Power-Pipelining for Enhanced Query Performance

Jun Rao
450 Computer Science Building
Columbia University
New York, NY, 10027
junr@cs.columbia.edu
Phone:(212)939-7054
Fax:(212)666-0140

Kenneth A. Ross[*]Columbia University
kar@cs.columbia.edu

Columbia University Technical Report CUCS-007-00
May. 19, 2000

Abstract

As random access memory gets cheaper, it becomes increasingly affordable to build computers with large main memories. In this paper, we consider processing queries within the context of a main memory database system and try to enhance the query execution engine of such a system.

An execution plan is usually represented as an operator tree. Traditional query execution engines evaluate a query by recursively iterating each operator and returning exactly one tuple result for each iteration. This generates a large number of function calls. In a main-memory database system, the cost of a function call is relatively high.

We propose a new evaluation method called power-pipelining. Each operator processes and passes many tuples instead of one tuple each time. We keep the number of matches generated in a join in a local counter array. We reconstitute the run-length encoding of the join result (broken down by input tables) at the end of the join processing or when necessary. We describe an overflow handling mechanism that allows us to always evaluate a query without using too much space. Besides the benefit of reducing the number of function calls, power-pipelining compresses the join result automatically and thus can reduce the transmission cost.

However, power-pipelining may not always outperform the traditional pipelining method when the join selectivity is low. To get the benefit of both techniques, we propose to use them together in an execution engine and let the optimizer choose the preferred execution. We discuss how to incorporate this in a cost-based query optimizer.

We implemented power-pipelining and the reconstitution algorithm in the Columbia main memory database system. We present a performance study of power-pipelining and the traditional pipelining method. We show that the improvement of using power-pipelining can be very significant. As a result, we believe that power-pipelining is a useful and feasible technique and should be incorporated into main memory database query execution engines.

[*]This research was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by an NSF Young Investigator Award, by NSF grant number IIS-98-12014, and by NSF CISE award CDA-9625374.

1 Introduction

As random access memory gets cheaper, it becomes increasingly affordable to build computers with large main memories. There are many existing applications with large datasets of the order of several gigabytes which can fit in RAM [GMS92]. Recently, the “Asilomar Report” ([BBC$^+$98]) predicts “Within ten years, it will be common to have a terabyte of main memory serving as a buffer pool for a hundred-terabyte database. All but the largest database tables will be resident in main memory.” An important factor in main memory data processing is query performance. In this paper, we aim to improve query performance by enhancing the conventional query execution engine.

Some past work on main memory databases has addressed the problems of concurrency, transaction processing and logging [GMS86, LN88, JLRS94], and recovery [Hag86, LC87]. Systems with a significant query-processing component include OBE [WK90], MM-DBMS [LC87], and Starburst [LSC92]. More recently, the TimesTen corporation (formerly the Smallbase project at Hewlett-Packard) has developed a commercial main-memory database system, with claims of tenfold speedups over disk-based systems [Sof97]. We'll elaborate on this in Section 1.2.

However, none of this work has looked at how to improve the design of the execution engine for a main memory database system. Most of the existing systems either do not have a comprehensive execution engine or simply reuse the execution engine for a disk-based system. In the rest of this section, we first evaluate two conventional query execution engines and point out their shortcomings in main memory database systems. We then propose our evaluation method to overcome these problems.

1.1 Existing Execution Engines

An execution engine takes as input an execution plan, which is usually represented as an operator tree. There are two traditional ways of evaluating an operator tree. The first one is a step-by-step method. The complete intermediate result of each operator is generated step by step (bottom up). This method requires the space to store intermediate results at each step and thus is not ideal when those intermediate results are large.

The second method is a pipelining method and is used by most of the commercial systems such as Ingres [Sto80] and System R [CAB$^+$81] and research projects such as Starburst [HCL$^+$90] and Volcano [Gra94]. A more thorough survey can be found in [Gra94]. In this method, each operator supports an open-next-close interface, where open() does most of the preparation work and close() cleans up everything. In the next() calls, the operator simply generates one tuple result and then passes it to the operator above it in the tree. Pipelining requires much less space during execution. For operators that are pipelinable, we only need the space to hold one record of the result. There are two different ways of implementing the “Next-Record” in pipelining. First, one can pass complete records between operators. This means that a join operator has to create new records and copy fields from two input records. Creating complete new records can be prohibitively expensive. Alternatively, one can leave the original records in the buffer as they were retrieved from the base tables, and the Next-Record is composed of an array of tuple pointers. This method avoids much memory copying across operators and is usually the preferred method. Both implementations share the feature that each next() call generates only one tuple result. So we refer to this pipelining method as *1-pipelining* in this paper.

The techniques described above were developed for disk-based database systems, but they can also be used in the execution engine for main memory database systems. Are they adequate for this purpose? Let us consider what the important issues in main memory database systems are. First of all, the gap between CPU and memory speed is widening. While CPU speed is increasing at a rate of 60% each year, memory speed only increases 10% each year [CLH98]. As a result, cache miss penalties are getting higher (relative to CPU cost), which makes cache behavior more and more important. The step-by-step method has poor cache behavior. This is because if the intermediate result is larger than the cache size (which is usually the case), it will be evicted out of cache by the time the next step starts. Thus, the intermediate result has to be loaded into cache again and this will increase the number of cache misses significantly. The 1-pipelining method, on the other hand, has very good cache behavior. Every time only one tuple result is generated, so it is very likely to remain in cache for the following operators. By using a pipelining execution engine, we get a cache conscious method “for free”.

The second important factor, which hasn’t been stressed before, is the overhead of function calls. Function calls are expensive and can be a significant portion of cost in main memory database systems. Let us analyze the number of function calls introduced when traversing the execution tree for both methods. The step-by-step method will call each operator once to get the intermediate results and thus requires few function calls. By comparison, the 1-pipelining method requires one next() call for each tuple result generated by an operator. So the total number of next() calls can be huge. This hasn’t been a big concern for disk-based systems. The implementation of the operators in disk-based systems are usually very complex and many other function calls (such as pinning data in the buffer) may need to be invoked in order to compute one record. As a result, the overhead of next() calls may be negligible. However, the implementation of an operator in a main memory system could be much simpler. Data is available much faster and there is no need to pin data in RAM any more. So, the overhead of function calls can no longer be overlooked. Although most of the function calls in the body of next() can be avoided (through in-lining), next() function calls themselves *cannot* be easily in-lined since they are virtual functions (in C++) and are implemented differently for different operators. The complexity of the shape of the execution tree and the number of possible kinds of operators make it difficult to evaluate a query non-recursively without introducing other kinds of overhead.

Before proposing a solution to this problem, we first summarize previous research work on main memory query processing.

1.2 Main Memory Query Processing

People have long known that simply increasing the size of the buffer pool in a disk-based database system is not the right way to use a large amount of RAM. Instead, a specialized main memory storage unit should be built to take advantage of the fact that all the data reside persistently in RAM. [LSC92] reported a fourfold speed improvement when using a specialized main memory resident storage component.

[Fre95, Fre97] contrasted the requirements of OLAP with OLTP systems and contended that the real performance gains can be obtained by separating the two systems. A dedicated OLAP system can have a much better query performance if we are willing to sacrifice the update performance. For example, if only batch updates are supported, we can have a different data layout that favors querying.

[AHK85] and others introduced the concept of domain. When data is first loaded into main memory, distinct data values are stored in an external structure, the domain, and only pointers to domain values are stored in place in each column. In the main memory database project at Columbia University, we go further by keeping the domain values in order and associate each value with a domain ID (represented by an integer). Besides saving storage space, these simplify query evaluation since we can process pointers (IDs), rather than the original values. In [WK90], the authors transformed foreign key values into tuple pointers. This provides a more efficient way of doing foreign key joins.

The implementation of each operator in a main memory database system can be much simpler than that in a disk-based system. We don’t have to do I/Os and pin data in the buffer. We may not need to lock data during query processing. Since all the tuples reside in main memory, we can use tuple IDs during query processing. As a result, the overhead of function calls can no longer be ignored.

The rest of the paper is organized as follows: In section 2, we revisit the traditional 1-pipelining method in detail and explain why it is difficult to widen the pipe. We describe our power-pipelining method, the reconstitution algorithm and the overflow handling mechanism in Section 3. We discuss the tradeoffs between power-pipelining and 1-pipelining in Section 4. In Section 5, we propose a way to let the query optimizer choose between power-pipelining and 1-pipelining. We show our experimental results in Section 6 and we conclude in Section 7.

Query 1:

select * from R, S, T
where T.a=S.a and S.b=R.b

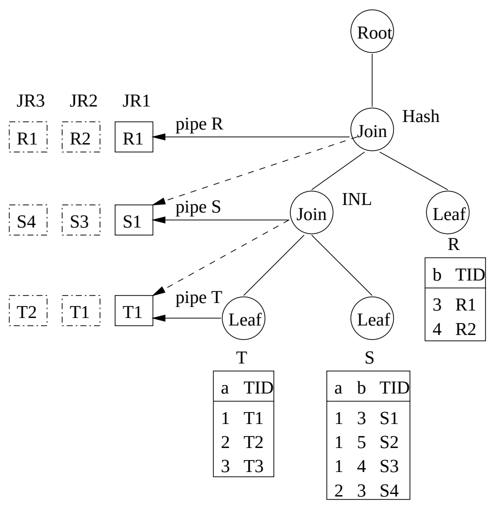


Figure 1: Evaluation using 1-pipelining

2 1-Pipelining Revisit and a First Attempt

Before going into the details of power-pipelining, let's take a closer look of the evaluation process using the traditional 1-pipelining method. The SQL query in Figure 1 specifies a 3-way join query. Suppose that table S is large and it has an index on column a. The optimizer may decide to in table T and S using indexed nested loop join first. The join result is then used to probe the hash table built on table R. We show the execution tree in the right picture of Figure 1. Each leaf node corresponds to a base table. We list the content of each table under its corresponding leaf node. There are two join nodes. The lower one uses an indexed nested loop join and the upper one uses a hash join. The most important thing here is the arrangement of the pipes. A pipe is created for each participating table and it has the space to hold exactly one tuple from that table. The three pipes for the plan are shown in solid boxes in Figure 1. Leaf node T owns pipe T and it will fill pipe T during the table scan. Each join node owns two pipes, an outer one and an inner one. A join operator reads data from the outer pipe (pointed to by a dotted line) and puts the matching tuple into the inner pipe (pointed to by a solid line). We assume there is always a root node that collects the query result.

During preparation time, we build the hash table and reset the index for probing. Then the root node starts calling next() on its immediate child. This call is propagated all the way to leaf node T. Leaf node T retrieves the first tuple from table T and copies it to pipe T (for the sake of simplicity, we only put the tuple ID in the pipe). It then returns the control to its parent node. The lower join node extracts the join key from its outer pipe (pipe T) and uses it to search the index on table S. The first matching tuple S1 is copied to pipe S. Before returning, the join node saves some state information so that it knows where to start to find the remaining matches next time. The process continues in the top join, where the join key is extracted from pipe S and is used to probe the hash table. The matching tuple R1 is copied to pipe R. Now the contents in all the three pipes are shown under JR1 in Figure 1. The root node will copy the data in the three pipes to a buffer and save it as the first join result tuple.

The same process continues. Every time we return to the root node, we get back a new join result tuple. We show the two subsequent join results under JR2 and JR3 in Figure 1. The whole process finishes when no more tuples can be generated. Note that the space of the pipes are shared across all the iterations. A newly inserted tuple overwrites the previous data in a pipe. 1-pipelining has a simple 0-1 logic which makes the implementation easy. But to generate one new tuple, the overhead includes at least one function call and at least one saving of state for each operator involved.

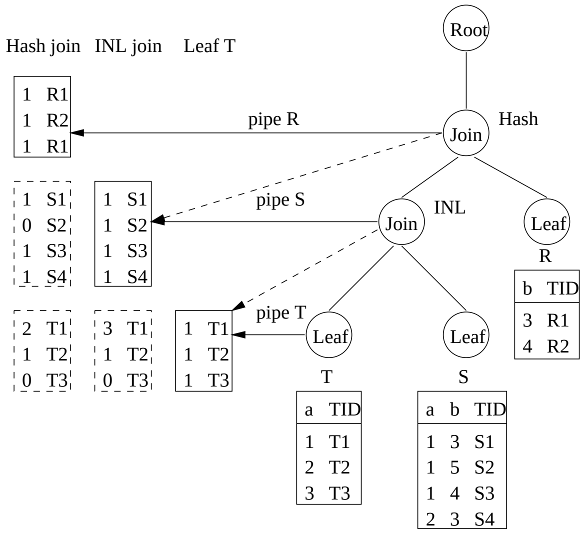


Figure 2: Widening pipeline, first attempt

2.1 A First Attempt

Industry people have also noticed the problems with conventional pipelining [Mac97]. However, no new evaluation method has been proposed so far. As we will see shortly, the seemingly obvious solution doesn't work in practice.

To avoid the overhead caused by 1-pipelining, it's natural to think of increasing the pipe size. If we can process many tuples at each next() call, the overhead can be reduced. During a join, a probing tuple may have any number of matches. Deleting or duplicating tuples in a pipe can be expensive. To avoid that, we can use a run-length encoding to represent the tuples in a pipe. For each tuple, we associate it with a counter indicating its cardinality. So, our first attempt is to keep all the counters up-to-date during the evaluation. Figure 2 shows this approach assuming all the pipes can accommodate at least four tuples. We list each counter to the left of each tuple in the pipe. Again, we start with leaf node T. It fills pipe T with all three tuples at once and sets all the counts to one. In the indexed nested loop join, each tuple in pipe T with a count larger than zero is used for index probing. We add all the matching tuples sequentially in pipe `S`. Each of them is given a count of one. We update the count of each tuple in pipe `T` to be the number of matches it has found. Since `T3` doesn't have any matches, the count for `T3` is updated to zero. The contents in pipe `T` and `S` at this point are shown in the middle column in Figure 2. We then continue in the hash join. Now comes the problem. `S2` doesn't have a match. So we update its count to zero. However, that's not enough. We also have to decrease the count of `T1` to two since logically, all the pipes together form the complete tuples. This means that every time we update a counter in one pipe, we have to update counters in other pipes and make them consistent with each other. This introduces a larger overhead than that of function calls. As a result, our initial implementation of this attempt did not show any benefit.

3 Power-Pipelining

We now present our power-pipelining method. We design each pipe to consist of an array of tuples, a bitmap and a counter array. Each bit indicates the tuple's validity and each counter gives the cardinality of the tuple. During the evaluation process, we only update the tuple list and the bitmap in each pipe. We may keep some additional information local to each operator. However, we defer the update of the counter array in each pipe until necessary. All the updates are local and this avoids the main drawback of the previous attempt. We present the implementation of two typical operators in Section 3.1. In Section 3.2, we describe how to use all the local information we have kept to reconstitute the real join result. We discuss the overflow handling in Section 3.3.

3.1 Update Local Counters

We discuss the implementation of two typical operators, namely the leaf operator and the join operator. A leaf operator owns a base relation and optionally a local selection predicate. We assign a pipe to each leaf operator. The task of the leaf operator is to fill the pipe with as many satisfying tuples as possible and to turn on all the bits in the bitmap.

A join operator usually consists of two operands. We distinguish them as the outer and the inner. The outer is used for probing and the inner is the one to be probed on. For example, in a hash join operator, the inner is used to build the hash table and the outer is used to probe the hash table. In an indexed nested loop join, the inner operand is the indexed table (a leaf operator) and the outer operand is the one used to search the index. Besides the two pipes, we associate each join operator with a local counter array. The counter array stores, for each tuple in the outer pipe, the number of matches it generates. The join procedure is as follows: Each tuple from the outer pipe is checked for its validity first. A valid tuple is then used to find the matching tuples from the inner. If no matches are found, we turn off the corresponding bit of the tuple indicating that it's not valid any more. We still keep the invalid tuples in the pipe since it's expensive to move data around. The matching numbers are kept in a local counter array and will be used by the reconstitution algorithm to be described later. The pseudo code for the join procedure is shown in Algorithm A.1 in Appendix A. Right now, we assume that the inner pipe is large enough to hold all the matching tuples. We defer the handling of overflows in Section 3.3.

If we know that each outer tuple can have no more than one match in a join (e.g., foreign key constraints), we can avoid generating the local counter array at all. When there is a match for an outer tuple, we put the match in the corresponding slot in the inner pipe. Otherwise, we simply turn off the corresponding bit of the outer tuple. Notice, that we can share the bitmap in the inner and the outer pipe in this case.

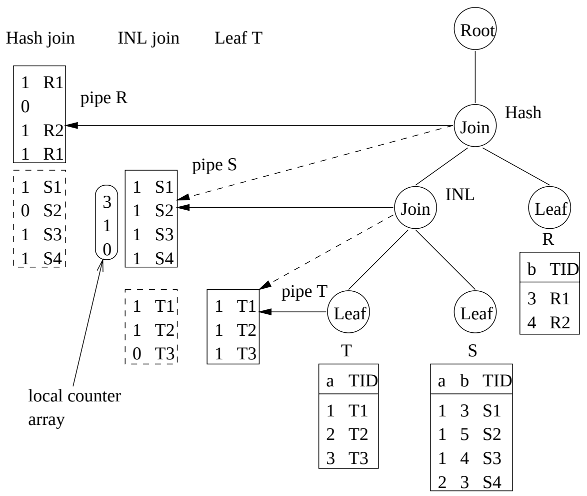


Figure 3: Power-Pipelining

In Figure 3, we present our new approach using the same example as in the previous section. We show in each pipe only the bitmap and the tuple array. We show the content in the pipes after each step. Initially, pipe `T` is filled and all the bits are set to one. During the first join, the matching numbers for `T1`, `T2` and `T3` are kept in the local counter array owned by the lower join operator (next to pipe `S`). Since there is no match for `T3`, its bit in pipe `T` is turned off. Had pipe `T` been used as the outer pipe in a subsequent join, T3 wouldn't be used for probing. The contents of the pipes after the first join are shown under "INL join" in Figure 3. Suppose that we know the hash join is a many-to-1 join in advance. Each matching tuple is put in the corresponding slot in the inner pipe (pipe `R`). The bit for `S2` is turned off since it has no match. Although not shown in the figure, the bitmaps in pipe `S` and `R` are shared. Observe that we didn't try to update the local counter array next to pipe `S` when processing the upper join. In the whole process, each operator calls its next() function only once.

3.2 Reconstitute the Join Result

For each execution tree, we create a corresponding *pipe generating tree*, which shows how all the pipes are populated. Basically, each pipe corresponds to a node in the tree. There is a directed edge between an outer pipe and an inner pipe in the same join operator, with the arrow pointing to the inner pipe. We further distinguish an edge as either a *1-branch* or an *n-branch*. An edge is a 1-branch if each tuple in the outer pipe can generate no more than one match. Otherwise, it's an n-branch. We list the generated local counter arrays on the n-branches. The pipe generating tree of the example in the previous section is shown in the top picture of Figure 4.

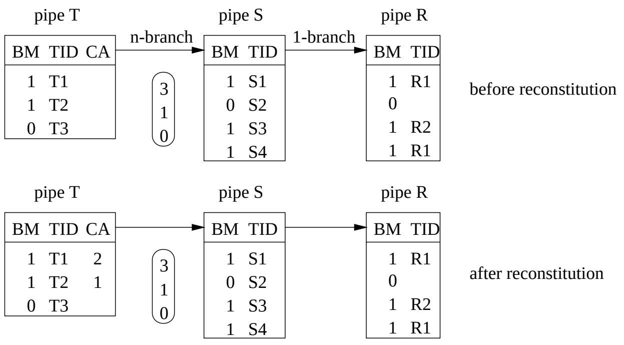


Figure 4: A pipe generating tree of a simple plan

3.2.1 Simple Plan Reconstitution

Definition 3.1: We call the corresponding plan of a pipe generating tree a *simple plan* if all the n-branches are covered in a single path starting from the root and they are all at the beginning of that path. □

The majority of the real world join queries are foreign key join queries. So, simple plans will cover most of the plans generated by an optimizer. We can reconstitute the run-length encoding of tuples in each pipe for simple plans by making one traversal of the path including all the n-branches in a pipe generating tree.

We illustrate the reconstitution algorithm using the example in Figure 4, which happens to a simple plan. The shared bitmap in pipe `S` and `R` already gives the correct breakdown of tuples in these two pipes (since each tuple has a count of either 0 or 1) . So we don't really need counter arrays in these two pipes. To fill in the counter array in pipe `T`, we go through the local counter array associated with the n-branch. Since there are three matches for `T1`, we add the bits of the first three tuples in pipe `S` and assign it to the count for `T1`. We then assign the bit of the next tuple in pipe `S` to `T2` since `T2` has only one match. We don't care about the count for `T3` since its bit is already off. The final result is shown in the lower picture in Figure 4. Observe that we have now reconstituted the join result in the format of a run-length encoding, broken down by the participating tables. For simple plans, the counter arrays in all the pipes are independent of each other. Within each pipe, we can iterate the tuples sequentially. If a tuple's bit is off, it's skipped. Otherwise, it's repeated the number of times specified by its counter, or once if it doesn't exist. The details of the reconstitution algorithm are described in Algorithm 3.1. The cost of the algorithm only depends on the number of n-branches, not the number of pipes in the plan.

For simple plans, many of the bitmaps and counter arrays in the pipes can be shared. Basically, we only need to allocate a new bitmap and a new counter array for all the pipes that have an n-branch going out. For the first pipe (closest to the root) without any n-branches going out, we allocate a new bitmap. Each of the rest of the pipes share the bitmap and/or the counter array with its closest ancestor pipe that has been allocated a bitmap and/or a counter array.

Algorithm 3.1: Reconstitute the join result for simple plans.
Input: The root node of a pipe generating tree (*root*). Each node has a bitmap, a counter array (*ca*) and a tuple array.
Output: A run-length representation of the join result broken down by the pipes.
Method:

```
reconstitute(root) {
  if (root has no n-branches) {
    set root.ca to be the same as the bitmap in root
    return
  }

  Let child be the node that is linked to root by an n-branch
  reconstitute (child)
  //start updating the counter array in root
  m=1
  let local_ca be the local counter array associated with the n-branch
  for the jth tuple in root {
    if (the jth bit of the bitmap in root is on)
      for k= 1 to local_ca[j] {
        if (the mth bit in child is on)
          root.ca[j]=root.ca[j]+child.ca[m]
        m=m+1
      }
    else
      m=m+local_ca[j]
  }
}
```
▮

3.2.2 Reconstitution for General Plans

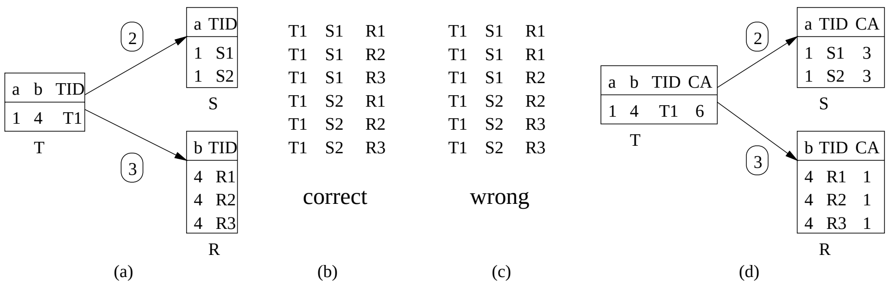


Figure 5: A pipe generating tree of a more general plan

The reconstitution becomes more complicated when each node can have more than one n-branch. Let's take a look at an example in Figure 5(a). Here, the only tuple in pipe `T` generates two matches in pipe S and three matches in pipe R respectively. The correct join result is shown in Figure 5(b). `T1` should have a count of six because of the Cartesian product. Tuples in pipe `S` each should appear three times while tuples in pipe `R` each should appear twice. However, a naive sequential iteration of all the pipes gives the wrong result shown in Figure 5(c). Although the total number of tuples is right, they are not the right combination. The reason for this problem is that now we have two n-branches that can't be covered by a single path from the root. To get the correct join result, we have to align the tuples in pipe S and pipe R properly.

We have a more general algorithm that will cover all the cases. Due to space limit, we omit the details in the paper. The basic idea is to keep an iterating window for each pipe. The iterating window for the root pipe always contains one tuple. The root pipe then sets the iterating windows of it children to include all the matching tuples of the current tuple it's iterating. We then recursively determine the iterating windows of the rest of the pipes. Each pipe keeps iterating the tuples in its iterating window repeatedly until the window is moved. We update the count of each tuple to be the number of times it should be repeated in a single iteration. We have an algorithm that assigns the counter values using two traversals of the pipe generating tree. We show the assignment of the counters in our example in Figure 5(d). Notice that each tuple in pipe `R` is given a count of one, instead of two. While we iterate through the first three copies of `T1`, we get three copies of `S1` and one copy of `R1`, `R2` and `R3` each. Then, since the iterating window of pipe `R` remains the same, its three tuples will be iterated again matching three copies of `T1` and `S2`. Now, we get the correct combination of the join result. Observe that the iteration of pipes are not independent. We always have to start from the root pipe.

We expect this general case to be infrequent. However, if it does occur, there is another benefit we can get using power-pipelining. If we use 1-pipelining for the example in Figure 5, for each of the two matches in pipe `S`, we have to use `T1` to find all the matches in pipe `R`. By using power-pipelining, we only need to find all the matches from `R` for `T1` once. This is because when a tuple is used for probing, we don't care about its count (as long as its bit is on). This can save a lot of redundant probings that will exist in 1-pipelining.

3.3 Overflow Handling

In the previous sections, we have assumed that during a join, the inner pipe always has enough slots to hold all of the matching tuples. In practice, the number of matches generated in a join can vary and a pipe can always overflow no matter how large we set the pipe size. One solution is to keep a linked list of pipes. Every time a pipe overflows, a new pipe is created and added to the linked list. However, that doesn't solve the problem completely. In the extreme case, tuples in an outer pipe can generate so many matching tuples that we won't even have enough memory to store them. Additionally, the good cache behavior of pipelining is based on the assumption that all the data in the pipe can fit in the cache. Thus, we may not want to generate all the matching tuples during each next() call even if we have enough space. Thus we can't assume that we can always finish the processing of all the tuples in an outer pipe in one next() call.

A similar problem exists in 1-pipelining. 1-pipelining always returns one tuple result for each next() function call. Thus if a tuple ever has more than one match, only the first one can be returned immediately. To find the subsequent matches next time, each join node usually keeps some state information. For example, a hash join operator will remember which bucket to search next after finding the first match. The next time next() is called, it can start searching from the saved bucket.

In power-pipelining, we handle overflow by keeping an "active window" for each pipe. Only tuples within the active window are being processed. Initially, a leaf operator will set the active window to include all the tuples in the pipe. If an overflow occurs during join processing, the join operator remembers the current tuple being processed (that's where we should resume processing in the next iteration) in the outer pipe and sets the active window of the outer pipe to include all the tuples it has processed in this iteration. The active window of the inner pipe is set to include all the newly inserted matching tuples. The operator also sets the state of the join to be "incomplete". When this join operator is reached again, it sees the incomplete sign and starts processing from the tuple it previously remembered in the outer pipe. When all the tuples in the outer pipe have been processed, the operator sets its state to be "complete" and allows its child to fill in a new batch of tuples in the outer pipe. The details of the algorithm is described in Algorithm A.2 in Appendix A. The reconstitution algorithm in Section 3.2 still applies. But now it operates within the scope of the active window.

We illustrate our overflow handling method in Figure 6. The top picture is the same example as we used in Figure 4. At that time, we assumed all the pipes are large enough (i.e., no overflow). Now, we deliberately shrink the size of pipe `S` and `R` to two and see how the overflow is handled. After the first two matches are inserted into pipe `S`, it overflows. So the join operator sets the active window of pipe `T` to include `T1` only. The second join doesn't have overflow and finishes processing `S1` and `S2` in a single pass. When pipe `T` is used again, the join operator starts from the tuple it remembers last time (`T1`). This time, the rest of the tuples in pipe `T` can all be processed. The active window of pipe `T` after this iteration includes all the three tuples. So we use two passes to finish the query. After each pass, we can run the reconstitution algorithm

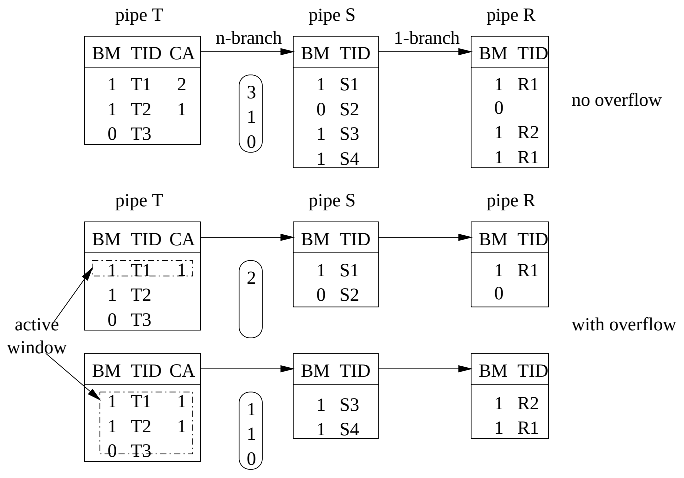


Figure 6: Overflow handling

and update the counter array in pipe T as shown in the bottom two pictures in Figure 6. If we combine the results from the two passes, we get the same join result as in the top picture in Figure 6.

Although overflows can be correctly handled, they require extra work. We can reduce overflows by choosing the proper size for each pipe. We'll discuss this issue in detail in Section 5.

4 Comparing Power-Pipelining and 1-Pipelining

In this section, we summarize the tradeoffs between power-pipelining and 1-pipelining. We first list the advantages of using power-pipelining.

First of all, power-pipelining saves the overhead of function calls since it batches tuple processing. This saving is more significant when most probing tuples have matches. Next, the reconstituted join result is in the form of a run-length encoding. So we get a compact representation automatically. If the join result needs to be sent back to the client or to a different node in a parallel system, we can just send the compact data directly. This allows the server to be freed more quickly and also saves on transmission cost. 1-pipelining would need a separate compression phase in order to achieve the same effect.

Power-pipelining also has its disadvantages. In a sparse join, there will be many tuples with zero matches. These invalid tuples still have to be carried along in subsequent operators. In this case, the benefit of saving function calls may be offset. Also, iterating the compact join result introduces extra cost. If the iteration has to be done on the server, power-pipelining may not be more efficient. In those cases, 1-pipelining may still be the best way to evaluate the query. In the next section, we discuss how we can combine both methods through an optimizer.

5 Issues in the Optimizer

As we have discussed in the previous section, neither power-pipelining nor 1-pipelining dominates the other. So, to get the benefits of both, we want to keep both implementations coexisting and let the query optimizer choose between the two.

Typical join optimizers are cost-based optimizers [SAC$^+$79]. In this section, we adapt the design of an existing join optimizer and let it consider using power-pipelining in a query plan. Our first approach is to use either power-pipelining or 1-pipelining completely in an execution tree. For each subplan, we estimate its costs of using power-pipelining and 1-pipelining. When the full plan is generated, we choose the implementation with the lower cost. To estimate the join cost using power-pipelining, we need to estimate the number of tuples that will be generated in both the outer and inner pipes. We keep that information in each subplan so that it can be used later. We also need to estimate the selectivity of the probing pipe, i.e., the percentage of tuples that have matches. The join cost of using power-pipelining is thus the cost of generating all the tuples in the inner pipe, the cost of iterating tuples in the outer pipe and the cost of updating the bitmap. The cost estimation using 1-pipelining has to account for the cost of function call overheads. The number of function calls a join needs is the number of tuples in the join result. This cost can be ignored in power-pipelining as long as we choose a pipe size that's large enough. If an operand of a join operator is blocking (e.g., the operand used to build a hash table in a hash join), we have to add the reconstitution cost and the cost of iterating the compact join result to the power-pipelining cost. The cost of reconstitution is straightforward from the algorithm. Actually, we don't have to choose the same implementation for the subplan under a blocking operator as for the rest of the plan. We should be free to use 1-pipelining to build a hash table and power-pipelining for hash probing, or the other way round, depending on their costs. This gives us more opportunities for optimization. We summarize the change in join cost estimation in Algorithm 5.1. Notice that when estimating the cost using power-pipelining, there is a term related to join selectivity. The more restrictive the join, the higher the cost. This is because of the overhead of carrying invalid tuples. If the join result has to be sent to other sites, we also need to add the transmission cost to both implementations. The result representation of power-pipelining can be more compact than that of 1-pipelining.

Another issue is choosing the proper size for each pipe. First of all, we have to use a pipe size large enough so that the overhead of function calls can be ignored. However, we can't choose the pipe size to be arbitrarily large. If the total amount of space consumed by all the pipes exceeds the cache size, we will introduce a high cache miss penalty. We also have to choose the relative size of different pipes properly based on the estimation of join selectivity. Intuitively, the relative size between an inner and an outer pipe should correspond to the average number of matches each tuple in the outer pipe has. Proper relative pipe size reduces overflows.

Algorithm 5.1: Estimate Join Cost

Input: Two subplans A and B, their associated costs for both power-pipelining (Cp) and 1-pipelining (C1), their cardinalities size1 and size2. Also, the estimated number of tuples to be filled in the outer pipe.

Output: The combined plan AB, its costs and cardinality, number of tuples in the inner pipe.

Method:

```
Decide the optimal join algorithm to combine the two parts A and B.
Estimate the 1-pipelining join cost as before.
Estimate the join result size N.
Add N*(cost of a function call) to the 1-pipelining cost.

Estimate the join selectivity of the outer pipe.
Estimate the number of tuples in the inner pipe.
AB.Cp=|inner|*(per probe cost)+|outer|*(iteration cost per tuple + (1-join selectivity)*(set bitmap cost))

if (A is blocking) {
   AB.C1=AB.C1 + min(A.Cp+reconstitution cost+iterating cost, A.C1)  (*)
   AB.Cp=AB.Cp + min(A.Cp+reconstitution cost+iterating cost, A.C1)
}
else {
   AB.C1=AB.C1 + A.C1                           (**)
   AB.Cp=AB.Cp + A.Cp
}

if (B is blocking)
   AB.C1=AB.C1 + min(B.Cp+reconstitution cost+iterating cost, B.C1)  (*)
   AB.Cp=AB.Cp + min(B.Cp+reconstitution cost+iterating cost, B.C1)
}
else {
   AB.C1=AB.C1 + B.C1                           (**)
   AB.Cp=AB.Cp + B.Cp
}
```

▮

A more aggressive approach is to mix power-pipelining with 1-pipelining *within* pipelinable operators. It's possible to switch from power-pipelining to 1-pipelining during the execution. We can first reconstitute the join result and iterate all the tuples one by one. However, it is difficult to switch from 1-pipelining to power-pipelining. This is because in 1-pipelining, each pipe has only enough space to hold one tuple result. To accumulate tuples into a wider pipe, we can't simply copy the tuple pointers since they are going to be invalid on the next iteration. Instead, we have to copy the full tuple to the wider pipe. As a result, it's usually not beneficial to switch from 1-pipelining to power-pipelining within a pipeline. To incorporate this idea, we need to change Algorithm 5.1 a little bit. The computation of AB.Cp is still the same. But to compute AB.C1, we take the lower cost of the subplan using power-pipelining and 1-pipelining even if the subplan is non-blocking. Basically, we change each of the two lines marked with (**) to the first line above it marked with (*).

The approaches proposed in this section do not increase the join searching space. We just increase the estimation cost a little bit when combining two subplans.

6 Experimental Results

6.1 Implementation Details

We implemented both power-pipelining and 1-pipelining in the execution engine of the Columbia Main Memory Database System. More specifically, we implemented the leaf operator, two join operators, a scalar aggregate operator and the root operator. We have two kinds of join operators. The first one is bitmap join. Since we transform all the foreign key values into IDs of the referenced tuple, we can first build a bitmap on the foreign relation and then scan through the referring relation and check the bitmap using the foreign key IDs. Bitmap join is efficient when the foreign relation is relatively small. The second join method is indexed nested loop join. The index we used is the cache-sensitive search tree [RR99], which has better searching performance than B+-trees using the same amount of space. Our implementation handles pipe overflows. We also implemented the reconstitution algorithm (with "active window"). We inlined all the functions that are inlinable. Most of the inlining can be done automatically by the compiler. The only exception is the index probing in the nested loop join, which we had to inline by hand (for both power-pipelining and 1-pipelining). The compiler we used is Sun's native C++ compiler.

We have two different implementations of pipelining. So there is the important issue of version maintenance. Instead of duplicating the code for the two implementations, we keep only one version of all the files and use *#ifdef* for different implementations. As a result, both implementations can share a lot of the code that's common to both.

Since we will never choose a pipe size larger than 2^{16}, we use a *short int* to represent each count. This reduces the amount of data to be transmitted.

We implemented a cache simulator ourselves and instrumented our code (using `#ifdef SIMULATOR` macros) to log all memory accesses (more accurately, memory reads). We report the number of data accesses and cache misses in the secondary level cache in our results.

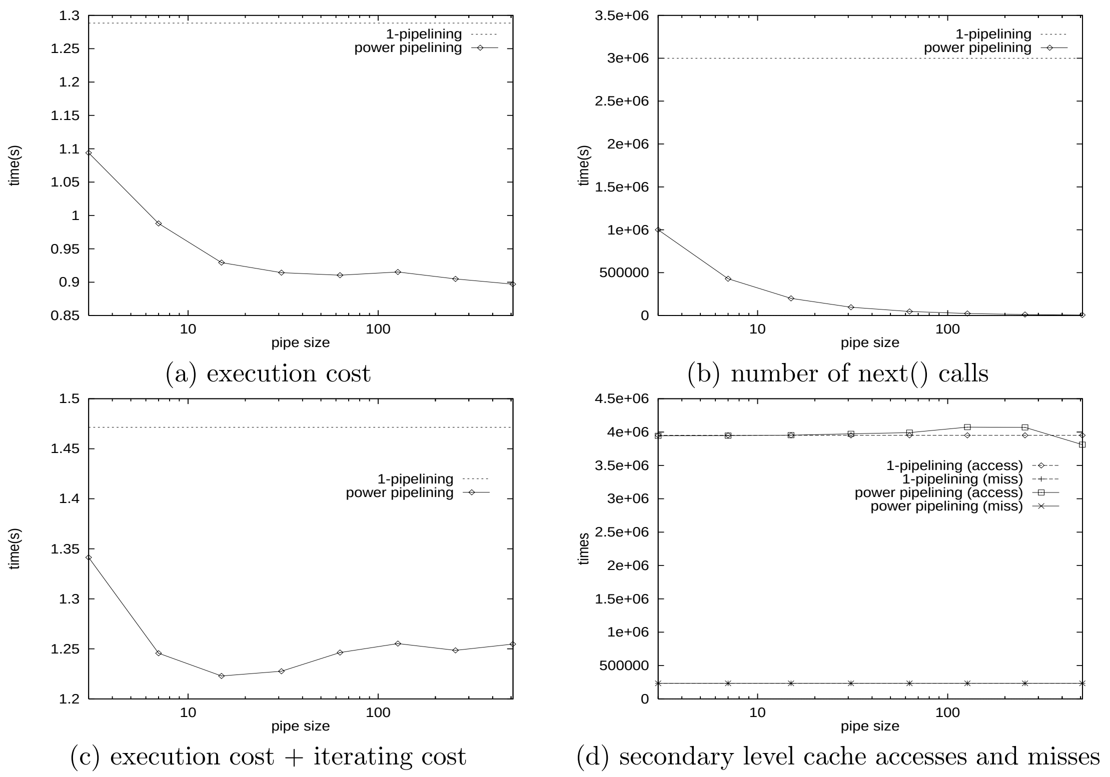
In all our tests, we use tables consisting of four fields, all of which are integers. Each field is populated randomly within its value range. We ran our experiments on an Ultra Sparc II machine (296MHz, 1GB RAM) running Solaris 2.6. It has a 16KB on-chip cache and a 1MB secondary level cache. We repeated each test three times and report the minimal time.

6.2 Results

```
Query 1:
select a.col2, b.col2, c.col2
from T1M a, T10K b, T2K c
where a.col4=b.col1 and
      b.col4=c.col1
```

```
Query 1(a):
select sum(a.col2), sum(b.col2), sum(c.col2)
from T1M a, T10K b, T2K c
where a.col4=b.col1 and
      b.col4=c.col1
```

Our first test uses Query 1. The number in the table name indicates its cardinality. `T1M.col4` is a foreign key column referring to table `T10K` and `T10K.col4` is a foreign key column referring to table `T2K`. The



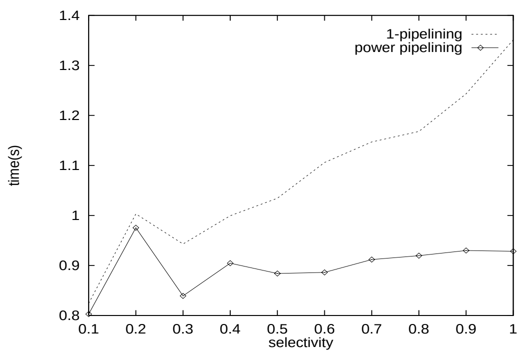
(a) execution cost (b) number of next() calls

(c) execution cost + iterating cost (d) secondary level cache accesses and misses

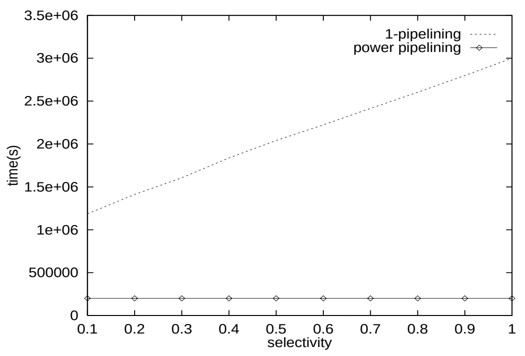
Figure 7: 3-way bitmap join

optimizer decides to use bitmap joins to join the three tables together since all the foreign key values have been transformed into foreign tuple IDs. When building the bitmap, new tuples are created to include only the needed columns and are stored along with the bitmap. In this test, all the three pipes have the same size in power-pipelining. We vary the size of the pipe and present the result in Figure 7. We first compare the "pure" execution cost of the two methods by discarding all the results after they are computed in the root node. Figure 7(a) shows that power-pipelining can be 50% better than 1-pipelining. This is explained by the saving in function calls as shown in Figure 7(b). We then try to let the server iterate all the join results. So we evaluate Query 1(a) and uses a scalar aggregate operator on the top to add all the tuples one by one. The iterating of tuples in power-pipelining requires checking the bitmaps. In this example, since all the three bitmaps are shared, we only need to check one bitmap. Figure 7(c) shows that, including the extra iterating cost, power-pipelining is still about 20% better. The incremental benefit of power-pipelining decreases when the pipe size gets larger. A moderate pipe size (15) can achieve most of the benefit. Figure 7(d) shows our cache simulation result. Although power-pipelining needs to access some additional data such as the bitmap in the pipes, these data stay in the on-chip cache most of the time. As a result, power-pipelining doesn't increase the number of data accesses and misses in the secondary level cache and retains the good cache conscious behavior of 1-pipelining.

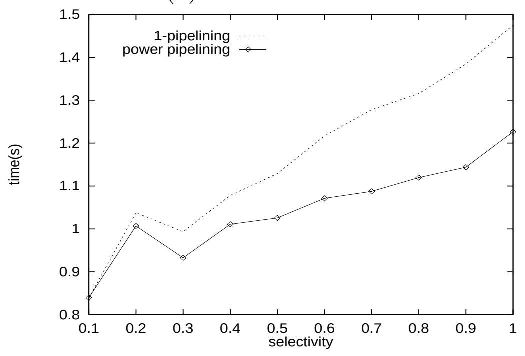
In our second test, we change Query 1 by adding a local selection predicate `T2K.col3 > X`. The optimizer still decides to use bitmap joins. The local selection predicate is evaluated while building the bitmap for table `T2K`. When building the bitmap on table `T10K`, the operator checks the bitmap on table `T2K` first. So all the information in the bitmap on table `T2K` is forwarded to the bitmap on table `T10K`. In some sense, this plan has a composite inner. We vary the value of `X` to control the selectivity of the query. We fix the pipe size to be 15 for power-pipelining. The result is presented in Figure 8. As we can see, power-pipelining still has some advantages over 1-pipelining even when the selectivity is low. There is a bump when the selectivity is 0.2 for both implementations. This is caused by cache conflicts between data and instructions. A few similar bumps exist in later experiments also. We want to investigate this further in the future. Figure 8(d) shows that selectivity only affects the number of data accesses, not the number of cache misses. This is because



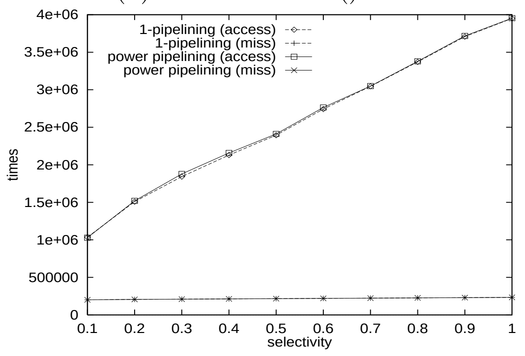
(a) execution cost



(b) number of next() calls



(c) execution cost + iterating cost



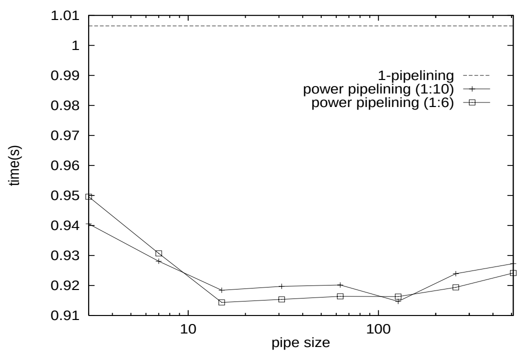
(d) secondary level cache accesses and misses

Figure 8: 3-way bitmap join, with local selection

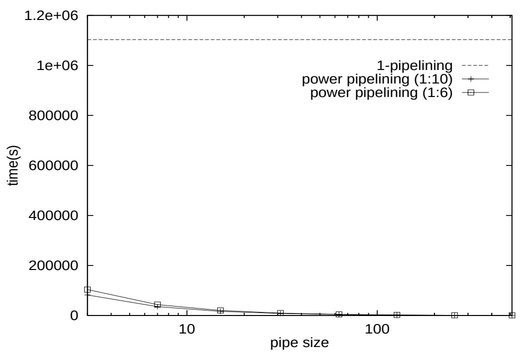
we always have to scan the largest table T1M.

```
Query 3:
select a.col2, b.col2
from T2M a, T200K b
where a.col4=b.col1 and
      b.col3>100,000
```

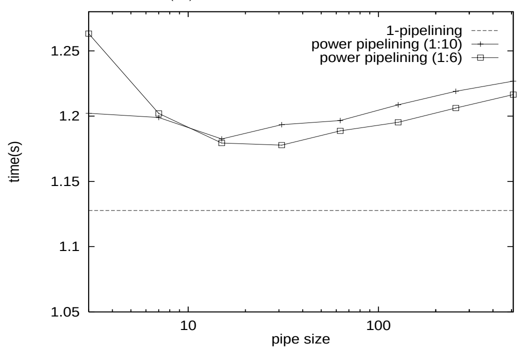
Our third test uses Query 3. T200K is relatively large and it's more expensive to build a bitmap on it. Also, there is a local selection predicate on T200K which eliminates half of its tuples. Since there is an index on T2M.col4, the optimizer decides to use an indexed nested loop join for the query. This avoids a full scan of T2M. Figure 9 shows the result. On average, each tuple from T200K will match ten tuples from T2M. To reduce overflow, we choose the size of the inner pipe to be ten times or six times larger than that of the outer pipe for power-pipelining. The x-axis represents the size of the smaller pipe. In Figure 9(a), the benefit of using power-pipelining is a little bit more than 10%. Indexed nested loop join needs to traverse a search tree to find the first match. So its cost of computing each tuple is more expensive than that of the bitmap join. That's why the relative benefit of saving function calls is smaller here. Hash indexes are much faster than tree-based indexes at the expense of more storage space. If hash indexes are used for indexed nested loop join (assuming there is enough memory), the relative benefit of saving function calls will be more significant. The cost of power-pipelining first decreases when we increase the pipe size. However, it starts to increase when the pipe size gets too large. This is because not all the pipes can now fit in the cache and the cache miss ratio is higher. The line for power-pipelining with a larger pipe ratio is better when the pipe size is small, but it's worse when the pipe size is relatively large. Figure 9(c) compares the performance of the two method with the additional cost of iterating the join result (achieved by computing the scalar aggregates). Power-pipelining performs worse than 1-pipelining. The reason is that the two pipes have to be iterated differently. Tuples in the outer pipe have to be repeated as many times as specified by the counters. For simple plans with 1-branches, many bitmaps and counter arrays are shared and the iterating cost will



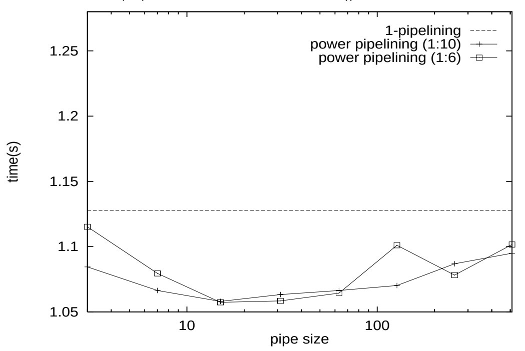
(a) execution cost



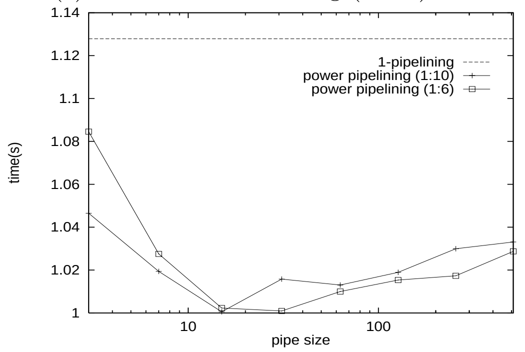
(b) number of next() calls



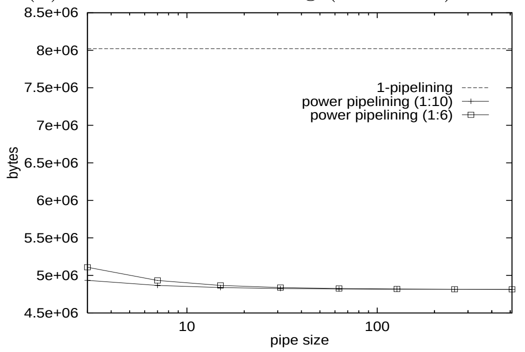
(c) execution + iterating (basic) cost



(d) execution + iterating (advanced) cost



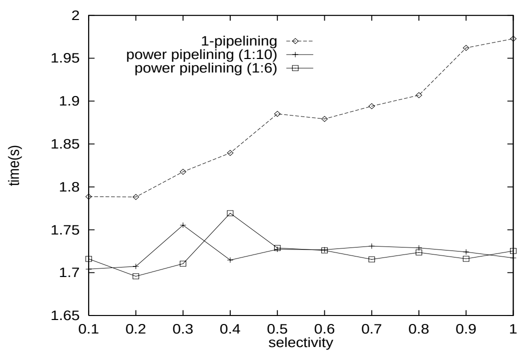
(e) execution + copying result



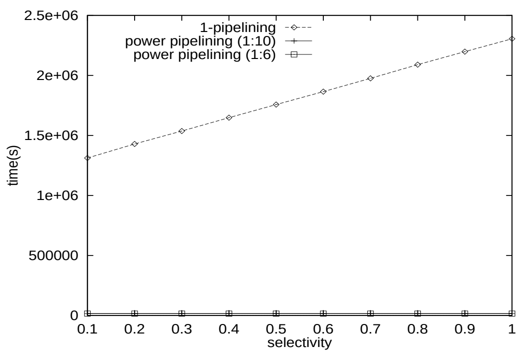
(f) data communication cost

Figure 9: 2-way index join, with local selection

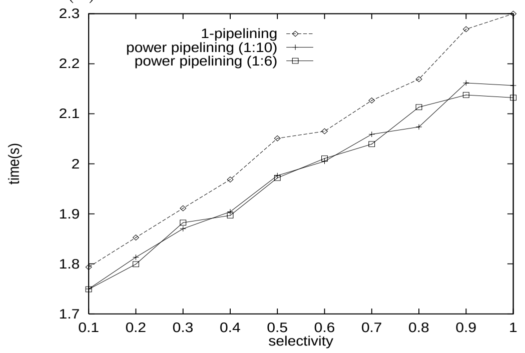
be amortized. In Figure 9(c), power-pipelining iterates all the tuples one by one. To calculate the scalar aggregates, we can use a more "advanced" iterating method. For example, to compute `SUM`, we can add the product of each value and its count to the aggregate value (instead of adding the same value multiple times). We show the result of using the advanced iterating method in Figure 9(d). Now, power-pipelining is better than 1-pipelining. The advanced iterating method can also be used for aggregation queries with `group-bys`, as long as we compute one group at a time. As we have discussed before, another important benefit of using power-pipelining is compact representation of join result. If the join result needs to be iterated at a different site, sending the compact join result can save both the execution and the transmission cost. Figure 9(e) shows the cost including copying the join result. Power-pipelining is now 13% better than 1-pipelining. If more columns from the outer pipe are needed, the improvement will be more significant. Figure 9(f) compares the amount of data that needs to be transmitted. Power-pipelining sends 40% less data than 1-pipelining. 1-pipelining can also compress its join result, but this requires extra cost. Although not shown here, the cache simulation result is similar to what we have seen before.



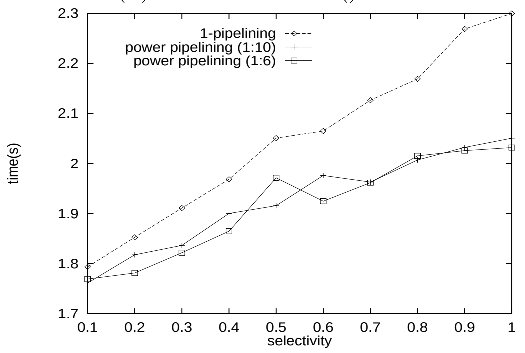
(a) execution + reconstitution cost



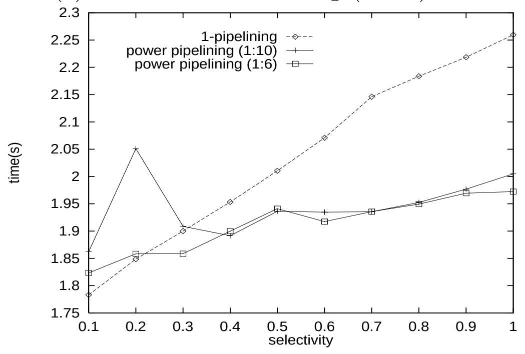
(b) number of next() calls



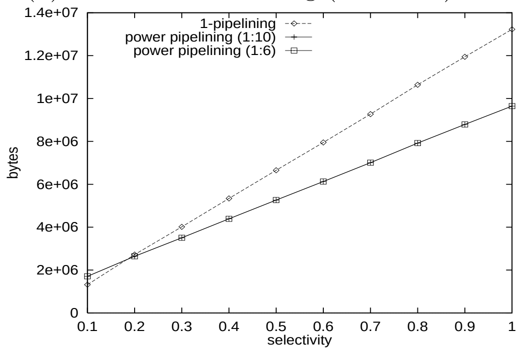
(c) execution + iterating (basic) cost



(d) execution + iterating (advanced) cost



(e) execution + copying result



(f) data communication cost

Figure 10: 3-way mixed join, with local selection

```
Query 4:
select a.col2, b.col2, c.col2
from T2M a, T200K b, T10K c
where a.col4=b.col1 and
      a.col3=c.col1 and
      b.col3>100,000 and
      c.col3>X
```

We use Query 4 for our final test. `T2M.col3` is a foreign key column referring to table `T10K`. The optimizer decides to join `T200K` with `T2M` using indexed nested loop join first. `T10K` is then joined together using bitmap join. We fix the pipe size for table `T200K` to be 31 and use a pipe size ten times or six times larger for both pipe `T2M` and `T10K`. We vary `X` to control the selectivity of the join. This plan is a simple plan and we need to use the algorithm we described in Section 3.2.1 to reconstitute the join result. In Figure 10(a), we include the reconstitution cost for power-pipelining. Even with this additional cost, power-pipelining can be up to 15% better than 1-pipelining. Since the pipes for `T2M` and `T10K` share the same bitmap, the iterating cost is amortized. Power-pipelining performs better using both the basic and the advanced iterating method as shown in Figure 10(c) and 10(d). When the selectivity is low, many tuples in the outer pipe will have only one matching tuple. As a result, the run-length encoding takes more space than the normal representation. This is why when the selectivity is low, power-pipelining takes longer to copy the result and also has to send more data as shown in Figure 10(e) and 10(f). An optimizer should catch this and choose to use 1-pipelining instead when the selectivity is low.

7 Conclusion

Conventional pipelining has existed in query execution engines for almost twenty years. Although its one-tuple-at-a-time mechanism makes the implementation easy, it introduces problems such as the overhead of function calls. In this paper, we propose a power-pipelining method that takes advantage of a wider pipe while retains the good cache behavior of conventional pipelining. In addition to saving function calls, power-pipelining also compresses the join result automatically. We demonstrated that our method is feasible and can provide significant benefit in the context of a main memory database system.

We believe that widening the pipe is the right approach to improving conventional query execution engine performance. Given the trend that future data processing will be memory resident and data intensive, techniques such as power-pipelining should be incorporated into main memory execution engines.

Acknowledgments

We thank Xiaotao Wu for part of the implementation used for this paper, Lei Liu for the discussion on the iterating method in Section 3.2 and John Grogan for the implementation of the cache simulator.

References

[AHK85] Arthur C. Ammann, Maria Butrico Hanrahan, and Ravi Krishnamurthy. Design of a memory resident DBMS. In *Proceedings of the IEEE COMPCOM Conference*, pages 54–57, 1985.

[BBC+98] Phil Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, H. V. Jagadish, Michael Lesk, Dave Maier, Jeff Naughton, Hamid Pirahesh, Mike Stonebraker, and Jeff Ullman. The Asilomar report on database research. *ACM Sigmod Record*, 27(4), 1998.

[CAB+81] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system R. *Communication of the ACM*, 24(10):632–646, 1981.

[CLH98] Trishul M. Chilimbi, James R. Larus, and Mark D. Hill. Improving pointer-based codes through cache-conscious data placement. Technical report 98, University of Wisconsin-Madison, Computer Science Department, University of Wisconsin-Madison Madison, Wisconsin 53706, 1998.

[Fre95] Clark D. French. “One size fits all” database architectures do not work for DDS. In *Proceedings of the ACM SIGMOD Conference*, pages 449–450, 1995.

[Fre97] Clark D. French. Teaching an OLTP database kernel advanced data warehousing techniques. In *Proc. IEEE Int'l Conf. on Data Eng.*, pages 194–198, 1997.

[GMS86] Hector Garcia-Molina and Kenneth Salem. High perfromance transaction processing with memory resident data. In *Proceedings of the International Workshop on High Performance Transaction Systems*, 1986.

[GMS92] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering*, 4(6):509–516, 1992.

[Gra94] Goetz Graefe. Volcano, an extensible and parallel query evaluation system. *IEEE Transactions on knowledge and data enginnering*, 6(6):934–944, 1994.

[Hag86] R. B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computing*, C-35:839–842, 1986.

[HCL+90] L. Haas, W. Chang, G. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M.J. Carey, and E. Shekita. Starburst mid-flight: as the dust clears. *IEEE Transactions on knowledge and data engineering*, 2(1):143, 1990.

[JLRS94] H.V. Jagadish, Daniel Lieuwen, Rajeev Rastogi, and Avi Silberschatz. Dali: A high performance main memory storage manager. In *Proceedings of the 20th VLDB Conference*, pages 48–59, 1994.

[LC87] Tobin J. Lehman and Michael J. Carey. A recovery algorithm for a high performance memory-resident database system. In *Proceedings of the ACM SIGMOD Conference*, pages 104–117, 1987.

[LN88] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *International Symposium on Databases in Parallel and Distributed Systems*, pages 177–189, 1988.

[LSC92] Tobin J. Lehman, Eugene J. Shekita, and Luis-Felipe Cabrera. An evaluation of starburst's memory resident storage component. *IEEE Transactions on knowledge and data enginnering*, 4(6):555–566, 1992.

[Mac97] Roger MacNicol, Summer, 1997. personal communication at Sybase IQ.

[RR99] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th VLDB Conference*, 1999.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomsa G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference*, pages 23–34, 1979.

[Sof97] TimesTen Performance Software. Main-memory data management technical white paper (available from http://www.timesten.com). 1997.

[Sto80] M. Stonebraker. Retrospection on a database system. *ACM Transactions on Database Systems*, 5(2):225, 1980.

[WK90] Kyu-Young Whang and Ravi Krishnamurthy. Query optimization in a memory-resident domain relational calculus database system. *ACM Transactions on Database Systems*, 15(1):67–95, 1990.

A Algorithms

Algorithm A.1: Join.next() (without overflow handling)
Input: outer_pipe, inner_pipe and a local counter array local_ca.
Output: return 1 if there is join result; otherwise, return 0.

```
Join.next()  {
  outer_operand.next()    //ask the outer child to fill in the outer pipe
  if (outer pipe empty) //no more tuples
    return 0
  for the ith tuple Ti in outer_pipe {
    if (the ith bit in outer_pipe is on) {
      find all the tuples from the inner operand that match Ti.
      insert all the matches into inner_pipe and set their bits to on
      assign Ti's matching number to local_ca[i]
      if (the matching count is zero)
        turn off the ith bit in outer_pipe
    }
  }
  return 1
}
```

▌

Algorithm A.2: Join.next() (with overflow handling)
Input: outer_pipe, inner_pipe and a local counter array local_ca.
Output: return 1 if there is join result; otherwise, return 0.

```
Join.next()  {
  if (complete) {
    outer_child.next();
    if (outer pipe empty)
      return 0   //no more results
    save the initial active window of the outer pipe
  }
  else   // the state is incomplete
    restore the active window of the outer pipe and the operator state using the information previously saved

  for the ith tuple Ti within the active window in outer_pipe {
    if (the ith bit in outer_pipe is on) {
      while (Ti has a matching tuple S) {
        if (there is space in the inner pipe) {
          insert S into the inner pipe
          increase Ti's matching number
        }
        else {  // we have an overflow here
          assign Ti's matching number to local_ca[i]
          save the position of tuple Ti in the outer pipe
          save other operator related state
          reset the current active window of the outer and inner pipe properly
          set state to be incomplete
          return 1
        }
      }
      assign Ti's matching number to local_ca[i]
      if (the matching count is zero)
        turn off the ith bit in outer_pipe
    }
  }
  //finished all the tuples in the active window
  set state to complete
  return 1
}
```

▌