FachPraktikum 1590
„Erweiterbare Datenbanksysteme“

Aufgaben Phase 1

Wintersemester 2004/2005

Ralf Hartmut Güting, Dirk Ansorge, Thomas Behr, Markus Spiekermann

Praktische Informatik IV, Fernuniversität Hagen

D-58084 Hagen, Germany

Einführung

Liebe Studierende,

dieses Dokument enthält eine Reihe von Aufgaben, die zur Einarbeitung in das SECONDO-System dienen. Inhaltlich werden einige sehr wichtige Teile des Systems behandelt und von Ihnen erweitert. Wenn Sie alle Aufgaben gelöst haben, haben Sie schon große Kenntnisse über die Funktionsweise von SECONDO gesammelt und sind dann bereit, schwierigere Aufgaben zu lösen. Dies wird in der Phase 2 des Praktikums von Ihnen verlangt.

Wichtige zusätzliche Unterlagen, ohne die Sie die Aufgaben nicht bewältigen können, sind in erster Linie das *Secondo User Manual* und der *Secondo Programmer's Guide*. Wie Sie leicht feststellen werden, ist die Struktur der Aufgaben zum Aufbau des *Programmer's Guide* sehr ähnlich. Bevor Sie also eine Aufgabe angehen, sollten Sie das entsprechende Kapitel durcharbeiten.

Wir wünschen Ihnen viel Erfolg beim Lösen der Aufgaben.

Ihre Praktikumsbetreuer

1 Implementierung einer Algebra

Aufgabe 1.1: (Implementierung der `PSTAlgebra`)

Schreiben Sie die C++-Algebra `PSTAlgebra`, die folgende Datentypen zur Darstellung eines Punktes, eines Liniensegmentes, bzw. eines Dreiecks in der Ebene (mit reellen Koordinaten) enthält:

- `PSTPoint`
- `PSTSegment`
- `PSTTriangle`

Für die einzelnen Datentypen sind einige Operationen zu implementieren. Dies sind im einzelnen (sortiert nach ihrer Signatur):

- `equal: PSTPoint x PSTPoint -> bool`
- `equal: PSTSegment x PSTSegment -> bool`
- `intersects: PSTSegment x PSTSegment -> bool`
- Diese Operation liefert `TRUE` zurück, wenn der Schnitt zweier Segmente genau ein Punkt ist. Für zwei überlappende Segmente trifft dies z.B. nicht zu.
- `intersection: PSTSegment x PSTSegment -> PSTPoint`
  Für zwei Segmente *a*, *b*, für die `intersects(a,b) = TRUE` gilt, wird mit `intersection` der Schnittpunkt berechnet.
- `equal: PSTTriangle x PSTTriangle -> bool`
- `inside: PSTPoint x PSTSegment -> bool`
  Diese Operation liefert `TRUE`, wenn der Punkt auf dem Segment liegt.
- `inside: PSTPoint x PSTTriangle -> bool`
  Die Operation `inside` gibt `TRUE` zurück, wenn sich der Punkt innerhalb des Dreiecks oder auf seinem Rand befindet.
- `intersects: PSTSegment x PSTTriangle -> bool`
  Das Ergebnis ist `TRUE`, wenn das Segment eine Kante des Dreiecks schneidet oder völlig innerhalb des Dreiecks liegt.

Aufgabe 1.2: (Einbinden der Algebra in SECONDO)

Erstellen Sie die noch notwendigen Dateien (`make` und `.spec`) und integrieren Sie die neue Algebra in SECONDO.

Aufgabe 1.3: (Erweiterung der `StreamExampleAlgebra`)

Erweitern Sie die `StreamExampleAlgebra` um folgende Operationen:

- `filterdiv: stream(int) x int -> stream(int)`
  Diese Operation läßt alle int-Werte durch, die durch die übergebene Zahl teilbar sind.
- `sum: stream(int) -> int`
  Die Summe aller Zahlen im Stream wird mit `sum` berechnet.

2 Erweiterung der RelationAlgebra

Aufgabe 2.1: (Erweiterung um die Operatoren `Forall` und `Exists`)

In relationalen Datenbanken haben Aggregatfunktionen eine besondere Bedeutung. Beispiele für solche Funktionen in der Algebra `Relation-C++` in Secondo sind `sum` und `avg`, die die Summe bzw. den Durchschnitt über alle Werte eines Attributs einer Relation berechnen.

Hier sollen nun die beiden neuen Operatoren `forall` und `exists` implementiert werden. Beide Operatoren erhalten einen Strom von Tupeln und einen Attributnamen eines `bool`-Attributs. Rückgabewert ist wiederum ein `bool`. Der Operator `forall` liefert `TRUE`, wenn alle Werte eines Attributs `TRUE` sind, sonst `FALSE`. Für `exists` gilt, daß das Ergebnis `TRUE` ist, falls es mindestens ein Tupel gibt, für das das Attribut den Wert `TRUE` hat. Ansonsten wird `FALSE` zurückgegeben.

Hinweise:

- Überlegen Sie, wie Sie das `APPEND`-Kommando in der Type Mapping Funktion verwenden können.
- Vegessen Sie nicht, die Spezifikation der neuen Operatoren im `.spec`-File anzugeben.

Aufgabe 2.2: (Implementierung des `rdup2`-Operators)

Um Duplikate zu entfernen, gibt es in der relationalen Algebra von SECONDO einen `rdup`-Operator. Damit dieser Operator korrekte Ergebnisse liefert, ist es allerdings notwendig, einen sortierten Strom zu übergeben. Nun soll ein Operator `rdup2` implementiert werden, der diese Voraussetzung nicht hat und stattdessen Duplikate mittels Hashing erkennt und entfernt. In der Value Mapping Funktion wird dabei für jedes Tupel ein Hash-Wert berechnet und das Tupel in die Hash-Tabelle eingetragen und in den Ergebnisstrom gegeben, sofern es an dieser Stelle noch kein Duplikat des einzutragenden Elements gibt.

Hinweise:

- Für jedes Attribut, das in Relationen verwendet werden kann, gibt es eine Hash-Funktion. Diese ist in der Klasse `Attribute` implementiert und heißt `HashValue`. Wenn Duplikate von Tupeln entfernt werden sollen, kann z.B. die Summe der Hash-Werte als Adresse für die Hash-Tabelle verwendet werden.
- Ein Beispiel für die Verwendung von Hash-Funktionen finden Sie im `hashjoin`-Operator.

Aufgabe 2.3: (Implementierung der `replace`-Operation)

In der relationalen Algebra von SECONDO findet sich der `extend`-Operator, mit dem eine Relation um neue Attribute erweitert werden kann, die aus bereits bestehenden Attributen berechnet werden. Ein ähnlicher Operator soll nun implementiert werden, der, statt ein neues Attribut zu erzeugen, ein altes überschreibt. Dieser `replace`-Operator erhält einen Tupelstrom, den Namen des zu ersetzenden Attributs und eine Funktion, die bestimmt, wie der neue Wert berechnet werden soll. Zu beachten ist, daß der berechnete Attributwert zum Typ des Attributs passen muß.

Hinweise:

- Schauen Sie sich die Arbeitsweise des `extend`-Operators an.
- Denken Sie daran, daß der Ergebnistyp der übergebenen Funktion zum zu ersetzenden Attribut passen muß.

3 Benutzung des DBArray

Aufgabe 3.1: (Erweiterung der `PSTAlgebra`)

(a) Erweitern Sie die Algebra um einen neuen Typkonstruktor `Segments`, der eine Menge von Segmenten repräsentiert.

(b) Führen Sie ein neues Prädikat `Contains: PSTSegment x Segments -> bool` ein, welches überprüft ob ein `PSTSegment` Objekt in einem `Segments` Objekt enthalten ist.

Aufgabe 3.2: (Komplexere Operationen)

(a) Implementieren Sie die Operation `Intersection: Segments x Segments -> Segments`, die alle Segmente, die in beiden `Segments` Objekten enthalten sind, berechnet.

(b) Schreiben Sie ein kleines Programm, welches Ihnen große `Segments` Objekte generieren kann und importieren Sie diese in eine Datenbank, um damit Tests mit großen Objekten durchzuführen.

4 Einbettung von Algebren in die Relationale Algebra

Erweitern Sie die C++ Klassen der `PSTAlgebra`, so daß die erzeugten Objekte in Relationen verwendet werden können.

5 Speichersystemschnittstelle (SMI)

Aufgabe 5.1:

(a) Machen Sie sich mit dem Interface des SMI näher vertraut, indem Sie das Header File `SecondoSMI.h` studieren.

(b) Schreiben Sie ein Programm, welches große Datenmengen in `SmiFiles` speichert. Lesen Sie dazu Bilddateien ein und speichern Sie diese in Records variabler Länge.

6 Erweiterung des Optimierers

Aufgabe 6.1: (Display-Regeln für Typkonstruktoren)

(a) Schreiben Sie `display`-Regeln zur Darstellung der von Ihnen in Aufgabe 1 eingeführten Typkonstruktoren `PSTPoint`, `PSTSegment`, und `PSTTriangle`.

(b) In der `ArrayAlgebra` gibt es einen Typkonstruktor `array`, dessen Argument ein beliebiger Typ ist. So kann man z.B. einen Array von integer-Werten anlegen:

```
let ia = [const array(int) value (1 2 3 4 5)]
```

Mit dem `loop`-Operator kann man für jedes Element des Arrays einen Ausdruck auswerten:

```
query ia loop[(. * 30) > 100].
```

Ergebnis ist ein Array von booleschen Werten.

Beachten Sie, daß auch Arrays von Relationen gebildet werden können. Z.B. kann man mit dem `distribute`-Operator eine Relation auf verschiedene “Fächer” verteilen; jedes Fach (Feld eines Arrays) enthält dann eine Teilrelation. Eine Query, die einen Array von Relationen liefert, ist z.B.

```
query Staedte feed extend[Fach: .Bev div 400000] distribute[Fach]
```

Schreiben Sie `display`-Regeln zur Darstellung von Arrays. Eine gute Darstellung könnte z.B. so aussehen:

```
----------    1 ----------
<Darstellung des ersten Elementes>
----------    2 ----------
<Darstellung des zweiten Elementes>
...
----------    n ----------
<Darstellung des letzten Elementes>
```

Aufgabe 6.2: (Verwendung eines R-Baums für Selektion mit dem `touches` Operator.)

Es gibt im SECONDO System einen R-Baum, der im Optimierer noch nicht benutzt wird. Wir wollen den Optimierer erweitern, so daß R-Baum-Indexe verwendet werden können.

Zunächst sollten Sie eine Datenbank mit Geodaten einrichten. Dazu finden Sie im Verzeichnis `Data` die Datenbank `spatial`. Restaurieren Sie diese Datenbank mit dem `restore`-Befehl. Erzeugen Sie dann auf der Relation `Distrikte` einen R-Baum-Index (nehmen wir an, in `SecondoTTYBDB`):

```
let distrikte_Gebiet = Distrikte creatertree[Gebiet]
```

Wir bestimmen das Gebiet, das zum Distrikt Magdeburg gehört:[1]

1. In der folgenden Definition eines SECONDO-Objektes ist es wichtig, den Namen mit einem Kleinbuchstaben zu beginnen. Dann kann es in der where-Klausel einer SQL-Query direkt verwendet werden; andernfalls würde es von PROLOG als Variable aufgefaßt

```
let magdeburg = Distrikte feed filter[.DName contains "Magdeburg"]
  extract[Gebiet]
```

Nun kann man über den R-Baum alle Distrikte suchen, deren Bounding-Box die Bounding-Box von Magdeburg überlappt:

```
query distrikte_Gebiet Distrikte windowintersects[bbox(magdeburg)]
  project[DName] consume
```

Eigentlich würden wir aber gerne alle Nachbardistrikte von Magdeburg finden. Dies läßt sich über eine zusätzliche Filterbedingung erreichen:

```
query distrikte_Gebiet Distrikte windowintersects[bbox(magdeburg)]
  filter[.Gebiet intersects magdeburg] project[DName] consume
```

Erweitern Sie den Optimierer so, daß die Anfrage

```
sql select dname from distrikte where gebiet touches magdeburg
```

entsprechend übersetzt, also mit einer Suche auf dem R-Baum-Index ausgeführt wird.

Aufgabe 6.3: (Optimierungsregel zur Verwendung von loopjoin)

Der Optimierer übersetzt bisher ein Join-Prädikat, das einen anderen Operator als "=" verwendet, stets in Filtern auf dem kartesischen Produkt. Hingegen werden Joins mit einem Gleichheitsprädikat (also Equijoins) auf viele Arten übersetzt.

Der `loopjoin` Operator wird bisher nur verwendet, wenn ein Index vorhanden ist. Man könnte einen `loopjoin` aber auch ganz allgemein anstelle der Kombination `product-filter` einsetzen. Z.B. läßt sich die Anfrage

```
sql select *
from [staedte as s1, staedte as s2]
where [s1:bev < s2:bev]
```

einerseits übersetzen in

```
Staedte feed {s} Staedte feed {t} product filter[.Bev_s1 < .Bev_s2]
```

und andererseits in den loopjoin

```
Staedte feed {s1} loopjoin[fun(t: TUPLE)
  Staedte feed {s2} filter[attr(t, Bev_s1) < .Bev_s2]]
```

Der `loopjoin` Operator arbeitet effizienter, da er Ergebnistupel nur dann erzeugt, wenn die Bedingung erfüllt ist. (Der `loopjoin` Operator selbst verknüpft einfach alle Tupel des äußeren Stromes mit allen Tupeln, die als Ergebnis des inneren Stromes entstehen). Insbesondere ist der `loopjoin` sehr viel effizienter, wenn Tupel z.B. `region`-Werte enthalten, da das Kopieren solcher Werte sehr teuer ist. Im `product`-Operator werden hingegen in jedem Fall zunächst alle Paare von Argumenttupeln gebildet und in Ergebnistupel kopiert, selbst wenn dann in der anschließenden `filter`-Operation die meisten dieser Tupel wegfallen.

(a) Erweitern Sie den Optimierer um eine Regel zur Verwendung des loopjoin-Operators. Behandeln Sie zunächst den Fall, daß das Prädikat die Form X < Y hat und X und Y jeweils Attributnamen sind. Weiterhin dürfen Sie annehmen, daß keine Variablennamen in der

Anfrage vorkommen, die verwendeten Relationen also keine gleichen Attributnamen enthalten und daher Umbenennen nicht nötig ist. Konstruieren Sie Beispielanfragen, die Ihr so erweiterter Optimierer tatsächlich in loopjoin übersetzt und damit ausführt.

(b) Erweitern Sie Ihre Lösung dahingehend, daß auch Variablen in der Anfrage bzw. Umbenennungen vorkommen dürfen.

(c) Erweitern Sie Ihre Lösung so, daß beliebige Operatoren vorkommen können anstelle von <.

Hinweis: Benutzen Sie dazu die Prädikate `functor` und `arg`, um herauszufinden, was der Funktor eines Terms ist (also der Operator) und was die Argumente sind. Beispiele für die Programmierung damit finden sich etwa in den Regeln für `plan_to_atom`.

(d) Schließlich wollen wir die Einschränkung fallen lassen, daß für das Prädikat `X op Y` die Argumente jeweils Attributnamen sein müssen. Statt dessen sollen beliebige Ausdrücke zulässig sein. Dabei ist das Prädikat so zu transformieren, daß der Zugriff auf das Attribut der äußeren Relation nun explizit erfolgt.

7 Erweiterung der Java-Oberfläche

Aufgabe 7.1: (Erstellung eines neuen Viewers)

Ein sehr wichtiger Typ in SECONDO ist die Relation. In dieser Aufgabe soll ein spezieller Viewer für Relationen implementiert und eingebunden werden. Innerhalb von Relationen können unterschiedliche Typen stehen. Für manche Typen ist die nested list Syntax kaum lesbar. Daher sollte die Möglichkeit bestehen, einige Typen als formatierten Text auszugeben. Z.B. könnte ein Segment mit der Listendarstellung `(segment (x1 y1 x2 y2))` durch `segment: (x1,y1) -> (x2,y2)` dargestellt werden.

(a) Implementieren Sie einen neuen Viewer, der diese Anforderungen erfüllt. Der Viewer sollte um neue Formatierungen ohne Änderungen am Quellcode erweitert werden können. Relationen mit sehr vielen Attributwerten sollen für den Viewer kein Problem darstellen.

(b) Binden Sie den Viewer in Javagui ein.

(c) Erweitern Sie Ihren Viewer um Formatierungen für alle Typen der Standardalgebra sowie für die Typen `PSTPoint`, `PSTSegment` und `PSTTriange`.

Aufgabe 7.2: (Erweiterung des Hoese-Viewers)

Erweitern Sie den Hoese-Viewer um Displayklassen für Segmente, Segmentmengen und Dreiecke.