Adaptive Optimierung von Datenbankanfragen

Andreas M. Weiner

Technische Universität Kaiserslautern
Fachbereich Informatik
D-67653 Kaiserslautern, Deutschland
a_weiner@informatik.uni-kl.de

Zusammenfassung Die meisten Verfahren zur Anfrageoptimierung in Datenbankverwaltungssystemen (DBVS) begründen ihre Entscheidung über die Auswahl des besten Anfrageausführungsplans mit Hilfe eines Kostenmodells. Die Kostenabschätzung hängt dabei sehr stark von den Schätzwerten des Optimierers für die Anzahl der Ergebnisse der einzelnen Planoperatoren ab, die im Anfrageausführungsplan verwendet werden. Bei konventionellen Verfahren zur Anfrageoptimierung wird keine automatische Aktualisierung der Statistiken vorgenommen. Daraus resultiert eine zunehmende Ungenauigkeit des Kostenmodells, was zur Auswahl von suboptimalen Anfrageausführungsplänen führt und damit die Leistung des DBVS negativ beeinflusst. Zur Lösung dieses Problems stelle ich die adaptive Optimierung von Datenbankanfragen vor.
Ich zeige, wie selbstwartende Histogramme zur effizienten Verwaltung von Statistiken eingesetzt werden können. Anschließend stelle ich das Verfahren der adaptiven Selektivitätsabschätzung als nicht histogrammbasierten Ansatz zur Adaption der Selektivitätswerte vor. Danach führe ich einen Feedback-Mechanismus ein, der die kontinuierliche Adaption der Statistiken an die tatsächlichen Gegebenheiten gestattet. Zum Schluss stelle ich den IBM LEarning Optimizer (LEO) als kommerziell verfügbares System zur adaptiven Anfrageoptimierung vor.

1 Einleitung

Zu der rasanten Verbreitung von relationalen Datenbankverwaltungssystemen (DBVS) in den letzten zwanzig Jahren hat die Standardisierung der Anfragesprache SQL (engl. *Structured Query Language*) in nicht unerheblichem Maße beigetragen. SQL ist eine deklarative Anfragesprache, d. h., der Benutzer gibt lediglich an, welche Daten er angezeigt haben möchte. Er muss jedoch nicht spezifizieren, wie das DBVS an die Daten gelangt. Zusätzlich lässt der deklarative Charakter von SQL großen Spielraum für die Optimierung der gestellten Anfragen. Nach Jarke und Koch [1] umfasst die Anfrageoptimierung (engl. *query optimization*) eine große Anzahl von Techniken und Strategien, beginnend bei der logischen Transformation von Anfragen, über die Optimierung von Zugriffspfaden reichend, bis hin zur effizienten Speicherung der Daten auf der Dateisystem-Ebene, da bereits auf der untersten Ebene das Fundament für eine effiziente Anfrageverarbeitung gelegt wird. Ibaraki und Kameda [2] haben gezeigt, dass

das Problem der Anfrageoptimierung zur Klasse der NP-vollständigen Probleme gehört. Das bedeutet, dass nach heutigem Kenntnisstand eine exakte Lösung des Optimierungsproblems in polynomieller Zeit nicht möglich ist. Aus diesem Grund wird in diesem Zusammenhang überwiegend auf Heuristiken zurückgegriffen.

Die allgemeine Zielsetzung der Optimierung bewegt sich im Spannungsverhältnis zwischen der Maximierung des Outputs bei gegebenen Ressourcen und der Minimierung der Ressourcennutzung bei gegebenem Output. Bei der Anfrageoptimierung spielen Kosten eine wichtige Rolle. Für Jarke und Koch [1] sind folgende Kosten relevant: Kommunikationskosten, Kosten für physische E/A-Zugriffe, Speicherungskosten sowie Berechnungskosten.

Im Rahmen des Forschungsgebietes der verlässlichen, adaptiven Informationssysteme (engl. *Dependable Adaptive Information Systems, DAIS*) wird die Anfrageoptimierung insbesondere unter dem Gesichtspunkt der Adaption betrachtet. Dabei stellt sich die Frage, wie sich der Optimierer an eine sich ändernde Umgebung – also eine Änderung der Parameter des Kostenmodells und der Anfragehäufigkeit – selbstständig anpassen kann, um weiterhin gute Ergebnisse zu erzielen.

Diese Ausarbeitung ist folgendermaßen aufgebaut: Im Abschnitt 2 gebe ich eine Einführung in die wichtigsten Begriffe und den Ablauf der Anfrageoptimierung. Abschnitt 3 zeigt einige Lösungsansätze für die adaptive Optimierung von Datenbankanfragen auf. In Abschnitt 4 werde ich *LEO (LEarning Optimizer)* vorstellen. Dabei handelt es sich um einen adaptiven Ansatz zur Anfrageoptimierung der Firma IBM, der bereits in kommerzielle Produkte wie z. B. die DB2 Universal Database eingeflossen ist. Im Abschnitt 5 werde ich dann ein Fazit ziehen und zukünftige Entwicklungen skizzieren.

2 Grundlagen der Anfrageoptimierung

In diesem Abschnitt stelle ich die Grundkonzepte der Optimierung von Datenbankanfragen vor. Ich beginne mit den wesentlichen Verarbeitungsschritten, die bei der Optimierung durchzuführen sind. Anschließend werde ich zeigen, welche Kostenmodelle und -abschätzungen man bei der Optimierung verwenden kann. Zum Schluss stelle ich einige Annahmen vor, die der Optimierung zugrunde liegen, und zeige, inwiefern diese problematisch sein können.

2.1 Verarbeitungsschritte bei der Anfrageoptimierung

Betrachtet man das fünfschichtige Architekturmodell für Datenbanksysteme nach Härder und Rahm [3], so lässt sich der Anfrageoptimierer der obersten Schicht zuordnen, welche die *mengenorientierte DB-Schnittstelle* zur Verfügung stellt. Diese Schicht ist hauptsächlich für die Übersetzung und Ausführung von mengenorientierten DB-Operationen zuständig.

Man kann zwei Arten von Optimierern unterscheiden: regelbasierte und kostenbasierte Optimierer. Für unsere Zwecke sind kostenbasierte Optimierer besser

geeignet, da diese die Kosten in jeder Schicht des Architekturmodells berücksichtigen können.

Abbildung 1 stellt die wesentlichen Verarbeitungsschritte bei der Übersetzung, Optimierung und Ausführung bzw. Interpretation von DB-Anfragen dar (vgl. [3] bzw. [4]). Im Kontext meiner Ausarbeitung möchte ich nur den zweiten Schritt (Optimierung) herausgreifen und seine vier Teilaufgaben (Standardisierung, Vereinfachung, Anfragerestrukturierung und -transformation) erläutern.

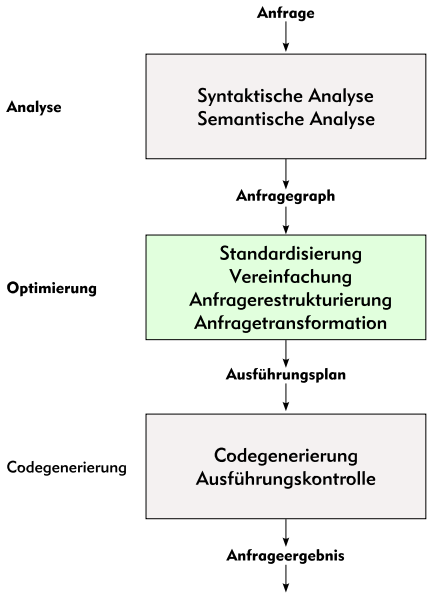

Abbildung 1. Die Optimierung im Kontext der Anfrageverarbeitung

Standardisierung. In diesem Schritt wird das Prädikat der Anfrage in eine Disjunktion von Konjunktionen (*disjunktive Normalform*) oder in eine Konjunktion von Disjunktionen (*konjunktive Normalform*) überführt. Dies wird durch die Anwendung der bekannten Umformungsregeln für Boole'sche Ausdrücke (De Morgan'sche Regeln, Assoziativitäts-, Kommutativitäts-, Distributivitäts- und Doppelnegationsregel) erreicht.

Beispiel. Man betrachte folgendes Prädikat:

```
((Alter <= 24 OR (Beruf = 'Informatiker' OR Beruf = 'Lehrer'))
  AND (Beruf = 'Lehrer' OR NOT(Alter <= 24)))
```

Man zeigt leicht, dass sich dieses Prädikat in die disjunktive Normalform überführen lässt. Als äquivalenten Ausdruck erhält man dann:

```
((Beruf = 'Informatiker' AND NOT(Alter <= 24)) OR Beruf = 'Lehrer')
```

Vereinfachung. Dieser Schritt hat die Aufdeckung von Redundanzen zum Ziel. Dabei wird ein redundanter Ausdruck in eine nicht redundante, aber semantisch äquivalente Form überführt; z. B. indem man die bekannten Idempotenz-Regeln anwendet. Außerdem kann hier überprüft werden, ob das Prädikat überhaupt erfüllbar ist und ob Integritätsbedingungen verletzt werden.

Beispiel. Man betrachte folgendes Prädikat:

```
(ALTER >= 30 OR (Alter >= 30 AND (Beruf = 'Informatiker')))
```

Mit Hilfe der Regeln zur Umformung von Boole'schen Ausdrücken und den bekannten Idempotenz-Regeln kann man dieses Prädikat auf `(ALTER >= 30)` reduzieren.

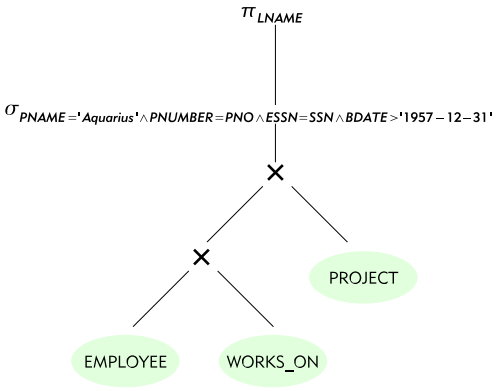

Abbildung 2. Anfragegraph vor der Restrukturierung

Anfragerestrukturierung. In der Phase der Anfragerestrukturierung werden Heuristiken angewendet, um die Anfrage derart umzuformen, dass möglichst optimale Operatorreihenfolgen oder kleine Zwischenergebnisse erzielt werden. Folgende Regeln sind in diesem Zusammenhang relevant [3]:

1. Selektionen (σ) und Projektionen (π) ohne Duplikateliminierung sollen möglichst frühzeitig ausgeführt werden.
2. Folgen von unären Operatoren (wie σ und π) auf einer Relation sind zu einer Operation mit komplexerem Prädikat zusammenzufassen.

3. Selektionen und Projektionen, die eine Relation betreffen, sollen so zusammengefasst werden, dass jedes Tupel nur einmal verarbeitet werden muss.
4. Bei Folgen von binären Operatoren (wie $\cap, \cup, -, \times, \bowtie$) ist eine Minimierung der Größe der Zwischenergebnisse anzustreben.
5. Gleiche Teile im Anfragegraphen sind nur einmal auszuwerten.

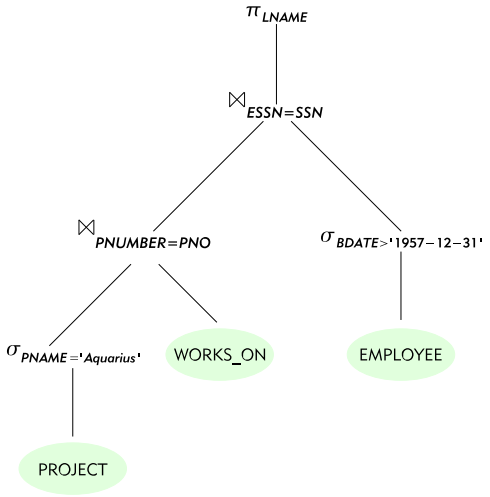

Abbildung 3. Anfragegraph nach der Restrukturierung

Beispiel. Man betrachte folgende SQL-Anfrage (vgl. Navathe und Elmasri [5], S. 515 ff.):

```
SELECT LNAME
FROM   EMPLOYEE, WORKS_ON, PROJECT
WHERE  PNAME = 'Aquarius'      AND
       PNUMBER = PNO           AND
       ESSN = SSN              AND
       BDATE > '1957-12-31';
```

Abbildung 2 zeigt den entsprechenden Anfragegraphen vor der Restrukturierung. Nach Anwendung der Restrukturierungsregeln sieht der Anfragegraph wie in Abbildung 3 aus.

Anfragetransformation. Im letzten Schritt der Optimierungsphase werden die logischen Operatoren zusammengefasst und durch sog. *Planoperatoren* ersetzt. Zu den Planoperatoren, die auf einer Tabelle definiert sind, gehören Selektion, Projektion, Sortierung, Aggregation, Änderungsoperationen sowie Rekursion und Gruppierung. Auf zwei Tabellen lassen sich folgende Planoperatoren definieren: Verbund- und Mengenoperationen sowie das Kartesische Produkt.

In einem weiteren Schritt kann eine Gruppierung von direkt benachbarten Operatoren vorgenommen werden, die Verknüpfungsreihenfolge bei binären Operatoren verändert, und gemeinsame Teilbäume können erkannt werden.

2.2 Kostenmodelle und -abschätzungen

Die Entscheidung über die Auswahl eines physischen Zugriffsplans wird auf Grundlage des Kostenmodells für die Speicherungsstrukturen und Zugriffsoperationen getroffen. Nach Härder und Rahm [3] werden folgende Grundannahmen über die Werteverteilung verwendet:

1. Attributwerte aller Attribute sind gleichverteilt.
2. Die Werte verschiedener Attribute sind voneinander stochastisch unabhängig.

Welche negativen Implikationen diese Annahmen haben, werden wir im Abschnitt 2.3 genauer betrachten. Des Weiteren sind folgende Kostenarten bei der Kostenberechnung zu berücksichtigen [3]:

1. Kommunikationskosten (Anzahl der Nachrichten, Menge der zu übertragenden Daten)
2. Berechnungskosten (Nutzung der CPU, Pfadlänge der Anfrage)
3. E/A-Kosten (Anzahl der physischen Seitenreferenzen)
4. Speicherungskosten (durch temporäre Speicherbelegung im DB-Puffer, in speziellen Arbeitsbereichen und auf Externspeichern).

Die Speicherungskosten werden selten in das Kostenmodell aufgenommen, da sie sich nur schwer mit den Zugriffs- oder Berechnungskosten vergleichen lassen. Zur Berechnung eines Kostenvoranschlages kann man folgende Formel verwenden, die als gewichtetes Maß für die E/A- und CPU-Auslastung dienen soll [3]:

$$C = \#\,\text{der physischen Seitenzugriffe} + W \cdot (\#\,\text{der Aufrufe des Zugriffssystems}) \tag{1}$$

Die Variable W gibt das Verhältnis des Aufwandes von Zugriffssystemaufrufen zu Seitenzugriffen an. Ist das System *„CPU-bound“*, so sollten Ausführungspläne mit höherem E/A- und geringerem CPU-Aufwand bevorzugt werden. Ist das System hingegen *„IO-bound“*, so sollten sich die Größenverhältnisse umkehren.

Kostenmodell. Für die Aufstellung eines Kostenmodells sind folgende statistische Angaben im DB-Katalog zu speichern [3]:

1. *für jedes Segment* S_k
   (a) $M(S_k)$ = Anzahl der Datenseiten des Segmentes S_k
   (b) $L(S_k)$ = Anzahl der leeren Datenseiten von S_k

2. *für jede Relation* R_i *in Segment* S_k
   (a) $N(R_i)$ = Anzahl der Tupel der Relation R_i

   (b) $T(R_i)$ = Anzahl der Seiten von S_k mit Tupeln von R_i
   (c) $C(R_i)$ = Cluster-Faktor (durchschnittliche Anzahl von Tupeln pro Seite)

3. *für jeden Index* $I_{Ri}(A)$ *auf einem Attribut* A *der Relation* R_i
   (a) $j(I_A)$ = Anzahl der Schlüsselwerte im Index I_A
   (b) $B(I_A)$ = Anzahl der Blattseiten der Indexstruktur (B^*-Baum)
   (c) $h(I_A)$ = Höhe des entsprechenden B^*-Baumes.

Für jedes Prädikat p lässt sich ein Selektivitätsfaktor SF berechnen, der angibt, wieviele Tupel höchstwahrscheinlich p erfüllen werden ($\mathrm{Card}(\sigma_p(R)) = SF(p) \cdot \mathrm{Card}(R)$). Bei der Berechnung wird stets eine Gleichverteilung der Werte unterstellt. Die Selektivitätsfaktoren komplexer Ausdrücke lassen sich aufgrund der Annahme der stochastischen Unabhängigkeit sich qualifizierender Mengen folgendermaßen berechnen:

1. $SF(p(A) \wedge p(B)) = SF(p(A)) \cdot SF(p(B))$
2. $SF(p(A) \vee p(B)) = SF(p(A)) + SF(p(B)) - SF(p(A)) \cdot SF(p(B))$
3. $SF(\neg p(A)) = 1 - SF(p(A))$.

Für die Abschätzung der Größe der Ergebnisrelation bei Verbundoperationen wird der sog. Join-Selektivitätsfaktor JSF verwendet. Für den Gleichverbund der Relationen R und S ist er folgendermaßen definiert: $\mathrm{Card}(R \bowtie S) = JSF \cdot \mathrm{Card}(R) \cdot \mathrm{Card}(S)$.
Bei einem verlustfreien (n:1)-Gleichverbund bietet sich folgende Abschätzung an: $\mathrm{Card}(R \bowtie S) = \max(\mathrm{Card}(R), \mathrm{Card}(S))$.

Histogramme. Die zuvor skizzierten Selektivitätsabschätzungen basieren auf den Annahmen der Gleichverteilung und stochastischen Unabhängigkeit. Da diese Annahmen nicht immer zutreffen (vgl. Abschnitt 2.3), sind die Abschätzungen in manchen Fällen extrem ungenau. Um die Güte der Abschätzungen zu erhöhen, werden heute die Attributverteilungen durch Histogramme beschrieben.

Histogramme sind eine Möglichkeit zur Darstellung der Häufigkeitsverteilung von Messwerten. Ein Histogramm über dem Attribut a der Relation R partitioniert den Wertebereich von a in $\beta \geq 1$ disjunkte Teilmengen (sog. *Buckets*).

Es lassen sich verschiedene Varianten von Histogrammen unterscheiden. Für unsere Zwecke eignen sich besonders gut *äquidistante Histogramme*. Dabei wird der Wertebereich so unterteilt, dass jedes Bucket den Inhalt eines gleichgroßen Intervalls widerspiegelt.

2.3 Problematische Annahmen

Im Abschnitt 2.2 wurden einige Annahmen vorgestellt, die bei der Anfrageoptimierung verwendet werden. In diesem Abschnitt sollen diese nun einer kritischen Betrachtung unterzogen werden. Als Grundlage der Kritik dient ein Beispiel, welches aus Härder und Rahm [3] entnommen ist.

Beispiel. Man betrachte folgendes Prädikat:

```
(GEHALT >= 100000 AND (ALTER BETWEEN 21 AND 25))
```

Weiterhin sei der Wertebereich von Gehalt bzw. Alter durch $(0, 10^6)$ bzw. $(16, 65)$ begrenzt. Führt man eine semantische Analyse des Prädikates durch, so lässt sich feststellen, dass es sehr unwahrscheinlich ist, dass die Menge der qualifizierenden Tupel ungleich der leeren Menge ist.

Betrachten wir zunächst die Annahme der Gleichverteilung und der stochastischen Unabhängigkeit von Prädikaten. Man sieht leicht ein, dass es sowohl beim Attribut Alter als auch beim Attribut Gehalt keine Gleichverteilung gibt. In unserer Arbeitswelt ist das Gehalt eng an das Lebensalter gekoppelt. Aus der daraus folgenden Korrelation folgt aber, dass diese beiden Attribute nicht stochastisch unabhängig sein können. Würde man nämlich eine Gleichverteilung unterstellen, so würden sich $(9/10) \cdot (5/50) = 9\%$ der Tupel qualifizieren. Somit würden sich 9/10 der entsprechenden Altersgruppe qualifizieren. Dies entspricht aber nicht im Geringsten der Realität.

Noch nicht erwähnt wurde die implizite Annahme, dass die Statistiken in der Datenbank immer den aktuellen Stand der Datenbank widerspiegeln (vgl. Stillger et al. [6]). In der Praxis wäre es zu teuer, bei jedem DML-Befehl die Statistiken zu aktualisieren. Aus diesem Grund gibt es Werkzeuge (z. B. `runstats` im Falle von DB2), die eine manuelle Aktualisierung der Statistiken durch den Administrator gestatten.

Eine weitere Annahme ist das Prinzip der Inklusion (vgl. Stillger et al. [6]). Dabei wird unterstellt, dass bei einem Verbund dasjenige Attribut mit dem kleineren Wertebereich immer einen Verbundpartner in der Attribut-Menge mit dem größeren Wertebereich findet. Diese Annahme mag zwar bei Primär- und Fremdschlüsselbeziehungen richtig sein, allgemeingültig ist sie jedoch nicht.

3 Adaptive Anfrageoptimierung

Seit mehr als zwanzig Jahren wird bei der Verarbeitung von Datenbankanfragen nach der sog. *plan-first-execute-next*-Methode vorgegangen. Diese besteht aus drei Phasen: Optimierung, Ausführung und Aktualisierung der Statistiken. Die erste Phase wurde bereits in Abschnitt 2 vorgestellt. In der Ausführungsphase wird der kostengünstigste Ausführungsplan ausgewählt und ausgeführt. Anschließend wird die Aktualisierung der Statistiken durch den DB-Administrator manuell durchgeführt. Da eine automatische Aktualisierung der Statistiken nicht stattfindet und der Kostenberechnung nicht allgemeingültige Annahmen zu Grunde liegen (vgl. Abschnitt 2.3), werden möglicherweise suboptimale Ausführungspläne erzeugt und ausgeführt, die ihrerseits die Leistung des DBVS negativ beeinflussen können.

Als Lösungsansatz für dieses Problem möchte ich nun die adaptive Optimierung von Datenbankanfragen vorstellen. In Anlehnung an Hellerstein et al. [7] lässt sich der Begriff eines adaptiven Systems wie folgt definieren:

Definition 1 (Adaptives System). *Ein adaptives System muss allgemein folgende drei Bedingungen erfüllen:*

1. *Das System ist in der Lage Informationen aus seiner Umgebung zu verarbeiten.*
2. *Das System beurteilt und verwendet die aufgenommenen Informationen und kann daraus eigene Entscheidungen ableiten.*
3. *Das System verfügt über einen Feedback-Mechanismus, durch den es im Zeitverlauf über Änderungen der Umgebung informiert werden kann.*

Die statische Anfrageoptimierung[1] erfüllt Definition 1 nicht, da sie über keinen Feedback-Mechanismus verfügt.

Abbildung 4 zeigt das Zusammenspiel des Optimierers mit der Ausführungskontrolle und der Statistik-Einheit. Die Ausführungskontrolle kann eine Reoptimierung der Anfrage veranlassen, wenn die Schätzwerte stark von den tatsächlich erfassten Werten abweichen. Die Verwendung eines Feedback-Mechanismus bei der adaptiven Anfrageoptimierung ermöglicht eine bessere Anpassung der Statistiken an aktuelle Gegebenheiten. Dabei werden kontinuierlich die statistischen Daten des Optimierers aktualisiert. Dadurch wird eine genauere Kostenberechnung ermöglicht, die dann zur Auswahl eines (fast) optimalen Ausführungsplans führt.

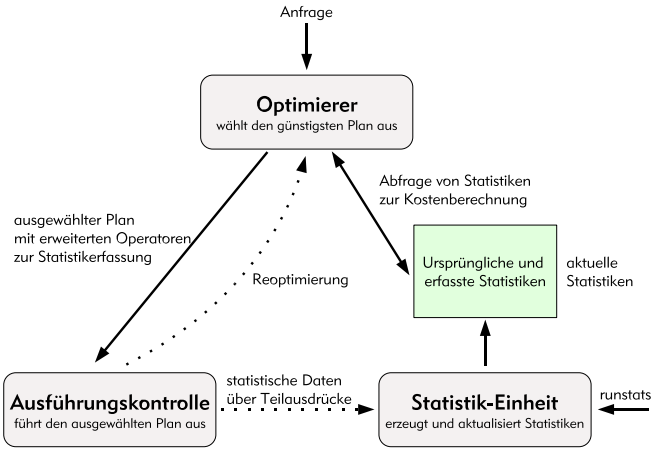

Abbildung 4. Adaptive Anfrageoptimierung (vgl. Babu und Bizzaro [8])

[1] Unter statischer Anfrageoptimierung soll hier die Anfrageoptimierung im klassischen Sinne verstanden werden, bei der ein erzeugter Ausführungsplan im Nachhinein nicht mehr veränderbar ist.

3.1 Adaptive Anfrageoptimierung im historischen Kontext

Das Forschungsgebiet der adaptiven Anfrageoptimierung ist noch relativ jung. Die meisten Publikationen zu diesem Thema lassen sich ab Ende der 90er Jahre des vergangenen Jahrhunderts nachweisen.

Geht man in der historischen Betrachtung noch weiter zurück, so lässt sich das *INGRES*-System von Stonebraker et al. [9] als Vater der Systeme zur adaptiven Anfrageoptimierung nennen [7].

In diesem System wird eine Anfragedekomposition (engl. *query decomposition*) durchgeführt. Dabei alterniert das System zwischen der Auswahl und der Ausführung eines Anfrageplans. Für eine Anfrage, die sich über n Tabellen erstreckt, wird dabei eine adaptive Sequenz von Nested-Loops-Joins durchgeführt. Für jedes Tupel in der kleinsten Tabelle wird in der zweitkleinsten Tabelle nach Verbundpartnern gesucht und das Ergebnis duplikatfrei in der Datenbank materialisiert. Dieses Verfahren wird dann rekursiv auf alle restlichen $n-1$ Tabellen angewendet ($n-2$ Basistabellen und eine Tabelle mit den materialisierten Zwischenergebnissen). Nach der Terminierung der Rekursion wird dieses Verfahren auf die restlichen Tupel der kleinsten Tabelle angewendet. Die Join-Reihenfolge ist bei diesem System nicht statisch. Da die Anzahl der Tupel im materialisierten Zwischenergebnis von Durchlauf zu Durchlauf schwankt und abhängig davon ist, wieviele Verbundpartner in der Probing-Phase des Join-Algorithmus gefunden werden, kann sich die Join-Reihenfolge insgesamt von Tupel zu Tupel der kleinsten Relation ändern. Das Feedback zwischen den einzelnen Durchläufen ermöglicht eine Adaption an die Verbundreihenfolge nachfolgender Relationen. Die Effekte der Adaption bleiben bei diesem System allerdings auf eine einzige Anfrage beschränkt.

3.2 Offene Forschungsprobleme und Lösungsansätze

Volker Markl diskutiert mehrere Forschungsprobleme und mögliche Lösungsansätze, die ihm im Rahmen der Entwicklung des *LEarning Optimizer (LEO)* (vgl. Abschnitt 4) begegnet sind [10]:

1. *Stabilität und Konvergenz.* Ein kardinalitätenbasiertes Modell, das durch den Feedback-Mechanismus verfeinert wird, muss mit unvollständigen Informationen umgehen können. Es lassen sich dabei zwei Kategorien von Kardinalitäten unterscheiden: Es gibt einerseits solche, die aus dem Anfrage-Feedback logisch gefolgert werden können und andererseits Kardinalitäten, die auf Statistiken und Modellen beruhen und somit Unschärfen aufweisen können. Die Lernfähigkeit des Systems hängt in erheblichem Maße von der Last und den getroffenen Annahmen und verwendeten Modellen ab. Zur Lösung dieses Problems sollte der Optimierer Kardinalitäten, die auf harten Fakten und somit sicherem Wissen basieren, denen vorziehen, die auf Statistiken und Annahmen basieren.

2. *Aufdecken von Korrelationen.* In der Praxis treten häufig Korrelationen zwischen Daten auf. Es gibt verschiedene Arten von Korrelationen wie z. B. funktionale Abhängigkeiten zwischen Attributen oder anwendungsspezifische Constraints (vgl. das Beispiel in Abschnitt 2.3). Korrelationen können sich über beliebig viele Attribute einer Relation erstrecken. Für komplexe Anfragen mit mehreren Prädikaten ist es schwer, festzustellen, welche Teilmengen der Prädikate stark korrelieren und welche nicht. Ebenso schwierig ist die Verallgemeinerung von stark korrelierenden Attributwerten zu Abhängigkeiten zwischen Attributen.
   Es stellt sich die Frage, wieviele verschiedene Werte Attribute, die in Prädikaten vorkommen, annehmen müssen, damit man zweifelsfrei eine Abhängigkeit zwischen Attributen daraus folgern kann. Anstatt abzuwarten, bis man genug Anfragen untersucht hat, um Aussagen darüber machen zu können, welche Attribute korrelieren, wäre es besser, wenn das System solche Attribute selbstständig aufspüren und überwachen könnte. Für diese potentiell korrelierenden Attribute könnten dann mehrdimensionale Histogramme verwendet werden, um die Selektivität besser abschätzen zu können.

3. *Zeitpunkt der Reoptimierung.* Die Anzahl der Reoptimierungen pro Anfrage sollte sich auf ein Minimum beschränken. Wird eine Reoptimierung der Anfrage zur Laufzeit durchgeführt, so kann die Erzeugung des neuen Plans sehr teuer werden, wenn das System nicht effizient auf temporäre Zwischenergebnisse zugreifen kann. Der Optimierer kann dies nur während der Reoptimierung herausfinden. Es stellt sich hier allerdings immer die Frage, ob der durch die Reoptimierung erzielte Gewinn größer als die entstandenen Kosten ist. Es kommt weniger darauf an, wie ungenau die geschätzten Werte sind, sondern viel wichtiger ist es, dass die Differenz zwischen den tatsächlichen und geschätzten Werten gering bleibt. Eine Lösung besteht darin, dass das System die Planoperatoren überwacht und entscheidet, ob die Differenz zwischen den tatsächlichen und geschätzten Werten derart hoch ist, dass eine Reoptimierung gerechtfertigt werden kann.

4. *Lernen von anderen Informationen.* Das Lernen muss sich nicht alleine auf Selektivitäten und Kardinalitäten beschränken. Da die physischen E/A-Zugriffe einen erheblichen Anteil an der Höhe der Kosten haben, wäre es sinnvoll, kontinuierlich Feedback über die Anzahl der Treffer im DB-Puffer in den Optimierungsvorgang einfließen zu lassen. Zusätzlich könnte man Informationen über den zur Sortierung verfügbaren Anteil am Hauptspeicher kontinuierlich überwachen und bei zu geringem Speicher den Hauptspeicheranteil beim Sort-Heap derart erhöhen, dass externes Sortieren nicht notwendig wird.

3.3 Selbstwartende Histogramme

Histogramme werden heute in den meisten Datenbankverwaltungssystemen zur Abschätzung von Attribut-Selektivitäten eingesetzt. Während der Anfrageopti-

mierung fallen nur sehr geringe Kosten für den Zugriff auf die Histogramme an. Zusätzlich müssen aber auch die Kosten für die Verwaltung und Aktualisierung, die bei einer Änderung der Daten anfallen, berücksichtigt werden. Da die Kosten dafür aber sehr hoch sind, verzichtet man bei großen Datenbanken darauf, für alle Attribute Histogramme zu erzeugen.

Aboulnaga und Chaudhuri [11] stellen selbstwartende Histogramme (SW-Histogramme, engl. *self-tuning histograms*) als adaptiven Ansatz zur Wartung von Histogrammen vor. Diese SW-Histogramme erhalten ihre Werte nicht direkt aus den Tabellen, sondern werden auf Grundlage des Feedbacks über Selektivitäten aufgebaut, das sie während der Anfrageausführung erhalten.

Sobald Anfragen an die Datenbank gestellt werden, verwendet der Anfrageoptimierer die Histogramme, um die Selektivität der Attribute abzuschätzen. Bei der Anfrageausführung kann das Laufzeitsystem die Anzahl der Tupel zählen, die die einzelnen Operatoren als Output lieferten. Dieses „kostenlose“ Feedback kann dann verwendet werden, um die Histogramme zu verfeinern. Diese Verfeinerung führt dann zu einer fortschreitenden Verringerung des Schätzfehlers und letztendlich zu genaueren Selektivitäten im Histogramm.

Für SW-Histogramme lassen sich zwei Verfeinerungsmodi unterscheiden (vgl. Abbildung 5): Befindet sich das SW-Histogramm im *Online-Modus*, so wird

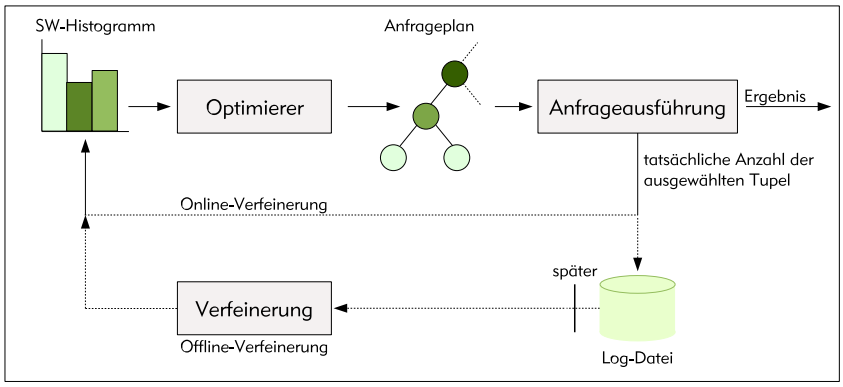

Abbildung 5. Verfeinerung von SW-Histogrammen

es sofort aktualisiert, wenn eine Anfrage ausgeführt wird. Befindet sich das SW-Histogramm im *Offline-Modus*, so wird jede Anfrage, zusammen mit der tatsächlichen Anzahl an Ergebnissen pro Operator, in einer Log-Datei (*Workload Log*) vermerkt. Diese Log-Datei wird dann verwendet, um die Histogramme zu einem späteren Zeitpunkt zu aktualisieren.

Eindimensionale SW-Histogramme. Eindimensionale SW-Histogramme bilden die Grundbausteine für mehrdimensionale SW-Histogramme. Ein SW-Histogramm besteht aus einer Menge von sog. *Buckets*. Jedes Bucket b spei-

chert die zur jeweiligen Klasse gehörenden Werte mit der Hilfe eines Intervalls $[\text{low}(b), \text{high}(b)]$, sowie die Bucket-Häufigkeit (Anzahl der enthaltenen Elemente) $\text{freq}(b)$. Je zwei benachbarte Buckets überlappen sich in ihren Endpunkten. Außerdem decken die Buckets den gesamten Wertebereich des Attributs ab.

Ein SW-Histogramm setzt immer voraus, dass die Daten gleichverteilt sind, solange keine Aktualisierung des Histogramms vorgenommen wurde. Der Lebenszyklus eines SW-Histogramms besteht aus zwei Phasen: Initialisierung und Verfeinerung. Die Verfeinerung kann noch weiter unterteilt werden: Aktualisierung der Bucket-Häufigkeit und Restrukturierung des Histogramms.

Initialisierung. Um ein Histogramm h über dem Attribut a zu erstellen, benötigt man folgende Informationen: die Anzahl B der benötigten Buckets[2], die Anzahl T der Tupel in der Relation und den kleinsten bzw. größten Wert von a (min bzw. max). Da bei der Initialisierung noch kein Feedback vorhanden ist, wird eine Gleichverteilung der Werte unterstellt.

Aktualisierung der Bucket-Häufigkeit. Algorithmus 1 wird zur Aktualisierung der Bucket-Häufigkeit $\text{freq}(b)$ im Bucket b sowohl im Online- als auch im Offline-Modus verwendet. Zuerst werden die Buckets bestimmt, die partiell oder vollständig den Wertebereich [rangelow, rangehigh] beinhalten. Die geschätzte Anzahl *est* der erzeugten Ergebnisse stellt der Anfrageoptimierer während der Optimierung zur Verfügung. Anschließend wird der geschätzte Fehler *esterr* berechnet. Danach wird für jedes Bucket der Fehler-Anteil berechnet, der diesem Bucket zuzurechnen ist. Bei der Berechnung von $\text{freq}(b)$ wird ein Korrekturfaktor α verwendet, der Werte zwischen 0 und 1 annehmen kann und verhindert, dass $\text{freq}(b)$ zu stark verändert wird.

Algorithm 1: *UpdateFreq* [11]

Input: Histogram h, range selection [rangelow, rangehigh], actual result size *act*
Output: h with updated bucket frequencies

1 Get the set of k buckets overlapping the selection range $\{b_1, b_2, \ldots, b_k\}$;
2 est = Estimated result size of selection using histogramm h;
`/* Compute the absolute estimation error. */`
3 $esterr = act - est$;
`/* Distribute the error among the buckets in proportion to frequency. */`
4 **for** $i = 1$ *to* k **do**
5 $\quad \text{frac} = \frac{\min(\text{rangehigh}, \text{high}(b_i)) - \max(\text{rangelow}, \text{low}(b_i)) + 1}{\text{high}(b_i) - \text{low}(b_i) + 1}$;
6 $\quad \text{freq}(b_i) = \max(\text{freq}(b_i) + \alpha \cdot \text{esterr} \cdot \text{frac} \cdot \text{freq}(b_i)/\,\text{est}, 0)$;
7 **end**

[2] Die Anzahl B der benötigten Buckets lässt sich mit Hilfe der Sturgess-Regel berechnen: $B = 1 + 3.3 \cdot \log_{10} n$. Dabei sei n die Anzahl der verschiedenen Werte, die das Attribut a annehmen kann (vgl. [12]).

Restrukturierung des Histogramms. Die Aktualisierung der Bucket-Häufigkeit reicht alleine nicht aus, um die Genauigkeit des Histogramms zu erhöhen, da diese aus der durchschnittlichen Häufigkeit der einzelnen Werte im Bucket berechnet wird. Stattdessen ist zusätzlich eine Restrukturierung notwendig, da manche Werte im Bucket häufiger vorkommen können als andere. Deshalb werden im Restrukturierungsschritt die Buckets mit den höchsten Bucket-Häufigkeiten ausgewählt und auf mehrere Buckets verteilt. Um zu verhindern, dass sich die Anzahl der Buckets nach dem Split-Vorgang insgesamt erhöht, wird zusätzlich eine Verschmelzung von Buckets, die ähnliche Bucket-Häufigkeiten haben, durchgeführt.

Zwei Buckets haben eine ähnliche Bucket-Häufigkeit (engl. *similar frequencies*), wenn die Differenz zwischen der Anzahl der Elemente in den Buckets, weniger als m Prozent der Gesamtzahl T der Tupel in der Relation beträgt. Zusätzlich werden s Prozent der Buckets mit den größten Bucket-Häufigkeiten zum Split ausgewählt.

Algorithmus 2 beschreibt, wie die Restrukturierung durchgeführt wird.

Algorithm 2: *RestructureHist* [11]

Input: Histogram h
Output: restructured h
`/* Find buckets with similar frequencies to merge */`
1 Initialize B runs of buckets such that each contains one histogram bucket;
2 For every two consecutive runs of buckets, find the maximum difference in frequency between a bucket in the first run and a bucket in the second run;
3 Find the minimum of all these maximum differences, mindiff;
4 **if** mindiff $\leq m \cdot T$ **then**
5 Merge the two runs of buckets corresponding to mindiff into one run;
6 Look for other runs to merge. goto line 3;
7 **end**
`/* Assign the extra buckets freed by merging to the high frequency buckets. */`
8 $k = s \cdot B$;
9 Find the set $\{b_1, b_2, \ldots, b_k\}$ of buckets with the k highest frequencies that were not chosen to be merged with other buckets in the merging step;
10 Assign the buckets freed by merging to the buckets of this set in proportion to their frequencies;
`/* Construct the restructured histogram by merging and splitting */`
11 Merge each previously formed run of buckets into one bucket spanning the range represented by all the buckets in the run and having a frequency equal to the sum of their frequencies;
12 Split the k buckets chosen for splitting, giving each one the number of extra buckets assigned to it earlier. The new buckets are evenly spaced in the range spanned by the old bucket and the frequency of the old bucket is equally distributed among them;

Zunächst werden Folgen (engl. *runs*) von benachbarten Buckets mit ähnlichen Bucket-Häufigkeiten bestimmt. Danach wird für je zwei Bucket-Folgen die maximale Differenz der Bucket-Häufigkeiten eines Buckets in der ersten Bucket-Folge und eines Buckets in der zweiten Bucket-Folge berechnet. Das Minimum dieser Differenzen bezeichnen wir mit *mindiff*.

Die beiden Bucket-Folgen mit der minimalen Differenz *mindiff* werden genau dann zu einer Bucket-Folge verschmolzen, wenn *mindiff* kleiner als $(m \cdot T)/100$ ist. Jede Bucket-Folge wird dann durch ein Bucket ersetzt, das den gesamten Wertebereich der Bucket-Folgen beinhaltet. Die Bucket-Häufigkeit ergibt sich dann aus der Gesamtsumme der Bucket-Häufigkeiten des einzelnen Buckets in der Folge. Dieses Verfahren wird so lange angewendet, bis keine Bucket-Folgen mehr verschmolzen werden können.

Die durch die Aufteilung freigesetzten Buckets kann man nun verwenden, um die s Prozent der Buckets mit den höchsten Bucket-Häufigkeiten darauf zu verteilen. Dabei wird ein aufzuteilendes Bucket b_i auf $\text{freq}(b_i)/\,\text{totalfreq}$ Buckets aufgeteilt. Die Variable totalfreq bezeichne hier die Summe der Bucket-Häufigkeiten der aufzuteilenden Buckets.

Mehrdimensionale SW-Histogramme. Mehrdimensionale SW-Histogramme bieten eine Möglichkeit, um die Selektivität von Prädikaten, die konjunktiv verknüpfte Attribute beinhalten, besser abschätzen zu können. Für diese Zwecke können eindimensionale SW-Histogramme nicht verwendet werden, da sie die Korrelationen zwischen den Attributen nicht korrekt widerspiegeln können.

Zunächst wird eine sog. *Grid-Partitionierung* des mehrdimensionalen Raumes vorgenommen. Jede Dimension i eines n-dimensionalen SW-Histogramms wird in B_i Partitionen unterteilt. Dabei kann jede Partition unterschiedlich fein unterteilt werden. Die Partitionierung des Raums wird durch n Arrays beschrieben – eines für jede Dimension. Solche Arrays bezeichnen wir als Skalen (engl. *scales*). Jedes Element einer Skala steht für den Wertebereich einer Partition $[\text{low}, \text{high}]$. Außerdem verfügt ein mehrdimensionales SW-Histogramm über eine n-dimensionale Häufigkeitsmatrix, die die Anzahl der Elemente der einzelnen Grid-Zellen speichert.

Beispiel. Abbildung 6 zeigt ein zweidimensionales SW-Histogramm für die Attribute `Attribut1` und `Attribut2`. Der grün hervorgehobene Bereich zeigt, welche Zellen für das Prädikat `((Attribut1 BETWEEN 8 AND 18) AND (Attribut2 BETWEEN 18 AND 35))` ausgewählt werden, um die Anzahl der enthaltenen Tupel zu berechnen.

Initialisierung des Histogramms. Um ein n-dimensionales SW-Histogramm zu erstellen, werden n eindimensionale SW-Histogramme verwendet. Dabei werden die Skalen durch die Aufteilung des Raums entlang der Bucketgrenzen der eindimensionalen Histogramme aufgebaut. Die Häufigkeitsmatrix wird mit

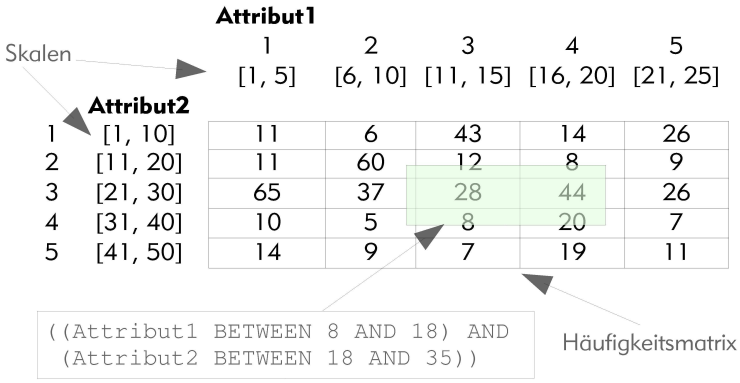

Abbildung 6. Ein zweidimensionales SW-Histogramm

den Bucket-Häufigkeiten der eindimensionalen Histogramme unter der Annahme der stochastischen Unabhängigkeit initialisiert. Die Anzahl der Elemente einer Grid-Zelle ist durch $\text{freq}[j_1, j_2, \ldots, j_n] = \frac{1}{T^{n-1}} \prod_{i=1}^{n} \text{freq}_i\,[j_i]$ gegeben. Dabei bezeichnet $\text{freq}_i\,[j_i]$ die Bucket-Häufigkeit des Buckets j_i im eindimensionalen SW-Histogramm für die Dimension i.

Restrukturierung des Histogramms. Der Algorithmus zur Verfeinerung der Bucketgrößen im mehrdimensionalen Fall ist fast identisch zum Algorithmus für den eindimensionalen Fall. Allerdings muss jetzt ein mehrdimensionaler Raum untersucht werden. Ein Bucket entspricht hier einer mehrdimensionalen Zelle in der Häufigkeitsmatrix.

Da die mehrdimensionalen SW-Histogramme mit Hilfe von eindimensionalen SW-Histogrammen aufgebaut wurden, ist hier keine Restrukturierung notwendig. Dabei unterstellt man, dass die Partitionierung jedes einzelnen eindimensionalen SW-Histogramms auf Grundlage der betrachteten Daten genauer ist, als eine Restrukturierung des gesamten mehrdimensionalen SW-Histogramms durch Neuaufteilung und Verschmelzung der einzelnen Skalen.

3.4 Adaptive Selektivitätsabschätzungen

In Abschnitt 3.3 haben wir gesehen, wie man SW-Histogramme verwenden kann, um die Selektivität von Prädikaten besser abzuschätzen. Neben den histogrammbasierten Verfahren gibt es aber noch weitere Möglichkeiten zur Verwaltung von Selektivitätswerten. So stellen Chen und Roussopoulos [13] die adaptive Selektivitätsabschätzung (*Adaptive Selectivity Estimation, ASE*) als ein Verfahren zur Approximation der Werteverteilung von Attributen vor, das auf Feedback des Laufzeitsystems zurückgreift. Dabei wird vollständig auf die Sammlung von statistischen Daten verzichtet. Für die Approximierung der Werteverteilung wird ein Polynom verwendet, das dann graduell verbessert wird. Dabei „lernt“ das Verfahren die Selektivität nicht nur durch die Analyse vorhergehender Anfragen,

query sequence	1	2	3	4	5
$[l_i, h_i]$	[1935,1966]	[1925,1950]	[1904,1939]	[1890,1923]	[1908,1913]
$\hat{s_i}$	1073	1138	1248	567	2
s_i	1872	1399	890	136	14

6	7	8	9
[1948,1989]	[1957,1980]	[1964,1989]	[1916,1981]
1956	1103	1041	3173
2033	1130	1134	3045

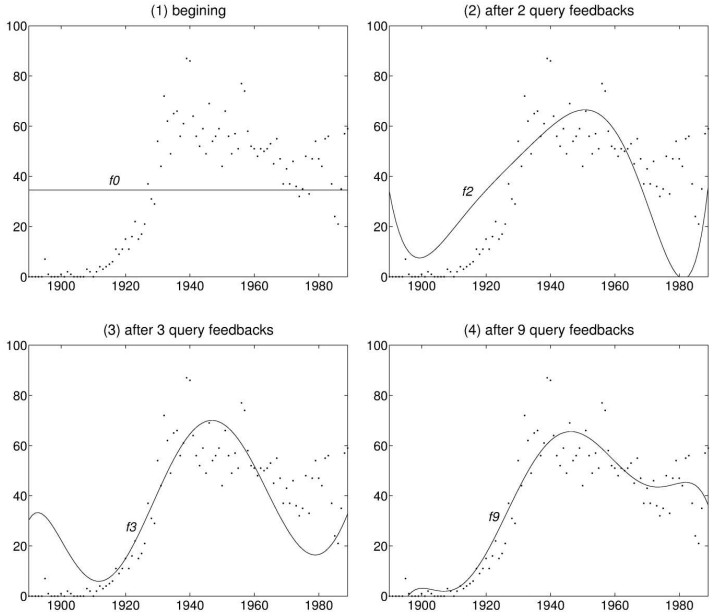

Abbildung 7. Adaptive Selektivitätsabschätzung [13]

sondern kann auch die Selektivität zukünftiger Anfragen voraussagen. Dazu verwenden sie die sog. *recursive-least-square-error*-Technik, die die Werteverteilung an die aktuellen Gegebenheiten anpasst. Das Feedback fließt dabei in gewichteter Form in die Berechnung ein, d. h., dass älterem Feedback ein geringeres Gewicht, verglichen mit neuerem Feedback, zugewiesen wird.

Beispiel. Zur Illustration der Wirkungsweise des Verfahrens diene eine Film-Datenbank mit 3424 Filmen. Abbildung 7 zeigt die Entwicklung der approximierten Werteverteilung für eine Folge von Anfragen. Die Anfragen sind in der Tabelle in Abbildung 7 aufgeführt. Dabei bezeichnet $[l_i, h_i]$ den angefragten Wertebereich der Anfrage i; $\hat{s_i}$ sei dabei die Selektivitätsschätzung vor der Anfrageausführung und s_i sei die tatsächliche Selektivität. Die vier Graphen der Werteverteilung zeigen, wie sich die geschätzte Werteverteilung (durchgezogene

Linie) mit zunehmendem Feedback der tatsächlichen Werteverteilung (diskrete Punkte) annähert.

3.5 Effiziente Reoptimierung während der Anfrageverarbeitung

In Abschnitt 3.3 haben wir gesehen, wie man ein- und mehrdimensionale Histogramme effizient verwalten kann, um diese dann für die kostenbasierte Anfrageoptimierung einzusetzen. Anstatt die Histogramme an die sich ändernde Umgebung anzupassen, kann man auch den Anfrageausführungsplan zur Laufzeit verändern. Kabra und DeWitt [14] stellen hierzu ein Verfahren zur dynamischen Reoptimierung von Datenbankanfragen vor. Dabei handelt es sich um einen Algorithmus, der feststellt, ob ein suboptimaler Anfrageausführungsplan ausgeführt wird. Dabei kann der Plan gleichzeitig reoptimiert werden. Dazu wird der Anfrageausführungsplan bei seiner Erstellung mit verschiedenen Annotationen (Statistiken des Anfrageoptimierers) angereichert. Während der Anfrage-

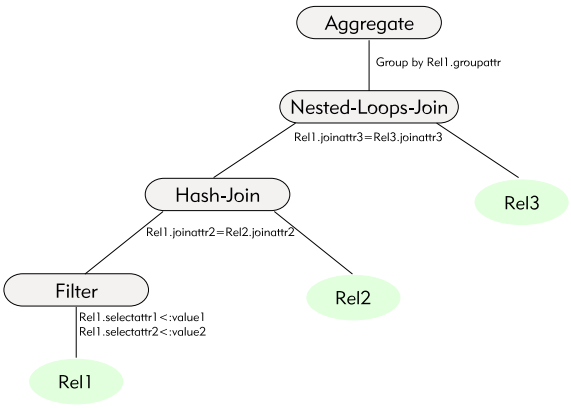

Abbildung 8. Anfrageausführungsplan

ausführung werden dann aktuelle Statistiken aufgezeichnet. Ist die Differenz zwischen den tatsächlichen und geschätzten Werten hinreichend groß, so kann man darauf schließen, dass der Anfrageausführungsplan suboptimal ist. Die aktuellen Statistiken werden dann zur Reoptimierung verwendet. Da die Aufzeichnung dieser Statistiken teuer ist, gibt der Optimierer Hinweise darauf, an welchen Stellen der Anfrage diese Daten aufgezeichnet werden sollen.

Annotation von Anfrageausführungsplänen. In Abschnitt 2.1 haben wir gesehen, dass die Anfrageoptimierung als Resultat einen Anfrageausführungsplan liefert. Dabei handelt es sich im Wesentlichen um eine Baum, der als Knoten die Planoperatoren (z. B. *Hash-Join*) enthält. Abbildung 8 zeigt einen Anfrageausführungsplan zu folgender Anfrage:

```
SELECT   AVG(Rel1.selectattr1),
         AVG(Rel1.selectattr2),
         Rel1.groupattr
FROM     Rel1, Rel2, Rel2
WHERE    Rel1.selectattr1 < :value1       AND
         Rel1.selectattr2 < :value2       AND
         Rel1.joinattr2 = Rel2.joinattr2  AND
         Rel1.joinattr3 = Rel3.joinattr3
GROUP BY Rel1.groupattr;
```

Im Rahmen der dynamischen Reoptimierung wird der Anfrageoptimierer derart verändert, dass er den Ausführungsplan mit statistischen Daten und Schätzwerten anreichern kann (engl. *annotated query execution plan*, annotierter Anfrageausführungsplan). Bei den statistischen Daten handelt es sich um Informationen über die Kardinalitäten von Zwischenergebnissen, Selektivitäten von Selektions- und Joinprädikaten oder die Anzahl der zu bildenden Gruppen bei Aggregationsoperatoren.

Sammeln von statistischen Daten zur Laufzeit. Abbildung 9 zeigt den Ausführungsplan für die Anfrage aus Abbildung 8, der um den *Statistics-Collector*-Operator ergänzt wurde. Dieser wurde nach dem *Filter*-Operator eingefügt, der Selektionen auf der Relation *Rel1* durchführt. Sobald der Filter-Operator Tupel als Ausgabe liefert, können diese vom Statistics-Collector-Operator aufgezeichnet und analysiert werden, ohne den normalen Anfrageausführungsvorgang zu unterbrechen. So kann z. B. die Kardinalität des Ergebnisses, das durch den Filter-Operator erzeugt wird, berechnet werden, indem der Statistics-Collector-Operator die eingehenden Tupel zählt.

Als Einschränkung dieses Verfahrens könnte man die Tatsache anführen, dass die zu sammelnden Statistiken durch eine einmalige Verarbeitung der Eingabe extrahiert werden müssen. In der Praxis stellt das aber kein Problem dar, da die notwendigen Berechnungen wie z. B. der Kardinalitäten oder der durchschnittlichen Tupelanzahl in einem Durchlauf vorgenommen werden können.

Eine tatsächliche Einschränkung dieses Verfahrens hat mit der Beschaffenheit und Platzierung des Statistics-Collector-Operators zu tun. Betrachtet man eine Folge von nicht blockierenden Planoperatoren, die den Statistics-Collector-Operator umschließen, so kann zur Laufzeit keiner der Planoperatoren von den gesammelten Statistiken profitieren, solange nicht alle Tupel verarbeitet wurden, da der Statistics-Collector-Operator lediglich in der Lage ist, die Anzahl der verarbeiteten Tupel zu zählen. Die gewöhnlichen Statistiken aus dem System-Katalog unterscheiden sich erheblich von denen, die zur Laufzeit durch den Statistics-Collector-Operator gesammelt werden. Die Statistiken im System-Katalog werden gewöhnlich nur einmal berechnet, sind eher von allgemeiner Natur und werden für verschiedene Zwecke eingesetzt. Da der Statistics-Collector-Operator aber vom Optimierer an einer bestimmten Stelle in den Ausführungsplan eingefügt wird, kann er auf die jeweilige Situation zugeschnittene Daten liefern.

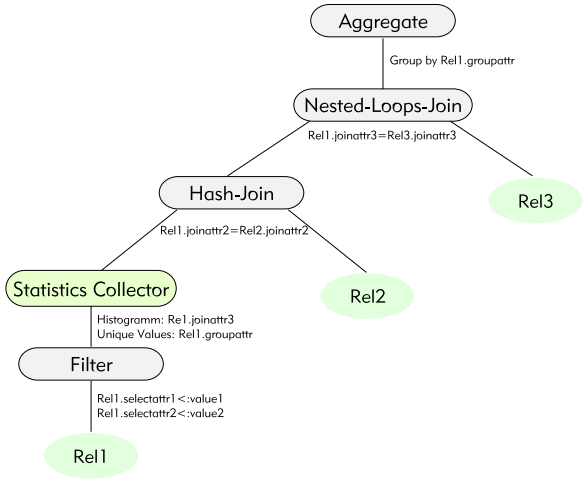

Abbildung 9. Sammeln von statistischen Daten zur Laufzeit

Dynamische Ressourcen-Reallokation. Kabra und DeWitt [14] weisen darauf hin, dass die meisten relationalen Planoperatoren einen großen Teil des Hauptspeichers in Anspruch nehmen, um ihre Aufgabe schnell lösen zu können. Da bei komplexen Anfragen viele solcher Operatoren verwendet werden, stellt sich die Frage, wie der Speicher auf die Anfragen und zugehörigen Operatoren aufgeteilt werden kann. Entweder wird diese Entscheidung vom Anfrageoptimierer während der Optimierung getroffen oder von der Anfrageausführungskontrolle während der Anfrageausführung. In beiden Fällen wird der Speicherbedarf mit Hilfe von statistischen Daten abgeschätzt. Wird zu wenig Speicher allokiert, so werden zusätzliche E/A-Zugriffe notwendig. Dagegen führt die Allokation von zu viel Speicher dazu, dass anderen Anfragen bzw. Operatoren nicht ausreichend Speicher zur Verfügung steht. Wird der Optimierer zum ersten Mal ausgeführt, so verwendet er zur Bestimmung des Speicherbedarfs Standardwerte aus dem System-Katalog. Sobald genauere Schätzungen des Speicherbedarfs zur Verfügung stehen, kann zur Laufzeit eine Reallokation des Speichers für den Rest der Anfrage vorgenommen werden. Aus diesem Grund werden bei der zukünftigen Bestimmung des Speicherbedarfs die genaueren Statistiken des Statistics-Collector-Operators denen des System-Katalogs vorgezogen.

Anpassung des Anfrageausführungsplans. Betrachtet man erneut den Anfrageausführungsplan aus Abbildung 9 und unterstellt man, dass der Schätzwert für die Größe der Ausgabe des Filter-Operators fehlerhaft ist, so kann dies bedeuten, dass der gesamte Ausführungsplan suboptimal ist. Da die neuen Statistiken erst dann verfügbar sind, wenn der Filter-Operator seine Verarbeitung beendet hat, kann der *Hash-Join*-Operator bereits seine Arbeit aufgenommen haben. Unterstellt man, dass der Hash-Join-Operator die Probing-Phase noch

nicht beendet hat und noch keine Ausgabe produziert hat, so lassen sich drei verschiedene Wege unterscheiden, die der Reoptimierungsalgorithmus an dieser Stelle einschlagen könnte: Um dieses Problem zu lösen, wäre es am einfachsten

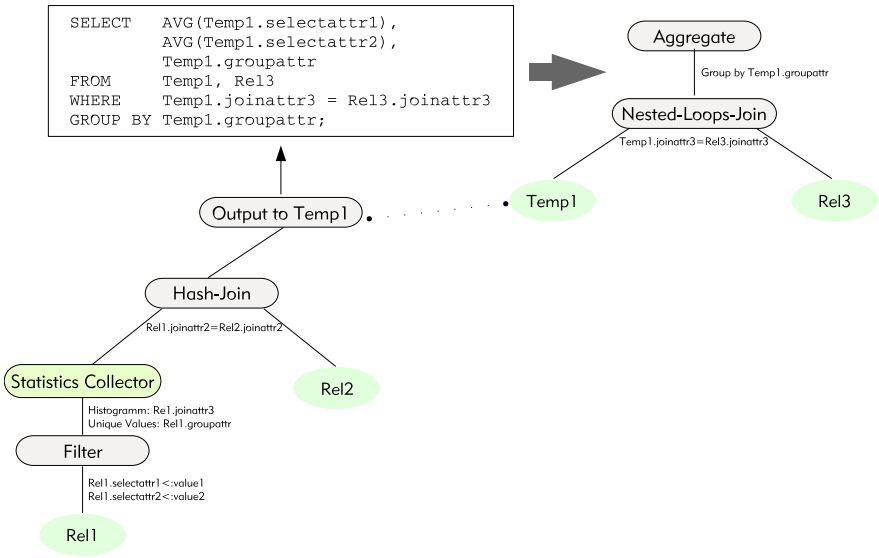

Abbildung 10. Reoptimierung durch temporäre Speicherung von Zwischenergebnissen

den gesamten Anfrageplan zu verwerfen und anschließend die Anfrage mit Hilfe der neuen statistischen Daten zu reoptimieren. Diese Vorgehensweise hat den großen Nachteil, dass ein großer Teil der bereits verrichteten Arbeit verworfen würde. Die Verwendung dieses Ansatzes ist nur dann sinnvoll, wenn die Suboptimalität in einer sehr frühen Phase der Anfrageausführung entdeckt wird.

Als zweite Alternative kann man die Unterbrechung der Anfrageausführung betrachten. Dabei werden nur diejenigen Teile des Ausführungsplans reoptimiert, die noch nicht ausgeführt wurden. In unserem Beispiel unterstellen wir, dass der Filter-Operator bereits seine Arbeit beendet hat und der Hash-Join-Operator seine Arbeit aufgenommen hat. Des Weiteren wird unterstellt, dass der nachfolgende *Nested-Loops-Join*- und der *Aggregate*-Operator ihre Arbeit noch nicht begonnen haben. Deshalb kann der Anfrageausführungsplan derart verändert werden, dass die bereits durchgeführte Arbeit nicht wiederholt werden muss. Dabei wird der Nested-Loops-Join-Operator vom Anfrageoptimierer durch einen Hash-Join-Operator ersetzt.

Die dritte Alternative stellt eine Verbesserung der zweiten dar. Dabei wird die Anfrageausführung nicht mehr unterbrochen, sondern es wird abgewartet, bis alle Operatoren, die zu diesem Zeitpunkt ihre Arbeit begonnen haben, die Verarbeitung abgeschlossen haben (dazu zählt insbesondere der Hash-Join-Operator). Anstatt nun die Ergebnisse an den nachfolgenden Planoperator (Nested-Loops-Join-Operator) weiterzureichen, werden die, vom Hash-Join-Operator erzeugten Tupel in der Datenbank temporär zwischengespeichert. Danach wird eine neue SQL-Anfrage generiert, die den noch nicht ausgeführten Teil der ursprünglichen Anfrage mit dem materialisierten Zwischenergebnis verbindet (vgl. Abbildung 10). Diese SQL-Anfrage wird dann, wie in Abschnitt 2.1 dargestellt, verarbeitet und schließlich zur Ausführung gebracht.

Zeitpunkt der Reoptimierung. Durch die Reoptimierung einer Anfrage entstehen zusätzliche Kosten. Dabei handelt es sich um die Kosten, die durch die erneute Verarbeitung der Anfrage entstehen. Zusätzlich sind – im Fall der dritten Alternative zur Anpassung – die Kosten für die Speicherung und das Lesen des materialisierten Zwischenergebnisses zu berücksichtigen. Aufgrund dieser Zusatzkosten sollte die Reoptimierung nicht bei jeder Änderung der Statistiken durchgeführt werden. Stattdessen lässt sich mit Heuristiken entscheiden, wann eine Reoptimierung sinnvoll ist.

Sei $T_{\text{cur\_plan}}$ der Schätzwert des Optimierers für die Ausführungsdauer des aktuellen Plans und $T_{\text{cur\_plan,improved}}$ sei der verbesserte Schätzwert für den aktuellen Plan. Des Weiteren sei $T_{\text{new\_plan,total}}$ die gesamte Ausführungsdauer (bestehend aus der Dauer für die bereits abgeschlossene Verarbeitung, der Dauer der Optimierung der Anfrage, der Dauer für die Materialisierung der Zwischenergebnisse sowie der Dauer für die Ausführung der restlichen Anfrage) des neuen Anfrageausführungsplans. Die Reoptimierung lohnt sich nur dann, wenn die Ungleichung $T_{\text{cur\_plan, improved}} > T_{\text{new\_plan, total}}$ erfüllt ist.

Weiterhin sei $T_{\text{opt,estimated}}$ ein konservativer Schätzwert für $T_{\text{new\_plan, total}}$. Die Dauer der Reoptimierung hängt nicht von der Größe der Datenbank ab, sondern vielmehr von der Anzahl der verwendeten Operatoren.

Ob es sich wirklich lohnt, $T_{\text{opt,estimated}}$ Zeiteinheiten für die Reoptimierung aufzuwenden, lässt sich mit Hilfe einiger Heuristiken feststellen. In Gleichung 2 nimmt der Parameter θ_1 nur sehr kleine Werte wie z. B. 0.05 an. Solange Gleichung 2 erfüllt ist, wird keine Reoptimierung durchgeführt.

$$\frac{T_{\text{opt, estimated}}}{T_{\text{cur\_plan, improved}}} > \theta_1 \tag{2}$$

Des Weiteren ist eine Reoptimierung nicht notwendig, solange der aktuelle Plan nicht suboptimal ist. In Gleichung 3 dient die Differenz von $T_{\text{cur\_plan}}$ und $T_{\text{cur\_plan, improved}}$ als Indikator dafür, ob der aktuelle Plan suboptimal ist. Somit wird eine Reoptimierung genau dann durchgeführt, wenn Gleichung 3 erfüllt ist. Der Parameter θ_2 wird mit dem Wert 0.2 initialisiert.

$$\left| \frac{T_{\text{cur\_plan, improved}} - T_{\text{cur\_plan}}}{T_{\text{cur\_plan}}} \right| > \theta_2 \tag{3}$$

4 LEO – der lernende Optimierer

In diesem Abschnitt möchte ich den *LEarning Optimizer* (LEO) von Stillger et al. [6] vorstellen. Dieses System, das bereits in die *DB2 Universal Database* der Firma IBM integriert wurde, ist in der Lage, falsche Statistiken und Kardinalitätsschätzungen während der Anfrageausführung zu erkennen und zu korrigieren. Dazu vergleicht LEO in jedem Schritt des Anfrageausführungsplans die Schätzwerte des Optimierers mit den aktuellen Werten. Stellt das System fest, dass sich die geschätzten Werte stark von den tatsächlichen Werten unterscheiden, so werden die vom Optimierer verwendeten Werte für zukünftige Optimierungsvorgänge angepasst. Zusätzlich verfügt LEO über einen Feedback-Mechanismus (*feedback loop*), der es dem System ermöglicht, aus früheren Fehlern zu lernen. Es handelt sich dabei um eine Erweiterung und Verallgemeinerung des Ansatzes von Aboulnaga und Chaudhuri [11] (vgl. Abschnitt 3.3). LEO kann aus jedem Fehler an jeder Stelle im Modell lernen. Dazu zählen insbesondere Fehler bei der Selektivitätsabschätzung von Attributen, von benutzerdefinierten Funktionen, von Prädikaten mit Host-Variablen sowie von Join-Prädikaten. Zusätzlich kann LEO auch Informationen über die Pufferauslastung, den Speicherverbrauch beim Sort-Heap und das E/A-Verhalten liefern. Bisher ist LEO noch nicht in der Lage, einen Ausführungsplan zur Laufzeit zu verändern, wie es z. B. das Verfahren von Kabra und DeWitt [14] gestattet (vgl. Abschnitt 3.5). Stattdessen sind alle korrigierten Statistiken erst für die Optimierung nachfolgender Anfragen verfügbar.

4.1 Die Architektur von LEO

LEO verwendet statistische Daten, die er während der Ausführung von Anfragen gesammelt hat, um inkrementell das Kostenmodell des Optimierers zu validieren. Dadurch kann er feststellen, welche Schätzwerte im Modell falsch sind und diese korrigieren. LEO besteht aus vier Komponenten: einer Komponente, die den Ausführungsplan speichert, einer Komponente zur Aufzeichnung der statistischen Daten, einer Komponente zur Analyse der Daten sowie einer Komponente zur Erzeugung des Feedbacks. Die Analyse-Komponente muss nicht zwingend auf dem selben Rechner ausgeführt werden, auf dem sich auch der DB2-Server befindet. Die anderen drei Komponenten ergänzen die Funktionalität des DB2-Servers.

Abbildung 11 zeigt, wie LEO in die Architektur von DB2 eingebunden ist. Die linke Hälfte der Abbildung zeigt die klassischen Komponenten zur Anfrageverarbeitung (SQL-Compiler, Optimierer, Code-Generator und das Laufzeitsystem). Die grau hervorgehobenen Komponenten stellen die Erweiterung der Architektur dar.

Für jede übersetzte Anfrage schreibt der Code-Generator einen rudimentären Anfrageausführungsplan (engl. *plan „skeleton“*) in eine zusätzliche Datei (*QEP Skeleton File*), die dann später von der Analyse-Komponente (*LEO Analysis Daemon*) verwendet werden kann. Auf ähnliche Art und Weise stellt das Laufzeitsystem der Analyse-Komponente kontinuierlich Informationen über die

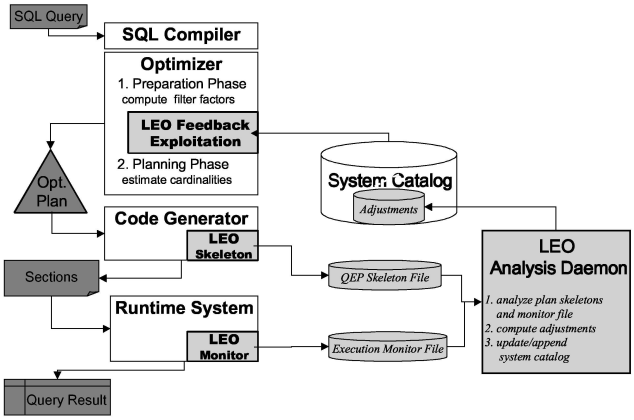

Abbildung 11. Die Architektur von LEO [6]

Kardinalitäten der einzelnen Operatoren zur Verfügung. Dies ermöglicht der Analyse-Komponente die zur Optimierung verwendeten statistischen Daten anzupassen und im System-Katalog abzulegen.

Man betrachte folgende SQL-Anfrage:

```
SELECT   *
FROM     X, Y, Z
WHERE    X.Price >= 100    AND
         Z.City = 'Denver' AND
         Y.Month = 'Dec'   AND
         X.ID = Y.ID       AND
         Y.NR = Z.NR
GROUP BY A;
```

Abbildung 12 zeigt den zugehörigen rudimentären Anfrageausführungsplan, der mit statistischen Informationen angereichert wurde. Dabei bezeichnet *Stat* die Anzahl der Tupel in der Relation, wie sie im System-Katalog gespeichert ist. Die Variable *Est* bezeichnet den Schätzwert des Optimierers für die Kardinalität des Ergebnisses, das vom jeweiligen Planoperator erzeugt wird. Schließlich wird mit *Act* die tatsächliche Kardinalität des Ergebnisses bezeichnet, das der Planoperator zur Laufzeit geliefert hat.

Der Vergleich zwischen den geschätzten und tatsächlichen Kardinalitäten ermöglicht LEO bei zu starker Abweichung, sein Kostenmodell anzupassen. Zum Beispiel ist in Abbildung 12 die tatsächliche Kardinalität beim *Index-Scan* über der Relation Y fast dreimal so hoch wie bei der Optimierung angenommen.

Während der Entwicklung des LEO mussten einige Design-Entscheidungen getroffen werden. Dazu zählt auch die Entscheidung, dass LEO nie die Original-Statistiken im System-Katalog verändert. Stattdessen werden zusätzliche Statistiken eingeführt, die zur Anpassung der ursprünglichen dienen. Dabei werden bei der Übersetzung einer Anfrage sowohl die ursprünglichen als auch die

neuen Statistiken gelesen. Bei Bedarf (z. B. bei großen Abweichungen zwischen den Daten) werden dann die Statistiken neu berechnet. Durch dieses Verfahren hat man immer die Möglichkeit, das Lernen zu deaktivieren, was besonders für Debugging-Zwecke wichtig ist. Außerdem können zu jedem Plan die angepassten Schätzwerte gespeichert werden, so dass man immer herausfinden kann, wo sich Änderungen ergeben haben.

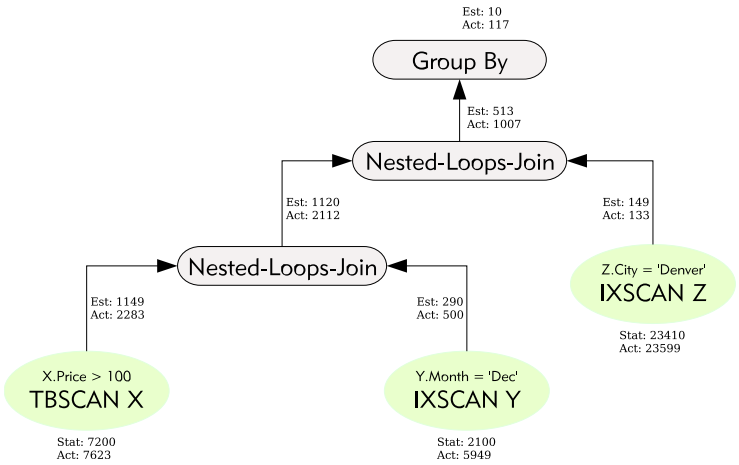

Abbildung 12. Rudimentärer Anfrageausführungsplan mit zusätzlichen statistischen Informationen (vgl. [6])

DB2 speichert Statistiken für Basisrelationen, Spalten, Indizes usw., die gegenseitig voneinander abhängig sind. Deshalb muss LEO immer die Konsistenz dieser Abhängigkeit sicherstellen. Wird z. B. die Selektivität eines Attributs verändert, auf dem ein Index definiert ist, so muss auch entsprechend die Statistik des Indexes angepasst werden.

Das Sammeln von statistischen Informationen ist sehr kostspielig, da ein vollständiger Scan einer einzelnen Relation oder sogar der gesamten Datenbank notwendig ist. Deshalb sind genaue statistische Informationen oft nicht vorhanden. Stattdessen werden vom Optimierer Standardwerte verwendet. Somit stellt sich Frage, ob der Optimierer möglicherweise veralteten, aber angepassten Daten mehr Glauben schenken sollte als den unscharfen Standardwerten. Durch eine Aktualisierung der Statistiken werden viele Anpassungen ungültig. Die Menge der angepassten Werte enthält als Teilmenge die Menge der von `runstats` zur Verfügung gestellten Werte und enthält darüberhinaus zusätzliche statistische Daten. Darum würde die Verwerfung aller Anpassungen zu Informationsverlust führen. Deshalb muss jede Aktualisierung auch die angepassten Werte korrekt aktualisieren.

Das LEO-System ist nicht als Ersatz für die bisher verwendeten Statistiken gedacht, sondern eher als komplementärer Ansatz anzusehen. LEO erweitert

`runstats` z. B. um Möglichkeiten zur besseren Abschätzung von Selektivitäten bei Join-Operatoren. Im Zeitverlauf werden die Schätzwerte des Optimierers besonders für die Relationen verbessert, die am häufigsten angefragt werden. Für die Kostenabschätzung neuer Anfragen kann LEO aber nicht auf die Daten von `runstats` verzichten.

4.2 Der Feedback-Mechanismus von LEO

In der Übersetzungsphase erzeugt der Code-Generator ein ausführbares Programm (Sektion, engl. *Section*) aus dem optimalen Anfrageausführungsplan. Diese Sektion kann dann entweder direkt ausgeführt werden oder in der Datenbank gespeichert werden, um zu einem späteren Zeitpunkt ausgeführt zu werden. Zur Laufzeit der Sektion steht der optimale Anfrageausführungsplan nicht mehr zur Verfügung.

Eine Sektion besteht aus Sequenzen von Operatoren (sog. *threads*), die zur Laufzeit interpretiert werden. Deshalb wäre es im Prinzip möglich, daraus den Anfrageausführungsplan wieder zu rekonstruieren. Da dieses Vorgehen aber in der Praxis zu teuer wäre, wird ein anderer Weg eingeschlagen: Während der Übersetzung einer Anfrage wird ein rudimentärer Anfrageausführungsplan in der Datenbank gespeichert, der dann als Grundlage für die weitere Analyse dient (vgl. Abbildung 12). Zur Erfassung der tatsächlichen Anzahl der verarbeiteten Tupel, wird jeder Planoperator in der Sektion mit einem Zähler versehen. Dieser Zähler wird immer dann inkrementiert, wenn der Planoperator ein Tupel verarbeitet hat.

Um für eine Anfrage die zur Laufzeit aufgezeichneten Statistiken mit den Schätzwerten des Optimierers vergleichen zu können, muss LEO zunächst den zugehörigen rudimentären Anfrageausführungsplan finden. Bei einem Treffer werden dann die Statistiken im rudimentären Ausführungsplan mit Hilfe der tatsächlichen Werte angepasst.

Berechnung der Anpassungen. Durch den Vergleich der tatsächlichen Selektivität eines Prädikates mit den geschätzten Werten, die im rudimentären Anfrageausführungsplan gespeichert wurden, kann LEO einen Anpassungsfaktor *adj* (engl. *adjustment factor*) ableiten, der später dazu verwendet werden kann, um den korrekten Selektivitätsfaktor zu berechnen. Dieser Anpassungsfaktor wird dann in der Datenbank gespeichert. Der aktuelle Selektivitätswert ergibt sich aus dem Produkt des Anpassungsfaktors und der geschätzten Selektivität. Die Variable old_est bezeichne die geschätzte Selektivität, old_adj sei der alte Anpassungsfaktor, der zur Berechnung von old_est verwendet wurde. Außerdem sei act die aktuelle Selektivität, die aus den aufgezeichneten Daten abgeleitet wurde. Mit hoher Wahrscheinlichkeit liegt ein Fehler vor, wenn folgende Bedingung erfüllt ist:

$$\frac{|\,\text{old\_est} - \text{act}\,|}{\text{act}} > 0.05 \tag{4}$$

Liegt ein Fehler z. B. für das Prädikat `(col < X)` vor, so lässt sich der Anpassungsfaktor folgendermaßen bestimmen: $\text{adj} = (\text{act} \cdot \text{old\_adj}) / \text{old\_est}$. So-

mit lässt sich auch leicht die Selektivität für das Prädikat `(col >= X)` durch $1-$*Selektivität*(`(col < X)`) bestimmen.

Beispiel. Man betrachte den *TBSCAN*-Operator in Abbildung 12 mit dem Prädikat `(Price > 100)`. Für dieses Prädikat erhält man für die Kardinalität der Relation den Anpassungsfaktor $7632/7200 = 1.06$. Die Selektivität des Prädikates wurde hierbei vom Optimierer auf $1149/7200 = 0.1595$ geschätzt. Tatsächlich war der Selektivitätswert aber $2283/7632 = 0.2994$. Somit ergibt sich für das Prädikat `(Price < 100)` ein Anpassungsfaktor von $(1-0.2994)/(1-0.1595) = 0.8335$. Daraus folgt der Anpassungsfaktor für das Prädikat `(Price > 100)`: $1-(0.835\cdot(1-0.1595)) = 0.2994$. Der Optimierer berechnet nun in Zukunft für das Prädikat `(Price > 100)` den korrekten Wert $1.06 \cdot 7200 \cdot 0.2994 \approx 2283$.

Verwendung des gelernten Wissens. Bevor der Optimierer beginnt, verschiedene Anfragepläne aufzubauen, sucht er zunächst im DB-Schema nach allen referenzierten Tabellen und den zugehörigen Statistiken. Aus diesen Statistiken entnimmt der Optimierer dann die Kardinalität der jeweiligen Relation und berechnet für jedes Prädikat den Selektivitätsfaktor. Wurde der Feedback-Mechanismus aktiviert, so kann der Optimierer zusätzlich die Anpassungsfaktoren aus dem DB-Katalog entnehmen und kann mit deren Hilfe die Tabellenstatistiken und Selektivitäten entsprechend korrigieren.

Anpassung der Kardinalitäten von Basistabellen. Die Kardinalitäten der Basistabellen werden zuerst angepasst, da sie die Grundlage aller weiteren Abschätzungen bilden. Dies wird durch die Multiplikation der Kardinalitäten mit dem jeweiligen Anpassungsfaktor erreicht. Die Schwierigkeit besteht in der Wahrung der Konsistenz zwischen den korrigierten Kardinalitäten und den anderen Statistiken der Tabellen. Da während der Ausführung von `runstats` die Anzahl *NPAGES* der physischen Seiten, die die Tabelle belegt, bestimmt wird und im Kostenmodell zur Berechnung der Anzahl der E/A-Operationen für den

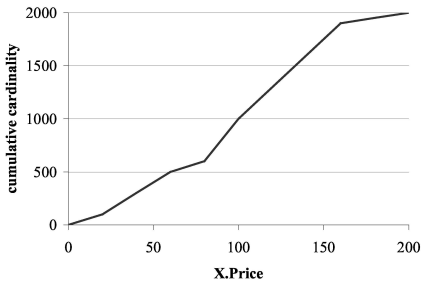

Abbildung 13. aktuelle Werteverteilung für das Attribut `X.Price` (vgl. [6])

TBSCAN-Operator verwendet wird, muss LEO den *NPAGES*-Wert anpassen,

wenn die Kardinalität der Tabelle angepasst wird. Außerdem darf die Kardinalität eines Attributs nicht größer sein als die der gesamten Tabelle. Aber die Erhöhung der Tabellenkardinalität kann die Erhöhung von Attributkardinalitäten zur Folge haben.

Beispiel. Das Einfügen von Zeilen in eine Tabelle mit Personaldaten führt nicht

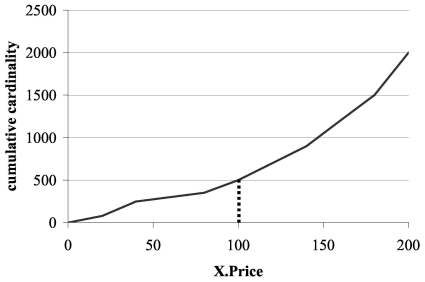

Abbildung 14. gesammelte Statistiken für das Attribut `X.Price` (vgl. [6])

zur Erhöhung der Kardinalität beim Attribut *Geschlecht*. Durchaus möglich ist aber eine Erhöhung der Kardinalität für das Attribut *Personalnummer*.

Anpassung der Selektivität bei einfachen Prädikaten. Man betrachte die Anpassung der Selektivität für das einfache Prädikat `(X.Price < 100)`. Abbildung 13 zeigt die aktuelle Werteverteilung für das Attribut `X.Price`. Abbildung 14 zeigt die vom Laufzeitsystem aufgezeichnete Werteverteilung für das Attribut `X.Price`. Für die Selektivität des Prädikates `(X.Price < 100)` berechnet der Optimierer den Wert: cardinality($X < 100$) / Maximal_Cardinality = $500/2000 = 0.25$. Korrigiert man diesen Wert mit Hilfe des Anpassungsfaktors für das Attribut (vgl. Abbildung 15), so erhält man für die Selektivität den Wert $0.25 \cdot 2 = 0.5$.

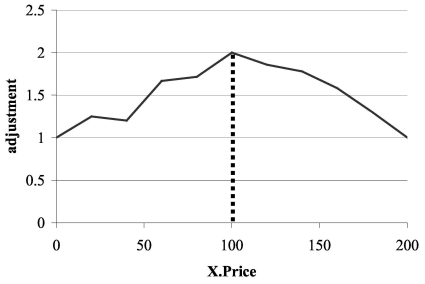

Abbildung 15. Werte des Anpassungsfaktors für das Attribut `X.Price` (vgl. [6])

5 Fazit und Ausblick

In dieser Ausarbeitung habe ich mich mit der adaptiven Optimierung von Datenbankanfragen beschäftigt. Zu Beginn habe ich die Grundlagen der Anfrageoptimierung vorgestellt. Dabei habe ich insbesondere auf die problematischen Annahmen hingewiesen, die der statischen Anfrageoptimierung zu Grunde liegen. Wir haben dann gesehen, wie man Histogramme zur effizienten Verwaltung von Statistiken verwenden kann. Im Rahmen der effizienten Reoptimierung von Datenbankanfragen habe ich einen Feedback-Mechanismus vorgestellt, der den Anfrageoptimierer kontinuierlich über die sich ändernde Systemumgebung informieren kann. Der danach vorgestellte Learning Optimizer (LEO) von IBM verwendet dann diesen Feedback-Mechanismus, um aus Fehlern bei der Anfrageoptimierung zu lernen und um diese mit dessen Hilfe zu korrigieren.

Das Forschungsgebiet der adaptiven Anfrageoptimierung ist noch relativ jung. Bisher war der Fokus auf die Verbesserung der Adaptionsfähigkeit einzelner Datenbankverwaltungssysteme gerichtet. Unter dem Schlagwort der *Enterprise Application Integration, (EAI)* wird in Zukunft auch die Adaptionsfähigkeit von föderierten Datenbankverwaltungssystemen betrachtet werden. Ewen et al. (vgl. [15] bzw. [16]) erweitern LEO derart, dass er auch im Kontext von föderierten Datenbankverwaltungssystemen eingesetzt werden kann. Dabei zeigen sie, wie man die Statistiken aus verschiedenen heterogenen Datenquellen aufzeichnen und für die Optimierung von föderierten Anfrageplänen verwenden kann.

Literatur

1. Jarke, M., Koch, J.: *Query Optimization in Database Systems.* ACM Computing Surveys **16**(2) (1984) 111–152
2. Ibaraki, T., Kameda, T.: *On the Optimal Nesting Order for Computing n-Relational Joins.* ACM Transactions on Database Systems **9**(3) (1984) 482–502
3. Härder, T., Rahm, E.: *Datenbanksysteme – Konzepte und Techniken der Implementierung.* Springer-Verlag Heidelberg (2001)
4. Mitschang, B.: *Anfrageverarbeitung in Datenbanksystemen: Entwurfs- und Implementierungskonzepte. Reihe Datenbanksysteme.* Vieweg-Verlag (1995)
5. Elmasri, R., Navathe, S.B.: *Fundamentals of Database Systems, Fourth Edition.* Addison-Wesley, Boston (2004)
6. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: *LEO - DB2's LEarning Optimizer.* In: VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11–14, 2001, Roma, Italy, Morgan Kaufmann Publishers (2001) 19–28
7. Hellerstein, J.M., Franklin, M.J., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V., Shah, M.A.: *Adaptive Query Processing: Technology in Evolution.* Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **23**(2) (2000) 7–18
8. Babu, S., Bizarro, P.: *Adaptive Query Processing in the Looking Glass.* In: CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, January 4-7, 2005 Asilomar, CA, USA. (2005) 238–249
9. Stonebraker, M., Held, G., Wong, E., Kreps, P.: *The Design and Implementation of INGRES.* ACM Transactions on Database Systems **1**(3) (1976) 189–222

10. Markl, V.: *LEO: An autonomic query optimizer for DB2 – LEarning Optimizer.* http://www.findarticles.com/p/articles/mi_m0ISJ/is_1_42/ai_98695291/print – zuletzt abgerufen am 9. September 2005 um 11:30 Uhr. (2005)
11. Aboulnaga, A., Chaudhuri, S.: *Self-tuning Histograms: Building Histograms Without Looking at Data.* In Delis, A., Faloutsos, C., Ghandeharizadeh, S., eds.: SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1–3, 1999, Philadelphia, Pennsylvania, USA, ACM Press (1999) 181–192
12. Wikipedia: *Histogramm.* http://de.wikipedia.org/wiki/Histogramm – zuletzt abgerufen am 4. Dezember 2005 um 11:50 Uhr. (2005)
13. Chen, C.M., Roussopoulos, N.: *Adaptive Selectivity Estimation Using Query Feedback.* In Snodgrass, R., Winslett, M., eds.: SIGMOD 1994, Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, May 24–27, 1994, Minneapolis, Minnesota, USA, ACM Press (1994) 161–172
14. Kabra, N., DeWitt, D.: *Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans.* In Haas, L., Tiwary, A., eds.: SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2–4, 1998, Seattle, Washington, USA, ACM Press (1998) 106–117
15. Ewen, S., Ortega-Binderberger, M., Markl, V.: *A Learning Optimizer for a Federated Database Management System.* Informatik – Forschung und Entwicklung **20**(3) (2005) 138–151
16. Ewen, S., Ortega-Binderberger, M., Markl, V.: *A Learning Optimizer for a Federated DBMS.* http://btw2005.aifb.uni-karlsruhe.de/Folien/S022Ewen.pdf – zuletzt abgerufen am 16. Dezember 2005 um 14:36 Uhr. (2005) BTW '05: 11. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web, 2. – 4. März 2005 Karlsruhe, Deutschland.