CS740 Project

Fall 2001

Part 2

Abstract

The goal of the project is to develop an extensible single-user (at this point) relational database system in JAVA.

Relational Algebra and Query Processor

The goal of the second part of the project:

1. implement *sufficient number* of relational algebra operators based on the same `Iterator` interface (here we use the file I/O from part 1.).

2. implement a *query processor*—an user interface that reads (text representation of) a set of assignment statements of an expression trees specifying a physical plan for query to file representation of relations of the form

$$R := E(R_1, \ldots, R_k)[p_1, \ldots, p_k],$$

   where R_i are access methods to relations (stored in files) and $[p_1, \ldots, p_k]$ is a parameter tuple. The query processor verifies the validity of the sequence (as much as it can) and executes it creating new relations (files) and/or printing results.

As everyone is using the same `Iterator` interface, you may *trade* th individual relational operations, as long as the authors of the code are acknowledged.

Here is the generic template for the operations:

```
class Operation implements Iterator {
    public Operation(init_params) { ... }

    public void open(Tuple param) { ... }
    public Tuple fetch()          { ... }
    public void close()           { ... }
}
```

where the `Operation` is one of (the constructors with appropriate parameters are listed instead):

1. `Plus(Param1,Param2)`, `Minus`, `Times`, etc.
   virtual tables representing arithmetic operations;
2. `Less(Param1,Param2)`, `Equal(Param1,Param2)`
   virtual tables representing order and comparison operations;
3. `Project(Iterator input, Attr_list)`
   the projection operation, `Attr_list` specifies which attributes are to be retained by the operation;
4. `CountAggregate(Iterator input, Group_list)`
   the grouping operation, `Group_list` specifies which attributes are used for grouping in the operation, an additional `count(*)` attribute is added. The operation may assume the input has been sorted by the grouping attributes;
5. `LoopJoin(Iterator input1, Iterator input2, Join_condition)`
   a nested loop join; the parameters passed to the inner (right) subexpression is the combination of the original parameter tuple and the tuple obtained from the outer (left) subexpression.
6. `MergeJoin(Iterator input1, Iterator input2, Join_condition)`
   a merge join; the operator may assume that its inputs have been ordered appropriately to support the join condition.
7. `LoopExceptIn(Iterator input1, Iterator input2, Join_condition)`,
   `MergeExceptIn(Iterator input1, Iterator input2, Join_condition)`
   similarly to `LoopJoin` and `MergeJoin`, these are implementations of set difference (duplicate-preserving for the left operand). The `LoopExceptIn` variant allows passing parameters from the outer to the inner.
8. `CatUnion(Iterator input1, Iterator input2)`
   union operation based on concatenation;
9. `MergeUnion(Iterator input1, Iterator input2)`
   union operation based on merging ordered inputs (again the operation may assume that the inputs have been ordered appropriately);

In addition to the parameters above (needed to make the operations work, you may want to add *tuning* parameters (e.g., to specify number of buffers used by the operation, etc.)

Query Processor

The query processor should read commands from *stdin* using the grammar

```
cmd ::= lhs ":=" rhs "[" parameter* "]" | "<end-of-file>"
lhs ::= writer_name "[" parameter* "]"
rhs ::= operation_name "[ parameter* "](" rhs* ")"
```

using the usual ways to delimit literal constants, skip white space, etc.