Practical Course on „Extensible Database Systems“

Programming Tasks for SECONDO

2006/2007

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

Chair for Database Systems for New Applications, FernUniversität in Hagen

D-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.

Have fun with these exercises!

Scientific staff of the chair PI4

1 Implementing an Algebra

Exercise 1.1: (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):

- `equal: PSTPoint x PSTPoint -> bool`
- `equal: PSTSegment x PSTSegment -> bool`
- `intersects: PSTSegment x PSTSegment -> bool`
  This operation returns `TRUE`, if the intersection point of two segments is a point. Thus, this does not hold for e.g. two overlapping segments.
- `intersection: PSTSegment x PSTSegment -> PSTPoint`
  For two segments *a*, *b*, with `intersects(a,b) = TRUE`, `intersection` computes the intersection point.
- `equal: PSTTriangle x PSTTriangle -> bool`
- `inside: PSTPoint x PSTSegment -> bool`
  Returns `TRUE`, if the point lies inside of the segment.
- `inside: PSTPoint x PSTTriangle -> bool`
  Returns `TRUE`, if the point lies inside of the triangle or on its border.
- `intersects: PSTSegment x PSTTriangle -> bool`
  Returns `TRUE`, if the segment intersects at least one of the triangle's border segments or if the segment completely lies inside of the triangle.

Exercise 1.2: (Linking the Algebra to SECONDO)

Write the `make` and `.spec` files and link your new algebra to the SECONDO system.

Exercise 1.3: (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 1.4: (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.

2 Extension of the Relational Algebra

Exercise 2.1: (Implementation of the `forall` and `exists` operators)

In relational databases aggregate functions have a special meaning. Examples of such functions in the algebra `Relation-C++` 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 2.2: (Implementation of the `rdup2`-operator)

In order to remove duplicate tuples, there is an `rdup`-operator in the relational algebra of SECONDO. So that this operator supplies correct results, it is necessary to hand over a sorted stream. Now an operator `rdup2` has to be implemented. It does not have this condition anymore, and 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 registered into a hash table and can be returned into the result stream if the index is not used already.

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 2.3: (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 which values will be replaced, and a function determining how the new value is calculated. Pay attention that the computed value must fit the attribute's type.

Notes:

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

Exercise 2.4: (Automatic tests using `TestRunner`)

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

(b) Make sure that your implementation has no memory leaks. Activation of the flags `SI:Printcounters` and `SI:RelStatistics` in the `SecondoConfig.ini` file may be helpful.

3 Use of DBArray

Exercise 3.1: (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 3.2: (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 3.3: (Tests)

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

4 Embedding of Algebras into the Relational Algebra

(a) Extend the C++ 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.

5 Storage Manager Interface (SMI)

Exercise 5.1:

(a) Get familiar with the Interface of the SMI by studying the header file `SecondoSMI.h`.

(b) Write a program that stores large datasets in `SmiFiles`. For this purpose read in pictures and store them in records of variable length.

6 Extension of the Optimizer

This exercise requires some basic knowledge of PROLOG. In our practical project, it is optional; one of the exercises 2 or 5 can be replaced by this one.

Exercise 6.1: (Display Rules for Type Constructors)

(a) Write `display`-rules fo 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 exist the possibility 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) Built 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`.

Exercise 7.3: (Extension of the SecondoTTY)

Implement and register display function for the datatyes `PSTPoint`, `PSTSegment`, and `PSTTriangle`.