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

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

Figure 1 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 and described in an extended version of this paper [4].

If the data needed by the application are 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 2. This would improve not only response time, but also system throughput.

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 2. We called this technique *semantic prefetching*.

In our earlier work, we focused on one type of request pattern: query nesting. 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 1. The primary contribution of this paper is a technique for *automatically identifying optimizable query batches* in an application request stream.

2. Scalpel System Overview

The components of the Scalpel system are illustrated in Figure 3. 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.

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. Figure 4 shows a hypothetical applica-

```
1  procedure GetCustomer(cust)
2    fetch row r1 from Qa:
3      SELECT name, accno FROM customer c
4      WHERE c.id = :cust.id
5    cust.name ← r1.name
6    if not cust.shipto  then
7      cust.shipto ← GetDefaultShipTo(cust)
8    fetch row r3 from Qc:
9      SELECT SUM(amount-paid) as balance
10     FROM ar a WHERE a.accno = :r1.accno
11   cust.balance = r3.balance
12 end
13 function GetDefaultShipTo(info)
14   fetch row r2 from Qb:
15     SELECT addr FROM shipto s
16     WHERE s.cid = :info.id AND s.default='Y'
17   return r2.addr
18 end
19 procedure GetVendor(vend)
20   fetch row r4 from Qd:
21     SELECT name FROM vendor v
22     WHERE v.id = :vend.id
23   vend.name ← r4.name
24   vend.mailto ← GetDefaultShipTo(vend)
25   open c5 cursor for Qe:
26     SELECT partname, invlevel-onhand AS qty
27     FROM part p WHERE p.vid = :vend.id
28     AND p.onhand < p.invlevel
29   while r5 ← fetch c5 do AddOrder(vend,r5) end
30   close c5
31 end
```

Figure 1. 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 2. Manually joining queries Q_a and Q_b.

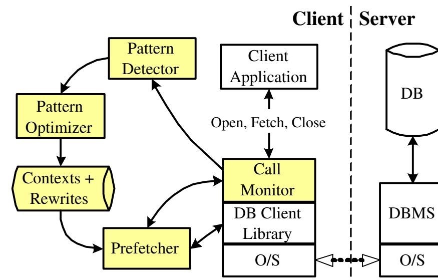

Figure 3. Components of the Scalpel system.

#	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 4. An example trace.

tion 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 1, as well as other code that we have not shown. Each row of the trace table in Figure 4 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 1, 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.

3. Training Scalpel

Scalpel predicts upcoming queries based on the queries that it has observed so far. Like many other prefetchers, 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 $p-(k-1), p-(k-2), \dots, p-1, p$. For example, the 5-context at position 7 in the trace of Figure 4 is the list $[Q_b, Q_c, Q_d, Q_b, Q_e]$.

Scalpel works by learning k-contexts from which it can predict that a particular query will occur next. Scalpel learns by examining a training request trace like the one illustrated in Figure 4. For example, from the trace in Figure 4, Scalpel might learn that the 2-context $[Q_a, Q_b]$ predicts the query Q_c. These predictions are recorded in a rewrite database shown in Figure 3 and used to control prefetching at run

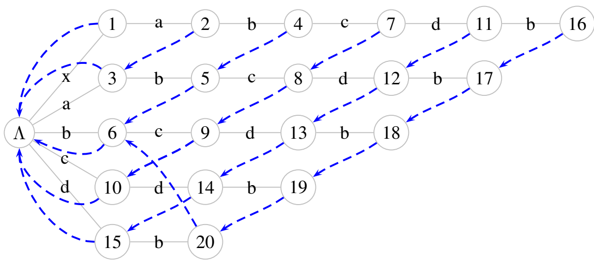

Figure 5. Suffix Trie After $Q_{\rm x}, Q_{\rm a}, Q_{\rm b}, Q_{\rm c}, Q_{\rm d}, Q_{\rm b}$ from Figure 4

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 4, the 1-context $[Q_{\rm b}]$ is sometimes followed by $Q_{\rm c}$ and sometimes by $Q_{\rm d}$. However, the 2-context $[Q_{\rm a}, Q_{\rm b}]$ is *always* followed by $Q_{\rm c}$. 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.

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 5 shows the suffix trie as it would look after the sixth query from Figure 4. Edges in the trie are labeled with the subscripts of queries from the trace, e.g., edge label $\rm a$ refers to query $Q_{\rm a}$. 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 $[Q_{\rm a}, Q_{\rm b}]$. 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 ($[Q_{\rm x}, Q_{\rm a}]$) points to node 3, which represents the more general context $[Q_{\rm a}]$.

Construction of a suffix trie as illustrated in Figure 5 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 [9].

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. 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. To illustrate how this works, consider Scalpel's behavior when it observes query $Q_{\rm c}$ on line 4 of Figure 4. (The trace in Figure 4 shows the query input and output parameter values.) Scalpel will note that $Q_{\rm c}$'s input parameter value (501) matches the value of the second result attribute of the preceding query $Q_{\rm a}$ as well as the value of the first result attribute of the preceding query $Q_{\rm x}$. Later, at line 13 of the trace, Scalpel will again observe $Q_{\rm c}$. This time, it will verify that $Q_{\rm c}$'s input parameter value (now 523) matches the second result attribute of the preceding query $Q_{\rm a}$. However, it will be unable to verify the correlation between $Q_{\rm c}$ and the result of the preceding $Q_{\rm x}$, 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 $Q_{\rm c}$'s input and $Q_{\rm x}$'s output. More information about detection of parameter correlations in Scalpel can be found in the extended paper [4].

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 $C = [Q_1, \ldots, Q_{\rm k}]$ is a k-context in the trie and that $C' = [Q_1, \ldots, Q_{\rm k}, Q_{\rm k+1}]$ is a *successor* context to C in the trie. This indicates that, on at least one occasion, the Pattern Detector observed that the query $Q_{\rm k+1}$ 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_{k+1} from context C.

If Scalpel is to prefetch Q_{k+1} from C, the prefetch should be both *feasible* and *beneficial*. We say that Q_{k+1} is a feasible prefetch from context C if the Pattern Detector observed at least one correlation for each input parameter of Q_{k+1} in context C'. Intuitively, this means that whenever Q_{k+1} followed C in the training trace, its input parameters were predictable. If query Q_{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_{k+1} would be beneficial.

3.2.1 Estimating the Benefit of Prefetching

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

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

where $\textsc{Cost}(Q_k)$ and $\textsc{Cost}(Q_{k+1})$ are the estimated costs of executing queries Q_k and Q_{k+1}, respectively, and $P[Q_{k+1}|C]$ is the probability that the application will request query Q_{k+1}, given that it is in context C. Scalpel estimates $\textsc{Cost}(Q_k)$ and $\textsc{Cost}(Q_{k+1})$ by monitoring, at the client, the observed execution times of Q_k and Q_{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 $P[Q_{k+1}|C]$, 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. For example, if $n(C) = 10$ and $n(C') = 4$, Scalpel will estimate a 40% probability that Q_{k+1} will occur next in context C.

If, on the other hand, Scalpel chooses to prefetch, then Q_k and Q_{k+1} will be replaced by a single, larger query that combines the two. We denote the cost of this combined query by $\textsc{Cost}(Q_k Q_{k+1})$. Unlike $\textsc{Cost}(Q_k)$ and $\textsc{Cost}(Q_{k+1})$, $\textsc{Cost}(Q_k Q_{k+1})$ 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:

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

This reflects the fact that combining the two queries eliminates the per-request overhead associated with submitting Q_{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_{k+1} from context C as

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

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

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

This formula provides a basis for deciding whether prefetching Q_{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_{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_{k+1} and on its estimate $\hat{p}$ of the probability $P[Q_{k+1}|C]$. Of particular concern is $\hat{p}$, which is determined by the number of times Q_{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_{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_{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_{k+1} from C, it considers *generalizations* of the context C to resolve the uncertainty. For example, if

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

Figure 6. Combining Two Arbitrary Queries (Q1 and Q2) Using LATERAL.

the Optimizer is uncertain about prefetching Q_{x} from $[Q_{\mathrm{b}}, Q_{\mathrm{c}}, Q_{\mathrm{d}}, Q_{\mathrm{b}}, Q_{\mathrm{e}}]$, 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.

3.2.3 Query Rewrites

Scalpel uses a greedy heuristic optimization procedure to choose a sequence of queries to prefetch from a each context C. 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 2 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 6 [3]. By doing so, Scalpel is effectively leaving the task of “flattening” the combined query to the more sophisticated query optimizer at the server.

4. Running Scalpel

At run time, Scalpel's Prefetcher uses the decisions made by the Pattern Optimizer to prefetch query results that are expected to be needed. Details of Scalpel's run-time behavior are described in detail in the extended paper [4], along with experimental results and a case study.

5. Related Work

The idea of *prefetching* the results of anticipated future requests has been well studied in a number of areas. 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]. Ukkonen described a method for building path-compressed suffix tries on-line in linear space and time [9]. 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].

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 [10] and Bilgin, Chirkova, Salo, and Singh [2] considered ways to combine queries for the purposes of reducing latency if the probability of future sequences of requests is known.

6. Conclusions

We have presented Scalpel, a system that learns to predict occurrences of optimizable sequences of correlated queries by monitoring a query request stream. We evaluated Scalpel's optimizations empirically and through a case study [4] and found that it produces significant benefits.

7 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 efficient 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] I. T. Bowman and K. Salem. Semantic prefetching of correlated query sequences. Technical Report CS-2006-43, David R. Cheriton School of Computer Science, University of Waterloo, Nov. 2006.

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

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