachträglichen Prüfung herausstellt, daß sie für die Beantwortung der vorliegenden Query überflüssig ist. Folglich benötigen wir eine Differenzierung verschiedener dyn. Filter für X hinsichtlich ihres Filterungs-Wirkungsgrads.

Definition 4.2:
Seien $F^1_X[r]$ und $F^2_X[r]$ zwei dyn. Filter für X.
$F^2_X[r]$ ist selektiver als $F^1_X[r]$:iff $F^2_X[r] \longrightarrow F^1_X[r]$

Diese suggestive Bezeichnung bezieht ihre Berechtigung davon, daß das stärkere Prädikat $F^2_X[r]$ mindestens soviel Nein-Antworten liefert wie $F^1_X[r]$ (vgl. auch Def. 4.1).
In Beispiel 4.1 sieht man:
$F^3_X[r_1]$ ist selektiver als $F^2_X[r_1]$, welcher wiederum selektiver als $F^1_X[r_1]$ ist.
(z.B.: $F^3_X[r_1](810)$=false, aber $F^2_X[r_1](810)$=true;
$F^2_X[r_1](100)$=false, aber $F^1_X[r_1](100)$=true.)

Definition 4.3:
Sei $F_X[r]$ ein dyn. Filter für X.
(a) $F_X[r]$ heißt trivialer Filter, wenn gilt:
$\forall\ vr \in dom(R,r)$: $F_X[r](vr)$=true
(b) $F_X[r]$ heißt totaler Filter, wenn

$$F_X[r] = \text{'} \bigvee_{vr \in X} (r=vr) \text{'}$$

(c) $F_X[r]$ heißt Min-Max Filter, wenn

$$F_X[r] = \text{'} \min\{vr \mid vr \in X\} \leq r \leq \max\{vr \mid vr \in X\} \text{'}$$

(dabei ist eine Ordnung $\leq$ auf dom(R,r) vorausgesetzt)

In Beispiel 4.1 ist $F_X^1[r_1]$ ein Min-Max Filter, $F_X^3[r_1]$ ein totaler Filter. ($F_X^2[r_1]$ könnte als iterierter Min-Max Filter bezeichnet werden.)

Lemma 4.2:
Sei $F_X^{triv}[r]$ trivialer Filter für X, $F_X^{tot}[r]$ totaler Filter für X, dann gilt:

$\forall$ dyn. Filter $F_X[r]$ für X:

$$F_X^{tot}[r] \longrightarrow F_X[r] \longrightarrow F_X^{triv}[r]$$

Beweis:
Sei $vr \in dom(R,r)$.

- $F_X^{tot}[r](vr)$=true iff $\exists\ x \in X$: $vr=x$ iff $vr \in X$
  Vielleicht-Antworten sind für totale Filter somit stets definitive Ja-Antworten.
- $\forall\ vr \in dom(R,r)$: $F_X^{triv}[r](vr)$=true, d.h. der triviale Filter liefert nur Vielleicht-Antworten.

Der totale Filter für X ist somit selektiver als alle anderen dyn. Filter für X. ($F_X^{tot}[r]$ ist feinster Filter für X.)
Der triviale Filter für X ist weniger selektiv als alle anderen dyn. Filter für X. ($F_X^{triv}[r]$ ist gröbster Filter für X.)[72]

[72] Die Menge $MF_X[r]$ aller dyn. Filter für X bildet einen Verband ($MF_X[r], \longrightarrow$); $F_X^{tot}[r]$ ist kleinstes Element, $F_X^{triv}[r]$ ist größtes Element, kleinstes gemeinsames Oberelement zu $F_X^1[r]$ und $F_X^2[r]$ ist $F_X^1[r] \vee F_X^2[r]$, größtes gemeinsames Unterelement ist $F_X^1[r] \wedge F_X^2[r]$.

Berechnung dynamischer Filter.

Wie aus der Def. eines dyn. Filters ersichtlich ist, kann ein solcher Filter erst während der Abarbeitung einer Query berechnet werden. Die Berechnung von totalen sowie Min-Max Filtern ist in unserer mengenorientierten Architektur sehr effizient möglich. Generell bieten sich zwei Möglichkeiten für die Ermittlung eines dyn. Filters $F_X[r]$ im DB-Cache an:

(a) Von einer ZER, welche von Relation R abgeleitet ist und das Attribut r von R beinhaltet.

(b) Von einer Tupelidliste, welche mittels eines Indexes auf r berechnet wurde und welche zusätzlich die jeweiligen Atributwerte für r enthält.

Die ZER sowie die Tupelidliste befindet sich im Normalfall vollständig im DB-Cache, in beiden Fällen liegt i.a. eine Sortierung nach den Attributwerten für r vor.[73] Somit können totale sowie Min-Max Filter quasi kostenlos ermittelt werden. Natürlich ist neben totalen sowie Min-Max Filtern noch eine Vielfalt von unterschiedlich selektiven Filtertypen denkbar. Dieses interessante Teilproblem, allgemeine Konstruktionsverfahren für praktisch sinnvolle Filter zu entwickeln sowie deren Analyse bzgl. effizienter Ermittlung/Anwendung und Filterungs-Wirkungsgrad, kann im Rahmen dieser Breitbandarbeit nicht weiter verfolgt werden und bleibt somit späteren Untersuchungen vorbehalten.

Auswahl eines geeigneten dyn. Filtertyps.

Da dyn. Filter zur Querymodifikation (z.B. durch konjunktives Anfügen an eine bereits vorhandene Restriktion) eingesetzt werden sollen, ist es natürlich rein theoretisch vorteilhafter, immer den selektivsten Filter, d.h. den totalen Filter zu verwenden, da auf diese Weise eine bestmögliche Größenreduktion von ZERs vor Transport in das DB-Cache zu erzielen ist. Falls jedoch der somit zu modifizierende RelA-Ausdruck durch LI-Filteroperationen (insbesondere bei EXH_FILTER) ausgewertet werden

[73] Im Falle einer Restriktion der Form $const_1 \leq r \leq const_2$ wird die Tupelidliste bereits sortiert nach r-Werten erstellt. Falls bei der ZER vorher Duplikateliminination erforderlich war, ist hierfür zweckmäßigerweise die Sortierung nach r-Werten vorzunehmen.

soll, treten aufgrund der Im-Flug-Bedingung praktische Beschränkungen hinsichtlich der Komplexität dyn. Filter auf (vgl. dazu Lemma 4.1). Deshalb ist bei der Auswahl eines dyn. Filtertyps sorgfältig darauf zu achten, daß diese LI-spezifische Im-Flug-Filterungseigenschaft nicht verletzt wird. Für Min-Max Filter wird dies i.a. gewährleistet sein, für einen totalen Filter $F_X^{tot}[r]$ in der angegebenen Darstellung jedoch nur bei einigermaßen geringer Kardinalität von X. Falls sich jedoch für totale Filter eine kompakte Darstellung angeben läßt, so ist die Im-Flug-Bedingung nicht an kleine Mengen X gekoppelt.

Man betrachte etwa folgende Menge X, welche 4999 Elemente enthält:

$X = \{x \in \mathbb{N}_0 : (1 \leq x \leq 10000) \wedge (x \text{ ist gerade}) \wedge (x \neq 2346)\}$

Falls diese Darstellung von X bekannt ist, dann kann man auch für $F_X^{tot}[r]$ diese äquivalente Darstellung wählen, welche eine Im-Flug-Filterung i.a. ermöglichen sollte (4 Tests pro Attributwert für r je betrachtetes Tupel).

Falls solche kompakte Darstellungen einer umfangreicheren Menge X nicht bekannt sind bzw. nicht einfach zu ermitteln sind, so müssen weniger selektive Filtertypen verwendet werden[74] .

Aus diesen Überlegungen ist klar ersichtlich, daß man bei der Wahl eines geeigneten dyn. Filtertyps wieder einmal mit dem klassischen 'space-time tradeoff' konfrontiert ist: Ein selektiverer Filter reduziert die DB-Cache Platzanforderungen, jedoch auf Kosten einer höheren LI-Filterungszeit, falls die Im-Flug Schranken überschritten werden.

4.2.2. Kategorisierung von SQL-Queries.

Für die korrekte Anwendung der High-level Heuristik 'Projektionen und Restiktionen so früh wie möglich' ist eine Einteilung von SQL-Prädikaten in gewisse Kategorien erforderlich. Das Ziel dieser Kategorienbildung ist dabei zweierlei:

[74] Als Approximation eines totalen Filters bietet sich z.B. ein sogenannter Hash-Filter an: Sei HA[0:1] ein boolsches Feld, $H: dom(R,r) \longrightarrow \{0,\ldots,1\}$ eine (Hash-)Funktion. $F_X[r]$ ist Hash-Filter für X, wenn gilt:
(a) HA[i]=true iff $\exists\, x \in X: H(x)=i$
(b) $\forall\, x \in dom(R,r): F_X[r](x) = HA[H(x)]$.

(a) Es soll herausgefunden werden, welche Projektionen und Restriktionen vor einem Join ausgewertet werden können, d.h. im Operatorbaum nach unten verschoben werden können.

(b) Prädikate, welche vor dem Join ausgewertet werden können, sollen klassifiziert werden bezüglich spezieller Typen von Auswertungsverfahren, welche sie benötigen.

Der Grundbaustein einer SQL-Query ist der Queryblock ([CHAM76]). Ein Queryblock ist aufgebaut aus

* SELECT-Liste der gewünschten Attribute,
* FROM-Liste der Relationen, von denen Tupel geholt werden sollen,
* WHERE-Qualifikation, welche alle Ergebnistupel erfüllen müssen.[75)]

Die anschließende Kategorisierung von SQL-Queries ist in ähnlicher Form auch in [MAKI81] zu finden. Sie wird in Kap.4.2.3 bei der Beschreibung von Queryauswertungsalgorithmen Verwendung finden.

Die WHERE-Qualifikation eines Queryblocks Q besteht aus logischen Verknüpfungen (AND,OR) von Prädikaten P der Form

(i) 'r θ const'
(ii) 'r θ s'
(iii) 'r θ aggr_fct(Q_{emb})'
(iv) 'r ∈ (Q_{emb})'[76)]

wobei: r und s sind Atrribute der Relation R bzw. S, const ∈ dom(R,r);
θ ∈ {<, ≤, =, ≠, >, ≥},
Q_{emb} ist ein in Q geschachtelter Queryblock,
aggr_fct ist eine Aggregatsfunktion wie z.B. MIN, MAX, SUM, AVG.

Terminologie.
Wir sprechen davon, daß ein Prädikat P das Attribut r (bzw. r und s im Falle (ii)) direkt enthält.
Sei P ein Prädikat in der WHERE-Qualifikation eines Queryblocks Q, P sei

[75)] Die GROUP BY Anweisung wird vorerst nicht betrachtet.

[76)] Prädikate der Form '(Q_{emb1}) ⊆ (Q_{emb2})' werden von uns nicht betrachtet; ebenso 'r ∉ (Q_{emb})'.

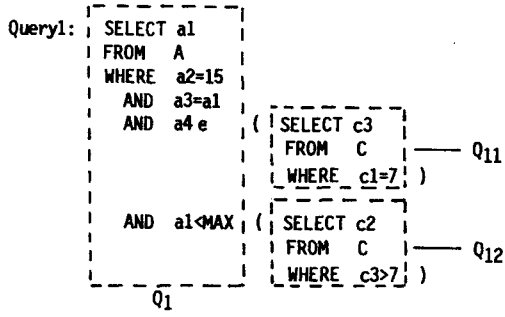
von der Form (iii) oder (iv):

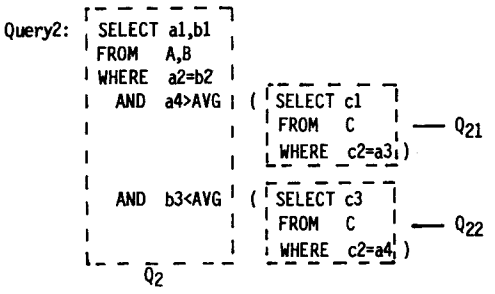
* Q_{emb} liegt direkt in P.
* Q ist direkt äußerer Block von Q_{emb}.
* Q_{emb} ist direkt innerer Block von Q.

Aufgrund der Schachtelung können Queryblöcken sogenannte Blockschachtelungstiefen (BST) zugeordnet werden:
Der äußerste Queryblock hat BST=1, ein direkt innerer Queryblock Q_2 eines Queryblocks Q_1 mit $BST(Q_1)=i$ hat $BST(Q_2)=i+1$.
Dieser Schachtelungsbegriff bewirkt eine partielle Ordnung zwischen Queryblöcken bzgl. Inklusion und kann graphisch durch einen statischen Blockstrukturbaum illustriert werden.

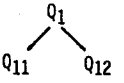
Beispiel 4.2:

Relationen : A(a1,a2,a3,a4), B(b1,b2,b3), C(c1,c2,c3)





Für Query1 gilt: $BST(Q_1)=1$, $BST(Q_{11})=BST(Q_{12})=2$.
Statischer Blockstrukturbaum:



(analog für Query2)

Definition 4.4:
Sei P Prädikat von Q, P enthalte direkt das Attribut r einer Relation R.
(a) r ist lokal zu Q :iff R ist in der FROM-Liste von Q enthalten.
(b) r ist global zu Q :iff r ist nicht lokal zu Q.
(c) Ein globales r bezieht sich auf Queryblock Q' :iff
Q' ist der nächst äußere Block von Q, welcher R in seiner FROM-Liste enthält.

Die Existenz eines globalen r in Q induziert eine Präzedenzrelation bzgl. der Auswertungsreihenfolge von Queryblöcken: Falls sich ein globales r von P in Q auf ein Q' bezieht, dann kann P nicht ausgewertet werden ohne r-Werte von Q' geliefert zu bekommen.
Die Beziehung 'liegt direkt in' läßt sich wie üblich zu ihrer transitiven Hülle erweitern:

Definition 4.5:
Sei P ein Prädikat von Q, Q' ein weiterer Queryblock.
Q' liegt in P :iff (Q' liegt direkt in P
oder

$\exists$ Q_{emb}: Q_{emb} liegt direkt in P und
$\exists$ P' in Q_{emb}: Q' liegt in P')

Definition 4.6:
Sei P ein Prädikat von Q.
P ist lokal auswertbar zu Q :iff
(es gibt kein globales r in P oder in einem in P liegenden Queryblock:
r bezieht sich auf Q' mit BST(Q')<BST(Q))

Definition 4.7:
Sei P lokal auswertbar zu Q.
P ist korreliert zu Q :iff
($\exists$ globales r in einem in P liegenden Queryblock :
r bezieht sich auf Q)

Anmerkung: Die Bezeichnung 'korreliert' ist in Anlehnung zu der bei SystemR für solche Queries verwendeten Terminologie 'correlation queries' gewählt worden.
Aus Def.4.6 und 4.7 ist sofort ablesbar:
P ist korreliert zu Q $\longrightarrow$ ($\exists$ Q_{emb} mit BST(Q_{emb})>BST(Q) :
$\exists$ P_{emb} von Q_{emb}, welches nicht lokal auswertbar ist zu Q_{emb})

In Bsp. 4.2 erkennt man:
Query1: 'a2=15', 'a3=a1' sind lokal zu Q_1 auswertbar und nicht korreliert zu Q_1.
'c1=7' ist lokal zu Q_{11} auswertbar, a4 ist lokal zu Q_1 $\longrightarrow$ 'a4 $\in$ (Q_{11})' ist lokal zu Q_1 auswertbar und nicht korreliert zu Q_1.
Ebenso ist 'a1<MAX(Q_{12})' lokal zu Q_1 auswertbar und nicht korreliert zu Q_1.
Query2: a3 ist global zu Q_{21} und bezieht sich auf Q_2 $\longrightarrow$ 'c2=a3' ist nicht lokal zu Q_{21} auswertbar, 'a4>AVG(Q_{21})' ist korreliert zu Q_2.

Ebenso ist 'c2=a4' nicht lokal zu Q_{22} auswertbar und 'b3<AVG(Q_{22})' ist korreliert zu Q_2.

Definition 4.8:
Sei P ein Prädikat von Q, P sei lokal auswertbar zu Q.

(K-1) P heißt einfache Selektion von Q, wenn gilt:
- (1.1) P enthält direkt nur lokale Attribute einer Relation R.
- (1.2) P ist nicht korreliert zu Q.
- (1.3) P enthält direkt keine Mengenoperatoren.

(K-2) P heißt einfache Mengenselektion von Q, wenn gilt:
- (2.1) wie (1.1).
- (2.2) wie (1.2).
- (2.3) P enthält direkt den Mengenoperator $\in$.

(K-3) P heißt einfache Korrelation von Q, wenn gilt:
- (3.1) wie (1.1).
- (3.2) P ist korreliert zu Q, wobei gilt:
  Alle globalen Attribute von in P liegenden Queryblöcken, welche sich auf Q beziehen, gehören zur selben Relation R wie in (3.1).

(K-4) P heißt Join-Prädikat von Q, wenn gilt:
- (4.1) P enthält direkt genau zwei lokale Attribute von verschiedene Relationen.

(K-5) P heißt komplexe Korrelation von Q, wenn gilt:
- (5.1) wie (1.1).
- (5.2) P ist korreliert zu Q, wobei gilt:
  $\exists$ globales s in einem in P liegenden Querblock, welches sich auf eine Relation S ($\neq$R in (5.1)) in der FROM-Liste von Q bezieht.

Für Bsp.4.2 erhalten wir:

Query1
- 'a2=15', 'a3=a1', 'a1<MAX(Q_{12})' sind einfache Selektionen.
- 'a4 $\in$ (Q_{11})' ist eine einfache Mengenselektion.

Query2
- 'a2=b2' ist ein Join-Prädikat.
- 'a4>AVG(Q_{21})' ist eine einfache Korrelation (a3,a4 von A).

- 'b3<AVG(Q_{22})' ist eine komplexe Korrelation (a4 von A, b3 jedoch von B).

Eigenschaften dieser Prädikat-Klassen.

(A) Ein Prädikat P der Klasse K-1 oder K-2 ist lokal auswertbar zu einem Queryblock Q, jedoch nicht korreliert zu Q

⟶ es gibt keine globalen Attribute in inneren Queryblöcken von P, welche sich auf Q beziehen

⟶ die Auswertungsreihenfolge kann von 'innen nach außen' geschehen, P braucht statisch nur einmal ausgewertet zu werden.

(B) Verhältnis einfache Korrelation zu komplexer Korrelation:
Diesen Sachverhalt wollen wir anhand Query2 aus Bsp.4.2 erläutern. Query2 läßt sich in RelA-Schreibweise wie folgt darstellen:

$$\text{Query2} \equiv \sigma_{b3<AVG(Q_{22})}(\sigma_{a4>AVG(Q_{21})}(A \;{}_{a2=b2}|X|\; B))$$

Bzgl. der am Join beteiligten Relationen A und B enthält 'a4>AVG(Q21)' nur Attribute von A (nämlich a4 und a3), 'b3<AVG(Q_{22})' enthält die Attribute a4 von A und b3 von B. Entsprechend einer früheren Transformationsregel kann somit die einfache Korrelation 'a4>AVG(Q_{21})' vor dem Join ausgewertet werden, die komplexe Korrelation 'b3<AVG(Q_{22})' jedoch erst nachher, d.h.:

$$\text{Query2} \equiv \sigma_{b3<AVG(Q_{22})}(\sigma_{a4>AVG(Q_{21})}(A) \;{}_{a2=b2}|X|\; B)$$

Mit Hilfe dieser Prädikat-Klassifizierungen lassen sich nun die folgenden Kategorien von SQL-Querytypen bilden[77] :

Definition 4.9:
Gegeben sei eine SQL-(Sub)query mit äußerstem Queryblock Q, alle Prädikate in der WHERE-Qualifikation von Q seien lokal auswertbar.
Q heißt Kategorie-l (Sub)query, wenn gilt ($l \in \{1,\ldots,5\}$):

[77] Analoge Klassifizierungen wurden bereits früher in [MAKI81] und [ASTR75] definiert.

(a) Es gibt ein Prädikat P von Q in der Klasse K-l.
(b) Kein Prädikat P' von Q ist in der Klasse K-m, $1<m \leq 5$.

Bezeichnungen:
Kategorie-1: Q_{select}, Kategorie-2: Q_{set}, Kategorie-3: Q_{corr},
Kategorie-4: Q_{join} , Kategorie-5: Q_{compl}.

4.2.3. Beschreibung der neuen Algorithmen.

Vorbemerkung: Aufgrund unserer gewählten High-level Strategie werden Projektionen immer so früh wie möglich ausgewertet. Für die anschließenden Algorithmen sind Projektionen nicht von gesondertem Interesse und werden somit nicht eigens in den zu untersuchenden Querytypen betrachtet.

Bezeichnungen:
- SELECT * in einer SQL-Query bedeutet die Auswahl irgendwelcher Attribute (vgl. obige Vorbemerkung).
- Eval(Q) bezeichnet die Auswertung der (Teil-)Query Q, das berechnete Ergebnis ist eine ZER im DB-Cache.
- Cache-interner Join:
  Wir nehmen an (Normalfall), daß beide Join-Operanden ZER_R und ZER_S sortiert nach dem jeweiligen Join-Attribut r bzw. s im DB-Cache zur Verfügung stehen. Das vom QM berechnete Ergebnis des Joins heiße ZER_{RS}.
  Notation: $ZER_{RS} := JOIN_{r\,\theta\,s}(ZER_R, ZER_S)$
  Weitere Festlegung: Für $\theta \equiv =$ sei durch JOIN der Natural Join realisiert.
- Sei $F_X[r]$ ein dyn. Filter für X, r und s Attribute der Relation R bzw. S. $F_X[r]|_s^r$ bezeichnet die Substitution aller Vorkommen von r in $F_X[r]$ durch s. Statt $F_X[r]$ schreiben wir nur noch F[r], wenn die betrachtete Menge X klar ist.

In den folgenden drei Teilkapiteln sollen nun neuartige Auswertungsalgorithmen für Queries vom Typ Q_{set}, Q_{corr} und Q_{join} präsen-

tiert werden, welche maßgeschneidert auf die Fähigkeiten unserer DB-Architektur sind und eine wesentliche Effizienzsteigerung gegenüber den bekannten Verfahren in existierenden DB-Systemen erzielen. Alle diese Verfahren basieren auf dem Prinzip der dynamischen Querymodifikation durch Einsatz dyn. Filter. Dabei werden wir auf die Angabe konkreter Filter in manchen Fällen verzichten (vgl. Diskussion über Berechnung und Auswahl dyn. Filter).
Anmerkung: Die Auswertungsalgorithmen für Q_{select}-Queries ändern sich nicht im Vergleich zu existierenden Lösungen, jedoch ändert sich die Optimierung bei der Zugriffspfadauswahl (vgl. Kap.5). Ferner sei darauf hingewiesen, daß alle anschließend präsentierten Algorithmen als Basisalgorithmen zur Illustration der Verwendung dyn. Filter dienen sollen. Mögliche Variationen werden am Ende von Kap.4.2.3 diskutiert.

4.2.3.1. Auswertungsalgorithmus für Q_{set}-Querytypen.

Als Teilmenge der Kategorie-2 Queries Q_{set} wollen wir folgendes Queryschema $\overline{Q}_{set}$ betrachten:

```
Q̄set ≡ SELECT *
       FROM   R
       WHERE GR AND r ∈ (Q^s_emb)
```

Dabei gilt:
- G_R ist eine Restriktionsformel auf R (d.h. G_R ist in der Klasse K-1).
- '$r \in (Q^s_{emb})$' ist in der Klasse K-2 $\longrightarrow$ Q^s_{emb} ist lokal auswertbar.
- r und s sind Attribute von R bzw. S, wobei dom(R,r) = dom(S,s) gilt.
- Plausiblerweise muß das Ergebnis der Auswertung von Q^s_{emb} eine ZER sein, welche aus nur einem Attribut besteht. Dieses einzige Ergebnisattribut ist s.

Desweiteren bezeichne SF[r] ein Prädikat in der Variablen r.

Auswertungsalgorithmus für $\overline{Q}_{set}$-Queries:

Algorithmus1

Schritt1: ZER_S := Eval(Q^S_{emb});
Schritt2: Berechne einen dyn. Filter F[s] für ZER_S;
SF[r] = F[s]$|^S_r$;
Schritt3: (*Ermittlung einer Obermenge des Ergebnisses für $\overline{Q}_{set}$ mittels Querymodifikation.*)
ZER_R := EVAL($\sigma_{G_R \wedge SF[r]}(R)$);
Schritt4: (*Feststellung des exakten Ergebnisses.*)
if "F[s] ist nicht der totale Filter für ZER_S"
then result $\equiv ZER'_R$:= $JOIN_{r=s}(ZER_R, ZER_S)$
else result $\equiv ZER_R$
fi;

Ende Algorithmus1

Die Korrektheit von Algorithmus1 folgt unmittelbar aus der Definition eines dyn. Filters sowie unter Berücksichtigung, daß
$JOIN_{r=s}(ZER_R, ZER_S) = \{Tupel\ t \in ZER_R: t.r \in ZER_S\}$,
wobei t.r den Wert von Attribut r für Tupel t bezeichnet. Letzteres folgt aus der speziellen Gestalt von ZER_S (nur 1 Attribut) und aus der Tatsache, daß $JOIN_{r=s}$ als Natural Join vorausgesetzt ist.
Anmerkung: Die Wahl eines geeigneten Filtertyps in Schritt2 ist offengelassen.

Beispiel 4.3:

Relationen:

A	a1	a2	a3	a4	a5
	1	2	3	4	3
	4	1	1	2	3
	2	1	2	5	5
	3	3	1	4	3
	5	4	1	2	2
	2	1	4	5	5

B	b1	b2	b3
	2	1	3
	3	2	3
	2	1	5
	5	4	4
	1	2	5
	1	1	1

Query:
```
SELECT a1,a2,a3
FROM A
WHERE a3 e ( SELECT b3
             FROM   B          )
             WHERE 2 ≤ b1 ≤ 4
```
—— Q^{b3}_{emb}

Auswertung gemäß Algorithmus1:

Schritt1: (* $ZER_B := Eval(Q^{b3}_{emb}) \equiv Eval(\pi_{b3}\sigma_{2 \le b1 \le 4}(B))$ *)

ZER_B	b3
	3
	5

Schritt2: (* Wir wählen den Min-Max Filter. *)
$SF[a3] = F^{MM}[b3]|^{b3}_{a3} = '3 \le a3 \le 5'$

Schritt3: (* $ZER_A := Eval(\pi_{a1a2a3}\sigma_{3 \le a3 \le 5}(A))$ *)

ZER_A	a1	a2	a3
	1	2	3
	2	1	4

Schritt4: (* result $\equiv ZER'_A := JOIN_{a3=b3}(ZER_A, ZER_B)$ *)

ZER'_A	a1	a2	a3
	1	2	3

Bemerkungen:

(1) Der zu erwartende Effizienzgewinn von Algorithmus1 ist im wesentlichen durch Lemma 4.1 zu begründen.

(2) In SQL kann Q^S_{emb} auch eine Liste konstanter Werte sein. In diesem Fall wird man als dyn. Filter sicherlich meistens den totalen Filter

wählen können.

4.2.3.2. Auswertungsalgorithmus für Q_{corr}-Querytypen.

Im Gegensatz zu Algorithmen für Q_{join}-Queries hat das Problem einer effizienten Auswertung von Q_{corr}-Queries in der Literatur bisher kaum Beachtung gefunden.
Als Teilmenge der Kategorie-3 Queries Q_{corr} wollen wir folgendes Queryschema $\overline{Q}_{corr}$ betrachten:

```
Q̄corr ≡ SELECT *
        FROM   R
        WHERE GR AND
              r1 θ aggr_fct(Q_emb^s1{r2=t2})
```

Dabei gilt:

- r1,r2 sind Atrribute von R, s1 (t2) ist Attribut von S (T); R ≠ T,S (T=S ist zulässig).
- θ ist ein arithmetischer Vergleichsoperator.
- aggr_fct ist eine Aggregatsfunktion, welche berechnet wird aus den Attributwerten von s1.
- Die Notation $Q_{emb}^{s1}\{r2=t2\}$ gibt an, daß Q_{emb}^{s1} das Prädikat 'r2=t2' enthält. Die Auswertung von Q_{emb}^{s1} ist wieder eine ZER, welche nur das Attribut s1 umfaßt.
- Wegen R ≠ T,S ist r2 global in $Q_{emb}^{s1}\{r2=t2\}$ und bezieht sich auf den äußeren Queryblock; 'r2=t2' ist somit nicht lokal auswertbar[78] .
- G_R ist ein Prädikat der Klasse K-1 oder K-2.

Als Illustration soll folgende Query dienen:

```
SELECT *
FROM   A
WHERE  a3 ≠ 7 AND a2<AVG( SELECT b18
```

[78] Das korrelierende Prädikat 'r2=t2' kann man in Anlehnung an den Equi-Join als Equi-Korrelation bezeichnen.

```
                    FROM   B,C
                    WHERE b3=c7 AND a1=c4 )
```

Die Eigenschaft, daß das korrelierende Prädikat 'r2=t2' nicht lokal zu Q^{sl}_{emb} auswertbar ist, impliziert, daß Attributwerte für das globale Attribut r2 vom äußeren Queryblock an Q^{sl}_{emb} geliefert werden müssen. Diese Versorgung von Q^{sl}_{emb} mit Werten für r2 erfolgt in allen bekannten DB-Systemen auf die in den meisten Fällen ineffizienteste Weise, nämlich mittels der Technik der sogenannten iterierten Tupelsubstitution[79] (siehe etwa [SELI79], [BANE78], [MAKI81]). Bei dieser Methode wird der innere Queryblock Q^{sl}_{emb} so oft ausgewertet, wie es Treffertupel gemäß G_R gibt. Unsere mengenorientierte DB-Architektur hingegen ermöglicht eine sehr viel effizientere Auswertungstechnik für solche Korrelationsqueries.

Notation für den nachfolgenden Algorithmus:

$Q^{sl}_{emb}\{r2=t2\}\,|^{'r2=t2'}_{CF[t2]}$

bedeutet, daß das Prädikat 'r2=t2' in Q^{sl}_{emb} durch das Prädikat CF[t2] ersetzt wird, d.h. wir nehmen hier eine Querymodifikation durch Prädikatsubstitution vor. Desweiteren sei auf die Semantik der GROUP BY Anweisung in SQL verwiesen[80] .

Auswertungsalgorithmus für $\overline{Q}_{corr}$-Querytypen:

Algorithmus2

Schritt1: ZER_R := Eval($\sigma_{G_R}(R)$);

Schritt2: Berechne dyn. Filter F[r2] für $\pi_{r2}(ZER_R)$;

$CF[t2] = F[r2]\,|^{r2}_{t2}$;

Schritt3: (* Einmalige Auswertung von Q^{sl}_{emb} *)

---

79) Oft auch als 'nested iteration method' bezeichnet.

80) Eine GROUP BY Anweisung wird zumeist in Verbindung mit einer Aggregatsfunktion benutzt. Dabei muß jeder Attributwert eines Attributs aus der SELECT-Liste eine Eigenschaft der Gruppe sein (was eben durch die Angabe von Aggregatsfunktionen gewährleistet ist; vgl. [CHAM76]).

ZER_{emb} := Eval(Q^{sl}_{emb}[r2=t2]|$^{'r2=t2'}_{CF[t2]}$);

Schritt4: (* Cache-interne Auswertung durch QM. *)

4a: (* *Auswertung der Aggregatsfunktion.* *)

ZER_{aggr_fct} := Eval(SELECT sl1 = aggr_fct(sl), t2
FROM ZER_{emb}
GROUP BY t2);

4b: (* Exakte Auswertung von 't2=r2'. *)

ZER_{Raggr_fct} := $JOIN_{r2=t2}(ZER_R, ZER_{aggr\_fct})$;

4c: (* Auswertung von θ. *)

result ≡ ZER'_R := $RESTRICT_{r1\,\theta\,sl1}(ZER_{Raggr\_fct})$;

Ende Algorithmus2

Dabei ist für die Auswertung nach Algorithmus2 noch sicherzustellen, daß:
- r1,r2 in Schritt1 nicht wegprojeziert werden, und
- s1,t2 *in Schritt3 nicht wegprojeziert werden.*

Beispiel 4.4:

Relationen:

C	c1	c2	c3	c4	c5
	1	2	3	4	3
	4	1	1	2	3
	2	1	2	5	5
	3	3	1	4	3
	5	4	1	2	2
	2	1	4	5	1
	6	4	2	1	6

D	d1	d2	d3
	1	4	1
	2	3	3
	3	5	3
	4	4	5
	5	2	5
	7	4	2
	6	7	6

```
Query: SELECT c1,c2,c3
       FROM   C
       WHERE  c1 ≤ 5 AND c3 ≤ 3 AND
              c4=AVG(SELECT d2
                     FROM   D
                     WHERE  c5=d3
                       AND  d1<7)
```

$\longrightarrow Q^{d2}_{emb}\{c5=d3\}$

Auswertung gemäß Algorithmus2:

Schritt1: (* $ZER_C := Eval(\pi_{c1c4c5}\sigma_{c1 \le 5 \wedge c3 \le 3}(C))$ *)

ZER_C	c1	c4	c5
	1	4	3
	4	2	3
	2	5	5
	3	4	3
	5	2	2

Schritt2: (* Wir verwenden den Min-Max Filter. *)
$CF[d3] = F^{MM}[c5]|^{c5}_{d3} = '2 \le d3 \le 5'$

Schritt3: (* $ZER_D := Eval(Q^{d2}_{emb}\{c5=d3\}|^{'c5=d3'}_{'2 \le d3 \le 5'}) \equiv$

$Eval(\pi_{d2d3}\sigma_{2 \le d3 \le 5 \wedge d1<7}(D))$ *)

ZER_D	d2	d3
	3	3
	5	3
	4	5
	2	5

Schritt4a: (* Auswertung von AVG, gruppiert nach d3-Werten, aus ZER_D;
d22 = AVG(d2) *)

ZER_{AVG}	d22	d3
	4	3
	3	5

Schritt4b: (* $ZER_{CAVG} := JOIN_{c5=d3}(ZER_C, ZER_{AVG})$ *)

ZER_{CAVG}	c1	c4	c5	d22
	1	4	3	4
	4	2	3	4
	2	5	5	3
	3	4	3	4

Schritt4c: (* ZER_C' := $RESTRICT_{c4=d22}(ZER_{CAVG})$
mit Wegprojezieren von d22 *)

ZER_C'	c1	c4	c5
	1	4	3
	3	4	3

Anmerkungen zu Algorithmus2:

(a) Der benötigte DB-Cache-Platz kann durch Pipelining der Schritte 4b und 4c weiter reduziert werden.

(b) Die frühzeitige Freigabe nicht mehr benötigter ZERs ist nicht explizit erwähnt; z.B. kann ZER_{emb} nach Schritt4a durch den RELEASE-Operator freigegeben werden.

(c) Weitere interessante Variationen von Algorithmus2 werden am Ende von Kap.4.3.2 diskutiert.

Die Korrektheit von Algorithmus2 läßt sich folgendermaßen begründen:
Durch den dyn. Filter CF[t2] wird eine Obermenge (für totale Filter die exakte Menge) der korrelierenden r2-Werte bestimmt. In Schritt3 erfolgt dann die einmalige Auswertung des inneren Queryblocks Q^{sl}_{emb} mit CF[t2] anstelle von 'r2=t2', d.h. wir benutzen quasi die Technik der Tupelmengen-Substitution. Schritt4a wertet dann die Aggregatsfunktion für sl, gruppiert nach t2-Werten, aus. ZER_{aggr_fct} enthält dabei evt. mehr Tupel als notwendig, falls CF[t2] nicht der totale Filter ist. Die spezielle Verwendung des JOIN-Operators in Schritt4b wertet danach exakt das korrelierende Prädikat 'r2=t2' aus und verbindet die zusammengehörigen r1-Werte und aggr_fct-Werte. Schließlich kann dann cache-intern das äußere Prädikat 'r1θ...' ausgewertet werden.
Weiter ist zu vermerken, daß aufgrund der speziellen Gestalt von

ZER_{aggr_fct} (Attribut t2 ist Schlüsselattribut wegen GROUP BY) gilt, daß $card(ZER_{Raggr\_fct}) \leq card(ZER_R)$. Falls auch Pipelining der Schritte 4b und 4c gemacht wird, so ist für 4b und 4c zusammen nur Platz für ZER_R' erforderlich. Insgesamt ergeben sich somit im Normalfall keine übermäßigen DB-Cache-Platzanforderungen.[81)]

Effizienz von Algorithmus2:
Von dem vorgestellten Tupelmengen-Substitutionsmechanismus unter Einsatz dyn. Filter wird erwartet, daß er (erstmalig) eine effiziente Auswertung von Korrelationsqueries erlaubt. Der Vorteil von Algorithmus2 liegt darin, daß der innere Queryblock nur einmal ausgewertet werden muß. Dieser Vorteil ist besonders augenfällig, wenn die Anzahl der verschiedenen zu substituierenden r2-Werte hoch ist oder/und $Q^{s1}_{emb}\{r2=t2\}$ einen Join enthält (wie die eingangs angegebene $\forall_{corr}$-Query).
Die Verwendung von Algorithmus2 ist natürlich nicht auf unsere spezielle DB-Architektur eingeschränkt (was auch für die anderen Algorithmen gilt), sondern kann - mit entsprechenden Effizienzverlusten - auch in konventionellen DB-Systemen zur Auswertung von $\forall_{corr}$-Queries eingesetzt werden. Selbst für diesen Fall wird in vielen Fällen eine deutliche Effizienzsteigerung gegenüber der klassischen Methode der iterierten Tupelsubstitution erwartet. Im Normalfall (alle ZERs passen in das DB-Cache) ist in unserer Architektur mit einer noch weitaus größere Effizienzverbesserung zu rechnen. Der Autor erwartet insbesondere bei komplexen Q_{corr}-Queries eine Verbesserung um Größenordnungen; auch für solche Fälle, wo für die Auswertung der inneren Korrelation keine schnellen Zugriffspfade existieren, wird die Effizienzsteigerung sehr beträchtlich sein. Ausführliche quantitative Analysen in dieser Richtung müssen jedoch späteren weiterführenden Untersuchungen vorbehalten bleiben.

Abschließende Anmerkung:
I.a. sind Korrelationen nicht nur zwischen verschiedenen Relationen (wie in Bsp.4.4), sondern auch innerhalb einer Relation in SQL zulässig. SQL ermöglicht dies durch die Verwendung von Tupelvariablen, z.B.:

81) Vgl. dazu auch die Diskussion an Ende von Kap.4.2.3.

```
SELECT c1,c2
FROM   C tvar
WHERE  3 ≤ c4 ≤ 5 AND
       c2>MIN( SELECT c2
               FROM   C
               WHERE  c3=tvar.c4 )
```

Eine mit einer Tupelvariablen quantifizierte Relation (hier C tvar) kann als verschiedene Relation (also C tvar $\neq$ C) aufgefaßt werden. Damit sind die Definitionen zur Querykategorisierung und somit auch Algorithmus2 sofort anwendbar.

4.2.3.3. Auswertungsalgorithmen für Q_{join}-Querytypen.

Unsere Untersuchungen wollen wir auf einen Q_{join}-Querytyp konzentrieren, welcher oft als 'Restriktion-Projektion-Equijoin Kern' der RelA bezeichnet wird.

```
Q_join ≡ SELECT *
         FROM   R,S
         WHERE  G AND H AND r=s
```

Dabei gilt:

- G und H sind jeweils Restriktionsformeln auf den Relationen R bzw. S.
- r (s) ist Attribut von R (S); r und s bezeichnen wir als Joinattribut von R bzw. S.

In reiner RelA-Form läßt sich obiges Queryschema wie folgt formulieren:

$$Q_{join} \equiv \sigma_{G \wedge H}(R \underset{r=s}{|X|} S) \equiv \sigma_G(R) \underset{r=s}{|X|} \sigma_H(S)$$

Restriktionen so früh wie möglich auszuführen reduziert zwar die Kardinalität der beiden Joinoperanden von card(R) auf card(σ_G(R)) bzw. von card(S) auf card(σ_H(R)), jedoch stellt diese Verfahren noch keine maximale Reduktion der für den Join benötigten ZERs dar. Damit auch für

diesen Querytyp der Normalfall, d.h. beide Joinoperanden-ZERs passen in das DB-Cache, fast immer in praktischen Anwendungsfällen eintritt, ist nach Möglichkeiten für weitere Reduktionen zu suchen. Selbstverständlich werden dyn. Filter wieder das geeignete Instrument zur Erreichung dieses Ziels sein.

Grundidee.

Nach Definition ist der Join eine Restriktion auf dem cartesischen Produkt und es gilt somit:

$$0 \leq \text{card}(\sigma_G(R) \underset{r=s}{|X|} \sigma_H(S)) \leq \text{card}(\sigma_G(R)) * \text{card}(\sigma_H(S))$$

Es nehmen also i.a. nicht alle Tupel von $\sigma_G(R)$ und $\sigma_H(S)$ am Join teil. Unser Ziel muß es daher sein zu vermeiden, daß Tupel, welche nicht am Join teilnehmen, durch eine LI von einer DB-Platte in das DB-Cache transportiert werden.

Die Berechnung des Joins soll am Ende durch einen cache-internen Merge-Join mit zwei ZERs als Operanden geschehen. Um Filter für Tupel konstruieren zu können, welche 'am Join teilnehmen', muß dieser Begriff genauer spezifiziert werden.

Aufgrund der Semantik des Joins $\sigma_G(R) \underset{r=s}{|X|} \sigma_H(S)$ ergibt sich:

Ein Tupel t_R von $\sigma_G(R)$ erfüllt das Joinprädikat 'r=s' genau dann, wenn gilt:

$$\exists \text{ Tupel } t_S \in \sigma_G(S): t_R.r = t_S.s$$

Definition 4.10:

Sei t_R Tupel von $\sigma_G(R)$.

t_R nimmt am Join $\sigma_G(R) \underset{r=s}{|X|} \sigma_H(S)$ teil :iff $t_R.r \in \pi_s\sigma_H(S)$

Bem. Die Definition für Tupel t_S von S ist symmetrisch.

Zur Beantwortung der Fragen '$t_R.r \in \pi_S\sigma_H(S)$?' und '$t_S.s \in \pi_r\sigma_G(R)$?' sind dyn. Filter wieder das geeignete Instrument. Die zu ermittelnden dyn. Filter sollen dann diejenigen Tupel von R und S qualifizieren, welche vielleicht am Join teilnehmen. Durch die anschließend vorzunehmende Querymodifikation durch konjunktives Anfügen der Filter wird somit vermieden, daß einige oder alle Tupel (je nach Selektivität des verwendeten Filters), welche nicht am Join teilnehmen (definitive Nein-Antworten), aufgrund von EXH_FILTER oder SEL_FILTER Operationen durch eine LI von der DB-Platte in das DB-Cache transportiert werden müssen.

In den bisherigen Algorithmen haben wir dyn. Filter stets nur von ZERs abgeleitet. Für die Auswertung von $\bar{\sigma}_{join}$-Queries bietet sich folgende einfache Erweiterung bzgl. der Indexverarbeitung an. Falls ein Index über einem Joinattribut existiert, so bezeichnen wir ihn als Joinindex. Als zusätzliche Forderung an die Indexverarbeitung durch SAM/QM verlangen wir nun:

Die Tupelidliste einer Restriktionsauswertung, an der ein Joinindex beteiligt ist, enthält neben den Tupelids (lids bzw. (lid,pid)-Paare) auch die zugehörigen Joinattributwerte.

In Erweiterung früherer Notation (Op1) $Id_R := IND_G^+(R)$ zur Auswertung der Restriktion G^+ über Indexe führen wir für den Fall der Joinindexverwendung für Attribut r von R folgende Bezeichnung ein:

(Op1') $\boxed{Id_R(r) := JIND_G^+(R)}$ (*Join-INDexauswertung*)

$Id_R(r)$ ist dabei eine Liste von Paaren (Tupelid,vr), wobei vr der zugehörige Joinattributwert von r ist.

Mit Hilfe dieser natürlichen Erweiterung erhalten wir eine zusätzliche Möglichkeit zur Berechnung dyn. Filter, nämlich außer von ZERs auch von Tupelidlisten bei Joinindexverwendung. Wir werden nun drei verschiedene Auswertungsalgorithmen für $\bar{\sigma}_{join}$-Querytypen angeben, welche sich alle zuerst mit Relation R 'befassen'. Drei symmetrische Algorithmen würde man dadurch erhalten, indem man die Rollen von R und S vertauscht. Ferner sei nochmals an die beiden Zugriffspfadtypen zur Berechnung von $Eval(\sigma_G(R))$ erinnert, nämlich

- Vollständige Segmentfilterung: Hierbei ist eine LI (EXH_FILTER) sowie i.a. der QM zur Sortierung für den nachfolgenden Join beteiligt.
- Indexverarbeitung & selektive Segmentfilterung: Hierbei ist der SAM (IND_SCAN), i.a. auch der QM (INTERSEC, UNION von Tupelidlisten), sowie eine LI (SEL_FILTER) beteiligt; i.a. muß dann noch der QM eine Sortierung für den nachfolgenden Join vornehmen.

Anmerkung: Zusätzliche Optimierungsmöglichkeiten durch frühzeitige Freigabe von nicht mehr benötigtem DB-Cache-Platz (für ZERs oder Tupelidlisten) sind nicht explizit erwähnt.

Algorithmus O-JFM: (O-Feedback Joinfilter Methode)

O-JFM

Schritt1: $ZER_R := Eval(\sigma_G(R))$;

Schritt2: Berechne dyn. Filter F[r] für $\pi_r(ZER_R)$;
$JFO[s] = F[r]!_S^r$;

Schritt3: (* Ermittlung einer Obermenge der Tupel von S, welche am Join teilnehmen. *)
$ZER_S := Eval(\sigma_H \wedge JFO[s](S))$;

Schritt4: (* Cache-interner Join. *)
result $\equiv ZER_{RS} := JOIN_{r=s}(ZER_R, ZER_S)$

Ende O-JFM

Besondere Voraussetzungen: keine.
Die Korrektheit von Algorithmus O-JFM folgt unmittelbar aus der Def. der dyn. Filter sowie aus Def. 4.10.

Anmerkungen:

(1) Alg. O-JFM filtert keine Tupel von $\sigma_G(R)$ weg, welche nicht am Join teilnehmen; in Filter-Terminologie hieße das die Verwendung des trivialen Filters.

(2) Falls JFO[s] der totale Filter ist, dann werden alle Tupel von

$\sigma_H(S)$, welche nicht am Join teilnehmen, rechtzeitig weggefiltert. (Algorithmen, welche solches leisten, kann man als optimale Joinfiltermethoden bzgl. R und/oder S bezeichnen.)

(3) Falls JFO[s] als Min-Max Filter gewählt wird, so kann es passieren, daß JFO[s] nur eine ganz geringe Einschränkung liefert, weil i.a. $\pi_r(ZER_R)$ ein stochastischer Teil von $\pi_r(R)$ ist.[82] Aber auch in diesem ungünstigen Fall ist Alg. O-JFM nicht schlechter als bekannte Joinalgorithmen wie die 'Nested-Loops'-Methode (vgl. SystemR), welche überhaupt keine frühzeitige Wegfilterung von nicht am Join beteiligten Tupeln bewirkt.

Für die restlichen zwei Algorithmen benötigen wir Zerlegungen von G und H der Art

$$G \equiv G^+ \wedge G^-,\ H \equiv H^+ \wedge H^-.$$

Dabei verlangen wir, daß

- G^+ (H^+) alle Restriktionen auf dem Joinattribut r (s) enthält, sowie
- G^+ (H^+) vollständig über Indexe auswertbar ist (das Ergebnis ist eine Tupelidliste plus zugehöriger Joinattributwerte, vgl. (Opl')).

Bemerkung: Auch der Spezialfall $G^+ \equiv$ true ($H^+ \equiv$ true) ist zulässig, falls keine Restriktionen auf r (s) existieren.

Algorithmus 1-JFM: (1-Feedback Joinfilter Methode)

1-JFM

Schritt1: (* *Indexauswertung von* G^+ *)*
$Id_R(r) := JIND_{G^+}(R)$;

Schritt2: Berechne dyn. Filter F[r] für $\pi_r(Id_R(r))$;
$JF1[s] = F[r]|^r_S$;

Schritt3: (* Querymodifikation für $\sigma_H(S)$ *)
$ZER_S := Eval(\sigma_H \wedge JF1[s](S))$;

82) Ein (noch zu entwickelndes) Optimierungsverfahren für die Wahl eines geeigneten Filtertyps würde jedoch diese Situation erkennen und anstelle des Min-Max Filters einen selektiveren Filter wählen, etwa iterierte Min-Max Filter oder Hash-Filter.

Schritt4: (* Feedback von ZER_S auf $Id_R(r)$ *)
Berechne $F^{tot}[s]$ für $\pi_S(ZER_S)$;
$JF1[r] = F^{tot}[s]|^r_S$;
$Id_R'(r) := RESTRICT_{JF1[r]}(Id_R(r))$;
Schritt5: (* Materialisierung von Tupeln von R *)
$ZER_R := SEL_{\sigma_G}(Id_R'(r))$;
Schritt6: (* Cache-interner Join *)
result ≡ $ZER_{RS} := JOIN_{r=s}(ZER_R, ZER_S)$

Ende 1-JFM

Besondere Voraussetzungen:
Existenz geeigneter Indexe zur Auswertung von G^+, d.h. insbesondere Existenz eines Joinindexes für r.

Die Korrektheit von Alg. 1-JFM ergibt sich wie folgt:
- Schritt 1-3 ist korrekt aufgrund der Def. der dyn. Filter und Def. 4.10.
- Durch die spezielle Anwendung des RESTRICT-Operators in Schritt4 werden aus $Id_R(r)$ all diejenigen Tupelids entfernt, welche den Filter JF1[r] nicht erfüllen. JF1[r] kann als totaler Filter gewählt werden, da die Filterung im DB-Cache erfolgt und somit die Im-Flug Beschränkungen der LIs nicht relevant sind. Somit enthält ZER_R in Schritt5 alle Tupel von R, welche $G \wedge JF1[r]$ erfüllen, was aufgrund der Def. dyn. Filter wieder eine Obermenge der am Join teilnehmenden Tupel von R darstellt.
  (Anm.: Schritt4 kann auch wie folgt implementiert werden:
  $Id_R'(r) := JOIN_{r=s}(Id_R(r), \pi_S(ZER_R))$)

Zur Terminologie:
Schritt4 bezeichnen wir als Feedback[83] , da die dadurch bewirkte Restriktion von $Id_R(r)$ abhängt von dem dyn. Filter JF1[s], welcher vorher von $Id_R(r)$ selbst abgeleitet wurde.

[83] Die Idee des Feedbacks ist auch in [YA079] zu finden.

Spezielle Eigenschaften von Alg. 1-JFM:
Bei Alg. 1-JFM werden sowohl gewisse Tupel von $\sigma_H(S)$ als auch von $\sigma_G(R)$, welche nicht am Join teilnehmen, vor der Übertragung in das DB-Cache weggefiltert. Falls JFO[s]≡JF1[s], dann erreichen wir dieselbe Reduktion von ZER_S in 0-JFM und 1-JFM. Da die Elimination von Tupelids aus $Id_R(r)$ in Schritt4 durch die Anwendung eines totalen Filters geschieht, erreichen wir bei 1-JFM für ZER_R immer einen hohen Joinfilter-Wirkungsgrad, der jedoch möglicherweise durch hohe Kosten für die Indexauswertung von G^+ in Schritt1 erkauft werden muß.

Algorithmus 2-JFM: (2-Feedback Joinfilter Methode)

2-JFM

Schritt1: (* Indexauswertung von G^+ *)
$Id_R(r) := JIND_{G^+}(R)$;
Schritt2: Berechne dyn. Filter F[r] für $\pi_r(Id_R(r))$;
JF2[s] = F[r]¦^r_s;
Schritt3: (* Querymodifikation für Indexauswertung *)
$Id_S(s) := JIND_{H^+} \wedge JF2[s](S)$;
Schritt4: (* 1. Feedback *)
Berechne $F^{tot}[s]$ für $\pi_s(Id_S(s))$;
JF2[r] = F^{tot}[s]¦^s_r;
$Id_R'(r) := RESTRICT_{JF2[r]}(Id_R(r))$;
Schritt5: (* Materialisierung von Tupeln von R im DB-Cache *)
$ZER_R := SEL_{\sigma_G}(Id_R'(r))$;
Schritt6: (* 2. Feedback *)
Berechne $F^{tot}[r]$ für $\pi_r(ZER_R)$;
JF2'[s] = F^{tot}[r]¦^r_s;
$Id_S'(s) := RESTRICT_{JF2'[s]}(Id_S(s))$;
Schritt7: (* Materialsierung von Tupeln von S im DB-Cache *)
$ZER_S := SEL_{\sigma_H}(Id_S'(s))$;
Schritt8: (* Cache-interner Join *)
result ≡ $ZER_{RS} := JOIN_{r=s}(ZER_R, ZER_S)$

Ende 2-JFM

Besondere Voraussetzungen:
Existenz geeigneter Indexe zur Auswertung von G^+ und H^+, d.h. insbesondere Existenz von Joinindexen für r und s.

Der Korrektheitsnachweis kann wiederum analog zu 1-JFM geführt werden.
Explizit sei nochmals auf die Wirkung der beiden Feedbacks eingegangen. Durch die jeweilige Anwendung eines totalen Filters auf bereits berechnete Tupelidlisten erreichen wir einen hohen Joinfilter-Wirkungsgrad für R und S, was jedoch evt. durch hohe Indexauswertungskosten in Schritt1 und Schritt3 erkauft werden muß.
Als neue Einsatzmöglichkeit eines dyn. Filters ist bei Alg. 2-JFM die Querymodifikation in Schritt3 für die Indexauswertung hervorzuheben. Aufgrund der bei einer Indexauswertung entstehenden Kosten sind bei der Wahl eines geeigneten dyn. Filters eigene Maßstäbe anzulegen. In vielen Fällen könnten Min-Max Filter sich als geeignet erweisen,wenn dadurch der Restriktionsbereich auf s stark genug eingeschränkt wird, z.B. wenn H^+ = '5<s<500', JF2[s] = '450<s<2000' $\longrightarrow$ $H^+ \wedge$ JF2[s] = '450<s<500'. (Bei der Wahl von JF2[s] als totaler Filter sind die Zusatzkosten für die vielen benötigten Indexdurchläufe von der Wurzel bis zu den Blattknoten zu berücksichtigen.)
Abschließend soll anhand eines Beispiels die Auswertung durch die drei JFM-Algorithmen demonstriert werden.

Beispiel 4.5:

Relationen:

C	c1	c2	tid
	1	9	T10
	1	8	T11
	2	7	T12
	2	6	T13
	3	5	T14
	8	4	T15
	9	3	T16
	9	2	T17
	5	1	T18

D	d1	d2	d3	tid
	8	1	1	T20
	3	2	2	T21
	4	3	3	T22
	9	4	9	T23
	2	5	8	T24
	3	6	7	T25
	6	7	4	T26
	6	8	5	T27
	7	9	6	T28

Query:

```
SELECT c2,d2,d3
FROM   C,D
WHERE  (2 ≤ c1 ≤ 6) AND (3 ≤ d1 ≤ 6)
       AND (c1=d1)
```

(Anmerkung: $G=G^+$, $H=H^+$ in diesem Beipiel.)

(1) Auswertung gemäß Alg. O-JFM:

Schritt1:

ZER_C	c1	c2
	2	7
	2	6
	3	5
	5	1

(*durch SAM&LI oder LI*)

Schritt2: (* Wir verwenden den Min-Max Filter. *)
JFO[d1] = '2 ≤ d1 ≤ 5'; (*durch QM*)

Schritt3: (* ZER_D := Eval($\sigma_{3 \le d1 \le 5}(D)$); $3 \le d1 \le 5$ entsteht aus $(2 \le d1 \le 5) \wedge (3 \le d1 \le 6)$. *)

ZER_D	d1	d2	d3
	3	2	2
	4	3	3
	3	6	7

(*durch SAM&LI oder SAM*)

Schritt4: (* $JOIN_{c1=d1}$ mit entspr. Projektionen. *)

ZER_{CD}	c2	d2	d3
	5	2	2
	5	6	7

(*durch QM*)

(2)

Auswertung gemäß Alg. 1-JFM:

Schritt1 (* $Id_C(c1) := JIND_{2 \le c1 \le 6}(C)$ *)

$Id_C(c1)$	c1	tid
	2	T12
	2	T13
	3	T14
	5	T18

(*durch SAM*)

Schritt2: (* Min-Max Filter durch QM *)
JF1[d1] = '$2 \le d1 \le 5$'

Schritt3: (* $ZER_D := Eval(\sigma_{3 \le d1 \le 5}(D))$ *)

ZER_D	d1	d2	d3
	3	2	2
	4	3	3
	3	6	7

(*durch SAM&LI oder LI*)

Schritt4: (* Feedback: JF1[c1] = '(c1=3) $\vee$ (c1=4)' *)

$Id_C'(c1)$	c1	tid
	3	T14

(*durch QM*)

Schritt5: (* $ZER_C := SEL_{\underline{true}}(Id_C'(c1))$ *)

ZER_C	c1	c2
	3	5

(*durch LI*)

Schritt6:

ZER_{CD}	c2	d2	d3
	5	2	2
	5	6	7

(*durch QM*)

Anmerkung:
Die Tupelids in Schritt1 werden an dieser Stelle noch nicht benötigt, sondern werden in unserem Fall nur gleich miterzeugt, da es billig ist (vgl. Indexorganisation als B-Baum). Eigentlich werden die Tupelids erst in Schritt5 benötigt. Aufgrund dieses Freiheitsgrads sind weitere interessante Algorithmen zur Filterkonstruktion denkbar, welche wir aber von unseren Basisuntersuchungen ausklammern wollen.

(3)

Auswertung gemäß Alg. 2-JFM:

Schritt1:

$Id_C(c1)$	c1	lid
	2	T12
	2	T13
	3	T14
	5	T18

(*durch SAM*)

Schritt2: (* Min-Max Filter durch QM *)
JF2[d1] = '2 ≤ d1 ≤ 5'

Schritt3: (* $Id_D(d1)$:= $JIND_{3 \le d1 \le 5}(D)$ *)

$Id_D(d1)$	d1	lid
	3	T21
	3	T25
	4	T22

(*durch SAM*)

Schritt4: (* 1.Feedback: JF2[c1] = '(c1=3) ∨ (c1=4)' *)

$Id_C'(c1)$	c1	lid
	3	T14

(*durch QM*)

Schritt5:

ZER_C	c1	c2
	3	5

(*durch LI*)

Schritt6: (* 2.Feedback: JF2'[d1] = 'd1=3' *)

$Id_D'(d1)$	d1	lid
	3	T21
	3	T25

(*durch QM*)

Schritt7:

ZER_D	d1	d2	d3
	3	2	2
	3	6	7

(*durch LI*)

Schritt8:

ZER_{CD}	c2	d2	d3
	5	2	2
	5	6	7

(*durch QM*)

Ein Vergleich der Kardinalitäten der jeweiligen Joinoperanden ergibt hier:

	CARD(ZER_C)	card(ZER_D)
0-JFM	4	3
1-JFM	1	3
2-JFM	1	2

Ohne Verwendung dyn. Filter, d.h. nur Auswertung der Restriktionen $\sigma_{2 \le c1 \le 6}(C)$ und $\sigma_{3 \le d1 \le 6}(D)$ vor dem Join, ergäbe sich: $card(ZER_C)=4$, $card(ZER_D)=5$.

Weitere Aspekte der dyn. Filteralgorithmen.

Aspekt1: Tradeoff DB-Cache-Platzbedarf $\longleftrightarrow$ LI-Filterungszeit.
Bei einer sehr hohen Parallelität bei der Transaktionsabarbeitung könnte, trotz großer DB-Cache-Kapazität, der Fall eintreten, daß der für parallele Transaktionen vom CM zur Verfügung zu stellende DB-Cache-Platz dennoch zum Engpaß wird. Bei solchen Anwendungen ist eine Variation der angegebenen Basisalgorithmen denkbar, welche wir anhand von Bsp. 4.4 diskutieren wollen.
Im Bsp. 4.4 muß ZER_C relativ lange im DB-Cache gehalten werden, nämlich von Schritt 1 bis Schritt 4b.
Alternativ-Algorithmus 2':

- ZER_C wird im Schritt 1 nicht im DB-Cache materialisiert, sondern der dynamische Filter $CF[d_3]$ aus Schritt 2 wird direkt von der LI im Flug ermittelt, während Relation C gemäß $\pi_{c5}\sigma_{c1 \le 5 \wedge c3 \le 3}(C)$ gefiltert wird.
- Vor Schritt 4b ist dann i.a. eine erneute Filterung von C gemäß $\pi_{c1c4c5}\sigma_{c1 \le 5 \wedge c3 \le 3}(C)$ notwendig, welche jetzt erst die ZER_C als Ergebnis im DB-Cache liefert.
  An dieser Stelle (vor Ausführung von Schritt 4b) ist nun erneut ein profitabler Einsatz eines dynamischen Filters möglich:
  - Berechne dynamischen Filter F[d3] für $\pi_{d3}(ZER_{AVG})$;
  - $CF[c5] = F[d3]|^{d3}_{c5}$;
  - $ZER_C := Eval(\pi_{c1c4c5}\sigma_{c1 \le 5 \wedge c3 \le 3 \wedge CF[c5]}(C))$;

Die Eigenschaften dieses Algorithmus sind:

(a) ZER_C ist nur noch von Schritt 4b bis 4c im DB-Cache.

(b) Die LI muß C zweimal filtern.
Jedoch wird die Kardinalität von ZER_C hier durch den Einsatz des zweiten dynamischen Filters CF[c5] noch weiter reduziert.

Bei der Entscheidung, ob der Basisalgorithmus oder dieser Alternativ-Algorithmus vorzuziehen ist, muß man somit abwägen zwischen DB-Cache-Platzbedarf und LI-Filterungszeit. Diese Überlegungen lassen sich auf alle übrigen Basisalgorithmen ebenso übertragen.[84]

Aspekt 2: Welche dynamischen Filter lassen sich von einer LI im Flug berechnen?

Bei obigen Alternativ-Algorithmus haben wir stillschweigend eine geringfügige Modifikation unserer bisherigen Architektur vorgenommen, indem wir die dynamische Filterermittlung durch eine LI vorsahen. (Bisher erfolgte dies stets durch den QM.)

Die Berechnung dynamischer Filter durch eine LI ist nur sinnvoll, wenn sie -ohne die vollständige Erstellung der zugrundeliegenden ZER- im Flug geschehen kann. Dies ist sicherlich nicht für alle dynamischen Filter möglich. Für Min-Max Filter ist es jedoch bestimmt möglich,[85] bei totalen Filtern in der disjunktiven Darstellung dagegen spielt wieder die in Kap.3.2.4.1 erwähnte Im-Flug-Komplexitätsgrenze eine Rolle. Interessant erscheint hier auch die Idee, dyn. Filter für gewisse Attribute permanent in der DB zu speichern und bei Updates ähnlich wie Indexe dynamisch fortzuschreiben. Die Untersuchung dieser interessanten Teilfrage - welche dyn. Filter sind von einer LI im Flug berechenbar - kann jedoch im Rahmen dieser Arbeit nicht vorgenommen werden.

Aspekt 3: Lassen sich weitere Operationen sinnvollerweise in die LIs auslagern?

In Aspekt2 wurde bereits eine Erweiterung des Aufgabenbereichs der LIs diskutiert. Als Fortsetzung läßt sich darüber reflektieren, ob es weitere

---

84) Eine weitere denkbare Variante wäre, ZER_C nach Schritt 2 auf einen anderen Speicherbereich auszulagern und erst in Schritt 4b wieder in das DB-Cache einzubringen.

85) Die Auswertung der Aggregatsfunktion MIN und MAX durch eine LI ist sowieso bereits vorgesehen.

lineare Operationen gibt, welche von den LIs nach Möglichkeit im Flug ausgewertet werden können. Für den heutigen Stand der Technik haben wir bereits eine sehr gute Aufgabenverteilung zwischen LIs und Host-DBMS getroffen. Eine denkbare Variante für die Zukunft wäre z.B. wie folgt: Jede LI verfügt über einen 'Supersortierer'[86] , welcher im Flug sortieren könnte. Als Konsequenz davon wären Duplikatelimination, sowie die Auswertung von GROUP BY mit Aggregatsfunktionen von den LIs schnell durchführbar.[87] Würde man diese Aufgabenumverteilung vornehmen, so verblieben für den QM nur noch die zweistelligen RelA-Operatoren JOIN, UNION, INTERSEC (und Division).
Die unter diesem Aspekt3 diskutierten Architekturvarianten werden, insbesondere aufgrund der letzten Feststellung, wie folgt eingeschätzt: Nach unserer Meinung ist mit der bisher angegebenen Architektur eine sehr gute Aufgabenverteilung zwischen LIs und Host-DBMS verwirklicht. Eine weitergehende Umverteilung vom Host auf die LIs würde die Gefahr eines LI-Engpasses in sich tragen; als gegenläufiger Effekt würde sich eine 'Unterbeschäftigung' der leistungsfähigen Host-CPU(s) einstellen.[88]

Zusammenfassend kommen wir zu folgender Bewertung der angegebenen Varianten:
Aspekt3 scheint im Augenblick nicht sonderlich vielverprechend zu sein. Interessant hingegen sind die unter Aspekt1 und 2 erörterten Varianten. Ein weiteres Musterbeispiel dafür, daß man den erforderlichen DB-Cache-Platzbedarf sorgfältig abwägen sollte, liegt in dem früheren Bsp. aus Abb. 3.10, Kap. 3.2.4 vor: Dort bringt eine erhöhte Intra-Transaktionsparallelität einen erhöhten DB-Cache-Platzbedarf mit sich.
Damit wollen wir den Ausblick auf eine weitere Optimierung der dyn. Filteralgorithmen abschließen und uns für den restlichen Teil dieser

[86] Solche Supersortierer sind heute bereits für Blasenspeicher entwickelt worden (siehe z.B. [CHEN78]).

[87] In Bsp. 4.4 könnte man sich dann die Zwischenspeicherung von ZER_D im DB-Cache sparen; nur die (wesentlich kleinere) ZER_{AVG} würde von einer LI in das DB-Cache transportiert werden müssen.

[88] In der Entwicklung von Hochleistungs-Host-CPUs sind heute bereits enorme Fortschritte erzielt worden: z.B. hat Fujitsu einen 28 MIPS Rechner angekündigt.

Arbeit wieder auf die in Kap.3 präsentierte Architektur konzentrieren.

4.2.4. Auswertung von SQL-Queries mit GROUP BY HAVING Anweisungen.

Die allgemeine Form eines SQL-Queryblocks lautet:

```
SELECT <target list>
FROM   <relation list>
       [WHERE  <boolean>]
       [GROUP BY <field list>]
       [HAVING <boolean>]
```

Eine Relation kann in Gruppen nach gewissen Attributwerten aufteilt werden, dann kann eine eingebaute Funktion auf jede Gruppe angewendet werden. Die Verwendung einer GROUP BY Klausel verlangt, daß jedes Element der <target list> eine Eigenschaft der Gruppe ist. In der target list erscheinen als eingebaute Funktionen daher meistens auch Aggregatsfunktionen. Ein expliziter GROUP BY Operator ist nicht Bestandteil der RelA. Die Semantik von GROUP BY kann jedoch wie folgt definiert werden (vgl. [GRAY81]):
GROUP BY verhält sich wie eine verallgemeinerte Projektion (auf den zu gruppierenden Attributen), es liefert zusätzlich abgeleitete Attribute, welche funktional abhängig von den zu gruppierenden Attributen sind.

Beispiel 4.6:

Relation: R

r1	r2	r3	r4
1	2	3	1
3	4	3	2
5	2	5	4
4	1	2	5
3	4	2	1

Query:

```
SELECT r2,SUM(r3),MIN(r4),COUNT()
FROM   R
GROUP BY r2
```

Dadurch ist folgende Relation R' charakterisiert:

R'	r2	SUM(r3)	MIN(r4)	COUNT()
	2	8	1	2
	4	5	1	2
	1	2	5	1

Zum Vergleich, die Projektion von R auf r2 ergibt:

$\pi_{r2}(R)$	r2
	2
	4
	1

Konsequenz:
GROUP BY als verallgemeinerter Projektionsoperator ist eine nicht-lineare Operation, d.h. GROUP BY kann von den LIs i.a. nicht im Flug ausgewertet werden (jedenfalls nicht in einem Durchgang durch die Relation).

Auswertung von GROUP BY:
Die Auswertung von GROUP BY auf permanenten Relationen muß wie folgt aufgeteilt werden:

(1) Partielle Projektionen durch eine LI.
(2) Durch den QM:
    (a) Sortieren nach zu gruppierendem Attribut;
    (b) Berechnung von Aggregatsfunktionen sowie Duplikatelimination.

Eigenschaften der HAVING Anweisung:
Restriktionen in einer HAVING Anweisung beziehen sich immer auf Gruppen. Die Auswertung von GROUP BY HAVING hat somit in folgender Reihenfolge zu geschehen:

(1) Gruppenbildung durch GROUP BY.
(2) Restriktionen auf Gruppen durch HAVING.

Das heißt aber, daß HAVING Anweisungen im DB-Cache durch den QM auszuwerten sind.

5. Optimierungen auf der Zugriffspfadebene.

Um die Intentionen dieses Kapitels ins rechte Licht zu rücken, seien einige Anmerkungen vorweggeschickt. Das Ziel dieses Kapitels wird darin bestehen, grundlegende Techniken zur Lösung von Problemen der Optimierung auf der Zugriffspfadebene zu präsentieren, welche sich wesentlich von den bekannten Optimierungstechniken wie etwa in SystemR oder INGRES unterscheiden. Die Notwendigkeit einer Revision der Optimierungsmethoden auf diesem Gebiet ist durch die spezifischen Fähigkeiten unserer DB-Architektur bedingt. Im wesentlichen werden wir auf zwei Problemkreise näher eingehen, nämlich:

(a) Entwicklung konkreter Kostenfunktionen und Optimierungsalgorithmen für einen Optimizer.
Selbstverständlich können wir keinen kompletten Optimizer spezifizieren, sondern wir werden die neuen Konzepte exemplarisch am Beispiel der Optimierung von Restriktionsauswertungen diskutieren.

(b) Probleme beim physischen DB-Design im Hinblick auf Kriterien zur Einrichtung von Sekundärindexen.

Die Grundlage für die Problemlösungen wird ein analytisches Modell für das Zugriffsverhalten der DB-Platten bilden. Hierfür werden wir einige vereinfachende Annahmen zur Erstellung eines übersichtlichen Basismodells treffen. Der modulare Aufbau der angegebenen Kostenfunktionen und Optimierungsalgorithmen erlaubt jedoch an einigen Stellen eine Auswechslung von Teilformeln, falls die getroffenen Annahmen in der Praxis nicht bestätigt sind und genauere Aussagen über die statistische Verteilung gewisser Zufallsgrößen vorliegen.

5.1. Analytisches Modell eines DB-Plattenlaufwerks.

5.1.1. Grundlegende Leistungsparameter.

Als DB-Plattenlaufwerk sollen konventionelle Plattengeräte verwendet werden.

Das Leistungsverhalten solcher Standardplatten wird durch folgende elementare Kenngrößen beeinflußt:

(a) Speicherplatzbezogene Kenngrößen:
  (a1) Spurkapazität : cap_tr [bytes]
  (a2) Anzahl der Spuren pro Zylinder : tr_cyl
  (a3) Anzahl der Zylinder pro Plattenlaufwerk: cyl_dp

(b) Mechanische/elektronische Kenngrößen:
  (b1) Umdrehungszeit : rot [msec]
  (b2) Startzeit für Armbewegung : start_h [msec]
  (b3) Überquerungszeit pro Zylinder : next_cyl [msec]

Das betrachtete Plattenlaufwerk verfügt demnach über tr_cyl Plattenoberflächen, sowie tr_cyl Lese/Schreib-Köpfe, von denen jeweils nur einer aktiv ist. Jede Plattenoberfläche enthält cyl_dp Spuren.

Anmerkungen:
- Die Kopfberuhigungszeit einer Plattenarmbewegung wird nicht einzeln erwähnt, da man sie in der Armstartzeit start_h mit berücksichtigen kann.
- Der (elektronische) Umschaltvorgang von einem aktiven Lesekopf auf einen anderen wird als vernachlässigbar klein betrachtet.

Weitere Eigenschaften:
- Jede Spur ist hardwaremäßig in mehrere gleichlange Sektoren formatiert. Ein Sektor stellt die Einheit des Zugriffs für einen Lesekopf dar; die Sektorenlänge beträgt gewöhnlich 256 Bytes. Da man aber bei jedem Lesezugriff nicht nur den relativ geringen Datenumfang von 256 Bytes auslesen will, geht man zu sogenannten Blöcken als Lesezugriffseinheit über. Ein Block ist definiert als eine physisch sequentielle Folge mehrerer Sektoren derselben Spur. Die Blocklänge ist konstant, ein Block ist die Übertragungseinheit zwischen Platte und Arbeitsspeicher.
- Für die Lokalisierungstechnik eines gesuchten Blocks auf einer Spur legen wir folgende Annahme zugrunde:
  Der Suchvorgang kann bei beliebiger Rotationsposition gestartet werden;

sobald der gewünschte Blockbeginn zum ersten Mal unter den Lesekopf rotiert, kann mit dem Auslesen begonnen werden. (Diese Eigenschaft wird bisweilen als Rotational Position Sensing RPS bezeichnet; dabei ist während des Suchvorgangs der Kontroller, der Kanal und die CPU frei für andere Aufgaben ([BUZE75]).)

Als weiterer grundlegender Leistungsparameter ist zu notieren:
(c) Blockgröße : bsize [bytes]

Als zusätzliche Bezeichnungen verwenden wir:
- Zylinderkapazität : cap_cyl := cap_tr·tr_cyl [bytes]
- Plattenstapelkapazität : cap_dp := cap_cyl·cyl_dp [bytes]
- Anzahl der Blöcke pro Spur : $blo\_tr := \left\lfloor \frac{cap\_tr}{bsize} \right\rfloor$
- Anzahl der Blöcke pro Zylinder : blo_cyl := blo_tr·tr_cyl

Da i.a. cap_tr kein Vielfaches von bsize ist, fällt bei der Blockaufteilung in einer Spur 'Verschnitt' an. Wir legen fest, daß dieser Verschnitt gleichmäßig über die ganze Spur verteilt ist.

5.1.2. Zugriffsverhalten elementarer DB-Plattenoperationen.

Aufbauend auf diesen grundlegenden Leistungsparametern soll nun ein einfaches analytisches Modell für das Zugriffszeitverhalten der elementaren DB-Plattenoperationen erstellt werden. Dieses Modell wird später als Grundlage zur Modellierung des Zeitverhaltens der LI-Filteroperationen Verwendung finden.

Die Zugriffszeit, um einen bestimmten Block zu lesen, setzt sich aus drei Komponenten zusammen:
(1) Armbewegungszeit (seek time): die Zeit, um den Plattenarm über dem Zylinder zu positionieren, in dem der gesuchte Block liegt.

(2) Spursuchzeit (search time): die Zeit, um nach erfolgter Armpositionierung und Aktivierung des entsprechenden Lese/Schreibkopfs mit dem Auslesen des gesuchten Blocks beginnen zu können.
(3) Blockübertragungszeit (transfer time): Zeit für einen Blocktransfer von der DB-Platte in den Arbeitsspeicher der betreffenden LI.

Armpositionierungszeit seek_cyl(c), wenn c Zylinder überquert werden müssen: Idealisierend nehmen wir an, daß seek_cyl(c) linear ist in c.

(M1)
$$\text{seek\_cyl}(c) = \begin{cases} \text{start\_h} + c \cdot \text{next\_cyl} & \text{[msec]}, \ 0 < c \leq \text{cyl\_dp} \\ 0 & \text{[msec]}, \text{ falls } c = 0 \end{cases}$$

Durchschnittliche Armpositionierungszeit avg_seek:
Es wird vorausgesetzt, daß die Plattenarmbewegungen nicht durch zusätzliche LI-Software optimiert werden.
X_i sei eine Zufallsvariable, welche die Anzahl der beim nächsten Plattenzugriff zu überquerenden Zylinder angibt, falls der Arm momentan auf Zylinder i positioniert ist. Abkürzend setzen wir c := cyl_dp.
Sei j der Zylinder, auf den neu positioniert werden muß, $j \in \{1,\ldots,c\}$. Armbewegungen sind notwendig, wenn $j \in \{1,\ldots,i-1\}$ oder $j \in \{i+1,\ldots,c\}$. Unter der Annahme, daß die Wahl von j gleichwahrscheinlich aus $\{1,\ldots,c\}$ ist, ergibt sich für den Erwartungswert $E(X_i)$ für X_i:

$$E(X_i) = \frac{1}{c} \cdot \left(\sum_{j=1}^{i-1} j + \sum_{j=1}^{c-i} j \right) = \frac{1}{2 \cdot c} \cdot ((i-1) \cdot i + (c-i) \cdot (c-i+1))$$

Die durchschnittliche Anzahl von zu überquerenden Zylindern beträgt dann

$$\frac{1}{c}\cdot\sum_{i=1}^{c} E(X_i) = \frac{1}{2\cdot c^2}\cdot\Big(\sum_{i=1}^{c}(i-1)\cdot i + \sum_{i=1}^{c}(c-i)\cdot(c-i+1)\Big) =$$

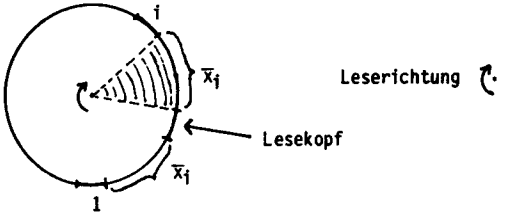
$$\frac{1}{2\cdot c^2}\cdot\Big(\sum_{i=1}^{c} i^2 - \sum_{i=1}^{c} i + \sum_{i=1}^{c-1} i^2 + \sum_{i=1}^{c-1} i\Big) =$$

$$\frac{1}{2\cdot c^2}\cdot\Big(2\cdot\frac{c\cdot(c+1)\cdot(2c+1)}{6} - c^2 - c\Big) = \frac{1}{3}\cdot\Big(c - \frac{1}{c}\Big)$$

Daraus ergibt sich: avg_seek = seek_cyl($\frac{1}{3}\cdot(c - \frac{1}{c})$).
Da in der Praxis c>100 gilt, geben wir als Näherung

(M2) | avg_seek = seek_cyl($\frac{1}{3}$.cyl_dp) [msec] |

Zeitaufwand read_bl(i), um i (beliebig verstreute) Blöcke von einer Spur zu lesen (nach Armpositionierung auf den entspr. Zylinder):
Für die Herleitung der gesuchten Formel legen wir dabei folgende Annahme zugrunde: Das Lesen mehrerer Blöcke auf einer Spur ist innerhalb einer Rotation möglich, d.h. bei der Ablage physisch-sequentieller Blöcke auf einer Spur ist keine Verschränkung nötig (interleaving factor = 1). Ferner darf das Auslesen der i relevanten Blöcke bei einem beliebigen relevanten Block begonnen werden.
Wir verwenden folgendes Modell:



Die gesuchten i Blöcke werden mit 1,...,i nummeriert. O.E. befinde sich bei Suchbeginn der Lesekopf irgendwo zwischen Block i und Block 1; im Mittel befindet er sich in der Mitte eines Blocks, sodaß erst einmal ein halber Block überlesen werden muß. Mit $\overline{x}_i$ bezeichnen wir die mittlere Anzahl von

ganzen zu Überlesenden, nicht relevanten Blöcken bis Block 1. Diese Anzahl ist identisch mit der mittleren Anzahl ganzer nicht relevanter Blöcke zwischen Block i und der Startposition. Aus dieser Überlegung ergibt sich

$$\text{read\_bl(i)} = \left(\frac{1}{2} + (\text{blo\_tr} - \overline{x}_i)\right) \cdot \frac{\text{rot}}{\text{blo\_tr}}.$$

Berechnung von $\overline{x}_i$:

Sei b := blo_tr; der Binominialkoeffizient werde mit C_s^r bezeichnet.

Es gibt C_i^b Möglichkeiten, die i relevanten Blöcke in den b Blöcken unterzubringen, d.h.:

$$\overline{x}_i = \frac{1}{C_i^b} \cdot \left(\underbrace{0 \cdot C_{i-1}^{b-1}}_{\text{falls kein Block überlesen werden muß}} + 1 \cdot C_{i-1}^{b-2} + x \cdot C_{i-1}^{b-1-x} + \ldots + \underbrace{(b-i) \cdot C_{i-1}^{i-1}}_{\text{falls b-i nicht relevante Blöcke überlesen werden müssen}} \right)$$

====>

$$\overline{x}_i = \frac{1}{C_i^b} \cdot \sum_{x=0}^{b-i} x \cdot C_{i-1}^{b-1-x} = \quad \text{(* Terme für } x \geq b-i+1 \text{ sind 0 *)}$$

$$\frac{1}{C_i^b} \cdot \sum_{x=0}^{b-1} C_1^x \cdot C_{i-1}^{b-1-x} = \quad \text{(* [KNUT68], S.58, Gl.(24) *)}$$

$$\frac{1}{C_i^b} \cdot C_{i+1}^b = \frac{b-i}{i+1}$$

(M3)
$$\text{read\_bl(i)} = \left(\frac{1}{2} + \text{blo\_tr} - \frac{\text{blo\_tr} - i}{i+1}\right) \cdot \frac{\text{rot}}{\text{blo\_tr}} \quad \text{[msec]}$$

$$1 \leq i \leq \text{blo\_tr}$$

Zeitaufwand read_tr, um eine gesamte Spur zu lesen (wieder nach entsprechender Armpositionierung) :

Wie soeben nehmen wir auch hier an, daß eine Spur innerhalb einer Umdrehung gelesen werden kann. Read_tr berechnet sich aus

- Zeit, bis erster Blockbeginn zum Lesen unter den Lesekopf rotiert:

$\frac{rot}{2 \cdot blo\_tr}$
- 1 Rotation zum Lesen der Spur.

(M4) $$read\_tr = (1 + \frac{1}{2 \cdot blo\_tr}) \cdot rot \quad [msec]$$

Zeitaufwand read_tr(j,i), um von j Spuren desselben Zylinders jeweils i Blöcke zu lesen (nach Armpositionierung) :
Es sind zwei Situationen zu unterscheiden:

(a) *Der Zeitaufwand, um i Blöcke von der ersten (dieser j) Spuren zu lesen* beträgt read_bl(i).

(b) Der Zeitaufwand, um i Blöcke von einer der restlichen j-1 Spuren zu lesen, läßt sich wie folgt berechnen:
Nach einer unserer Voraussetzungen kann die elektronische Umschaltzeit *von einer Spur auf eine andere vernachlässigt werden. In unserem* Modell kann dies so interpretiert werden, daß sich der aktive Lesekopf nach erfolgter Umschaltung genau an einem Blockanfang befindet. Das bedeutet, daß das Überlesen eines halben Blocks entfällt (vgl. Herleitung (M3)). Somit ergibt sich
$read\_tr(j,i) = read\_bl(i) + (j-1) \cdot (blo\_tr - \frac{blo\_tr-i}{i+1}) \cdot \frac{rot}{blo\_tr}$, also

(M5) $$read\_tr(j,i) = j \cdot (blo\_tr - \frac{blo\_tr-i}{i+1}) \cdot \frac{rot}{blo\_tr} + \frac{rot}{2 \cdot blo\_tr} \quad [msec]$$

$$1 \leq j \leq tr\_cyl, \; 1 \leq i \leq blo\_tr$$

Zeitaufwand read_cyl, um einen ganzen Zylinder zu lesen:
Diese Kosten setzen sich zusammen aus
- Zeit, bis erster Blockbeginn von erster Spur zum Lesen unter den Lesekopf rotiert: $\frac{rot}{2 \cdot blo\_tr}$
- tr_cyl Rotationen zum Lesen von tr_cyl Spuren.

(M6) $$\text{read\_cyl} = \left(\frac{1}{Z \cdot \text{blo\_tr}} + \text{tr\_cyl} \right) \cdot \text{rot} \quad [\text{msec}]$$

Damit stehen alle Grundelemente für eine Modellierung der LI-Filteroperationen zur Verfügung. Abschließend geben wir noch einige charakteristische Eigenschaften des soeben entwickelten Modells.

Lemma 5.1: (Spezialfälle elementarer DB-Plattenoperationen)

(a) read_bl(blo_tr) = read_tr

(b) read_tr(1,i) = read_bl(i)

(c) read_tr(tr_cyl,blo_tr) = read_cyl

Beweis: (b := blo_tr, t := tr_cyl)

(a) $\text{read\_bl}(b) = \left(b + \frac{1}{Z} \right) \cdot \frac{\text{rot}}{b} = \left(1 + \frac{1}{Z \cdot b} \right) \cdot \text{rot} = \text{read\_tr}$

(b) klar

(c) $\text{read\_tr}(t,b) = j \cdot \left(b - 0 \right) \cdot \frac{\text{rot}}{b} + \frac{\text{rot}}{Z \cdot b} = \text{rot} \cdot \left(j + \frac{1}{Z \cdot b} \right) = \text{read\_cyl}$

5.1.3. Zugriffszeitverhalten komplexer DB-Plattenoperationen.

Die Autonomie und Asynchronität jeder LI gestattet es, komplexe DB-Plattenoperationen atomar auszuführen. Diese Operationen werden zur effizienten Implementierung der SI-Operatoren EXH_FILTER und SEL_FILTER benötigt. Die Retrieval-Schnittstelle zwischen einer LI und dem zugeordneten DB-Plattenstapel besteht aus den folgenden drei unteilbaren Operationen:

(1) Lesen eines bestimmten Blocks (random Blocklesen).

(2) Vollständiges Lesen eines kompletten Bereichs.

(3) Selektives Lesen einiger Blöcke eines Bereichs.

Lesen in diesem Zusammenhang bedeutet eine Datenübertragung von der DB-Platte in den LI-Arbeitsspeicher, wo in den Fällen (2) und (3) die Datenfilterung stattfindet. Die Unteilbarkeitseigenschaft für die komplexen Operationen (2) und (3) ist unerläßlich für eine schnelle Ausführung. Neue

SI-Aufträge vom Host an eine LI können laufende Typ(2)- und Typ(3)-Operationen nicht unterbrechen, und somit auch nicht den Plattenarm an ungünstige Positionen manövrieren.
Im folgenden erstellen wir ein Modell zur Ermittlung des Zeitaufwands für die genannten drei Operationen.

Zeitaufwand bat für random Blocklesen: (block access time)
Die Kosten für I/O von der DB-Platte in den Arbeitsspeicher der LI belaufen sich auf:

(M7) $$bat = avg\_seek + read\_bl(1) \quad [msec]$$

Zeitaufwand exh_at(n) für vollständiges Lesen eines Bereichs:
Der Parameter n gibt an, wieviel Zylinder von dem betreffenden Bereich belegt sind. Dabei wird angenommen, daß alle n·blo_cyl Blöcke dieses Bereichs auszulesende Daten enthalten. Der gesuchte Zeitaufwand setzt sich wie folgt zusammen:
- Positionierung auf ersten Zylinder des Bereichs.
- Lesen des ersten Zylinders.
- Für alle weiteren Zylinder des Bereichs:
  * Positionieren auf nächsten Zylinder.
  * Lesen dieses Zylinders.

Unter Berücksichtigung der physisch sequentiellen Speicherung von Bereichen ergibt sich:

(M8) $$exh\_at(n) = avg\_seek + (n-1)\cdot seek\_cyl(1) + n\cdot read\_cyl \quad [msec]$$
$$1 \le n \le cyl\_dp$$

Zeitaufwand sel_at(n,m) für selektives Lesen eines Bereichs:
Parameter: n = Zylinderanzahl des Bereichs, (wie bei (M8))

m ≡ Anzahl der selektiv aus diesem Bereich zu lesenden Blöcke, $1 \leq m \leq n \cdot blo\_cyl$.

Für die Berechnung von sel_at(n,m) ist natürlich die Kenntnis der Verteilung der m Blöcke auf die n Zylinder notwendig. Dazu treffen wir folgende Annahme:

Die Verteilung der m Blöcke auf die n Zylinder ist zufällig.

Sei k die durchschnittliche Anzahl von Zylindern (von den n Zylindern), in denen mindestens einer der m Blöcke liegt. Der gesuchte Zeitaufwand sel_at(n,m) errechnet sich jetzt aus:

- Positionierung auf ersten der k Zylinder.
- Selektives Lesen der betroffenen Blöcke dieses Zylinders.
- Für alle weiteren der k Zylinder:
  - * Positionieren auf nächsten (der k) Zylinder.
  - * Selektives Lesen der betroffenen Blöcke dieses Zylinders.

Die Berechnung von k ist äquivalent zur Lösung folgenden Urnenproblems:

Gegeben seien n Urnen mit je blo_cyl Kugeln. Man mache m zufällige Zugriffe ohne Zurücklegen, bei leerer Urne wird der Versuch wiederholt, $1 \leq m \leq n \cdot blo\_cyl$. Wieviel Urnen sind im Durchschnitt von mindestens einem Zugriff betroffen?

Eine exakte Lösung dieses Problems wurde z.B. in [YAO77] diskutiert.[89] Da die numerische Auswertung dieser exakten Formel relativ teuer ist, sind einfache Näherungslösungen von Interesse. Eine solche Formel, welche die Komplikation 'ohne Zurücklegen' außer acht läßt, wurde vorher in [CARD75] erwähnt:

$$k = n \cdot (1 - (1 - \frac{1}{n})^m)$$

In [YAO77] wurden die Abweichungen dieser Cardenas-Formel von der exakten Lösung untersucht. Dabei stellte sich heraus, daß der Fehler der Cardenas-Formel für die Fälle, bei denen sich mehr als 10 Kugeln in jeder Urne befinden, praktisch vernachlässigt werden kann. Gerade dieser Umstand trifft bei uns zu, da in der Praxis ein Plattenstapel aus 10-20 Platten-

[89] Yao betrachtete das Problem der zufälligen Verteilung von Tupel auf Blöcke.

oberflächen besteht, und somit blo_cyl>10 gilt (typischerweise gilt blo_cyl>100). Die Cardenas-Approximation liefert somit für k fast exakte Werte[90] , was sehr wesentlich ist, da k bei weitem der größte Kostenfaktor für sel_at(n,m) ist.
Wir definieren daher:

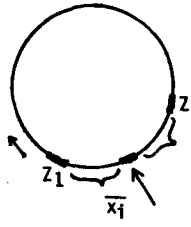
(M9)
$$k := \begin{cases} n\cdot(1-(1-\frac{1}{n})^m) & \text{,falls } 1 \le m < n\cdot blo\_cyl \\ n & \text{, falls } m = n\cdot blo\_cyl \end{cases}$$

In jedem der k Zylinder befindet sich also mindestens je einer der m Blöcke. Für die Verteilung der restlichen m-k Blöcke auf diese Zylinder nehmen wir als gute Approximation wieder die Binomialverteilung (d.h. mit Zurücklegen). Damit erhält man: Die durchschnittliche Anzahl von Blöcken, die in jedem der k Zylinder gelesen werden muß, beträgt $1 + \frac{m-k}{k} = \frac{m}{k}$.
Für die Ermittlung des Zeitaufwands zum selektiven Lesen von $\frac{m}{k}$ Blöcken aus einem Zylinder benötigen wir noch die Verteilung dieser $\frac{m}{k}$ Blöcke auf die tr_cyl Spuren. Dies ereichen wir dadurch, indem wir das gewählte Modell zur Ermittlung von k rekursiv auf die Auswahl von Spuren innerhalb eines Zylinders übertragen.
Sei l die durchschnittliche Anzahl von Spuren (in einem der k Zylinder), von denen mindestens ein (von den $\frac{m}{k}$) Block gelesen werden muß.
Unter Berücksichtigung, daß blo_cyl = blo_tr·tr_cyl, ergibt sich:

(M10)
$$l := \begin{cases} tr\_cyl\cdot(1-(1-\frac{1}{tr\_cyl})^{\frac{m}{k}}) & \text{, falls } \frac{m}{k} < blo\_cyl \\ tr\_cyl & \text{, falls } \frac{m}{k} = blo\_cyl \end{cases}$$

90) Ein anderes Modell zur Berechnung von read bl(i), read tr(j,i) und somit sel_at(n,m) wäre wie folgt: Anstatt zu fordern, daß genau j Spuren und i Blöcke zu lesen sind, verlangt man nur, daß im Mittel j Spuren und i Blöcke zu lesen sind. Für dieses Modell liefert die Cardenas-Formel sogar die exakte Lösung, da dann die Annahme 'ohne Zurücklegen' gerechtfertigt ist.

Schließlich stellen wir analog zur Herleitung von $\frac{m}{k}$ fest, daß in jeder der l Spuren im Mittel $1 + \frac{1}{l}\cdot(\frac{m}{k} - 1) = \frac{m}{k\cdot l}$ Blöcke gelesen werden müssen.
Als letzter Punkt verbleibt nun noch die Ermittlung des Zeitaufwands für die Positionierung von einem der k Zylinder auf den nächstliegenden dieser k Zylinder. Dazu läßt sich wieder das bei der Berechnung von read_bl(i) und read_tr(j,i) benützte Modell anwenden:



$\overline{x_i} = \frac{n-k}{k+1}$ nicht relevante Zylinder

Ganz gleich wie die k Zylinder $Z_1,\ldots,Z_k$ auf die n Zylinder verteilt sind, im Mittel beträgt die Anzahl der Zylinder von Z_1 bis Z_k (jeweils einschließlich)

$$n - 2\cdot\frac{n-k}{k+1} - 1$$

Also ergibt sich für k>1:

$$\text{avg\_dist}(k) = \frac{1}{k-1}\cdot(n - 2\cdot\frac{n-k}{k+1} - 1) = \frac{n+1}{k+1}$$

(M11)
$$\text{avg\_dist}(k) := \begin{cases} \frac{n+1}{k+1} & \text{, falls } 1 < k \leq n \\ 0 & \text{, für } k = 1 \end{cases}$$

Die Summation der einzelnen Zeitanteile ergibt:

$$\text{sel\_at}(n,m) = \text{avg\_seek} + \text{read\_tr}(l, \tfrac{m}{k\cdot l}) + (k-1)\cdot(\text{seek\_cyl}(\text{avg\_dist}(k)) + \text{read\_tr}(l, \tfrac{m}{k\cdot l}))$$

(M12)

$$sel\_at(n,m) = \begin{cases} avg\_seek + (k-1)\cdot seek\_cyl(avg\_dist(k)) + k\cdot read\_tr(1, \frac{m}{k\cdot 1}) & [msec],\ 0<m \le n\cdot blo\_cyl \\ 0 \quad [msec] & , \text{falls } m=0 \end{cases}$$

$$1 \le n \le cyl\_dp$$

Lemma 5.2: (Spezialfälle komplexer DB-Plattenoperationen)

(a) sel_at(n,1) = bat

(b) sel_at(n,n·blo_cyl) = exh_at(n)

Beweis:

(a) $m = 1 \Rightarrow k = n\cdot(1-(1-\frac{1}{n})^1) = 1 \Rightarrow \frac{m}{k} = 1 \Rightarrow 1 = 1.$

Somit: sel_at(n,1) = avg_seek + read_tr(1,1) = (* L.5.1(b) *)

avg_seek + read_bl(1) = bat.

(b) (bc := blo_cyl, tc := tr_cyl)

$m = n\cdot bc \Rightarrow k = n \Rightarrow \frac{m}{k} = bc \Rightarrow 1 = tc \Rightarrow$

$\frac{m}{k\cdot 1} = \frac{n\cdot bc}{n\cdot tc} = blo\_tr.$

Außerdem gilt: $k = n \Rightarrow avg\_dist(k) = 1$

Somit:

sel_at(n,n·bc) = avg_seek + (n-1)·seek_cyl(1) +

n·read_tr(tc,blo_tr) = (*L.5.1(c) *)

avg_seek + (n-1)·seek_cyl(1) + n·read_cyl = exh_at(n).

Satz 5.1:

(a) sel_at(n,m) ist streng monoton wachsend in m.

(b) sel_at(n,m) < exh_at(n) für $0 \le m < n\cdot blo\_cyl$

Beweis:

(a) Siehe Anhang.

(b) Folgt unmittelbar aus (a) und L.5.2(b).

Ergänzend zu Satz 5.1 sei (ohne expliziten Beweis) angemerkt, daß sel_at(n,m) < m·bat für 1 < m ≤ n·blo_cyl gilt. Diese plausible Tatsache ergibt sich aus folgender Überlegung: Falls man die m einzelnen random Blockzugriffswünsche aufsammelt, die Plattenadressen sortiert und danach alle m Zugriffe hintereinander ausführt, so ergibt sich offensichtlich das Zeitverhalten von sel_at(n,m), da dann die Armbewegungen minimiert worden sind.[91] Durch ähnliche zusätzliche Software zur Minimierung der Plattenarmbewegungen wird in konventionellen DB-Systemen versucht, die Nachteile der Einzelverarbeitungsweise zu mildern (es können natürlich nur Blockrequests verschiedener Transaktionen optimiert werden). Dennoch ist der Vorteil, den unser Intelligentes Speichersystem in Verbindung mit der Mengenverarbeitung auch bei adressiertem Zugriff bietet, deutlich zu sehen.

5.2. Kostenmodell für Restriktionsauswertung.

Das Ziel dieses Kapitels ist die Entwicklung von für unsere DB-Architektur adäquaten Kostenfunktionen, welche dann in Kap.5.3 zur Optimierung von Projektion-Restriktion Queries der Gestalt $\pi_{r_1 \ldots r_h}\sigma_F(R)$ herangezogen werden sollen.

Das Kostenmodell.

In DB-Systemen ohne Backend-Rechner wie SystemR oder INGRES werden die Kosten einer Queryauswertung hauptsächlich aus der geschätzten Anzahl von notwendigen Blockzugriffen auf die DB-Platten ermittelt. Der Optimierer in SystemR etwa benutzt eine gewichtete Summe aus geschätzten Blockzugriffen

[91] Falls wie in konventionellen DB-Systemen üblich, die m Blöcke über die ganze Platte verstreut sind, dann ist sel_at(n,m) in unserer Architektur trotzdem etwas günstiger aufgrund der bereichsmäßigen Plattenbelegung.

und CPU-Aufwand. Dieses Kostenmodell ist jedoch auf unsere DB-Architektur nicht übertragbar infolge der komplexen unteilbaren DB-Plattenoperationen, deren Zugriffszeitverhalten sich erheblich von konventionellen DB-Plattenzugriffen unterscheidet (vgl. Bem. nach S.5.1).
Um ein analytisch einfach behandelbares Kostenmodell zu bekommen, wollen wir folgende Kostenfaktoren vernachlässigen:

- Host-CPU-Kosten werden außer acht gelassen, da wir annehmen, daß die LIs die Anforderungen an die Rechenleistung des Hosts soweit verringern, daß beim Host kein CPU-Engpaß auftritt.
- Kommunikationskosten sowie Datenübertragungskosten zwischen Host und LIs werden nicht betrachtet.

Somit wählen wir als Kostenfaktor den Zeitaufwand für die DB-Plattenoperationen (random Blocklesen, selektives Bereichslesen, vollständiges Bereichslesen).

Zugriffspfadtypen.

Wir wollen die Kosten der Auswertung von $\pi_{r_1 \ldots r_h}\sigma_F(R)$ untersuchen. Dieser RelA-Ausdruck ist äqivalent zu der SQL-Query

```
Q_restr = SELECT UNIQUE r1,...,rh
          FROM   R
          WHERE  F
```

wobei F eine Restriktionsformel auf R ist.
Wie bereits in Kap.4 dargelegt, verfügen wir über zwei wesentlich unterschiedliche Typen von Zugriffspfaden:

Zugriffspfadtyp1: (Vollständige Filterung)

$ZER_R := EXH\pi_{r_1 \ldots r_h}\sigma_F(R)$

Zugriffspfadtyp2: (Indexverarbeitung & selektive Filterung)
Sei $F = F^+ \wedge F^-$ eine Zerlegung derart, daß F^+ über Indexe auswertbar ist.[92]

$Id_R := IND_{F^+}(R);$

[92] Das Problem, wie man zu einer 'guten' solchen Zerlegung kommt, sei hier ausgeklammert.

$$ZER_R := SEL_{\pi r_1 \ldots r_h \sigma_{F^-}}(Id_R).$$

Zugriffspfadtyp1 steht dabei immer zur Verfügung.
In die anschließend zu definierenden Kostenfunktionen gehen die folgenden Kostenparameter ein:

(1) a ≡ Anzahl der von LI in das DB-Cache zu transportierenden Indexknoten zur Auswertung von F^+.
(2) m ≡ Anzahl der Blöcke, in denen Tupel von R, welche F^+ erfüllen, gespeichert sind.
(3) n ≡ Anzahl der Zylinder, die von R belegt sind.

Dabei treffen wir die Zusatzannahme, daß R in einem Segment liegt, welches aus nur einem Bereich besteht, und daß R stets volle Zylinder belegt.
Für die Erstellung der Kostenfunktionen für die beiden Zugriffspfadtypen legen wir eine weitere Einschränkung bzgl. der Komplexität von F fest (vgl. auch die Diskussion über dyn. Filter): F (und somit auch F^-) ist durch die LI im Flug auswertbar.
Die Filterung von Daten im LI-Arbeitsspeicher ist somit mit voller Plattengeschwindigkeit möglich, d.h. beim Füllen der Wechselpuffer mit rohen Daten von der DB-Platte entstehen keine Wartezeiten, und somit können die komplexen DB-Plattenoperationen ohne Verlust von Rotationen abgewickelt werden. Aufgrund dieser Überlegungen wählen wir folgende Kostenfunktionen:

Kosten für Zugriffspfadtyp1:

$$cost1(n) := exh\_at(n) \quad [msec]; \quad 1 \leq n \leq cyl\_dp$$

Kosten für Zugriffspfadtyp2:

$$cost2(n,a,m) := a.bat + sel\_at(n,m) \quad [msec]; \quad 1 \leq n \leq cyl\_dp,\ 1 \leq m \leq n \cdot blo\_cyl$$

Anmerkung:
cost2(n,a,m) ist streng monoton wachsend in a und m (vgl. S.5.1(a)).

5.3. Optimierung von Restriktionsauswertungen.

Einleitend wollen wir kurz einen Hinweis darauf geben, wo ein Optimierer die in diesem Kapitel dargelegten Optimierungsalgorithmen einsetzen kann. Außer zur Optimierung einer einfachen Projektion-Restriktion Query sind folgende weitere Anwendungen möglich:

- Optimierung von Q_{select}-Queries:

  Beispiel: Q ≡ SELECT UNIQUE b1
  FROM B
  WHERE b2 < MAX(SELECT d1
  FROM D — Q_{in}
  WHERE d2>7)

  Die Auswertung kann von innen nach außen erfolgen:
  (1) $Eval(Q_{in}) \equiv Eval(\pi_{d1}\sigma_{d2>7}(D))$;
  (2) Berechnung von MAX, das Ergebnis sei d1max;
  (3) $Eval(Q) \equiv Eval(\pi_{b1}\sigma_{b2<d1max}(B))$.
  Bei (1) und (3) ergibt sich somit das zu untersuchende Problem der Optimierung einer Projektion-Restriktion Query.
- An all denjenigen Stellen in den in Kap.4.2.3 entwickelten Algorithmen für Q_{set}-, Q_{corr}- und Q_{join}-Queries, welche den Term $Eval(\sigma_F(R))$ betreffen (für entspr. F und R).

5.3.1. Auswahl des schnellsten Zugriffspfadtyps.

Unser Optimierungsziel ist die Minimierung des Zeitaufwands für I/Os von der DB-Platte in den LI-Arbeitsspeicher. Für die Query $\pi_{r_1 \ldots r_h}\sigma_F(R)$ soll nun

anhand der Kostenparameter a und m entschieden werden, welcher der beiden zur Verfügung stehenden Zugriffspfadtypen effizienter ist.
Betrachten wir den Fall gleicher Kosten cost1(n) = cost2(n,a*,m) bei festen n und m. Der Grenzpunkt a* berechnet sich dann zu

$$a^* = \frac{exh\_at(n)-sel\_at(n,m)}{bat}$$

Aus S.5.1(b) und L.5.2(b) wissen wir, daß exh_at(n) - sel_at(n,m) ≥ 0 gilt. Also ist a* bei festem n und m eindeutig definiert und liegt im Intervall $[0, \frac{exh\_at(n)}{bat}]$.

Als Grenzfunktion $ta_n(m)$ für Indexverwendung definieren wir

$$ta_n(m) := \frac{exh\_at(n) - sel\_at(n,m)}{bat}, \quad 0 \leq m \leq n \cdot blo\_cyl$$

(threshold value for a).

Eigenschaften von $ta_n(m)$:

(1) Für gegebenes m und n bestimmt $ta_n(m)$ den zugehörigen Grenzpunkt a*, d.h. es gilt:
- cost1(n) = cost2(n,$ta_n(m)$,m)

(2) - $ta_n(0) = \frac{exh\_at(n)}{bat} > 0$
- $ta_n(n \cdot blo\_cyl) = 0$

(3) $ta_n(m)$ ist streng monoton fallend, da nach S.5.1(a) sel_at(n,m) streng monoton steigt in m.

Satz 5.2: (Optimierungskriterium 1 für Q_{restr}-Queries)
Sei Relation R in n Zylindern abgespeichert, $\overline{m}$ die Blocktreffer für ein F^+ auf R, sowie $\overline{a}$ die Indexkosten zur Auswertung dieses F^+.
(a) $\overline{a} < ta_n(\overline{m})$ ==> Indexverarbeitung & selektive Filterung ist schneller
(b) $\overline{a} > ta_n(\overline{m})$ ==> Vollständige Filterung ist schneller

Beweis: Sei $a^* = ta_n(m)$.
(a) $\overline{a} < a^*$ ==> $cost2(n,\overline{a},\overline{m}) < cost2(n,a^*,\overline{m})$, da cost2 streng monoton steigt in a. Wegen $cost2(n,a^*,\overline{m}) = cost1(n)$ ist Aussage (a) somit richtig.
(b) Analog.

Anmerkung: Das Problem der Ermittlung von Schätzwerten für a und m wird in Kap.5.4.3.1 behandelt.

Graphische Illustration von Satz 5.2 (schematisch, nur das monotone Fallen von $ta_n(m)$ gegen 0 soll dargestellt werden):[93]

[93] $ta_n(m)$ ist in Wirklichkeit nicht linear.

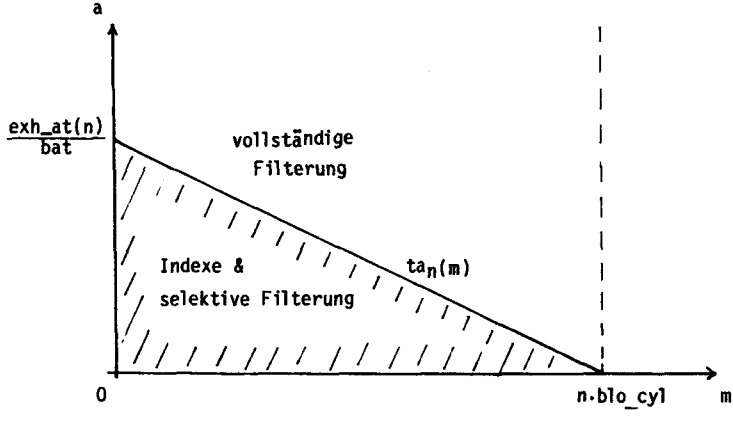


Abb.5.1: Q_{restr} - Optimierung mittels $ta_n(m)$.

Als unmittelbare Anwendung von Satz 5.2 geben wir einen Vergleich mit Zugriffspfaden zur Restriktionsauswertung, wie man es etwa in SystemR vorfindet.

SystemR kennt drei verschiedene Typen von Zugriffspfaden, den Segment-Scan, den Cluster-Index und den Nonclustered-Index; Segmente in SystemR sind beliebig über die Platte verstreut und können mehrere Relationen beinhalten. Betrachten wir die Auswertung der Query $Q \equiv \sigma_{r1>7}(R)$, welche die Ungleichheitsrestriktion r1>7 beinhaltet. Angenommen, es existiere ein Nonclustered-Index auf r1 und ein Cluster-Index auf r2. In SystemR tritt nun sehr häufig der Fall ein, daß die Verwendung des Cluster-Index auf r2 ohne jegliche Restriktion der effizienteste Zugriffspfad ist ([ASTR80]).

Diese Zugriffsmethode sei als vollständiger Cluster-Indexscan bezeichnet. Es müssen alle Blöcke mit Tupeln von R des betreffenden Segments in den DB-Puffer geholt werden; die Anzahl ist also relativ zu unserer Organisation gleich n·blo_cyl =: $\overline{m}$. Wegen $ta_n(\overline{m}) = 0$ folgt $\overline{a} > ta_n(m)$, falls während des vollständigen Cluster-Indexscans zur Ermittlung der relevanten Blöcke des

betreffenden Segments mindestens ein benötigter Indexknoten erst von der DB-Platte gelesen werden muß. Übertragen auf unsere Architektur heißt das, daß vollständige Indexscans nicht mehr in Frage kommen (was natürlich auch ohne S.5.2 aufgrund unserer DB-Plattenorganisation plausibel ist). Diese Feststellung unterstützt nachträglich die Wahl unserer Plattenorganisation (unsortiert!).

Betrachten wir als weitere Anwendung folgenden Fall. Falls in SystemR kein Index zur Auswertung einer Restriktion existiert, so wird manchmal die Einrichtung eines temporären Indexes über dem betreffenden Attribut erwogen. Konkret tritt dieses Problem dann auf, wenn diese Restriktion im inneren Queryblock einer Q_{corr}-Query liegt und somit bei der iterierten Tupelsubstitution wiederholt ausgewertet werden muß. Für den Aufbau des temporären Indexes müssen alle $\overline{m}$ = n·blo_cyl Blöcke mit Tupeln der betreffenden Relation R gelesen werden. Somit gilt auch hier $\overline{a} > ta_n(\overline{m}) = 0$, falls mindestens ein benötigter Indexknoten nach zwischenzeitlicher Verdrängung erst wieder von der DB-Platte gelesen werden muß; zusätzlich sind i.a. erneute Zugriffe auf Blöcke mit Tupeln von R für die Indexauswertung nötig.

Diese Beobachtung unterstreicht nochmals die Effizienz des in Kap.4.2.3.2 entwickelten Algorithmus für Q_{corr}-Queries. Ergänzend zur Diskussion von Kap.4.1.2 über sinnvolle Optimierungsstrategien in unserer Architektur halten wir fest:

Korollar 5.1:
Die Einrichtung temporärer Indexe lohnt sich in unserer Architektur nicht mehr.

5.3.2. Indexauswahl bei konjunktiver Zerlegung.

Wir wollen nun einen wichtigen Spezialfall unserer allgemeinen Projektion-

Restriktion-Query $Q_{restr} \equiv \pi_{rl...rh}\sigma_F(R)$ untersuchen, nämlich für folgende Zerlegung von F:

$F \equiv F^+ \wedge F^-$ mit $F^+ \equiv F_1^+ \wedge F_2^+$ (konjunktive Zerlegung von F^+)

F_1^+ und F_2^+ sollen dabei verschiedene Attribute von R betreffen.

SQL-Form: Q_{conj} ≡ SELECT *
FROM R
WHERE F_1^+ AND F_2^+ AND F^-

FÜr die Auswertung dieser Query stehen in unserer DB-Architektur folgende vier Zugriffspfade zur Auswahl:

- (AP1) Vollständige Filterung
- Drei verschiedene Zugriffspfade des Typs Indexverarbeitung & selektive Filterung:

  $(AP2_1)$ $Id_R := IND_{F_1^+}(R)$; $ZER_R := SEL_{\pi_{rl...rh}\sigma_{F_2^+ \wedge F^-}}(Id_R)$

  $(AP2_2)$ $Id_R := IND_{F_2^+}(R)$; $ZER_R := SEL_{\pi_{rl...rh}\sigma_{F_1^+ \wedge F^-}}(Id_R)$

  $(AP2_{1 \wedge 2})$ $Id_R := IND_{F_1^+ \wedge F_2^+}(R)$; $ZER_R := SEL_{\pi_{rl...rh}\sigma_{F^-}}(Id_R)$

Die Kostenparameter für die Indexauswertung von F_i^+ seien a_i und m_i, i=1,2.
Die Kosten für die aufgezählten Zugriffspfade belaufen sich dann auf:[94]

- cost_AP1 = cost1(n)
- $cost\_AP2_1$ = cost2(n,a_1,m_1)
- $cost\_AP2_2$ = cost2(n,a_2,m_2)
- Ermittlung von $cost\_AP2_{1 \wedge 2}$:
  - Anzahl benötigter Indexknoten: $a_1 + a_2$
  - Anzahl von Trefferblöcken für selektive Filterung (unter der Annahme der statistischen Unabhängigkeit von F_1^+ und F_2^+):

[94] Zur Erinnerung:
- F ist im Flug filterbar, somit auch F^+, F_1^+ sowie F_2^+ und auch F^-.
- cost1(n) = exh_at(n),
- cost2(n,a,m) = a-bat + sel_at(n,m),
- n-blo_cyl ≡ Anzahl der Blöcke in einem n-Zylinder Bereich,
- Relation R ist in einem n-Zylinder Bereich abgespeichert.

$\frac{m_i}{n.blo\_cyl}$ gibt die Wahrscheinlichkeit an, daß ein Block der Relation R Trefferblock aufgrund der Indexauswertung von F_i^+ ist; i=1,2. Somit ist die Wahrscheinlichkeit, daß ein Block von R Trefferblock von $F_1^+ \wedge F_2^+$ ist, gegeben durch

$$\frac{m_1}{n.blo\_cyl} \cdot \frac{m_2}{n.blo\_cyl}$$

====> die gesuchte Anzahl von Trefferblöcken beträgt $\frac{m_1 \cdot m_2}{n.blo\_cyl}$

Für $cost\_AP2_{1 \wedge 2}$ erhält man somit:

$$cost\_AP2_{1 \wedge 2} = cost2(n, a_1+a_2, \frac{m_1 \cdot m_2}{n.blo\_cyl})$$

Satz 5.2 liefert nun sofort:

(0) $a_2 < ta_n(m_2)$ ==> $AP2_2$ ist schneller als AP1.

(1) $a_1 < ta_n(m_1)$ ==> $AP2_1$ ist schneller als AP1.

(2) $a_1 + a_2 < ta_n(\frac{m_1 \cdot m_2}{n.blo\_cyl})$ ==> $AP2_{1 \wedge 2}$ ist schneller als AP1.

Neue Probleme bei der Auswahl des schnellsten Zugriffspfades treten genau dann ein, wenn sowohl (0) oder (1) als auch (2) zutreffen. Für diese Situation werden herkömmlich meist Optimierungsheuristika eingesetzt wie z.B. "Verwende alle in Frage kommenden Indexe" oder "Verwende den selektivsten Index" (vgl. etwa RDB/V1, [MAKI81]). Für das weitere nehmen wir an, daß $m_1 \leq m_2$ gilt und daß $AP2_1$ schneller als $AP2_2$ ist.

Mit den bisherigen Mitteln läßt sich noch keine Entscheidung zwischen $AP2_1$ und $AP2_{1 \wedge 2}$ im vorliegenden Fall treffen, da $a_1 < a_1 + a_2$, jedoch $m_1 \geq \frac{m_1 \cdot m_2}{n.blo\_cyl}$ gilt, und somit die Monotonie-Eigenschaft von cost2 nicht zum Tragen kommt.

Für den Fall, daß sowohl (1) als auch (2) zutreffen, vergleichen wir die *entsprechenden Differenzen* zu AP1:

ad (1): $ta_n(m_1) - a_1 = \frac{1}{bat} \cdot (exh\_at(n) - sel\_at(n, m_1)) - a_1$

ad (2): $ta_n(\frac{m_1 \cdot m_2}{n.blo\_cyl}) - (a_1 + a_2) \approx$

$$\frac{1}{bat}\cdot(exh\_at(n) - sel\_at(n, \frac{m_1 \cdot m_2}{n \cdot blo\_cyl})) - (a_1 + a_2)$$

Diese Unterschiede sind gleich groß, wenn gilt:

$$a_2 = \frac{1}{bat} \cdot(sel\_at(n,m_1) - sel\_at(n, \frac{m_1 \cdot m_2}{n \cdot blo\_cyl}))$$

Für diesen Fall bezeichnen wir dieses a_2 als Grenzpunkt a_2^* für die Indexauswertung von F_2^+ mit Kostenparameter m_2. Als Grenzfunktion für zweifache Indexauswertung bei konjunktiver Zerlegung definieren wir:

$$t2a_{n,m_1}(m) := \frac{1}{bat} \cdot(sel\_at(n,m_1) - sel\_at(n, \frac{m_1 \cdot m}{n \cdot blocyl}))$$

$$m_1 \leq m \leq n \cdot blo\_cyl$$

Eigenschaften von $t2a_{n,m_1}(m)$:

(1) Für gegebenes n, m_1, a_1 und m bestimmt $t2a_{n,m_1}(m)$ den zugehörigen Grenzpunkt a_2^*, d.h. es gilt:

- $cost2(n,a_1,m_1) = cost2(n,a_1 + t2a_{n,m_1}(m), \frac{m_1 \cdot m}{n \cdot blo\_cyl})$

(Die Festsetzung $m_1 \leq m$ ist wieder darin begründet, daß $m_1 \leq m_2$ ggf. durch Umnummerierung erhalten werden kann.)

(2) - $t2a_{n,m_1}(m_1) \geq 0$
- $t2a_{n,m_1}(n \cdot blo\_cyl) = 0$

(3) $t2a_{n,m_1}(m)$ ist streng monoton fallend, da $m_1 \geq \frac{m_1 \cdot m}{n \cdot blo\_cyl}$ und sel_at streng monoton steigt im 2.Argument.

Satz 5.3:
Sei Relation R in n Zylindern abgespeichert, F_1^+ und F_2^+ Restriktionen auf R mit den Kostenparametern a_1 und m_1 bzw. a_2 und m_2, $m_1 \leq m_2$, dann gilt:
(a) $a_2 < t2a_{n,m_1}(m_2) \longrightarrow AP2_1 \wedge 2$ ist schneller als $AP2_1$
(b) $a_2 > t2a_{n,m_1}(m_2) \longrightarrow AP2_1$ ist schneller als $AP2_1 \wedge 2$

Beweis:
Sei $a_2^* = t2a_{n,m_1}(m_2)$, $\overline{m} = \frac{m_1 \cdot m_2}{n \cdot blo\_cyl}$.
(a) $a_2 < a_2^* \longrightarrow$

$$cost\_AP2_1 \wedge 2 = cost2(n,a_1 + a_2,\overline{m}) < cost2(n,a_1 + a_2^*,\overline{m})$$
$$= cost2(n,a_1,m_1) = cost\_AP2_1.$$

(b) Analog.

Graphische Illustration von Satz 5.3 (wiederum nur schematisch):

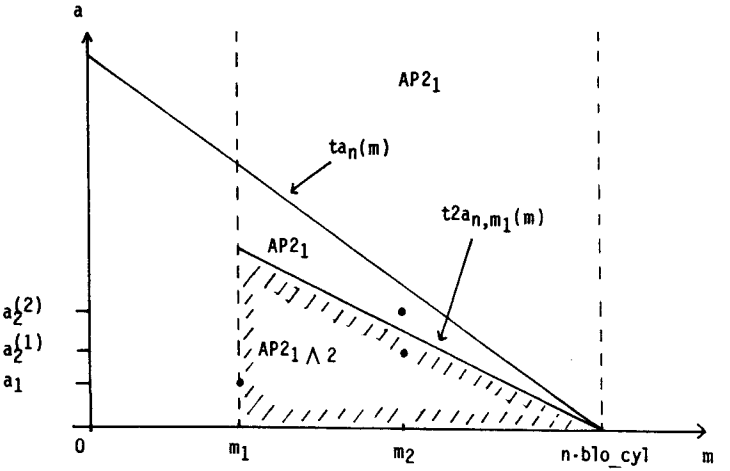


Abb. 5.2: Q_{conj}-Optimierung mittels $ta_n(m)$ und $t2a_{n,m_1}(m)$.

Die Blocktreffer für F_1^+ und F_2^+ seien m_1 bzw. m_2. Auf der a-Achse sind die Indexkosten für F_1^+ (a_1) sowie zwei Fälle für die Indexkosten von F_2^+ aufgezeichnet ($a_2^{(1)}$, $a_2^{(2)}$). Da $a_1 < ta_n(m_1)$, ist die Indexauswertung von F_1^+ mit nachfolgender selektiver Filterung gemäß $F_2^+ \wedge F^-$ auf alle Fälle schneller als die vollständige Filterung.

Fall (1): Indexkosten für F_2^+ sind $a_2^{(1)}$

$\longrightarrow$ Indexauswertung von F_1^+ und F_2^+ ist am schnellsten.

Fall (2): Indexkosten für F_2^+ sind $a_2^{(2)}$

$\longrightarrow$ Nur Indexauswertung für F_1^+ ist am schnellsten.

Abschließend sei noch der Fall einer disjunktiven Zerlegung von F^+ in $F^+ = F_1^+ \vee F_2^+$ erwähnt. Dabei seien F_1^+ und F_2^+ wieder allgemeine Restriktionsformeln auf disjunkten Attributmengen von R, welche über Indexe auswertbar sind.

Für die Query

$Q_{disj} \equiv$ SELECT *
FROM R
WHERE (F_1^+ OR F_2^+) AND F^-

stehen nur zwei (sinnvolle) Zugriffspfade zur Auswahl:

- (AP1) Vollständige Filterung
- ($AP2_{1 \vee 2}$) $Id_R := IND_{F_1^+ \vee F_2^+}(R)$;
  $ZER_R := SEL_{*\sigma_{F^-}}(R)$

Kostenparameter für $AP2_{1 \vee 2}$:

- $a_1 + a_2$ für benötigte Indexknoten
- Anzahl der Trefferblöcke (statistische Unabhängigkeit vorausgesetzt):

$$m_1 + m_2 - \frac{m_1 \cdot m_2}{n \cdot blo\_cyl}$$

Zur Optimierung läßt sich dann direkt Satz 5.2 anwenden.

5.4. Kriterien für die Einrichtung von Sekundärindexen.

5.4.1. Problemstellungen beim physischen DB-Entwurf.

Das Problem einer optimalen Organisation der physischen Datenbank ist der Kategorie der Low-level Optimierungsprobleme zuzurechnen, wobei wiederum die Erstellung und Minimierung konkreter Kostenfunktionen eine wichtige Rolle einnimmt. Der Optimierung beim physischen DB-Entwurf wurde in der Literatur große Aufmerksamkeit gewidmet, vgl. etwa [SCHK75], [HAMM76] oder [WHAN81]. Traditionell beinhaltet dieser Komplex u.a. die folgenden wichtigen Aspekte:

(1) Auswahl eines geeigneten physischen Speicher-Layouts zur Abspeicherung von Relationen auf den DB-Platten, verbunden mit der Festlegung eines Primärzugriffpfades.

(2) Auswahl geeigneter zusätzlicher Sekundärzugriffpfade.

Bei Designproblem (1) stand bisher zumeist die Auswahl eines Cluster-Attributs im Vordergrund, da die Relationen sortiert nach einem Attribut (zumeist dem Schlüsselattribut) auf die DB-Platten abgebildet wurden. Sortierte Abspeicherung erlaubt die Einrichtung eines Primärzugriffpfades sowohl über dichte als auch über nicht-dichte Indexe. Designproblem (2) hat die Aufgabe zu lösen, solche Attribute zu bestimmen, für die sich die Einrichtung eines Sekundärzugriffspfades "lohnt", d.h. der Effizienzgewinn bei Lesezugriffen über Sekundärindexe muß die Indexeinrichtungskosten (zusätzlicher Speicherplatz) sowie Indexwartungskosten (infolge Tupel-Updates/Einfügungen/Löschungen) signifikant kompensieren.

Allgemein läßt sich der Entwurfsprozeß zur optimalen Lösung von Designproblem (1) und Designproblem (2) wie folgt skizzieren:

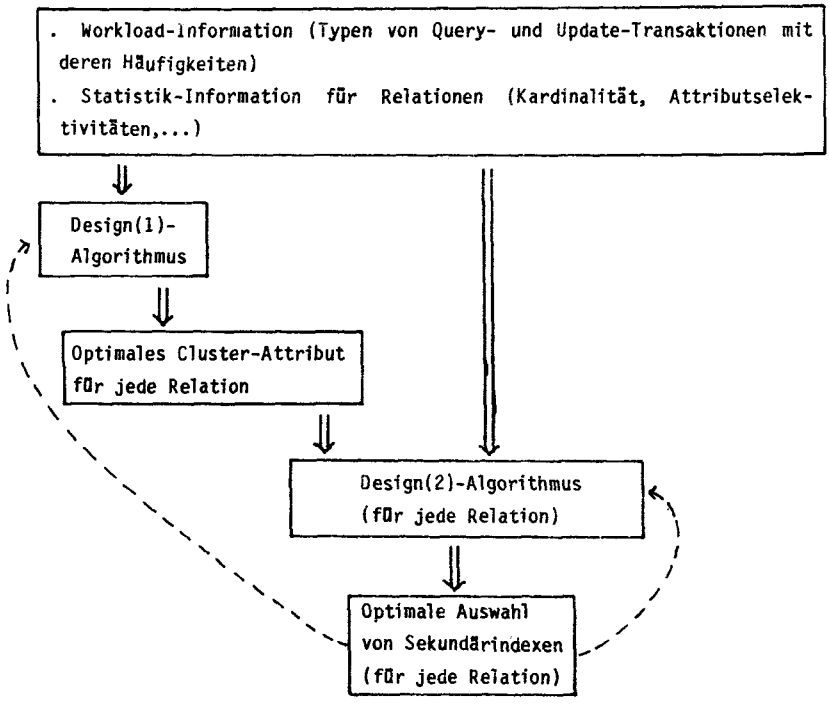


Abb.5.3: Schema zur optimalen Einrichtung von Zugriffspfaden.

Im allgemeinen läßt sich die Bestimmung einer optimalen Menge von Zugriffspfaden praktisch nicht durchführen, da dieses Problem NP-vollständig ist. Aber auch bei der Erstellung von heuristischen Optimierungsalgorithmen ergeben sich beträchtliche Schwierigkeiten, welche hauptsächlich aus den komplexen Wechselwirkungen und Rückkoppelungen resultieren (----> in Abb.5.3), die zwischen Design(1)- und Design(2)-Algorithmus existieren (vgl. dazu z.B. [HAMM76]).
Wie jedoch die Diskussion über eine geeignete Ablage von Relationen auf den DB-Platten gezeigt hat, ist es für unsere DB-Architektur vorteilhaft,

Relationen so kompakt und physisch-sequentiell wie irgend möglich abzuspeichern, was wiederum impliziert, daß keine Sortierordnung erzwungen werden darf. Der Primärzugriffspfad ist eine vollständige Segmentfilterung, alle zusätzlich eingerichteten Indexe haben den Charakter von Sekundärindexen.

Konsequenz:

Design(1)-Problem entfällt für unsere Architektur.

Diese wesentliche Vereinfachung hat wiederum zur Folge, daß eventuell mögliche Interferenzen zwischen Design(1)- und Design(2)-Problem entfallen. Somit vereinfacht sich auch das Design(2)-Problem, welches nun isoliert gelöst werden kann.

Die existierenden Algorithmen für Design(2)-Problem (vgl. z.B. [HAMM76]) beruhen darauf, daß man für jedes Attribut einen sogenannten Rank-Faktor bestimmt. Ein höherer Rank-Faktor deutet dabei auf eine höhere Kostenersparnis bei Lesezugriffen hin und spricht somit mehr für die Einrichtung eines Sekundärzugriffspfades.[95] Ein Schätzwert für die obere Grenze eines Rank-Faktors für ein gegebenes Attribut r einer Relation R wird wie folgt ermittelt:

Sei - bl(R) die Anzahl der Blöcke, in denen Tupel von R gespeichert sind,
- avg_sel(r) die "durchschnittliche Selektivität" einer Restriktion auf r, gemittelt über alle Queries der vorliegenden Workload, ($0 \leq avg\_sel(r) \leq 1$),
- occ(r) die durchschnittliche Anzahl des Auftretens von Restriktionen auf r in Queries der vorliegenden Workload.

In einem konventionellen DBMS ergeben sich damit folgende Kosten zur Berechnung einer Restriktion auf r:

(a) Es existiert kein Index auf r: bl(R) Blockzugriffe

(b) Es existiert ein Index auf r:

Wenn Indexzugriffskosten vernachlässigt werden und man annimmt, daß jeder Trefferblock nur einmal in den DB-Puffer gelesen werden muß, so erhält man als Kosten bl(R)·avg_sel(r) Blockzugriffe.

Die Kostenersparnis bei Existenz eines Indexes auf r beläuft sich dann

---

[95] Der Primärschlüssel einer Relation wird i.a. einen sehr hohen Rank-Faktor aufweisen (vgl. auch Kap.5.4.3.2).

höchstens auf:

save(r) := bl(R)·(1-avg_sel(r))

Die obere Grenze für den Rankfaktor liegt dann bei:

upperbound(rank_factor(r)) := occ(r)·save(r)

Offensichtlich ergibt sich in unserer DB-Architektur die Notwendigkeit, save(r) neu zu berechnen. Insbesondere kann save(r) nicht in Anzahl von Blockzugriffen gezählt werden, sondern muß in Ausführungszeiten für DB-I/O-Operationen gemessen werden. Die Absicht des restlichen Kapitels ist es nun, nicht neue Optimierungsalgorithmen zur Einrichtung von Sekundärindexen zu erstellen, sondern konkrete Verfahren zur Ermittlung von Grenzselektivitäten für Indexverwendung zu entwickeln. Mittels dieser Schrankenwerte kann man dann die Kostenersparnis save(r) in unserer Architektur geeignet abschätzen.

5.4.2. Schrankenwerte.

Die in Kap.5.3 definierten Grenzfunktionen $ta_n(m)$ und $t2a_{n,m_1}(m)$ können zur Optimierung allgemeiner Q_{restr}- bzw. Q_{conj}-Queries herangezogen werden. Das Ziel der folgenden Kapitel wird die Bestimmung von Grenzselektivitäten sein, oberhalb derer sich der Einsatz einzelner Indexe nicht mehr lohnt. Zu diesem Zwecke nehmen wir eine weitere Spezialisierung vor.

Definition 5.1:

Sei EP[r] eine Restriktionsformel auf r von R.

EP[r] heißt elementare Restriktion, wenn gilt:

$$EP[r] = 'vr^{(1)}\ \theta_1\ r\ \theta_2\ vr^{(2)}'$$
$$\text{mit } vr^{(1)},\ vr^{(2)} \in dom(R,r),\ \theta_1,\theta_2 \in \{<, \leq\}$$

Wir betrachten ab jetzt ausschließlich die Auswertung elementarer Restriktionsprädikate. Der Einschränkung auf elementare Restriktionen liegt folgender in der Praxis oft zutreffender Sachverhalt zugrunde, den wir für

die weiteren Rechnungen unterstellen wollen:

Annahme (A):
Sei EP[r] elementare Restriktion mit Kostenparametern a und m, I_r Index auf r, dann gilt:

$a = a_{I_r}(m)$, wobei $a_{I_r}(m)$ eine monoton steigende Funktion in m ist.

Als Spezialfall von Q_{restr} untersuchen wir die Query

Q_{Erestr} ≡ SELECT * FROM R
WHERE EP[r] AND F'

Der Effekt des funktionalen Zusammenhangs zwischen a und m läßt sich graphisch gut (schematisch) verdeutlichen:

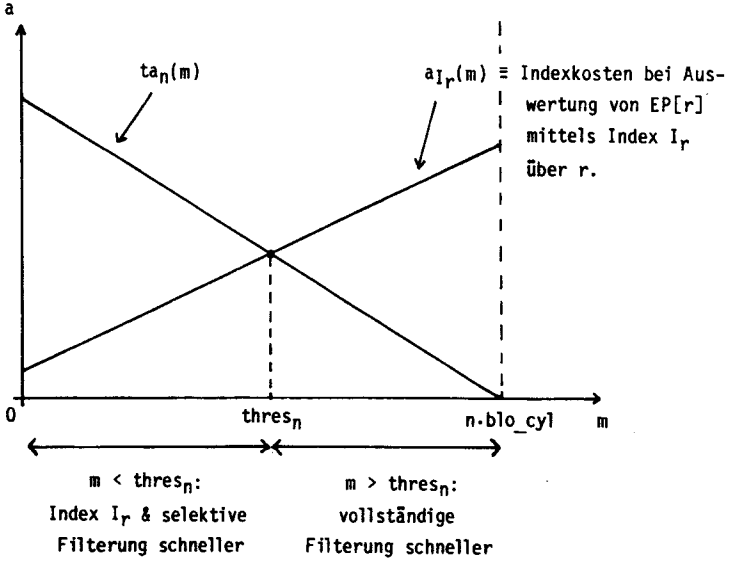


Abb.5.4: Schrankenwert für m bei elementarer Restriktion EP[r].

Im Unterschied zu Abb.5.1 in Kap.5.3.1 ist bei $F^+ \equiv EP[r]$ die Auswahl des schnellsten Zugriffpfadtyps alleine anhand von m im Vergleich mit dem Schrankenwert $thres_n$ möglich.

Definition 5.2:
Der Schrankenwert $thres_n$ (bei Einsatz eines Indexes I_r) ist definiert wie folgt:

$$thres_n := \begin{cases} 0 & \text{, falls } a_{I_r}(0) > ta_n(0) \\ m^*\text{: } ta_n(m^*) = a_{I_r}(m^*) & \text{sonst} \end{cases}$$

Anmerkungen:
- $thres_n$ ist effizient berechenbar mittels Bisektionsalgorithmus.
- m^* bezeichnet den eindeutig bestimmten Schnittpunkt.

Korollar 5.2: (Optimierungskriterium 1' für $Q_{Erestr}(\subseteq Q_{restr})$)
Sei $\overline{m}$ die Anzahl der Blocktreffer bei Indexauswertung von EP[r].
(a) $\overline{m} < thres_n$ ===> I_r & selektive Filterung ist schneller als vollständige Filterung.
(b) $\overline{m} > thres_n$ ===> Vollständige Filterung ist schneller als I_r & selektive Filterung.

Beweis: Definition 5.2 plus Satz 5.2.

Analog betrachten wir einen Spezialfall von Q_{conj}-Queries, nämlich daß F_1^+ und F_2^+ elementare Restriktionen sind; i.Z. $F_1^+ = EP_1[r_1]$, $F_2^+ = EP_2[r_2]$.

SQL-Form:

```
Qconj = SELECT *
        FROM   R
        WHERE  EP1[r1] AND EP2[r2] AND F-
```

Um die Indexkosten der verschiedenen Indexe I_{r_1} auf r_1 und I_{r_2} auf r_2

miteinander vergleichen zu können, treffen wir eine weitere Annahme, welche in der Praxis oft zutrifft:

Annahme (B):

Für die Indexe I_{r_1} und I_{r_2} gelte $a_{I_{r_1}}(m) = a_{I_{r_2}}(m)$.

Bezeichnung: $a(m) := a_{I_{r_1}}(m)$.

Folgerungen aus den Annahmen (A) und (B):

- $m_1 \leq m_2$ ===> $a_1 \leq a_2$ ===> sel_at$(n,a_1,m_1) \leq$ sel_at(n,a_2,m_2).
  Das bedeutet aber, daß cost_AP2$_1 \leq$ cost_AP2$_2$, d.h. daß die Indexauswertung von $EP_1[r_1]$ mit selektiver Filterung gemäß $EP_2[r_2] \wedge F^-$ schneller ist als die Indexauswertung von $EP_2[r_2]$ mit selektiver Filterung gemäß $EP_1[r_1] \wedge F^-$.
- Falls $m_1 \leq m_2$ und $a_1 = a(m_1) > t2a_{n,m_1}(m_1)$
  ====> $a_2 = a(m_2) > t2a_{n,m_1}(m_1)$.

Definition 5.3:

Der Schrankenwert $thres2_{n,m_1}$ (für den Einsatz zweier Indexe) ist definiert wie folgt:

$$thres2_{n,m_1} := \begin{cases} m_1 & \text{, falls } t2a_{n,m_1}(m_1) < a(m_1) \\ m^*\text{: } t2a_{n,m_1}(m^*) = a(m^*) & \text{sonst} \end{cases}$$

Anmerkung:

Da $t2a_{n,m_1}(m)$ streng monoton fällt, $a(m)$ aufgrund von Annahme (A) hingegen monoton steigt, ergibt sich ein eindeutiger Schnittpunkt m^*.

Korollar 5.3: (Optimierungskriterium 2' für Q_{Econj} ($\subseteq Q_{conj}$))

Seien m_i die entsprechenden Kostenparameter bei Auswertung von $EP_i[r_i]$ über I_{r_i}, i=1,2; $m_1 \leq m_2 \leq n \cdot$blo_cyl. Dann gilt:

(a) $m_2 <$ $thres2_{n,m_1}$ ===> $AP2_1 \wedge 2$ ist schneller als $AP2_1$

(b) $m_2 >$ $thres2_{n,m_1}$ ===> $AP2_1$ ist schneller als $AP2_1 \wedge 2$

Beweis: Folgt direkt aus Definition 5.3 und Satz 5.3.

Graphische Illustration von Kor.5.3 (wiederum nur schematisch; nur die Monotonie von $ta_n(m)$ und $t2a_{n,m_1}(m)$ soll dargestellt werden):

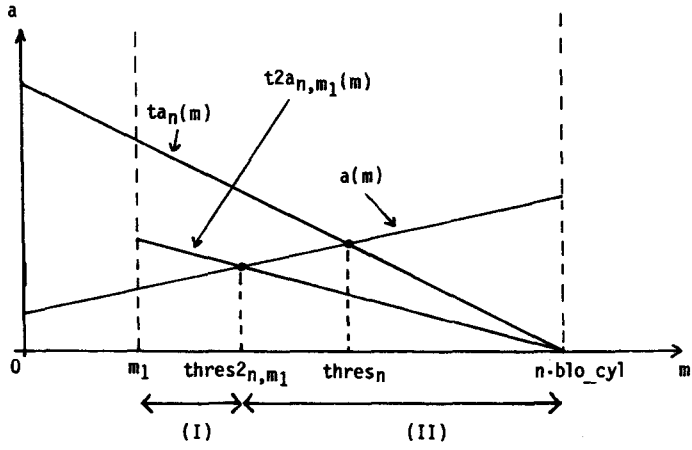


Abb.5.5: Schrankenwert für m_2 bei Q_{Econj}.

(I): $m_2 < thres2_{n,m_1}$ ===> 2 Indexe I_{r_1} und I_{r_2} & selektive Filterung gemäß F^- am schnellsten.

(II): $m_2 > thres2_{n,m_1}$ ===> Index I_{r_1} & selektive Filterung gemäß $EP_2[r_2]$ am schnellsten.

5.4.3. Auswertungen des Schrankenmodells.

5.4.3.1 Parameterschätzungen.

Dieses Kapitel soll als Vorbereitung für eine numerische Auswertung des Schrankenmodells zur Ermittlung von Grenzselektivitäten in Kap.5.4.3.2

dienen. Zunächst werden aus der Literatur bekannte Methoden zur Schätzung von Tupeltrefferraten geschildert; im nächsten Schritt wird dann eine Transformation von der Ebene der Tupeltreffer als die Ebene der im Schrankenmodell benötigten Blocktreffer vollzogen. Dabei geben wir eine Rechtfertigung für die im vorherigen Kapitel eingeführten Annahmen (A) und (B) für die Indexkosten zur Auswertung elementarer Restriktionen.
Betrachten wir einen Index I_r auf dem Attribut r der Relation R. I_r soll ein dichter, unabhängiger Index in Form eines B-Baumes sein, welcher lid- bzw. (lid,pid)-Einträge nur in den Blättern enthält; die Blätter seien verkettet. Zur Schätzung von Tupeltrefferraten bei Indexauswertung benötigt man einige Index-Statistikdaten, welche normalerweise in den Systemkatalogen dem Optimierer zur Verfügung stehen.

Index-Statistikdaten:

- MAX(r) := max(π_r(R)); analog MIN(r)
- ndkey(r) := card(π_r(R))
- height(r) ≡ Höhe von I_r
- nleaf(r) ≡ Anzahl der Blattknoten von I_r

Ferner benötigen wir

- card(R) ≡ Kardinalität der Relation R

Der Selektivitätsfaktor sf(F) einer Restriktion F auf R ist definiert als die relative Tupeltrefferrate bei Auswertungen von σ_F(R), d.h.:

$$sf(F) := \frac{card(\sigma_F(R))}{card(R)} \quad ; 0 \leq sf(F) \leq 1.$$

Die Tupeltrefferrate t_hits(F) für F ist gegeben durch

$$t\_hits(F) := sf(F) \cdot card(R) \qquad [\#Tupel]$$

Ein Teil der aufgezählten Index-Statistikdaten wird nun dazu verwendet, um Schätzwerte für sf(EP[r]) für elementare Restriktionen EP[r] zu ermitteln. Da eine genaue Schätzung von Selektivitätsfaktoren nur möglich ist, wenn genaue Aussagen über die statistische Verteilung der betreffenden Attributwerte vorhanden sind (was aber meist nicht der Fall ist), wurden in

der Literatur die Schätzwerte meist unter der Annahme von Gleichverteilung ermittelt (vgl. etwa [SELI79], [MAKI81]). Für diesen Fall definieren wir:

(1) $sf('vr^{(1)} \leq r \leq vr^{(2)}') := \max\{\frac{vr^{(2)} - vr^{(1)} + 1}{MAX(r) - MIN(r) + 1}, 1\}$

(2) $sf('vr = r') := \frac{1}{ndkey(r)}$

Für unser Schrankenmodell zur Optimierung elementarer Restriktionen benötigen wir Schätzwerte für die Parameter a und m, d.h. für Indexknotenzugriffe und Blocktrefferraten. (Relation R sei wieder in n Zylindern mit n.blo_cyl Blöcken abgespeichert.)
Nach der Ermittlung eines Schätzwertes für ein EP[r] ist somit als nächster Schritt die Abbildung der t_hits(EP[r]) Tupel auf die n.blo_cyl Blöcke der Relation R zu lösen, d.h. wir suchen die durchschnittliche Anzahl $\bar{m}$ von Blöcken von R, in denen mindestens eines der t_hits(EP[r]) Tupel gespeichert ist. Unter der Annahme der Gleichverteilung können wieder das bereits in Kap.5.1.3 bei der Berechnung von sel_at(n,m) verwendete Modell anzuwenden. Mittels der Cardenas-Formel erhalten wir:

$$\bar{m} = n.blo\_cyl \cdot (1 - (1 - \frac{1}{blo\_cyl})^{t\_hits(EP[r])})$$

Die Anzahl der aufzusuchenden Indexknoten $\bar{a}$ läßt sich folgendermaßen abschätzen (man beachte die spezielle Form eines EP[r] und die Tatsache, daß die Indexblätter verkettet sind):
- aufzusuchende Nichtblattknoten: height(r) - 1
- aufzusuchende Blattknoten : sf(EP[r]).nleaf(r)

====> $\bar{a} = sf(EP[r]).nleaf(r) + height(r) - 1$

Unsere gesuchten Kostenparameter m und a betreffen nur die Anzahl der von der LI in das DB-Cache zu transportierenden Blöcke, d.h.:

$$a = \delta_a \cdot \bar{a}$$

$$m = \delta_m \cdot \bar{m}$$

wobei $0 \leq \delta_a, \delta_m \leq 1$; δ_a und δ_m sind Lokalitätsfaktoren für das DB-Cache. Aufgrund unserer DB-Cache-Organisation, daß ZERs im Private_C liegen (siehe

Kap.3.2.3), gilt nun $\delta_m = 1$.
Für die in Kapitel 5.4.3.2 durchzuführenden Experimente wollen wir erst einmal von Lokalitätseffekten für Indexknoten im DB-Cache abstrahieren, und setzen $\delta_a = 1$.
Somit verwenden wir folgende Schätzwerte für m und a:

$$(S1)\quad m = n.blo\_cyl \cdot \left(1 - \left(1 - \frac{1}{n.blo\_cyl}\right)^{t\_hits(EP[r])}\right)$$

$$(S2)\quad a = sf(EP[r]).nleaf(r) + height(r) - 1$$

Für die Anwendung des Schrankenmodells benötigen wir jetzt nur noch den Nachweis des in der Annahme (A) geforderten funktionalen Zusammenhangs zwischen a und m, nämlich $a = a_{I_r}(m)$, wobei $a_{I_r}(m)$ monoton wächst in m.
Dazu lösen wir die Gleichung (S1) nach t_hits(EP[r]) auf:

$$(S1) \Leftrightarrow -\frac{m}{n.blo\_cyl} + 1 = \left(1 - \frac{1}{n.blo\_cyl}\right)^{t\_hits(EP[r])}$$

$$\Leftrightarrow -t\_hits(EP[r]) = \log_{1 - \frac{1}{n.blo\_cyl}}\left(1 - \frac{m}{n.blo\_cyl}\right)$$

Wegen $t\_hits(EP[r]) = sf(EP[r]).card(R)$ folgt damit aus (S2):

$$a = \frac{nleaf(r)}{card(R)} \cdot \log_{1 - \frac{1}{n.blo\_cyl}}\left(1 - \frac{m}{n.blo\_cyl}\right) + height(r) - 1$$

Beobachtung:
$\frac{card(R)}{nleaf(r)}$ gibt die durchschnittliche Anzahl von Tupelverweisen ((tid,pid) - Einträge) *in einem Indexblatt an.*

$$lfout(r) := \frac{card(R)}{nleaf(r)} \qquad (\underline{l}eaf\text{-}\underline{fanout})$$

Eigenschaften von lfout(r):
Aufgrund unserer Gleichverteilungsannahme für die Attributwerte in π_r ist lfout(r) eine Größe, welche nur von den festen Werten Blockgröße (bsize), Länge eines Attributwertes für r und Länge der tid- bzw. (tid,pid)-Einträge abhängt; lfout(r) ist also ein konstanter Systemparameter.

Damit erhält man:

$$a_{I_r} := \frac{1}{lfout(r)} \cdot \log_{1-\frac{1}{n \cdot blo\_cyl}} \left(1 - \frac{m}{n \cdot blo\_cyl}\right) + height(r) - 1$$

Aufgrund der Monotonie von log trifft Annahme (A) unter den betrachteten Voraussetzungen zu (Basis des log ist < 1).
Die Annahme (B), daß $a_{I_{r_1}}(m) = a_{I_{r_2}}(m)$ gilt, trifft genau dann zu, wenn -unter ansonsten gleichen Bedingungen- die Längen der Attributwerte für r_1 und r_2 in etwa gleich sind. (In diesem Fall gilt dann $lfout(r_1) = lfout(r_2)$.)

5.4.3.2 Numerische Resultate.

Die anschließenden numerischen Auswertungen sollen Aussagen bezüglich Tupeltrefferraten machen, während das in 5.4.2 erstellte Schrankenmodell mit Blocktrefferraten arbeitet. Um Rückschlüsse von Grenztrefferraten für Blöcke auf die entsprechenden Tupeltrefferraten machen zu können, führen wir den Begriff des durchschnittlichen Blockungsfaktors bf_R ein.
bf_R gibt an, wieviele Tupel von R im Durchschnitt in einem Block des von R belegten Segments gespeichert sind. Für eine konkrete DB-Anwendung wird bf_R durch die Blockgröße bsize, die durchschnittliche Tupellänge sowie durch die Segment-Verwaltung bestimmt; typische Werte dürften $1 \leq bf_R \leq 50$ sein.
Festzuhalten ist, daß gilt:[96] $card(R) = n \cdot blo\_cyl \cdot bf_R$
Eine Motivation für die Einführung unserer neuen Architektur bildete u.a. folgende plausible Vermutung:
Durch die immer höher werdende Speicherdichte moderner Plattengeräte bei fast gleichbleibender Leistungsfähigkeit der mechanischen Komponenten wird die vollständige Filterung die Verwendung von Indexen immer mehr zurück-

[96] Die angegebene Formel gilt unter der bisher immer angenommenen Voraussetzung, daß R stets volle Zylinder belegt.

drängen. Um einen ersten Eindruck von den veränderten Eigenschaften unserer DB-Architektur zu gewinnen, wurden mit den beiden analytischen Schrankenmodellen zwei Simulationsreihen ausgeführt, welche quantitative Aufschlüsse über die diesbezüglichen Eigenschaften von repräsentativen konventionellen Platten liefern.

Simulationsreihe 1:

Hierbei wurden zwei IBM-Plattenlaufwerke als DB-Platte einem Vergleich unterzogen, nämlich die relativ alte 3330 sowie die neueste 3380. Beide Plattengeräte besitzen in etwa dieselbe Mechanik, sie unterscheiden sich jedoch enorm bezüglich der Speicherdichte. Zur Verringerung der Positionierzeiten verfügt die IBM 3380 über zwei Kämme mit Lese/Schreibköpfen; diese Fähgkeit bleibt jedoch in den vorgenommenen Auswertungen unberücksichtigt.

Kenngrößen der IBM 3330: (ca. 100 MByte Speicherkapazität)

- cap_tr = 13030 Bytes
- tr_cyl = 19
- cyl_dp = 404
- rot = 16.7 msec
- start_h = 10 msec
- next_cyl = 0.1 msec

Kenngrößen einer IBM 3380-ähnlichen Platte: (ca. 730 Mbyte Kapazität)

- cap_tr = 47500 Bytes
- tr_cyl = 19
- cyl_dp = 808
- rot = 16.7 msec
- start_h = 10 msec
- next_cyl = 0.05 msec

Ferner setzen wir für beide Plattentypen voraus:

- bsize = 2048 Bytes

Weiterhin legen wir fest

- bf = 10, (d.h. Tupellänge ist $\frac{bsize}{bf}$ = ca.200 Bytes)
- height(r) = 3

Ziel von Simulationsreihe 1 ist es, Schrankenwerte bzgl. der Tupeltrefferrate bei Einsatz eine Indexes für die Auswertung von

Q_{Erestr} = SELECT * FROM R WHERE EP[r] AND F'

zu ermitteln. Die Grundlage dafür liefert Korollar 5.2.

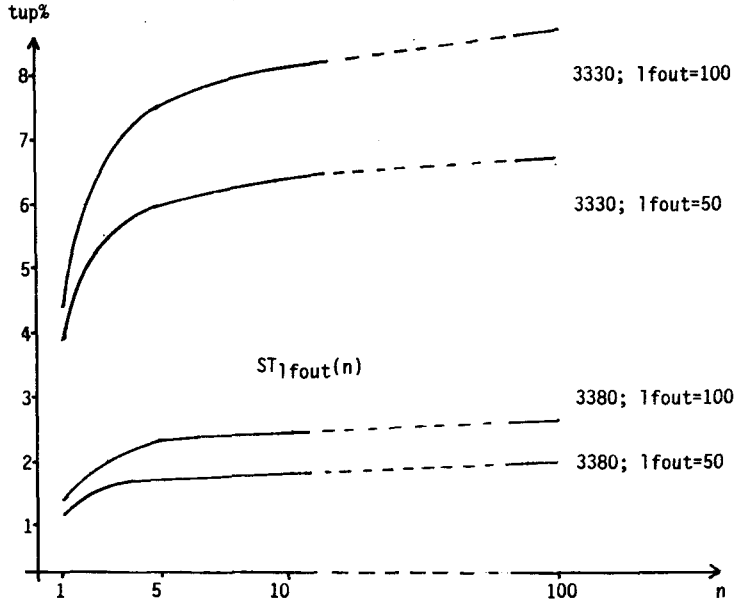


Abb.5.6: Schrankenwerte für Einsatz eines Indexes.

Nehmen wir an, die Treffertupel für Q_{Erestr} sind in $thres_n$ Blöcken gespeichert. Aufgrund von Schätzwert (S1) ergibt sich dann als Rückschluß auf die Tupeltrefferrate von EP[r]:

$$t\_hits(EP[r]) = \log_{1-\frac{1}{n \cdot blo\_cyl}} \left(1-\frac{thres_n}{n \cdot blo\_cyl}\right)$$

$$\Longrightarrow \quad \frac{t\_hits(EP[r])}{card(R)} = \frac{t\_hits(EP[r])}{n \cdot blo\_cyl \cdot bf}$$

gibt die relative Grenz-Tupeltrefferrate an, oberhalb derer sich der Einsatz des Indexes nicht mehr lohnt.

Somit gibt die nachfolgende Funktion die prozentuale Schranke bzgl. der Tupeltreffer an, oberhalb derer sich der Indexeinsatz nicht mehr lohnt.

$$ST_{1fout}(n) := \frac{100}{n.blo\_cyl \cdot bf} \cdot \log_{1-\frac{1}{n.blo\_cyl}} (1 - \frac{thres_n}{n.blo\_cyl}) \qquad [tup\ \%]^{97)}$$

Aus Abbildung 5.6 ist unmittelbar deutlich, daß sich der lohnende Einsatz eines Indexes zur Auswertung einer elementaren Restriktion bei den neuen, schnelleren Platten drastisch verringert.

Simulationsreihe 2:

Die nächste Analyse soll Aufschluß darüber geben, für welche Tupeltrefferraten sich der Einsatz von zwei Indexen zur Auswertung von Q_{Econj} ≡ SELECT * FROM R WHERE $EP_1[r_1]$ AND $EP_2[r_2]$ AND F^- lohnt. Das relevante Optimierungskriterium dazu gibt Korollar 5.3 an. Wir definieren nun:

$$ST2_{n,1fout}(\overline{m}_1) := \frac{100}{n.blo\_cyl \cdot bf} \cdot \log_{1-\frac{1}{n.blo\_cyl}} (1 - \frac{thres2_{n,m_1}}{n.blo\_cyl})$$

wobei

$$\overline{m}_1 := \frac{m_1}{n.blo\_cyl \cdot bf} \cdot 100, \ (\equiv \text{prozent. Blocktreffer von } EP_1[r_1])$$

$1 \leq m_1 < thres_n$ gilt.[98)]

Analog zur Herleitung von $ST_{1fout}(n)$ in Simulationsreihe 1 kann man $ST2_{n,1fout}(\overline{m}_1)$ wie folgt interpretieren:

---

97) Der Parameter 1fout (= 1fout(r)) ist in dieser Definition nicht explizit auf der rechten Seite ersichtlich, 1fout steckt jedoch implizit in $thres_n$, da nach Definition:

$$thres_n = \begin{cases} 0, & \text{falls } a_{I_r}(0) > ta_n(0) \\ m^*: ta_n(m^*) = a_{I_r}(m^*) & \text{sonst} \end{cases}$$

und

$$a_{I_r}(m) = \frac{1}{1fout} \cdot \log_{1-\frac{1}{n.blo\_cyl}} (1 - \frac{m}{n.blo\_cyl}) + height(r) - 1$$

98) 1fout erscheint implizit in $thres2_{n,m_1}$.

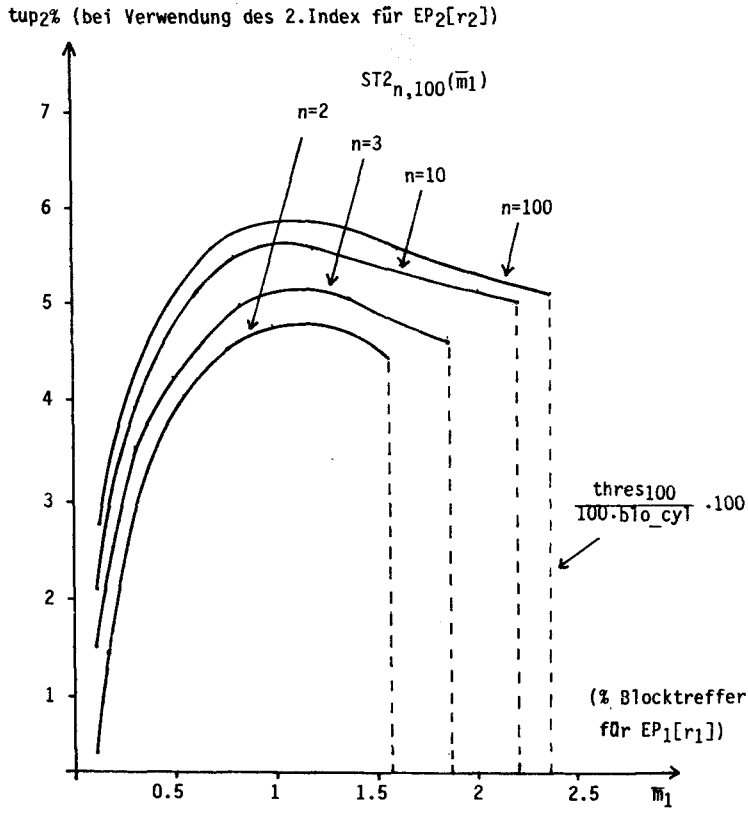


Abb.5.7: Schrankenwerte für Einsatz von zwei Indexen.

$ST2_{n,1fout}(\bar{m}_1)$ gibt die prozentuale Grenze der Tupeltrefferrate ($tup_2\%$) von $EP_2[r_2]$ an, oberhalb derer sich der Einsatz eines zweiten Indexes für $EP_2[r_2]$ nicht mehr lohnt.

Das Diagramm in Abb.5.7 wurde für folgende Kostenparameter ermittelt:

- Kenngrößen ein IBM 3380-ähnlichen Platte wie in Reihe 1.
- bsize, bf wie in Reihe 1.
- height(r_1) = height(r_2) = 3.
- lfout := lfout(r_1) = lfout(r_2) = 100.

n bezeichnet wie üblich die Anzahl der Zylinder, auf denen eine Relation gespeichert ist. Die Kurven brechen ab, sobald die vollständige Filterung besser ist. Im Bereich oberhalb einer Kurve lohnt sich der Einsatz des 2.Indexes nicht mehr, unterhalb ist die Verwendung beider Indexe am besten.

Wertung dieser exemplarischen Simulationsreihen:
Die Einrichtung von Sekundärindexen für unsere DB-Architektur wird nur dann noch eine Leistungsverbesserung bringen, wenn folgende drei Punkte gleichzeitig zutreffen:

(a) Der betrachtete Index I_r auf einem Attribut r muß ein sehr hohes Auflösungsvermögen haben.
(b) Die Retrieval-Häufigkeit für r muß sehr hoch sein.
(c) Die Update-Häufigkeit für muß relativ gering sein.

Aus Abb.5.6 ist die Verschärfung des Zugriffsengpasses für random Plattenzugriffe bei zunehmenden Plattenspeicherdichten[99] ablesbar. Im Vergleich zum Einsatz von Indexen in Standard-DB-Systemen ergibt sich für uns eine wesentliche Reduktion für eine profitable Indexverwendung.
Jedoch werden Indexe auf Schlüssel-Attributen weitehin unerläßlich für eine gute Retrieval-Leistungsfähigkeit sein, da deren prozentuale Tupeltrefferrate von 100/card(R) weit unter den errechneten Schrankenwerten liegt.
Die beiden durchgeführten Simulationsreihen haben bereits einen ersten Eindruck von den neuen Leistungsmerkmalen unserer DB-Architektur mit ihren intelligenten Subsystemen vermittelt. Eine detaillierte Auswertung (z.B. Variation anderer interessanter Parameter wie bsize oder bf_R) konnte aus Zeitgründen nicht mehr durchgeführt werden.

[99] Prognosen sagen eine Verdoppelung alle 2 bis 3 Jahre voraus.

6. Zusammenfassung und Ausblick.

In dieser Arbeit wurde auf der Basis moderner Rechner-Architektur der Entwurf eines funktional vollständigen Datenbanksystems entwickelt. Bei den betrachteten Hardware-Komponenten wurde dabei großer Wert darauf gelegt, daß diese momentan bereits oder in sehr naher Zukunft verfügbar und wirtschaftlich sind. Darauf aufbauend wurden die internen Schnittstellen des DB-Systems so konstruiert, daß sie sorgfältig auf das Leistungsvermögen dieser Hardware-Komponenten abgestimmt sind.
Als wesentlichste Voraussetzung für ein Hochleistungs-Datenbanksystem wurde dabei das Prinzip der Mengenverarbeitung herausgestellt. Die effiziente Realisierung einer Mengenverarbeitung wird durch den geschickten Einsatz von intelligenten Subsystemen für die autonome Verwaltung der physischen Datenbank ermöglicht. Durch eine frühzeitige Datenfilterung dieser Subsysteme wird das Volumen des Datentransports zwischen Peripheriespeicher und Arbeitsspeicher stark reduziert. Dieser Effekt wird noch unterstützt durch eine Reihe von neuen, sehr effizienten Algorithmen zur Queryauswertung, welche alle auf dem Prinzip der dynamischen Filter basieren. Damit ist es für eine sehr große Klasse von Transaktionen möglich, daß sich die von parallelen Transaktionen benötigte Information, solange wie für eine effiziente Mengenverarbeitung benötigt, vollständig im großen DB-Cache befindet. Desweiteren wurde die harmonische Integration moderner Synchronisations- und Commit/Recovery-Verfahren in den Gesamtsystemaufbau verwirklicht.
Aufgrund dieses synergistischen Zusammenspiels von Hardware- und Softwarekomponenten weist der vorliegende Entwurf einen vielversprechenden Weg in Richtung Hochleistungs-Datenbanksysteme. Als nächster Schritt müßte sich eine Studie anschließen, wie aufwendig eine (schrittweise) Konversion konventioneller DB-Systeme in die neue Umgebung ist. Ohne wesentliche Änderungen werden zuerst die dynamischen Filteralgorithmen integrierbar sein und vermutlich in vielen Fällen bereits eine merkliche Leistungssteigerung bewirken. Der Anschluß eines intelligenten Subsystems (anstelle der

bisherigen Platten) ist durch die strikte Entflechtung logischer Aspekte (im Host) und physischer Aspekte (im Subsystem) erleichtert. Durch die Herausnahme der Subsysteme aus dem Transaktionskonzept ist die Kopplung zwischen diesen und dem Hostrechner auf ein notwendiges Minimum reduziert, was eine einfachere Integration ermöglicht.

Neben diesen für die Praxis wichtigen Aspekten wurde in der Arbeit eine Reihe interessanter Spezialfragen aufgedeckt, deren theoretische Untersuchungen von Interesse sind. Offene Probleme für weiterführende Forschungsarbeiten betreffen in erster Linie die dynamischen Filter, wie z.B.:

- Allgemeine Konstruktionsverfahren für Filtertypen, sodaß ein breites Spektrum von Filterungswirkungsgraden abgedeckt ist, sowie die Untersuchung der Frage nach permanenten, dynamisch leicht fortschreibbaren Filtern.
- Eine Analyse von Filtertypen bzgl. Selektivität, Aufwand für Berechnung und Anwendung, sowie die Ermittlung von Im-Flug-Schranken für konkrete Mikrorechner.
- Anwendung der dyn. Filteridee auf ein volles relationales DBS.

Weiterhin bietet der vorliegende Entwurf eine gute Basis, um Aspekte eines lokal verteilten Datenbanksystems zu studieren (z.B. horizontale Partitionierung von Relationen auf die intelligenten Subsysteme, Tradeoff Parallelität und DB-Cache-Platzbedarf).

Um die vermuteten großen Leistungssteigerungen quantitativ belegen zu können, ist ein Effizienzvergleich mit konventionellen DB-Systemen mittels Simulationsmodellen wünschenswert. Außerdem bietet der Systementwurf ein gutes Objekt, um die besonders bei DB-Systemen noch weitgehend ungelösten Probleme einer geeigneten Prozeßstrukturierung studieren zu können. Schließlich ist mit der Entwicklung eines analytischen Basismodells die Grundlage geschaffen worden, um genauere Einblicke in die veränderten Eigenschaften der neuen Architektur, insbesondere im Hinblick auf die Frage des Nutzens von Indexen, zu gewinnen.

7. Anhang.

Satz 5.1(a): sel_at(n,m) ist streng monoton wachsend in m.

Bevor der Beweis von Satz 5.1(a) erbracht wird, sei die Def. von sel_at(n,m) wiederholt:

$$\text{sel\_at}(n,m) = \begin{cases} \text{avg\_seek} + (k-1)\cdot\text{seekcyl}(\text{avg\_dist}(k)) \\ \qquad + k\cdot\text{read\_tr}(1, \frac{m}{k\cdot 1}), & \text{falls } 0 < m \leq n\cdot\text{blo\_cyl} \\ 0, & \text{falls } m = 0 \end{cases}$$

wobei

$$\text{avg\_dist}(k) = \begin{cases} \frac{n+1}{k+1}, & \text{falls } 1 < k \leq n \\ 0, & \text{für } k \leq 1 \end{cases}$$

$$k = n\cdot(1 - (1 - \frac{1}{n})^m), \quad \text{für } 1 \leq m \leq n\cdot\text{blo\_cyl}$$

$$1 = \text{tr\_cyl}\cdot(1 - (1 - \frac{1}{\text{tr\_cyl}})^{\frac{m}{k}}), \quad \text{für } 1 \leq \frac{m}{k} \leq \text{blo\_cyl}$$

Beweis von Satz 5.1(a):
Wir wollen sel_at(n,m2) - sel_at(n,m1) für m1 < m2 abschätzen. Für m1 = 0 ist Satz 5.1(a) offensichtlich richtig. Seien k1, l1 und k2, l2 die zu m1 bzw. m2 gehörigen Werte für k, l. Für das folgende sei 0 < m1 vorausgesetzt.

(1) Aufgrund der Def. von k gilt:
m1 < m2 $\longrightarrow$ k1 < k2

(2) Wir zeigen: m1 < m2 $\longrightarrow$ l1 < l2
Beweis:
Aufgrund der Def. von l gilt: l1 < l2 $\longleftrightarrow$ $\frac{m1}{k1} < \frac{m2}{k2}$

$\longleftrightarrow m1\cdot(1-(1-\frac{1}{n})^{m2}) < m2\cdot(1-(1-\frac{1}{n})^{m1})$ [M]

$\longleftrightarrow m2\cdot(1-\frac{1}{n})^{m1} - m1\cdot(1-\frac{1}{n})^{m2} < m2 - m1$ [@]

Sei $f(y) := m2\cdot y^{m1} - m1\cdot y^{m2}$ für $0 \le y \le 1$.
$f'(y) = m1\cdot m2\cdot(y^{m1-1} - y^{m2-1})$
$f''(y) = m1\cdot m2\cdot((m1-1)\cdot y^{m1-2} - (m2-1)\cdot y^{m2-2})$
$f'(y) = 0 \longleftrightarrow y = 1$ oder $y = 0$
Wegen $f''(1) = m1\cdot m2\cdot(m1-m2) < 0$ hat f ein Maximum bei $y = 1$. Somit ist f(y) streng monoton steigend für $0 \le y \le 1$.
Es gilt also: $f(y) < f(1) = m2 - m1$ für $0 \le y < 1$.
Da $0 \le 1 - \frac{1}{n} < 1$ gilt, ist mit $y = 1 - \frac{1}{n}$ die Ungleichung @ bewiesen.

(3) Wir zeigen: $m1 < m2 \longrightarrow \frac{m1}{k1\cdot l1} < \frac{m2}{k2\cdot l2}$
Beweis:
Sei $g(m) := 1 - (1-\frac{1}{n})^{m}$

$h(m) := (1-\frac{1}{tr\_cyl})^{\frac{m}{n\cdot g(m)}}$

Damit gilt $k = n\cdot g(m)$ sowie $l = tr\_cyl\cdot(1-h(m))$.

$\frac{m1}{k1\cdot l1} < \frac{m2}{k2\cdot l2} \longleftrightarrow m1\cdot k2\cdot l2 < m2\cdot k1\cdot l1$
$\longleftrightarrow m1\cdot n\cdot g(m2)\cdot tr\_cyl\cdot(1-h(m2)) < m2\cdot n\cdot g(m1)\cdot tr\_cyl\cdot(1-h(m1))$
$\longleftrightarrow m2\cdot h(m1)\cdot g(m1) - m1\cdot h(m2)\cdot g(m2) < m2\cdot g(m1) - m1\cdot g(m2)$ [@@]

Sei $t(z) := m2\cdot z^{\frac{m1}{n\cdot g(m1)}}\cdot g(m1) - m1\cdot z^{\frac{m2}{n\cdot g(m2)}}\cdot g(m2)$ für $0 \le y \le 1$

$t'(z) = \frac{m1\cdot m2}{n}\cdot(z^{\frac{m1}{n\cdot g(m1)}-1} - z^{\frac{m2}{n\cdot g(m2)}-1})$
$t'(z) = 0 \longleftrightarrow m1\cdot g(m2) = m2\cdot g(m1)$ oder $z = 1$.
Aufgrund der Ungleichung M aus (2) gilt aber wegen
$g(m) = 1 - (1-\frac{1}{n})^{n}$, daß $m1\cdot g(m2) < m2\cdot g(m1)$. Somit verbleibt nur ein Extremalwert bei $z = 1$.

$t''(z) = \frac{m1\cdot m2}{n}\cdot((\frac{m1}{n\cdot g(m1)} - 1)\cdot z^{\frac{m1}{n\cdot g(m1)}-2} - (\frac{m2}{n\cdot g(m2)} - 1)\cdot z^{\frac{m2}{n\cdot g(m2)}-2})$

$$t''(1) = \frac{m1 \cdot m2}{n} \cdot \left(\frac{m1}{n \cdot g(m1)} - \frac{m2}{n \cdot g(m2)}\right)$$

$$= \frac{m1 \cdot m2}{n} \cdot \frac{m1 \cdot g(m2) - m2 \cdot g(m1)}{n \cdot g(m1) \cdot g(m2)} < 0 \quad \text{aufgrund Ungleichung M}$$

Also hat t(z) ein Maximum bei z = 1; es gilt demnach:
$t(z) < t(1) = m2 \cdot g(m1) - m1 \cdot g(m2)$ für $0 \leq z < 1$.
Da $0 \leq 1 - \frac{1}{tr\_cyl} < 1$, ist mit $z = 1 - \frac{1}{tr\_cyl}$ die Ungleichung @@ bewiesen.

Nun stehen alle benötigten Hilfsmittel zur Verfügung.

$$sel\_at(n,m2) - sel\_at(n,m1) =$$
$$[(k2 - 1) \cdot seek\_cyl(avg\_dist(k2)) - (k1 - 1) \cdot seek\_cyl(avg\_dist(k1))] +$$
$$[k2 \cdot read\_tr(12, \frac{m2}{k2 \cdot 12}) - k1 \cdot read\_tr(11, \frac{m1}{k1 \cdot 11})]$$

Da read_tr(j2,i2) > read_tr(j1,i1) für j2 ≥ j1 und i2 > i1 (vgl. Def. von read_tr in Kap.5.3.1), ist der zweite Term aufgrund der soeben bewiesenen Aussagen positiv.
Letztendlich ergibt sich: (* seek_cyl(c) = start_h + c·next_cyl *)

$$(k2 - 1) \cdot seek\_cyl(avg\_dist(k2)) - (k1 - 1) \cdot seek\_cyl(avg\_dist(k1)) =$$
$$(k2 - 1) \cdot (start\_h + \frac{n+1}{k2+1} \cdot next\_cyl) - (k1 - 1) \cdot (start\_h + \frac{n+1}{k1+1} \cdot next\_cyl) =$$
$$(k2 - k1) \cdot start\_h + (n + 1) \cdot (\frac{k2-1}{k2+1} - \frac{k1-1}{k1+1}) \cdot next\_cyl =$$
$$(k2 - k1) \cdot start\_h + (n + 1) \cdot \frac{2 \cdot (k2 - k1)}{(k2+1) \cdot (k1+1)} \cdot next\_cyl =$$
$$(k2 - k1) \cdot (start\_h + \frac{2 \cdot (n+1)}{(k2+1) \cdot (k1+1)} \cdot next\_cyl) \geq 0$$

Also: sel_at(n,m2) - sel_at(n,m1) > 0 für m1 < m2 .

Literaturverzeichnis.

[ASTR75] Astrahan, M.M; Chamberlin, D.D.:
'Implementation of a Structured English Query Language',
CACM, Oct.1975, Vol.18, No.10, S. 580-588.

[ASTR76] Astrahan, M.M.; et al.:
'SystemR: Relational Approach to Database Management',
ACM TODS, Vol.1, No.2, Juni 1976, S. 97-137.

[ASTR80] Astrahan, M.M.; Schkolnik, M.:
'Performance of the SystemR Access Path Selection Mechanism',
Proc. IFIP 1980, S. 487-491.

[BAHR80] Bayer, R.; Heller,H.; Reiser, Angelika:
'Parallelism and Recovery in Database Systems',
ACM TODS, Vol.5, No.2, Juni 1980.

[BANC80] Bancilhon, F.; Scholl, M.:
'Design of a Backend Processor for a Database Machine',
Proc. ACM SIGMOD 1980, Int. Conf. on Management of Data, S. 93-93g.

[BANE78] Banerjee, J.; Hsiao, D.K.:
'Performance Study of a Database Machine in Supporting Relational Databases',
VLDB 1978, S. 319-329.

[BAYE77] Bayer, R.; Schkolnik, M.:
'Concurrency of Operations on B-Trees',
Acta Informatica 9, 1977, S. 1-21.

[BITT82] Bittmann, P.:
'The Architecture of the Personal Computerized Workstation LEO',
in: Struktur und Betrieb von Rechensystemen, NTG/GI Fachtagung

Ulm, 1982.

[BLAS76] Blasgen, M.W.; Eswaran, K.P.:
'Storage and Access in Relational Data Bases',
IBM Syst. Journ., Vol.16, No.4, 1977.

[BLAS79a] Blasgen, M.W.; et al.:
'SystemR: An Architectural Update',
IBM Res. Lab. San Jose, RJ2581(33481), 1979.

[BLAS79b] Blasgen, M.W.; et. al.:
'The Convoy Phenomenon',
Operating Systems Review, 13, 2, April 1979, S. 20-25.

[BORA81] Boral, H.; DeWitt, D.J.; Wilkinson, W.K.:
'Performance Evaluation of Associative Disk Designs',
Comp. Sc. Dep. Univ. Wisconsin-Madison.

[BUZE75] Buzen, J.P.:
'I/O Subsystem Architecture',
Proc. IEEE, Vol.63, No.6, Juni 1975,S. 871-879.

[CARD75] Cardenas, A.F.:
'Analysis and Performance of Inverted Data Base Structures',
CACM, Mai 1975, Vol.18,No.5, S. 253-263.

[CHAM76] Chamberlin, D.D.; et al.:
'SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control',
IBM J. Res.&Dev., Vol.20, No.6, Nov.1976, S. 560-575.

[CHEN78] Chen, T.C.; et al.:
'Simplified Odd-Even Sort Using Multiple Shift-Register Loops',
Int. J. of Comp. and Inf. Sc., Vol.7, No.3, 1978, S.295-314.

[DATE82] Date, C.J.:
'An Introduction to Database Systems',
The Systems Programming Series, Vol.2, Addison-Wesley Publ. Comp., 1982, S. 341-349.

[DEWI79] DeWitt, D.J.:
'DIRECT - A Multiprocessor Organization for Supporting Relational Data Base Management Systems',
IEEE Trans. on Comp., Juni 1979, S.395-406.

[DEWI81] DeWitt, D.J.; Hawthorn, P.B.:
'A Performance Evaluation of Database Machine Architectures',
Proc. 7th Conf. on VLDB, Sept. 1981, Cannes, S. 199-213.

[DITT77] Dittmann, E.L.:
'Datenunabhängigkeit beim Entwurf von Datenbanksystemen',
S.Toeche-Mittler Verlag, 220 S.

[ELHA82] Elhardt, K.:
'Das Datenbank-Cache: Entwurfsprinzipien, Algorithmen, Eigenschaften'
Dissertation, TU München 1982.

[FINK82] Finkelstein, S.:
'Common Expression Analysis in Database Applications',
ACM SIGMOD 1982, S. 235-245.

[GRAY77] Gray, J.N.:
'Notes on Database Operating Systems'
Lecture Notes in Comp. Sc. 60, 1977, Springer-Verlag Berlin.

[GRAY81] Gray, P.M.D.:
'The "Group By" Operation in Relational Algebra',
Deen and Hammersley, Databases 1981.

[HAER78] Härder, T.:
'Implementierung von Datenbank-Systemen',
Hanser-Verlag 1978, 310 S.

[HALL76] Hall, P.A.V.:
'Optimization of Single Expressions in a Relational Data Base System',
IBM J. Res.&Dev., Mai 1976, S.244-257.

[HAMM76] Hammer, M.; Chan, A.:
'Index Selection in a Self-Adaptive Database Management System',
ACM SIGMOD, Juni 1976, S. 1-8.

[IMSFP] IMS/VS Version1, Fast Path Feature,
General Information Manual, GH 20-9069 2.

[INTE79] Intel Corporation: 'The 8086 Family User's Manual', Okt. 1979.

[KING80] King, W.F.III:
'Relational Database Systems: Where We Stand Today',

Proc. IFIP Congress 1980.

[KNUT68] Knuth, D.E.:
'The Art of Computer Programming',
Vol.1, 1968, Addison-Wesley.

[LAGA75] Lagally, K.:
'Das Projekt Betriebssytem BSM',
TU München, Institut für Informatik, Bericht Nr. 7509, Mai 1975.

[LANG77] Lang, T.; et al.:
'An Architectural Extension for a Large Database System Incorporating a Processor for Disk Search',
VLDB 1977, S. 204-210.

[LAUE78] Lauer, H.C.; Needham, R.M.:
'On the Duality of Operating System Structures',
Proc. 2nd Intern. Symp. on Operating Syst., IRIA, Okt. 1978,
Nachdruck in Operating Systems Review, Apr. 1979, S. 3-19.

[LORI77] Lorie, R.A.:
'Physical Integrity in a Large Segmented Database',
ACM TODS, Vol.2, No.1, März 1977, S. 91-104.

[MAKI81] Makinouchi, A.; et al.:
'The Optimization Strategy for Query Evaluation in RDB/V1',
VLDB 1981, S. 518-529.

[MAKI82] Makino, T.; et al.:
'An Evaluation of a Generalized Database Subsystem',
J. of Inf. Proc. Vol.5, No.1, 1982, S. 30-37.

[MARY80] Maryanski, F.J.:
'Backend Database Systems',
ACM Computing Surveys, Vol.12, No.1, März 1980, S. 3-25.

[OZKA77] Ozkarahan, E.A.; Schuster, S.A.; Sevcik, K.C.:
'Performance Evaluation of a Relational Associative Processor',
ACM TODS 2, No.2, 1977, S. 175-195.

[PARN72] Parnas, D.L.:
'On the Criteria to be used in Decomposing Systems into Modules',
CACM, Vol.15, No.12, Dez. 1972, S. 1053-1058.

[RODR76] Rodriguez-Rosell, J.:
'Empirical Data Reference Behavior in Data Base Systems',
IEEE Comp., Nov. 1976, S. 9-13.

[SCHK75] Schkolnik, M.:
'The Optimal Selection of Secondary Indices for Files',
Inf. Syst., Vol.1, März 1975, S. 141-146.

[SELI79] Selinger, P.G.; et al.:
'Access Path Selection in a Relational Database Management System',
IBM Res. Lab. San Jose, 1979, RJ2429(32240).

[SENK72] Senko, M.E.; et al.:
'Concepts of a Data Independent Accessing Model',
ACM SIGFIDET, 1972, Workshop on Data Description, Access and Control, S. 349-362.

[SIWI77] Siwiec, J.E.:
'A High-Performance DB/DC System',
IBM Syst. Journal, No.2, 1977, S. 169-195.

[SMIT75] Smith, J.M.; Chang, P.:
'Optimizing the Performance of a Relational Algebra Database Interface',
CACM, Vol.18, No.10, Okt. 1975, S. 568-579.

[SMIT81] Smith, A.J.: 'Input/Output Optimization and Disk Architectures: A Survey', Performance and Evaluation1, North-Holl., 1981, S. 104-117.

[STON76] Stonebraker, M.; Wong, E.; Kreps, P.:
'The Design and Implementation of INGRES',
ACM TODS, Vol.1, No.3, Sept. 1976, S. 189-222.

[STON80] Stonebraker, M.:
'Retrospection on a Database System',
ACM TODS, Vol.5, No.2, Juni 1980, S. 225-240.

[STON81] Stonebraker, M.:
'Operating System Support for Database Management',
CACM, Vol.24, No.7, Juli 1981, S. 412-418.

[TODD76] Todd, S.J.P.:

'The Peterlee Relational Test Vehicle - a system overview',
IBM Syst. Journal, No.4, 1976, S. 285-308.

[ULLM80] Ullman, J.D.:
'Principles of Database Systems',
Computer Software Engineering Series 1980, Pitman Publishing Limited.

[WEBE78] Weber, H.:
'A Software Engineering View of Data Base Systems',
VLDB 1978, S. 36-51.

[WHAN81] Whang, K.-Y.; Wiederhold, G.:
'Separability - An Approach to Physical Database Design',
VLDB 1981, S. 320-332.

[WONG76] Wong, E.; Youssefi, K.:
'Decomposition - A Strategy for Query Processing',
ACM TODS, Vol.1, No.3, Sept. 1976, S. 223-241.

[YAO77] Yao, S.B.:
'Approximating Block Accesses in Database Organizations',
CACM, April 1977, Vol.20, No.4, S. 260-261.

[YAO78] Yao, S.B.; DeJong, D.:
'Evaluation of Database Access Paths',
ACM SIGMOD, 1978, S. 66-77.

[YAO79] Yao, S.B.:
'Optimization of Query Evaluation Algorithms',
ACM TODS, Vol.4, No.2, Juni 1979, S. 133-155.

[YEH77] Yeh, R.T.; Baker, J.W.:
'Toward a Design Methodology for DBMS: A Software Engineering Approach',
VLBD 1977, S. 16-27.

[YOUS79] Youssefi, K.; Wong, E.:
'Query Processing in a Relational Database Management System',
VLDB 1979, S. 409-417.