Conjunctive Selection Conditions in Main Memory

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

ABSTRACT

We consider the fundamental operation of applying a conjunction of selection conditions to a set of records. With large main memories available cheaply, systems may choose to keep the data entirely in main memory, in order to improve query and/or update performance.

The design of a data-intensive algorithm in main memory needs to take into account the architectural characteristics of modern processors, just as a disk-based method needs to consider the physical characteristics of disk devices. An important architectural feature that influences the performance of main memory algorithms is the branch misprediction penalty. We demonstrate that branch misprediction has a substantial impact on the performance of an algorithm for applying selection conditions.

We describe a space of "query plans" that are logically equivalent, but differ in terms of performance due to variations in their branch prediction behavior. We propose a cost model that takes branch prediction into account, and develop a query optimization algorithm that chooses a plan with optimal estimated cost. We also develop an efficient heuristic optimization algorithm.

We provide experimental results for a case study based on an event notification system. Our results show the effectiveness of the proposed optimization techniques. Our results also demonstrate that significant improvements in performance can be obtained by applying a methodology that takes branch misprediction latency into account.

1. INTRODUCTION

Main memories are getting bigger and cheaper. It is now feasible for many applications to store the application data completely in a main memory database, in order to improve query and/or update performance.

Many traditional database algorithms need to be reconsidered for main memory databases. In this paper, we focus on one commonly-used database operation, namely applying a conjunction of selection conditions to a set of database records. One wishes to obtain those records satisfying the conjunction of conditions in as efficient a way as possible.

Our discussion will take the perspective that the application's data is stored in a main memory database. However, the problem we shall address is also relevant for information processing systems that are not considered "traditional" database systems. Examples include search engines, event notification systems, and network management systems. In each of these types of systems, one commonly poses queries involving the selection of records satisfying a conjunction of conditions.

In a disk-based database, it is usual to consider the performance parameters of the disk devices when designing database algorithms. For example, the high cost of random I/O compared with sequential I/O leads to algorithms that process the data in physical order. The relatively large size of a disk block leads to algorithms that try to cluster related data into disk-block sized units.

In a main-memory database we face similar design criteria, although the device characteristics are different. A feature with a significant impact on algorithm design is the delay induced when the CPU executes a conditional branch instruction and predicts the outcome incorrectly (i.e., the branch misprediction penalty). All else being equal, algorithms that have fewer branch mispredictions are likely to perform better than alternatives.

In this paper we consider how to design efficient algorithms for applying a conjunction of selection conditions given the characteristics of the CPU and memory hierarchy. We show that the branch misprediction penalty can have a significant impact on the performance of an algorithm.

We propose a class of algorithms that we consider as potential "plans" for combining selection conditions. To address the branch prediction issue, we develop a cost model that takes branch prediction into account. We then develop an exhaustive query optimization algorithm for choosing among

*This research was supported by NSF grants IIS-98-12014, IIS-01-20939, EIA-98-76739, and EIA-00-91533. Part of this work was performed while the author was visiting the INRIA Rocquencourt research institute.

such plans in a cost-based fashion, using dynamic programming. We also derive results that allow us to safely prune the search space of potential plans. We then develop a heuristic optimization method with lower complexity that performs well in practice.

We present a case study of the proposed methods in the context of an event-based notification system [16, 8]. Our experimental results show that significant performance improvements can be obtained. Our optimization algorithm and its cost model are validated against actual performance.

Past work has identified that branch misprediction has a significant impact on modern database systems [1]. To our knowledge, the present paper provides the first discussion of methods for avoiding branch misprediction penalties in database systems.

2. BACKGROUND

Modern CPUs have a pipelined architecture in which many instructions are active at the same time, in different phases of execution. Conditional branch instructions present a significant problem in this context, because the CPU does not know in advance which of the two possible outcomes will happen. Depending on the outcome, different instruction streams should be read into the pipeline.

CPUs try to *predict* the outcome of branches, and have special hardware for maintaining the branching history of many branch instructions. Such hardware allows for improvements of branch prediction accuracy, but branch misprediction rates may still be significant. Branches that are rarely taken, and branches that are almost always taken are generally well-predicted by the hardware. The "worst-case" branch behavior is one in which the branch is taken roughly half of the time, in a random (i.e., unpredictable) manner. In that kind of workload, branches will be mispredicted half of the time.

A mispredicted branch incurs a substantial delay. [1] reports that the branch misprediction penalty for a Pentium II processor is 17 cycles.

As a result, one might aim to design algorithms for "kernel" database operations that exhibit good branch-prediction accuracy on modern processors [10]. In fact, this is precisely our approach.

Future architectures, such as Intel's IA-64, support a technique called "predication" that converts control dependencies (i.e., conditional branches) into data dependencies. This technique allows the elimination of some branch instructions. However, it is not always beneficial to use it [7]; sometimes the original branching code is more efficient. Thus we expect branch misprediction penalties to continue to be a significant issue for the next generation of architectures.

There has been some past work on main memory database performance. Since pointer following is inexpensive in a main-memory database, it can pay to store attribute values as pointers to some external piece of allocated memory, often called a *domain* [17, 23]. Specialized algorithms for query processing in main-memory databases have been proposed in [17]. In [20], the authors suggested several ways to improve the cache reference locality of query processing operations such as joins and aggregations. [3] proposes improving cache behavior by storing tables vertically and by using a cache conscious join method. Cache-sensitive indexes for main memory databases are described in [18, 19].

It has been observed that specialized memory-resident techniques allow substantial performance gains over buffer-resident data in a disk-based system [9, 13, 14]. More recently, [2] describes ways to organize pages in a disk-based database system so that database operations give good CPU performance when the pages are memory resident in the database buffer.

3. COMBINING SELECTIONS

We define the *selectivity* of a condition applied to a table to be the proportion of records in the table satisfying the condition. This definition applies whether we're testing a single condition or a conjunction of conditions. Since one typically does not know the exact selectivities in advance, one performs query optimization using estimates of the selectivities. For simplicity of presentation we assume that the selectivities are independent, so that one can multiply estimates of the single-condition selectivities to get joint selectivity estimates. Non-independent selectivities can also be handled by our techniques; see Appendix B.

Suppose we have a large table stored as a collection of arrays, one array per column, as advocated in [3].[1] The column datatypes are assumed to have fixed length. (Variable length attribute types can use the array representation by introducing an extra level of indirection, storing pointers in the array.) Let's number the arrays `r1` through `rn`. We wish to evaluate a number of selection conditions on this table, and return pointers (or offsets) to the matching rows.

Suppose the conditions we want to evaluate are `f1` through `fk`. For simplicity of presentation, we'll assume that each `fi` operates on a single column which we'll assume is `ri`. (The methods developed in this paper are not dependent on the assumption that the functions test just a single argument, or that a column is used in a single function.) So, for example, if `f1` tests whether the first attribute is equal to 3, then both the equality test and the constant 3 are encapsulated within the definition of `f1`. We also assume that functions are well-defined in a self-contained way, in the sense that they always execute without error for any possible parameter value. For example, if `f2` dereferences a pointer that is not guaranteed to be non-null, then `f2` must also encapsulate a precondition testing whether the pointer is null. `f2` cannot rely on `f1` testing that pointer, say, because we intend to reorder the execution of the functions. Functions are discussed at more length in Appendix D.

3.1 Context

Our discussion assumes that the cost of processing the selections is a significant cost within the overall query, and

[1] If we have a single array of rows, as opposed to an array per column, the formulation of the problem is the same. The disadvantage of row-wise storage is that it has poor data reference locality for scans that consult just a few columns.

therefore worth optimizing. This assumption is certainly true when the selections constitute the entire query. When the selections form the initial step of a more complex query, processing the selections may still be a significant (or even dominant) cost since a selective selection operation will need to consult many more records than operations applied after the selection.

We describe three typical contexts in which a set of selection conditions is applied. In the first context, we simply apply the conditions to each record in the underlying table. This approach would be used if indexes are not helpful, either because we lack the required index, or because the condition selects such a large proportion of the records that it is not worth the overhead of using the index.

In the second context, we identify one (or more) of the selection conditions as corresponding to an indexed attribute; using the index can speed up processing. In the third context, a selection condition is applied to a "dimension" table referenced by a foreign key in the main "fact" table. Preprocessing the dimension table can improve efficiency.

As we shall see, each of the contexts has a common structure: There is a loop that iterates over all (partially matching) records, and inside the loop is code to (a) test the records for the remaining conditions, (b) AND the results together, and (c) add qualifying record-ids to the answer list.

The straightforward way to code the selection operation applied to all records (context 1) would be the following. The result is returned in an array called `answer`. In each algorithm below, we assume that the variable `j` has been initialized to zero.

```
/* Basic Algorithm Structure */
for(i=0;i<number_of_records;i++) {
  if(f1(r1[i]) AND ... AND fk(rk[i]))
    {answer[j++] = i;}
  }
```

Alternatively, suppose that `f1` was a condition that could be evaluated efficiently using an index on `r1` (context 2). For example, `f1` might be an equality test, and using an index on `r1` we may be able to obtain an array `matches` of offsets `i` of records satisfying `f1(r1[i])`. Then the remaining conditions can be tested using the following code.

```
/* Index Algorithm Structure */
for(m=0;m<number_of_matches;m++) {
  i=matches[m];
  if(f2(r2[i]) AND ... AND fk(rk[i]))
    {answer[j++] = i;}
  }
```

Indexes may be combined by intersecting `match` arrays.

It is common for queries over a fact table in a data warehouse to place selections on dimension tables (context 3). Suppose `r1` was a foreign key (i.e., offset) to a dimension table, and that `f1` was a selection condition on some column `c` of the dimension table. Then `f1(r1[i])` could be written as `g1(c[r1[i]])`. Since dimension tables are generally small, it may pay to evaluate `g1` on all rows of `c` in advance, and store the result in a temporary array `t`. (This saves repetitive execution of `g1` on duplicate values.) Thus we could modify the basic algorithm structure to perform the selection as

```
/* Preprocess Dimension Table */
for(i=0;i<records_in_c;i++){t[i]=g1(c[i]);}
for(i=0;i<number_of_records;i++) {
  if(t[r1[i]] AND ... AND fk(rk[i]))
    {answer[j++] = i;}
  }
```

3.2 Implementing the Loop

In the following discussion we'll use the code from the first context, i.e., applying the selection conditions to all records one by one. However, similar principles apply to the other contexts. Translated into `C`, the code for the inner loop might be:

```
/* Algorithm Branching-And */
for(i=0;i<number_of_records;i++) {
  if(f1(r1[i]) && ... && fk(rk[i]))
    {answer[j++] = i;}
  }
```

The important point is the use of the `C` idiom "`&&`" in place of the generic "`AND`". (See Appendix A for a discussion of how `&&` is typically compiled into assembly language containing conditional branch instructions.) This implementation saves work when `f1` is very selective. When `f1(r1[i])` is zero, no further work (using `f2` through `fk`) is done for record `i`. However, the potential problem with this implementation is that its assembly language equivalent has `k` conditional branches. If the initial functions `fj` are not very selective, then the system may execute many branches. The closer each selectivity is to 0.5, the higher the probability that the corresponding branch will be mispredicted, yielding a significant branch misprediction penalty. (Recall the discussion of branch prediction effectiveness in Section 2.) An alternative implementation uses logical-and (`&`) in place of `&&`:

```
/* Algorithm Logical-And */
for(i=0;i<number_of_records;i++) {
  if(f1(r1[i]) & ... & fk(rk[i]))
    {answer[j++] = i;}
  }
```

Because the code fragment above uses logical "`&`" rather than a branching "`&&`", there is only one conditional branch in the corresponding assembly code instead of `k`. (Again, see Appendix A for a discussion of how `&` is compiled into assembly language.) We may perform relatively poorly when `f1` is selective, because we always do the work of `f1` through `fk`. On the other hand, there is only one branch, and so we expect the branch misprediction penalty to be smaller.

The branch misprediction penalty for that one branch may still be significant when the combined selectivity is close to

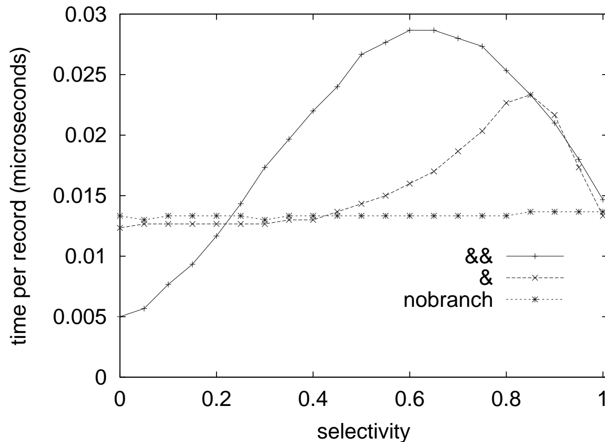

Figure 1: Three implementations: Pentium.

0.5. The following loop implementation has *no* branches within the loop.

```
/* Algorithm No-Branch */
for(i=0;i<number_of_records;i++) {
  answer[j] = i;
  j += (f1(r1[i]) & ... & fk(rk[i]));
  }
```

Note that we would not expect an optimizing compiler to be able to transform one of these plans into another. Most importantly, such transformations are not valid in the general case. For example, in the condition `(A && B)`, `A` may check that a pointer is not null, while `B` dereferences that pointer. Executing `(A && B)` makes sense, while executing `(A & B)` would cause an error if the pointer was null. While our assumption about functions does make `(A & B)` valid in the case where `A` and `B` represent functions `fi` and `fj`, it is not possible to communicate such information to modern compilers. Further, even if one was to extend the compiler with such a mechanism, the decision on whether to rewrite the code depends on database-level metadata, such as condition selectivities, that are not generally available to the compiler.

To see the difference between these three methods, we implemented them in `C` and ran them on a 750Mhz Pentium III under Linux, and a 300Mhz UltraSparc IIi under Solaris. In the following experiment, we used $\mathtt{k} = 4$ and let all of the `rj` arrays be offsets into an array `t` of `char`s of size 5000. Elements of `t` are either 1 or 0, simulating the preprocessing of conditions on dimension tables. The `fj` functions are then lookups in `t`. We ran several thousand scans over four arrays of size 3000, using the same `t` array. That way, both `t` and the arrays are in the L1 cache and the experiments will not reflect delays due to cache misses. (We briefly address caching issues Appendix D.) The code was compiled with `gcc` under maximum optimization, with several `register` hints present in the code.

Figure 1 shows the Pentium results. (See Appendix C for the Sun results.) While both architectures show some dependence on the selectivity, the Pentium results are more sensitive to the selectivity because the branch misprediction penalty is higher on that architecture [25]. The time per record is shown in microseconds on the vertical axis, measured against the probability that a test succeeds. The probability is controlled by setting an appropriate threshold for an element of the `t` array to be randomly set to 1. All functions in this graph have the same probability.

Our preliminary analysis of the three implementations is borne out by this graph. For low selectivities, the branching-and implementation does best by avoiding work, and the one branch that is frequently taken can be well-predicted by the machine. For intermediate selectivities, the logical-and method does best. However, when the combined selectivity gets close to 0.5, the performance worsens. The no-branch algorithm is best for nonselective conditions; it does more "work" but does not suffer from branch misprediction.

Each of the three implementations is best in some range, and the performance differences are significant. On other ranges, each implementation is about twice as bad as optimal. Thus we will need to consider in more depth how to choose the "right" implementation for a given set of query parameters.

Looking at the performance numbers, one might wonder why we care about per-record processing times that are fractions of a microsecond. The reason we care is that this cost is multiplied by the number of records, which may be in the tens or hundreds of millions. When we don't have an index, we have no choice but to perform a full scan of the whole table. Even when we're scanning fewer records per query, the overall performance in queries-per-second is directly impacted by these performance numbers. In a dynamic query environment, for example, we might be aiming for video-rate screen refresh, and thus require the completion of 30 queries per second for each user. See Section 5 for another example.

From now on, when we show an implementation, we will omit the `for` loop, just showing the code inside the loop.

4. OPTIMIZING INNER LOOP BRANCHES

Using standard database terminology, we will refer to a particular implementation of a query as a *plan*. We now formulate our optimization question:

> Given a number `k`, functions `f1` through `fk`, and a selectivity estimate p_m ($m = 1, \ldots, \mathtt{k}$) for each `fm`, find the plan that minimizes the expected computation time.

So far we have seen three ways to write the inner loop. Each such plan has different performance characteristics. There are, in fact, many additional plans that can be formed by combining the three approaches. An example that combines all three is the following:

```
/* A Mixed Algorithm (loop code omitted) */
  if((f1(r1[i]) & f2(r2[i])) && f3(r3[i]))
    { answer[j] = i;
      j += (f4(r4[i]) & ... & fk(rk[i]));
    }
```

Significantly, several of these combination plans turn out to be superior to the three basic methods shown in Figure 1 over some selectivity ranges.

We will focus on finding a plan, consisting of some combination of the three methods presented above, giving the best expected time. We remark that there are other methods besides the three we have chosen for evaluating the inner loop. For example, one could add the function values rather than ANDing them, and compare with `k` at the end. (This alternative method might be useful in a hypothetical architecture in which an addition operation was faster than a logical AND.) Nevertheless, we expect that on realistic architectures, the three basic methods are among the most efficient.

4.1 A Normal Form for Combined Plans

For now, let us just consider plans involving a combination of the "branching-and" and the "logical-and" algorithms. We formulate how these two algorithms can be mixed, and consider when certain combinations are never optimal. Based on this notion, we derive a normal form for potentially optimal plans, and enumerate them.

A first glance at the two algorithms might suggest that all we need to do is consider all expressions within the `if` condition that can be formed out of the two kinds of "and" operation. However, this is clearly too many because `&` is commutative,[2] and both `&` and `&&` are associative. Additionally, if we are only interested in finding at least one optimal plan, we need only consider expressions in which all "outer" conjunctions are via `&&` and the conjuncts are terms involving only `&`.

To justify this assertion, consider the expression `E` given by `E0 && (E1 & (E2 && E3))` for arbitrary expressions `E0`, `E1`, `E2` and `E3`. (We allow `E0` to be empty, in which case there is no outer `&&`.) Consider the alternative expression `E'` given by `E0 && E2 && (E1 & E3)`. We claim `E'` is always more efficient than `E` on a non-parallel machine. In both cases the expression `E0` is evaluated. If `E0` is false, the performance is equivalent. If `E0` is true, then `E2` is evaluated in both `E` and `E'`. If `E2` is true, then both plans are again equivalent in terms of performance, since both `E1` and `E3` will be evaluated, and the same number of operations will be performed. However, if `E2` is false, then `E'` is superior to `E` because (a) it does not evaluate `E1` and (b) it avoids one `&` operation. By repeatedly applying the transformation from `E` to `E'` whenever we have a subexpression matching `E`, we essentially "pull up" all instances of `&&` to the top level. Each such transformation does not harm the performance, and in many cases improves it.

[2] We mean commutative in terms of performance rather than in terms of logic. Both arguments of `&` are evaluated and ANDed together; the order of evaluation does not affect the overall performance. Similarly, when we talk about associativity, we mean in terms of performance.

The order of the inner conjunctions (via `&`) does not matter, due to commutativity, and the parenthesization of the outer conjunctions (via `&&`) does not matter, due to associativity. We thus consider the inner conjuncts as sets of basic expressions, and the outer conjunction as being parenthesized from left to right. As outlined above, there must be an optimal plan in this normal form.

DEFINITION 4.1. *A single-function condition is called a* basic term. *A conjunction via* `&` *of basic terms is called an* `&`-term. *A conjunction via* `&&` *of* `&`*-terms is called an* expression. □

Let $t_{m,n}$ denote the number of normal-form plans over n basic terms, with exactly m occurrences of `&&`. Then $t_{0,n} = 1$ for all n. For the inductive case, consider prepending (via `&&`) an additional `&`-term to an expression with m occurrences of `&&`. Then

$$t_{m+1,n} = \sum_{i=1}^{n-1} \binom{n}{i} t_{m,n-i}.$$

We are actually interested in a_n, the number of plans, given by $a_n = \Sigma_{k=0}^{n} t_{k,n}$. Then $a_0 = 1$ and for $n > 1$ one can rearrange the above recurrence to get:

$$a_n = \sum_{j=1}^{n} \binom{n}{j} a_{n-j}.$$

This recurrence has been well-studied, as early as 1859 [4]; see [21] for further references. One representation of the solution [24] is that a_n is the closest integer to $n!/(2\ln^{n+1}(2))$.

Algorithm No-Branch can be thought of as a potential optimization to remove the final `if` test of a combined method. There is thus just one way to apply the optimization: to replace

```
if(E1 && ... && Ek)
  {answer[j++] = i;}
```

with

```
if(E1 && ... && Ek-1)
  { answer[j] = i; j += Ek;}
```

where the `Ei` terms are `&`-terms. Thus we should consider plans both with and without this optimization; the total number of potentially optimal plans is now $2a_n$.

4.2 Cost Functions

To compare the cost of the various plans, we need a cost model. The basic parameters of the model are: r, the cost of accessing an array element `rj[i]` in order to perform operations on it; t, the cost of performing an `if` test; l, the cost of performing a logical "and"; m, the cost of a branch misprediction; p_i, the selectivity of basic term i equal to the probability that basic term number i is 1; a, the cost of writing an answer to the answer array and incrementing the

answer array counter; f_i, the cost of applying function `fi` to its argument.

In our model, we will assume that the processor is perfect in its branch prediction, i.e., that it predicts the branch to the next iteration will be taken when the selectivity $p \leq 0.5$, and will not be taken when $p > 0.5$.

Given a plan, we add up the expected cost given the selectivities and the structure of the algorithm. We count just the cost of the code inside the loop, and not the loop iteration cost itself (since that's the same across all methods). We emphasize that in practice, one must model the costs for the assembly-language instructions generated by the compiler, rather than directly modeling the cost of the `C` code (see Appendix A).

EXAMPLE 4.1. *Consider Algorithm No-Branch on k basic terms. The total cost for each iteration is* $kr + (k-1)l + f_1 + \cdots + f_k + a$. □

EXAMPLE 4.2. *Consider Algorithm Logical-And on k basic terms, with selectivities* $p_1, \ldots, p_k$. *The total cost for each iteration is* $kr+(k-1)l+f_1+\cdots+f_k+t+mq+p_1 \cdots p_k a$, *where* $q = p_1 \cdots p_k$ *if* $p_1 \cdots p_k \leq 0.5$ *and* $q = 1 - p_1 \cdots p_k$ *otherwise. The q term describes the branch prediction behavior: we assume the system predicts the branch to the next iteration will be taken exactly when* $p_1 \cdots p_k \leq 0.5$. □

EXAMPLE 4.3. *Consider Algorithm Branching-And on k basic terms, with selectivities* $p_1, \ldots, p_k$ *(in the order listed in the* `if` *condition). The cost formula is the solution for* c_1 *of the recurrence*

$$c_n = r + t + f_n + mq_n + p_n c_{n+1} \qquad (1 \leq n \leq k)$$

where $q_n = p_n$ *if* $p_n \leq 0.5$ *and* $q_n = 1 - p_n$ *otherwise, and* $c_{k+1} = a$. *Again, the* q_n *terms describe the branch prediction behavior; in this algorithm we can execute as many as k conditional branches.* □

While this model captures the important aspects of the problem that are common across most modern architectures, it is not an exact cost calculation. Several architecture-dependent features make it approximate, including: out-of-order execution of instructions, overlapping memory access and computation, imperfect branch prediction based on just the most recent branches, and the degree of instruction-level parallelism present.

DEFINITION 4.2. *Let* `E` *be an* `&`*-term. The* fixed cost *of* `E`, *written* $\mathit{fcost}(\texttt{E})$, *to be the part of the cost of* `E` *that does not vary with the selectivity of* `E`. *In particular, if* `E` *contains k basic terms using* `f1` *through* `fk`, *then* $\mathit{fcost}(\texttt{E}) = kr + (k-1)l + f_1 + \cdots + f_k + t$. □

We can combine the observations of Examples 4.2 and 4.3 to derive a general recurrence for mixed plans: Consider the plan P_1 given by

```
if (E && E1) {answer[j++] = i;}
```

where `E` is an `&`-term and `E1` is a nonempty expression. Then the cost of this plan is

$$\mathit{fcost}(\texttt{E}) + mq + pC \qquad (1)$$

where p is the overall combined selectivity of `E`, $q = \min(p, 1-p)$, and C is the cost of the plan P_2:

```
if (E1) {answer[j++] = i;}
```

In particular, for P_1 to be an optimal plan, P_2 must also be an optimal plan (for fewer terms). We use this observation as the basis for developing a dynamic programming solution to our problem in Section 4.4. First, though, we investigate ways to limit the plans we consider by eliminating term orders that cannot be optimal.

4.3 Term Order in Optimal Plans

Hellerstein et al. consider *expensive predicates*, i.e., where the computation needed for evaluating whether the predicate is true or false dominates the overall cost [12]. In that context, it is shown that predicates should be ranked in ascending order according to the metric $\frac{\mathit{selectivity}-1}{\mathit{cost\text{-}per\text{-}tuple}}$. Our context differs in that our predicates are often *cheap*, meaning that other costs such as the branch misprediction penalty cannot be ignored. Further, there could be a *higher* misprediction penalty for a *lower* selectivity, meaning that this ranking would not be correct when the penalty is sufficiently high. Nevertheless, our derivation of term orders below bears some similarity to this rank ordering approach.

LEMMA 4.1. *Consider plans of the form*

```
if (E1 && E2 && E) {answer[j++] = i;}
```

where `E1` *and* `E2` *are nonempty* `&`*-terms, and* `E` *is an arbitrary (possibly empty) expression. Let* p_1 *and* p_2 *be the selectivities for* `E1` *and* `E2` *respectively. Such plans cannot be optimal if* $p_2 \leq p_1$ *and* $\frac{p_2-1}{\mathit{fcost}(\texttt{E2})} < \frac{p_1-1}{\mathit{fcost}(\texttt{E1})}$. □

A corollary of this lemma is that whenever two consecutive `&`-terms appear anywhere as conjuncts of `&&` (i.e., not just leftmost) in an optimal plan, then the one with lower selectivity must appear first if it has the same *fcost*.

Note that Lemma 4.1 says nothing about the case where there is an intervening expression between the two `&`-terms. An analogous statement to Lemma 4.1 when there are intervening expressions between `E1` and `E2` fails for two reasons. First, when $p_1 > 1/2$ it is always possible to find a sufficiently large branch misprediction penalty and a value for p_2 less than p_1 such that switching the two basic terms leads to an *inferior* plan. Second, even when $p_1 \leq 1/2$, the condition $\frac{p_2-1}{\mathit{fcost}(\texttt{E2})} < \frac{p_1-1}{\mathit{fcost}(\texttt{E1})}$ is not strong enough to guarantee that switching `E1` and `E2` is a win. Nevertheless, when there are intervening terms we can state the following weaker lemma.

LEMMA 4.2. *Consider plans of the form*

```
if (E1 && X1 && E2 && X2)
  {answer[j++] = i;}
```

where `X1` *and* `X2` *are arbitrary (possibly empty) expressions,* `E1` *and* `E2` *are nonempty* `&`*-terms with respective selectivities* p_1 *and* p_2*, and* $p_1 \leq 1/2$*. Such plans cannot be optimal if* $p_2 < p_1$ *and* $fcost(\texttt{E2}) < fcost(\texttt{E1})$*.* □

A corollary of Lemma 4.2 is that when all selectivities are at most 1/2, a relatively common case, we can order `&`-terms `E` with selectivity p by the pair $(fcost(\texttt{E}), p)$. In our case $(x, y) < (x', y')$ if $x < x'$ and $y < y'$. This ordering on `&`-terms is partial, since it is possible to have incomparable pairs. The partial order constrains the order of `&`-terms in optimal plans.

DEFINITION 4.3. *We call the pair* $(\frac{p-1}{fcost(\texttt{E})}, p)$ *the c-metric of* `&`*-term* `E` *having combined selectivity* p*. We call the pair* $(fcost(\texttt{E}), p)$ *the d-metric of* `&`*-term* `E` *having combined selectivity* p*.* □

Note that if `E1` is less than `E2` according to the d-metric, then `E1` is also less than `E2` according to the c-metric, but not vice versa. We use Lemmas 4.1 and 4.2 in the dynamic programming algorithm below.

4.4 Finding Optimal Plans

When the number of basic terms is small, we could simply enumerate all normal form plans and calculate the cost, choosing the plan with the smallest cost. However, the number of plans grows factorially in the number of basic terms (Section 4.1), and so alternative methods are necessary in general.

We propose a dynamic programming solution to the problem that is outlined below.

ALGORITHM 4.1. **Optimal-Plan** *Let* S *denote the set of basic terms, and let* k *be the cardinality of* S*. Create an array* $A[]$ *of size* 2^k *indexed by the subsets of* S*. The array elements are records containing: The number* n *of basic terms in the corresponding subset; the product* p *of the selectivities of all terms in the subset; a bit* b *determining whether the no-branch optimization was used to get the best cost, initialized to 0; the current best cost* c *for the subset; the left child* L *and right child* R *of the subplans giving the best cost.* L *and* R *range over indexes for* $A[]$*, and are initialized to* $\emptyset$*.*

In the loops over subsets of S*, we iterate in an order consistent with the partial order of subsets of* S*. In other words, if* $s_1 \subset s_2$*, then* s_1 *comes before* s_2 *in the loop. We call such an order an "increasing" order below. Note that a standard encoding of subsets as bitmaps yields an increasing order if we simply increment the bitmap on each iteration.*

1. */* Consider all plans with no* `&&`*s */*
   Generate all $2^k - 1$ *plans using only* `&`*-terms, one plan for each nonempty subset* s *of* S*. Store the computed cost (Example 4.2) in* $A[s].c$*. If the cost for the No-Branch algorithm is smaller, replace* $A[s].c$ *by that cost (Example 4.1) and set* $A[s].b = 1$*.*
2. *For each nonempty* $s \subset S$ *(in increasing order)*
   */** s *is the right child of an* `&&` *in a plan */*
   For each nonempty $s' \subset S$ *(in increasing order) such that* $s \cap s' = \emptyset$ */** s' *is the left child */*
   if (the c-metric of s' *is dominated by the c-metric of the leftmost* `&`*-term in* s*) then*
   {/ do nothing; suboptimal by Lemma 4.1 */}*
   else if ($A[s'].p \leq 1/2$ *and the d-metric of* s' *is dominated by the d-metric of some other* `&`*-term in* s*) then*
   {/ do nothing; suboptimal by Lemma 4.2 */}*
   else {
   Calculate the cost c *for the combined plan (*s' `&&` s*) using Equation 1. If* $c < A[s' \cup s].c$ *then:*
   (a) *Replace* $A[s' \cup s].c$ *with* c*.*
   (b) *Replace* $A[s' \cup s].L$ *with* s'*.*
   (c) *Replace* $A[s' \cup s].R$ *with* s*. }*

At the end of the algorithm, $A[S].c$ *contains the optimal cost, and its corresponding plan can be recursively derived by combining the* `&`*-conjunction* $A[S].L$ *to the plan for* $A[S].R$ *via* `&&`*.* □

Because the loops over the subsets of S are performed in increasing order, any newly-generated partial plan will be considered as part of larger plans later on, within the same loop. One never has to revisit plans that have already been considered.

The utility of the metric tests is that we avoid generating a large number of intermediate-quality plans that improve on the currently computed best cost, without being optimal. In practice, we need to verify that the reduction of the search space afforded by these tests outweighs the costs of the tests themselves.

The complexity of this algorithm is $O(4^k)$ which, while exponential, is asymptotically much better than generating and testing all normal-form plans (Section 4.1). Note that the algorithm simultaneously solves the optimization problem for all subsets of S too, so that one run of the algorithm can cover many potential loop structures.

Since we are typically interested in small values of k, the exponential complexity is not a barrier to its use in practice. We implemented the optimization algorithm in `C++` and ran it on both the Pentium III and the UltraSparc. The optimization time itself was always less than 0.01 seconds when $k \leq 9$, for various probability values. We investigate how well the output of the optimization algorithm matched actual performance time in Section 5.1.

4.5 A Heuristic Optimization Algorithm

While the optimization algorithm of the previous section is guaranteed to find the optimal solution, it still has exponential complexity. Thus, if we were to be presented with an optimization problem having a sufficiently large number of conditions, it would not be practical. Additionally, when the number of records to be processed is only moderate, we would want to spend just a small amount of time on optimization; the method of the previous section may be too expensive compared with the expected gains in evaluation time.

To address this problem, we present a heuristic method that takes linear space and has complexity $O(k \log k)$ in the average case, and $O(k^2)$ in the worst case. While the heuristic method is not guaranteed to find the optimal solution, we will demonstrate experimentally that it finds good solutions.

We begin by ordering the terms of the conjunction in ascending order according to the metric $\frac{\textit{selectivity}-1}{\textit{cost-per-tuple}}$. Our intuition is that, as for the expensive predicate case, ordering predicates in this way will be generally effective. However, this is just the start of the process: we still need to decide how to evaluate the conjunction using the three kinds of plans described above.

We treat the conjunction of k conditions as if it were to be evaluated using a Logical-And plan. We then move from left to right within the plan, evaluating the cost of the plan formed by replacing an `&` by an `&&`. We keep moving from left to right as long as the measured cost decreases. As soon as the measured cost increases, or we reach the end of the list, we terminate the left-to-right traversal. If we didn't reach the end of the list, we then spawn two recursive suboptimization processes, one for the left half of the expression, and one for the right. As a final tweak (not within the recursion), we replace the rightmost Logical-And subplan by a No-Branch subplan if the latter has lower cost.

For example, consider the basic terms ordered according to the metric above as $E_1, E_2, \ldots, E_k$. We evaluate the cost of $E_1 \&\& (E_2 \& \ldots \& E_k)$, then $(E_1 \& E_2) \&\& (E_3 \& \ldots \& E_k)$, and so on, until the plan $(E_1 \& \ldots \& E_i) \&\& (E_{i+1} \& \ldots \& E_k)$ is less costly than $(E_1 \& \ldots \& E_{i+1}) \&\& (E_{i+2} \& \ldots \& E_k)$. We then recursively apply the heuristic to the subexpressions $(E_1 \& \ldots \& E_i)$ and $(E_{i+1} \& \ldots \& E_k)$ to get plans P_1 and P_2 respectively. The final returned plan is $P_1 \&\& P_2$, with a possible modification of P_2 to use a No-Branch plan for its rightmost term.

The analysis of this algorithm is very similar to the analysis of quicksort. It takes linear space, worst-case quadratic time, and $k \log k$ time on average assuming randomly distributed termination points in the left-to-right traversal.

The intuition behind the method is that once we have decomposed a plan P into one of the form $P_1 \&\& P_2$, then P_1 and P_2 can be optimized independently; they do not depend on each other. The placement of the top-level && within P is done heuristically, assuming that the plan for the right-hand-side is the Logical-And plan. At the cost of adding complexity, one could consider alternative plans for the right-hand-side in order to determine a better partitioning point.

We shall study the quality of plans generated by the heuristic optimization method experimentally in Section 5.1. In terms of optimization time, our implementation on both the Pentium III and the UltraSparc takes less than 0.01 seconds consistently for $k \leq 60$. For $k = 4$ the optimization time was consistently less than 16 microseconds.

5. CASE STUDY

To demonstrate that our solution constitutes a feasible solution to realistic classes of problems, we describe a case study in which we apply these techniques in the context of a prototype event-based notification system called "Le Subscribe" [16, 8].

Le Subscribe aims to store millions of subscriptions, and to match hundreds of events per second against these subscriptions. Each subscription specifies a conjunction of simple conditions to apply to events, such as numeric equalities and inequalities. Where possible, subscriptions are partitioned into clusters based on equality conditions in the subscriptions. When an event arrives, it needs to be matched against clusters that agree with the event on the value of the partitioning attribute(s), as well as against subscriptions having no equality conditions.

Subscriptions are grouped based on the number of conditions. So, subscriptions with two conditions are grouped together for example. A group with `k` conditions is stored as a collection of `k` one-dimensional arrays `r1[i]`, ..., `rk[i]`. The `i`th entry in each array is a condition from the `i`th subscription.

Conditions are simply pointers to memory locations containing boolean values. Whenever an event arrives, the global set of boolean values is updated to reflect the characteristics of the event. That way, repetitive checking of conditions by thousands of subscriptions is avoided. The overall performance of the matching system is measured by how many events per second can be matched for a given number of subscriptions.

Matching against a group of subscriptions takes place using a sequential scan of the corresponding arrays. For a discussion of how Le Subscribe employs prefetching, see [8]. Subscriptions do not change rapidly. Thus one can obtain good estimates of selectivity for each `ri` by either estimating the distribution of events, or by keeping track of historical selectivities.

It is important to realize that the selectivities in each cluster are unlikely to be extremely small, since most (if not all) of the equality conditions would have already been applied in the partitioning step. The remaining inequalities (such as `price<100`) may have selectivities distributed (not necessarily uniformly) across the whole $[0, 1]$ range.

The simplicity of the subscription language means that the functions `fj` are both cheap and small in number. Further, the functions that are actually executed in the inner loop are just pointer lookups: the code will look like `if (*r1[i]`

`&& *r2[i] ....` This implementation is very similar to our dimension-table preprocessing example (context 3) in Section 3.1, with *every* function being treated in the same way.

We can reap two immediate benefits in terms of function specialization here. The first benefit is that all of the functions can be inlined, yielding very efficient code. The second, more subtle benefit is that we can get away with fewer pieces of code to implement all of the various candidate plans, because of the symmetry of the functions. For example, we can use the same subroutine to execute both the test `if (*r1[i] && *r2[i]) ...` and the "opposite" test `if (*r2[i] && *r1[i]) ...` by simply switching the positions of `r1` and `r2` in the parameter list when calling the subroutine.

The maximum number of subroutines we thus need to precompute is equal to the number of distinct normal form expressions when we consider all basic terms to be equivalent. A simple induction shows that for $n \geq 1$ basic terms we have 2^{n-1} such expressions. If we allow the No-Branch optimization, the number of expressions doubles, and the total is 2^n.

We expect in practice that the bulk of the subscriptions will have at most 6 basic expressions per subscription [16, 8]. Since the code for the inner loop is quite small, it is feasible to precompile all $2^1+2^2+\ldots+2^6 = 126$ code alternatives into the system, without using any sophisticated run-time code generation. For the small number of subscriptions having more than our predefined limit, we can use a generic loop. The generic loop will be more expensive per subscription than the specialized ones, but with few subscriptions of that form, the net cost will be small.

Based on the estimated selectivities, the best method for each group within each cluster can be determined off-line using the algorithm of Section 4.4. A function pointer can be stored with the sub-list to indicate which of the various plans should be used for this sub-list. (A permutation indicating the order of the arguments is also required.)

5.1 Validation

We validate our approach for an implementation consistent with the event notification scenario above. All functions `fi` are simple lookups in a corresponding character array `ti` of size 1000. Values in this array are either 1 or 0, set randomly according to a probability parameter p_i. The selectivities of each condition can thus be separately controlled.

We chose values for the cost model parameters that were consistent with both published reports [6, 1] and with the typical assembly code generated by `gcc`. The numbers for a Pentium III, measured in machine cycles, are: $r = 1$, $t = 2$, $l = 1$, $m = 17$, $a = 2$, $f_1 = \ldots = f_k = 1$.

In our first experiment, we show how the optimizer and the heuristic perform for four conditions when all probabilities are the same. This is the same scenario described by Figure 1. We ran many scans against a single cluster in memory, so that there is no cache miss penalty. Figure 2 shows the results for a 750 MHz Pentium III machine. The cost prediction of the optimizer is given as the solid line in the graph; the dotted line is the heuristic prediction. The actual performance numbers of all plans selected by the optimizer on some range are plotted as points. The order of the legend indicates the left-to-right ordering of ranges in which that plan was selected by the optimizer. In particular, the nobranch variant of the branching-and plan was optimal for $p \leq 0.14$; the nobranch variant of the `(1&2) && (3&4)` plan was selected from $p = 0.15$ to $p = 0.45$; the nobranch version of the `(1&2&3) && 4` plan was chosen for $p = 0.46$ through $p = 0.52$; for $p \geq 0.53$, the nobranch plan was chosen.

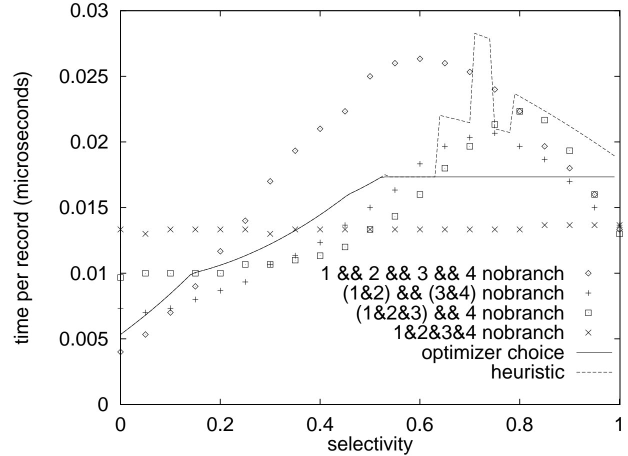

Figure 2: Prediction and actual performance.

For architecture-dependent reasons that we've already mentioned we don't expect our cost models to be exact cost estimates. Thus, we don't expect a perfect match of predicted cost with actual cost. The optimizer consistently overestimates the performance by about 20%. Nevertheless, the optimizer's choice is usually the best method for the given range.

To quantify how well our model measures branch misprediction, we compared the model's estimate of the number of mispredicted branches per record with the actual number of mispredictions. The actual number is obtained by using the hardware counters available on Pentium III processors to count the exact number of branch mispredictions; we used the "rabbit" tool to perform the actual counting [11]. The results for the branching-and plan, the plan having the most branches, are given in Figure 3. The closeness of the curves indicates that we are doing a good job of modeling branch misprediction.

The heuristic performs well except for high probabilities, when the no-branch algorithm is best. This observation suggests a simple modification to the heuristic algorithm: compare the result of the heuristic algorithm with the no-branch algorithm as a final step before choosing a plan.

In our second example, we consider a four-way conjunction in which the selectivities are unequal. The selectivity of the first condition is varied between 0 and 1, and is plotted on the x-axis. We let the second condition have a selectivity of

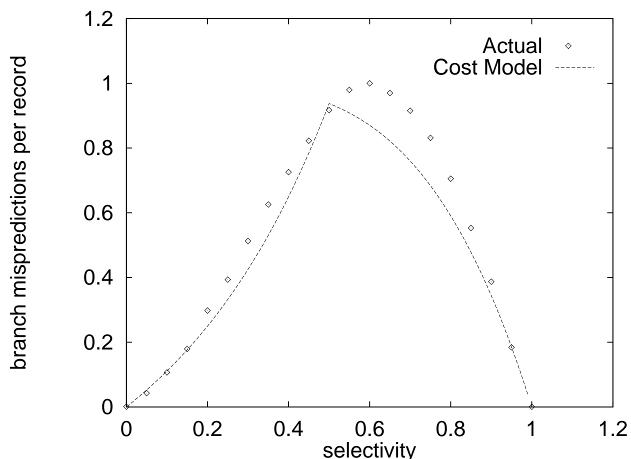

Figure 3: Branch misprediction count.

0.25, the third a selectivity of 0.5, and the fourth a selectivity of 0.75. Figure 4 shows the results. There are three plans

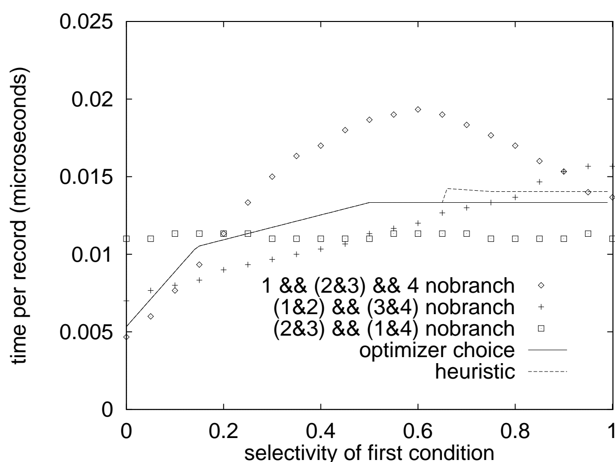

Figure 4: Unequal probabilities.

chosen by the optimizer in different ranges; the boundaries of those ranges are clear from the bumps in the optimizer selection curve. We see that when condition 1 is very selective, it appears on its own at the beginning of the test. When is it moderately selective, it is combined with the second condition. When it is not very selective, it appears at the right of the test. The heuristic performs adequately, although it gives plans about 10% worse than optimal for high probabilities.

5.2 Impact

We now try to measure the degree to which our techniques would affect the overall performance of subscription matching for Le Subscribe. Consider an example based on [8] in which there are six million subscriptions, and for which a number L of those subscriptions contain just inequality predicates. Because these subscriptions cannot be hash-partitioned, Le Subscribe would sequentially scan all L subscriptions for each event.

Using the parameter settings of [8], a default method would need between 12 and 45 nanoseconds per event per record. When L exceeds 150,000, i.e., 2.5% of the subscriptions, the cost of processing this subscription array (which is linear in L) dominates the overall cost. Our optimization techniques allow significant improvements (up to a factor of two) in this component of the cost. As a result, significant improvements in event throughput can be realized.

6. CONCLUSIONS

We have considered the problem of applying a conjunction of selection conditions to a large number of records in main memory. We have proposed a framework in which plans come from a space of plans representing combinations of three basic techniques. We have developed a cost model for plans that takes branch misprediction into account. We have developed a cost-based optimization technique using dynamic programming, for choosing among a space of plans, and have also developed a heuristic method of lower complexity. We have implemented an experimental case study based on a real-world event-notification system, and shown that significant performance gains can be achieved in that context.

The extent to which these kinds of performance gains can also be achieved in other kinds of query processing systems is highly dependent on the nature of their "inner loops." It is conceivable that many systems, including conventional database systems, have a relatively high overhead even for basic operations. For example, in order to handle arbitrary data types (possibly allowing null values) in a general way there may need to be some extra code in the inner loop. The benefits of our optimizations are significant only when the inner loops are tight, i.e., when the branch prediction overhead is a significant fraction of the total cost of the inner loop.

Acknowledgements

Thanks to Françoise Fabret, François Llirbat, João Pereira, Dennis Shasha and Eric Simon, whose Le Subscribe project motivated this work. Thanks also to the anonymous referees for several valuable suggestions.

7. REFERENCES

[1] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on a modern processor: Where does time go. In *VLDB 1999*, pages 266–277, 1999.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of VLDB Conference*, 2001.

[3] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th VLDB Conference*, pages 54–65, 1999.

[4] A. Cayley. On the theory of the analytical forms called trees ii. *Phil. Mag.*, 18:374–378, 1859.

[5] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Symposium on Principles of Programming Languages*, pages 145–156, 1996.

[6] I. Corp. *Intel Architecture Optimization: Reference Manual*, February 1999.

[7] I. Corp. *Intel IA-64 Architecture Software Developer's Manual, Volume 1 Rev. 1.0*, 2000. Available at `http://developer.intel.com/design/ia-64/manuals/`.

[8] F. Fabret, H.-A. Jacobsen, F. LLirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proceedings of the ACM SIGMOD Conference*, May 2001.

[9] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[10] J. Gray and P. J. Shenoy. Rules of thumb in data engineering. In *International Conference on Data Engineering*, pages 3–12, 2000.

[11] D. Heller. Rabbit: A performance counters library for intel/amd processors and linux., 2000. http://www.scl.ameslab.gov/Projects/Rabbit/.

[12] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the ACM SIGMOD Conference*, 1993.

[13] T. J. Lehman, E. J. Shekita, and L.-F. Cabrera. An evaluation of starburst's memory resident storage component. *IEEE Transactions on knowledge and data enginnering*, 4(6):555–566, 1992.

[14] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *Proceedings of the VLDB Conference*, pages 339–350, 2000.

[15] F. Noel, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, 1998.

[16] J. Pereira, F. Fabret, F. Llirbat, R. Preotiuc-Pietro, K. A. Ross, and D. Shasha. Publish/subscribe on the web at extreme speed. In *Proceedings of the VLDB Conference*, pages 627–630, 2000.

[17] P. Pucheral, J.-M. Thevenin, and P. Valduriez. Efficient main memory data management using the DBGraph storage model. In *International Conference on Very Large Databases*, pages 683–695, 1990.

[18] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th VLDB Conference*, pages 78–89, 1999.

[19] J. Rao and K. A. Ross. Making B^+-trees cache conscious in main memory. In *Proceedings ACM SIGMOD Conference*, pages 475–486, 2000.

[20] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th VLDB Conference*, pages 510–521, 1994.

[21] N. J. A. Sloane. The on-line encyclopedia of integer sequences, 2000. published electronically at `http://www.research.att.com/~njas/sequences`.

[22] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

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

[24] H. S. Wilf. *Generatingfunctionology*. Academic Press, NY, 1990.

[25] R. Yung. Design of the UltraSPARC instruction fetch unit. Technical Report SMLI TR-96-59, Sun Microsystems Laboratories, 1996.

APPENDIX

A. COMPILING IF STATEMENTS

In `C`, there is a distinction between the use of `&` and `&&` in conditional tests. This is best understood by considering the translation of a `C` code fragment into assembly code. We show two `C` code fragments, one for each of `&` and `&&`, and show the corresponding pseudo-assembly code next to it. Assume that the integer variables `a` and `b` are in registers `ra` and `rb` respectively.

```
if (a&b) {        load         rc,ra
  <innercode>     and          rc,rb
}                 compare      rc,0
<body>            branch-eq    bodylabel
                  <innercode>
                bodylabel:
                  <body>
```

```
if (a&&b) {       compare      ra,0
  <innercode>     branch-eq    bodylabel
}                 compare      rb,0
<body>            branch-eq    bodylabel
                  <innercode>
                bodylabel:
                  <body>
```

For `&&`, if the first argument is zero, we branch immediately to the body code, without checking the second argument. For `&`, we perform a logical `and` of the two arguments, and then check for zero. The `&` code has one conditional branch, while the `&&` code has two. The code for `&` could potentially be optimized. For example, if there is no further need for one of `a` or `b` after the test, we could use one of those registers and omit the load into `rc`. On many machines, the logical `and` instruction automatically sets the condition codes, meaning that a separate compare with zero is not needed.

B. NON-INDEPENDENT SELECTIVITIES

For selectivities that are not independent, the dynamic programming method of Section 4.4 still applies. When optimizing the subplan for a subset S of the attributes, one assumes that all branches in the complement of S have succeeded. Thus for an attribute $A_i \in S$, we use the conditional selectivity $p_i|\bar{S}$, i.e., the selectivity that the test on A_i succeeds given that the tests on all attributes in the complement of S have succeeded.

Note that for non-independent selectivities, sub-optimization steps no longer generate optimal sub-plans for fewer attributes, since the selectivities are conditioned on attributes not appearing in the subplan. Also, it may be difficult to represent all of the conditional selectivities: there are exponentially many of them corresponding to different combinations of attributes S.

C. SUN RESULTS

The results for the experiment of Section 3.2 on a Sun UltraSparc are given in Figure 5. Unlike the Pentium, as the

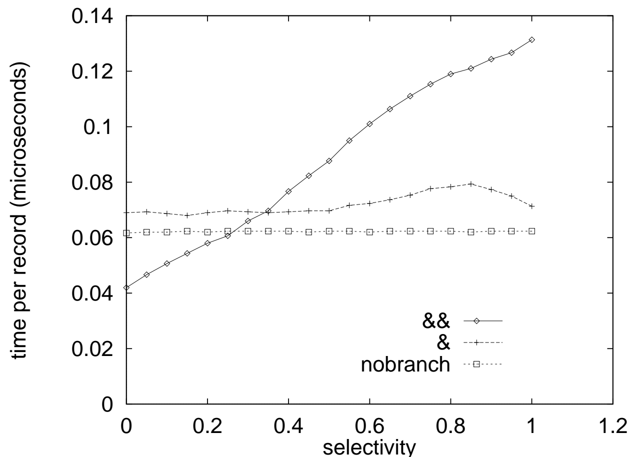

Figure 5: Three implementations: Sun.

selectivity approaches 1, the performance of the `&&` plan continues to worsen. The reason for this behavior is that the Sun can execute multiple instructions at a time. For the `&` algorithm and the nobranch algorithm, there are plenty of opportunities for executing multiple instructions in parallel. Instructions for the second test can be overlapped with instructions for the first, for example. However, in the `&&` algorithm there is much more dependence on the control flow, resulting in less effective parallelism. Taking such effects into account is a direction for future research. Note that even the first step of our approach (pulling up all instances of `&&` to the top level in Section 4.1) is not necessarily justified if subexpressions can be evaluated in parallel on a superscalar processor.

D. PREFETCHING AND FUNCTIONS

We need to address two important performance barriers: the cost of transferring data from RAM to the CPU cache, and the cost of evaluating functions. In this section we outline solutions to these barriers.

A potential performance problem is that we may have significant latency due to cache misses on the `r` arrays. After each cache-line's worth of entries from each `r` array is used, we have to wait until the next cache-line is brought into the cache from RAM. Given the tightness of the inner loop, this delay could be significant. This penalty can be reduced by employing *prefetching* [22, 6]. One instructs the processor to bring the `r` cache lines into the cache ahead of their actual use, using an explicit assembly language `prefetch` instruction. On a Pentium 4, the hardware *automatically* prefetches data ahead of its use for common access patterns, such as sequential access.

If we were to naively implement the code as written, we would need to execute a function call for each function evaluation. If the functions are known at compile time, they can be inlined, avoiding this overhead. Thus, if we know that certain "canned" queries are frequently posed, we can compile a single specialized loop for each one if we can derive estimates for the function cost and selectivity for the optimization algorithm. Since the loop code is small, we can probably tolerate thousands of such queries with a small expansion in the executable code size.

However, for ad-hoc queries we need to be able to allow the functions to be specified at run-time. There are two complementary problems. First, executing a function call (and potentially dereferencing a function pointer as well) may be a significant performance overhead in a tight inner loop. Secondly, we don't know the selectivities and function costs until query time, and these statistics are important for the selection of the appropriate inner-loop plan. There are several potential solutions to this problem. We outline one below.

When responding to an ad-hoc query, we still may have time to perform the optimization described above, compile a new version of the loop, with the appropriate combination of `&&`s and `&`s, and link it into the running code. Systems such as Tempo [5, 15] allow such run-time compilation. Run-time code specialization of this sort would be beneficial only if the optimization time plus the compilation time are smaller than the improvement in the running-time of the resulting plan. As we saw in Sections 4.4 and 4.5, the optimization time is relatively small. The code to be compiled is also relatively small. For scans of large tables, such an approach may indeed pay off.