Selection Conditions in Main Memory

KENNETH A. ROSS
Columbia University, New York, New York

We consider the fundamental operation of applying a compound filtering condition 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 for conjunctive conditions. We also develop an efficient heuristic optimization algorithm. We also show how records can be ordered to further reduce branch misprediction effects.

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.

Categories and Subject Descriptors: C.1 [**Processor Architectures**]; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.4 [**Database Management**]: Systems—*query processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Performance

Additional Key Words and Phrases: Branch misprediction

This research was supported by National Science Foundation 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.

A preliminary version of this article appeared as Ross, K. A. 2002. Conjunctive selection conditions in main memory. In *Proceedings of the ACM Symposium on Principles of Database Systems*. ACM, New York, 109–120.

Author's address: Department of Computer Science, Columbia University, 1214 Amsterdam Avenue MC 0401, New York, NY 10027; email: kar@cs.columbia.edu, Web: http://www.cs.columbia.edu/~kar.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2004 ACM 0362-5915/04/0300-0132 $5.00

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 article, we focus on one commonly-used database operation, namely applying a compound selection condition to a set of database records. One wishes to obtain those records satisfying the condition 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, stream processing systems, and network management systems. In each of these types of systems, one commonly poses queries involving the selection of records satisfying certain 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 article, we consider how to design efficient algorithms for applying a compound selection condition 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.

Our development first considers filters expressed as conjunctions of basic selection conditions. 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 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 extend the approach to conditions allowing negation and disjunction. We partition a compound filter into nested conjunctions and disjunctions. The duality of disjunction and conjunction allows us to use a dual algorithm for disjunctions.

We present a case study of the proposed methods in the context of an event-based notification system [Pereira et al. 2000; Fabret et al. 2001]. Our

experimental results show that significant performance improvements can be obtained. Our optimization algorithm and its cost model are validated against actual performance. Finally, we show how one can reorder records to further reduce branch misprediction effects.

Past work has identified that branch misprediction has a significant impact on modern database systems [Ailamaki et al. 1999]. To our knowledge, this article 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. Ailamaki et al. [1999] report 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 [Gray and Shenoy 2000]. In fact, this is precisely our approach.

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 [Intel Corp. 2000]; 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* [Pucheral et al. 1990; Whang and Krishnamurthy 1990]. Specialized algorithms for query processing in main-memory databases have been proposed in [Pucheral et al. 1990]. Shatdal et al. [1994] suggest several ways to improve the cache reference locality of query processing operations such as joins and aggregations. Boncz et al. [1999] propose 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 Rao and Ross [1999, 2000].

It has been observed that specialized memory-resident techniques allow substantial performance gains over buffer-resident data in a disk-based system [Garcia-Molina and Salem 1992; Lehman et al. 1992; Manegold et al. 2000]. More recently, Ailamaki et al. [2001] describe 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.

Single-Instruction Multiple-Datastream (SIMD) operations are available on modern processors. Zhou and Ross [2002] propose techniques for utilizing SIMD instructions for speeding up database operations. Like this article, some of the SIMD techniques benefit from a reduction in the number of branch mispredictions.

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 compound condition. 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 for conjunctions of those conditions. Nonindependent 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 Boncz et al. [1999].[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 compound selection condition 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 article 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,

[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.

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 Section 6.2.

We initially assume that the condition we wish to test is a conjunction of basic conditions. We consider disjunction and negation in Section 5.

3.1 Context

Our discussion assumes that the cost of processing the selections is a significant cost within the overall query, and 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, that is, 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 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 750-Mhz Pentium III under Linux, and a 300-Mhz UltraSparc IIi under Solaris. In the following experiment, we used $k = 4$ and let all of the rj arrays be offsets into an array t of chars 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 address caching issues in Section 6.1.) The code was compiled with gcc under maximum optimization, with several register hints present in the code.

Figure 1 shows the Pentium results. (See Section 6.3 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

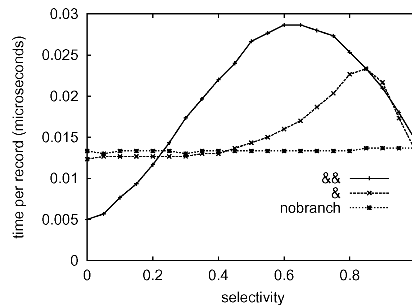


Fig. 1. Three implementations: Pentium.

penalty is higher on that architecture [Yung 1996]. 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 7 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 FOR CONJUNCTIONS

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 according to a simple cost model. 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 associative. We now show that if we are only interested in finding at least one optimal plan, we need only consider expressions in a particular "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 plan is said to be in *normal form* if it has the form `E1 && E2 && ··· && En` where the conjuncts `Ei` are &-terms.[3]

We now define the notion of a *cost model*. We allow the cost of an `&&` operation to depend on the selectivity of its first argument to model branch misprediction effects.

Definition 4.2. Each basic term is assigned a fixed cost. The *actual cost* of a plan P on a record is the sum of (a) the cost of each basic term in P that is

[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.

[3]The implicit parenthesization is `(E1) && [ (E2) && [ ··· [(En−1) && (En)] ··· ]]`.

actually evaluated, (b) the cost of each `&` operation that is actually evaluated, and (c) the cost of each `&&` operation that is actually evaluated. The cost of an `&` operation is independent of its arguments. The cost of an `&&` operation is independent of its second argument, but may depend on the selectivity of its first argument. The *estimated cost* is the expected actual cost, derived by considering records distributed in accordance with the independent selectivity estimates for each column. The estimated cost function defines a *cost model* for plans. A plan is *optimal* within a space of plans if it has the lowest estimated cost among all plans.

LEMMA 4.3. *Every expression has an optimal plan that is in normal form.*

PROOF. See Appendix C. □

Lemma 4.3 relies on the fact that the cost function is additive. Lemma 4.3 does not necessarily hold in nonadditive cost models, such as response time on a machine with multiple processors that can process subexpressions in parallel.

The order of the inner conjunctions (via `&`) does not matter, due to commutativity. We thus consider the inner conjuncts as sets of basic terms.

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 $n \geq 1$. For the inductive case, consider prepending (via `&&`) an additional `&`-term to a normal form expression with m occurrences of `&&`. Then

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

We are actually interested in a_n, the number of plans with n basic terms, given by $a_n = \Sigma_{k=0}^{n-1} t_{k,n}$. Then $a_1 = 1$ and for $n > 1$ one can rearrange the above recurrence to get:

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

which, with a_0 conventionally assigned the value 1, can be expressed for $n \geq 1$ as

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

This recurrence has been well studied, as early as 1859 [Cayley 1859]; see Sloane [2000] for further references. One representation of the solution [Wilf 1990] 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:E to replace

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

with

```
if (E1 && ··· && Ek-1) {answer[j] = i; += 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 A Detailed Cost Model

To compare the cost of the various plans, we need to choose a specific 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, that is, 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 4.4. 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 4.5. 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 4.6. 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 4.7. Let E be an &-term. The *fixed cost* of E, written *fcost*(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 $fcost(\mathtt{E}) = kr + (k-1)l + f_1 + \cdots + f_k + t$.

We can combine the observations of Examples 4.5 and 4.6 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

$$fcost(\mathtt{E}) + mq + pC, \tag{1}$$

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*, that is, where the computation needed for evaluating whether the predicate is true or false dominates the overall cost [Hellerstein and Stonebraker 1993]. In that context, it is shown that predicates should be ranked in ascending order according to the metric $\frac{selectivity-1}{cost\text{-}per\text{-}tuple}$. 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. Proofs of the results of this section can be found in Appendix C.

LEMMA 4.8. *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* $\frac{p_2-1}{fcost(\mathtt{E2})} < \frac{p_1-1}{fcost(\mathtt{E1})}$.

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 4.8 says nothing about the case where there is an intervening expression between the two &-terms. An analogous statement to Lemma 4.8 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 $\frac{p_2-1}{fcost(\texttt{E2})} < \frac{p_1-1}{fcost(\texttt{E1})}$ 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 4.9. *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 4.9 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 4.10. 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 4.8 and 4.9 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 4.11 **(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 4.5) in $A[s].c$. If the cost for the No-Branch algorithm is smaller, replace $A[s].c$ by that cost (Example 4.4) 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 4.8 */}
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 4.9 */}
else {
Calculate the cost c for the combined plan (s' && s) using Eq. (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 7.1.

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 the expression $E_1 \&\& (E_2 \& \cdots \& E_k)$, then $(E_1 \& E_2) \&\& (E_3 \& \cdots \& E_k)$, and so on, until $(E_1 \& \cdots \& E_i) \&\& (E_{i+1} \& \cdots \& E_k)$ is less costly than $(E_1 \& \cdots \& E_{i+1}) \&\& (E_{i+2} \& \cdots \& E_k)$. We then recursively apply the heuristic to the subexpressions $(E_1 \& \cdots \& E_i)$ and $(E_{i+1} \& \cdots \& E_k)$ 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 7.1. 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. NEGATION AND DISJUNCTION

We now generalize our approach to handle negation and disjunction. First, we can use De Morgan's laws to push negation down to basic terms, switching conjunctions to disjunctions and vice-versa as we go. Pushing negation does not change the total number of conjunction/disjunction operations. Second, we observe that negation applied to basic conditions poses no difficulties: the compiler simply reverses the sense of the corresponding `if` test. As a result, we can assume that we are dealing with a compound condition in which the only logical operators are conjunctions and disjunctions. Conjunctions and disjunctions may be nested to an arbitrary degree.

Example 5.1. Consider the condition

$$p \wedge (\neg(q \wedge \neg r) \vee \neg u \vee (s \wedge t)).$$

This condition can be rewritten as

$$p \wedge ((q' \vee r) \vee u' \vee (s \wedge t)).$$

The primed conditions are the complements of the originals, so that if u was $x > 3$ then u' would be $x \leq 3$.

In `C`, there is an operator `||` that implements disjunction, with the property that in evaluating `(p || q)`, `q` is only evaluated if `p` is false. If `p` is true, then the evaluation of `q` is skipped. Disjunction and conjunction are duals: `||` is the dual of `&&` and `|` is the dual of `&`.

This duality extends to deriving plans for evaluating disjunctive conditions. For example, implementing a "`(p||q||r)`" plan requires three conditional branches, implementing a "`(p|q|r)`" plan requires one conditional branch, and the corresponding no-branch plan uses no conditional branches. The difference between conjunction and disjunction is that for disjunction, work is saved if the condition *succeeds*, while for conjunction, work is saved if the condition *fails*. As a result, the formulation of the cost for a plan of the form

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

is given by

$$fcost(\mathtt{E}) + mq + (1 - p)C, \tag{2}$$

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;}.
```

The difference between Eq. (2) and the corresponding equation for conjunctions, Eq. (1), is that the coefficient of C is $(1-p)$ rather then p. The results of Section 4 all have analogs for disjunction (without conjunctions), where the selectivity p is replaced by $1 - p$ in the statement of the results.

If we have an expression of the form $\wedge\{p, \wedge\{q, r\}, \ldots\}$ we will rewrite it as $\wedge\{p, q, r, \ldots\}$. Similarly, $\vee\{p, \vee\{q, r\}, \ldots\}$ would be rewritten as $\vee\{p, q, r, \ldots\}$. This rewriting gives query optimization algorithms the most flexibility about the ordering of the arguments of each operator. Thus we can assume that in a

subexpression of the form $\wedge\{p, \ldots\}$, p is either a basic term or a disjunction of expressions (and conversely for $\vee$). We can write the compound expression of Example 5.1 as

$$\wedge\{p, \vee\{q', r, u', \wedge\{s, t\}\}\}.$$

At this point, we can extend the dynamic programming method to the general case with both conjunction and disjunction.

Algorithm 5.2. We optimize an expression inductively from the innermost subexpressions outwards.

For a subexpression E of the form $\wedge\{E_i\}$, where each E_i is a basic term, optimize E locally using Algorithm Optimal Plan. For a subexpression E of the form $\vee\{E_i\}$, where each E_i is a basic term, optimize E locally using Algorithm Optimal Plan, but using Eq. (2) in place of Eq. (1).

Now consider a subexpression E of the form $\wedge\{E_i\}$, where each E_i is an expression that has already been optimized. We use the computed cost and selectivity of E_i to optimize $\wedge\{E_i\}$ using Algorithm Optimal Plan, substituting the computed cost for $fcost(E_i)$ and the net selectivity for p in Eq. (1). Disjunctive subexpressions are handled in a dual fashion.

At the end of this process we have an overall plan for the entire expression, and a cost estimate for this plan.

Similarly, we can extend the heuristic method by first locally optimizing subexpressions. For example, to optimize $p \wedge (q \vee r)$, we first use the heuristic algorithm on the subexpression $q \vee r$ to get a plan for this subexpression. (For disjunctions, we order terms by $\frac{-selectivity}{cost\text{-}per\text{-}tuple}$.) The overall selectivity and cost of this plan is then used in deciding how to combine the $(q \vee r)$ subexpression with p in the overall plan.

Unlike the case for conjunctions alone, the plan chosen depends on the syntax of the expression. For example, the expression of Example 5.1 is equivalent to both

$$(p \wedge q') \vee (p \wedge r) \vee (p \wedge u') \vee (p \wedge s \wedge t)$$

in disjunctive normal form (DNF), and

$$p \wedge (q' \vee r \vee u' \vee s) \wedge (q' \vee r \vee u' \vee t)$$

in conjunctive normal form (CNF). If we use one of these expressions in place of the one in Example 5.1, we would come up with a different space of plans. As a result, Algorithm 5.2 is not guaranteed to find a globally optimal plan.

Neither the DNF expression nor the CNF expression are good choices here, because there is redundant evaluation of subexpressions. In any plan generated from the CNF expression, p is potentially evaluated four times. Similarly, in any plan generated from the DNF expression, $q' \vee r \vee u'$ is potentially evaluated twice. Thus, there seems to be a premium to be placed on the *compactness* of an expression.[4] In other words, as a general rule it is better to have as few operators as possible.

To get compact expressions, one could apply some form of logic minimization or common subexpression elimination, at the cost of some extra optimization

[4]For noncompact plans, in which an operand is repeated in different parts of an expression, we cannot assume that the subexpression success probabilities are independent of one another.

time. Nevertheless, in the context of ad-hoc query processing, we expect that human-generated conditions are likely to be relatively compact.

6. OTHER PERFORMANCE ISSUES

6.1 Prefetching

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* [Vanderwiel and Lilja 2000; Intel Corp. 1999]. 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.

When only part of the array is accessed, such as using a `matches` array as in the index-based loop of Section 3, prefetching is more difficult, for several reasons. Firstly, since there is a level of indirection, it takes more effort to compute the address of the next cache line to bring in. One needs several instructions to dereference the `matches` array for a future iteration, and to then execute the prefetch. Secondly, the density of the `matches` array may range from covering the whole array to covering a very small fraction. A fixed prefetch policy will not be good for both. If one tries to prefetch for every access to the `matches` array, one may be unnecessarily prefetching the same cache line many times. If one tries to prefetch just a portion of the `matches` array, then one may end up not prefetching as much as possible. A prefetch policy that is sensitive to the data in the `matches` array would probably use more time making decisions than the benefit obtained by prefetching. Finally, recall that the main benefit of prefetching is the opportunity to overlap computation and data transfer. If we are accessing each cache line to process one record in it (as opposed to say eight records for a scan) then there is much less computation per cache line, and the opportunity for overlapping computation and data transfer may be reduced.

Nevertheless, there are a few things we can do to improve data locality when accessing the array via an index. For example, we can make sure that the `matches` array is in increasing order. That way, accesses to the same cache line will happen consecutively, and cache lines won't be evicted from the cache before they are completely processed. Also, despite the difficulties in applying prefetching here, the index-based code may still perform best when the corresponding condition is selective because many fewer records need to be processed.

6.2 Code Specialization

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

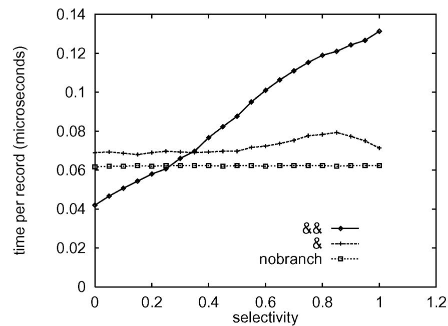


Fig. 2. Three implementations: Sun.

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. Second, 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 [Consel and Noel 1996; Noel et al. 1998] 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.

Run-time code specialization is different from *self-modifying code*. With self-modification, a program changes its own byte-code during its execution. While such a technique might actually present the most efficient solution to our code specialization problem, code modification is generally considered to be a bad idea. Such code is not reentrant, sharable, able to reside in ROM; it leads to cache coherency problems; it isn't easy to understand and it is architecture dependent.

6.3 Internal Parallelism

The results for the experiment of Section 3.2 on a Sun UltraSparc are given in Figure 2. Unlike the Pentium, as the 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. 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.

This aspect of chip performance is hard to model in an abstract way. The precise drop in potential parallelism depends in complex ways on the instruction set, the quality of the compiler, and the particular set of instructions being executed in the inner loop. Extending the cost-model to take such behavior into account is a topic for future research.

7. 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" [Pereira et al. 2000; Fabret et al. 2001].

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 Fabret et al. [2001]. 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]` $\cdots$. 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])` $\cdots$ and the "opposite" test `if (*r2[i] && *r1[i])` $\cdots$ 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 [Pereira et al. 2000; Fabret et al. 2001]. Since the code for the inner loop is quite small, it is feasible to precompile all $2^1 + 2^2 + \cdots + 2^6 = 126$ 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 sublist to indicate which of the various plans should be used for this sublist. (A permutation indicating the order of the arguments is also required.)

7.1 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 [Intel Corp. 1999; Ailamaki et al. 1999] and with the typical assembly code generated by `gcc`. The numbers for a Pentium III, measured in machine cycles, are: $r = 1$, $t = 2$, $l = 1$, $m = 17$, $a = 2$, $f_1 = \cdots = f_k = 1$.

In our first experiment, we show how the optimizer and the heuristic algorithm 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

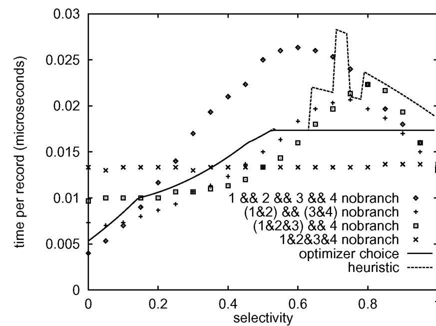


Fig. 3. Prediction and actual performance.

cluster in memory, so that there is no cache miss penalty. Figure 3 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.

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 [Heller 2000]. The results for the branching-and plan, the plan having the most branches, are given in Figure 4. 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 0.25, the third a selectivity of 0.5, and the fourth a selectivity of 0.75. Figure 5 shows the results. There are three plans 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

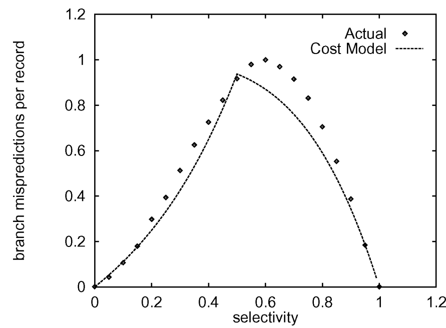


Fig. 4. Branch misprediction count.

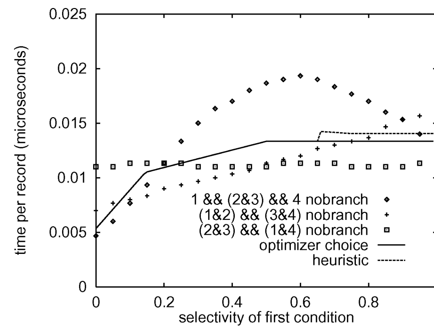


Fig. 5. Unequal probabilities.

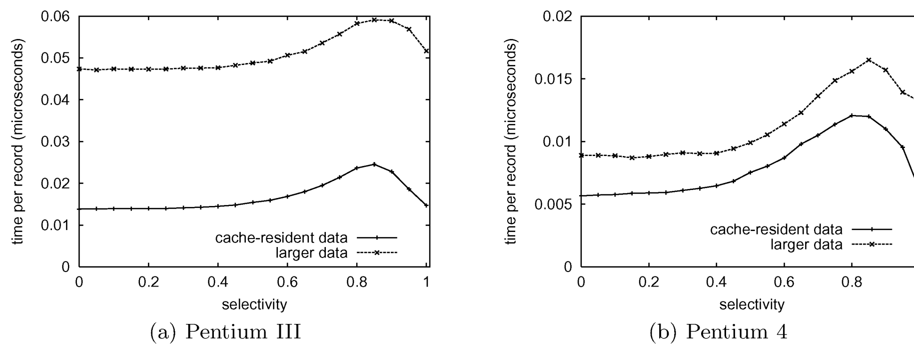
appears on its own at the beginning of the test. When is it moderately selective, it is combined with the second condition. When it is not very selective, it appears at the right of the test. The heuristic performs adequately, although it gives plans about 10% worse than optimal for high probabilities.

7.2 Cache Miss Penalties

Our experiments so far have iterated over a data set that fits in the cache of the machine. The cost of a cache miss is larger than the cost of a branch misprediction. Thus, we need to verify that branch misprediction effects are still significant even in the presence of cache misses. One can optimize the cache behavior by including explicit assembly language prefetch instructions as described in Section 6.1. The disadvantage of such an approach is that the code becomes architecture-dependent.

Fortunately, we can study the cache behavior more carefully (without assembly language coding) by contrasting the performance on a Pentium III with that of a Pentium 4. Unlike the Pentium III, the Pentium 4 automatically prefetches data for common reference patterns such as sequential access.

The graph of Figure 6(a) shows the performance for a 4-way logical-and plan on the 750 MHz Pentium III, and Figure 6(b) shows the same plan on a 1.8-GHz Pentium 4. The curves labeled “cache-resident data” are generated using a single small cluster over and over again, as in the previous experiments. That



(a) Pentium III (b) Pentium 4

Fig. 6. Effect of cache misses.

way, the whole cluster is cache-resident and there are no cache misses. The other curves are generated by cycling through a cluster that is much larger than the L2 cache. By the time the cluster is revisited, the required records have been expelled from the cache. These results show that prefetching can be effective in reducing the cache miss latency. They also show that branch misprediction effects are still significant even when cache miss penalties are considered.

7.3 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 Fabret et al. [2001] 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 Fabret et al. [2001], 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.

8. DATA PLACEMENT

So far, we have assumed that we have no control over the organization of the data. In applications such as stream processing, where the data is dynamic, such an assumption is natural. Even for applications in which the data is static and under the control of a database system, there may be design criteria more important than branch misprediction effects that dictate the structure of the data.

Nevertheless, in some cases we may have flexibility to determine the structure of the data. In a low-dimensional data set that is intended to support range queries, for example, it would be natural to store the data in a structure such as a k-d-tree or a quad-tree. The "curse of dimensionality" precludes such

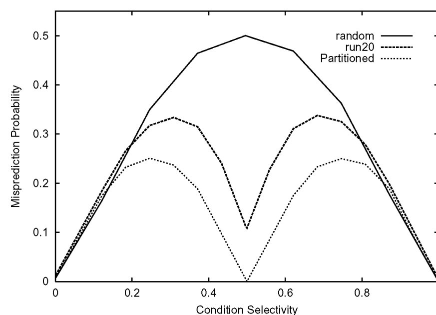


Fig. 7. Range predicate runs and branch misprediction.

data structures for high-dimensional data sets, unless one is willing to tolerate substantial redundancy.

In such a case, or when arbitrary queries need to be supported, a sequential scan over the data seems like the most robust choice. We still have some freedom to choose the *order* in which the data elements are stored. Can we choose an order that mitigates the branch misprediction penalties?

If our basic predicates are arbitrary and unrelated, there is little hope of benefiting from a particular order. An order designed for one predicate will not help for other predicates on the same attribute(s). In practice, though, predicates are often correlated. For concreteness, we shall assume that predicates on each attribute are range predicates. The important property of range predicates that we will exploit is that randomly chosen nearby values are likely to branch the same way for a given range predicate.

Imagine a column of 2^{20} Boolean values x that are randomly ordered. Suppose that the distribution of values is half zeroes and half ones. Then a scan of the column applying the predicate $x = 1$ (using any plan other than the no-branch plan) will cause roughly 2^{19} branch mispredictions. However, if the x values were sorted so that all zeroes preceded all ones, then there would only be a handful of branch mispredictions, occurring just at the start and at the transition point from zeroes to ones.

Sorting the data in this way is not desirable, because it excludes the possibility of optimizing other columns to reduce their branch mispredictions. If instead we were to group the x values into a run of zeroes followed by a run of ones, followed by another run of zeroes, and so on, then the number of branch mispredictions would be a small multiple of the number of *transitions* from zero to one or from one to zero. We could reduce the branch misprediction rate by a factor of roughly the average run length.

To map range predicates to Booleans, we first identify the range predicate p_i that has the worst-case branch misprediction behavior, that is, a success probability of about 1/2, for each column i. This predicate is a comparison with the median. If a row satisfies p_i, then we think of it as having a one bit for column i; otherwise it has a zero bit.

Figure 7 shows the impact of longer run lengths on range predicates on a Pentium 4 machine. We stored a large array of values chosen uniformly from

the range 0 to 255, and applied the selection condition $x \geq c$ for various values of c between 0 and 256. The selectivity (on the horizontal axis) is $1 - c/256$. We explicitly measured the number of branch mispredictions using the Intel VTune performance monitoring tool, and divided that number by the total number of loop iterations to get the branch misprediction probability.

The curve labeled "random" shows the impact of branch misprediction if the values are randomly ordered. The curve labeled "run20" shows the branch misprediction probabilities when the array is grouped into runs of length 20, with each group being either all less than 128, or all greater than or equal to 128. Within each run, the values are randomly distributed within the subrange. One can see that the branch misprediction probability has dropped from 0.5 to a little over 0.1 for selectivity 1/2. The worst-case branch misprediction probability is now about 1/3. For reference, the curve labeled "Partitioned" shows the branch misprediction penalties that would result if the first half of the array had values less than 128, and the second half had values greater than or equal to 128.

Now suppose that we have c columns x_1 through x_c of Booleans, and we wish to simultaneously reduce branch misprediction on all columns for the predicate $x_i = 1$. Assume a 50/50 distribution on each column, and suppose that the column distributions are independent. Assume 2^{20} records as before. Then it is possible to order the records using a *balanced Gray code* [Bhat and Savage 1996] on the c-bit key. A balanced Gray code has the property that each column has approximately $2^c/c$ transitions. For example, if $c = 20$, then any given column would have an average run length of approximately 20.

It would be possible to improve the branch prediction performance of Figure 7 even further by devoting two bits rather than one to the encoding of the given column, encoding which quartile the data value comes from. Combinatorial Gray codes [Savage 1997] are then appropriate. However, there is likely to be a fixed "budget" of bits, and we need to consider the workload to determine which columns deserve extra bits, potentially at the expense of other columns.

Once we have performed the reordering, we need to inform the optimizer that the column has an improved branch misprediction behavior. The best way to do this is to generalize Equation 1 so that the misprediction probability q is computed as a general function of the selectivity p. Rather than approximating q as the minimum of p and $1 - p$, one should derive a formula that matches the actual dependence on p, such as that given in Figure 7. While Lemmas 4.8 and 4.9 do not hold when the data placement is nonrandom, algorithm Optimal-Plan remains optimal if the metric tests are omitted from step 2.

9. CONCLUSIONS

We have considered the problem of applying a compound selection condition 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. We have described data ordering techniques that further improve the branch misprediction behavior.

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, that is, when the branch prediction overhead is a significant fraction of the cost of the inner loop.

The magnitude of the performance gains obtained here depend to some degree on the details of the hardware architecture considered. As architectures evolve, the proposed techniques should be re-evaluated. Significant architectural changes would set the stage for additional research.

APPENDICES

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. (Code fragments comparing `||` and `|` are similar.) 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. NONINDEPENDENT SELECTIVITIES

There are two common cases where selectivities of conditions are not independent. The first is when two attributes are correlated in some way, such as age and income for an employee database. The second is when two conditions test the same attribute. An example of the second would be a range constraint phrased as the pair $x > l$ and $x < h$. The probability of $x < h$ holding changes if we know that the record in question has met the condition that $x > l$. A more extreme example of this second case is when the same condition appears in multiple subexpressions of a condition.

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 $p_i|\bar{S}$, that is, 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 nonindependent selectivities, suboptimization steps no longer generate optimal subplans 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. PROOFS OF LEMMAS

PROOF OF LEMMA 4.3. Let P be an optimal plan for an expression. We will perform several optimality-preserving transformations on P that result in a normal-form plan.

Step 1. Suppose P has a subexpression of the form `E1 & (E2 && E3)`, where the order of the operands of the outer `&` is not important, and the `Ei` are (possibly compound) expressions. (If there is no such subexpression, we proceed to Step 3.) We claim that a plan P' in which this subexpression is replaced by `E2 && (E1 & E3)` is no more costly than P. If the subexpression as a whole is not evaluated, then P and P' are equally costly. Otherwise, in both P and P' the expression `E2` is evaluated, and both plans execute one `&&` operation for the subexpression. The `&&` operation is equally costly in P and P' since the first argument is the same in both plans. In both plans, `E3` is executed precisely when `E2` succeeds. However, in P, `E1` and the `&` operation are always executed, while in P' they are executed only when `E2` is true. Since the selectivity of the subexpression is not changed by this transformation, the cost of the enclosing operator is unchanged. Since P' does no more work than P, and since P is an optimal plan, P' must also be optimal.

Step 2. We recursively apply Step 1 to P'. A well-ordering argument, based on the total number of enclosing `&` operations of all `&&` operations, shows that the recursion must terminate, meaning that we do eventually reach Step 3.

Step 3. At this point, the transformed plan (call it P'') has no occurrences of `&&` within the scope of a `&` operator. However, P'' may violate normal form due to an inappropriate parenthesization of the `&&` operators. If there is an occurrence of the subexpression `(E1 && E2) && E3` in P'', we transform it into `E1 && (E2 && E3)`. The subexpression `E1 && (E2 && E3)` does no more work than `(E1 && E2) && E3`; the rightmost `&&` operator is not evaluated if `E1` is false. Thus, the optimality of the transformed plan is preserved. We recursively apply this transformation for every subexpression of the form `(E1 && E2) && E3`. A well-ordering argument, based on the total number of left-enclosing `&&` operations of all `&&` operations, shows that this recursion must terminate.

Once Step 3 is complete, the resulting plan is optimal, and in normal form. □

PROOF OF LEMMA 4.8. Suppose $p_2 \leq p_1$ and $\frac{p_2-1}{fcost(\mathtt{E2})} < \frac{p_1-1}{fcost(\mathtt{E1})}$. Let C be the cost of this plan, and C' the cost of the plan formed by switching the positions of `E1` and `E2`. Using Eq. (1),

$$C - C' = [fcost(\mathtt{E1})(1-p_2) - fcost(\mathtt{E2})(1-p_1)] + m(q_1(1-p_2) - q_2(1-p_1)).$$

The term inside square brackets is positive since $\frac{p_2-1}{fcost(\mathtt{E2})} < \frac{p_1-1}{fcost(\mathtt{E1})}$. For the remainder R of the expression we consider three cases. If $1/2 \geq p_1 \geq p_2$ then $q_1 = p_1$ and $q_2 = p_2$ and $R = m(p_1 - p_2) \geq 0$. If $p_1 \geq p_2 \geq 1/2$, then $q_1 = 1 - p_1$ and $q_2 = 1 - p_2$ and $R = 0$. If $p_1 \geq 1/2 \geq p_2$, then $q_1 = 1 - p_1$ and $q_2 = p_2$ and $R = m(1-p_1)(1-2p_2) \geq 0$. In each case, C' is less than C, and so our original plan cannot be optimal. □

PROOF OF LEMMA 4.9. Let C be the cost of this plan, and C' the cost of the plan formed by switching the positions of `E1` and `E2`. Suppose $1/2 \geq p_1 > p_2$. Using Eq. (1),

$$C - C' = [fcost(\mathtt{E1})(1-p_2 p) - fcost(\mathtt{E2})(1-p_1 p)] + (m+e)(p_1 - p_2)$$

where e is the cost of expression `X1` and p is the selectivity of `X1`. $fcost(\mathtt{E2}) < fcost(\mathtt{E1})$, $p_2 < p_1$, and $0 \leq p \leq 1$ together imply that the term in square brackets is positive. If $p_1 > p_2$, then the remaining term is also positive, C' is less than C, and so our original plan cannot be optimal. □

ACKNOWLEDGMENTS

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.

REFERENCES

AILAMAKI, A., DEWITT, D., HILL, M., AND WOOD, D. 1999. DBMSs on a modern processor: Where does time go. In *Proceedings of the VLDB Conference*. 266–277.

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. 2001. Weaving relations for cache performance. In *Proceedings of the VLDB Conference*. 169–180.

BHAT, G. S. AND SAVAGE, C. D. 1996. Balanced Gray codes. *Elect. J. Combin. 3*, 1, R25.

BONCZ, P. A., MANEGOLD, S., AND KERSTEN, M. L. 1999. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th VLDB Conference*. 54–65.

CAYLEY, A. 1859. On the theory of the analytical forms called trees II. *Phil. Mag. 18*, 374–378.

CONSEL, C. AND NOEL, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the Symposium on Principles of Programming Languages*. 145–156.

FABRET, F., JACOBSEN, H.-A., LLIRBAT, F., PEREIRA, J., ROSS, K. A., AND SHASHA, D. 2001. Filtering algorithms and implementation for very fast publish/subscribe. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York. 115–126.

GARCIA-MOLINA, H. AND SALEM, K. 1992. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng. 4*, 6, 509–516.

GRAY, J. AND SHENOY, P. J. 2000. Rules of thumb in data engineering. In *Proceedings of the International Conference on Data Engineering*. 3–12.

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

HELLERSTEIN, J. M. AND STONEBRAKER, M. 1993. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York. 267–276.

INTEL CORP. 1999. *Intel Architecture Optimization: Reference Manual*.

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

LEHMAN, T. J., SHEKITA, E. J., AND CABRERA, L.-F. 1992. An evaluation of starburst's memory resident storage component. *IEEE Trans. Knowl. Data Eng. 4*, 6, 555–566.

MANEGOLD, S., BONCZ, P. A., AND KERSTEN, M. L. 2000. What happens during a join? Dissecting CPU and memory optimization effects. In *Proceedings of the VLDB Conference*. 339–350.

NOEL, F., HORNOF, L., CONSEL, C., AND LAWALL, J. L. 1998. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the International Conference on Computer Languages*. 132–142.

PEREIRA, J., FABRET, F., LLIRBAT, F., PREOTIUC-PIETRO, R., ROSS, K. A., AND SHASHA, D. 2000. Publish/subscribe on the web at extreme speed. In *Proceedings of the VLDB Conference*. 627–630.

PUCHERAL, P., THEVENIN, J.-M., AND VALDURIEZ, P. 1990. Efficient main memory data management using the DBGraph storage model. In *Proceedings of the International Conference on Very Large Databases*. 683–695.

RAO, J. AND ROSS, K. A. 1999. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th VLDB Conference*. 78–89.

RAO, J. AND ROSS, K. A. 2000. Making B^+-trees cache conscious in main memory. In *Proceedings ACM SIGMOD Conference*. ACM, New York, 475–486.

SAVAGE, C. D. 1997. A survey of combinatorial Gray codes. *SIAM Review 39*, 4, 605–629.

SHATDAL, A., KANT, C., AND NAUGHTON, J. F. 1994. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th VLDB Conference*. 510–521.

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

VANDERWIEL, S. P. AND LILJA, D. J. 2000. Data prefetch mechanisms. *ACM Comput. Surv. 32*, 2, 174–199.

WHANG, K.-Y. AND KRISHNAMURTHY, R. 1990. Query optimization in a memory-resident domain relational calculus database system. *ACM Transactions on Database Systems 15*, 1, 67–95.

WILF, H. S. 1990. *Generatingfunctionology*. Academic Press, New York.

YUNG, R. 1996. Design of the UltraSPARC instruction fetch unit. Tech. Rep. SMLI TR-96-59, Sun Microsystems Laboratories.

ZHOU, J. AND ROSS, K. A. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York. 145–156.

Received December 2002; revised July 2003; accepted September 2003