Semantic Prefetching of Correlated Query Sequences

Ivan T. Bowman Kenneth Salem
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
itbowman@acm.org, kmsalem@uwaterloo.ca

Technical Report CS-2006-43, School of Computer Science, University of Waterloo, November 2006

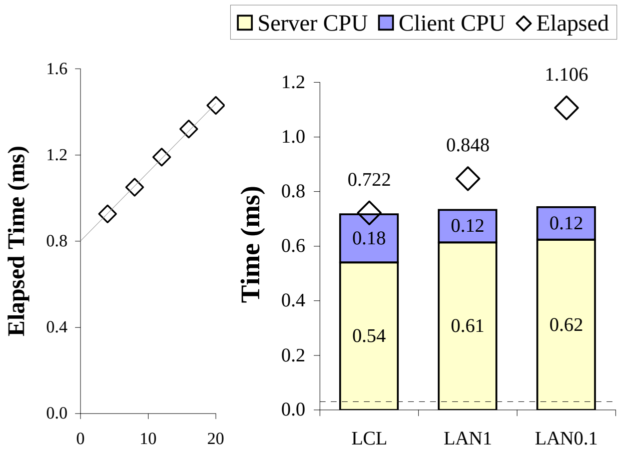
Abstract

We present a system that optimizes sequences of related client requests by combining small requests into larger ones, thus reducing per-request overhead. The system predicts upcoming requests and their parameter values based on past observations, and prefetches results that are expected to be needed. We describe how the system makes its predictions and how it uses them to optimize the request stream. We also characterize the benefits with several experiments.

1. Introduction

Each request made by a database client application to a database server incurs overhead associated with the interconnection network and the layers of interface software at both ends of the client/server connection. When client requests are expensive, the request overhead is dwarfed by I/O costs. As the size of main memory grows, however, an increasing portion of the working set of an application is maintained in the buffer pool. In such situations, which are quite common, the request handling cost is dominated by overhead.

To illustrate, consider the case of a single-row selection query that can be answered using an index. We measured the costs of such queries using three commercial DBMSs in 1 Gbps LAN configurations with drivers using C++/ODBC and Java/JDBC. In all of the cases we measured, overhead accounts for 95% or more of the total request cost when all of the needed pages are in memory at the server. Most of the overhead is server-side processing time consumed by the DBMS, which must interpret the OPEN request, initialize execution structures, and format the returned rows. A smaller amount of overhead is associated with the DBMS client, which must format the OPEN request and interpret the query results. Finally, the network introduces some latency.



(a) Cost of N probes (b) Overhead components

Figure 1. Costs for fetching from a cached table.

Figure 1(a) shows the cost of performing N index probes of a ten million row table by using a single query with an N-element IN predicate when all of the needed pages are cached in the buffer pool. The slope gives the cost of a single index probe, while the intercept represents fixed per-request overhead. Notice that the fixed overhead dominates the cost of these small queries. For example, we can perform 20 index probes without even doubling the cost of performing one probe.

Figure 1(b) shows the breakdown of execution overhead for three configurations (defined in Section 5). Most of the overhead is server-side processing time consumed by the DBMS, which must interpret the OPEN request, initialize execution structures, and format the returned rows. These costs are slightly higher in the Ethernet configurations (LAN0.1 and LAN1) than in the local shared-memory configuration (LCL) because of the more complex communication interface in the former. A smaller amount of overhead is associated with the DBMS client, which must format the Open request and interpret the query results. Finally, in the Ethernet configurations, the network introduces some latency: about 0.12ms in the gigabit (LAN1) configuration and 0.37ms in the 100Mbps (LAN0.1) configuration. For reference, the dashed horizontal line near the bottom of Figure 1(b) shows the calibrated cost of performing a single index probe (0.03ms), as determined by the slope of the line in Figure 1(a). This represents the cost of the useful work done by the query.

Although the single-row selection query is an extreme example, we have found that many real applications do submit small, cheap requests. For example, in our SQL-Ledger application case study (Section 6), we measured a median execution time of 1.1ms over all of the query types issued by the application. Less than 1% of all queries took longer than 10ms. For such applications, fixed overhead is by far the most significant factor in total query execution time.

Figure 2 shows pseudo-code for an application that issues a series of small queries to a DBMS. It is a simplified, artificially constructed application; however, its features are a composite of elements we observed in a set of database applications that we studied (described in more detail in Section 6). The GetCustomer procedure takes a partially-filled customer structure (`cust`) as input, and retrieves additional customer information from the database. It first issues query Q_a to retrieve the customer name and account number. If the application does not already have shipping information for the customer, `GetCustomer` calls the `GetDefaultShipTo` procedure to obtain the default shipping address. Finally, the application checks the customer's outstanding balance (Q_c) and uses that information to determine the available credit. The `GetVendor` procedure operates similarly on a vendor structure (`vend`); it retrieves the vendor name with Q_d, the mailing address with `GetDefaultShipTo`, and a list of parts to order with Q_e.

If the data needed by the application is mostly buffered, the execution time for queries $Q_\mathrm{a} \ldots Q_\mathrm{e}$ will be dominated by overhead. One way to optimize the application's performance is to combine the small queries into larger ones. For example, queries Q_a and Q_b could be combined into a single query like the one shown in Figure 3. Since overhead accounts for almost all of the execution time of small requests, replacing n small requests with one large request can reduce the total query execution time for those queries by almost a factor of n. Some of this improvement would be achieved by the elimination of communications latency. However, as shown in Figure 1, much of the savings would be achieved through the elimination of per-request computational overhead at the database server. This would improve not only response time, but also system throughput.

```
procedure GetCustomer(cust)
  fetch row r1 from $Q_\mathrm{a}$:
    SELECT name, accno FROM customer c
    WHERE c.id = :cust.id
  cust.name ← r1.name
  if not cust.shipto then
    cust.shipto ← GetDefaultShipTo(cust)
  fetch row r3 from $Q_\mathrm{c}$:
    SELECT SUM(amount-paid) as balance
    FROM ar a WHERE a.accno = :r1.accno
  cust.balance = r3.balance
end
function GetDefaultShipTo(info)
  fetch row r2 from $Q_\mathrm{b}$:
    SELECT addr FROM shipto s
    WHERE s.cid = :info.id AND s.default='Y'
  return r2.addr
end
procedure GetVendor(vend)
  fetch row r4 from $Q_\mathrm{d}$:
    SELECT name FROM vendor v
    WHERE v.id = :vend.id
  vend.name ← r4.name
  vend.mailto ← GetDefaultShipTo(vend)
  open c5 cursor for $Q_\mathrm{e}$:
    SELECT partname, invlevel-onhand AS qty
    FROM part p WHERE p.vid = :vend.id
    AND p.onhand < p.invlevel
  while r5 ← fetch c5 do AddOrder(vend,r5) end
  close c5
end
```

Figure 2. An Example Application

```
SELECT c.name, c.accno, s.addr
FROM   customer c LEFT JOIN shipto s
       ON s.id = c.id AND s.default = 'Y'
WHERE  c.id = :cust.id
```

Figure 3. Manually joining queries Q_a and Q_b.

One way to implement this kind of optimization is to manually tune the application code. Manual tuning of application code is not uncommon, and it can certainly be valuable. As a general approach to performance optimization, however, it has some weaknesses. First, performance-motivated optimizations may destroy other desirable application properties. For example, in Figure 2, the application logic that determines the default shipping address is encapsulated in a separate function, `GetDefaultShipTo`, and that function is called from both `GetCustomer` and `GetVendor`. Replacing Q_a and Q_b with the combined query in Figure 3 breaks this encapsulation. This, in turn, leads to code duplication in the application and potential increases in application development and maintenance costs. A second problem is that tuning may depend on information such as program parameter values, data distributions, and properties of the execution environment, such as communications latencies and system loads. These kinds of run-time information are unknown at application development time. Furthermore, they may vary across different installations of the application program. In the case of Figure 2, query Q_{b} follows Q_{a} only when the condition at line 6 is true. If this is rarely the case, combining Q_{a} and Q_{b} in the application will not help performance, and may actually hurt performance.

In previous work [3] we introduced a system called Scalpel that can perform this kind of optimization automatically, and transparently to both the client application and the underlying database system. Scalpel is located between the client application and the server. It monitors client/server communications and attempts to identify sequences of queries that can be replaced by a single combined query, as was illustrated in Figure 3. We called this technique *semantic prefetching*.

In our earlier work, we identified several common types of application query sequences for which semantic prefetching can help and we focused on one type: query nesting. In a nested query pattern, the application issues an inner query for each row fetched from the result of an outer query. In this paper, we focus instead on a second common application pattern, which we call *query batches*. A query batch is simply a sequence of non-nested related queries, such as the sequence $Q_{\mathrm{a}}, Q_{\mathrm{b}}, Q_{\mathrm{c}}$ from the application of Figure 2. Although semantic prefetching can be applied to both query batches and nested queries, handling query batches is quite different from handling nesting. Query nesting is relatively simple for Scalpel to detect. Thus, the focus of the earlier work was on how Scalpel should decide among the many possible ways of optimizing the detected nesting patterns. In contrast, the primary challenge with query batches is identifying suitable batches. Thus, the primary contribution of this paper is a technique for *automatically identifying optimizable query batches* in an application request stream. In addition, we present the results of experiments that quantify the performance benefits that can be expected from these optimizations as well as the costs introduced by the Scalpel system itself. Finally, we present an application case study which serves to demonstrate both the presence of optimizable query batches and Scalpel's ability to detect them.

The remainder of this paper is organized as follows. In Section 2, we give an overview of the Scalpel system architecture. This architecture was described in [3], but we have included a summary here so that the current paper is self-contained. Sections 3 and 4 describe how Scalpel identifies and optimizes query batches. The LATERAL-based query batch rewrites described in Section 3.2.5 are a subset of those described in our previous work on nested queries [3], but the detection and run-time mechanisms are specific to query batches. In Sections 5 and 6 we evaluate the performance of Scalpel's query batch optimizations using synthetic workloads and a case study of SQL Ledger, a database application. We used SQL Ledger as a case study in our previous work [3]. However, the performance evaluation presented in this paper focuses on query batches rather than nesting.

2. Scalpel System Overview

The components of the Scalpel system are illustrated in Figure 4. The system operates in two phases: training and run-time. During the initial training phase, Scalpel's Call Monitor passes all client requests through to the server without modifying them. In addition, the Call Monitor passes these requests to the Pattern Detector component, which monitors and records a representation of the request stream. At the conclusion of the training period, Scalpel's Pattern Optimizer analyzes this recorded information to identify optimizable batch patterns and produce corresponding rewrites, as described in Section 3. These are recorded in Scalpel's rewrite database for use during the subsequent run-time phase.

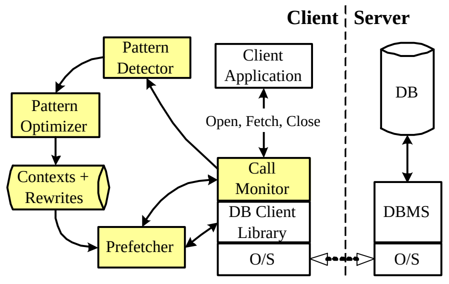


Figure 4. Components of the Scalpel system.

At run-time, Scalpel again monitors the application's request stream. This time, the Call Monitor passes each application request to the Prefetcher, which compares it against the request patterns recorded in the rewrite database. When the Prefetcher observes the start of a batch pattern for which it has a prefetch optimization, the Prefetcher issues the prefetch query to the database server. If the application behaves as expected, Scalpel uses the results of the prefetch query to answer the application's subsequent requests. If the application behaves unexpectedly, Scalpel ignores the results of the prefetch query and instead passes the application's actual requests through to the server.

Scalpel models an application's request stream as a sequence of queries. For each such query in the sequence, Scalpel will observe an `Open` request from the application, followed by zero or more `Fetch` requests, followed by a `Close` request. In general, application request streams may not always follow this sequential query pattern. For example, a stream may contain nested `Open` requests. Although Scalpel is capable of handling a more general class of request streams [3], for the purposes of this paper we will focus on request streams consisting of sequential queries.

Figure 5 shows a hypothetical application request trace as seen by Scalpel. The trace illustrates a query sequence that might be generated by an application that includes the code from Figure 2, as well as other code that we have not shown. Each row of the trace table in Figure 5 represents a single query (`Open`, `Fetch`, and `Close`). The `Query` column indicates the query that was opened, and the `Input` column shows the query parameter values with which it was opened. The query identifiers Q_a, Q_b, Q_c, Q_d and Q_e refer to the SQL queries shown in Figure 2, while queries Q_x, Q_y, and Q_z refer to other unspecified queries from elsewhere in the application. The `Output` column shows the query result tuple that was fetched by the application. If the application fetches more than one tuple from a cursor (such as Q_e), a set of tuples is shown.

#	Query	Input	Output
1	Q_x	(42)	(501)
2	Q_a	(101)	('Alice', 501)
3	Q_b	(101)	('1500 Robie St.')
4	Q_c	(501)	($400.00)
5	Q_d	(201)	('Mary')
6	Q_b	(201)	('1400 Barrington St.')
7	Q_e	(201)	{ ('Bell',3), ('Tire',6) }
8	Q_a	(121)	('Bob', 537)
9	Q_c	(537)	($0.00)
10	Q_x	(43)	(31337)
11	Q_a	(107)	('Cindy', 523)
12	Q_b	(107)	('1100 Sackville St.')
13	Q_c	(523)	($800.00)
14	Q_y	(189)	('Elbereth')
15	Q_d	(255)	('Ned')
16	Q_b	(255)	('1200 Weber St.')
17	Q_e	(255)	{ ('Pedal',7), ('Seat',3) }
18	Q_z	(42)	('Xyzzy')

Figure 5. An example trace.

3. Training Scalpel

Scalpel predicts upcoming queries based on the queries that it has observed so far. Like many other prefetchers,

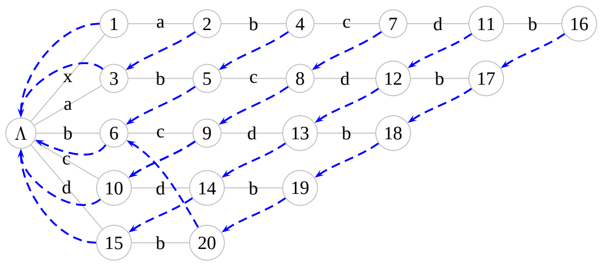


Figure 6. Suffix Trie After $Q_\mathrm{x}, Q_\mathrm{a}, Q_\mathrm{b}, Q_\mathrm{c}, Q_\mathrm{d}, Q_\mathrm{b}$ from Figure 5

Scalpel bases its predictions on queries that have been observed in the recent past. To define what we mean by "recent past", we define the notion of a k*-context*. The k-context at position p in a trace is the ordered list of queries at trace positions $p-(k-1), p-(k-2), \ldots, p-1, p$. For example, the 5-context at position 7 in the trace of Figure 5 is the list $[Q_\mathrm{b}, Q_\mathrm{c}, Q_\mathrm{d}, Q_\mathrm{b}, Q_\mathrm{e}]$.

Scalpel works by learning k-contexts from which it can predict that a particular query[1] will occur next. Scalpel learns by examining a training request trace like the one illustrated in Figure 5. For example, from the trace in Figure 5, Scalpel might learn that the 2-context $[Q_\mathrm{a}, Q_\mathrm{b}]$ predicts the query Q_c. These predictions are recorded in a rewrite database shown in Figure 4 and used to control prefetching at run time.

The first training challenge faced by Scalpel is how to choose the length k of the contexts on which it should base its predictions. Unfortunately, there is no single value of k that is always appropriate for prediction. If k is too small, Scalpel may miss valuable special cases. For example, in the trace of Figure 5, the 1-context $[Q_\mathrm{b}]$ is sometimes followed by Q_c and sometimes by Q_d. However, the 2-context $[Q_\mathrm{a}, Q_\mathrm{b}]$ is *always* followed by Q_c. Thus, in this case, $k=2$ leads to a better prediction than $k=1$. On the other hand, unnecessarily large values of k can lead to overly specific predictions. Longer contexts also require much longer training periods because each specific context will only be observed infrequently.

For these reasons, the Pattern Detector tracks k-contexts for *all* values of k during the training phase. Specifically, the Pattern Detector records every k-context ($0 \leq k \leq N$, where N is the trace length) that occurs in at least one position in the training trace. The Pattern Detector also records the number of times that each k-context occurs in the trace. These frequencies are used by the Pattern Optimizer (Section 3.2) to estimate whether prefetching will be cost-effective in a particular context.

[1]Scalpel is actually capable of predicting multiple queries from a particular k-context. For now, we will focus on the the prediction of a single query. Multiple query predictions are discussed in Section 3.2.3.

Our implementation of Scalpel uses a suffix trie to track k-contexts. The suffix trie representation offers a non-redundant encoding of overlapping contexts of different lengths. Figure 6 shows the suffix trie as it would look after the sixth query from Figure 5. Edges in the trie are labeled with the subscripts of queries from the trace, e.g., edge label `a` refers to query Q_a. Nodes represent the contexts that have been observed in the trace. Each node is labeled with a unique identifier, and represents the k-context consisting of the queries labeled on the path from the root to that node. For example, node 5 represents the 2-context $[Q_\mathrm{a}, Q_\mathrm{b}]$. The root node, labeled Λ, represents the 0-context. Dashed edges are suffix links. Suffix links are related to generalization of contexts, which we will describe in Section 3.2. For example, the suffix link from node 2 ($[Q_\mathrm{x}, Q_\mathrm{a}]$) points to node 3, which represents the more general context $[Q_\mathrm{a}]$.

Construction of a suffix trie as illustrated in Figure 6 would require $O(N^2)$ space and time, where N is the length of the training trace. However, Scalpel actually builds a *path compressed suffix trie* in $O(N)$ space and time using an algorithm due to Ukkonen [10].

3.1. Query Parameter Correlations

Queries are often parameterized. If Scalpel is to prefetch a query, it must predict not only that the query will occur, but also the parameter values with which the query will be invoked by the application. This is a significant distinction between prefetching queries and prefetching other non-parameterized objects, such as data blocks from an I/O system.

Often, the parameter values that are used for a given query are related, through data dependencies in the application code, to the input parameter values or the results of previously-issued queries. For example, consider the `info.id` parameter of query Q_b in the `GetDefaultShipTo` query in Figure 2. If `GetDefaultShipTo` is called from `GetVendor`, then `info.id` will have the same value as the `vend.id` parameter to `GetVendor`'s query Q_d. Similarly, the input parameter `r1.accno` to query Q_c is determined by the result of the preceding query Q_a.

Since Scalpel does not directly observe the application code, it cannot infer such dependencies through data flow analysis of that code. Instead, it tries to detect correlations among the query input parameter values and the query results that it observes in its training trace. Specifically, for each query in the training trace, Scalpel's pattern detector observes the correlations between that query's input parameters and the the input and output parameters of the preceding queries. To illustrate how this works, consider Scalpel's behavior when it observes query Q_c on line 4 of Figure 5. (The trace in Figure 5 shows the query input and output parameter values.) Scalpel will note that Q_c's input parameter value (501) matches the value of the second result attribute of the preceding query Q_a as well as the value of the first result attribute of the preceding query Q_x. Later, at line 13 of the trace, Scalpel will again observe Q_c. This time, it will verify that Q_c's input parameter value (now 523) matches the second result attribute of the preceding query Q_a. However, it will be unable to verify the correlation between Q_c and the result of the preceding Q_x, as their parameter values do not match this time around. Scalpel considers a parameter correlation to hold only if it never fails to hold in the training trace. Thus, Scalpel will dismiss the potential correlation between Q_c's input and Q_x's output.

We formalize this with the following definition. Suppose that $C = [Q_1, \ldots, Q_\mathrm{x}, \ldots, Q_\mathrm{k}]$ is a k-context. We say that the ith input parameter of Q_k is correlated to the jth input (output[2]) parameter of query Q_x in C iff the value of Q_k's ith input parameter matches the value of Q_x's jth input (output) parameter *every time* context C is observed in the training trace. So that it can track parameter correlations, the Pattern Detector records the input parameter values of each query that it observes in the training trace, as well as the most recent result tuple from each query.

As presented, Parameter correlation detection would require $O(N^3 m^2)$ time and space complexity, where m is the number of parameters per query and N is the length of the training trace. However, by adapting some of the same optimizations used by Ukkonen and by limiting the number of previous queries that Scalpel will check for parameter correlations, we can limit the overall time and space complexity to $O(NSm^2)$, where S is a system configuration parameter that limits the number of preceding queries that Scalpel will check. As discussed in Section 6, we have found that the training overhead is reasonable in practice.

3.2. The Pattern Optimizer

At the conclusion the training period, the Pattern Optimizer analyzes the suffix trie recorded by the Pattern Detector to determine, for each k-context (suffix trie node), whether semantic prefetching should take place. Suppose that $C = [Q_1, \ldots, Q_\mathrm{k}]$ is a k-context in the trie and that $C' = [Q_1, \ldots, Q_\mathrm{k}, Q_\mathrm{k+1}]$ is a *successor* context to C in the trie. This indicates that, on at least one occasion, the Pattern Detector observed that the query $Q_\mathrm{k+1}$ was executed immediately after the sequence of queries in C. The task of the Pattern Optimizer is to make a cost-based decision as to whether Scalpel should prefetch $Q_\mathrm{k+1}$ from context C.

If Scalpel is to prefetch $Q_\mathrm{k+1}$ from C, the prefetch should be both *feasible* and *beneficial*. We say that $Q_\mathrm{k+1}$

[2]The jth output parameter of Q_y refers to the jth attribute of Q_y's result. If Q_y returns several tuples, the value of each output parameter is determined by the most recently fetched tuple.

is a feasible prefetch from context C if the Pattern Detector observed at least one correlation for each input parameter of $Q_{\mathrm{k+1}}$ in context C'. Intuitively, this means that whenever $Q_{\mathrm{k+1}}$ followed C in the training trace, its input parameters were predictable. If query $Q_{\mathrm{k+1}}$ is not a feasible prefetch from context C, then Scalpel will not attempt to prefetch it from that context. If it is feasible, then Scalpel estimates whether prefetching $Q_{\mathrm{k+1}}$ would be beneficial.

3.2.1 Estimating the Benefit of Prefetching

If Scalpel chooses *not* to prefetch $Q_{\mathrm{k+1}}$, then the total cost of Q_{k} and $Q_{\mathrm{k+1}}$ (in context C) can be estimated as

$$\textsc{Cost}(Q_{\mathrm{k+1}}, C) = \textsc{Cost}(Q_{\mathrm{k}}) + P[Q_{\mathrm{k+1}}|C]\textsc{Cost}(Q_{\mathrm{k+1}})$$

where $\textsc{Cost}(Q_{\mathrm{k}})$ and $\textsc{Cost}(Q_{\mathrm{k+1}})$ are the estimated costs of executing queries Q_{k} and $Q_{\mathrm{k+1}}$, respectively, and $P[Q_{\mathrm{k+1}}|C]$ is the probability that the application will request query $Q_{\mathrm{k+1}}$, given that it is in context C. Scalpel estimates $\textsc{Cost}(Q_{\mathrm{k}})$ and $\textsc{Cost}(Q_{\mathrm{k+1}})$ by monitoring, at the client, the observed execution times of Q_{k} and $Q_{\mathrm{k+1}}$ during the training period. These observed times include the overhead and latency associated with communication between the client and the server, as well as the server-side cost of query execution. To estimate $P[Q_{\mathrm{k+1}}|C]$, Scalpel can use an estimator $\hat{p} = \frac{n(C')}{n(C)}$, where $n(C)$ and $n(C')$ are the observed frequencies of contexts C and C', as recorded by the Pattern Detector during the training period.[3] For example, if $n(C) = 10$ and $n(C') = 4$, Scalpel will estimate a 40% probability that $Q_{\mathrm{k+1}}$ will occur next in context C.

If, on the other hand, Scalpel chooses to prefetch, then Q_{k} and $Q_{\mathrm{k+1}}$ will be replaced by a single, larger query that combines the two. We denote the cost of this combined query by $\textsc{Cost}(Q_{\mathrm{k}}Q_{\mathrm{k+1}})$. Unlike $\textsc{Cost}(Q_{\mathrm{k}})$ and $\textsc{Cost}(Q_{\mathrm{k+1}})$, $\textsc{Cost}(Q_{\mathrm{k}}Q_{\mathrm{k+1}})$ cannot be directly estimated from observations, since Scalpel will not have observed the combined query during the training period. Instead, Scalpel estimates this cost to be the sum of the costs of the component queries minus a per-request overhead U_0:

$$\textsc{Cost}(Q_{\mathrm{k}}Q_{\mathrm{k+1}}) = \textsc{Cost}(Q_{\mathrm{k}}) + \textsc{Cost}(Q_{\mathrm{k+1}}) - U_0 \quad (1)$$

This reflects the fact that combining the two queries eliminates the per-request overhead associated with submitting $Q_{\mathrm{k+1}}$ to the server as a separate query. The value of U_0 is configuration-specific, and Scalpel estimates its value during a calibration period in the training phase. Equation 1 is conservative in that it assumes that the server and client costs are independent in the combined query. In some cases, the combined query may actually be cheaper than the sum of the individual costs, for example if the DBMS is able to exploit common sub-expressions within the two queries. However, we do not expect the combined query to be more expensive than this sum of individual costs as the naïve nested loops strategy will give this cost.

We define the benefit of prefetching $Q_{\mathrm{k+1}}$ from context C as

$$\textsc{Benefit}(Q_{\mathrm{k+1}}, C) = \textsc{Cost}(Q_{\mathrm{k+1}}, C) - \textsc{Cost}(Q_{\mathrm{k}}Q_{\mathrm{k+1}})$$

That is, prefetching is beneficial if the cost of doing so is less than the cost of not prefetching. Substituting and rearranging terms, we can rewrite this formula as

$$\textsc{Benefit}(Q_{\mathrm{k+1}}, C) = U_0 - (1 - P[Q_{\mathrm{k+1}}|C])\textsc{Cost}(Q_{\mathrm{k+1}}) \quad (2)$$

This formula provides a basis for deciding whether prefetching $Q_{\mathrm{k+1}}$ is a cost-effective execution strategy. It shows that the maximum benefit for a single prefetch operation is given by U_0, and that prefetching is most beneficial when $Q_{\mathrm{k+1}}$ is inexpensive and highly likely to occur.

3.2.2 Estimation Confidence

To use Equation 2 to determine the benefit of prefetching from context C, the Pattern Optimizer must rely on estimates of the cost of $Q_{\mathrm{k+1}}$ and on its estimate $\hat{p}$ of the probability $P[Q_{\mathrm{k+1}}|C]$. Of particular concern is $\hat{p}$, which is determined by the number of times $Q_{\mathrm{k+1}}$ was observed to occur in context C. That estimate may be very uncertain for contexts that were not observed frequently in the training trace. For example, $n(C') = 1$ and $n(C) = 2$ yields $\hat{p} = 0.5$, as does $n(C') = 100$ and $n(C) = 200$. However, the former estimate is based on a single observation of $Q_{\mathrm{k+1}}$ in context C, while the latter is based on a hundred such observations. In general, very specific k-contexts (those with large k values) will be observed much less often than very general k-contexts (those with small values of k). Thus, Scalpel's estimates of the benefit of prefetching from rarely-observed contexts will be less certain than its estimates from frequently-observed contexts. We would like Scalpel's cost-based prefetching decisions to reflect this.

To achieve this, the Pattern Optimizer defines a confidence interval around its each of its estimates $\hat{p}$, using a confidence level which is specified as a parameter to the Scalpel system. If the Pattern Optimizer can determine with the specified confidence that it is beneficial to prefetch $Q_{\mathrm{k+1}}$ from context C, then it will decide to prefetch from C. If it can determine with confidence that prefetching is not beneficial, then it will not prefetch. Otherwise, the Optimizer is said to be *uncertain*.

When the Pattern Optimizer is uncertain about prefetching $Q_{\mathrm{k+1}}$ from C, it considers *generalizations* of the context C to resolve the uncertainty. For example, if the Optimizer is uncertain about prefetching Q_{x} from

[3]Note that $n(C) \geq n(C')$ since the application must always enter context C prior to entering context C'.

$[Q_\mathrm{b}, Q_\mathrm{c}, Q_\mathrm{d}, Q_\mathrm{b}, Q_\mathrm{e}]$, then it considers prefetching from $[Q_\mathrm{c}, Q_\mathrm{d}, Q_\mathrm{b}, Q_\mathrm{e}]$, $[Q_\mathrm{d}, Q_\mathrm{b}, Q_\mathrm{e}]$, and so on until it is able to decide with certainty about Q_x. These more general contexts will have been observed at least as frequently as C in the training trace. Thus, as the Optimizer considers more general contexts, it should become more certain about its estimates, until eventually it can decide with confidence that prefetching is or is not beneficial. If the Optimizer can find no generalization for which it is confident, then the prefetching is deemed not to be beneficial. Note from the preceding example that the generalization of a 5-context is a 4-context, the generalization of a 4-context is a 3-context, and so on. Thus, this is the mechanism by which Scalpel avoids basing prefetching decisions on overly-specific k-contexts, i.e., those for which k is too large.

3.2.3 Choosing Queries to Prefetch

So far, we have described how Scalpel makes a cost-based decision about whether it is feasible and beneficial to prefetch a given query from a particular k-context. In general, Scalpel can decide to use *deep* prefetching or *wide* prefetching (or both) from any context C. There may be several queries that have been observed to follow C, and that are feasible and beneficial to prefetch from C. Wide prefetching from C means prefetching one or more of these alternative queries, in the hope that the application will actually request one of them. Scalpel may also be able to predict that an entire *sequence* of queries will follow C, and prefetch the entire sequence when the application enters C. For example, from the training trace shown in Figure 5, Scalpel might determine that it is feasible and beneficial to prefetch the sequence $Q_\mathrm{b}Q_\mathrm{c}$ from context $[Q_\mathrm{a}]$. We refer to this as deep prefetching. We have only described how Scalpel estimates the benefit of prefetching a single query from a given context, but it is relatively straightforward to generalize our cost-based approach to sequences of queries.

Our current Pattern Optimizer considers deep prefetching but not wide prefetching. It uses a greedy heuristic optimization procedure to choose a sequence of queries (possibly empty) to prefetch from a each context C. This procedure first determines the *most beneficial*, feasible single query prefetch from C. If there is no feasible, beneficial query then Scalpel does not prefetch from C. Otherwise, suppose that $Q_\mathrm{k+1}$ is the most beneficial, feasible query to prefetch. Scalpel then considers prefetching two-query sequences for which the first query is $Q_\mathrm{k+1}$. If there is no such feasible sequence that is more beneficial than prefetching $Q_\mathrm{k+1}$ alone, then it stops and elects to prefetch only $Q_\mathrm{k+1}$ from C. Otherwise, it considers three query sequences that have the best two-query sequence as a prefix, and so on.

3.2.4 Removing Redundancy

The Pattern Optimizer applies its optimization procedure to each k-context that was observed during the training trace. As a result, each such context is annotated with a set of zero or more queries that should be prefetched from that context. These annotations are used to control Scalpel's prefetching behavior at run-time, as described in Section 4.

There may be a significant amount of redundancy in these annotations. Redundancy arises from the fact that the Pattern Optimizer may make the same prefetching decisions for a context and its generalizations. To understand this issue, consider the following contexts from Figure 6: $C_{10} = [Q_\mathrm{c}]$, $C_9 = [Q_\mathrm{b}, Q_\mathrm{c}]$, $C_8 = [Q_\mathrm{a}, Q_\mathrm{b}, Q_\mathrm{c}]$, $C_7 = [Q_\mathrm{x}, Q_\mathrm{a}, Q_\mathrm{b}, Q_\mathrm{c}]$. This is a sequence of successively less general contexts. Suppose that, after the optimization procedure has been run, Scalpel has decided not to prefetch from context C_{10}, but to prefetch some query Q_p from context C_9. This may happen because Scalpel observes that the conditional probability $P[Q_\mathrm{p}|C_9]$ is higher than the conditional probability $P[Q_\mathrm{p}|C_{10}]$. (In this way, Scalpel avoids basing prefetching decisions on contexts that are too short.) The Pattern Optimizer may also decide that it is worthwhile to prefetch Q_p in context C_8. Such a prefetching decision is redundant, because whenever the system is in context C_8, it is also in the more general context C_9, for which the same prefetching decision has been made. In general, we define a k-context to be *redundant* if it has the same prefetching annotation as its most specific generalization. Although redundant contexts will not affect the way that Scalpel behaves at run-time, they do introduce additional storage and run-time execution overhead. To avoid this, the Pattern Optimizer prunes all redundant contexts.

3.2.5 Query Rewrites

Suppose that $C = [Q_1, \ldots, Q_\mathrm{k}]$ is a k-context and that the Pattern Optimizer has decided to prefetch $Q_\mathrm{k+1}$ from C. To accomplish the prefetch, Scalpel must generate a single, combined query that will return the results of both Q_k and $Q_\mathrm{k+1}$, and that can be executed in place of Q_k when context C is entered. Figure 3 showed one way to accomplish this for two specific queries, Q_a and Q_b. To combine arbitrary queries, Scalpel uses the `LATERAL` derived table construct of SQL 99, as illustrated in Figure 7 [3].[4] By doing so, Scalpel is effectively leaving the task of "flattening" the combined query to the more sophisticated query optimizer at the server.

In Figure 7, the text of the second query (`Q2.sql`) may contain parameter markers. Scalpel replaces these (not

[4]The SQL 99 standard does not support an outer-join variant of lateral derived tables, although several commercial systems do support this capability using vendor-specifi c syntax. We use the natural extension of `LATERAL` to outer joins to combine queries for prefetching.

```
SELECT   <Q1.columns>, <Q2.columns>
FROM     ( <Q1.sql> ) T1
         LEFT OUTER LATERAL ( <Q2.sql> ) I
ORDER BY <Q1.orderby>, <Q2.orderby>
```

Figure 7. Combining Two Arbitrary Queries (Q1 and Q2) Using LATERAL.

shown in Figure 7) with references to the result attributes or input parameters of the first query with which they are correlated.

Scalpel also implements a second type of rewrite based on outer union [3], where each branch of the union is identified by a unique type field, and the rows are ordered to match the expected sequence of queries. In principle, Scalpel's Pattern Optimizer should consider both rewrites for each prefetch, and choose the one with the least cost. At present, however, Scalpel uses a heuristic to choose one of the two rewrites for each prefetch. Specifically, Scalpel uses the outer join rewrite if it is able to infer that the first query will return at most one row. Otherwise, it uses the outer union rewrite. The outer join rewrite introduces redundancy into the result of the combined query when the second query returns multiple rows, because the values returned by the first query will occur multiple times in the combined query's result. However, the at-most-one heuristic ensures that no redundant rows are generated.

4. Running Scalpel

At run time, Scalpel's Prefetcher loads the non-redundant contexts that were saved by the Pattern Optimizer after the training period, together with their prefetching annotations. We will denote the set of such contexts by $\mathcal{C}$. As the application program runs, Scalpel monitors the requested queries and the Prefetcher tracks tracks a current context within $\mathcal{C}$. Suppose that the Prefetcher's current context is $C = [Q_1, \ldots, Q_{\mathrm{k}}]$. When the application requests $\textsc{Open}(Q)$, the Prefetcher chooses a new current context as follows. If $CQ = [Q_1, \ldots, Q_{\mathrm{k}}, Q]$ is in $\mathcal{C}$, then CQ becomes the new current context. Otherwise, the new context becomes C_gQ, where C_g is the most specific generalization of C for which $C_gQ \in \mathcal{C}$. If there is no such C_g, then the empty context becomes the new current context. This procedure ensures that the Prefetcher bases its actions on the most specific non-redundant context that matches the application's observed behavior.

After it has updated the current context, the Prefetcher responds to the application's request. There are several cases that it must consider. The first case is that the requested query Q has been prefetched from a previous context. In this case, the Prefetcher checks the input parameter values for Q to ensure that those values are correlated to input and/or output parameters of previous queries as was predicted during training. So that it can make this check, the Prefetcher records the input and output parameter values of all of the queries in the current context. If the actual parameter values are not correlated as predicated, then Scalpel cannot use the prefetched results to answer Q. Instead, it issues Q unmodified to the server to satisfy the application's request. This ensures that the application will receive correct query results, but work that was required to prefetch Q has been wasted. If the actual parameter values are correlated as predicted, then the Prefetcher can avoid sending any request to the server. When the application subsequently fetches the results of Q, Scalpel satisfies those requests by extracting the required data from the result of the prefetch query.

If query Q has not been prefetched, then the Prefetcher checks the new current context to determine whether it has a prefetching annotation. If there is not an annotation, then the Prefetcher simply submits Q unmodified to the server. However, if there is an annotation, that indicates that the Pattern Optimizer decided that it was beneficial to prefetch one or more queries from the current context. In that case, the Prefetcher submits the rewritten prefetch query (which returns results for Q as well as some predicted future queries) to the server instead of Q. When the application subsequently fetches results from Q, Scalpel satisfies these requests by extracting the necessary values from the results of the prefetch query.

When an application issues an update request, that update may conflict with query results that have been prefetched by Scalpel on the application's behalf. Scalpel handles this potential problem conservatively by simply invalidating any prefetched query results when an update occurs. Further discussion of updates can be found in [3].

5. Performance

We performed a variety of experiments that were designed to answer two general questions. First, what are the costs associated with executing a sequence of requests? Second, how effective is semantic prefetching at reducing the execution time of query batches when they occur? The first question is addressed in Section 5.1. The second is addressed in Section 5.2.

We have built a prototype version of Scalpel, which has been used to run a variety of experiments. The client-side driver programs are implemented in Java using JDBC, and all results are reported using Java 2 Standard Edition, version 1.5.0. On the server side, we experimented with three different commercial database systems behind Scalpel (license restrictions prevent us from identifying them). The results for the three database systems were consistent, although with different constants, so we have presented the results for only one system.

We studied three different system configurations with varying network latency. These configurations are described in Table 1 using the computers described in Table 2.

Config.	Client	Server	Connection
LCL	A	A	Shared memory
LAN1	B	A	1Gbps LAN
LAN0.1	B	A	100Mbps LAN

Table 1. Tested Configurations

Name	Processor	O/S
A	2 × 2.2GHz XEON	Win2003 Server
B	3GHz Pentium IV	WinXP

Table 2. Available Computers

5.1. Alternative Prefetch Strategies

Our first experiment is intended to evaluate whether Scalpel's query rewriting approach is an effective way to reduce the total cost of executing a sequence of small queries. We consider a sequence of length L of simple queries of the form '`SELECT pk FROM T WHERE pk=:i`', where `i` is the position within the sequence, and we consider four different ways of executing such a sequence:

Sequential (S) Each of the L requests is submitted individually, in sequence.

Stored Procedure (P) A single stored procedure is used, returning a separate result set for each of the L queries.

Join (J) A single request is used, combining all L queries using lateral derived tables as is done by Scalpel.

IN-List (I) The L queries are collapsed into a single query that uses an IN-list to identify the L rows of the table to be fetched.

The IN-list approach is not a general query rewriting technique. It is only possible because of the special structure of this benchmark workload. However, we have considered it here because it provides a useful lower bound.

Figure 8 shows the measured cost for each of these four strategies. Each data point was obtained as follows. First, all queries used by the strategy were prepared. Then, the queries were executed 1000 times, fetching all rows. This experiment does not include the cost of running Scalpel at all, merely the costs of executing the prefetch queries.

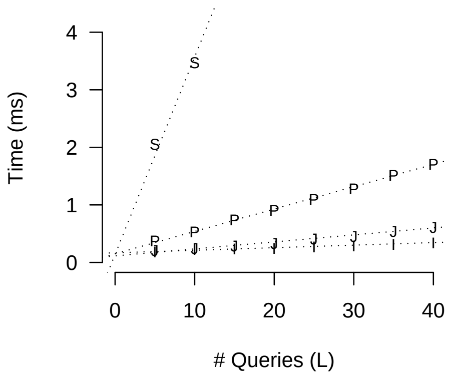


Figure 8. Cost of executing L queries.

As we would expect, the sequential strategy (S) grows quite quickly with increasing L as per-request overheads are incurred for each request. The stored procedure approach (P) improves on this by amortizing the costs associated with calling the procedure over all L queries. However, the stored procedure approach still has costs associated with crossing the relational/procedural boundary for each encoded result. These costs lead the stored procedure approach to be rather far from the IN-list lower bound.

In contrast, the join-based approach (J) is much closer to the IN-list lower bound (I). The join strategy uses a separate quantifier for each of the L original queries, while the IN-list exploits the structure of our workload to use a single quantifier. Despite this, the join approach is relatively close in cost to this lower bound. For this reason, Scalpel chooses not to use stored procedure based rewrites. Although these rewrites are simpler to express, the performance is not as good.

5.2. Effectiveness of Semantic Prefetching

To study the effectiveness of Scalpel's prefetching, we consider a scenario in which Scalpel, during its training phase, predicts that a query Q_0 will be followed by a batch of queries Q_1,Q_2,. . .,Q_L. Each query is a simple single-row selection from a table `T`, and each predicted query Q_i is correlated with Q_{i-1}, its immediately predecessor in the query batch.

The program generates the initial query, Q_0, followed by a prefix of the remainder of the batch, and then repeats this N times. The length of the prefix is PL, where $0 < P \leq 1$. When $P = 1$, each run-time batch exactly matches Scalpel's prediction. When $P < 1$, some of the queries predicted by Scalpel are not generated at run-time. This models the situation in which Scalpel's prefetching predictions are not completely accurate.

We wrote a simple driver application that generates such a query batch repeatedly, with the batch length (L) as an application parameter. For each experiment, we choose a

```
▷ N is the number of batches to generate.
▷ L is the batch length.
▷ P controls predicate selectivity.
procedure GenQueryBatches( N, L, P )
  for iteration ← 1 to N do
    generate query $Q_0$
    for i ← 1 to PL do
      generate query $Q_i$
end
```

Figure 9. Code for Generating Query Stream

value for L and we execute the resulting application twice, once without Scalpel and once with it. In the former case, which we call *unoptimized*, each query is passed directly to the database server for execution. In the latter *optimized* case, queries are passed through Scalpel, which applies its rewrite to prefetch Q_1,Q_2,. . .,Q_L when the application requests Q_0. This experiment measures the benefit that can be obtained through semantic prefetching for the ideal case in which Scalpel has accurately predicted the occurrence of a query batch. Figure 10 shows the results.

5.2.1 Batch Length

The benefit of prefetching depends on the number of queries that are successfully prefetched and on the latency of the communication. In this experiment, we fixed $P = 1$, varied the batch length L, and measured execution time for the original and optimized strategies.

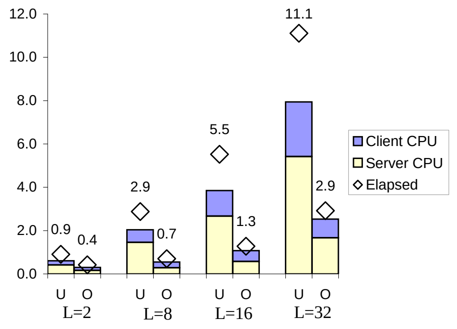


Figure 10. Time for unoptimized (U) and optimized (O) strategies with varying batch length L, $P = 1$.

As the number of successful prefetches increases, the relative benefit increases. It is interesting to note that client processing costs are *reduced* in the optimized case despite the presence of Scalpel on the client side. The overhead that Scalpel adds to track the current context is more than offset by overhead reductions achieved by prefetching. On the server side, costs are reduced substantially because of reduced overhead.

5.2.2 Useful Prefetches

For the next experiment, we fixed $L = 8$ and varied P in the range $0 \leq P \leq 0.5$. When P is small, only a small portion of the query batch prefetched by Scalpel is actually used by the application. Such overly aggressive prefetching can occur for several reasons. First, Scalpel may be unaware that some of the batch queries are conditional; that is, there may be prediction error. Second, Scalpel may be aware that part of the batch is conditional and yet decide, on a cost basis, that prefetching is still worthwhile.

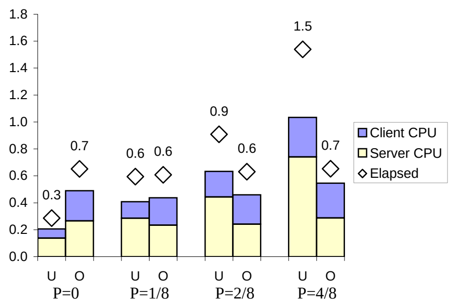


Figure 11. Time for unoptimized (U) and optimized (O) strategies with $L = 8$ and varying P.

Figure 11 shows the results of our measurements for the LAN1 configuration. As expected, the unoptimized strategy has a strong dependence on the proportion P of queries that are actually submitted, while the optimized strategy has only a weak dependence on P resulting from the costs associated with tracking the context as queries are opened, detecting if prefetched results are valid, and interpreting the prefetched results.

In general, as P increases, there is a threshold above which the optimized (prefetching) strategy becomes worthwhile. This threshold depends on system parameters, in particular network latency and the cost of individual queries relative to per-request overhead. This illustrates one of the advantages of Scalpel's run-time optimization strategy, since these costs may not be well understood at development time, or they may vary among instances of the application program.

6. Case Study

We have examined a number of real applications to understand the opportunities that are available and the benefits that are possible with the types of optimizations we have described. In this section, we present the results for the SQL-Ledger system. SQL-Ledger is a web-based double-entry accounting system implemented in Perl [9]. Business logic executes in a web server, communicating with a DBMS via TCP/IP.

We tested the system with a synthetic instance representing the data needed by a medium-sized company. This synthetic instance easily fits completely in memory. We generated an artificial workload based on a mix of 5 of SQL-Ledger's accounts receivable activities, a subset of the system functionality. These user activities contain between 2 and 20 steps, with each step involving an HTML form being submitted to the web server where it is parsed, interpreted, and used to generate a reply. For each step, the business logic in the web server may initiate one or more DBMS transactions. These transactions submit between 1 and 68 OPEN query requests, with median 2 and mean 5.5.

We generated two traces using the workload mix. One (39,909 requests) was used for training; the other (38,525 requests) was used for evaluation both with and without the selected optimizations. Approximately 60% of the queries in our traces were nested within open cursors, with the remaining 40% being top-level queries. Since nested queries are not considered for the batch optimizations described in this paper, we focused on the top-level queries and ignored the nested ones.

We first trained Scalpel using the training trace, and measured the training overhead for the systems that we considered. Training overhead ranged as high as 2ms per query, with an average increase of 35% per query. Most of this increase comes from client processing costs related to tracking correlations, maintaining the suffix trie, and optimizing the resulting trie. Server costs did increase slightly, but it appears that training does not greatly affect other connected clients. This allows us to train Scalpel while other users are using a production server.

Because of limitations of the underlying DBMS used by SQL-Ledger, we restricted Scalpel to consider prefetching only single-row queries.[5] From the training trace, Scalpel constructed a trie with approximately 23,000 nodes (contexts). Pruning of redundant nodes reduced this to 27 contexts, 7 of which had associated single-row prefetches. After training, we ran Scalpel using the evaluation trace. Of the 15,981 top-level requests in that trace, Scalpel prefetched the results for 1,416 queries; of these, 124 were not needed, representing wasted work. The other 1,292 represent 8% of the top-level requests. Top level queries accounted for 63.8s of execution time without optimization. Scalpel's prefetching reduced this by 11.3s, a reduction of 18% of the cost available for optimization in this setup.

If we allow Scalpel to consider prefetching multi-row queries, it identifies 21 contexts with recommended prefetch lists, in addition to the 7 contexts reported above. We would expect a commensurate reduction in cost had Scalpel been able to exploit these additional opportunities.

In addition to SQL-Ledger, we surveyed a small set of other systems: dbunload, a database schema extractor; TM, a Java-based time management GUI; and, Compière, a Java-based ERP system. We also examined a number of traces generated by proprietary applications. We found batches of sequential queries are submitted by each of these systems. Further, many of the requests issued by these applications are very cheap (relative to the per-request overhead). For these sequences, Scalpel can improve total execution time by combining requests.

7. Related Work

The idea of *prefetching* the results of anticipated future requests has been well studied in a number of areas. Many operating systems and DBMSs use predictions of simple patterns such as sequential access to prefetch results that may be needed in the future.

While simple patterns such as sequential access are useful, there are many workloads that have predictable access patterns that can not be simply characterized using heuristics such as sequential access. Palmer and Zdonik [7] described Fido, a predictive cache that uses an associative memory to learn to recognize patterns and predicts accesses. Krishnan, Vitter and Curewitz extended this idea using techniques from data compression [5, 6]. By using a variant of the Lempel-Ziv compression algorithm, they were able to provide a prefetcher that converges in the limit to the best possible prefetcher; they also used a prefetcher based on PPM; in practice, that algorithm provided faster convergence. PPM compression is based on suffix tries. Ukkonen described a method for building path-compressed suffix tries on-line in linear space and time [10]. Bunton demonstrated how path compressed suffix tries can be used to predict the probability of future characters in the PPM* algorithm [4].

The above techniques treat each request as an opaque block. Bernstein, Pal and Shutt suggested that the context of a fetch be considered to make prefetching decisions [1]. For example, if a list of objects is fetched to the client and then attributes of the first list elements are accessed, it may make sense to prefetch the same attributes for further elements of the list. In previous work, we used similar semantic observations to prefetch results for queries submitted within a loop over query results [3].

Once a sequence of predicted requests has been found, they can be combined by a DBMS optimizer to exploit common expressions [8]. Further, Yao and An [11] and Bilgin, Chirkova, Salo, and Singh [2] considered ways to combine

[5] Scalpel itself is capable of prefetching multi-row queries, but this specifi c DBMS requires casts for the NULL values used in the outer union (unlike the other 2 DBMSs we tested).

queries for the purposes of reducing latency if the probability of future sequences of requests is known.

In contrast to other work, our work considers prefetching parameterized requests. We learn the correlations between values observed earlier in the request trace, and use these to form combined queries for prefetching.

8. Conclusions

We have presented Scalpel, a system for optimizing streams of requests from database client applications. Scalpel learns to predict occurrences of optimizable sequences of correlated requests by monitoring a request stream during a training period. It optimizes a predicted request sequence by combining the individual requests in the sequence into a single larger query which it can issue when it predicts that the sequence will occur. The ideal way to execute a particular sequence of requests depends on run-time factors, such as the communication latency between the client and the server. Furthermore, optimizations may depend on the selectivities of application predicates, which may depend on input parameter values and on the database itself. Our experiments illustrate some of the advantages of optimizing the client request stream at run-time, as performed by Scalpel.

9 Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada and by Sybase iAnywhere.

References

[1] P. A. Bernstein, S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. In *VLDB*, pages 327–338, 1999.

[2] A. S. Bilgin, R. Y. Chirkova, T. J. Salo, and M. P. Singh. Deriving effi cient SQL sequences via read-aheads. In *Data Warehousing and Knowledge Discovery*, 2004.

[3] I. T. Bowman and K. Salem. Optimization of query streams using semantic prefetching. *ACM Transactions on Database Systems*, 30(4):1056–1101, 2005.

[4] S. Bunton. Semantically motivated improvements for PPM variants. *The Computer Journal*, 40(2/3):77–93, 1997.

[5] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *SIGMOD*, pages 257–266, 1993.

[6] P. Krishnan and J. S. Vitter. Optimal prediction for prefetching in the worst case. *SIAM Journal on Computing*, 27(6):1617–1636, 1998.

[7] M. Palmer and S. B. Zdonik. Fido: A cache that learns to fetch. In *VLDB*, pages 255–264, 1991.

[8] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[9] D. Simader. *SQL Ledger Accounting: User Guide and Reference Manual for Version 2.2*. DWS Systems Inc., Mar. 2004.

[10] E. Ukkonen. On-line construction of suffi x-trees. *Algorithmica*, 14:249–260, 1995.

[11] Q. Yao and A. An. Characterizing database user's access patterns. In *DEXA*, pages 528–538, 2004.