Autonomes Index Tuning – DBMS-integrierte Verwaltung von Soft Indexen

Martin Lühring[1] Kai-Uwe Sattler[1] Eike Schallehn[2] Karsten Schmidt[3]
[1]Fakultät für Informatik und Automatisierung, TU Ilmenau
[2]Fakultät für Informatik, Universität Magdeburg
[3]FB Informatik, TU Kaiserslautern

Abstract: Das Self Management in DBMS, und das Self Tuning als wichtiger Teil davon, gewinnt auf Grund der wachsenden Komplexität von Systemen und Anwendungen und den daraus resultierenden steigenden Betriebskosten zunehmend an Beachtung. Der Stand der Technik bezüglich des Index Tunings sind Administrations-Tools, die aus einem gegebenen Workload eine empfohlene Indexkonfiguration ableiten. Diese muss jedoch bei Änderungen des Systems, dessen Umgebung oder der Zugriffsmuster wiederholt angepasst werden. Wir stellen in diesem Beitrag einen dynamischen Ansatz zum Index Tuning vor, der zur Laufzeit das Systemverhalten und die -nutzung überwacht, Entscheidungen bezüglich möglicher Verbesserungen der aktuellen Indexkonfiguration trifft und diese integriert mit der Anfrageverarbeitung umsetzt. Für den zuletzt genannten Aspekt führen wir mit *IndexBuildScan* und *SwitchPlan* neue Operatoren zur Erzeugung von Indexen während der Bearbeitung einer Anfrage sowie deren sofortiger Nutzung ein. Die Beschreibung der Implementierung der vorgestellten Konzepte als Erweiterung des PostgreSQL-Systems, sowie eine Evaluierung an Hand eines von TPC-H abgeleiteten Benchmarks beschließen den Beitrag.

1 Einführung

Datenbanksysteme sind heutzutage durch Anforderungen wie die Verwaltung großer Datenmengen, kurze Antwortzeiten und hohen Transaktionsdurchsatz charakterisiert. Neben Investitionen in teure Hardware ist das Datenbanktuning eine unverzichtbare Maßnahme. Datenbanktuning umfasst mehrere Teilaspekte – beginnend bei der Parametrisierung (z.B. der Puffergröße) über den physischen Entwurf (Partitionierung, die Auswahl von Indexen oder materialisierten Sichten) bis hin zum Applikationsentwurf (Umschreiben von Anfragen, Einfügen von Optimizer-Hints). Allerdings erfordern alle diese Aufgaben profunde Kenntnisse der Systeminternas, der Datencharakteristika, der Anwendungen und des Workloads. Daher nehmen inzwischen die Kosten für die Administration gerade in unternehmenskritischen Anwendungen einen großen Teil der Gesamtkosten ein.

Vor diesem Hintergrund rücken seit wenigen Jahren Self-Management-Features in den Blickpunkt des Interesses sowohl der Datenbanksystemhersteller als auch im Forschungsbereich. Trotz großer Fortschritte in den aktuellen Systemversionen ist jedoch leicht einzusehen, dass es bei der Komplexität heutiger Systeme und der Vielzahl von „Tuning-Reglern“ noch ein weiter Weg bis zu wirklichen autonomen Tuning ist.

Eine der wichtigsten Tuning-Aufgaben ist das Index Tuning, oft auch als *Index Selection Problem (ISP)* bezeichnet. Hierbei geht es um die Bestimmung der Indexe (die *Indexkonfiguration*), die eine gegebene Menge von Anfragen (Workload) am besten unterstützen, wobei zwischen dem Gewinn durch die Indexe und die zusätzlichen Kosten für den Speicherplatz bzw. die Verwaltung (Anlegen und Aktualisieren) abgewogen werden muss. Obwohl das ISP seit den 80er Jahren in der Literatur behandelt wird, wurde es bis heute im Wesentlichen als Entwurfsproblem betrachtet. Entsprechend bieten die führenden kommerziellen DBMS sogenannte Index Advisors oder Wizards an, die einen gegebenen Workload analysieren und darauf aufbauend eine Empfehlung für anzulegende Indexe geben. Die Entscheidung ob und wann diese Indexe erzeugt werden, bleibt weiterhin dem DBA vorbehalten.

Aufgrund der Wirkungsweise als (physisches) Entwurfswerkzeug arbeiten diese Werkzeuge hervorragend bei statischen Workloads. In eher explorativen Szenarien mit vielen Ad-hoc-Anfragen sind die passenden Indexe jedoch nur schwer vorherzusagen. Ein Beispiel hierfür ist der Business-Intelligence-Bereich etwa mit ROLAP-Tools, die Folgen von Anfragen (teilweise auf zwischenzeitlich erzeugten temporären Tabellen) als Folge von OLAP-Anfragen generieren [KSRM03].

Ein zweites Beispiel ist die Verwaltung von XML-Daten in relationalen Backends, bei der die Dokumentenstrukturen auf Tabellen abgebildet werden und über in SQL transformierte XPath bzw. XQuery Anfragen zugegriffen wird. Zusammenfassend lassen sich eine Reihe von Aspekten identifizieren, die zu Änderungen der Indexkonfiguration einer Datenbank führen können:

- Änderungen des Workloads wie in den oben genannten Beispielen,
- Änderungen des (relativen) Datenvolumens in Tabellen und der Datenverteilung,
- Änderungen am Datenbankschema,
- Änderungen der Infrastruktur, z.B. Austausch von Hardwarekomponenten,
- Interferenzen mit anderen Tuning-Maßnahmen, z.B. Partitionierung oder die Auswahl materialisierter Sichten.

In der Konsequenz muss das ISP in derartigen dynamischen Szenarien wiederholt bei allen diesen Änderungen gelöst werden. Damit stellt sich die Frage, warum das DBMS nicht völlig selbständig über das Anlegen hilfreicher Indexe bzw. das Löschen nicht mehr benötigter Indexe entscheiden kann? Oder ob Indexe für Anwender und DBA komplett unsichtbar werden können? Im Folgenden stellen wir einen Ansatz zum autonomen, anfragegetriebenen Index Tuning vor, der diese Fragen beantwortet. Wir bezeichnen dabei derartige, autonom vom DBMS verwaltete, Indexe als *Soft Indexe* zur Abgrenzung von Hard Indexen, die explizit vom DBA oder dem DBMS (zur Überprüfung von Integritätsbedingungen) angelegt wurden. Unser Ansatz, der vollständig in PostgreSQL implementiert ist, basiert auf einem kontinuierlichen Monitoring des Workloads und dem Lösen des ISP. Weiterhin betrachten wir indexerzeugende Anfragen, die potenziell nützliche Indexe als Teil der Anfrageausführung anlegen und so den Erstellungsaufwand reduzieren.

2 Problemstellung und verwandte Arbeiten

Wie im vorangegangenen Abschnitt erwähnt, kann Index Tuning als Form des Index-Selection-Problems (ISP) betrachtet werden. Formal kann dies wie folgt ausgedrückt werden [CFM95]: Gegeben sei eine Menge von Anfragen $Q_1, \ldots, Q_m$ sowie die Menge der Indexkandidaten $I_1, \ldots, I_n$ jeweils mit den Verwaltungskosten *mcost*(I_i) (für die Aktualisierungen) und der Größe *size*(I_i). Der Gewinn *profit* eines Indexes I_i für eine Anfrage Q_k ergibt sich aus der Differenz der Ausführungskosten mit (*cost*(Q_k, I_i)) und ohne Nutzung (*cost*(Q_k)) von I_i:

$$\textit{profit}(Q_k, I_i) = \max\{0, \textit{cost}(Q_k) - \textit{cost}(Q_k, I_i)\} \tag{1}$$

Gesucht ist eine Indexkonfiguration $\mathcal{C} \subseteq \{I_1, \ldots, I_n\}$ aus materialisierten Indexen (d.h. für Anfragen nutzbar), die

$$\sum_{i=1}^{m} \max\{\textit{profit}(Q_i, I_j) : I_j \in \mathcal{C}\} - \sum_{I_j \in \mathcal{C}} \textit{mcost}(I_j) \tag{2}$$

maximiert und dabei eine vorgegebene Größenschranke S nicht überschreitet:

$$\sum_{I_j \in \mathcal{C}} \textit{size}(I_j) \leq S \tag{3}$$

Es handelt sich hierbei um ein NP-Problem [Com78] als eine Variante des bekannten Rucksackproblems bzw. der ganzzahligen linearen Optimierung [KPP04]. In der Literatur sind hierzu verschiedene Lösungen beschrieben. Neben der naheliegenden Greedy-Variante und Ansätzen auf Basis der dynamischen Programmierung als exakte Lösungen wurden auch approximative Varianten vorgeschlagen [CFM95].

Autonomes Index Tuning als dynamische Lösung des ISP basiert auf grundlegenden Ansätzen zum Self Management in DBMS, die den Bemühungen in verschiedenen Teilgebieten gemein sind. Ein wichtiger Aspekt ist dabei das generelle Vorgehen zur Steuerung autonomen Systemverhaltens, an der auch die folgenden Darstellungen ausgerichtet sind. Grob kann das dynamische Verhalten in Phasen/Aufgaben unterteilt werden, welche kontinuierlich und iterativ durchlaufen werden.

(1) **Monitoring:** die Überwachung des vorhergehenden Systemverhaltens in Form von Messwerten und/oder Ereignissen über einen bestimmten Zeitraum, sowie eine geeignete Verwaltung der daraus gewonnenen Informationen.

(2) **Decision:** basierend auf den gewonnenen Informationen müssen Entscheidungen getroffen werden, die Änderungen des aktuellen Systemzustands nach sich ziehen, da zum Beispiel eine Optimierung des Systemverhaltens möglich ist oder sich abzeichnende Probleme oder Fehler vermieden werden können.

(3) **Action:** darauf folgend muss durch geeignete Aktionen die getroffene Entscheidung zur Änderung des Systemzustands umgesetzt werden, wobei der laufende Betrieb während der Aktionsausführung möglichst wenig beeinträchtigt werden soll.

Diese Phasen sind angelehnt an das im Rahmen des COMFORT-Projektes [WHMZ94] entwickelte Auto-Tuning-Modell eines Regelkreises aus den Phasen Observation (Monitoring), Prediction (Decision) und Reaction (Action). Die folgenden Ausführungen positionieren aktuelle Ansätze und den hier dargestellten Ansatz entsprechend dieser Phasen und beschreiben die einzelnen Aspekte des Monitoring, der Entscheidungsfindung und deren Umsetzung.

Die mit den kommerziellen DBMS inzwischen verfügbaren Index bzw. Design Wizards legen als physische Entwurfswerkzeuge den Fokus hauptsächlich auf dem Decision-Aspekt. Die erste verfügbare Lösung wurde von Microsoft im SQL Server 7.0 mit dem Index Wizard vorgestellt [CN97]. Ausgehend von einem Workload werden zunächst Indexkandidaten ausgewählt. Anschließend werden Indexkonfigurationen untersucht, indem sogenannte What-If-Indexe genutzt werden, deren Existenz dem Anfrageoptimierer „vorgetäuscht" wird. Der Optimierer wird dabei für die Kostenabschätzung genutzt. Die Auswahl der besten Indexkonfiguration erfolgt durch ein Greedy-Verfahren. DB2 bietet seit Version 6.1 mit dem Index Advisor ein ähnliches Werkzeug [VZZ+00]. Dieses nutzt eine Erweiterung des Anfrageoptimierers aus, der einen neuen Explain-Modus für Indexempfehlungen bietet. Auch hier werden durch einfache Heuristiken potenziell nützliche Indexe zunächst nur virtuell angelegt. Danach bestimmt der Optimierer unter Verwendung der virtuellen Indexe den besten Plan. Die im Plan genutzten virtuellen Indexe werden dann als Indexempfehlung ausgegeben. Der DB2 Index Advisor sammelt für eine gegebene Anfragemenge alle Indexempfehlungen und bestimmt wie oben beschrieben die optimale Indexmenge. Zusätzlich gibt es noch einen Modus, in dem zufällig einige Indexe der initialen Lösung des Rucksackproblems ausgetauscht werden bis eine vorgegebene Zeitschranke überschritten wird. In den folgenden DB2-Versionen wurde der Index Advisor zum Design Advisor weiterentwickelt, der nun auch materialisierte Sichten und multidimensionale Cluster-Tabellen unterstützt [ZRL+04].

Im Gegensatz zu früheren Arbeiten wie etwa [ISR83, FST88] sind diese Lösungen dadurch charakterisiert, dass sie den Anfrageoptimierer des DBMS nutzen. Auf diese Weise kann sichergestellt werden, dass bei der Ermittlung der Indexempfehlungen das gleiche Kostenmodell verwendet wird. Teilweise wird auch die Bestimmung eines Workloads unterstützt (Monitoring-Aspekt) – auch dies jedoch entkoppelt vom eigentlichen Betrieb. Techniken, die hier zum Einsatz kommen, sind u.a. das Clustering von Anfragen [GPSH02, CGN02].

In aktuellen Ansätzen wird der dynamische Aspekt des Tunings zunehmend berücksichtigt. So wird in [BC06b] ein Alerter vorgestellt, der ohne den Overhead eines Advisors im laufenden Betrieb Hinweise zu möglichen Verbesserungen der aktuellen Konfiguration und eine Abschätzung der unteren und oberen Grenze für das Optimierungspotential angibt. Dies entspricht einer Unterstützung der Phasen Monitoring und Decision. Der dynamische Aspekt in der Action-Phase wird in [ACN06] in der Hinsicht berücksichtigt, dass Workloads als sich wiederholende Sequenzen von Anfragen aufgefasst werden, die um geeignete Veränderungen der Indexkonfiguration ergänzt werden. In [BC06a] wird die Problemstellung des „Incremental Tuning" betrachtet, wobei die Operationen *Merge* und *Reduce* für das Anpassen der Konfiguration von materialisierten Sichten und Indexen vorgeschlagen werden. Diese betreffen jedoch nur die Decision-Phase und werden unabhängig vom Monitoring eines Workloads oder einer konkreten Strategie für die physische Veränderung (Action) dargestellt.

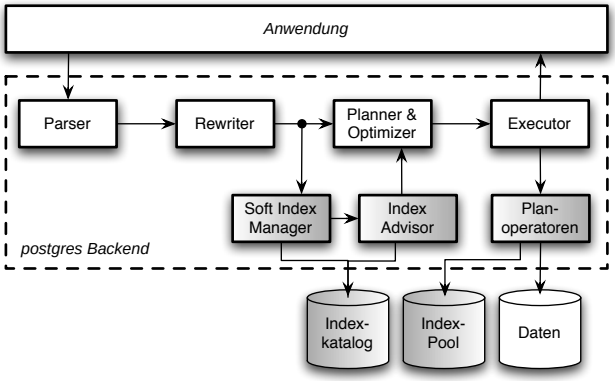


Abbildung 1: Integration des Soft-Index-Managements in PostgreSQL

Ein erster Ansatz eines echten autonomen Online-Tunings ist das auf DB2 basierende QUIET-System [SGS03, SSG04]. Dieses sammelt langfristig Informationen über vom Optimierer empfohlene Indexe und legt diese dann automatisch an bzw. löscht nicht mehr benötigte Indexe. Jedoch ist das System nicht in das DBMS integriert und verursacht einen gewissen Overhead.

3 Autonomes Index Tuning

Der in den folgenden Abschnitten vorgestellte Soft-Index-Ansatz stellt eine Weiterentwicklung des QUIET-Systems dar, indem neben einer engen Integration mit dem DBMS drei Ziele verfolgt werden: (1) das kontinuierliche Monitoring von Anfragen zur Identifikation von (potenziell) nützlichen Indexen, (2) das autonome Anlegen bzw. Löschen von Indexen unter Berücksichtigung von Speicherplatzschranken und Verwaltungskosten und (3) die Erzeugung von Indexen „on the fly“, d.h. durch Ausnutzung von Table-Scans.

3.1 Architektur

Die PostgreSQL-Erweiterung zum Soft-Index-Management besteht aus drei neuen Komponenten: dem Index-Advisor, dem eigentlichen Soft Index Manager, neue physische Planoperatoren, konkret der IndexBuildScan- sowie der SwitchPlan-Operator. Das Zusammenwirken dieser Komponenten im Rahmen der Anfrageverarbeitung in PostgreSQL ist in Abbildung 1 dargestellt. Die grau hinterlegten Komponenten stellen dabei die Erweiterungen dar. Der Soft-Index-Manager ist zwischen Rewriter und Planner eingefügt, um die Anfragen zu überwachen. Hierzu wird wiederum der Index Advisor genutzt (Abschnitt 3.2). Die Datenbank selbst ist um einen Index-Pool – einen Bereich fester Größe zum Speichern der Indexe – sowie den Indexkatalog als Systemtabelle mit den Indexinformationen erweitert.

Darüber hinaus wurden die B+-Baum-Indexe um folgende Eigenschaften erweitert:

- *managed* sind alle Soft Indexe, die vom System automatisch verwaltet (angelegt bzw. entfernt) werden,
- *virtual* Indexe werden nur für die Anfrageoptimierung bzw. die Indexempfehlung angelegt, d.h. sind im Data Dictionary eingetragen (inkl. der notwendigen Größeninformationen) jedoch nicht materialisiert und für andere Anfragen nicht sichtbar,
- *deferred* Indexe werden dagegen zunächst „leer" angelegt und erst später mit Daten gefüllt.

Die weiteren Betrachtungen beschränken sich daher auf B+-Baum-Indexe. Prinzipiell ist jedoch eine Übertragung auf andere Indexarten problemlos möglich.

3.2 Monitoring: Workload-Monitoring und Indexempfehlung

Das Monitoring zum Soft-Index-Management umfasst zwei Ebenen: die lokale Anfrageebene und die globale Workload-Ebene. Die *Anfrageebene* wird durch den Index Advisor behandelt, der dem in [VZZ+00] beschriebenen Ansatz folgt.

Bei der Analyse einer Anfrage Q zur Ableitung von Indexempfehlungen wird diese zuerst auf konventionelle Weise optimiert und die Kosten $cost(Q)$ werden ermittelt, wobei hierbei zunächst alle Soft Indexe ignoriert werden. Dann wird die Anfrage syntaktisch analysiert um potenzielle Indexkandidaten zu ermitteln. Indexkandidaten sind zum Beispiel einzelne Spalten in den **`where`**, **`group by`**, **`order by`** und **`select`**-Klauseln sowie Spaltenkombinationen in diesen Klauseln. Für jeden Indexkandidaten wird ein virtueller Index mit abgeschätzten Statistikinformationen angelegt. Nun wird die Anfrage Q unter Berücksichtigung der virtuellen Indexe noch einmal optimiert und die Kosten $cost(Q, I)$ werden bestimmt. Die im Plan genutzten Soft-Indexe (einschließlich der virtuellen Indexe) werden als Indexempfehlung $\mathcal{I} = \{I_1, \ldots, I_k\}$ zurückgegeben. Der Profit für $\mathcal{I}$ ergibt sich danach aus

$$profit(Q, \mathcal{I}) = cost(Q) - cost(Q, \mathcal{I}) \tag{4}$$

Der letzte Schritt lässt noch offen, wie der Gesamtprofit auf die einzelnen Elemente von $\mathcal{I}$ aufgeteilt wird. In [SSG04] haben wir dazu verschiedene Alternativen untersucht. Ein praktikabler Ansatz ist die Aufteilung entsprechend der Indexgrößen, d.h. der Profit eines Index $I \in \mathcal{I}$ berechnet sich aus:

$$profit(Q, I) = \frac{profit(Q, \mathcal{I}) \cdot size(I)}{\sum_{I_j \in \mathcal{I}} size(I_j)} \tag{5}$$

Hierbei ist zu berücksichtigen, dass es sich nur um Annäherungen handelt, da gilt:

$$profit(Q, \mathcal{I}) \neq \sum_{I \in \mathcal{I}} profit(Q, I) \tag{6}$$

Dies wird beispielsweise bei einer Anfrage mit einem Merge-Join deutlich: der Profit eines Plans mit zwei Indexen auf den Verbundattributen ist hier sicher größer als die Summe der Profite, wenn jeweils nur ein Index vorhanden wäre.

Ein zweites Problem ist die Behandlung von Update-Operationen, die ggf. eine Aktualisierung der Indexe erfordern. Ein möglicher Ansatz ist die Bestimmung eines negativen Profits für alle betroffenen Indexe auf Basis der Anzahl der betroffenen Tupel *nrows* der Operation Q_U und der Höhe des Baums (F bezeichnet hierbei einen empirisch ermittelten Kostenfaktor):

$$\textit{profit}(Q_U, I) = -\textit{height}(I) \cdot \textit{nrows}(Q_U) \cdot F \tag{7}$$

Auf *Workload-Ebene* besteht die Aufgabe des Monitorings im Sammeln und Aggregieren der Indexempfehlungen für die laufenden Anfragen. Dies wird vom Soft Index Manager übernommen, der dafür den Index Advisor nutzt. Eintreffende Anfragen werden dabei wie oben beschrieben vom Index Advisor verarbeitet. Hierbei muss nicht jede Anfrage berücksichtigt werden – es genügt eine repräsentative Stichprobe zu beobachten. Die gewonnenen Indexempfehlungen werden in einem Indexkatalog $\mathcal{D}$ gesammelt, der pro Index I folgende Informationen enthält:

- den Gesamtprofit $\textit{benefit}(I)$ aus den kumulierten Einzelprofiten pro Anfrage,
- die Größe $\textit{size}(I)$, die für noch nicht materialisierte Indexe aus der Kardinalität der Basisrelation und der Attributgröße abgeschätzt wird und bei materialisierten Indexen durch den tatsächlichen (aktuellen) Wert ersetzt,
- den Status $\textit{state}(I)$ des Index, d.h. ob der Index gegenwärtig materialisiert ist oder nicht.

Da die Bestimmung der optimalen Indexkonfiguration ein aufwändiger Prozess ist, wird dies nicht nach jeder Anfrage ausgeführt. Vielmehr wird jeweils eine *Epoche* von Anfragen betrachtet, in der die Indexempfehlungen zunächst nur gesammelt werden. Die Länge der Epochen kann einfach über die Anzahl der Anfragen oder durch das Erreichen eines vorgegebenen Gesamtprofits definiert werden.

Das Monitoring in Epochen ermöglicht darüber hinaus eine „Dämpfung“ des Einflusses länger zurückliegender Indexempfehlungen, sodass eine bessere Anpassung an Drifts im Workload erzielt werden kann. Hierzu werden die Indexempfehlungen (inklusive des erwarteten Profits) mit Zeitstempeln versehen (d.h. beispielsweise einen monoton wachsenden Anfragezähler). Der Gesamtprofit für eine Epoche, die zum Zeitpunkt ts_E endet, mit den Empfehlungen zu einem Index I aus Anfragen mit dem Zeitstempeln $ts_1, ts_2, \ldots, ts_k$ berechnet sich danach aus:

$$\textit{benefit}(I) = \sum_{j=1\ldots k} \frac{\textit{profit}(Q_{ts_j}, I)}{ts_E - ts_j} \tag{8}$$

Dieser Gesamtgewinn wird am Ende einer Epoche auf alle korrespondierenden Einträge im Indexkatalog aufsummiert.

Ein weiteres Problem ist die Berücksichtigung von sich überlappenden Indexen. Dies bedeutet beispielsweise, dass ein Index-Scan über einem Index auf $R(A)$ auch den Index auf $R(A, B)$ nutzen könnte. Demzufolge kann der Profit einer Indexempfehlung zu $R(A)$ auch dem Index zu $R(A, B)$ zugewiesen werden. Hierfür benötigen wir den Begriff des Index-Containment: ein Index I_1 ist in I_2 enthalten ($I_1 \sqsubseteq I_2$) gdw. beide Indexe auf der gleichen Relation definiert sind und die Attribute von I_1 unter Berücksichtigung der Sortierordnung des Indexes (aufsteigend bzw. absteigend) ein Präfix der Attribute von I_2 sind. In diesem Fall kann der Profit entsprechend zugewiesen werden:

$$\forall I_i \in \mathcal{D} : I_r \sqsubseteq I_i \Rightarrow \mathit{profit}(Q, I_i) = \mathit{profit}(Q, I_r) \tag{9}$$

3.3 Decision: Soft-Index-Auswahl

Aufgabe der Soft-Index-Auswahl ist die Bestimmung der optimalen Indexkonfiguration unter Berücksichtigung des aktuellen Workloads und der gegebenen Speicherplatzschranke. Wie bereits in Abschnitt 2 dargestellt, handelt es sich hierbei um das Rucksackproblem, wobei der Fokus auf der Online-Verarbeitung liegt, d.h. jeweils zum Ende einer Epoche soll die Entscheidung darüber getroffen werden, ob neue Indexe anzulegen sind bzw. ob ggf. materialisierte Indexe gelöscht werden sollen.

Ein praktikabler Lösungsansatz für dieses Problem ist die Nutzung eines Greedy-Verfahrens, das wie folgt arbeitet: Die Indexkandidaten (d.h. alle materialisierten und nicht-materialisierten Soft Indexe) werden bezüglich ihres relativen Gewinns

$$\mathit{relative\_benefit}(I) = \frac{\mathit{benefit}(I)}{\mathit{size}(I)}$$

absteigend sortiert. Anschließend wird Algorithmus 1 ausgeführt, der solange Indexkandidaten in eine initial leere Konfiguration aufnimmt, bis die Speichergrenze *plimit* erreicht ist.

Das Ergebnis ist eine neue Indexkonfiguration $\overline{\mathcal{C}}$, die jedoch nur dann realisiert wird, wenn der Gewinn gegenüber der alten Konfiguration $\mathcal{C}$ oberhalb eines vorgegebenen (empirisch ermittelten) Schwellwertes *threshold* liegt. Auf diese Weise wird das Thrashing – das wechselweise Anlegen und Löschen der gleichen Indexe – reduziert.

Das Realisieren einer Indexkonfiguration bedeutet prinzipiell, dass alle noch nicht materialisierten Indexe der neuen Konfiguration angelegt werden und die nicht mehr in der neuen Konfiguration vorhandenen materialisierten Indexe entsprechend zu löschen sind. Es sei angemerkt, dass sich Anlegen und Löschen nur auf den eigentlichen B+-Baum-Index bezieht – die Einträge im Indexkatalog bleiben in jedem Fall erhalten.

Der Aufwand dieses Auswahlverfahrens ist einschließlich der Sortierung offensichtlich $O(n \log n)$ bei n Indexkandidaten. Ein Nachteil des Greedy-Ansatzes ist jedoch, dass keine Garantie für das Finden der optimalen Lösung besteht. Trotzdem haben bisherige Forschungsergebnisse gezeigt, dass der gewählte Ansatz mit vertretbarem Aufwand ausreichend gute Ergebnisse liefert. Dies liegt an der Größe des Suchraums, der Verwendung

Algorithmus 1 Greedy-Algorithmus zur Indexauswahl

```
1: I[1 ... n] := sort(D) by relative benefit;
2: C := ∅;
3: avail_space := plimit;
4: overall_benefit := 0;
5: for all k := 1 ··· n do
6:    if avail_space - size(I[k]) > 0 then
7:       C := C ∪ {I[k]}
8:       avail_space := avail_space - size(I[k])
9:       overall_benefit := overall_benefit + benefit(I[k])
10:   end if
11: end for
12: if overall_benefit < threshold then C := C end if
13: return C
```

von Abschätzungen als Eingabeparameter und der Nutzung des Ergebnisses als Vorhersage des zukünftigen Anfrageverhaltens.

3.4 Action: Planoperatoren für Indexerstellung

Nachdem Soft Indexe als potenziell nützlich identifiziert sind, besteht der nächste Schritt in der Materialisierung dieser Indexe. Im Gegensatz zu den konventionellen Indexen sollen Soft Indexe automatisch erzeugt (und gelöscht) werden, ohne dass die Anfragen dabei signifikant verzögert werden. So können Indexempfehlungen lediglich in einer Tabelle des Schemakatalogs gesammelt werden, und der DBA kann dann später entscheiden, ob und wann er die Indexe von Hand anlegt bzw. löscht. Eine erste mögliche Automatisierung ergibt sich, wenn vorgeschlagene Indexe während einer Offline-Phase des Datenbanksystems oder Zeiten niedriger Systemlast angelegt werden. Soll die Indexkonfiguration im laufenden Betrieb möglichst aktuell sein, können die Indexe direkt vor der nächsten Anfrage, die diese Indexe nutzen könnte, erzeugt werden. Allerdings verzögert sich die Anfrageausführung dadurch unter Umständen erheblich.

Um sowohl eine möglichst aktuelle Indexkonfiguration als auch geringen Mehraufwand zu erreichen, verfolgen wir hier folgenden Ansatz. Die Indexe werden während der Anfrage erzeugt, indem z.B. ein Table-Scan auf der Basistabelle genutzt wird, um die Tupel zu lesen und in den Index einzufügen [Gra00]. Als weitere Optimierung wird der angelegte Index wenn möglich direkt in der Anfrage genutzt, indem der Plan entsprechend angepasst wird. Zur Umsetzung werden zwei neue Planoperatoren benötigt, die nachfolgend beschrieben werden. Basis hierfür bilden die neu eingeführten Deferred-Indexe – die Materialisierung der Indexe besteht somit im Erzeugen von Deferred-Indexen. Dies bedeutet, dass zunächst nur leere Indexe angelegt werden, die im Schemakatalog entsprechend markiert sind und erst später gefüllt werden.

3.4.1 IndexBuildScan

Der IndexBuildScan (notiert als ξ) ist eine Erweiterung des Table-Scan-Operators, die als zusätzlichen Parameter eine Liste von Deferred-Indexen erwartet. Der Operator liest zunächst wie der normale Scan-Operator blockweise alle Tupel der Basisrelation, fügt diese jedoch gleichzeitig in die entsprechenden Deferred-Indexe ein. Somit lassen sich prinzipiell in einem Scan mehrere Indexe zur gleichen Basisrelation füllen. In Algorithmus 2 ist das Verhalten entsprechend der klassischen Iterator-Schnittstelle dargestellt.

Algorithmus 2 IndexBuildScan-Operator $\xi_{\mathcal{L}}$ auf einer Relation r mit opt. Prädikat P

```
State:
 1: scan_complete := false
Open:
 2: for all I ∈ ILIST do
 3:    if deferred(I) then
 4:       prepare_index(I)
 5:    end if
 6: end for
 7: open_relation(r)
Close:
 8: close_relation(r)
 9: if scan_complete then
10:    for all I ∈ L do
11:       if deferred(I) then finish(I) end if
12:    end for
13: else
14:    for all I ∈ L do
15:       if deferred(I) then reset(I) end if
16:    end for
17: end if
Next:
18: tuple := seqscan_next(r)
19: if tuple ≠ ⊥ then
20:    for all I ∈ L do
21:       if deferred(I) then
22:          insert(I, tuple)
23:       end if
24:    end for
25:    if P(tuple) then
26:       return tuple
27:    else
28:       next()
29:    end if
30: else
31:    scan_complete := true
32:    return tuple
33: end if
```

Eine besondere Behandlung erfordert dabei der Fall, dass der Scan nicht vollständig ausgeführt wurde, z.B. durch Abbruch der Anfrage. Da hierbei nicht mehr garantiert werden kann, dass der Index vollständig gefüllt ist, muss der Index in einem solchen Fall zurückgesetzt werden.

Rewriting Regeln. Der IndexBuildScan-Operator kann im Anfrageplan überall dort eingesetzt werden, wo der Table-Scan $\sigma_P^{\mathbf{SEQ}}$ (mit dem optionalen Selektionsprädikat P) auf der gleichen Basisrelation ausgeführt werden soll. Als Randbedingung gilt, dass die Deferred-Indexe der Parameterliste $\mathcal{L}$ alles Indexe auf der jeweiligen Basisrelation sein müssen:

$$\sigma_P^{\mathbf{SEQ}}(r) \Leftrightarrow \sigma_P(\xi_{\mathcal{L}}(r)) \quad \text{falls } \forall I \in \mathcal{L} : I \text{ ist Deferred-Index auf } r \tag{10}$$

Kostenabschätzungen. Die Kosten für die Ausführung des Operators setzen sich aus den normalen Scan-Kosten und den Kosten für den Indexaufbau zusammen. Während die Scan-Kosten nur das Lesen aller Blöcke beinhalten, umfasst der Indexaufbau das Schreiben der Indexblöcke sowie ggf. ein Sortieren der eingefügten Tupel. Letzteres wird gerade für B+-Tree-Indexe zum Erzielen eines besseren Füllgrades angewendet. Die Anzahl der für den Index I benötigten Blöcke kann anhand der Attribut- und Rowid-Größe sowie des Füllfaktors bestimmt werden.

$$idx\_pages = \left\lceil \frac{|r|}{\left\lfloor \frac{page\_size}{(size_{attr}+size_{rowid})} \right\rfloor} \right\rceil \cdot \frac{1}{fill\_factor} \tag{11}$$

Da die Anzahl der inneren Knoten typischerweise viel kleiner als die der Blattknoten ist ($< 5\%$), kann diese Zahl ignoriert werden:

$$cost = \underbrace{pages(r)}_{\text{Scan auf } r} + \sum_{I \in \mathcal{L}} (\underbrace{|r| \log |r|}_{\text{Sortierung}} + \underbrace{idx\_pages(I) \cdot cost_{\text{page\_write}}}_{\text{Indexseiten schreiben}}) \tag{12}$$

Diskussion. Durch die Indexerstellung während der Anfrageausführung werden Leseoperationen prinzipiell zu Schreiboperationen – wenn auch nicht auf den Basisrelationen. Hierbei können zwei Probleme auftreten, die eine spezielle Behandlung erfordern. Zum Einen kann es zum Abbruch der Transaktion kommen (im Falle der Nicht-Serialisierbarkeit). Hier muss die Atomarität der Indexerstellung gewährleistet werden, indem der neu erstellte Index nur dann zur weiteren Verwendung freigegeben wird, wenn der Aufbau erfolgreich war. Das zweite Problem kann durch konkurrierende Anfragen auftreten, wenn für mehrere Anfragen ein IndexBuildScan für den gleichen Index vorgesehen ist. Beide Probleme lassen sich jedoch durch eine einfaches Zustandsmodell mit den Zuständen „Deferred“, „Under Construction“, „Ready“ behandeln, wobei Deferred-Indexe im Zustand „Under Construction“ für weitere Pläne mit IndexBuildScans nicht berücksichtigt werden und eine Indexnutzung erst im Zustand „Ready“ erfolgt.

Weiterhin sei angemerkt, dass der IndexBuildScan prinzipiell auch in Kombination mit dem Sort-Operator eingesetzt werden kann, indem Tupel sortiert ausgelesen und entsprechend in den B+-Baum eingefügt werden.

3.4.2 SwitchPlan

Die Verwendung des IndexBuildScan-Operators allein erlaubt noch nicht die Nutzung des erzeugten Indexes in der gleichen Anfrage. Hierfür wird der SwitchPlan-Operator (notiert als $\rightleftharpoons$) eingesetzt, der eine spezielle Variante des in [GW89] vorgestellten Choose-Plan-Operator darstellt. Aufgabe dieses Operators ist es, während der Ausführung eines Plans zwischen zwei Teilplänen umzuschalten. In unserem Fall ist dies das Umschalten zwischen einem IndexBuildScan und einem IndexScan als Teil einer Nested-Loops-Operation (z.B. eines Verbundes). In Abbildung 2 ist dies an einem Beispielplan dargestellt: Der Nested-Loops-Join mit den beiden Table-Scans auf den Relationen r und s wird durch einen Plan

ersetzt, der einen IndexBuildScan zum Erzeugen eines Indexes auf dem Verbundattribut von s mit dem SwitchPlan-Operator kombiniert. Der IndexBuildScan wird dabei nur während der ersten Iteration auf der s-Relation ausgeführt, in allen folgenden Durchläufen wird der neu erzeugte Index mit dem Index-Scan verwendet – der Nested-Loops-Join wird somit zum Index-Nested-Loops-Join.

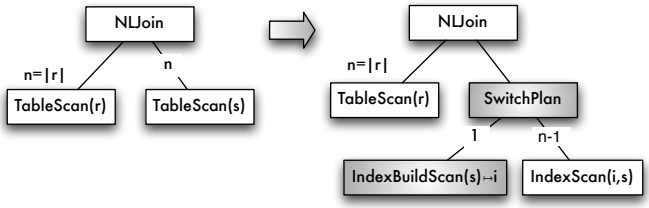


Abbildung 2: Beispiel für den Einsatz des SwitchPlan-Operators

Die Realisierung des Operators ist einfach: Es wird nur eine Zustandsvariable benötigt, die zwischen den beiden Phasen (links und rechts) unterscheidet (Algorithmus 3).

Algorithmus 3 SwitchPlan-Operator

State:
1: phase := 0
Open:
2: **if** phase = 0 **then**
3: phase := 1
4: left.open()
5: **else if** phase = 1 **then**
6: phase := 2
7: right.open()
8: **end if**

Next:
9: **if** phase = 1 **then** tuple := left.next() **end if**
10: **if** phase = 2 **then** tuple := right.next() **end if**
11: **return** tuple
Close:
12: **if** phase = 1 **then** left.close() **end if**
13: **if** phase = 2 **then** right.close() **end if**

Rewriting Regeln. Der SwitchPlan-Operator kann grundsätzlich genutzt werden, um zwischen einem normalen Scan und einem IndexBuildScan umschalten zu können (Regel 13 und 14). Daher kommt der Operator primär als (rechter) Kindknoten von Nested-Iteration-Knoten wie z.B. Nested-Loops-Join, Mengenoperationen oder geschachtelte Selektionen zum Einsatz (15). Voraussetzung für eine Ersetzung ist weiterhin die Existenz eines IndexBuildScan weiter unten im Baum, wobei einer der zu erzeugenden Indexe die Verbund- bzw. Selektionsbedingung der Nested-Iteration-Operation unterstützen muss:

$$\sigma_P^{\mathbf{SEQ}}(r) \quad \Leftrightarrow \quad (\sigma_\phi(\xi_{\mathcal{L}}(r)) \rightleftharpoons \sigma_P^{\mathbf{IND}}(r)) \tag{13}$$

$$\sigma_P^{\mathbf{IND}}(r) \quad \Leftrightarrow \quad (\sigma_\phi(\xi_{\mathcal{L}}(r)) \rightleftharpoons \sigma_P^{\mathbf{IND}}(r)) \tag{14}$$

$$r \bowtie_P s \quad \Leftrightarrow \quad (r \bowtie_P (\sigma_P(\xi_{\mathcal{L}}(s)) \rightleftharpoons \sigma_P^{\mathbf{IND}}(s))) \tag{15}$$

Kostenabschätzungen. Die Gesamtkosten für einen Teilplan mit einem SwitchPlan-Operator setzen sich aus den Kosten des linken Teilplans und den $(card_{left} - 1)$-fachen Kosten des rechten Teilplans zusammen, wobei $card_{left}$ die (geschätzte) Kardinalität des Ergebnisses des linken Teilpans bezeichnet.

$$cost = cost_{left} + (card_{left} - 1) \cdot cost_{right} \tag{16}$$

Ist der Teilplan mit der Nested Iteration wiederum rechter Teil einer weiteren Nested Iteration, muss beachtet werden, dass der IndexScan-Kostenanteil rekursiv pro Iteration in die Gesamtkosten eingeht, während der Anteil der Indexerstellungskosten nur einmal eingeht.

Diskussion. Der SwitchPlan-Operator zielt auf die Ausnutzung des erzeugten Indexes in der gleichen Anfrage ab (Intra-Query-Nutzung) und ist somit im Wesentlichen für Pläne mit Nested Iterations sinnvoll. Orthogonal dazu ist die Nutzung der erzeugten Indexe in einer parallel laufenden Anfrage (Inter-Query-Nutzung): Sobald ein Soft Index im Zustand „Ready“ vorliegt, kann er für einen IndexScan in anderen Anfragen eingeplant werden, auch wenn die erzeugende Anfrage noch nicht abgeschlossen ist. Grundsätzlich denkbar ist auch eine dynamischere Variante, bei der zum Ausführungszeitpunkt der Anfrage im SwitchPlan-Operator geprüft wird, ob inzwischen ein Index für einen IndexScan vorliegt. Dies würde jedoch eine Verschmelzung von Planungs- und Ausführungsphase [GW89] erfordern und ist von uns daher bisher nicht realisiert.

3.4.3 Entscheidungsmodell

Werden die Erstellungskosten für Indexe einfach in die Gesamtkosten eines Plans einbezogen, so würde dieser Plan aufgrund der höheren Kosten vom Optimierer zugunsten eines anderen, nicht-indexerzeugenden Planes verworfen. Daher werden die Gesamtkosten in Ausführungs- und Erzeugungskosten aufgeteilt, wobei letztere die Summe aller Erzeugungsanteile von IndexBuildScan-Operatoren im Plan sind. Der Optimierer wählt dann einen indexerzeugenden Plan (mit oder ohne SwitchPlan-Operator) aus, wenn folgende Bedingungen erfüllt sind:

1. die Verwendung von Soft Indexen ist eingeschaltet (Konfigurationsparameter),
2. es liegen Deferred-Indexe für die in der Anfrage referenzierten Basisrelationen vor,
3. der Plan ist bezüglich der reinen Ausführungskosten der günstigste,
4. der Gesamtprofit der zu erzeugenden Indexe abzüglich der Erzeugungskosten ist positiv bzw. überschreitet ein gegebenes Limit.

Das Anlegen der Deferred-Indexe erfolgt davon unabhängig durch den Soft-Index-Manager. Dies bedeutet, dass (1) nur dann indexerzeugende Pläne ausgewählt werden, wenn zuvor passende Soft Indexe angelegt wurden und dass (2) die Materialisierung von Indexen bis zum Ausführen einer entsprechenden Anfrage verzögert werden kann.

3.5 PostgreSQL-Implementierung

Die beschriebenen Komponenten wurden vollständig in PostgreSQL 7.4.8 implementiert. Hierzu waren Erweiterungen in folgenden Bereichen notwendig.

Systemkatalog. Der Systemkatalog wurde um Relationen zur Verwaltung der Indexkandidaten erweitert. Konkret sind dies

- die Erweiterung der `pg_index`-Relation um ein Feld zur Kennzeichnung von Soft-Indexen bzw. Deferred-Indexen,
- die Relation `pg_softindex_indexadvice` zur Speicherung der Empfehlungen einer Epoche,
- die Relation `pg_softindex_indexinfo` als Realisierung des Indexkatalogs.

Die beiden letzteren Relationen werden nur vom Soft-Index-Manager genutzt, während die Informationen der ersten Relation auch vom Planner verwendet werden.

Planner. In den Ablauf des Planners wurde der Aufruf des Soft-Index-Managers eingefügt. PostgreSQL steuert die Verarbeitung über einen sogenannten Traffic Cop (tcop), der wiederum für eine geparste Anfrage den Planner aufruft. Dementsprechend wurde der Soft-Index-Manager vor dem Aufruf des Planners eingefügt. Der Soft-Index-Manager nutzt wiederum den Index-Advisor zur Ableitung der Indexempfehlungen. Diese werden, wie in Abschnitt 3.2 und 3.3 beschrieben, verarbeitet, sodass die neue Indexkonfiguration anschließend im Indexkatalog abgelegt ist.

Der zweite Bereich an Erweiterungen des Planners betrifft die Berücksichtigung der neuen Planoperatoren sowie deren Kostenfunktionen. Bei der Plankonstruktion führt der PostgreSQL-Planner einen Schritt aus, in dem alle indexgestützten Zugriffspfade hinzugefügt werden. An dieser Stelle müssen nun auch Deferred-Indexe berücksichtigt werden, wobei hierzu jedoch IndexBuildScan-Operatoren eingefügt werden. Darüber hinaus wurde die Kostenberechnung für den Indexzugriff entsprechend der Abschätzungen aus Abschnitt 3.4 erweitert.

Executor. Die Executor-Erweiterungen betreffen im Wesentlichen die Umsetzung der neuen Planknoten. In der Implementierung wurden zur Vereinfachung speziell der Anfrageplanung die beiden Operatoren ξ und $\rightleftharpoons$ in einem Planknoten zusammengefasst, der beim ersten Open-Next-Close-Zyklus den IndexBuildScan (sofern Deferred-Indexe vorhanden sind, sonst einfach den TableScan) realisiert und – wenn benötigt – bei allen Folgeaufrufen in die zweite Phase umschaltet.

Konfiguration. Das Verhalten des Soft-Index-Managements kann (zumindest in der Testphase) über einige Parameter beeinflusst werden. Der wichtigste Parameter ist dabei die Variable `softindex_level`, die interaktiv über eine `set`-Anweisung belegt

werden kann. Die möglichen Werte reichen von 0 (kein Soft-Index-Management) über 3 (passiver Modus: Soft Indexe werden zwar identifiziert jedoch nicht angelegt) bis 5 (aktiver Modus: Soft Indexe werden ggf. als Deferred-Indexe angelegt).

4 Evaluierung

Zur Bewertung von Konzept und Implementierung des Soft-Index-Managements wurden eine Reihe von experimentellen Untersuchungen durchgeführt. Neben dem Einfluss bzw. Verhalten der einzelnen Komponenten sollte dabei insbesondere die Frage der Praktikabilität des Ansatzes beantwortet werden. Für alle Experimente wurde die Datenbank des TPC-H-Benchmarks[1] mit einem Scale-Faktor von 1 (= 1 GB) eingesetzt. Die verwendete Hardware war Pentium 4 mit 3 GHz und 1 GB RAM unter SUSE Linux 10. Vorgesehen war darüber hinaus auch die Nutzung des TPC-H-Anfragemixes, jedoch traten im Betrieb mit PostgreSQL eine Reihe von Problemen auf. So waren in der verwendeten Version 7.4.8 einige Anfragen nicht einmal optimierbar, bei anderen Anfragen führte das alleinige Vorhandensein zusätzlicher mehrspaltiger Indexe zu deutlich längeren Laufzeiten – offensichtlich ein Fehler im Kostenmodell in PostgreSQL. Daher wurden eigene Workloads erzeugt, auf die in den einzelnen Abschnitten genauer eingegangen wird.

Indexerzeugung. Es wurde der Overhead des IndexBuildScan-Operators im Vergleich zum äquivalenten **`create index`** bzw. sequenziellen Scan (als 100% gesetzt) untersucht. Dazu wurden verschiedene Indexe auf den Spalten der TPC-H-Tabellen betrachtet (Abbildung 3). Die Präfixe der Spaltennamen geben gleichzeitig die Tabelle an („`p_`“ für `part`, „`l_`“ für `lineitem`, „`ps_`“ für `partsupp`), „`multi`“ bezeichnet einen Multi-Column-Index auf `lineitem (l_partkey,l_suppkey,l_orderkey)`.

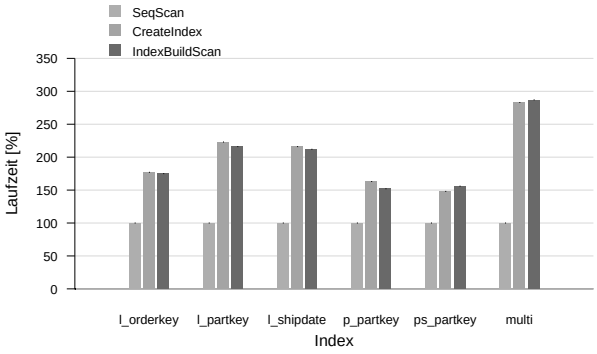


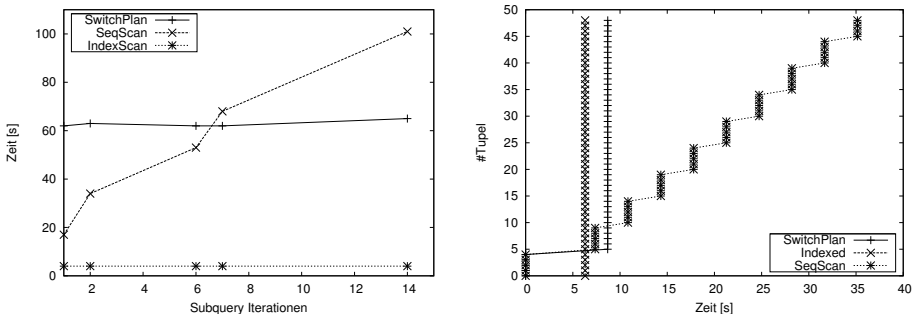
Abbildung 3: **`create index`** vs. IndexBuildScan

Die Zeiten für die Scans wurden mit einer Anfrage der Form „**`select count`**(*) **`from`** …“ bestimmt. Die Zeitunterschiede bei der Indexerstellung sind durch die unterschiedliche Anzahl der Distinct-Werte der jeweiligen Spalte bedingt. Ein Vergleich mit der Kombination „**`create index`**+IndexScan“ ist an dieser Stelle nicht angegeben, da die Zeit für den IndexScan im Wesentlichen von der Selektivität der Bedingung abhängig ist bzw. ein In-

[1] `www.tpc.org/tpch`

dexScan über die gesamte Relation nicht sinnvoll ist. Wie aus dem Diagramm ersichtlich wird, verursacht der IndexBuildScan wie erwartet die gleichen Kosten wie ein **`create index`**, hat demgegenüber jedoch den Vorteil, die Indexerstellung „nebenbei“ auszuführen. Der Preis dafür ist allerdings eine Verzögerung der Anfrage – im Falle eines einfachen Scans – um $50 - 150\%$.

Prinzipiell bietet der IndexBuildScan noch die Möglichkeit, mehrere Indexe auf der gleichen Tabelle in einem Scan zu erstellen. Allerdings haben unsere Experimente gezeigt, dass dies gegenüber einer sequenziellen Indexerstellung aufgrund des konkurrierenden Indexaufbaus und der Dominanz der Erstellungskosten keinen Gewinn liefert.



(a) Anfrage mit Subselect

(b) Nested Loops auf `part` ⋈ `partsupp`

Abbildung 4: Verhalten bei Nested Iterations

Nach dem IndexBuildScan-Operator wurde mit den folgenden Tests die Anwendung des SwitchPlan-Operators untersucht. Zunächst sollten geschachtelte Anfragen den Gewinn einer dynamischen Indexerstellung zeigen. Das Diagramm in Abbildung 4(a) zeigt, dass sich in Abhängigkeit von der Anzahl der Subselect-Iterationen die dynamische Indexerstellung typischerweise schon nach wenigen Iterationen lohnt, vorausgesetzt, dass die Anfragebedingung vom Index profitiert. Als Nächstes wurde die iterative Arbeitsweise eines Nested-Loop-Verbundes mit SwitchPlan-Unterstützung untersucht. Dafür wurden die Tabellen `part` und `partsupp` über die Spalte `partkey` verbunden und das Verbundattribut entsprechend dynamisch innerhalb der Anfrage indexiert. Abbildung 4(b) zeigt das Tupelrückgabeverhalten des Nested-Loop-Verbund-Operators in Abhängigkeit der Indexierungsarten. Die Kosten für die vorherige Indexerstellung („Indexed“) wurden als Offset zu dem Ausgabeverhalten addiert. Der stufenförmige Kurvenverlauf bei „SeqScan“ lässt auf die Iterationen schließen. Im Ergebnis liefert der SwitchPlan-Operator zunächst schneller Tupel, weist dann jedoch gegenüber der Indexed-Variante aufgrund der gleichzeitigen Indexerzeugung einen geringen Overhead auf.

Vor allem die Verarbeitung geschachtelter Anfragen profitiert von der dynamischen Indexerzeugung, allerdings nur bei Plänen mit Nested Iterations (Nested Loops, Subselects). Bei anderen Verbundoperationen ergeben sich aus dem SwitchPlan-Operator dagegen keine Vorteile. Die neuen Operatoren unterstützen komplexe Anfragen, können jedoch den **`create index`**-Befehl noch nicht komplett ersetzen. Die Integration in die Anfrageverarbeitung lässt allerdings die sofortige Indexnutzung zu und reduziert die Externspeicherzugriffe.

Soft-Index-Management. Aufgrund der oben dargestellten Probleme mit den TPC-H-Anfragen und PostgreSQL wurde ein eigener Workload auf dem TPC-H-Schema definiert, der wie folgt gebildet wurde. Zunächst wurden Anfragemixe aus je 10 Blöcken mit jeweils 23 Anfragen definiert. Jeder Anfragemix weist dabei ein charakteristisches Zugriffsverhalten (bestimmte Tabellen, Selektionen auf bestimmten Spalten) auf. Aus der Kombination dieser 3 Mixe zu einer Folge von 6 Mixen ergibt sich dann der Gesamtworkload. Der Wechsel der Mixe führt zu einer Änderung des Zugriffsverhaltens, was wiederum eine Adaption der Indexkonfiguration erfordert. Insgesamt führen die Anfragen zu 17 Indexkandidaten, davon 8 große (auf `lineitem`) und 9 kleine. Darüber hinaus wird für die Ausführung der Blöcke ungefähr die gleiche Zeit benötigt.

Zunächst wurde der Workload ohne und mit allen nützlichen (von Hand ausgewählten) Indexen ausgeführt (d.h. ohne Soft-Index-Management), um Ober- und Untergrenze zu bestimmen („Keine Indexe“ = `NONE` bzw. „Alle Indexe“ ≡ Hard Indexe = `ALL`). Die Gesamtzeit für die Ausführung wurde mit den Zeiten unter Einbeziehung des Soft-Index-Managements mit verschiedenen Größen des Index-Pools (angegeben in Seiten) sowie verschiedenen Epochenlängen (angegeben in Anzahl Anfragen) verglichen, wobei nur einspaltige Indexe berücksichtigt wurden.

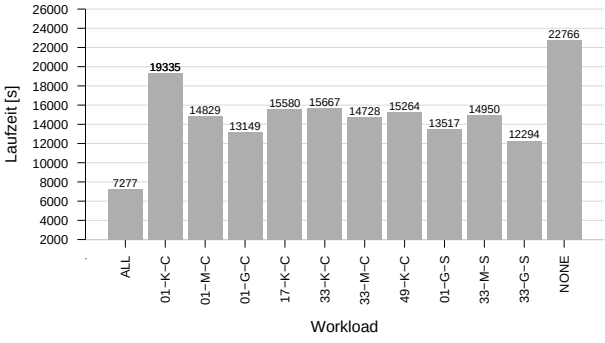


Abbildung 5: Gesamtzeit für Workload-Ausführung

Die Index-Pool-Größen wurden dabei so gewählt, dass alle Indexe materialisiert werden können („groß“ = `G`) bzw. nur die für einen Anfragemix optimalen Indexe, d.h. jeweils die großen `lineitem`-Indexe und die entsprechenden kleinen („mittel“ = `M`) bzw. nur die kleinen Indexe („klein“ = `K`). Die Konfigurationen sind in Abbildung 5 entsprechend durch `EE-P-I` kodiert, wobei `EE` für die Epochenlänge, `P` für die Poolgröße (groß ... klein) und `I` für die Art der Indexerstellung (`C` über **`create index`**, `S` über Index-BuildScan) stehen. Die Ergebnisse zeigen insbesondere, dass die Größe des Index-Pools hier einen wichtigen Einflussfaktor darstellt (von `01-K-C` zu `01-G-C` eine Verbesserung um 32%) und dass die Einführung der Epochen zu einem Gewinn führt (von `01-K-C` zu `49-K-C` 21%). Allerdings wird auch ersichtlich, dass die On-the-Fly-Indexerstellung nicht zwingend einen Vorteil bedeutet (`33-M-C` vs. `33-M-S`) – sondern nur dann zum Tragen kommt, wenn Anfragen direkt von den zu erstellenden Indexen profitieren.

Im nächsten Schritt wurde das Adaptionsverhalten während der Ausführung des Workloads für eine gewählte Kombination der Parameter (`33-K-C`, nur einspaltige Indexe) untersucht. Hierzu wurde die Ausführungszeit pro Anfrageblock bestimmt und mit den korrespondierenden Werten der Konfigurationen „Kein Index“ und „Alle Indexe“ ver-

glichen (Abbildung 6). Im Diagramm sind deutlich die Adapationsschritte (Ausreißer nach oben) der „Soft Index“-Kurve zu erkennen.

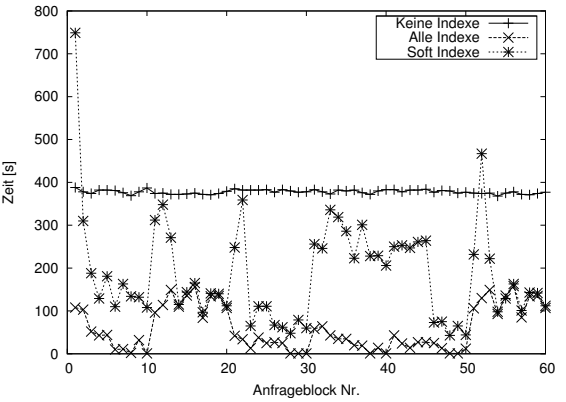


Abbildung 6: Adaptionsverhalten

Beim Vergleich mit der unteren Zeitschranke muss außerdem beachtet werden, dass dort die Zeit für die Indexerstellung nicht eingerechnet ist. Diese betrug für alle Indexe insgesamt 1010 Sekunden. Aus den Ergebnissen wird aber ersichtlich, dass hier einige Anfragen, die zufällig zum Zeitpunkt einer notwendigen Änderung der Indexkonfiguration eintreffen, die „Last“ der Indexerstellung tragen müssen (erkennbar an den Spitzen in der Kurve). Allerdings zeigt dieses Experiment auch, dass dies immer noch eine bessere Alternative als die Situation ohne geeignete Indexe ist.

Schließlich wurde noch der Overhead des Soft-Index-Managements untersucht. Dazu wurden bei einer Epochenlänge von 33 und einer Index-Pool-Größe von knapp 60.000 Seiten die Zeiten für die Bearbeitung der Workloads gemessen. Abbildung 7 gibt die prozentualen Anteile für die einzelnen Schritte an, ohne den Anteil für die Anfrageausführung.

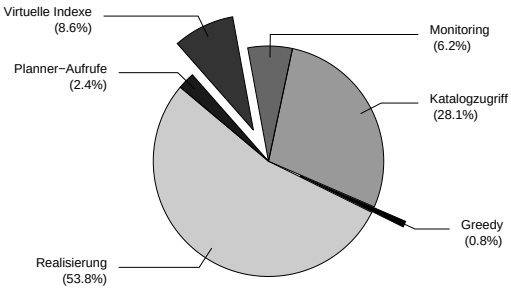


Abbildung 7: Aufwand des Soft-Index-Managements

Hierbei hat sich gezeigt, dass der Aufwand der eigentlichen Auswahl (Greedy) sowie der Behandlung der virtuellen Indexe vernachlässigbar ist. Neben dem nachvollziehbaren hohen Anteil zur Realisierung der Konfiguration sind in unserer Implementierung die Katalogzugriffe recht teuer: Nach jeder betrachteten Anfrage werden die Indexempfehlungen in die PostgreSQL-Katalogtabellen des Soft-Index-Managers übernommen, was mit Lese- und Schreibzugriffen auf den Katalog verbunden ist. Hier sind noch Optimierungen notwendig, etwa durch Halten der benötigten Informationen in einem Shared-Memory-Bereich.

5 Fazit & Zusammenfassung

In diesem Beitrag haben wir einen Ansatz zum autonomen Index Tuning in RDBMS vorgestellt, der auf der Idee der Soft Indexe basiert und einen Regelkreis aus den Phasen Monitoring, Decision und Action realisiert. Neben der kontinuierlichen Überwachung des Anfrage-Workloads und der Auswahl der nützlichen Indexe ist dabei die Erzeugung der Indexe und ihre direkte Nutzung während der Anfrageverarbeitung ein wesentlicher Beitrag.

Die Erfahrungen aus Implementierung und Evaluierung in PostgreSQL und der Vergleich mit unserem Vorhaben in QUIET haben gezeigt, dass eine enge Integration in die Anfrageoptimierung und -ausführung möglich und sinnvoll ist, speziell um das Kostenmodell zu nutzen und den Overhead zu reduzieren. Auf diese Weise ist ein autonomes Tuning möglich, dass insbesondere für sich ändernde Workloads geeignet ist.

Im Detail haben sich aber auch einige offene Punkte gezeigt. So bestehen noch mehrere „Performanzlöcher“ wie etwa beim Zugiff auf den Schemakatalog. Auch führen Ungenauigkeiten bzw. Fehler im Kostenmodell von PostgreSQL zu nicht erklärbaren Verhalten. Abgesehen von diesen technischen Schwierigkeiten haben wir zunächst auch weitere Tuning-„Regler“ eingeführt, wie etwa die Epochenlänge, die Größe des Index-Pools sowie mehrere Schwellwerte. Natürlich sollten diese Werte ebenfalls vom System durch geeignete Heuristiken selbst gesetzt werden, z.B. die Epochenlänge durch das Erkennen des Zeitpunkts eines notwendigen Re-Tunings [BC06b].

Ein weiterer Punkt ist, dass der Aufwand der Indexerstellung nicht vollständig eliminiert werden kann – vermeiden lassen sich nur mehrfache Scans. Hierbei muss auch noch berücksichtigt werden, dass die vorgestellte „Huckepack“-Indexierung eine gewisse Verzögerung der Anfrage verursacht und nicht immer ist dies akzeptabel. Ein möglicher Ausweg sind eventuell dynamische sich „selbst-tunende“ (d.h. zugriffsbalancierte) Indexstrukturen [SSG05].

Literatur

[ACN06] S. Agrawal, E. Chu und V. R. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *SIGMOD ’06*, Seiten 683–694, 2006.

[BC06a] N. Bruno und S. Chaudhuri. Physical Design Refinement: The Merge-Reduce Approach. In *EDBT*, Seiten 386–404, 2006.

[BC06b] N. Bruno und S. Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *Proc. 32nd VLDB Conference*, Seiten 499–510, 2006.

[CFM95] A. Caprara, M. Fischetti und D. Maio. Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):955–967, 1995.

[CGN02] S. Chaudhuri, A.K. Gupta und V. Narasayya. Compressing SQL Workloads. In *Proc. ACM SIGMOD Conference 2002*, Seiten 488–499, Madison, Wisconsin, 2002.

[CN97] S. Chaudhuri und V.R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proc. 23rd VLDB Conference*, Seiten 146–155. Morgan Kaufmann, 1997.

[Com78] D. Comer. The Difficulty of Optimum Index Selection. *ACM Transactions on Database Systems*, 3(4):440–445, 1978.

[FST88] S.J. Finkelstein, M. Schkolnick und P. Tiberio. Physical Database Design for Relational Databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988.

[GPSH02] A. Ghosh, J. Parikh, V.S. Sengar und J.R. Haritsa. Plan Selection based on Query Clustering. In *Proc. 28th VLDB Conference*, Seiten 179–190, Hong Kong, China, 2002.

[Gra00] G. Graefe. Dynamic Query Evaluation Plans: Some Course Corrections? *Bulletin of the Technical Committee on Data Engineering*, 23(2):3 – 6, June 2000.

[GW89] G. Graefe und K. Ward. Dynamic Query Evaluation Plans. In *ACM SIGMOD Conference 1989*, Seiten 358–366. ACM Press, 1989.

[ISR83] M.Y.L. Ip, L.V. Saxton und C.C. Raghavan. On the Selection of an Optimal Set of Indexes. *IEEE Transactions on Software Engineering*, 9(2):135–143, 1983.

[KPP04] H. Kellerer, U. Pferschy und D. Pisinger. *Knapsack Problems*. Springer-Verlag, Berlin, Heidelberg, 2004.

[KSRM03] T. Kraft, H. Schwarz, R. Rantzau und B. Mitschang. Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences. In *Proc. VLDB 2003*, Seiten 488–499, 2003.

[SGS03] K. Sattler, I. Geist und E. Schallehn. QUIET: Continuous Query-driven Index Tuning. In *Proc. 29th VLDB Conference*, Seiten 1129–1132, 2003.

[SSG04] K. Sattler, E. Schallehn und I. Geist. Autonomous Query-driven Index Tuning. In *Proc. Int. Database Engineering and Applications Symposium (IDEAS 2004), Coimbra, Portugal*, Seiten 439–448, Juli 2004.

[SSG05] K. Sattler, E. Schallehn und I. Geist. Towards Indexing Schemes for Self-Tuning DBMS. In *ICDE Workshops 2005*, Seite 1216, 2005.

[VZZ+00] G. Valentin, M. Zuliani, D. Zilio, G. Lohman und A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proc. ICDE 2000*, Seiten 101–110. IEEE, Marz 2000.

[WHMZ94] G. Weikum, C. Hasse, A. Moenkeberg und P. Zabback. The COMFORT Automatic Tuning Project, Invited Project Review. *Information Systems*, 19(5):381–432, 1994.

[ZRL+04] D.C. Zilio, J. Rao, S. Lightstone, G.M. Lohman, A. Storm, C. Garcia-Arellano und S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proc. 30th VLDB Conference*, Seiten 1087–1097, 2004.