Algorithmen für Sortierung und relationale Operatoren

In diesem Kapitel betrachten wir folgende Themen:

1. Relationale Operatoren und Sortieroperator
2. Grundlegende Operationen: Tabellen-Scan, Index-Scan, Sortier-Scan
3. Klassifizierung von Algorithmen
4. Kostenmodell für die Analyse der Algorithmen
5. Externes Sortieren
6. Algorithmen mit einem Durchlauf
7. Nested-Loop-Verbund
8. Algorithmen mit zwei Durchläufen
9. Algorithmen mit mehreren Durchläufen
10. Pipelining und das Volvano-Iterator-Modell

Relationale Operatoren und Sortieroperator

Die wichtigsten relationalen Operatoren sind:

- σ Restriktion/Selektion $\sigma_C(R)$
- π Projektion $\pi_L(R)$
- $\cup$ Vereinigung $R \cup S$
- $\cap$ Durchschnitt $R \cap S$
- $-$ Differenz $R - S$
- $\times$ Kartesisches Produkt $R \times S$
- $\bowtie$ Verbund/Join $R \bowtie S$
- γ Gruppierung und Aggregation $\gamma_L(R)$
- δ Duplikatelimination $\delta(R)$

Das relationale Modell sieht eine Relation als eine *Menge* von Tupeln, die folglich keine Duplikate enthält. Deshalb ist die Duplikatelimination δ eigentlich kein relationaler Operator. Gleichwohl wird auch im relationalen Modell Duplikatelimination angewandt und deshalb interessieren wir uns für Algorithmen dafür.

In SQL kann eine Tabelle auch Duplikate haben, d.h. es handelt sich um eine *Multimenge* von Datensätzen (Tupeln).

Im Folgenden werden wir Algorithmen für beide Auffassungen einer Relation bzw. Tabelle behandeln. Wenn nötig werden wir in der Darstellung der Operatoren unterscheiden, so steht z.B. $R \cup_S S$ für die Vereinigung der *Mengen* (*set*), während $R \cup_B S$ die Vereinigung von *Multimengen* (*bag*) bezeichnet.

Zusätzlich gibt es noch den Sortieroperator

- τ Sortierung $\tau_L(R)$

Der Sortieroperator τ ist kein relationaler Operator, denn sein Ergebnis ist eine *sortierte Liste*, keine Relation.

Algorithmen für die Sortierung sind jedoch auch unabhängig vom Sortieroperator wichtig. Sie werden nicht nur eingesetzt, wenn in einer Anweisung explizit τ vorkommt, sondern bei der Anfragebearbeitung kann Sortierung als Teil eines Algorithmus für einen relationalen Operator eingesetzt werden. Eine Klasse von Algorithmen beruht auf Sortierung, wie wir sehen werden.

Grundlegende Operationen: Tabellen-Scan, Index-Scan, Sortier-Scan

Voraussetzung für nahezu alle Algorithmen sind Operationen, die die Relationen – die ja oft Eingabeparameter eines Algorithmus sind – im Hauptspeicher zur Verarbeitung bereitstellen. Es gibt drei dieser grundlegenden Operationen:

1. Tabellen-Scan (*table scan*)
   – ein Algorithmus, der den kompletten Inhalt einer Relation liest. Auf physischer Ebene bedeutet dies, dass die Relation blockweise vom Sekundärspeicher in den Hauptspeicher gelesen wird. Konzeptionell gesehen steht die Relation tupelweise zur Verfügung.

   Eine Variante des Tabellen-Scans besteht darin, dass er direkt mit einer Selektion verbunden wird, d.h. beim Lesen wird eine Bedingung auf die Tupel (ein Prädikat) gleich unmittelbar ausgewertet.

2. Index-Scan (*index scan*)
   Hat eine Tabelle einen Index, dann kann man einen Index verwenden, um die Tupel der Relation sortiert in der Reihenfolge zu lesen, die der Index vorgibt. Auf diese Weise kann der Index-Scan als Grundlage von Algorithmen verwendet werden, die auf Sortierung basieren.

3. Sortier-Scan (*sort scan*)
   Bei diesem Operator wird nicht nur die Relation R vorgegeben, die durchlaufen werden soll, sondern auch eine Liste L von Attributen, nach denen die Ergebnistupel sortiert sein sollen. Die Sortierung kann dann auf unterschiedliche Weise durchgeführt werden:

   - Wenn es einen Index über die Liste der Sortierattribute gibt, entspricht der Sortier-Scan dem Index-Scan.
   - Die Sortierung wird (bei kleinen Relationen) im Hauptspeicher durchgeführt.
   - Der Sortier-Scan setzt einen der Algorithmen des externen Sortierens ein, wie wir sie später kennenlernen werden.

Klassifizierung von Algorithmen

Klassifizierung nach der Vorgehensweise

- Algorithmen, die auf *Sortierung* beruhen
- Algorithmen, die auf *Hashing* beruhen
- Algorithmen, die auf *Indexierung* beruhen

Klassifizierung nach Zahl der Durchläufe

Je nach der Größe der zu verarbeitenden Relationen im Verhältnis zur Größe des Pufferbereichs im Hauptspeicher unterscheidet man die Algorithmen nach der Zahl der Durchläufe:

- Algorithmen mit *einem* Durchlauf
  d.h. die beteiligten Relationen müssen genau einmal vom Sekundärspeicher gelesen werden.
- Algorithmen mit *zwei* Durchläufen
  d.h. die Relationen, die als Argumente des Algorithmus auftreten, sind so groß, dass einmal Zwischenergebnisse auf den Sekundärspeicher geschrieben werden müssen, die dann in einem zweiten Durchlauf zum Endergebnis verarbeitet werden können.

- Algorithmen mit *mehreren* Durchläufen.

Kostenmodell für die Analyse der Algorithmen

Die Größe, die für die Analyse der Kosten interessiert, sind die *I/O-Kosten*, gemessen in der Anzahl der Blöcke, die gelesen oder geschrieben erden müssen.

Wir gehen davon aus, dass alle Hauptspeicheroperationen im Vergleich zu I/O so wenig Zeit kosten, dass wir sie für die Analyse der Kosten der Algorithmen vernachlässigen können. Das bedeutet, dass wir ein relativ grobes Kostenmodell verwendet.[1]

Will man tatsächlich konkrete Zahlen für die Kosten der Plattenzugriffe haben, spielt es eine wichtige Rolle, ob die Blöcke, die transferiert werden sollen, fortfolgend auf der Platte liegen oder nicht. Auch diesen Effekt berücksichtigt unser Kostenmodell nicht.

Beim Vergleich von Algorithmen berücksichtigen wir *nur* die I/O-Kosten bei der Verarbeitung der Eingabeparameter der Algorithmen und unterstellen, dass das erzeugte Ergebnis *im Hauptspeicher* verbleibt. Wenn Algorithmen zusammengesetzt werden, muss man dies gegebenenfalls berücksichtigen. Es kommt aber oft vor, dass das Ergebnis eines Algorithmus gar nicht „materialisiert“ wird; deshalb verwenden wir diese Konvention bei der Berechnung der Kosten der Algorithmen.

Parameter für die Kostenberechnung

Sei R eine Relation. Wir verwenden:

$B(R)$	Zahl der Blöcke der Relation R
$T(R)$	Zahl der Tupel der Relation R
M	Zahl der Seiten der Pufferverwaltung für die Verarbeitung

Externes Sortieren

Es gibt viele Algorithmen zum Sortieren von Relationen einer Größe, die die des Hauptspeichers überschreitet. Wir betrachten in diesem Abschnitt zwei Extreme: den *2-Wege-Merge-Sort* in einer Variante, mit der man mit 3 Seiten im Hauptspeicher auskommt und den *2-Phasen-Mehrweg-Merge-Sort*, der möglichst viel Hauptspeicher ausnutzt.

[1] Diese Annahmen werden zusehends fragwürdig: 1. Heute haben Datenbankserver nicht mehr 1 MB Hauptspeicher wie vor Jahren, sondern bis zu 2 TB und 2. ist die Geschwindigkeit der Prozessoren im Vergleich zu Zugriffen auf RAM viel schneller – 1980 dauerte ein DRAM-Zugriff etwa 1-2 Prozessorzyklen, heute 300 [1].

2-Wege-Merge-Sort

Gegeben sei eine Relation mit $N = 2^k$ Seiten (oder einer Anzahl von $> 2^{k-1}$ und $\leq 2^k$ Seiten). Der Algorithmus soll als Ergebnis die Relation sortiert ergeben.

Der Algorithmus sortiert die Relation in $k+1$ Durchläufen auf folgende Weise:

Durchlauf 0
Input: $N = 2^k$ unsortierte Seiten.
Output: 2^k sortierte Läufe in der Größe einer Seite.

In diesem Durchlauf wird die Relation seitenweise gelesen, die Seite sortiert und als sortierter Lauf auf die Platte geschrieben (siehe Abb. 1).

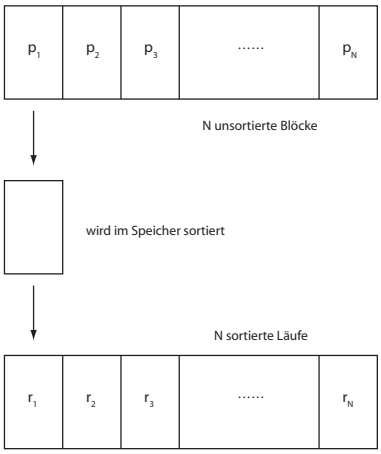

Abbildung 1: Durchlauf 0 im 2-Wege-Merge-Sort

Durchlauf 1
Input: 2^k sortierte Läufe.
Output: 2^{k-1} sortierte Läufe.

In diesem Durchlauf werden jeweils zwei sortierte Läufe gelesen, gemischt und dann ein sortierter Lauf der doppelten Größe ausgegeben (siehe Abb. 2).

Durchlauf n
Input: Sortierte Läufe.
Output: halb so viele sortierte Läufe der doppelten Länge.

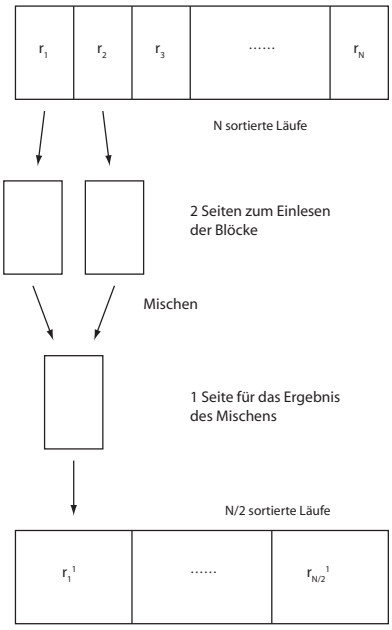

Abbildung 2: Durchlauf 1 im 2-Wege-Merge-Sort

In jedem weiteren Durchlauf, werden die sortierten Läufe seitenweise in einen der beiden Einleseseiten gelesen und das Ergebnis des Mischens wird in die Ausgabeseite geschrieben. Ist diese voll, wird sie auf den Sekundärspeicher geschrieben.

Eigenschaften des Algorithmus Der 2-Wege-Merge-Sort kann Relationen beliebiger Größe sortieren und benötigt dazu einen Hauptspeicher in der Größe von 3 Seiten.

Bei einer Größe der Relation von $B = 2^k$ Blöcken beträgt der Aufwand bei diesem Algorithmus $2(k+1)B$. Denn man hat $k+1$ Durchläufe und muss bei jedem Durchlauf die komplette Relation lesen und schreiben.

Beispiel Nehmen wir an, wir sortieren eine Tabelle von 8 GB mit einer Blockgröße von 8 KB. Dann ist $B = 10^6$ und $k = \lceil log_2 10^6 \rceil = 20$.

Daraus ergibt sich, dass 42×10^6 I/O-Zugriffe statt finden. Rechnet man für einen Zugriff auf einen Block $10\,ms$ dann dauert das externe Sortieren $42 \times 10^4\,sec = 116\,h$!

2-Phasen-Mehrweg-Merge-Sort (2PMMS)

Der 2PMMS verwendet möglichst viel Hauptspeicher. Wir gehen davon aus, dass die Pufferverwaltung $M+1$ Seiten hat und dass die Blockgröße

gleich der Seitengröße ist.

Der Algorithmus hat als Input eine unsortierte Relation mit der Eigenschaft, dass $B \leq M^2$ gilt. Unter dieser Voraussetzung kommt der Algorithmus mit 2 Phasen aus:

Phase 1
Die Relation wird blockweise in die M Seiten gelesen und dort im Hauptspeicher sortiert. Das Ergebnis wird als Lauf mit der Länge von M Seiten auf den Sekundärspeicher geschrieben.
Das Ergebnis von Phase 1 sind also sortierte Läufe, die maximal M Blöcke groß sind.

Phase 2
In der zweiten Phase werden die sortierten Läufe gemischt. Jeder Lauf wird sukzessive in eine der Seiten gelesen und in die Ausgabeseite durch Mischen sortiert ausgegeben (siehe Abb. 3). Ist die Ausgabeseite voll, wird sie in den Sekundärspeicher geschrieben.
Wenn es nach der ersten Phase maximal M Läufe gibt, kann der Algorithmus in der zweiten Phase die ganze Relation sortiert ausgeben.

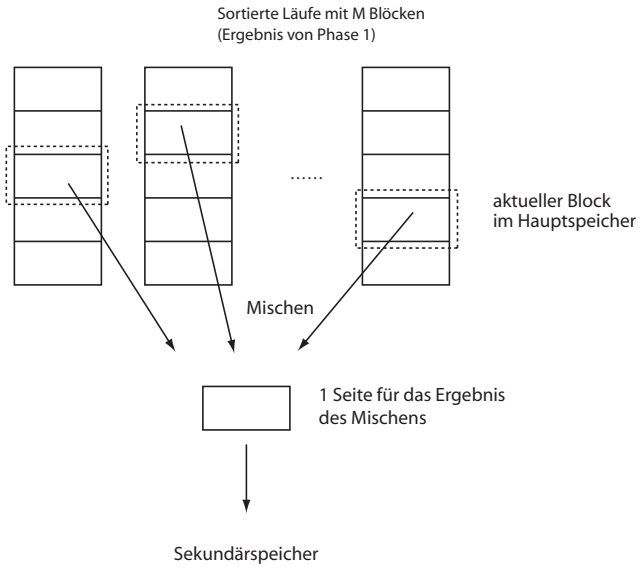

Abbildung 3: Phase 2 im 2-Phasen-Mehrweg-Merge-Sort

Eigenschaften des Algorithmus Der 2PMMS setzt voraus, dass $B \leq M^2$ ist, er lässt sich jedoch leicht verallgemeinern für beliebig große Relationen.

Ist die Voraussetzung erfüllt, dann benötigt der Algorithmus $3B$ I/O-Zugriffe. Denn in Phase 1 wird die Relation einmal komplett gelesen und geschrieben, also $2B$, in Phase 2 wird die Relation einmal gelesen, also B.

Beispiel Nehmen wir wieder an, dass eine Tabelle von 8 GB mit einer Blockgröße von 8 KB sortiert werden soll. Stellen wir uns vor, dass wir 100000 Seiten im Hauptspeicher haben können, d.h. die Pufferverwaltung hat Zugriff auf über 800 MB Hauptspeicher. Dann ist die Voraussetzung sicherlich erfüllt.

Also haben wir 3×10^6 I/0-Zugriffe. Dauert ein Zugriff auf einen Block $10\,ms$, dann dauert Sortieren mit dem 2PMMS in diesem Beispiel $3 \times 10^4\,sec \approx 8\,h$.

Voraussetzung des 2PMMS Wie schwer wiegt die Voraussetzung $B \leq M^2$? Hat man etwa Blöcke von 64 kB und 1 GB Hauptspeicher, dann ist die Zahl M der Seiten $16\,K$. Also können Relationen bis zu $(16\,K)^2$ in 2 Phasen sortiert werden, das sind $4\,TB$.

Algorithmen mit einem Durchlauf

Restriktion/Selektion $\sigma_C(R)$

Algorithmus
Lese die Relation R blockweise
Verwende für jedes Tupel die Bedingung C
Gebe Tupel aus, die C erfüllen.

Eigenschaften

- Der Algorithmus ist unabhängig von der Zahl B der Blöcke der Relation R.
- Er benötigt mindestens eine Seite im Hauptspeicher unabhängig von der Größe von B.

Varianten bei einfacher Bedingung C
Angenommen die Bedingung C hat die Form *attr op value*, dann kann man den Algorithmus unter bestimmten Voraussetzungen deutlich verbessern:

1. Ist R nach *attr* sortiert, kann man in R binäre Suche machen.

2. Gibt es einen B^+-Baum-Index über $attr$, kann man „binäre Suche“ über den Index machen. In diesem Fall kann man den Index auch bei einer Bereichsanfrage verwenden.

3. Gibt es einen Hashindex, kann man ihn für die Suche verwenden. Der Hashindex eignet sich aber nur bei der Punktsuche, nicht bei der Bereichssuche.

Varianten bei komplexer Bedingung C

1. Besteht die Bedingung C aus Termen, die mit $\wedge$ verbunden sind: Dann kann man überprüfen, ob eine der Teilbedingungen ein Attribut hat, auf dem ein Index definiert ist und dann zunächst diesen Index verwenden. Hat etwa das Attribut in der Teilbedingung C_1 einen Index, dann formuliert man die Anfrage um:

   $$\sigma_{C_1 \wedge C_2}(R) = \sigma_{C_2}(\sigma_{C_1}(R))$$

   Für den Teilausdruck $\sigma_{C_1}(R)$ kann man nun Suche über den Index anwenden. Man erhält in der Regel eine Ergebnismenge, die deutlich kleiner ist als R, auf der man dann C_2 auswertet. Durch das „Pipelining“, das wir später diskutierten werden, werden für den zweiten Schritt keine zusätzlichen I/O-Zugriffe benötigt.

2. Besteht die Bedingung C aus Termen, die mit $\wedge$ verbunden sind und wo bei mehreren Attributen ein Index vorhanden ist, kann man folgendes Gesetz ausnutzen:

   $$\sigma_{C_1 \wedge C_2}(R) = \sigma_{C_1}(R) \cap \sigma_{C_2}(R)$$

   In diesem Fall kann man für die beiden Teilbedingungen den jeweiligen Index verwenden und dann das Gesamtergebnis als Durchschnitt der Teilergebnisse bilden.

3. Im allgemeinen Fall bildet man die *konjunktive Normalform (CNF)*, d.h. die äquivalente Formel, die aus mit $\wedge$ verbundenen Klauseln besteht, deren Terme durch $\vee$ verbunden sind.
   Nun kann man die beiden obigen Überlegungen für die Klauseln anwenden.

4. Hat man es mit Bedingungen zu tun, deren Terme nur noch durch $\vee$ verbunden sind, prüft man für jeden der Terme, ob man einen Index verwenden kann.

   Ist unter den Termen einer, für den man keinen Index verwenden kann, dann bleibt nur ein Tabellen-Scan mit Filterung als

Strategie. Hat man hingegen einen Index für jeden der Terme zur Verfügung, wendet man folgende Gleichung an:

$$\sigma_{C_1 \vee C_2}(R) = \sigma_{C_1}(R) \cup \sigma_{C_2}(R)$$

Projektion $\pi_L(R)$

Algorithmus
Lese die Relation R blockweise
Erzeuge für jedes Tupel das Ausgabetupel
Gebe es aus.

Eigenschaften

- Der Algorithmus ist unabhängig von der Zahl B der Blöcke der Relation R.
- Er benötigt mindestens eine Seite im Hauptspeicher unabhängig von der Größe von B.
- Ein Tabellen-Scan ist in jedem Fall erforderlich.

Bemerkung
Der oben beschriebene Algorithmus für die Projektion ist nur korrekt, wenn man Relationen als *Multimengen* auffasst. SQL tut dies.

Wenn man Relationen als *Mengen* auffasst, schließt die Projektion automatisch eine Duplikatelimination δ ein und der Algorithmus wird dadurch komplizierter. Eine naive Vorgehensweise besteht darin, dass man obigen Algorithmus verwendet und ihm einen Algorithmus zur Duplikatelimination folgen lässt. Es kann aber auch sein, dass die Duplikatelimination gar nicht notwendig ist: dann etwa, wenn die Liste L der Attribute der Projektion ein in R eindeutiges Attribut enthält!

Vereinigung $R \cup S$

Algorithmus für Multimengen $R \cup_B S$
Lese die Relation R blockweise.
Schreibe die Tupel in die Ausgabe.
Lese die Relation S blockweise.
Schreibe die Tupel in die Ausgabe.

Eigenschaften

- Der Aufwand ist $B(R) + B(S)$.
- Er benötigt mindestens eine Seite im Hauptspeicher unabhängig von der Größe von B.

Algorithmus für Mengen $R \cup_S S$
Es sei $B(R) \geq B(S)$.
Lese die Relation S in max. $M-1$ Seiten ein. Organisiere die Tupel dabei so, dass man leicht Tupel vergleichen kann, z. B. durch Hash-Struktur im Hauptspeicher.
Lese die Relation R blockweise und vergleiche jedes Tupel t mit denen aus S. Ist $t \in S$, dann überspringe t; andernfalls gebe t aus.
Gebe die Tupel von S komplett aus.

Eigenschaften

- *Voraussetzung*: $min(B(R), B(S)) < M$
- Der Aufwand ist $B(R) + B(S)$.

Durchschnitt $R \cap S$

Algorithmus für Multimengen $R \cap_B S$
Es sei $B(R) \geq B(S)$.
Lese die Relation S in max. $M-1$ Seiten ein. Organisiere die Tupel dabei so, dass man leicht Tupel vergleichen kann, z. B. durch Hash-Struktur im Hauptspeicher.
Kommt ein Tupel in S mehrfach vor, wird es nur einmal gespeichert, und ein Zähler zum Tupel erhöht.
Lese R blockweise und analysiere die Tupel t von R:
Ist $t \notin S$: ignoriere t.
Ist $t \in S$ und der Zähler für t in S ist > 0: gebe t aus und erniedrige den Zähler für t in S.
Ist $t \in S$ und der Zähler für t in S ist $= 0$: ignoriere t.

Eigenschaften

- *Voraussetzung*: $min(B(R), B(S)) < M$
- Der Aufwand ist $B(R) + B(S)$.

Algorithmus für Mengen $R \cap_S S$
Es sei $B(R) \geq B(S)$
Lese die Relation S in max. $M-1$ Seiten ein. Organisiere die Tupel dabei so, dass man leicht Tupel vergleichen kann, z. B. durch Hash-Struktur

im Hauptspeicher.
Lese die Relation R blockweise und vergleiche jedes Tupel t mit denen aus S. Ist $t \notin S$, dann überspringe t; andernfalls gebe t aus.

Eigenschaften

- *Voraussetzung*: $min(B(R), B(S)) < M$
- Der Aufwand ist $B(R) + B(S)$.

Differenz $-$

Es sei im Folgenden $B(S) < M$

Algorithmus für Multimengen $S -_B R$
Lese S in $M - 1$ Blöcke ein und merke für jedes Tupel einen Zähler für die Zahl des Vorkommens in S.
Lese R blockweise ein und ziehe für jedes Tupel in R, das auch in S vorkommt, 1 vom Zähler ab.
Gebe alle Tupel von S mit positivem Zähler aus.

Algorithmus für Multimengen $R -_B S$
Lese S in $M - 1$ Blöcke ein und merke für jedes Tupel einen Zähler für die Zahl des Vorkommens in S.
Lese R blockweise ein und analysiere jedes Tupel t von R:
Ist $t \notin S$: gebe t aus.
Ist $t \in S$ und Zähler > 0: dekrementiere Zähler.
Ist $t \in S$ und Zähler $= 0$: gebe t aus.

Algorithmus für Mengen $S -_S R$
Lese S in $M - 1$ Blöcke ein und bilde geeignete Datenstruktur zum Vergleich von Tupeln.
Lese R blockweise ein und analysiere jedes $t \in R$:
Ist $t \notin S$: ignoriere t.
Ist $t \in S$: lösche t aus der Datenstruktur für S
Am Ende werden die in der Datenstruktur für S verbliebenen Tupel ausgegeben.

Algorithmus für Mengen $R -_S S$
Lese S in $M - 1$ Blöcke ein und bilde geeignete Datenstruktur zum Vergleich von Tupeln.
Lese R blockweise ein und analysiere jedes $t \in R$:
Ist $t \in S$: ignoriere t.
Ist $t \notin S$: gebe t aus.

Eigenschaften

- *Voraussetzung*: $min(B(R), B(S)) < M$
- Der Aufwand ist $B(R) + B(S)$.

Kartesisches Produkt $\times$

Es sei im Folgenden $B(S) < M$

Algorithmus für das kartesische Produkt $R \times S$
Lese S in den Hauptspeicher ein.
Lese R blockweise und bilde für jedes Tupel $t \in R$ alle Tupel $t \circ s$ für jedes Tupel $s \in S$ und gebe es aus.

Eigenschaften

- *Voraussetzung*: $min(B(R), B(S)) < M$
- Der Aufwand ist $B(R) + B(S)$.
- Beim kartesischen Produkt zeigt unser Kostenmodell kein realistisches Bild, denn es kann recht lange dauern, wenn alle Kombinationen der Tupel in R und S ausgegeben werden sollen. (Wenn man das Ausgeben mitberechnet.)

Natürlicher Verbund $R \bowtie S$

Es sei wieder $B(S) < M$.
Wir schreiben $R \bowtie S$ in der Form $R(X, Y) \bowtie S(Y, Z)$, wobei Y die gemeinsamen Attribute von R und S sind.

Algorithmus für $R \bowtie S$
Lese S in den Hauptspeicher ein und bilde dabei eine interne Datenstruktur, mit der man über die Attribute Y schnell suchen kann.
Lese R blockweise ein und vergleiche für jedes Tupel $t \in R$, ob es in den Attributen in Y mit einem Tupel $s \in S$ übereinstimmt. Ist dies der Fall, dann gebe $t \circ s$ aus.

Eigenschaften

- *Voraussetzung*: $min(B(R), B(S)) < M$
- Der Aufwand ist $B(R) + B(S)$.

Bemerkungen

- Equijoins kann man analog durchführen.
- Thetajoins betrachtet man als Equijoin bzw. kartesisches Produkt gefolgt von einer Restriktion, also im allgemeinen Fall:

$$R \bowtie_C S = \sigma_C(R \times S)$$

Gruppierung mit Aggregation $\gamma_L(R)$

Algorithmus für γ
Lese R blockweise ein und bilde eine Datenstruktur, die zu jeder Gruppe *einen* Eintrag enhält.
Bei jedem neuen Tupel wird dann entweder ein neuer Eintrag in dieser Datenstruktur erzeugt oder, wenn bereits Informationen zur Gruppe vorhanden sind, werden die Daten des Tupels entsprechend der Angaben zur Aggregation in L in diesen Eintrag eingearbeitet.

Dies geht in der beschriebenen Weise nur, wenn die Aggregatfunktionen in L *additiv* sind, d.h. der gewünschte Wert kann aus einem Zwischenergebnis zusammen mit einem neuen Tupel berechnet werden, etwa bei `count`, `sum`, `min`, `max` oder `avg`. Es gibt jedoch auch Aggregatfunktionen, bei denen *alle* Werte zur Berechnung zur Verfügung stehen müssen, wie etwa `median`[2].

Eigenschaften

- Dieser Algorithmus hat *Voraussetzungen*: man braucht eine Seite im Hauptspeicher für das sukzessive Lesen von R und dann so viele Seiten, wie die Ergebnisrelation nach Gruppierung groß wird. Da das Ergebnis sicherlich nicht größer als R ist, ist die Voraussetzung $B(R) < M$ sicherlich hinreichend.
- Die Kosten sind $B(R)$.
- Ist die Relation größer, kann man die Relation zunächst sortieren nach den Gruppierungsattributen. Danach genügt ein Durchlauf, um die Ergebnisrelation zu bilden. Möglich ist auch die *Nested-Loop-Gruppierung* [2, S.302f], die analog zum Nested-Loop beim Verbund funktioniert – dazu später mehr.

[2] Der Median einer Menge von Werten ist ein Wert, der die Menge bezüglich einer Vergleichsfunktion in zwei gleich große Teilmengen aufteilt.

Duplikatelimination δ

Algorithmus für δ
Lese R blockweise ein und bilde dabei eine interne Datenstruktur, die den Vergleich von Tupeln schnell erlaubt.
Bei jedem neu gelesenen Tupel überprüft man, ob es schon in der Datenstruktur enthalten ist, wenn ja, wird es ignoriert.
Am Ende gibt man die Tupel in dieser Datenstruktur aus.

Eigenschaften

- Der Algorithmus hat *Voraussetzungen*: man braucht eine Seite im Hauptspeicher für das sukzessive Lesen von R und dann so viele Seiten, dass das gesamte Ergebnis in den Hauptspeicher passt. Genau: es muss gelten $B(\delta(R)) < M$. Diese Zahl kann man jedoch kaum vorher berechnen, es sei denn man berechnet $\delta(R)$ selbst. Hinreichend ist: $B(R) < M$.

Nested-Loop-Verbund für $R \bowtie S$

Der Algorithmus, der Nested-Loop-Verbund (*nested loop join*) genannt wird, kann für Relationen beliebiger Größe verwendet werden. Dabei wird die eine Relation genau einmal in den Speicher gelesen, während das andere Argument ggfs. immer wieder gelesen wird. Man spricht deshalb auch von einem Algorithmus mit „1 1/2 Durchgängen“.

Wir schreiben wieder $R \bowtie S$ in der Form $R(X, Y) \bowtie S(Y, Z)$, wobei Y die gemeinsamen Attribute von r und S sind.

Tupelbasierter Nested-Loop-Verbund

```
nljoin( R, S) {

  // äußere Relation
  for each tuple r in R {
    // innere Relation
    for each tuple s in S {
      if r and s join to r·s {
        output r·s
      }
    }
  }
}
```

Wir bezeichnen mit $B(R)$ und $T(R)$ die Zahl der Blöcke bzw. Tupel von R und analog mit $B(S)$ und $T(S)$ die Zahl der Blöcke bzw. Tupel von

S.

Eigenschaften

- Die I/O-Kosten für den Nested-Loop-Verbund sind $B(R)+(T(R)\times B(S))$.
- Im Hauptspeicher kommt man mit 3 Seiten aus: eine für das Lesen von R, eine für das Lesen von S und eine für den Output.
- Der Algorithmus funktioniert für Relationen beliebiger Größe

Beispiel Angenommen wir haben eine Relation R mit 500 Blöcken und 50.000 Tupeln sowie eine Relation S mit 1000 Blöcken.

Dann ergibt sich ein I/O-Aufwand von $500+50.000\times 1.000$. Rechnet man mit $10\,ms$ für das Lesen eines Blocks benötigt man $500.000\,s \approx 140\,h$ für den Verbund in diesem Beispiel.

Ideen

- Der Algorithmus verwendet nur 3 Seiten im Hauptspeicher. Es ist günstiger, möglichst viele Blöcke der äußeren Relation in den Hauptspeicher zu lesen und dann jedes Tupel der inneren Relation mit all diesen Tupeln zu vergleichen. Diese Idee führt zum blockbasierten Nested-Loop-Verbund.
- Man kann auch die Suche nach den „passenden“ Tupeln intelligenter machen, indem man z.B. einen Index auf den Join-Attributen verwendet. Wir werden später Algorithmen für den Verbund betrachten, die auf Indexierung, Hashing oder Sortierung basieren.

Blockbasierter Nested-Loop-Verbund

Wieder schreiben wir $R(X,Y) \bowtie S(Y,Z)$.

Wir setzen voraus, dass wir $M+1$ Seiten im Hauptspeicher zur Verfügung haben und ferner, dass $B(R) \leq B(S)$ gilt, also die äußere Relation die kleinere ist.

```
block_nljoin( R, S ) {
  for each chunk R' of M-1 blocks of R {
    read R'
    organize R' for fast search w.r.t Y
    for each block S' of S {
```

```
        read S'
        for each tuple s of S' {
          find matching tuple r in R'
          output r·s
        }
      }
    }
  }
```

Eigenschaften

- R wird einmal gelesen, S aber nicht mehr für jedes Tupel von R, sondern nur noch für jeweils $M-1$ Blöcke von R, also ergeben sich als I/O-Kosten:

$$B(R) + \left\lceil \frac{B(R)}{M-1} \right\rceil \times B(S).$$

  Im Spezialfall, dass die kleinere Relation R komplett in den Hauptspeicher passt, also nur $B(R) + B(S)$ (siehe oben).

- Der Hauptspeicher wird mit seinen $M+1$ Seiten voll ausgenutzt.

Beispiel Angenommen $M = 101$. Dann ergibt sich mit $B(R) = 500$ und $B(S) = 1.000$ ein I/O-Aufwand von $500 + \frac{500}{100} \times 1000 = 5.500$.

Bei einer Zugriffszeit von $10\,ms$ pro Block dauert der blockbasierte Nested-Loop-Verbund in diesem Beispiel $5,5\,s$.

Algorithmen mit zwei Durchläufen

Für σ und π haben wir einen Algorithmus gesehen, der unabhängig von der Größe der Relation mit einem Durchlauf auskommt, ebenso für $\cup_B$. Also müssen wir nur die restlichen Operatoren untersuchen.

Vereinigung $\cup_S$ – auf Sortierung basierend

Algorithmus für $R \cup_S S$
Man verwendet einen modifizierten 2PMMS:

Phase 1: Erzeuge sortierte Teillisten für R und S.

Phase 2: Erzeuge die Mengenvereinigung aus den Teillisten: Man nimmt das entsprechend der Sortierung nächste Tupel und kopiert es in den Output. Kommt das Tupel auch in einer anderen Teilliste vor, ignoriert man es dort.

Eigenschaften

- *Voraussetzung:* $B(R) + B(S) \leq M^2$
- Die Kosten sind wie beim Sortieren *einer* Relation:
  $3 \times (B(R) + B(S))$

Durchschnitt $\cap$ – auf Sortierung basierend

Algorithmus für $R \cap S$
Auch hier verwendet man eine Variante des 2PMMS:

Phase 1: Erzeugen von sortierten Teillisten für R und S.

Phase 2:
Beim Mengendurchschnitt $R \cap_S S$ nimmt man das in der Sortierung nächste Tupel und kopiert es in den Output, wenn es in einer der Teillisten von R *und* S vorkommt.

Beim Durchschnitt von Multimengen $R \cap_B S$ nimmt man das in der Sortierung nächste Tupel und kopiert es so oft wie das Minimum des Vorkommens in den Teillisten von R und S, sofern dieses > 0 ist.

Eigenschaften

- *Voraussetzung:* $B(R) + B(S) \leq M^2$
- Die Kosten sind wie beim Sortieren *einer* Relation:
  $3 \times (B(R) + B(S))$

Differenz $-$ – auf Sortierung basierend

Algorithmus für $R - S$
Auch hier verwendet man eine Variante des 2PMMS:

Phase 1: Erzeugen von sortierten Teillisten für R und S.

Phase 2: Man liest t entsprechend der Sortierung.

Bei der Mengendifferenz $R -_S S$ kopiert man t in den Output, wenn $t \in R$, aber $t \notin S$.

Bei der Differenz von Multimengen $R -_B S$ kopiert man t in den Output so oft, wie die Differenz des Vorkommen in R und S, sofern diese > 0.

Eigenschaften

- *Voraussetzung:* $B(R) + B(S) \leq M^2$

- Die Kosten sind wie beim Sortieren *einer* Relation:
  $3 \times (B(R) + B(S))$

Kartesisches Produkt $R \times S$

Die Berechnung des kartesischen Produkts in einem Durchlauf hatte zur Voraussetzung, dass die kleinere Relation, sage S, komplett in den Hauptspeicher passt, d.h. $B(S) < M$.

Ist diese Voraussetzung nicht gegeben, macht man einen Nested-Loop-Verbung mit *leerer* Join-Bedingung, denn der natürliche Verbund ergibt das kartesische Produkt, wenn die beiden Relationen gar keine Attribute gemeinsam haben.

(Der Algorithmus mit einem Durchgang, den wir oben diskutiert haben, ist dann gerade der blockbasierte Nested-Loop-Verbund, bei dem die kleinere Relation vollständig in den Hauptspeicher passt.)

Natürlicher Verbund $\bowtie$ – auf Sortierung basierend

Algorithmus für $R(X, Y) \bowtie S(Y, Z)$ Variante 1

- Sortiere R mit 2PMMS und Y als Schlüssel.
- Sortiere S mit 2PMMS und Y als Schlüssel.
- Mische die beiden sortierten Relationen so:
  - nehme das Tupel t mit dem niedrigsten Wert bzgl. Y in R bzw. S.
  - Kommen passende Tupel in der anderen Relation nicht vor, überspringe alle Tupel mit denselben Schlüsseln.
  - Andernfalls: erzeuge Tupel für den Output.

Eigenschaften

- *Voraussetzung* $B(R) \leq M^2$ und $B(S) \leq M^2$. Außerdem müssen alle Tupel, die gemeinsame Werte an den Join-Attributen Y haben, in M Blöcke passen.
- Die Kosten sind $5 \times (B(R) + B(S))$, denn das Sortieren trägt den Faktor 3 bei, das Rausschreiben der sortierten Relationen kostet $B(R) + B(S)$ und das Mischen ebenso.

Algorithmus für $R(X,Y) \bowtie S(Y,Z)$ Variante 2

In dieser Variante wird die Sortierung durch eine Variante des 2PMMS optimaler eingesetzt.

Phase 1: Erzeuge sortierte Läufe für R und S mit dem Schlüssel Y.

Phase 2: Mische die Läufe wie im 2PMMS, berechne dabei aber zugleich die zusammengehörenden Tupel für den Verbund.

Eigenschaften

- *Voraussetzung*: $B(R) + B(S) \leq M^2$, Ferner müssen alle Tupel mit einem gemeinsamen Wert an den Join-Attributen in *eine* Seite passen.
- I/O-Kosten sind dann $3 \times (B(R) + B(S))$ wie bei der Sortierung.

Gruppierung mit Aggregation γ – auf Sortierung basierend

Algorithmus für $\gamma_L(R)$

Eine Variante des 2PMMS sorgt für Gruppierung und Aggregation:

Phase 1: Bilde sortierte Läufe für R.

Phase 2: Mische die Läufe und führe dabei zugleich die Aggregation für die Gruppen durch.

Eigenschaften

- *Voraussetzung*: $B(R) \leq M^2$
- Kosten sind $3 \times B(R)$

Duplikatelimination δ – auf Sortierung basierend

Algorithmus für $\delta(R)$

Wieder eine Variante des 2PMMS:

Phase 1: Bilde sortiert Läufe für R.

Phase 2: Mische die Läufe, gebe aber jeweils nur ein Tupel aus, wenn es Duplikate gibt.

Eigenschaften

- *Voraussetzung*: $B(R) \leq M^2$
- Kosten sind $3 \times B(R)$

Mengenoperatoren $R \cup S, R \cap S, R - S$ – auf Hashing basierend

Idee der Algorithmen
Man verwendet dieselbe Hashfunktion für R und S. Die Hashfunktion partioniert die beiden Relationen in Behälter im ersten Durchlauf. Nun kann man die Berechnung des Ergebnisses pro Behälter durchführen, da die Tupel, die für die Vereinigung, den Durchschnitt und die Differenz verglichen werden müssen, im gleichen Behälter sind.

Wenn M die Zahl der Seiten im Hauptspeicher ist, ist es sinnvoll, eine Hashfunktion mit $M - 1$ Werten im Wertebereich zu verwenden.

(Diese Idee ist auch die Leitidee bei der Parallelisierung von relationalen Operatoren in parallelen oder verteilten Datenbanken.)

Eigenschaften

- *Voraussetzung*: $min(B(R), B(S)) \leq M^2$, denn im zweiten Durchlauf muss man die Vereinigung, den Durchschnitt oder die Differenz der Behälter in einem Durchlauf machen können. Verteilt die Hashfunktion etwa gleichmäßig, hat ein Behälter etwa die Größe $\frac{B(R)}{M-1}$. Für den Algorithmus in einem Durchlauf darf das kleinere Argument höchstens $M - 1$ Seiten füllen.
- Kosten sind $3 \times (B(R) + B(S))$

Hash-Verbund-Algorithmus (*hash join*)

Algorithmus für $R(X, Y) \bowtie S(Y, Z)$

Man verwendet eine Hashfunktion für die Join-Attribute Y. Das hat zur Folge, dass die Tupel mit denselben Werten in Y im selben Behälter landen. Also

- Partitioniere R und S mit einer solchen Hashfunktion im ersten Durchlauf.
- Mache den Verbund im zweiten Durchlauf pro Paar von passenden Behältern.

Eigenschaften

- *Voraussetzung*: $min(B(R), B(S)) \leq M^2$, mit demselben Argument wie oben.
- Kosten sind $3 \times (B(R) + B(S))$

Gruppierung mit Aggregation γ – auf Hashing basierend

Algorithmus für $\gamma_L(R)$

Man wählt eine Hashfunktion, die auf den Gruppierungsattributen beruht, so dass Tupel mit denselben Werten an diesen Attributen im selben Behälter landen.

- Partitioniere R mit der Hashfunktion in, sage $M - 1$, Behälter im ersten Durchlauf.
- Verwende pro Behälter einen Algorithmus für die Gruppierung, der einen Durchlauf benötigt und bilde die Vereinigung der Ergebnisse durch sukzessive Ausgabe.

Eigenschaften

- *Voraussetzung*: $B(R) \leq M^2$
- Kosten sind $3 \times B(R)$

Duplikatelimination δ – auf Hashing basierend

Man wählt eine Hashfunktion, die auf allen Attributen beruht, so dass Duplikate im selben Behälter landen.

Algorithmus für $\delta(R)$

- Partitioniere R mit der Hashfunktion in sage $M - 1$ Behälter im ersten Durchlauf.
- Verwende pro Behälter einen Algorithmus, der einen Durchlauf benötigt und bilde die Vereinigung der Ergebnisse durch sukzessive Ausgabe.

Eigenschaften

- *Voraussetzung*: $B(R) \leq M^2$
- Kosten sind $3 \times B(R)$

Algorithmen für den Verbund – auf Indexierung beruhend

Indexe sind besonders geeignet für Algorithmen für die Restriktion $\sigma_C(R)$ und werden dort wo immer möglich verwendet. In diesem Abschnitt wollen wir uns noch zwei Varianten von Verbund-Algorithmen ansehen, die Indexe verwenden.

Wie stets betrachten wir den natürlichen Verbund $R(X, Y) \bowtie S(Y, Z)$.

Algorithmus für $R(X, Y) \bowtie S(Y, Z)$

Wir gehen davon aus, dass S einen Index auf den Attributen Y hat. Die Idee des Algorithmus besteht nun darin, dass man den blockbasierten Nested-Loop-Verbund macht, aber in der *inneren* Schleife über die Tupel von S verwendet man den Index im Zugriff, damit man nicht die komplette Relation S durchlaufen muss.

Eigenschaften

- Die I/O-Kosten kann man nicht pauschal bennen, sie hängen von der Verteilung der Werte in den Attributen Y bei den beiden Relationen ab.
- Besonders geeignet ist der indexbasierte Algorithmus, wenn R *klein* ist, weil dann der Index auf nur wenige (zutreffende) Tupel in S angewandt werden muss.

Zickzack-Verbund für $R(X, Y) \bowtie S(Y, Z)$

Beim Zickzack-Verbund gehen wir davon aus, dass wir Indexe auf $R(X, Y)$ bezüglich Y und auch auf $S(Y, Z)$ bezüglich Y haben.

Die Idee des Algorithmus besteht darin, dass man zwei Zeiger auf den beiden Relationen entsprechend der Sortierung durch den Index bewegt und dabei Tupel für die Ausgabe zusammenfügt oder nicht passende Tupel überspringt. D.h. man bewegt die beiden Zeiger auf den Indexen für R und S abwechselnd, deshalb die Bezeichnung „Zickzack-Verbund“.

Eigenschaften

- Die I/O-Kosten kann man nicht pauschal benennen. Aber typischerweise muss man von den Nutzdaten nur diejenigen Blöcke lesen, die tatsächlich für den Verbund benötigte Tupel enthalten.

Algorithmen mit mehreren Durchläufen

Man kann die Algorithmen mit zwei Durchläufen erweitern zu Algorithmen mit mehreren Durchläufen – wenn bei den Argumenten Relationen sind, die die Größenbeschränkungen überschreiten.

Die Grundidee wird ersichtlich am Beispiel des Sortierens der Relation R mit mehreren Durchläufen:

Induktionsbasis: Wenn R in den Hauptspeicher passt, d.h. wenn $B(R) \leq M$ gilt, dann liest man R ein, sortiert im Hauptspeicher und schreibt das Ergebnis, also $\tau(R)$ auf die Platte.

Induktionsschritt: Wenn R nicht in den Hauptspeicher passt, partitioniert man R in M gleich große Teile $R_1, R_2, \ldots, R_M$.
Man sortiert rekursiv diese Teile der Relation und mischt dann die M sortierten Läufe $\tau(R_1), \tau(R_2), \ldots, \tau(R_M)$ zu $\tau(R)$.

Diese Idee kann man auch auf die anderen Algorithmen mit zwei Durchläufen anwenden und sie so für beliebig große Relationen mit entsprechender Zahl der Durchläufe einsetzen.

Eigenschaften der Sortierung mit k Durchläufen

- *Voraussetzung*: $B(R) \leq M^k$
- Kosten: $2k \times B(R)$

Pipelining und das Volcano-Iterator-Modell

Wir haben jetzt Algorithmen für die relationalen Operatoren und das Sortieren kennengelernt. Tatsächlich wird man für die Ausführung einer SQL-Anweisung mehrere solche Algorithmen kombinieren müssen. Wie man aus der Anweisung ermittelt, welche Algorithmen in welcher Kombination benötigt werden, sehen wir im nächsten Kapitel. Jetzt wollen wir untersuchen, wie man Algorithmen kombiniert.

Man kann natürlich annehmen, dass jeder Algorithmus seine Eingabe von der Platte liest und seine Ausgabe wieder auf die Platte schreibt. Also etwa wie schematisch in Abb. 4 dargestellt.

Es ist offensichtlich, dass dieses naive Vorgehen hohe I/O-Kosten verursacht. Außerdem kann ein Operator erst dann mit seiner Ausführung beginnen, wenn alle vorherigen Operatoren ihr Ergebnis vollständig erstellt haben.

Die Alternative zu diesem naiven Vorgehen ist *Pipelining*, analog zur Pipes-and-Filters-Architektur:

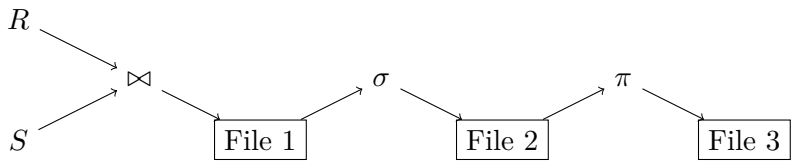

Abbildung 4: Naive Kombination relationaler Operatoren

- Jeder Algorithmus eines Operators kann seine Ergebnisse direkt an den nächsten Algorithmus weitergeben, ohne dass er gezwungen ist, sein Ergebnis zuerst auf die Festplatte zu schreiben.
- Die Algorithmen arbeiten (konzeptionell) parallel, d.h. die Ausgabe des einen Algorithmus wird sofort weiterverarbeitet, wenn dies möglich ist. (Algorithmen, die zunächst ihre komplette Eingabe verarbeiten müssen –etwa Algorithmen für δ– ehe sie eine Ausgabe erzeugen können, haben diese Eigenschaft nicht, man nennt sie deshalb *blockierend*. Gleichwohl haben auch sie die Schnittstelle, die wir gleich betrachten werden.)
- Typischerweise versucht man in relationalen DBMS das Pipelining tupelweise zu betreiben.

Das *Volcano-Iterator-Modell*[3] definiert eine uniforme Schnittstelle für Algorithmen relationaler Operatoren, durch die Pipelining einfach möglich wird.

Die Open-Next-Close-Schnittstelle oder das Volcano-Iterator-Modell

Jeder Algorithmus implementiert folgende Schnittstelle:

- `open()`
  Initialisiert die internen Datenstrukturen für den Algorithmus.
- `next()`
  Ermittelt und liefert das nächste Tupel der Ergebnismenge oder `<eof>`, falls die Ergebnismenge vollständig ist.
- `close()`
  Räumt die internen Informationen und Resourcen des Algorithmus wieder auf.

[3] Das Volcano-Iterator-Modell geht zurück auf Goetz Graefe, der 1990 diese Schnittstelle für das *Volcano Query Processing System* entwickelt und dort zuerst eingesetzt hat.
Goetz Graefe: *Encapsulation of Parallelism in the Volcano Query Processing System*, 1990 in: Joseph M. Hellerstein, Michael Stonebraker: *Readings in Database Systems* Fourth Edition, p.102-111.

Alle Zustandsinformation wird *innnerhalb* der jeweiligen Instanz des Algorithmus gehalten.

In Abb. 5 sieht man die Idee an einem Beispiel.

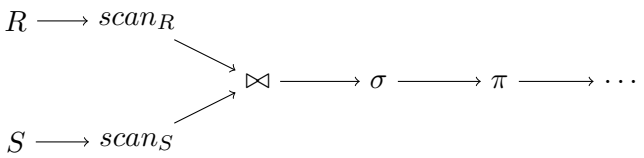

Abbildung 5: Pipelining relationaler Operatoren

An diesem Beispiel ergibt sich dann die Aufrufstruktur wie in Abb. 6.

π.open()

$\longrightarrow \sigma$.open()

$\longrightarrow \bowtie$.open()

$\longrightarrow scan_R$.open()

$\longrightarrow scan_S$.open()

π.next()

$\longrightarrow \sigma$.next()

$\longrightarrow \bowtie$.next()

$\longrightarrow \cdots$

$\cdots$

π.close()

$\longrightarrow \sigma$.close()

$\longrightarrow \bowtie$.close()

$\longrightarrow scan_R$.close()

$\longrightarrow scan_S$.close()

Abbildung 6: Aufrufhierarchie

Im Folgenden als Beispiel der Pseudocode für eine Implementierung des Nested-Loop-Verbunds für $R \bowtie S$ im Volcano-Stil:

(Den Scan einer Relation R schreiben wir einfach als `R.open()`, `R.next()` und `R.close`.

```
open() {
  R.open();
  S.open();
  r := R.next();
}

next() {
  while ( r != <eof> ) {
    while (( s:= S.next() ) != <eof> ) {
      if ( r and s have to be joined ) {
        generate t := r·s;
        return t;
      }
    }
    S.close(); //reset der inneren Schleife
    S.open();
    r: = R.next();
  }
  return <eof>;
}

close() {
  S.close();
  R.close();
}
```

Blockierende Algorithmen

Normalerweise wird die `open()`-Funktion nur den Algorithmus initialisieren und die `next()`-Funktion wird dann sukzessive Tupel der Ergebnismenge produzieren, dies schon ehe die gesamte Eingabe konsumiert ist.

Wie bereits erwähnt, können einige Operatoren nicht so implementiert werden, sie „blockieren“: sie konsumieren die komplette Eingabe, ehe sie anfangen Ergebnistupel zu liefern. Im Volcano-Iterator-Modell bedeutet dies, dass die ganze Arbeit in `open()` gemacht wird, dort die komplette Ergebnismenge bereitgestellt wird und `next()` sie nur sukzessive ausgibt. Typischerweise schreibt `open()` dann die Ergebnismenge auf sekundären Speicher; man sagt dann, die Ergebnismenge wird „materialisiert“.

Folgende Operatoren haben blockierende Algorithmen:

- (Externes) Sortieren τ.

- Gruppierung γ und Duplikatelimination δ bei unsortierter Eingabe.
- Hashbasierte Algorithmen für den Verbund.

Literaturverzeichnis

[1] Peter A. Boncz, Martin L. Kersten, Stefan Manegold. Breaking the Memory Wall in MonetDB. *Commun. ACM*, 51(12), Dezember 2008.

[2] Gunter Saake, Andreas Heuer, Kai-Uwe Sattler. *Datenbanken: Implementierungstechniken.* Heidelberg, 2011.

Burkhardt Renz
TH Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 3.5 – 5. Mai 2014