Practical Course on „Extensible Database Systems“

Programming Tasks for SECONDO

2007/2008

Ralf Hartmut Güting, Thomas Behr, Simone Jandt, Markus Spiekermann

Chair for Database Systems for New Applications, FernUniversität in Hagen

58084 Hagen, Germany

Introduction

Dear students,

in this document you will find some exercises which will help you to get familiar with the SECONDO system. In these exercises, all important parts of the system are addressed and, actually, some of them have to be extended by you. After having finished all tasks, you will have gained a lot of knowledge about SECONDO and then you will be able to solve more complex tasks. In the second part of the practical course, this knowledge is required.

There are two additional documents which you will need for these exercises, namely the *Secondo User Manual* and the *Secondo Programmer's Guide*. As you can easily see, the structure of the *Programmer's Guide* is very similar to the structure of this document. Therefore, you should read the appropriate section of that document before you address an exercise.

Exercise 6 requires basic knowledge of PROLOG. Because of this, exercise 6 can optionally be selected instead of exercise 7.

Have fun with these exercises!

Scientific staff of the chair for Database Systems for New Applications

1 Using SECONDO-Userinterfaces

Exercise 1.1: (The console `SecondoTTYBDB`)

Start SECONDO in the single user consol version.

- Have a look at the existing databases. Does the database `OPT` exist? If not, please restore it on your system. (You can find the files `opt` and `berlintest` in directory `secondo/bin`.)
- Open the database `OPT.` What kind of data is stored there?
- Examine the data of `OPT` to answer the following questions:
  - How many records are stored in the Relation `ORTE?`
  - How much is the total population number of the towns stored in the database?
  - Which town has the biggest population?
  - Which towns have the license plate „ME“ and how many people live there?
  - Which postal codes belong to towns with less than 100,000 people?
- Create a new database `TEST` with the following relation:

Name	Matrikelnr	Course
Meier	123	1590
Mueller	344	1590
Meier	123	1810
Schulze	516	1623
Becker	954	1810

  - Determine for each student the selected courses and for each course the subscriber’s of the course.

Exercise 1.2: (The Optimizer `SecondoPL`)

Start the console of the optimizer.

Examine again the database `OPT` to answer the following questions. Please pay attention to the optimizer output while computing the query plan:

- What is the population number of „Hagen“?
- Which towns have more than 1,000,000 residents?
- Which towns contain „burg“ or „bach“?
- Which 20 licence plates have the most associated towns? How many towns are associated with each of the licence plates given above and what is the total population of all of the towns belonging to each licence plate?

Exercise 1.3: (The graphical user interface `sgui`)

Start the graphical user interface and the optimizer server of SECONDO. (Don't forget to start the `SecondoMonitor` and the `SecondoListener` before!)

- Have a look at the existing algebras, type constructors, and operators. Compare the information shown on the display with the specification data shown in the `Programmer's Guide.`
- Check again which databases are already present. If the database `BERLIN` does not exist restore it using the file `berlin` stored at `SECONDO_INSTALLATION_KIT/data`.
- Have a look at the street map of Berlin (Relation „*strasse*“).
- Load the presentation categorie `BerlinU.cat` into the `HOESE_VIEWER` to use it for the following questions:
  - Which street names contain „allee“? Show them coloured in the `HOESE_VIEWER` within the streets of Berlin. Select a few example streets by mouse click on the street map and observe the changes of the display in the left part of the window. Now select a few example data sets in the left part of the window and observe the changes in the street map displayed on the windows right hand side.
  - Add the track of the underground railway „U6“ (Relation `UBahn`) to your view of Berlin and show also the outward journeys of the trains on line 6 (Relation `Trains, up = true`)[1]. Test different settings of the animation speed.
  - Hide the emphasis of the streets containing „allee“ in their name in the display so that you can always show them again by one mouse click without repeating the query.
- Have a look at the data representation of the queries above in the different viewers offered by the graphical user interface of SECONDO to learn about the different options available in SECONDO. Use for your queries also the optimizer.

2 Implementing an Algebra

Exercise 2.1: (Implementation of the `PSTAlgebra`)

Implement a C++-algebra `PSTAlgebra`, providing data types for the (2D-) representation of a point, a line segment and a triangle (using reals for the coordinates):

- `pstpoint`
- `pstsegment`
- `psttriangle`

Furthermore, some operations for these data types have to be implemented. They are listed below (sorted by signature):

1. When you use it the first time, the optimizer will complain, that he it cannot create the samples for the relation „Trains“ automatically. Create the samples manually typing the command „optimizer createSamples('Trains',100,50)“ in the Javagui (see also the SECONDO User Manual Section 8.5).

- `equal: pstpoint x pstpoint -> bool`
- `equal: pstsegment x pstsegment -> bool`
- `intersection: pstsegment x pstsegment -> pstpoint`
  Computes the intersection point for two segments *a* and *b*.
- `equal: psttriangle x psttriangle -> bool`
- `inside: pstpoint x psttriangle -> bool`
  Returns `TRUE`, if the point lies inside of the triangle or on its border.

Write the other essential files (`makefile, .spec and .example`) and link your new algebra to the SECONDO system.

Exercise 2.2: (Extending the `StreamExampleAlgebra`)

Extend the `StreamExampleAlgebra` with the following operations:

- `filterdiv: stream(int) x int -> stream(int)`
  This operation filters all values which are not divisible (without remainder) by the passed number.
- `sum: stream(int) -> int`
  Computes the sum of all stream values.

Exercise 2.3: (Automatic tests using `TestRunner`)

Learn how to use the `TestRunner` and write test cases for the data types and operators implemented before. The `TestRunner` files are located in the `bin` directory. In the `example.test` file it is described how the `TestRunner` works.

When writing test cases, do not only write correct queries but also wrong ones. This might be helpful to identify incorrect type mapping functions.

3 Extension of the Relational Algebra

Preliminary Remark

Below some operators of the Relational Algebra (file `ExtRelation-C++/ExtRelationAlgebra.cpp`) are given as a reference. In general, this algebra often provides two variants of value mapping functions whose compilation depends on the following preprocessor macros.

```
#ifndef USE_PROGRESS

...<simple version>

#else

....<progress version>

#endif
```

The `USE_PROGRESS` variant is a little bit more sophisticated since it supports concepts used for the progress estimation of a running query. Those techniques are not yet presented in the Programmer’s Guide. Thus always study the simple version as an example.

Exercise 3.1: (Implementation of the `forall` and `exists` operators)

In relational databases aggregate functions have a special meaning. Examples of such functions in the algebra `C++-Relation` in SECONDO are `sum` and `avg`, which compute the sum (average) over all values of a relation's attribute.

The two new aggregation operators to be implemented are called `forall` and `exists`. Both operators receive a stream of tuples and an attribute name of type `bool` as input and they return a `bool` value. The operator `forall` returns `TRUE` when all attribute values are `TRUE`, otherwise it returns `FALSE`. On the other hand, the operator `exists` returns `TRUE` if there is at least one tuple for which the attribute value is `TRUE`, otherwise `FALSE` is returned.

Notes:

- Consider how you can use the `APPEND` command in the type mapping function.
- Do not forget to indicate the specification of the new operators in the `.spec` File.

Exercise 3.2: (Implementation of the `rdup2`-operator)

In order to remove duplicate tuples, there is an `rdup`-operator in the relational algebra of SECONDO. To obtain correct results, it is necessary to pass a sorted stream to this operator. Now an operator `rdup2` has to be implemented. It does not have this requirements, but instead it recognizes duplicate values utilizing hashing and removes them. In the value mapping function for each tuple a hash value used as index for a table is computed. Then the tuple is entered into a hash table and can be returned into the result stream if it was not yet there.

An implementation suitable for a database system should only use a limited amount of memory for its hash table. Since a persistent implementation of a hash table is an elaborate task, just use an array of `TupleBuffer` (see file `Relation-C++/RelationAlgebra.h`) instances instead. These can be initialized to swap out their contained tuple set if a given size is exceeded.

Notes:

- For each attribute, which can be used in relations, there is a hash function. This is implemented in the class `Attribute` and is called `HashValue`. If duplicate tuples are to be removed, e.g. the sum of the hash values can be used as an address for the hash table.
- You can find an example of the use of hash functions in the `hashjoin`-operator.

Exercise 3.3: (Implementation of the `replace`-operator)

The relational algebra of SECONDO offers an `extend`-operator, which extends tuples by new attributes that are computed over existing ones. A similar operator has to be implemented now, but instead of appending a new attribute, it shall replace an old one. This operator receives a stream of tuples, the name of the attribute where values will be replaced, and a function determining how the new value is calculated.

Notes:

- Take a look at the implementation of the `extend` operator.
- Remember that the function's result type must fit the attribute that will be replaced.

- Take care for a correct usage of reference counters for the `Tuple`- und `Attribute`-instances, which are modified by the methods `IncReference`, `DecReference`, `Delete-IfAllowed`, ...

Exercise 3.4: (Automatic tests using `TestRunner`)

Write some test cases for the `TestRunner` to check your implementation.

4 Use of DBArray

Exercise 4.1: (Extending the `PSTAlgebra`)

(a) Extend the algebra adding a new type constructor `pstsegmentset`, which represents a set of segments.

(b) Introduce a new predicate `contains: pstsegment x pstsegmentset -> bool`, which examines if a `pstsegment` object is a member of a `pstsegmentset`.

Exercise 4.2: (More Complex Operations)

(a) Implement the operation
`intersection: pstsegmentset x pstsegmentset -> pstsegmentset`,
computing all segments contained in both `pstsegmentset` objects.

(b) Write a small program that generates large `pstsegmentset` objects and that imports them into a database, in order to accomplish tests with large objects.

Exercise 4.3: (Tests)

Write test cases for the newly implemented operators as a test file that can be executed with the `TestRunner`.

5 Embedding of Algebras into the Relational Algebra

(a) Extend the classes of the `PSTAlgebra` to support the usage of its instances as attributes in relation objects.

(b) Write further test cases that check the data types and operations of the `PSTAlgebra` within tuples. Take care that all of the Attribute functions will be called, e.g. `sortby` will call the `Compare` function and a `hashjoin` needs the `HashValue` function.

6 Extension of the Optimizer

This exercise requires some basic knowledge of PROLOG. In our practical project, it is optional; one exercise 7 can be replaced by this one.

Exercise 6.1: (Display Rules for Type Constructors)

(a) Write `display`-rules for representing the type constructors `pstpoint`, `pstsegment`, and `psttriangle` introduced by Exercise 1.

(b) The `ArrayAlgebra` offers a type constructor `array`, whose argument can be any data type. For example, one can create an array of integer values:

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

With the `loop`-operator one can evaluate an expression for each element of the array:

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

The result is an array of boolean values.

Note that arrays can also be formed by relations. For instance, the `distribute`-operator can be used to distribute the tuples of a relation into several buckets. Each bucket, which is a slot of an array, contains a subset of the relation. A query returning an array of relations can be expressed as follows:

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

Write `display`-rules for the representation of arrays. A good representation could be, for example:

```
----------    1 ----------
<presentation of the 1st element>
----------    2 ----------
<presentation of the 2nd element>
...
----------    n ----------
<presentation of the last element>
```

Exercise 6.2: (Adding the `between` Operator)

The SECONDO system has an operator `between` which checks whether an argument is contained in a given interval of values. For example, the query

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

would yield the value `TRUE`. Of course, this operator can be used in filter predicates on relations. For example, on the `opt` database, the query

```
query Orte feed filter[.BevT  between[30, 40]] consume
```

returns “Orte” (cities, small cities) with population between 30000 und 40000. This operator is not yet known to the optimizer.

(a) Explain the syntax of this operator to the optimizer, so that the query

```
select * from orte where between(bevt, 30, 40)
```

is translated into the form shown above.

(b) Such a `between`-condition can also be translated into a range query on a B-tree, if a corresponding index exists. For example, one could create a B-tree index on attribute `BevT` by saying

```
let orte_BevT = Orte createbtree[BevT]
```

Then the same query could be executed as follows:

```
query orte_BevT Orte range[30, 40] consume
```

Extend the optimizer in such a way that also this option for evaluating the selection is generated. Note that also the `range` operator is not yet known to the optimizer.

7 Extending Javagui

Exercise 7.1: (Implementation of a new Viewer)

The data type `relation` is a very important type in SECONDO. In this task, a special viewer for displaying relations should be implemented and embedded into the system. A relation can contain different types as attributes. For a lot of them the nested list structure is not readable for humans. Therefore, it should be possible to show some types as formatted text. For instance a segment with the list representation `(segment (x1 y1 x2 y2))` could be represented as
`segment: (x1, y1) -> (x2, y2)`.

(a) Implement a new viewer, which fulfills these requirements. The viewer should be able to be extended by new formatting rules without changes at the source code. It should be possible to display relations containing many attributes.

(b) Build in the viewer into Javagui.

(c) Extend the viewer by display rules for all types of the `StandardAlgebra` as well as the types `pstpoint`, `pstsegment` and `psttriangle`.

Exercise 7.2: (Extension of the Hoese-Viewer)

Extend the Hoese-Viewer by display classes for the types `pstpoint`, `pstsegment`, `pstsegmentset`, and `psttriangle`. It should be possible to represent spatial objects depending on the size of a `psttriangle`.

Implement and register display functions for the datatypes `pstpoint`, `pstsegment` and `psttriangle`.