Fachpraktikum 1590
„Erweiterbare Datenbanksysteme“

Aufgaben Phase 1

Wintersemester 2010/2011

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 6 erfordert Grundkentnisse in PROLOG. Sie ist optional wählbar, das heißt, Sie können diese Aufgabe anstelle der Aufgabe 7 lösen.

Wir wünschen Ihnen viel Erfolg beim Lösen der Aufgaben.

Ihre Praktikumsbetreuung

1 Benutzung der SECONDO-Benutzerschnittstellen

Bitte, notieren Sie zu den Aufgaben 1.1 bis 1.3 Ihre Eingaben und Ihre Ergebnisse und bringen Sie die Notizen zur 1.Präsenzphase mit.

Aufgabe 1.1: (Die Konsole `SecondoTTYBDB`)

Starten Sie SECONDO in der Konsolenversion für einen einzelnen Benutzer.

- Lassen Sie sich die vorhandenen Datenbanken anzeigen. Ist die Datenbank `OPT` vorhanden? Wenn nicht, erzeugen Sie diese aus der Datei `opt`. (Die Dateien `opt` und `berlintest` befinden sich im Verzeichnis `secondo/bin`.)
- Öffnen Sie die Datenbank `OPT`. Welche Daten sind dort abgelegt?
- Beantworten Sie durch Abfragen folgende Fragen:
  - Wieviele Datensätze umfasst die Relation `Orte`?
  - Wie groß ist die Gesamtbevölkerung der abgespeicherten Orte?
  - Welcher Ort hat die meisten Einwohner?
  - Welche Orte gehören zum Kennzeichen „ME“ und wie groß ist die Gesamtbevölkerung dieser Orte?
  - Wieviele Postleitzahlen gehören zu Orten mit weniger als 100.000 Einwohnern?
- Erstellen Sie eine neue Datenbank `TEST`, die eine Relation mit folgendem Inhalt enthält:

Name	Matrikelnummer	Kurs
Meier	123	1590
Mueller	344	1590
Meier	123	1810
Schulze	516	1623
Becker	954	1810

  - Ermitteln Sie für jeden Teilnehmer die belegten Kurse und für jeden Kurs die Teilnehmer.

Aufgabe 1.2: (Der Optimierer `SecondoPL`)

Starten Sie die Konsole des Optimierers.

Beantworten Sie folgende Fragen zur Datenbank `OPT`. Achten Sie dabei auf die Ausgaben des Optimierers bei der Berechnung des Zugriffsplans (der ausführbaren Anfrage):

- Wieviele Einwohner hat „Hagen“?
- Welche Städte haben mehr als 1 Million Einwohner?
- Welche Ortsnamen enthalten „burg“ oder „bach“?
- Finden Sie die 20 Kennzeichen heraus, denen die meisten Orte zugeordnet sind. Wieviele Orte sind es jeweils pro Kennzeichen und wie groß ist jeweils die Gesamtbevölkerung pro Kennzeichen?

Aufgabe 1.3: (Die graphische Benutzerschnittstelle `sgui`)

Starten Sie die graphische Benutzerschnittstelle und den Optimierer von SECONDO. (Vergessen Sie nicht, vorher den `SecondoMonitor` und den `SecondoListener` zu starten!)

- Lassen Sie sich die vorhandenen Algebren, Typ-Konstruktoren und Operatoren anzeigen. Vergleichen Sie dabei die Anzeige zu den Konstruktoren und Operatoren mit den im Bereich Algebraerstellung des Programmer's Guide vorgestellten Spezifikationsangaben.
- Lassen Sie sich die vorhandenen Datenbanken anzeigen. Wenn die Datenbank `BERLIN` nicht vorhanden ist, erstellen Sie diese aus der Quelldatei `berlintest.`
- Lassen Sie sich die Straßen von Berlin anzeigen.
- Laden Sie für die folgenden Anfragen die Darstellungs-Kategorien `BerlinU.cat` in den `Hoese-Viewer`.
- Ermitteln Sie dann die Straßen, deren Namen „allee" enthalten und heben Sie sie im `Hoese-Viewer` hervor. Klicken Sie auf einzelne Straßen und beachten Sie die Anzeigenänderung im linken Teilfenster. Wählen Sie im linken Teilfenster einzelne Datensätze aus und beobachten Sie die Anzeige im rechten Teilfenster.
- Lassen Sie sich die U-Bahn Strecke der „U6" anzeigen (Relation `UBahn`). Lassen Sie sich zusätzlich die Hinfahrten der Züge der Linie 6 anzeigen (Relation `Trains`, `up = true` entspricht der Hinfahrt).[1] Probieren Sie dabei verschiedene Geschwindigkeitseinstellungen aus.
- Blenden Sie die hervorgehobenen „allee"-Straßen aus der Ansicht so aus, dass sie sie jederzeit mit einem Klick wieder anzeigen lassen können, ohne die Abfrage zu wiederholen.

Betrachten Sie die oben angegebenen Abfragen jeweils in verschiedenen Viewern, um sich mit den unterschiedlichen Möglichkeiten der graphischen Oberfläche vertraut zu machen. Benutzen Sie für Anfragen (auch) den Optimierer.

1. Bei der erstmaligen Benutzung wird sich der Optimierer beschweren, dass er Samples für die Relation "Trains" nicht automatisch erstellen kann. Geben Sie dann in der Javagui den Befehl "`optimizer createSamples('Trains', 100, 50)`" ein (siehe auch SECONDO User's Manual Abschnitt 8.5).

2 Implementierung einer Algebra

Aufgabe 2.1: (Implementierung der `PSTAlgebra`)

Schreiben Sie die 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:

- `=: 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.
- `inside: pstpoint x psttriangle -> bool`
  Die Operation `inside` gibt `TRUE` zurück, wenn sich der Punkt innerhalb des Dreiecks oder auf seinem Rand befindet.

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

Aufgabe 2.2: (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.
- `sum2: stream(int) -> int`
  Die Summe aller Zahlen im Stream wird mit `sum2` berechnet.

Aufgabe 2.3: (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.

3 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 Präprozessor Makros 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 3.1: (Erweiterung um die Operatoren `forall` und `exists`)

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 sollen nun die beiden neuen Operatoren `forall` und `exists` implementiert werden. Beide Operatoren erhalten einen Strom von Tupeln und den Attributnamen eines `bool`-Attributs. Der 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.
- Vergessen Sie nicht, die Spezifikation der neuen Operatoren im `.spec`- und im `.example`-File anzugeben.

Aufgabe 3.2: (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 3.3: (Implementierung der `replace`-Operation)

In der relationalen Algebra von SECONDO gibt es einen `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.

Hinweise:

- Schauen Sie sich die Implementierung des `extend`-Operators in der Datei `ExtRelationAlgebra.cpp` an.
- Denken Sie daran, dass der Ergebnistyp der übergebenen Funktion zum zu ersetzenden Attribut passen muss.
- Achten Sie auf eine korrekte Verwendung der Referenzzähler für `Tuple`- und `Attribute`-Instanzen.

Aufgabe 3.4:

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

4 Benutzung des DBArray

Aufgabe 4.1: (Erweiterung der `PSTAlgebra`)

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

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

Aufgabe 4.2: (Komplexere Operationen)

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

(b) Schreiben Sie ein kleines Programm, welches Ihnen große `pstsegments`-Objekte zum Testen ihres Operators generieren kann.

Aufgabe 4.3: (Tests)

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

5 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.

6 Erweiterung des Optimierers

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

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: (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. Dies wäre also eine mögliche Auswertungsstrategie für die Anfrage

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

(a) Überprüfen Sie, ob dem Optimierer die Syntax dieses Operators bereits bekannt ist.

(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. Überprüfen Sie dabei, ob der `range`-Operator schon im Optimierer bekannt und korrekt implementiert ist.

(c) Schließlich wollen wir in der Lage sein, den *between*-Operator auch in Joinbedingungen zu verwenden und eine bessere Implementierung als die ineffiziente über den `symmjoin` errei-

chen (der `symmjoin` ist eine Variante des Nested-Loop-Join, die alle Paare von Tupeln prüft und deshalb quadratische Laufzeit hat). Das heißt, z.B. die folgende Anfrage sollte effizient ausgeführt werden:

```
select *
from [plz as p1, plz as p2]
where [p1:ort = “Hagen, between(p2:plz, p1:plz - 5, p1:plz + 5)]
```

Dazu sollte der `range`-Operator auch zur Implementierung eines solchen Verbundes eingesetzt werden, falls ein entsprechender Index existiert.

7 Erweiterung der Benutzeroberflächen

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 dürfen 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 `psttriangle`.

Aufgabe 7.2: (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.

Aufgabe 7.3: (Erweiterung der textbasierten Schnittstelle)

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