Relation Operators for Minirel

Raghu Mallena   Amit Shukla

May 8, 1995

"Codd created Relations, All else is the work of man"

1 Introduction

Our group had the responsibility of implementing various relational operators for minirel. The relational operators that we implement are: *nested loops join*, *block nested loops join*, *index nested loops join*, *simple hash join*, *grace hash join*, *hybrid hash join*, and *sort–merge join*. We have also implemented utilities for external sorting, elimination of duplicates, a selection predicate evaluator, projection of tuples, scanning files, and scanning indices.

To help in the implementation of these relational operators, we also implemented various kinds of input and output buffers to buffer input from disk files and from other iterators. These are:

- Output buffer to disk
- Input buffer to disk that informs the user when the buffer is full, and the user must explicitly read in the next chunk from disk.
- Input buffer taking input from a "faked" input file.
- Input buffer taking input from an iterator.

Relational operators provide some services to the Plan generator, and expect some services from Heap files and Index methods. This is represented in Figure 1. These interfaces are explained in futures sections of this report.

Section 2 explains the interface that we provide to the plan generator. In sections 3, we present the internal details of the relops. Section 4 details the work involved in this project. Section 5, discusses testing the join algorithms. A retrospective on the design of Minirel is presented in section 6. Appendix A contains the schema of the relations used for testing the join algorithms, and the sample tables.

2 External Interface

The only interface we provide (other than the constructor) is a `get_next()` function. This function takes a reference to a `Tuple *`, and makes it point to an internal output buffer for the next tuple to be returned by the relational operator.

For example, for a nested loops join, the `get_next()` function prototype looks like:

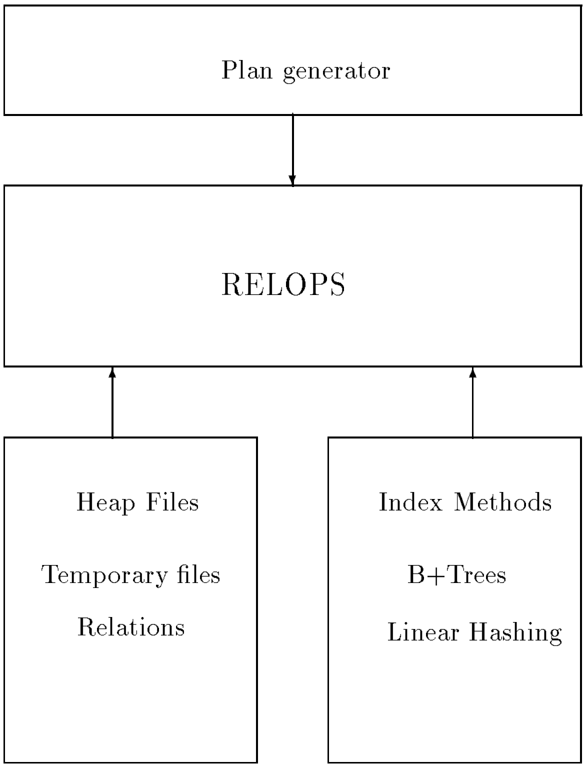

Figure 1: The big picture. Our module relative to other modules.

```
Status nested_loops_join::get_next(Tuple * & tuple);
```

A sample call to this function would look like:

```
Iterator * join;
Tuple    * tuple;
        .
        .
        .
Status s = join->get_next(tuple);
```

After the call to `get_next()`, tuple points to an internal buffer of the join algorithm which contains the tuple resulting from the join. `get_next()` returns `OK` if there are no errors, it returns `DONE` when all tuples have been returned. Other error codes are returned on error.

A sample Constructor

Since the arguments to most of the constructors are similar, we do not describe all of them. A point to be noted is that some join algorithms need to read their inner relations more than once (eg: nested loops). These constructors require an iterator (for the outer), and a file name of the inner relation to be scanned. Other joins algorithms do not iterate on this inner (sort–merge, hash joins). Their constructors take two iterators as input. We pick sort–merge join as being representative of joins which take 2 interator streams as input, and nested loops join as being representative of joins which take an iterator and a file name as input.

Nested loops constructor

The constructor for the nested loops join is:

```
nested_loops_join::nested_loops_join(
   AttrType    in1[],         // Array containing field types of R.
   int     len_in1,           // # of columns in R.
   short   t1_str_sizes[],
   AttrType    in2[],         // Array containing field types of S.
   int     len_in2,           // # of columns in S.
   short   t2_str_sizes[],
   int     amt_of_mem,        // IN PAGES
   Iterator *    am1,         // access method for left i/p to join
   char *  relationName,      // access method for right i/p to join
   CondExpr **outFilter,      // Ptr to the output filter
   CondExpr **rightFilter,    // Ptr to filter applied on right i/p
   FldSpec  * proj_list,
   int        nOutFlds,
   Status   & s
  );
```

R and S are the outer and inner relations respectively. in1[] and in2[] are arrays containing the types of R and S respectively. t1_str_sizes and t2_str_sizes are arrays containing the lengths of string fields in tuples of R and S respectively (This information is required to create a tuple). amt_of_mem is the number of buffer pages that can be used by the join algorithm. am1 is a pointer to an Iterator to the outer relation. relationName is the name of the relation containing the inner relation. outFilter and rightFilter are the output filter and filter to be applied to the inner relation respectively. proj_list contains a list of the fields to be projected to produce the output tuple. nOutFlds is the number of fields in the output tuple. s is set to return error codes due to errors in the constructor.

Sort–merge constructor

It is easy to see that this is similar to the nested loops constructor above. Thus, we explain only the parameters which are different from those of the nested loops join.

```
sort_merge::sort_merge(
   AttrType    in1[],           // Array containing fld types of R.
   int         len_in1,         // # of columns in R.
   short       t1_str_sizes[],
   AttrType    in2[],           // Array containing fld types of S.
   int         len_in2,         // # of columns in S.
   short       t2_str_sizes[],
   int         join_col_in1,    // The col of R to be joined with
   int         sortFld1Len,
   int         join_col_in2,    // the col of S.
   int         sortFld2Len,
   int         amt_of_mem,      // IN PAGES
   Iterator *  am1,             // Iterator for left input to join.
```

```
    Iterator *  am2,            // Iterator for right input to join.
    bool        in1_sorted,     // is am1 sorted?
    bool        in2_sorted,     // is am2 sorted?
    TupleOrder  order,
    CondExpr ** outFilter,      // Ptr to the output filter
    FldSpec  *  proj_list,      // Projection list
    int         nOutFlds,       // # of elements in the output tuple
    Status   &  s);
```

`join_col_in1` and `join_col_in2` are the column numbers of the join columns in the outer and inner relations respectively, `sortFld1Len` and `sortFld2Len` are their lengths. `am1` and `am2` are pointers to `Iterator`s to the outer relation and inner relations respectively. `order` is the sort order (`Ascending` or `Descending`). If the outer is sorted in `order` order, then `in1_sorted` is set to `TRUE`. A similar situation applies to `in2_sorted`.

The constructors for all the hash joins are more or less similar to that of a sort-merge. The only additional parameter required is the size of the inner (smaller) relation in pages.

2.1 What does *this* Relop buffer?

This section describes the buffering performed by the various relational operators. The output tuple is always placed in an output buffer local to the join operator, `get_next` makes its argument to point to this internal output buffer.
External sorting – blocks its iterator inputs. The runs which are generated are written to temporary files (this write is buffered). The runs that are read from temporary files on disk are also buffered.
Duplicate Elimination – Calls a sort routine which does all buffering.
Simple Nested loops Join – performs no buffering.
Block Nested loops Join – buffering of inputs.
Sort Merge Join – Calls 0/1/2 sort routine(s) which performs the required blocking.
Hash Joins – The initial read of relations is unbuffered. The runs written to disk and read from disk are buffered.

3 Internal Design

This section gives the internal details of the relational operators that we have implemented.

3.1 External Sorting

The external sorting algorithm we used uses tournament trees to sort a relation. The average length of a run is twice the size of available memory. Further details can be obtained in the Shapiro paper. After the sorted runs have been generated, we simply read some part of each run in and perform a merge of the sorted runs.

3.2 Duplicate Elimination

The duplicate elimination utility basically sorts its input on the first column of the tuple using the above external sorting utility. Then it scans the sorted relation and eliminates duplicate tuples by comparing all their fields using the function:

```
bool Equal(Tuple *t1, Tuple *t2, AttrType types[], int len);
```

This function is implemented in `tuple_utils.C`.

3.3 Nested Loops (simple and blocked)

The simple nested loops algorithm is:

`foreach` tuple $r \in R$ `do`
    `foreach` tuple $s \in S$ `do`
        `if` $r_i == s_j$ `then` add <r, s> to result

Thus, we scan the *outer* relation R, and for each tuple r $\in$ R, we scan the entire *inner* relation S.

Block nested loops join is a simple refinement of this. It performs a *page–at–a–time* scan: for each page of R, we can retrieve each page of S and write out tuples <r, s> for all matching tuples r $\in$ *R–page* and s $\in$ *S–page*. The advantage of this approach can be found elsewhere.

3.4 Index Nested Loops Join

If there is an index on one of the relations on the join attribute, we can take advantage of it by making it the inner. Suppose that we have a suitable index on S. For each tuple r $\in$ R, we use the index to retrieve matching tuples of S.

The index nested loops algorithm is:

`foreach` tuple $r \in R$ `do`
    `foreach` tuple $s \in S$ `where` $r_i == s_j$
        add <r, s> to result

3.5 Sort–Merge

The basic idea behind the Sort–Merge algorithm is to *sort* both relations on the join attribute, and then to look for matching tuples r $\in$ R and s $\in$ S by essentially *merging* the two relations. The external sort routine described above is the one used for sorting. The basic idea is as follows: We scan the relations R and S, looking for matching tuples ($R_i = S_j$). Initially, the two scans are positioned at the first tuple in each relation. We advance the scan of R as long as the current R tuple is less than the current S tuple (with respect to the values in the join columns). Similarly, we then advance the scan of R as long as the current S tuple is less than the current R tuple. We alternate between such advances until we find an R tuple and an S tuple with $R_i = S_j$.

When we find R and S such that $R_i = S_j$, we need to output the joined tuple. In fact we could have several R tuples and several S tuples with the same value in the join columns as the current R and S tuples. We refer to these tuples as the *current R group* and the *current S group*. For each tuple r in the current R group, we scan all tuples s in the current S group, and output the joined tuple <r, s>. We then resume scanning R and S, beginning with the first tuples that follow the groups of tuples that we just processed.

3.6 Hash Joins

In the hash algorithms that follow, R is the inner(smaller) relation and the outer relation is S.

3.6.1 Simple Hash Join

Simple Hash Join is an extension of the classic hash-join algorithm which imposes the requirement that the whole of R relation should fit into the available memory (i.e $|R| * F \leq |M|$). In the classic hash-join, a hash table is built in memory for the whole of R and for each tuple in S, this hash table is probed to see if a match exists. If there is not enough memory available, simple-hash join scans R repeatedly, each time partitioning off as much of R as can fit in a hash table in memory. The rest of the R tuples are written to disk and will be used as the input set for the next pass. After each scan of R, S is scanned and for tuples corresponding to those in memory, a probe is made for a match. The rest of the S tuples are written to the disk and will be used as the S-input set for the next pass. The above sequence is repeated until there are no more tuples of R or S written to disk.

Implementation

1. The constructor gets the required buffer pages(`_npages`) from the Buffer Manager, initializes the input and output buffers, creates temporary files and allocates memory for the hash table. It should be noted that in our implementation, the hash table is a virtual memory hash table which just stores pointers to the tuples which are actually in the buffer pages. The size of this hash table is set to max($|R| * F$, `_npages`). Each bucket in the hash table is implemented as a linked list.

2. A first level hash function is used to divide R into two sets: a MEMORY set and a DISK set, such that the whole of the MEMORY set will fit into the available buffer pages. If there are Rpages of R and Bpages of buffer pages available the in-memory hashed tuples, the hash function,h1, is logically(ignoring fudge factor F):

```
int h1(int i) {
      if( i % Rpages < Bpages)
         return MEMORY;
      else
         return DISK;
}
```

   For all the tuples which are hashed into MEMORY, a second level hash function, h2, is applied to hash them into different buckets of the hash table. A simple modulo-function is used for this too. The rest of the R tuples are written to a temporary file.

3. For each invocation of `get_next()`, get a tuple of S relation(not necessarily a new S tuple every time, as explained later) and apply h1. If the tuple maps to MEMORY, apply h2 to probe the hash table. If there is a matching R tuple, apply the selection filter and return the joined tuple if the selection succeeds. It should be noted that there could be more than one R tuples in the hash bucket. If a match is found before we reach the end of the bucket's list, we should store the present value of S-tuple and location in the bucket list so that the search is resumed from this point on when `get_next()` is called the next time. Write the DISK set of the S tuples into a temporary file.

4. If R–tempfile or S–tempfile is not empty at the end of steps 2 and 3, repeat steps 2 and 3 using these temporary files as inputs.

3.6.2 Grace Hash Join

Grace Hash Join executes in two phases. In the partitioning phase, R and S are partitioned into subsets using the same hash function, such that each subset of R can fit into memory when a hash table is built on it. Each of these R and S subsets are then written to disk(temporary files). In the second phase, a hash table is built in memory for a subset of R. For each S-tuple in the corresponding S subset, this hash table is probed for a match. This is repeated for all the subsets.

Implementation

1. The constructor does the whole of the partitioning phase. The number of runs, Nruns = sqrt(F∗R). Out of the available buffer pages, 'Nruns' of them are initialized as output buffers for writing R and S tuples to disk. A simple modulo–hash function is used to distribute tuples among these buffers. After this is done for all the R tuples, the buffers are flushed to temporary files and this is repeated for S relation. Also, a RunCounter is maintained to indicate the run number in the second phase. This is initialized to zero in the constructor.

2. For each invocation of `get_next()`,

   ```
   if this is the beginning of a run,
      read the R-tuples from the appropriate temporary file
      and build a hash table using a hash function h2, which
      is different from the one used for partitioning.
   else
      hash table for this run was already build earlier.
   ```

   Now, get the next tuple of S relation (exceptions to this apply as explained in simple hash join) and apply h2 to probe the hash table. If there is a matching R tuple, apply the selection filter and return the joined tuple. This is repeated for all the Nruns and a status of DONE is returned after the last run.

Restrictions

Grace join requires a minimum of sqrt(F*R) buffer pages to do the join. An error code is returned if the required buffer pages are not available.

Simplification from Shapiro's algorithm: Extra buffer pages, if available, can be used for buffering purposes. In our implementation, we do not use these extra buffer pages in the partitioning phase.

3.6.3 Hybrid Hash Join

Hybrid hash combines the features of the two preceding algorithms, doing both partitioning and hashing in the first pass over both relations. A part of the R relation is hashed and the rest of it is divided into runs just like Grace hash and written to disk. The tuples of S are either used for probing the hashed part of R or written to disk as runs. The number of runs is a function of the available memory and the size of R-relation. It is computed as:

```
N = ceil( (|R|*F - |M|)/(|M| - 1))
```

N buffers are reserved in pass 0 to write parts of R and S to disk.

Implementation

1. The constructor initializes the N output buffers for the R and S runs. It then processes all the tuples of R once. For each tuple, a hash function h1, similar to h1 of Simple Hash join, is applied, which basically determines if this tuple should go into MEMORY or DISK. If the tuple goes into MEMORY, another hash function,h2, is applied to put it into the appropriate bucket of an in-memory hash table. If the tuple goes to DISK, another hash function,h3, is applied to determine which run this belongs to and written to the appropriate output buffers. Flush all the output buffers to temporary files when all the R-tuples are done.

2. Each `get_next()` does the following:
   if this is the first pass over S, (i.e the S runs have not been built completely), get the next tuple of S and apply h1 to determine whether it goes to MEMORY or DISK. If it goes to MEMORY, use h2 to probe the in-memory hash table for R. If a match is found, apply selection filter and return the joined tuple if it passes the selection condition. If h1 returns DISK, then apply h3 to determine which run this belongs to and write to the appropriate output buffer. When the whole of S is done, flush all the output buffers to disk.

3. At this point, we are exactly in the same situation as we were at the end of the constructor in grace join. Follow the same sequence of steps as we did for the grace join from here on.

4 Accomplishments

This project involved a lot of coding. We have written more than 10,000 lines of code. We have put together 7 join algorithms, and in addition, written an external sorting utility, a file scan iterator, a record delete utility that deletes all tuples which are the same as an input tuple, a predicate evaluator, output and input buffers which flush/load themselves on becoming fill/empty.

5 Testing

The test suite for the joins consisted of 15 tests. The first 6 tests illustrate how the 3 hash joins function under different combinations of buffer pages and R-pages. Tuples for R and S were generated by simple file scans that we wrote for testing. Each tuple has 4 fields: a string, integer, integer and a string.(example:Str: 1, 1, 1, Str: 1) R relation had tuples ranging from 5 to 100 and S relation had tuples from 1 to 125. A join of R and S on the second integer field produced tuples ranging from 5 to 100 as expected.

Test 1: This used a simple hash join with 8 buffer pages and 7 Rpages. Since there are more buffer pages than Rpages, the join completed in one run.

Test 2: This used a grace hash join with 8 buffer pages and 7 Rpages. The number of runs = ceil(sqrt(F*R)) = ceil(sqrt(1.3 * 7)) = 4. The join results confirmed this. We got 4 runs with approximately 24 elements in each run.

Test 3: This used a hybrid hash join with 8 buffer pages and 7 Rpages. Since there are more buffer pages than Rpages, the hybrid hash join worked exactly like the simple hash join of Test 1.

Test 4: Simple hash join with 5 buffer pages and 7 Rpages. A part of the R-tuples are hashed into memory during the first pass. The rest are written to disk and used as input the next time. The effective number of buffer pages used for hashing are 3 out of the available 5. This is because 2 pages are used as input and output buffers after the first pass. The algorithm went through 3 iterations as expected ($7 = 3+3+1$, and hence 3 passes) before all the tuples were joined.
Test 5: Grace hash with 5 buffer pages and 7 Rpages. This performs exactly like Test2 because the number of runs in grace hash is independent of the number of buffer pages.
Test 6: Hybrid hash with 4 buffer pages and 7 Rpages. Calculations show that there should be 3 runs (which are like the ones in grace hash) subsequent to pass 0. Our results confirmed this calculations.
The rest of the tests use hand-generated query plans for SQL style queries on a Sailors-Boats-Reserves database we generated.
Test7: Uses simple hash join and selection for this query.
Query: Find the names of sailors who have reserved boat number 1

```
  SELECT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.rid AND R.bid = 1

Results were as expected:
{Mike Carey}
{David Dewitt}
{Jeff Naughton}
```

Test8: Uses grace hash join and selection for this query.
Query: Find the names of sailors who have reserved boat number 1

```
  SELECT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.rid AND R.bid = 1

Results were as expected:
{Mike Carey}
{David Dewitt}
{Jeff Naughton}
```

Test9: Uses hybrid hash join and selection for this query.
Query: Find the names of sailors who have reserved boat number 1

```
  SELECT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.rid AND R.bid = 1

Results were as expected:
{Mike Carey}
{David Dewitt}
{Jeff Naughton}
```

Test10: Combination of hybrid hash join and Nested Loop Join.
Query: Find the names of sailors who have reserved a red boat and return then in alphabetical order.

```
  SELECT   S.sname
  FROM     Sailors S, Boats B, Reserves R
  WHERE    S.sid = R.rid AND R.bid = B.bid AND B.color = 'red'
  ORDER BY S.sname

Plan used:
 Sort (Pi(sname) (B  |><|  Pi(sname, bid) (S  |><|  R)))

 Results were as expected:
{David Dewitt}
{Mike Carey}
{Raghu Ramakrishnan}
{Yannis Ioannidis}
```

Test11: Combination of grace hash and nested loop joins.
Query: Find the names of sailors who have reserved a red boat and return then in alphabetical order.

```
  SELECT   S.sname
  FROM     Sailors S, Boats B, Reserves R
  WHERE    S.sid = R.rid AND R.bid = B.bid AND B.color = 'red'
  ORDER BY S.sname

Plan used:
 Sort (Pi(sname) (B  |><|  Pi(sname, bid) (S  |><|  R)))

Results:
{David Dewitt}
{Mike Carey}
{Raghu Ramakrishnan}
{Yannis Ioannidis}
```

Test12: Uses a nested loops join to evaluate the following query.
Query: Find the names of sailors who have reserved boat number 1

```
  SELECT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.rid AND R.bid = 1

{Mike Carey}
{David Dewitt}
{Jeff Naughton}
```

Test13: Uses a combination of simple nested loops and sort–merge join.
Query: Find the names of sailors who have reserved a red boat and return then in alphabetical order.

```
  SELECT   S.sname
  FROM     Sailors S, Boats B, Reserves R
  WHERE    S.sid = R.rid AND R.bid = B.bid AND B.color = 'red'
  ORDER BY S.sname
```

```
Plan used:
 Sort (Pi(sname) (B  |><|  Pi(sname, bid) (S  |><|  R)))

{David Dewitt}
{Mike Carey}
{Raghu Ramakrishnan}
{Yannis Ioannidis}
```

Test14: Uses nest blocks join on the query:
Query: Find the names of sailors who have reserved a boat

```
  SELECT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.rid

{Mike Carey}
{Mike Carey}
{Mike Carey}
{David Dewitt}
{David Dewitt}
{Jeff Naughton}
{Miron Livny}
{Yannis Ioannidis}
{Raghu Ramakrishnan}
{Raghu Ramakrishnan}
```

Test15: Uses nest blocks join on the query:
Query: Find the names of sailors who have reserved a boat and print each name once

```
  SELECT DISTINCT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.rid

{Jeff Naughton}
{Mike Carey}
{Miron Livny}
{Raghu Ramakrishnan}
{Yannis Ioannidis}
```

6 Retrospective

We found two bugs in our design after a postmortem analysis. The block nested loops algorithm performs blocking of tuples at its input. The heap file from which the tuples are read also reads a page at a time from the disk. This causes a impedance–mismatch between the two of them. A closer integration of these two components would have solved this problem.

Secondly, tuples could have been non–padded so as to fit in the smallest possible amount of space.

A Sample Tables used in testing

The schemas are:
Sailors(int sid, String sname, int rating, float age)
Boats(int bid, String bname, String color)
Reserves(int sid, int bid, String date)

A.1 Sailors

sid	*sname*	*rating*	*age*
10	Mike Carey	9	40.3
21	David Dewitt	10	47.2
29	Tom Reps	7	39.1
31	Jeff Naughton	5	35.0
35	Miron Livny	7	37.6
37	Marv Solomon	10	48.9
39	Anne Condon	3	30.3
47	Charles Fischer	6	46.3
49	James Goodman	4	50.3
50	Mark Hill	5	35.2
53	Bob Holloway	9	53.6
54	Susan Horowitz	1	34.2
57	Yannis Ioannidis	8	40.2
59	Deborah Joseph	10	39.8
61	Landwebber	8	56.7
63	James Larus	9	30.3
64	Barton Miller	5	43.7
67	David Parter	1	99.9
69	Raghu Ramakrishnan	9	37.1
71	Guri Sohi	10	42.1
73	Prasoon Tiwari	8	39.2
75	Mary Vernon	7	43.1
79	David Wood	3	39.2
84	Mark Smucker	9	25.3
87	Martin Reames	10	24.1

A.2 Boats

bid	*bname*	*color*
1	Onion	white
2	Buckey	red
3	Enterprise	blue
4	Voyager	green
5	Wisconsin	red

A.3 Reserves

sid	*bid*	*date*
10	1	05/10/95
21	1	05/11/95
10	2	05/11/95
31	1	05/12/95
10	3	05/13/95
69	4	05/12/95
69	5	05/14/95
21	5	05/16/95
57	2	05/10/95
35	3	05/15/95