

# CS44800 Project 3

## Relational Operators

### Spring 2023

**Due: March 08, 2023, 11:59PM**

Total Points: **7 points**

---

### Learning Objectives

1. Understand how Relational Algebra operators are implemented inside of a database system
2. Understand algorithms for executing different Relational Algebra operators

### Background

The goal of this project is to implement Relational Algebra operators using the Iterator interface inside a working database system. Like Project 2, this project is based on Minibase, a small relational DBMS, structured in several layers in order to allow a modular, abstract approach to implementing a DBMS

You should start reading Section 18.7 from Chapter 18 (**Strategies for Query Processing**) on the Textbook (**Fundamentals of Database Systems by Elmasri & Navathe, 7<sup>th</sup> Edition**) to get an overview of the how operators are combined in a pipeline. If needed, you should review [slides from Chapter 8](#), which covers Relational Algebra and Relational Calculus.

When executing an SQL query, a DBMS transforms the query into an equivalent Relational Algebra expression. Each Relational Algebra is implemented through an *iterator interface*. This interface consists of the following functions:

- open()
- close()
- hasNext()
- getNext()

Relational Algebra operators are then chained together to form a Query Tree. A query is then evaluated by starting with the root operator of the tree and calling **hasNext()** and **getNext()** to recursively request the next tuple from its child Iterators until all rows have been exhausted.

This results in an implementation where the results of a query are evaluated lazily and in a non-blocking fashion. Any given Iterator only requests results from its child Iterators when necessary and in most cases does not need to compute its entire set of results before generating results for the Iterators higher up the query tree – tuples simply “flow” up the query tree as they are evaluated. This *pipelined* approach results in queries that can start generating results immediately, and do not require any storage space for intermediate results.

In this project, you will be implementing Relational Algebra using this Iterator interface, creating the functionality necessary to actually execute any arbitrary query in Minibase.

You can find the API for the Minibase classes you will be using [here](#). Please familiarize yourself with the API before starting this project. In particular, the classes you will need to use are:

- classes in the *relop* package,
- HeapFile and HeapScan in the *heap* package,
- HashIndex, HashScan and BucketScan in the *index* package, and
- AttrOperator and AttrType in the *global* package.

## Part 1: Scan Operators - From Records to Tuples

All Relational Algebra operators you will be implementing in this project will be subclasses of the Iterator abstract class. The Iterator class provides a *schema* field (of protected visibility) that can store the schema of any output tuple from that Iterator, as well as defines the following methods that must be implemented:

- public void restart()
- public boolean isOpen()
- public void close()
- public boolean hasNext()
- public Tuple getNext()

The constructor for each Iterator is equivalent to the isOpen() function of the Iterator interface.

### Note: you do not have to implement the explain() method

Your first task is to construct the initial scan operators. These are Iterators that read through an entire relation or index and report one Tuple at a time. Minibase already has methods defined to scan through and retrieve records in raw byte format from the HeapFile and HashIndex classes, but we need to create Iterator versions of these scans that will act as wrappers for these scans and will return Tuples with defined schemas as opposed to the raw byte information.

Specifically, we require the following Scan operators:

- **FileScan**  
A *HeapScan* that returns *Tuples* instead of `byte[]`'s
- **KeyScan**  
A *HashScan* that returns *Tuples* instead of *RIDs*
- **IndexScan**  
A *BucketScan* that returns *Tuples* instead of *RIDs*

The FileScan implementation is already provided to you. You will need to implement similar scan operators for the KeyScan and IndexScan. Some useful hints and tips:

- Each constructor should initialize the inherited field *schema*. (i.e. it is given as a parameter to these three iterators)
- Don't be surprised by how little code these classes require
- *HashScan* scans the hash index for records having a given search key. *BucketScan* scans the whole hash index. Those classes only return RIDs. You have to build wrapper classes KeyScan and IndexScan that return Tuple.

## Part 2: Primitive Operators

Now that you have the basic leaf nodes of most query trees, you can make some more interesting iterators. Your next task is to implement the three fundamental operations of relational algebra:

- **Selection**  
Filters the output of another Iterator on a set of Predicates. When there are multiple Predicates, they are assumed to all be ORed together (AND is implemented through chaining separate Selection operators together). Please read the Project 3 Q & A post for an explanation of this.
- **Projection**  
Performs a projection on a Tuple by removing columns from the output of another Iterator. This will change the Schema of all Tuples output by the Projection. Note: you should not remove duplicates.

## Part 3: HashJoin

For the last part of the project, you are required to implement a **Hash-Join**. Please read chapter 18.4.4 from your textbook carefully before proceed with this part. Furthermore, the source code for the "Nested Loops Join" is provided for you in SimpleJoin.java. You may consider to study the source code and use it as a guidance. For example, this join was implemented using a pre-fetching approach, which we recommend you follow for the primitive operators you have to implement.

A Hash-Join consists of two stages:

- First, partition phase, each input relation is partitioned in  $k$  partitions or buckets using the same hash function. Luckily, you do not have to implement most of this from scratch – *you can make use of the IndexScan within Minibase to perform this.*
- Then, probing phase, for each bucket on the smaller input relation, we build an in-memory hash table with its tuples. Next, this hash table is probed using the tuples from same bucket in the other relations. This process is repeat with all the buckets.

For the first phase of a Hash-Join, you need to transform the input Iterators into an IndexScan with the join column as the key. **Note: You cannot simply cast the child Iterators of the HashJoin as an IndexScan. You must construct a new temporary HashIndex and HeapFile populated with the results of the child Iterator and create an IndexScan.**

Once the child Iterators have been transformed into IndexScans with the join column as the search key, you can iterate over the IndexScans to determine matching results. An IndexScan iterates over an index **bucket-by-bucket**. Note that the contents of a bucket may be in any arbitrary order and may not all be matches, even if all entries have hashed to the same bucket. Therefore, you will have to track matching tuples within a bucket as you iterate through. To facilitate this, you are provided with a class **HashTableDup**. This class acts as a HashTable, but allows for multiple identical keys to exist. A common pitfall is to not comprehend that the matching is done bucket by bucket. Therefore, a tip is to regenerate the hash table when the program has completed with searching one bucket. The rationality behind this approach is that

The benefit(s) and reasoning for using Hash Join is briefly mentioned right after the pseudocode in the next section.

## Hash-Join Algorithm

We want to join tables R and S on their common  $p$  table as follows:

$$R \bowtie_p S$$

Let's assume we want to join tables R and S with an equijoin predicate  $p$ . The algorithm is described next:

```
// Partition R into k partitions
foreach tuple  $r \in R$  do
    read  $r$  and add it to partition  $i$  in  $R$ ;

// Partition S into k partitions
foreach tuple  $s \in S$  do
    read  $s$  and add it to partition  $i$  in  $S$ 

// Probing Phase
for  $i = 1, \dots, k$  do:

    // Build in-memory hash table for  $R_i$ 
    foreach tuple  $r \in$  partition  $R_i$  do:
        read  $r$  and insert into a hash table

    // Scan  $S_i$  and probe for matching  $R_i$  tuples
    foreach tuple  $s \in$  partition  $S_i$  do:
        read  $s$  and probe hash table;
};

clear hash table to prepare for next partition;
```

Some guidelines on implementing Hash Join are provided below:

1. As mentioned earlier in the handout, your first task would be to transform both iterators to *IndexScans* to facilitate scanning through the records. By using the *IndexScan* iterator, all the records from the relation will be partitioned in buckets by applying a hash function  $h$  to each tuple. This first step is called **partition stage**.
2. Ideally you would keep the table with smaller cardinality (let's assume  $R$  is the smaller relation) hashed as key value pairs in a hash table. The other table (i.e.,  $S$  in this case) would also be probed to against this hash table.
3. The second phase is the **probing stage**, where you will match the tuples for the relations  $R$  and  $S$ , bucket by bucket. Then, you will insert the tuples from bucket  $i$  into the in-memory hash table and probe the tuples of the  $S$  of the same bucket ( $i$ ). The rationality behind is that the memory will not be enough to fit all the tuples from  $S$ ; instead, we are only keeping in memory the tuples of the bucket we are currently probing.
4. Whenever you will find a match, store the tuple and continue.

The basic advantage of Hash Join is the fact that we only need one full scan of one of the tables/relations (in our example – the  $S$  relation) to find the output. This is almost double as efficient as nested loops join (except it is only limited to equijoins). The same join can be done with nested loops join but the memory overhead and the time overhead – both would be double. Therefore, we emphasize on your correct implementation of the Hash Join for acquiring full grade.

For the technical capabilities of the hash functions and hash code comparisons, we strongly encourage you to study the *IndexScan*, *SearchKey*, *Tuple* (particularly the join method), *Schema*, *HashJoin* and *HashTableDup* classes in the Minibase javadocs. You will need all of them for the complete implementation of the project. Once you understand the structure and the requirements, the coding is quite simple. But understanding your goal and the classes at your disposal is the most important part going forward.

## Hints

Some useful hints and tips:

- For Relational Algebra operators that may not always find results (i.e. Selection and HashJoin), you need to be careful about exactly how you are determining the result of the `hasNext()` method call. Unlike with the basic scans, you cannot simply just return the result of the child iterator's `hasNext()` method. You may check the `SimpleJoin.java` class for an example of how to properly handle this.
- The join column specification for HashJoin is based on the schemas of the individual child Iterators *before* the join. For example, a HashJoin constructor call with “(...,0,2)” would mean the join columns are column 0 of the left Iterator and column 2 of the right Iterator.
- Implementing the HashJoin will likely be the most complicated part of the project. Please budget your time accordingly.

## Running Minibase

As with the previous project, you may run the test cases using the Maven based make file like you did in the previous two projects.

```
mvn clean compile assembly:single
```

Upon making the binaries, you can use the following command to run the test cases on your implementation.

```
mvn test
```

As usual, remember to run this command from the parent project directory.

Moreover, the testcases are also provided as a JUnit suite inside the test directory in **src/test/java/tests**. The testcases consist on a mix of relational operators combined through pipelines. These testcases can be used to verify the correctness of your implementation for each of the parts described in this handout. In the specific case of the **Hash-Join**, you must follow the guidelines in this handout – *iterating through the HashIndexes bucket by bucket* – to be awarded the full credits. To validate this, we will manually inspect your submission.

The input relations, used by the testcases, are defined on a pair files for each table, a JSON with the schema and CSV with the corresponding data. Likewise, a text file, `solution.txt`, includes the expected solution for each testcase. You can play around with the input data located in **/src/test/resources** and accordingly modify the resultant records in the **solution.txt** file in your parent project directory. Make sure that you update in both places correctly if you are adding in your own data. This is quite helpful for a sanity check of testing your system with queries.

This project has been tested and run through the terminal on the university Linux machines. We cannot provide support for using different IDEs or running the program on different platforms, but some posts have been made on Piazza regarding setting up IDEs to run the Minibase projects that you can reference.

## What to Turn in

1. A .zip archive of the relop folder.

**These files should be submitted on Brightspace.**