# Homework Project 4: Implementing Join Operators

CS186 Introduction to Database Systems
UC Berkeley

March 14, 2007
Due: April 10, 2006, 22:00

## 1 Introduction

Up to this point, we have been exploring the internals of a Database Management System in a bottom-up fashion. With a brief detour to the Relational Model and languages, we started describing the functionality of the storage layer and then talked about how this layer is used by the Buffer Manager to bring into memory pages from disk and write them back when necessary. Homework 1 gave you the opportunity to implement Minibase's Buffer Manager from scratch, which you then used for your second homework, an implementation of a B+ Tree index, along with its iterator, through which you could scan the tuples of a relation in the order specified by its B+ Tree.

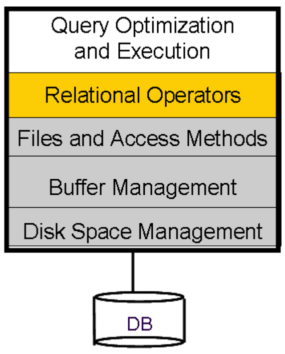

Figure 1: The layers of a DBMS.

Homework 4 intends to built upon the three DBMS layers you have partially explored, namely the *Disk Space Manager*, the *Buffer Manager* and the *Files and Access Methods* layers, and allow you to touch upon one of the most important relational operators used by a typical DBMS, the *join* operator. Figure 1 depicts an simplified version of a DBMS' architecture we have being seeing in the lectures so far. The encapsulation property implied by this figure permits each layer to interact with the ones underneath it, by using a set of higher-level interfaces, which help the implementor concentrate more on the functionality each layer is providing, rather than having to deal with issues already taken care of by lower layers of the architecture. Thus, the *Relational Operators* layer needs not be concerned with managing free-space within disk pages, (un)-pinning pages in memory, or even accessing relations in a particular way. All the interaction of the classes of this layer with the rest of the (implemented so far) system will proceed through the interface it is provided by the Files and Access Methods layer, which we will discuss in section 3.

For the purposes of this homework, you will implement the simplest version of the Nested Loops Join algorithm, presented in section 14.4.1 (page 454) of your textbooks (Ramakrishnan and Gehrke "Database Management Systems", third edition). After you get warmed up with Nested Loops, you will implement the simple version of the Hash Join algorithm, described in section 14.4.3 (pages 463–464). You can start by downloading the archive `http://inst.eecs.berkeley.edu/~cs186/sp07/homework/hw4/Homework4.zip`, which contains the source java files, the required libraries and the javadoc of the whole Minibase distribution.

We are also pleased to announce that in this final Minibase assignment, you will be working on a brand new code base, developed at Purdue, which is far more Object Oriented (and we hope easier to cope with) than the previous one!!! Please, use the javadoc provided in the `Homework4.zip` archive, or the updated API you can find online at: `http://inst.eecs.berkeley.edu/~cs186/sp07/homework/hw4/javadoc/index.html`.

## 2 Relational Operators and the Iterator Model

One of the most powerful features of relational databases is their support for declarative query languages such as SQL. With these languages, users describe the output of their queries without needing to specify *how* that query is supposed to be executed by the DBMS. It is the responsibility of the Query Optimizer to built a *query evaluation plan*, a tree of relational operators, so that the user's query can be answered with the least incurred cost. Result tuples flow from the nodes at the lower levels of the tree to their parent nodes, where they get further processed and outputted to their parents respectively, until they reach the top-most node, the root of the operator tree, at which point they are returned to the user.

To simplify the interaction between the different operators in a query plan tree, every operator is required to implement a uniform *iterator* interface, so that parent nodes in a query plan do not need to explicitly know the type of the operator(s) they accept as input, and thus what entry point functions they need to call to interact with them. The abstract iterator class which all operators of Minibase extend is `relop.Iterator`. The functions that it provides are the following:

- `Constructor(Iterator[] inputs, params)`: Sets up its member attributes (e.g. its iterator inputs – one if it is a unary operator such as selection or projection, two if it is a binary operator like join) and initializes ("opens") the iterator.
- `isOpen()`: Checks whether the iterator is open.
- `close()`: Closes the iterator, and releases any temporary resources (such as temporary files) held within its lifetime.
- `restart()`: Restarts the iterator, i.e. as if it were just constructed.
- `explain()`: A recursive function that gives a one- line explanation of the iterator, and recurses on its inputs (its children-iterators).
- `hasNext()`: Returns `true` if the iterator has not visited *all* the tuples that are to be returned by the operator implementing the interface.

  As you know, operators in a query plan can be *pipelined*; as soon as a tuple is generated by an operator which acts as input to another one (e.g. a selection which might be a left input of a Nested Loops Join (NLJ)), the higher level operator can immediately include this tuple to its computation. In our example, the NLJ operator can tell whether it has received all the available tuples of its left input by a call to the selection's `hasNext()` method, prior to its next attempt to retrieve the next available tuple. If the selection operator generates for some reason tuples slower than NLJ can consume them, the semantics of `hasNext()` require the method to *block*, until a definite answer can be returned (that is whether the selection has exhausted all its input (`false`), or is still in the process of generating tuples (`true`)).

  It is a good practice to "pre-compute" the next available tuple, so that it can be immediately returned by a call to:
- `getNext()`: Retrieves the next available tuple.

As you might have recognized by now, the class `BTFileScan.java` you implemented for your second homework was an instance of a "low-level" iterator, as its intention was to provide to higher level operators a way to uniformly access the tuples stored in a relation through the B+ Tree access method. The interface in this assignment has changed slightly, and sums up to the seven functions described above.

Your task is to implement these seven interface functions for the classes `relop.SimpleJoin` and `relop.HashJoin`, described in section 4, along with any private support functions you need. To help you get acquainted with the higher-level API utilized by the classes of this layer, we have provided an indicative implementation of the `relop.FileScan` access method, which serially scans a relation, physically stored in a Heap File that it takes as a parameter, and returns `relop.Tuple` objects, one for each tuple of the relation.

## 3 Support Classes

As mentioned in section 1, in this homework we all live in a "semantically richer" plane of existence, and thus we cannot be expected to still create files, pin pages and in general perform all these low-level, error prone activities we were forced to do in homework 2 :) . To simplify the task of relational operators coding, the implementors of Minibase architected the system in such a way that all the interaction with the buffer manager and the disk manager is taking place *up to* the "Files and Access Mathods" layer (see figure 1). Following, are a number of higher-level support classes that you might find useful when you implement your join operators. For a detailed description of their functionality, take a look at the javadoc included in the archive, or visit: .

- `relop.Tuple`: Each record of a relation, previously represented by a `byte[]` array, is now encapsulated into a Tuple object, which performs the association of the raw data values of a record with the types and lengths of attributes that comprise it. In other words, `relop.Tuple` provides a logical view its attributes, allowing get and set operations, while automatically handling offsets and type conversions. Of particular interest you will find the methods `join()`, which joins two tuples together given the schema of the result tuple, and `insertIntoFile()`, which inserts the current tuple object into a `heap.HeapFile` object.
- `relop.Schema`: Each tuple has a schema that defines the logical view of the raw bytes. This class describes the types, lengths, offsets, and names of a tuple's fields.
- `relop.Iterator`: All relational operators are extend this abstract class. For a general discussion about iterators see section 2 or section 12.4.3 (page 408) of your textbooks.
- `relop.Predicate`: Provides an implementation of simple SQL expressions of the form: `Relation.field <op> Relation.field`, or `Relation.field <op> constant` where `<op>` can be one of: $<$, $<=$, $=$, $!=$, $>$, $>=$ (for example `R.a >= S.b` or `R.a != 7`). This class is required by the Nested Loops Join algorithm you will implement (see section 4), in order to specify the conditions according to which the two inputs of the operator will join.
- `relop.FileScan`: The most basic access path of a relation, provided to you as an example implementation of an iterator.
- `global.SearchKey`: Provides a general and type-safe way to store and compare index search keys. This class substitutes the `index.Key` class of Homework 2.
- `index.HashIndex`: An implementation of the static hashing scheme, described on pages 371–373 of your text book. You will need to use this class to generate hash partitions of a relation, for the partitioning phase of the Hash Join algorithm. Keep in mind that this class implements an *unclustered index*, and thus the resulting hash buckets won't contain tuples, but `<global.SearchKey, global.RID>` pairs, as the leaf pages of the B+ Tree you implemented for Homework 2.
- `relop.IndexScan`: An iterator that scans a hash index file bucket by bucket. You will need it in the probing phase of the Hash Join algorithm to scan through the partitions of the two relations you generated during the partitioning phase. Notice that the class' constructor accepts as arguments the schema of the indexed relation, the `index.HashIndex` object according to which the relation is indexed, and the `heap.HeapFile` object that stores the tuples of the relation.

  To iterate through the indexed tuples one bucket at a time, `relop.IndexScan` changes the iterator interface slightly, by providing the functions:

  - `getNextHash()`: which returns the hash value of the tuple to be retrieved by `getNext()`. If the hash value is the same as the value of the previous tuple, both tuples hash in the same bucket. `getNext()` will retrieve all the tuples of a bucket sequentially, before going to the next one.
  - `getLastKey()`: which returns the `global.SearchKey` field of the tuple retrieved by the most recent call to `getNext()`.

  The following code excerpts exemplifies the use of the above methods:

  ```
  [...]
  IndexScan is = new IndexScan(...);
  ```

```
int currentHash = is.getNextHash(); // Get the hash value of the first tuple
                                    // residing in the first hash bucket.
while(is.hasNext())                 // Loop through all the tuples in all
                                    // buckets serially (bucket by bucket)
{
    // Loop through all the tuples inside the same hash bucket
    while(is.getNextHash() == currentHash)
    {
        Tuple tuple = is.getNext();     // Get the next tuple
        SearchKey key = is.getLastKey();// and the value of the key (tuple field)
                                        // that the hash index was built on.

        // Process the tuple
        [...]

    }

    // End of bucket. Do some processing to handle the transition from
    // one bucket to another. Perhaps...
    currentHash = is.getNextHash();
    // ...among other things.
    [...]
}
```

- `relop.HashTableDup`: This is an extension to Java's hash table that allows duplicate keys. Use this class for your in-memory hashtable, in the second phase of your Hash Join implementation.

## 4 What to Implement

### 4.1 relop.SimpleJoin.java

This class corresponds to the simple version of the nested loops join algorithm, which can be found in your textbooks on page 454. The only extension perhaps from the algorithm described in your books, is the utilization of the `relop.Predicate` class which subsumes the equijoin condition of figure 14.4, and allows for a conjunction of more general conditions on virtually all the fields of the two relations to be joined. If you remember, this general-purpose join was defined as a *theta* join.

Another thing to keep in mind is that join is a binary operator that accepts other iterators as inputs. You don't need to care what type of relational operators these iterators are. By using the support classes mentioned in section 3, and namely `relop.Tuple` and `relop.Schema`, you are in the position to know the schema to which all the tuples of an iterator conform.

### 4.2 relop.HashJoin.java

The heart of this assignment is to implement the Hash Join algorithm described on pages 462–464 of your textbooks. As mentioned in section 3, you will need the `index.HashIndex` class to help you partition the tuples of your input relations (or rather iterators) into buckets, which corresponds to the first phase of the algorithm. Then, you will have to use the class `relop.IndexScan`, to scan through these partitions during the probing phase of the algorithm. Finally, in order to be able to build an in-memory hashtable for the current partition of your "outer" input, as it is required by the probing phase, use the `relop.HashTableDup`, an extension of the `java.util.Hashtable` which allows you to associate more than one tuple with a particular key value, and thus provides support for duplicates.

Note that unlike Nested Loops Join which supports multiple predicates, the Hash Join operator you will be implementing will be an equijoin on a single attribute pair. The join attribute for the outer and the inner relation of the equijoin is specified by the number of the two attributes in the corresponding schemata.

In your textbooks on page 465, you will see a discussion about memory sizes with respect to the biggest partition. To simplify the algorithm, do not concern your selves with memory management issues. Assume that every partition (or equivalently the in-memory hashtable generated by a partition) will fit in memory.

Feel free to define as many private fields and methods you feel necessary to implement the algorithm. For efficiency in hash joins, you must consider the following:

- If the left and/or right input iterator is of type `relop.IndexScan` (i.e. it accesses the tuples bucket by bucket, as they are retrieved by an unclustered hash index), you don't have to rebuild the index **for that input** (i.e. create partitions for that particular input iterator – it is already partitioned for you!). You can assume that when an `relop.IndexScan` is given as an input, it will correspond to an hash index built on the join attribute (remember, the inputs to relational operators are provided by the optimizer, which already knows how to speed-up its underlying operators to save as much work as possible).

- The static hash index you will be building in the partitioning phase (for those inputs that need it) will be an *unclustered* index. Thus, its buckets will contain `<key, rid>` pairs that will point to the tuples of the underlying relation, *scattered randomly across the relation's pages.* If the iterator corresponding to an input of the Hash Join operator is not a `relop.FileScan` (or an `relop.IndexScan` as specified in the previous point), then you will not have an underlying relation, materialized into a file, for these `rids` to point to tuples inside it!

  Furthermore, notice that after you build your Hash Index, you will need to create an `relop.IndexScan` object to iterate through the generated partitions in the probing phase of the algorithm. The constructor of `relop.IndexScan` takes as arguments the `relop.Schema` of the relation you are indexing, the unclustered index (`index.HashIndex` object) and a `heap.HeapFile` object, corresponding to the file of the relation you are indexing. Thus, do not forget to materialize the *input iterators* of the Hash Join operator, whenever they are not an `relop.IndexScan` or a `relop.FileScan` object, into a temporary file, as you are creating the hash index over it (that is, you write every input tuple in a temporary file and then insert it into the secondary index in parallel).

  Perhaps we should add that using an *unclustered* index implementing a preprocessing phase of a blocking operator, such as the partitioning phase of the Hash Join operator, is not really a "good" idea in terms of efficiency. Being the hash index unclustered implies that when we use it in the probing phase to scan through the tuples bucket by bucket, we will keep jumping back and forth into the temporary file, reading each page multiple times. A more typical case would be to utilize a *clustered* hash index. This too might require materialization, but of the output indexed relation (tuples come in on the fly, are inserted in the index and then written in parallel to a temporary file on disk). The difference lies in the fact that the temporary file will be *clustered* on the search key the hash index is built upon. This is how both the Hash Join and the Sort-Merge Join algorithms are described in your books.

  The advantages of using a clustered index should be obvious: the pages of the temporary file are read sequentially, only once, during the probing phase of the algorithm, plus the algorithm is more intuitive to follow. Unfortunately, the current Minibase framework provides an unclustered hash index only. Thus, for the purposes of this homework, the temporary files you will be creating will be unclustered (they won't have a particular ordering according to the index). The ordering of the tuples in the file will coincide with the order you receive the tuples from their input iterator.

## 5 Testing your code

To help you test your code, we provide `tests.ROTest.java`, a java program that exemplifies the usage of several of the support classes mentioned in section 3, and tests your two join algorithms. Method `test1()` creates a `Drivers` relation, it populates it with values, it builds a hash index in the process, and then it tests the `relop.FileScan` operator we have given you and your implementation of the Nested Loops Join. Method `test2()` tests you Hash Join operator, by joining relations `Drivers` and `Rides` on `Drivers.DriverId = Rides.DriverId`. For the schemata of these relations, look at the `main()` method. As the relations used are fairly small, you can verify the correctness of your implementation, and you can devise further tests of your own (for instance, try to execute a three-way join, so that one of your HashJoin inputs is not a mere FileScan but a whole subtree of your query plan).

Your submission will be evaluated on similar tests, like the ones in `tests.ROTest.java`. Our criteria will be the following:

- The two join algorithms should produce the expected join results, when their inputs are base relations (ie `relop.FileScan` iterators).
- The results of your joins when the inputs of your join algorithms are other types of iterators (index scans, selections, projections, other joins).
- For your Hash Join implementation, we will check whether you materialize the input iterators only when it is necessary.
- We will check if the necessary clean-up is performed when your iterators close (e.g. if you destroy temporary files).

## 6 Deliverables

Please submit the following files, using the normal process:

- `HashJoin.java`
- `SimpleJoin.java`
- A `Readme.txt`, containing the members of your group, and a description of what works and what doesn't. Also note that if the synthesis of your group has changed since homework 2, this is a good place to mention it.

This homework is due on April $10^{th}$ 2007, at 22:00. As always, if you find any ambiguous points or bugs in the assignment, please report them as early as possible to the instructors and TAs.