Fachpraktikum 1590
„Erweiterbare Datenbanksysteme“

Aufgaben Phase 1

Wintersemester 2014/2015

Ralf Hartmut Güting, Dirk Ansorge, Thomas Behr, Christian Düntgen,

Simone Jandt, Markus Spiekermann

Lehrgebiet für Datenbanksysteme für neue Anwendungen, FernUniversität in Hagen

58084 Hagen

Einführung

Liebe Studierende,

dieses Dokument enthält eine Reihe von Aufgaben, die zur Einarbeitung in das SECONDO-System dienen. Inhaltlich werden einige 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 dem Aufbau des *Programmer's Guide* sehr ähnlich. Bevor Sie also eine Aufgabe angehen, sollten Sie das entsprechende Kapitel durcharbeiten.

Aufgabe 5 erfordert Grundkentnisse in PROLOG. Aufgabe 6 erfordert Java-Kennntnisse. Sie können wählen, ob Sie Aufgabe 5 oder Aufgabe 6 bearbeiten wollen. Alle anderen Aufgaben sind verpflichtend zu lösen.

Wir wünschen Ihnen viel Erfolg beim Lösen der Aufgaben.

Ihre Praktikumsbetreuung

1 Implementierung einer Algebra

In dieser Aufgabe sollen Sie Ihre erste Algebra selbst implementieren und eine vorhandene Algebra um zusätzliche Operatoren erweitern. Beachten Sie dabei insbesondere die Behandlung von Fehlerfällen. Beispiele hierfür sind:

- Division durch 0
- undefinierte Werte
- leere Ströme
- ...

Aufgabe 1.1: (Implementierung der `PSTAlgebra`)

Schreiben Sie die Algebra `PSTAlgebra`, die folgende Datentypen zur Darstellung eines Punktes, eines Liniensegmentes und 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:

- `=: pstpoint x pstpoint -> bool`
- `=: pstsegment x pstsegment -> bool`
- `=: psttriangle x psttriangle -> bool`
- `intersection: pstsegment x pstsegment -> pstpoint`
  Für zwei Segmente *a*, *b* wird mit `intersection` der Schnittpunkt berechnet. Überlegen Sie sich genau, was passieren soll, wenn sich die Segmente nicht schneiden.
- `inside: pstpoint x psttriangle -> bool`
  Dieser Operator prüft, ob sich ein Punkt innerhalb eines Dreiecks oder auf dessen Rand befindet.

Erstellen Sie die noch notwendigen Dateien (`makefile, .spec` und `.example`) und integrieren Sie die neue Algebra in SECONDO. Der =-Operator ist als überladener Operator zu implementieren.

Aufgabe 1.2: (Erweiterung der `StreamExampleAlgebra`)

Erweitern Sie die `StreamExampleAlgebra` um die Operation:

- `filterdiv: stream(int) x int -> stream(int)`
  Diese Operation lässt alle int-Werte durch, die durch die übergebene Zahl teilbar sind.

Aufgabe 1.3: (Automatisierte Tests mittels `TestRunner`)

Machen Sie sich mit dem `TestRunner` vertraut und schreiben Sie Testfälle für die zuvor implementierten Datentypen und Operatoren. Der `Testrunner` befindet sich im Verzeichnis `bin`, und die Datei `example.test` erläutert seine Funktionsweise.

Überlegen Sie sich sowohl fehlerhafte als auch korrekte Queries. Fehlerhafte Queries sind hilfreich, um zu überprüfen, dass die Type-Mapping-Funktionen auch unerwartete Eingaben korrekt behandeln.

2 Erweiterung der Relationalen Algebra

Vorbemerkung

Im Folgenden werden einige Operatoren der Relationalen Algebra (Datei `ExtRelation-C++/ExtRelationAlgebra.cpp`) als Beispiel zitiert. Generell gibt es in dieser Algebra häufig zwei Varianten von Value-Mapping-Funktionen, die durch die unten dargestellten Präprozessormakros als bedingte Übersetzungen zur Verfügung stehen.

```
#ifndef USE_PROGRESS
...<einfache Version>
#else
....<Progress-Version>
#endif
```

Die `USE_PROGRESS` Variante ist in der Regel etwas komplizierter und unterstützt Konzepte, die der Restlaufzeitschätzung einer bereits laufenden Anfrage dienen. Diese Techniken werden im Programmer's Guide beschrieben, müssen aber im Rahmen dieser Aufgabe von Ihnen nicht benutzt werden. Als Beispiel sollte daher stets die einfache Version studiert werden.

Aufgabe 2.1: (Erweiterung um den Operator `forall2`)

In relationalen Datenbanken haben Aggregatfunktionen eine besondere Bedeutung. Beispiele für solche Funktionen sind `sum` und `avg`, die die Summe bzw. den Durchschnitt über alle Werte eines Attributs einer Relation berechnen. Das Type- und Value-Mapping ist in den Funktionen `AvgSumTypeMap` und `SumValueMapping` bzw. `AvgValueMapping` implementiert. Die Selection-Funktion ist in der Funktion `AvgSumSelect` realisiert.

Hier soll nun der neue Operator `forall2` implementiert werden. Der Operator erhält einen Strom von Tupeln und den Attributnamen eines `bool`-Attributs. Der Rückgabewert ist wiederum ein `bool`. Der Operator `forall2` liefert `TRUE`, wenn alle Werte des Attributs `TRUE` sind, sonst `FALSE`.

Hinweise:

- Überlegen Sie, wie Sie das `APPEND`-Kommando in der Type-Mapping-Funktion verwenden können.
- Vergessen Sie nicht, die Spezifikation des neuen Operators im `.spec`- und im `.example`-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.

Eine datenbanktaugliche Implementierung darf natürlich für die Hashtabelle nur eine begrenzte Menge Hauptspeicher verwenden. Da eine persistente Implementierung einer Hashtabelle aber recht aufwändig ist, genügt es, die Hashtabelle als ein Array von Objekten der Klasse `TupleBuffer` (siehe Datei `Relation-C++/RelationAlgebra.h`) darzustellen. Diese können so initialisiert werden, dass die enthaltene Tupelmenge ab einer bestimmten Größe automatisch ausgelagert wird.

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`. Um den Hashwert eines Tupels zu berechnen, kann z.B. die Summe der Hash-Werte der enthaltenen Attribute verwendet werden.
- Ein Beispiel für die Verwendung von Hash-Funktionen finden Sie im `hashjoin`-Operator.

Aufgabe 2.3:

Benutzen Sie den `TestRunner` und schreiben Sie Testspezifikationen, die Ihre Implementierung überprüfen.

3 Benutzung des DBArray

Aufgabe 3.1: (Erweiterung der `PSTAlgebra`)

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

(b) Schreiben Sie ein kleines Programm (oder einen Operator), welches Ihnen große `pstsegments`-Objekte zum Testen ihrer Implementierung generieren kann.

(c) Führen Sie ein neues Prädikat `contains: pstsegments x pstsegment -> bool` ein, welches überprüft ob ein `pstsegment`-Objekt in einem `pstsegments`-Objekt enthalten ist.

(d) Schreiben Sie Testfälle für die neu implementierten Operatoren als weitere `TestRunner`-Spezifikationen.

4 Einbettung von Algebren in die Relationale Algebra

(a) Erweitern Sie die Klassen der `PSTAlgebra`, so dass die entsprechenden Typen in Relationen verwendet werden können.

(b) Schreiben Sie weitere Testfälle, die die Datentypen und Operationen der `PSTAlgebra` innerhalb von Tupeln testen.

5 Erweiterung des Optimierers

Diese Aufgabe erfordert Grundkenntnisse von PROLOG. Sie ist optional wählbar (siehe Einführung).

Der Optimierer in SECONDO ist im Laufe der Jahre durch zahlreiche Erweiterungen umfangreich und unübersichtlich geworden. Deshalb gibt es seit kurzem eine einfache Version, die zu den Anfängen zurückkehrt, aber schon alle wesentlichen Konzepte beinhaltet, den *Basic Optimizer*. Für den Anfang ist es leichter, sich in dieser Version zurecht zu finden. Deshalb sollen Sie die folgenden Aufgaben im *Basic Optimizer* lösen (Verzeichnis `secondo/OptimizerBasic`).

Diese Version funktioniert ganz ähnlich wie der aktuelle Optimierer. Anstatt ins Verzeichnis `Optimizer` zu wechseln, wechseln Sie in das Verzeichnis `OptimizerBasic`. Dort kann dann `SecondoPL` gestartet werden wie üblich.

Es gibt in der Benutzung zwei kleine Unterschiede zum Standard-Optimierer:

- Zum Öffnen einer Datenbank schreibt man

  ```
  open 'database opt'.
  ```

  anstatt

  ```
  open database opt.
  ```

- SecondoPL wird beendet mit `halt.` anstatt mit `quit.`

Zu dieser Version des Optimierers passende Kapitel des User Manual und des Programmer's Guide finden sich ebenfalls im Verzeichnis `OptimizerBasic`.

Aufgabe 5.1: (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 5.2: (Einbau `between`-Operator)

Es gibt im SECONDO-System einen Operator `between`, der überprüft, ob ein Argument innerhalb eines gegebenen Intervalls von Werten des gleichen Typs liegt. Z.B. würde

```
query 8 between[5, 10]
```

den Wert `TRUE` liefern. Selbstverständlich kann man diesen Operator in Filterbedingungen auf Relationen verwenden, z.B. liefert (auf der `opt`-Datenbank)

```
query plz feed filter[.PLZ between[30000, 31000]] consume
```

die Orte mit Postleitzahlen zwischen 30000 und 31000.

(a) Machen Sie dem Optimierer die Syntax dieses Operators bekannt, sodass die Anfrage

```
select * from plz where between(plz, 30000, 31000)
```

in die obige Form übersetzt wird.

(b) Eine solche `between`-Bedingung kann auch in eine Bereichsanfrage auf einem B-Baum übersetzt werden, falls ein entsprechender Index existiert. Auf `plz` existiert bereits ein B-Baum über dem Attribut `PLZ`. Damit kann die Anfrage auch so ausgeführt werden:

```
query plz_PLZ_btree plz range[30000, 31000] consume
```

Erweitern Sie den Optimierer so, dass diese Auswertungsmöglichkeit für die o.g. Anfrage erzeugt und verwendet wird, falls ein Index existiert.

6 Erweiterung der Benutzeroberflächen

Aufgabe 6.1: (Erweiterung der textbasierten Schnittstelle)

Implementieren und registrieren Sie Displayfunktionen für `SecondoTTY` sowie `SecondoTTYCS` für die Datentypen `pstpoint`, `pstsegment` sowie `psttriangle`.

Aufgabe 6.2: (Erweiterung des `HoeseViewers`)

Erweitern Sie den `HoeseViewer` um Displayklassen für alle vier Datentypen der `PSTAlgebra`. Graphische Objekte sollen dabei abhängig von der Größe der Fläche von Dreiecken dargestellt werden können, d.h. die Displayklasse für Dreiecke soll das Interface `RenderAttribute` implementieren.

Aufgabe 6.3: (Implementierung eines Viewers)

Erstellen Sie für die `Javagui` einen neuen Viewer. Dieser soll Objekte formatiert in Textform (ähnlich wie in den textbasierten Schnittstellen) darstellen können. Existiert für einen Datentyp eine spezielle Darstellungsform (Displayfunktion in der textbasierten Schnittstelle), so soll diese verwendet werden, ansonsten wird das Objekt als verschachtelte Liste dargestellt. Implementieren Sie „Displayfunktionen“ für die Datentypen der `PSTAlgebra`, für `string`, `int`, `bool`, `real` sowie für Relationen.