Effiziente Generierung und Ausführung von DAG-strukturierten Anfragegraphen

Thomas Neumann

Universität Mannheim
tneumann@pi3.informatik.uni-mannheim.de

Abstract: Datenbanksysteme verwenden traditionell baumstrukturierte Pläne für die Anfragebearbeitung. Einige Optimierungstechniken wie Faktorisierung lassen sich mit Bäumen aber nicht gut formulieren. Eine attraktive Möglichkeit, die Pläne ausdrucksstärker zu machen, ist die Verallgemeinerung von Bäumen zu gerichteten azyklischen Graphen (DAGs).

Existierende Ansätze betrachten DAGs nur in Spezialfällen, sie sind nicht voll in die Anfrageoptimierung integriert. Der hier vorgestellte Plangenerator ist der erste, der generisch optimale DAG-strukturierte Pläne erzeugt. Die experimentellen Ergebnisse zeigen, dass die so erzeugten Pläne teilweise deutlich effizienter sind.

1 Einleitung

Datenbanksysteme erlauben mit Hilfe von deklarativen Anfragesprachen auf die vorhandenen Daten zuzugreifen. Im Gegensatz zu typischen Programmiersprachen geben deklarative Anfragesprachen wie z.B. SQL nur an, welche Daten berechnet werden sollen, nicht aber, wie die Berechnung genau ablaufen soll. Dadurch kann das Datenbanksystem selbst bestimmen, wie die Berechnung durchgeführt werden soll und dabei Eigenschaften des Systems und der Daten ausnutzen.

Dazu wird die vom Benutzer gestellte Anfrage zunächst in eine andere Darstellung überführt, bei relationalen Datenbanksystemen meistens in einen Ausdruck der relationalen Algebra. Anschließend bestimmt der Anfrageoptimierer den eigentlichen Ausführungsplan, im einfachsten Fall ebenfalls in relationaler Algebra. Der Algebraausdruck hat eine Baumstruktur, die Teilausdrücke überlappen sich nicht. Das hat zur Folge, dass u.U. Berechnungen mehrfach durchgeführt werden müssen, weil Wiederverwendung von Zwischenergebnissen bei einer Baumstruktur nicht möglich ist. Ein Beispiel dafür ist die SQL-Anfrage und der zugehörige Baum in Abbildung 1: Hier sollen alle Kunden bestimmt werden, die überdurchschnittlich viel umgesetzt haben. Dazu werden die Kunden mit ihren Bestellungen verbunden, die Summe der Bestellungen berechnet und das Ergebnis mit dem Durchschnitt verglichen. Die Durchschnittsberechnung selbst erfordert aber den gleichen Verbund. Das Zwischenergebnis wird also zweimal berechnet. Wenn man den Ausdruck von einem Baum in einen gerichteten azyklischen Graph verallgemeinert (direct acyclic graph, DAG), können Zwischenergebnisse wiederverwendet werden (rechte Seite von Abbildung 1). Der Verbund muss nur einmal berechnet werden.

Dies ist die einfachste Form der DAG-Erzeugung und wird auch von kommerziellen Da-

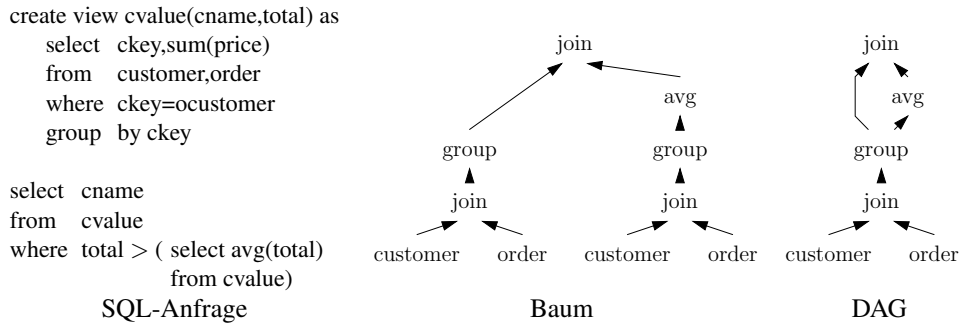


Abbildung 1: Umsetzung einer SQL-Anfrage

tenbanksystemen in einigen Fällen (z.B. für Sichten) verwendet [GLSW93]. Allerdings werden dabei DAGs immer nur als Sonderfall behandelt und sind nicht voll in die Optimierung integriert. I.A. kann es aber durchaus besser sein, Berechnungen zweimal durchzuführen, wenn dadurch Bedingungen früher ausgewertet werden können. Die Entscheidung darüber sollte der Optimierer treffen. Weiterhin materialisieren existierende Systeme die wiederverwendeten Zwischenergebnisse, was relativ teuer ist. Schließlich erlauben DAGs mehr als nur ein Wiederverwenden von Zwischenergebnissen. Wir werden in Abschnitt 6 ein Beispiel dafür sehen. Der hier vorgestellte Ansatz gestattet die Erzeugung von optimalen DAG-strukturierten Plänen für beliebige Ausdrücke und effiziente Ausführung solcher Pläne. Dadurch können einige Klassen von häufig vorkommenden Anfragen deutlich schneller beantwortet werden.

Der Rest des Beitrags ist wie folgt aufgebaut: Abschnitt 2 gibt eine kurze Einführung in Anfrageoptimierung, Abschnitt 3 erläutert dann die prinzipiellen Probleme bei der Erzeugung von DAG-strukturierten Plänen. Abschnitt 4 enthält den eigentlichen Algorithmus, mit dem DAG-strukturierte Pläne erzeugt werden können. Die Ausführung dieser Pläne wird dann in Abschnitt 5 diskutiert. Die Bedeutung dieser Techniken für konkrete Anfragen wird in Abschnitt 6 untersucht. Schließlich werden verwandte Arbeiten in Abschnitt 7 betrachtet und Abschnitt 8 gibt eine Zusammenfassung der Ergebnisse.

2 Anfragebearbeitung mit Bäumen

Wie bereits in der Einleitung erwähnt wird eine Anfrage vom Datenbanksystem für die Ausführung in einen Algebraausdruck überführt. Die Operatoren der (relationalen) Algebra sind dabei zum einen die normalen Mengenoperationen ($\cup$, $\cap$), zum anderen auch einige spezielle Operatoren wie Selektion (σ), Verbund ($\bowtie$), Umbenennung (ρ) und Gruppierung (Γ) und die Datenrelationen selbst. Diese Operatoren sind die Bausteine, aus denen ein Ausführungsplan zusammengesetzt wird. Die Konstruktion ist dabei im Prinzip einfach. Eine SQL-Anfrage kann z.B. übersetzt werden, indem alle beteiligten Relationen verbunden werden, anschließend die Bedingungen mit Selektionen überprüft werden und das Ergebnis ggf. gruppiert wird. Diese „kanonische" Übersetzung erzeugt allerdings extrem ineffiziente Pläne:

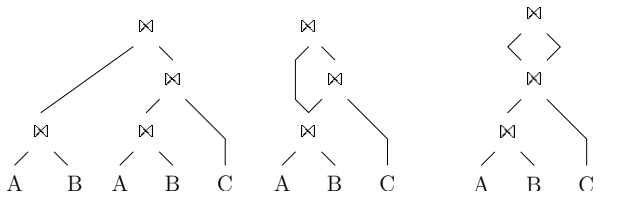


Abbildung 2: Ungültige Transformation von DAGs

Für die meisten Anfragen gibt es mehr als eine Möglichkeit, sie als Algebraausdruck zu formulieren. Beim Verbund dreier Relationen sind z.B. aufgrund der Assoziativität des Verbunds mehrere Klammerungen möglich. Diese Alternativen haben häufig drastisch unterschiedliche Laufzeiten, u.a. weil sie unterschiedlich große Zwischenergebnisse produzieren. Aus diesem Grund verwenden die meisten Datenbanksystem Anfrageoptimierer, die einen möglichst günstigen Ausführungsplan für eine Anfrage bestimmen. Die Optimierung basiert auf algebraischen Äquivalenzen, die bekannte Eigenschaften der verschiedenen Operatoren formalisieren. Das Spektrum reicht dabei von einfachen Äquivalenzen wie z.B. der Assoziativität bis zu komplexen Äquivalenzen wie z.B. der Entschachtelung von abhängigen Anfragen. Ein besonders interessantes Problem ist hierbei die Bestimmung der optimalen Reihenfolge von Verbundoperatoren. Die Reihenfolge hat große Auswirkungen auf die Laufzeit, lässt sich aber leicht formalisieren, da Verbundoperatoren frei anordenbar sind. Das bedeutet, dass jede syntaktisch korrekte Anordnung der Verbundoperatoren das korrekte Ergebnis erzeugt, was die Optimierung vereinfacht. Im nächsten Abschnitt werden wir sehen, dass diese Eigenschaft für DAGs so nicht mehr gilt.

3 DAG-strukturierte Pläne

3.1 Äquivalenzen

Die Anfrageoptimierung basiert im Wesentlichen auf algebraischen Äquivalenzen. Für baumstrukturierte Pläne sind sehr viele Äquivalenzen bekannt (z.B. [GMUW99]). Für DAGs sind diese Äquivalenzen allerdings nicht direkt anwendbar, wie das nachfolgende Beispiel zeigt: Wie bereits erwähnt kann bei Bäumen die optimale Reihenfolge von Verbundoperatoren bestimmt werden, indem einfach alle syntaktisch zulässigen Kombinationen ausprobiert werden. Abbildung 2 zeigt, dass das für DAGs nicht gilt. Die Transformation von a) nach b) ist zulässig, hier wird einfach das Zwischenergebnis wiederverwendet. Die Transformation zu c) ist hingegen nicht zulässig, hier wird ein anderes Ergebnis erzeugt. Intuitiv ist das einleuchtend (der Verbund mit C wird zweimal durchgeführt), ähnliche Umformungen können allerdings durchaus zulässig sein, z.B. Selektionen können häufig mehrfach angewandt werden ohne das Ergebnis zu verändern. Der Anfrageoptimierer benötigt deshalb ein formales Kriterium für die Zulässigkeit von solchen Transformationen.

Die normalen Äquivalenzen für Bäume lassen sich nicht direkt für DAGs anwenden, da

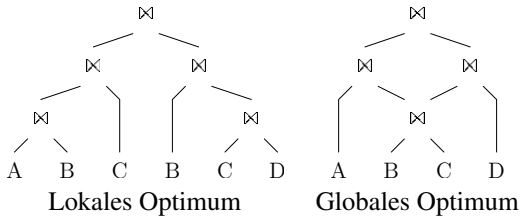


Abbildung 3: Möglicherweise nicht optimale Substruktur für DAGs

sich die Teilpläne überlappen können. Diese Beobachtung führt zu einer überraschenden Lösung für die Konstruktion von DAGs: Operatoren dürfen keine (*logischen*) DAGs erzeugen. Logischer DAG bedeutet hier, dass die gleichen logischen Operatoren in unterschiedlichen Zweigen auftreten. Zulässig sind nur *physische* DAGs, bei denen Operatoren zwar mehrfach gelesen werden, sie aber auch mehrfach in der Anfrage auftauchen. Als Konsequenz daraus können Operatoren nur durch Umbenennung wiederverwendet werden: Wenn ein Operator mehr als einen Konsumenten hat, müssen bis auf einen alle Umbenennungsoperatoren sein. Dabei werden die Umbenennungen (ρ) dazu verwendet, den Plan als Baum zu interpretieren (was er logisch gesehen auch ist) und die DAG-Struktur zu verstecken. Dadurch können Äquivalenzen für Bäume wiederverwendet werden.

Diese Beobachtung führt zu einem Äquivalenzbegriff für die Wiederverwendung von Teilplänen. Wir definieren zwei Algebraausdrücke als *share äquivalent*, wenn ein Ausdruck durch den anderen Ausdruck mit anschließendem Umbenennen berechnet werden kann. Formal definieren wir

$$A \equiv_S B \text{ genau dann wenn } \exists_{\delta_{A,B}:\mathcal{A}(A)\rightarrow\mathcal{A}(B)\, bijektiv}\, \rho_{\delta_{A,B}}(A) = B.$$

wobei mit $\mathcal{A}(A)$ alle Attribute im Ergebnis von A bezeichnet werden.

3.2 Optimale Substruktur

Optimierungstechniken wie Dynamische Programmierung und Memoization basieren auf der optimalen Substruktur eines Problems. Das heißt, dass die optimale Lösung eines Problems aus der optimalen Lösung von Teilproblemen konstruiert werden kann. Diese Eigenschaft gilt für Bäume, allerdings nicht für DAGs. Abbildung 3 zeigt zwei Pläne für $A \bowtie B \bowtie C \bowtie B \bowtie C \bowtie D$. Der linke Plan wurde von unten nach oben unter der Annahme der optimalen Substruktur konstruiert. Dabei wurde $(A \bowtie B) \bowtie C$ als optimale Lösung des Teilproblems $A \bowtie B \bowtie C$ bestimmt. Eine andere Klammerung – wie im rechten Plan – erlaubt aber das Wiederverwenden des Verbunds $B \bowtie C$, was insgesamt zu einem besseren Plan führt. Deshalb können DAGs nicht einfach aus optimalen Teillösungen zusammengesetzt werden. Der Algorithmus im nächsten Abschnitt vermeidet das Problem, indem er explizit mögliche Wiederverwendungen berechnet und diese bei der Dominanz von Plänen berücksichtigt.

4 Algorithmen

Der optimale Ausführungsplan wird in einem Datenbanksystem vom Plangenerator erzeugt, der den Kern des Anfrageoptimierers darstellt. Dazu wird der Raum der möglichen Pläne geeignet durchsucht. Wir verwenden ein verfeinernde Suche mit Memoization. Dabei unterscheidet sich die Erzeugung von DAGs zunächst einmal nicht stark von der von Bäumen. Der einzige Unterschied ist, dass sich Teilprobleme überlappen können. Die gängigen Suchstrategien könnten leicht daran angepasst werden; wie in Abschnitt 3 erläutert reicht das allerdings nicht. Eine einfache Ausweitung des Suchraums würde suboptimale oder sogar inkorrekte Pläne erzeugen. Hier wird deshalb während der Suche die Äquivalenzrelation $\equiv_S$ bei der Konstruktion und Auswahl der Pläne berücksichtigt. Weiterhin ist bei DAGs die Erkennung gleicher Teilprobleme sehr wichtig, da die Zwischenergebnisse wiederverwendet werden könnten. Dieses Problem wird hier durch eine geschickte Darstellung des Suchraums gelöst. Aus Platzgründen kann hier darauf nicht genau eingegangen werden. Prinzipiell wird der Suchraum als Menge von abstrakten binären Eigenschaften formuliert, so dass auch stark unterschiedliche Pläne mit gleicher Ausgabe die gleichen Eigenschaften aufweisen können.

Um die Suche durch Äquivalenztests nicht zu verlangsamen, führt der Plangenerator einige Vorausberechnungen durch. Zunächst einmal wird der Äquivalenzbegriff $\equiv_S$ für Operatoren definiert. Wir betrachten zwei Operatoren als share äquivalent, wenn sie bei äquivalenter Eingabe äquivalente Ausgabe erzeugen. Für Verbundoperatoren z.B. kann die Äquivalenz wie folgt definiert werden:

$$\bowtie_1 \equiv_S \bowtie_2 :\prec\succ \forall_{A_1,A_2,B_1,B_2} A_1 \equiv_S A_2 \wedge B_1 \equiv_S B_2 \Rightarrow (A_1 \bowtie_1 B_1) \equiv_S (A_2 \bowtie_2 B_2)$$

Diese Äquivalenz wird anschließend verwendet, um die an der Anfrage beteiligten Operatoren in Äquivalenzklassen einzuteilen. In jeder Äquivalenzklasse wird ein Repräsentant festgelegt. Da alle Operatoren in einer Klasse äquivalent bezüglich Wiederverwendung sind kann man ohne Einschränkung der möglichen Lösungen festlegen, dass nur das Ergebnis solcher Repräsentanten mehrfach verwendet wird. Damit lässt sich die optimale Substruktur des Problems wieder herstellen: Ein Plan dominiert einen anderen Plan nur dann, wenn er günstiger ist und mindestens die gleichen Repräsentanten enthält wie der andere Plan. Die Repräsentanten markieren damit die Möglichkeiten zur Wiederverwendung.

Die Suche selbst nutzt ebenfalls die Äquivalenzklassen: Wiederverwenden darf nur nach Umbenennungen stattfinden. Deshalb prüft der Plangenerator während der Suche, ob sich ein Teilproblem nur durch Äquivalenzklassenrepräsentanten formulieren lässt. Wenn ja, kann es in dieser Formulierung gelöst und das Ergebnis umbenannt werden, wodurch automatisch DAGs entstehen. Der schematische Ablauf der Suche wird Abbildung 4 dargestellt. Der Plangenerator versucht zunächst, ein Problem durch die Repräsentanten zu lösen und das Ergebnis wiederzuverwenden. Da sich das nicht lohnen könnte, wird auf jeden Fall auch versucht, das Problem direkt zu lösen. Dazu werden alle Operatoren untersucht, die etwas zu dem aktuellen Problem beitragen könnten, und das jeweils verbleibende Teilproblem rekursiv gelöst. Der skizzierte Pseudocode betrachtet nur unäre Operatoren; für binäre Operatoren muss das Teilproblem noch partitioniert werden. Das Entfernen von dominierten Plänen wird hier implizit in der Mengenvereinigung durchgeführt. Dabei werden

```
plangen(goal)
    plans← memoizationTable[goal]
    wenn plans undefiniert ist
        plans← ∅
        shared← goal mit Äquivalenzklassen-Repräsentanten formuliert
        wenn shared∩goal=∅
            plans← plangen(shared)
        ∀ r ∈ {relevante Operatoren}
            wenn r goal erzeugen könnte
                plans← plans∪{r(p)|p∈ plangen(goal\ r.produced)}
        memoizationTable[goal]←plans
    liefere plans zurück
```

Abbildung 4: Struktur des Suchalgorithmus

wie oben erläutert die Repräsentanten berücksichtigt. Insgesamt kann die Suchstrategie durch die Vorausberechnung relativ kompakt formuliert werden.

5 Ausführung

5.1 Strategien

Ein überraschend schwieriges Problem bei DAG-strukturierten Plänen ist die eigentliche Ausführung. Bei Bäumen wird die Ausführung mit Hilfe des Iterator-Modells implementiert, d.h. jeder Operator erzeugt auf Anfrage jeweils das nächste Tupel und jeder Operator bekommt seine Eingabe indem er seine direkten Kinder danach fragt. Dieses Modell funktioniert für DAGs nicht, da ein Operator mehr als einen Elternoperator haben kann. Die Operatoren würden sich gegenseitig die Tupel wegnehmen. Existierende Systeme sind in der Regel nur für Bäume ausgelegt, einige erzeugen aber für Sonderfälle auch DAGs. Abbildung 5 vergleicht einige Strategien, wie DAGs trotzdem ausgeführt werden können mit der hier vorgeschlagenen Strategie, die eine effiziente Ausführung beliebiger DAGs erlaubt. Ausgehend vom Originalplan auf der linken Seite skizzieren wir im Nachfolgenden die verschiedenen Strategien, von einfachen zu komplexen.

Der einfachste Ansatz ist die Umwandlung des DAGs in einen Baum durch Duplizierung (zweite Spalte). Das ist allerdings nicht wirklich eine Option: Die Strategie eliminiert alle Vorteile von DAGs. Ein in der Praxis häufig verwendeter Ansatz ist der Einsatz des Temp-Operators (dritte Spalte). Er materialisiert ein Zwischenergebnis und erlaubt anschließend mehreren Operatoren unabhängig voneinander die Daten zu lesen. Dadurch können DAGs ausgeführt werden. Die Materialisierung ist allerdings relativ teuer, u.U. werden dadurch die Vorteile von DAGs wieder eliminiert. Eine attraktivere Variante ist die Ausnutzung von materialisierenden Operatoren (vierte Spalte). Viele Operatoren wie z.B. Sortierung und Gruppierung materialisieren Daten von sich aus, so dass mit relativ wenig Aufwand mehrere Leser unterstützt werden könnten. Das funktioniert allerdings nicht für alle Operatoren und einige sehr teure Operatoren wie z.B. der abhängige Verbund materialisieren

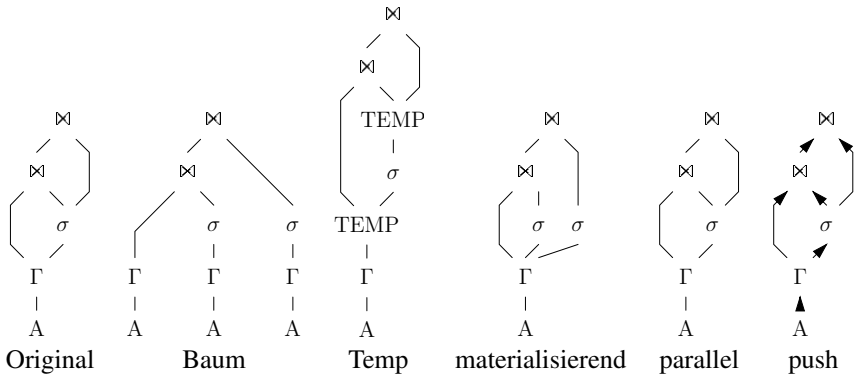


Abbildung 5: Ausführungsstregien für DAGs

nicht, so dass hier wieder auf Temp zurückgegriffen werden müsste.

Die bisherigen Strategien bauen alle auf der Ausführung von Bäumen auf und erweitern sie für DAGs. Eine völlig andere Strategie ist die parallele Ausführung von Operatoren (fünfte Spalte). Hier sind alle Operatoren gleichzeitig aktiv und die Daten können mit einem einfachen Rendezvous-Protokoll zwischen den Operatoren weitergegeben werden. Ähnliche Techniken wurden tatsächlich für verteilte oder parallele Datenbanken verwendet. Für normale Datenbanken ist dieses Ausführungsmodell aufgrund des hohen Verwaltungsaufwands aber relativ ineffizient.

Wir schlagen statt dessen eine Strategie vor, bei der die Daten von unten noch oben durch die Operatoren geschoben werden (letzte Spalte). Dadurch, dass Tupel nicht mehr abgeholt sondern explizit an die Konsumenten weitergereicht werden, können Operatoren sich nicht gegenseitig die Tupel wegnehmen. Tatsächlich erlaubt dieses Modell eine beliebige Anzahl von Konsumenten ohne jede Materialisierung, so dass DAGs keine zusätzlichen Kosten verursachen. Nachfolgend skizzieren wir eine Implementierung dieser Strategie.

5.2 Push-Strategie

Die Push-Strategie Tupel zu den Konsumenten zu bringen statt sie abholen zu lassen ist sehr effizient, da Daten nicht umkopiert werden müssen. Ein Operator erzeugt einmal ein Ausgabetupel und benachrichtigt dann alle Konsumenten, ähnlich dem Rendezvous-Protokoll bei paralleler Ausführung. Da hier allerdings die Operatoren nicht parallel ausgeführt werden, verursacht die Strategie Probleme mit dem Kontrollfluss. Im Iterator-Modell werden Tupel einfach über Funktionsaufrufe geholt. Hier aber werden Tupel nicht geholt sondern weitergereicht. Eine Kette von Funktionsaufrufen markiert also einen Weg von unten noch oben im Ausführungsplan. Wenn jetzt ein Operator Daten von einem weiteren Operator braucht, müsste er die aufrufenden Funktionen kontrollieren. Das ist in den allermeisten Programmiersprachen unmöglich, und selbst bei den Programmiersprachen, die eine ähnliche Funktionalität bieten, ist ein solcher Wechsel teuer. Dieses Problem kann durch ein explizites Steuerprogramm gelöst werden, das in den Kontrollfluss eingreift und

jeweils den richtigen Operator aktiviert. Dazu verwaltet es einen Abhängigkeitsgraph, so dass immer alle Voraussetzungen für eine Aktivierung erfüllt sind.

Diese explizite Steuerung ist allerdings sehr aufwendig, so dass damit die Ausführung von DAGs deutlich langsamer wäre als die von Bäumen. In aller Regel ist ein Wechsel des Kontrollflusses aber gar nicht notwendig, da die meiste Zeit Daten zwischen den gleichen Operatoren weitergereicht werden. Deshalb verwendet unsere Implementierung eine optimistische Strategie: Solange der Kontrollfluss gleich bleibt werden Daten mit einfachen Funktionsaufrufen weitergereicht. Nur wenn das nicht mehr möglich ist fällt der Kontrollfluss zum Steuerprogramm zurück, das dann den nächsten Operator bestimmt. Durch diese Technik kann der Aufwand stark reduziert werden, so dass die Unterstützung von DAGs nur minimale Mehrkosten verursacht.

6 Evaluation

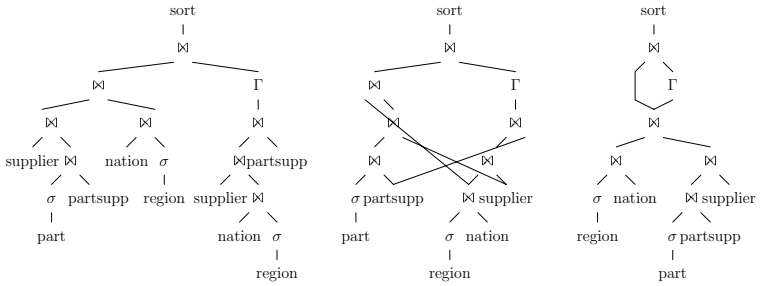
Die Unterstützung von DAG-strukturierten Plänen erfordert einige Änderungen in einem Datenbanksystem. Deshalb müssen sich DAGs lohnen, wenn sie unterstützt werden sollen. D.h. häufig vorkommende Anfragen müssen deutlich von DAGs profitieren. Hier stellen wir einige Anfragen vor und vergleichen deren Ausführung als Baum und als DAG. Ein optimaler DAG-Plan wird natürlich nie schlechter sein als ein optimaler Baum-Plan (jeder Baum ist ein DAG), aber die Erzeugung ist aufwendiger und muss deshalb bei der Laufzeit berücksichtigt werden. Alle Experimente wurden auf einem 2.2 GHz Athlon64 unter Windows XP durchgeführt.

Der TPC-H Benchmark ist ein Standard-Benchmark für relationale Datenbanksysteme der eine betriebswirtschaftliche Anwendung simuliert. Die Datenbankgröße bei Skalierungsfaktor 1 beträgt 1GB. Viele Anfragen des Benchmarks profitieren von DAGs, wir stellen hier zwei typische vor.

Anfrage 11 ist eine typische Anfrage, die durch das Wiederverwenden von Zwischenergebnissen von DAGs profitiert. Sie berechnet die wichtigsten Lagerbestände eines Landes. Dazu wird zunächst der Gesamtbestand berechnet und dann mit den Einzelbeständen verglichen. Der notwendige Verbund zwischen Lieferant und Lagerbestand wird dabei zweimal berechnet, was mit DAGs vermieden werden kann: Bei fast identischer Übersetzungszeit (10.5ms bzw. 10.6ms) kann der Einsatz von DAGs die Laufzeit von 4793ms auf 2436ms fast halbieren.

Anfrage 2 bestimmt den Zulieferer mit den minimalen Kosten innerhalb einer Region. Prinzipiell ist sie ähnlich zu Anfrage 11: Auch hier muss zweimal aggregiert werden, allerdings sind beide Aggregationen nicht völlig identisch. Deshalb ist der Gewinn durch einfaches Wiederverwenden gering. Allerdings kann man hier weitergehende Techniken anwenden, die an Magic Sets [MFPR90] angelehnt sind. Dadurch können wieder größere Teile wiederverwendet werden. Diese Magic Set-Transformationen sind rechenintensiv, wie die Zahlen in Abbildung 6 zeigen, aber die höhere Optimierungszeit fällt gegenüber der drastisch gesunkenen Laufzeit nicht ins Gewicht. Dies ist ein Beispiel für eine Anfrage, die von echten DAG-Techniken über Wiederverwenden hinaus profitiert.

Die Experimente zeigen, dass die Übersetzungszeit durch DAG-Unterstützung kaum beeinflusst wird solange nur Ergebnisse wiederverwendet werden. Lediglich neue Optimie-



Plan			
Übersetzung [ms]	9.3	9.2	9.7
Ausführung [ms]	11933	7480	3535
	Baum	DAG	DAG (Magic Sets)

Abbildung 6: Ausführungspläne für TPC-H Anfrage 2

rungstechniken wie Magic Sets erhöhen die Übersetzungszeit. Diese wird aber klar von der Laufzeit dominiert, die durch DAGs teilweise drastisch reduziert werden kann. DAGs sind Bäumen klar überlegen, da DAGs nie schlechter sind als Bäume und deren Erzeugung nicht oder nur unwesentlich langsamer ist.

7 Verwandte Arbeiten

Nur sehr wenige Arbeiten behandeln das Erzeugen von DAG-strukturierten Ausführungsplänen. Ein Starburst-Artikel erwähnt, dass DAGs nützlich wären, hält sie aber für zu schwierig [HFLP89]. Ein späterer Artikel über den DB2-Anfrageoptimierer [GLSW93] erläutert, wie mit DAGs Sichten besser unterstützt werden können. Dabei wird das Zwischenergebnis materialisiert. Ähnliche Techniken werden in [Cha98, GLJ01] erwähnt und sind vermutlich Stand der Technik in kommerziellen Datenbanksystemen.

Ein Plangenerator der explizit DAGs unterstützt wird in [Roy98] beschrieben. Er legt zunächst fest, welche Operatoren mehrfach verwendet werden sollen und führt dann eine Plangenerierung wie bei Bäumen durch. Dieser zweistufige Ansatz hat allerdings einige Nachteile: Zum einen ist es nicht leicht, diese Festlegung optimal zu treffen, was mehrere Aufrufe der teuren Suchphase nötig macht. Zum anderen unterstützt der Ansatz prinzipiell keine Operatoren, die Daten mehrfach lesen, da die Festlegung in der ersten Phase eine korrekte Kostenberechnung verhindert. Solche Operatoren können aber nicht immer vermieden werden. Komplexe Prädikate bei Verbundoperatoren erfordern z.B. mehrfaches Lesen.

Schließlich existieren noch eine Reihe von Arbeiten, die DAGs jeweils für bestimmte Probleme betrachten, z.B. [DSRS01, VVK02, BBD$^+$04]. Sie erzeugen allerdings entweder nur heuristische Lösungen oder betrachten DAGs nicht in der Allgemeinheit wie hier behandelt.

8 Zusammenfassung

Der hier vorgestellte Plangenerator erlaubt das Erzeugen optimaler DAG-strukturierter Anfragepläne. Im Gegensatz zu existierenden Ansätzen beschränkt er sich dabei nicht auf Spezialfälle sondern ist ein generischer Ansatz zur Erzeugung von DAGs. Wie in Abschnitt 3 gezeigt führt eine naive Vorgehensweise beim Erzeugen von DAGs zu suboptimalen oder sogar falschen Ausführungsplänen. Für DAGs müssen sowohl die Optimierung als auch die Planausführung angepasst werden. Dafür sind die erzeugten Pläne erheblich besser und erlauben neue Techniken, die mit Bäumen nicht zu realisieren waren. Insgesamt sind DAGs ein so großer Gewinn, dass auf lange Sichte alle Datenbanksysteme DAGs verwenden sollten.

Literatur

[BBD+04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani und Dilys Thomas. Operator scheduling in data stream systems. *VLDB J.*, 13(4):333–353, 2004.

[Cha98] Don Chamberlin. *A complete guide to DB2 universal database*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[DSRS01] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy und S. Sudarshan. Pipelining in Multi-Query Optimization. In *PODS*, 2001.

[GLJ01] César A. Galindo-Legaria und Milind Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *SIGMOD Conference*, 2001.

[GLSW93] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer und Yun Wang. Query Optimization in the IBM DB2 Family. *IEEE Data Eng. Bull.*, 16(4):4–18, 1993.

[GMUW99] Hector Garcia-Molina, Jeffrey D. Ullman und Jennifer Widom. *Database System Implementation*. Prentice-Hall, Inc., 1999.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman und Hamid Pirahesh. Extensible Query Processing in Starburst. In *SIGMOD'89*, Seiten 377–388. ACM Press, 1989.

[MFPR90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh und Raghu Ramakrishnan. Magic is Relevant. In *SIGMOD'90*, Seiten 247–258. ACM Press, 1990.

[Roy98] Prasan Roy. Optimization of DAG-Structured Query Evaluation Plans. M.tech. thesis, Dept. of Computer Science and Engineering, IIT-Bombay, January 1998.

[VVK02] Satyanarayana R. Valluri, Soujanya Vadapalli und Kamalakar Karlapalem. Sprinkling Selections over Join DAGs for Efficient Query Optimization. *CoRR*, cs.DB/0202035, 2002.

Thomas Neumann wurde am 22. Februar 1977 in Köln geboren. Nach dem Abitur studierte er Wirtschaftsinformatik an der Universität Mannheim. Während des Studiums war er Stipendiat der Studienstiftung des Deutschen Volkes. Nach dem Diplom promovierte er im Bereich Datenbanken mit dem Schwerpunkt Anfrageoptimierung. Zur Zeit ist er PostDoc in der Datenbankgruppe am Max-Planck-Institut für Informatik in Saarbrücken.