Project 4 - Design and Implementation Summary

CS222: Principles of Data Management

Christopher A. Wood (woodc1 - 54168394)
Tamir Husain (thusain - 40206791)
March 17, 2014

Task

Write a report to briefly describe the design and implementation of your query engine.

Summary

This document describes the index-centric RM component extensions that were needed to support the QE (Query Engine). We then describe the design and implementation of each major component (relational algebra operator interface) in the query engine, highlighting special design decisions where necessary.

RM Component Extension

In order to leverage the both tables and indexes from the RM component in the query engine, we had to extend the RM component to support basic maintenance routines for indexes. In particular, we had to store and remove index information in the system catalog tables upon index creation and deletion so as to persist indexes alongside tables on disk.

We also needed to implement the ability to scan across an index in a manner similar to scanning over an entire relation table. Since the query engine uses both indexes and tables when evaluation relational algebra operators, creating, destroying, and scanning indexes and tables was the only required functionality needed for this project.

Query Engine Design

Using the predefined and implemented `TableScan` and `IndexScan` interfaces base Iterator interface, we implemented the remaining relational algebra operators Filter, Filter, Join, and Aggregate in the natural way:

1. Filter uses an input iterator and condition to scan the relation (either a table or index), applying the condition on each tuple to determine the next viable candidate tuple in every `getNextTuple` call, and returns the result. An appropriate error is returned when the end of the index is reached.
2. Project uses an input iterator and subset of attributes for the relation to scan the entire relation, extracting the data fields that only correspond to those identified in the attribute name list, and returns the result in every `getNextTuple` call. As before, an appropriate error is returned when the end of the index is reached.

3. Both nested-loop join and index nested-loop join were generalized to use the common joining algorithm listed in Algorithm 1.

Algorithm 1: General nested-loop join algorithm.

```
for each tuple r in R:
    for each tuple s in S:
        if r and s satisfy the join condition:
            output (r || s)
```

Since this template behavior the same for both index and table nested-loop joins, where the index version differs in that the iterator over the relation S is an index instead of a table. To reduce code duplication, we use a common `Joiner` class from which both `NLJoin` and `INLJoin` inherit. The base class provides the logic for the join algorithm as shown above. The only specialized behavior in each of the subclasses is resetting the iterator. Specifically, the `NLJoin` class casts the inner iterator to a `TableScan` iterator and invokes its own reset method, whereas the `INLJoin` class casts the inner iterator to an `IndexScan` iterator and invokes its respective reset method. This class hierarchy is shown below in Figure 1.

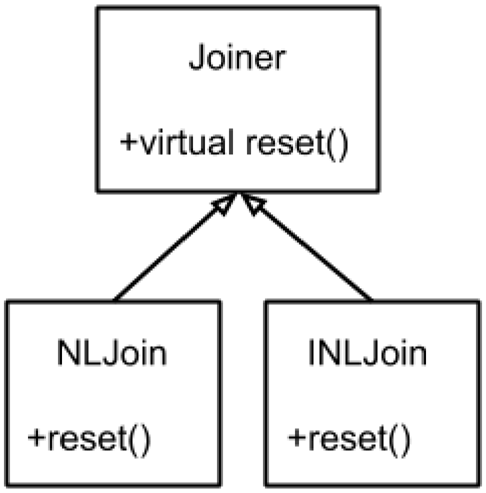

Figure 1. Joiner class hierarchy to support nested-loop and index nested-loop joins.

4. Both versions of Aggregate use an input iterator to scan a relation using some aggregation operator (`AggregateOp`). With the groupless Aggregate, a single counter is appropriately incremented on every call to `getNextTuple`. For grouped aggregates, the entire index is scanned on initialization and for every unique group value, we appropriately increment the aggregate value in a hash map for that group. The getNextTuple call for grouped aggregates will then simply iterate through the elements of this generated hash map.