Optimization of Query Streams Using Semantic Prefetching

Ivan T. Bowman
School of Computer Science
University of Waterloo
itbowman@uwaterloo.ca

Kenneth Salem
School of Computer Science
University of Waterloo
kmsalem@uwaterloo.ca

ABSTRACT

Streams of relational queries submitted by client applications to database servers contain patterns that can be used to predict future requests. We present the Scalpel system, which detects these patterns and optimizes request streams using context-based predictions of future requests. Scalpel uses its predictions to provide a form of semantic prefetching, which involves combining a predicted series of requests into a single request that can be issued immediately. Scalpel's semantic prefetching reduces not only the latency experienced by the application but also the total cost of query evaluation. We describe how Scalpel learns to predict optimizable request patterns by observing the application's request stream during a training phase. We also describe the types of query pattern rewrites that Scalpel's cost-based optimizer considers. Finally, we present empirical results that show the costs and benefits of Scalpel's optimizations.

1. INTRODUCTION

Relational database applications establish database server connections through which they issue streams of query and update query requests and fetch the results of those requests. Although there has been a great deal of work on relational query optimization, most of that work focuses on optimizing the processing of individual requests by the database server. Here, we consider the problem of optimizing the *stream* of client/server interactions that occurs over a connection, rather than individual requests.

As an illustration of the kind of optimization we hope to achieve, consider the client-side database application code shown in Figure 1. The code is adapted from SQL-Ledger, a web-based double-entry accounting system written in Perl. The `get_openinvoices` function retrieves a list of all open invoices for a given customer that were recorded with a given monetary currency. If the given currency differs from the configured system default, then the exchange rate for each invoice is retrieved using the `get_exchangerate` function. When this application runs, it issues a series of small single-table queries (Q_1,Q_2,Q_2,Q_2,...) to the database server. It uses the results of these queries to perform a two-way, nested loops join on the client side.

```
function get_openinvoices( cust_id, currency )
    // this is Q1
    c1 = open( SELECT id, curr, transdate
               FROM   ar
               WHERE  customer_id = :cust_id
                 AND  ar.curr = :currency
                 AND  NOT ar.amount = ar.paid
               ORDER BY id )
    while( r1 = fetch(c1) )
        ...
        if( currency != defaultcurrency )
            rate = get_exchangerate( r1.curr,
                                     r1.transdate );
end

function get_exchangerate( curr, transdate )
    // this is Q2
    r2 = fetch( SELECT exchangerate FROM exchangerate
                WHERE  curr = :curr
                  AND  transdate = :transdate )
    return r2.exchangerate;
end
```

Figure 1: SQL-Ledger `get_openinvoices` Function

In this paper we present the Scalpel system, which optimizes application request streams using context-based predictions of upcoming requests. Scalpel monitors the requests issued by the client application and learns to recognize and optimize query patterns. For example, if Scalpel recognizes the nested query pattern $Q_1, Q_2, Q_2, Q_2, Q_2, \ldots$ generated by the application in Figure 1, it can replace the entire pattern with a single, larger query similar to Q_{opt}, which is shown in Figure 2. Query Q_{opt} performs the join at the server and returns all of the data that would have been returned by Q_1 and the Q_2s. Scalpel is transparent to the database client: no changes to the application code are required.

Scalpel's rewrites are *predictive.* When Scalpel sees Q_1, it predicts that Q_1 will be followed by a series of nested Q_2 queries. Based on this prediction, it issues Q_{opt}, rather than Q_1, to the server. If the application then requests Q_2 as expected, Scalpel does not pass Q_2 to the server. Instead, it extracts the required data from the result of Q_{opt} and returns that to the application. If the application behaves

```
SELECT id, curr, exchangerate, ...
FROM ar LEFT JOIN exchangerate er
     ON ar.curr = er.curr
     AND ar.transdate=er.transdate
WHERE  customer_id = :cust_id
       AND  ar.curr = :currency
       AND  NOT ar.amount = ar.paid
ORDER BY id
```

Figure 2: The Join Query Q_{opt}

unexpectedly, perhaps by issuing a different query Q_3, then Scalpel can simply forward Q_3 to the server for execution. In this case Scalpel has done some extra work, since Q_{opt} is a larger and more complex query than Q_1. However, Scalpel always returns correct results to the client. By replacing Q_1 with Q_{opt}, Scalpel implements a kind of prefetching. We call it *semantic prefetching* because Scalpel must understand the queries Q_1 and Q_2 in order to generate an appropriate Q_{opt}.

There are two reasons to do semantic prefetching. First, it provides the query optimizer at the server with more scope for optimization. For example, the application shown in Figure 1 effectively joins two tables at the client site. However, the server's optimizer will be unaware that the join is occurring. Scalpel's rewrite makes the server aware of the join, allowing its optimizer to consider alternative join methods that it may implement.

Second, by replacing many small queries with fewer larger queries, Scalpel can reduce the latency and overhead associated with the interconnection network and the layers of system interface and communications software at both ends of the connection. These costs can be quite significant. We measured the cost of fetching a single in-memory row from several commercial relational database management systems. Regardless of whether the client used local shared memory or an inter-city WAN to communicate with the server, overhead was consistently over 99% of the total query time. For the application shown in Figure 1, this means that almost all of the time spent issuing queries Q_2 to the server is overhead. Even for stored procedures executing in the DBMS process, overhead accounted for about 70% of the cost of a single-row fetch.

A potential objection to Scalpel is that the problems we have illustrated can be solved by manually rewriting the client applications. For example, the two functions shown in Figure 1 could be replaced by a single function that opens Q_{opt} (Figure 2). However, we claim that there is a place for both manual application tuning and automatic, run-time optimization of application request streams. Manual tuning can clearly improve application performance, but run-time optimization has some strengths that application tuning does not. First, run-time optimization can take advantage of information that is not known at application development time, or that varies from installation to installation. For example, when the monetary currency of the report differs from the default currency, the implementation of the `get_openinvoices` function shown in Figure 1 is much worse than a revised implementation based on the join query of Figure 2. However, if the report currency and the default currency are the same, the implementation of Figure 1 will perform best. For which of these circumstances should the implementation be tuned? The programmer may not know the answer to this question; worse, the answer may be different for different instances of the program. Other examples of run-time information that may have a significant impact on the performance of the application are program parameter values, data distributions, and system parameters such as network latency. A run-time optimizer can consider these factors in deciding how best to interact with the database server.

A second argument in favor of run-time optimization is a software engineering argument. Performance is not the only issue to be considered when designing and implementing an application. For example, the SQL-Ledger application actually calls `get_exchangerate` from eight locations. Only one of these calls is shown in Figure 1. Rewriting `get_openinvoices` to use the join query of Figure 2 breaks the encapsulation of the exchange rate computation that was present in the original implementation, resulting in duplication of the application's exchange rate logic. This kind of duplication can lead to increased development cost and possible maintenance issues.

Finally, there is the issue of the time and effort required to tune applications. While we do not expect Scalpel to eliminate the need for manual application tuning, any performance tuning that can be accomplished automatically can reduce the manual tuning effort. Scalpel may be particularly beneficial for tuning automatically or semi-automatically generated application programs, for which there may be little or no opportunity for manual tuning.

This paper makes the following contributions. First, in Section 2, we identify some types of optimizable query patterns that are present in real applications. In Sections 3, 4 and 5 we describe how Scalpel monitors application request streams and learns to predict rewritable query patterns. In Sections 6 and 7 we describe Scalpel's cost-based optimizer, and show how it selects effective predictive rewrites for such patterns. Finally, in Section 8 we present experimental results which show the costs and benefits of Scalpel's semantic prefetching in several different environments.

2. APPLICATION REQUEST PATTERNS

We surveyed a small set of database application programs to identify the kinds of query patterns they produce, and the prevalence of those patterns. Our sample included the following applications.

- **SQL-Ledger:** A web-based double-entry accounting system written in Perl.
- **Slaschode:** A web forum written in Perl.
- **Compière:** A Java-based ERP system.
- **TM:** A Java-based time-management GUI.
- **Schema Dump:** A C program that writes DDL to re-create the schema of a database.

In addition to these systems, we investigated a number of proprietary applications, ranging from on-line order processing systems to report generating systems. While privacy concerns prevent us from giving details for those applications, the results of our analysis of those applications were consistent with the results from the applications listed above.

From our application sample, we identified three types of query patterns that are amenable to optimization: batches, nesting, and data structure correlations.

```
function batch_example( cust_id, get_shipto )
  c1 = open( SELECT name, tax_id
             FROM   customer
             WHERE  id = :cust_id );
  r1 = fetch( c1 );
  close( c1 );

  if( get_shipto ) {
      c2 = open( SELECT * FROM shipto
                 WHERE  trans_id=:cust_id );
      r2 = fetch( c2 );
      close( c2 );
  }

  tax_id = r1.tax_id
  c3 = open( SELECT c.accno FROM chart c
             WHERE  ct.chart_id = :tax_id );
  while( r3 = fetch( c3 ) ) { ... }
  close( c3 );
end
```

Figure 3: Example of Batch Pattern

Batches: A batch is a sequence of related queries. For example, consider the `batch_example` function shown in Figure 3. In the `batch_example` function, two or three small queries are submitted to the database to retrieve different types of information about a customer. These small queries could potentially be replaced with a single, larger query.

Nesting: In batches, each query is opened, fetched, and closed independently. In the nesting pattern, one query is opened, and other queries are opened, executed, and closed for each row of the outer query. Figure 1 showed an example of an application that generates nesting query pattern. The nesting pattern effectively implements a nested loops join in the application.

Data Structure Correlation: In the nesting pattern, inner queries are executed while the outer query is open, expressing direct nesting in the application. In some cases, a client application opens an outer query, fetches the results into a data structure, then closes the outer query. Then, an inner query is executed for some or all of the values stored in the data structure. The performance impact of the data structure correlation pattern is similar to that of the nesting pattern. However, the pattern may be more difficult to detect due to the indirect nesting.

Batches were common in all of the applications we considered. Nesting or data structure correlation also occurred in each of the applications, although less frequently than batches. Although nesting and data structure correlation are less common than batches, the potential payoff for optimizing these patterns is greater because they usually allow more database requests to be eliminated. For example, the SQL-Ledger application in Figure 1 issues Q_2 once for each customer invoice. All of these queries can be replaced by a single query. Although both nesting and data structure correlation offer a large optimization payoff, nesting patterns are easier than data structure correlations to detect and optimize. We are currently working on techniques for optimizing all three types of request patterns. However, in this paper we will focus exclusively on nesting patterns.

3. REQUEST STREAMS

The Scalpel system works by intercepting, monitoring, and optimizing an application's database request stream. We assume that the client request stream consists of three types of requests:

Open: An OPEN request is used to send a query to the database server. It returns a cursor, which is used by the application to retrieve rows from the query result. The first parameter of an OPEN request is the query text. Queries may be parameterized. If an opened query is parameterized, the OPEN request also includes a value for each query parameter.

Fetch: A FETCH request takes a cursor as an input parameter. It returns a single row of the query result to the application.

Close: A CLOSE request takes a cursor as an input parameter. It is used by the application to indicate that it is finished retrieving rows from the query result.

The third column of Figure 4 shows a simple example of a request stream. (The first two columns of Figure 4 will be explained in Section 4.) This request stream is based on the application program that is shown in Figure 1. For illustrative purposes, the example shows specific input parameters values for each OPEN request, as well as specific tuples returned by each FETCH.

The example request stream illustrates a nested query pattern, in which Q_2 is nested within Q_1. This is clear because each OPEN/CLOSE pair for Q_2 occurs between the OPEN and CLOSE requests for Q_1. Currently, Scalpel works only with request streams that are strictly nested, like the example. Strict nesting means that if Q_a's OPEN precedes Q_b's OPEN, then Q_b's CLOSE precedes Q_a's CLOSE.

4. SCALPEL SYSTEM ARCHITECTURE

Scalpel operates in two phases. In the training phase (Figure 5), Scalpel monitors and analyzes database interface calls made by the application, and generates a set of request rewrite rules, which are recorded in a rewrite database. Scalpel's rewrites are condition/action rules. The conditions (called *contexts*) identify situations in which a query rewrite can be applied.

At run-time (Figure 6), Scalpel again monitors the client request stream and tracks the current context. Each time the client opens a query, Scalpel checks the current context against the rules in the rewrite database. If the context matches a rule condition, Scalpel applies the rule to rewrite the current query.

For an example, consider the simple request stream shown in Figure 4. To rewrite this nested query pattern, Scalpel first needs to learn a rewrite rule whose context consists of the query Q_1, and whose action rewrites Q_1 to Q_{opt} (Figure 2). When Scalpel detects OPEN(Q_1) at run-time, it triggers the rule and issues OPEN(Q_{opt}) instead. When the application fetches from Q_1, Scalpel instead fetches the appropriate data from Q_{opt}. If the application issues OPEN(Q_2) as predicted, Scalpel needs to do nothing. On an application fetch from Q_2, Scalpel again fetches the appropriate data from Q_{opt}.

Query Context	Parameter Context	Trace
-	-	OPEN(Q_1(cust1,CDN\$)) $\rightarrow$ c1
Q_1	(cust1,CDN\$,-,-,-)	FETCH(c1) $\rightarrow$ (3305,CDN\$,30/10/03)
Q_1	(cust1,CDN\$,3305,CDN\$,30/10/03)	OPEN(Q_2(CDN\$,30/10/03)) $\rightarrow$ c2
Q_1,Q_2	(cust1,CDN\$,3305,CDN\$,30/10/03),(CDN\$,30/10/03,-)	FETCH(c2) $\rightarrow$ 1.4304
Q_1,Q_2	(cust1,CDN\$,3305,CDN\$,30/10/03),(CDN\$,30/10/03,1.4304)	CLOSE(c2)
Q_1	(cust1,CDN\$,3305,CDN\$,30/10/03)	FETCH(c1) $\rightarrow$ (3307,CDN\$,30/10/03)
Q_1	(cust1,CDN\$,3307,CDN\$,30/10/03)	OPEN(Q_2(CDN\$,30/10/03)) $\rightarrow$ c2
Q_1,Q_2	(cust1,CDN\$,3307,CDN\$,30/10/03),(CDN\$,30/10/03,-)	FETCH(c2) $\rightarrow$ 1.4304
Q_1,Q_2	(cust1,CDN\$,3307,CDN\$,30/10/03),(CDN\$,30/10/03,1.4304)	CLOSE(c2)
Q_1	(cust1,CDN\$,3307,CDN\$,30/10/03)	FETCH(c1) $\rightarrow$ (3308,CDN\$,31/10/03)
Q_1	(cust1,CDN\$,3308,CDN\$,31/10/03)	OPEN(Q_2(CDN\$,31/10/03)) $\rightarrow$ c2
Q_1,Q_2	(cust1,CDN\$,3308,CDN\$,31/10/03),(CDN\$,31/10/03,-)	FETCH(c2) $\rightarrow$ 1.4522
Q_1,Q_2	(cust1,CDN\$,3308,CDN\$,31/10/03),(CDN\$,31/10/03,1.4522)	CLOSE(c2)
Q_1	(cust1,CDN\$,3308,CDN\$,31/10/03)	CLOSE(c1)

Figure 4: Trace Example Showing Query and Parameter Contexts

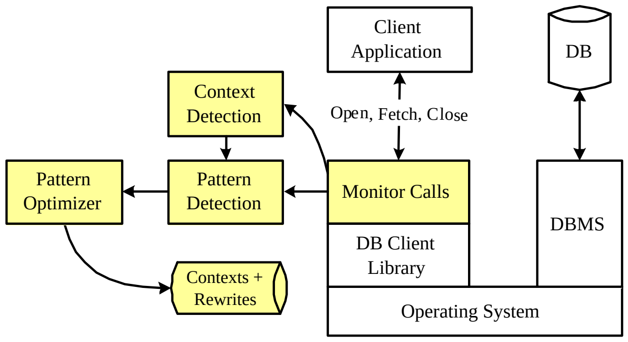

Figure 5: Scalpel Structure During Training Phase. Scalpel components are shaded.

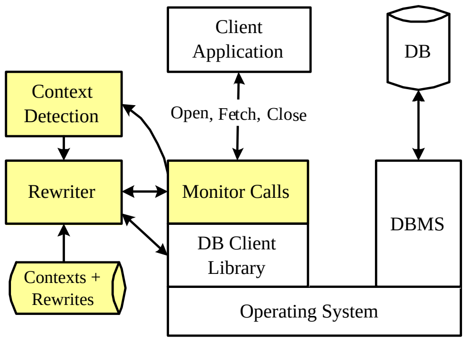

Figure 6: Scalpel Structure During Run-Time. Scalpel components are shaded.

5. TRAINING SCALPEL

During its training phase, Scalpel uses three functional modules, as illustrated in Figure 5. The context detection module is responsible for monitoring the application request stream and tracking the evolving request context. The pattern detection module is responsible for detecting correlations between queries and their contexts. The pattern optimizer takes candidate context/query pairs (patterns) from the pattern detector and determines whether there is a cost-effective rewrite for that pattern. Contexts and patterns are described in more detail in the following two subsections. The optimizer is discussed in Sections 6 and 7.

5.1 Request Context

Scalpel associates a *query context* with each request in the stream. Since Scalpel's focus is on detecting and optimizing nested query patterns, the query context of each request is defined to be the list of queries that are open at the time of the request. Queries are listed in the context in the order in which they were opened by the application. The first column in Figure 4 shows the query context that exists before each of the application requests in the example request stream.

Queries are often parameterized. Although the query context is sufficient to identify nested query patterns, Scalpel requires additional information if it is to be able rewrite these patterns. Consider the example optimization in which the nested Q_2 queries are predicted when Q_1 is opened, and the rewritten query Q_{opt} is issued to the database server in place of Q_1. Since Q_2 is parameterized, Scalpel needs to predict the values of Q_2's input parameters, and not just the occurrence of Q_2 itself. Otherwise, Scalpel will not be able to derive an appropriate query (Q_{opt}) with which to replace Q_1.

To accomplish this, Scalpel tries to identify nested queries whose input parameter values are *correlated* with specific input and output parameters of the queries in the current context. In the example request stream of Figure 4, the first input parameter of Q_2 (`curr`) matches the second output parameter (result column) of Q_1. More precisely, in each OPEN(Q_2), the value of Q_2's first parameter matches the value of the second column of the row returned by the most recent FETCH(Q_1). Similarly, the second input parameter of Q_2 (`transdate`) matches the third output parameter of Q_1. By monitoring input and output parameter values, Scalpel learns to predict correlations between the input parameters of the inner query and the input and output parameters of the queries in the context.

The context detection module maintains a *parameter context* in addition to the query context so that Scalpel can identify these parameter correlations. For each query in the context, the parameter context includes the values of its input parameters as well as the most recently fetched row from the query result. The second column of Figure 4 shows the parameter context at each point in the request stream. For each query in the query context, the parameter context lists the query's input parameter values followed by the values from the most recently fetched result tuple. When it is not important to distinguish the input elements of the parameter context from the output elements, we will simply refer to the kth element of the parameter context.

5.2 Query Patterns

Scalpel's pattern detection component tracks variable correlations and produces candidates for the pattern optimizer. Suppose that Q is a query and C is a query context. Let $Q[i]$ represent the ith input parameter of Q, and let $C[j][k]$ represent the k^{th} element of the parameter context of the jth query in C. We say that $Q[i]$ is correlated with C if there exist j and k such that for *every* Open(Q) in context C in the request stream, $Q[i] = C[j][k]$. In the example request stream in Figure 4, $Q_2[1]$ is correlated with the context consisting of query Q_1. This is because each time Q_2 is opened, the value of $Q[1]$ (the input parameter `curr`) is equal to the value of the second input parameter of Q_1 (as well as the value of second output parameter of Q_1). $Q_2[2]$ (the `transdate` parameter) is also correlated with C, since its value is always equal to the value of the third output parameter of Q_1.

To detect correlations, the pattern detector invokes the `TrackCorrelations` procedure, shown in Figure 7, each time the client opens a new query. `TrackCorrelations` takes the new query and the current query context (from the context detector) as input. In `inners`(C), the pattern detector records the queries that have appeared in context C. In `correlations`($C, Q[i]$), the pattern detector tracks the context parameters with which $Q[i]$ is correlated.

```
inners($C$): queries seen nested in context $C$
correlations($C, Q[i]$): correlations for $Q[i]$ in context $C$

procedure TrackCorrelations($C, Q$) {
  if ($Q \notin$ inners($C$)) {
      insert $Q$ into inners($C$);
      foreach $i$
         correlations($C, Q[i]$) = $\{(j,k) | \ C[j][k] = Q[i] \}$;
  } else
      foreach $i$
         foreach $(j,k) \in$ correlations($C, Q[i]$))
            if ( ($C[j][k] \neq Q[i]$)) then
               remove $(j,k)$ from correlations($C, Q[i]$);
}
```

Figure 7: Tracking Query Parameter Correlations

At the conclusion of the training phase, the pattern detection passes a set of candidate context/query pairs to the pattern optimizer. A context/query pair (C, Q) is a candidate if query Q is correlated with context C. A query is correlated with a context if all of the query's input parameters are correlated with that context. Thus, the pattern detector reports (C, Q) if $Q \in$ `inners`(C) and, for every i, `correlations`($C, Q[i]$) is not empty.

The parameter correlations in the example request trace are the manifestations of actual parameter correlations in the application code of Figure 1. In general, correlations observed by Scalpel may be the result of actual variable correlations in the application. However, they may also be mere coincidence. If we use a sufficiently long training period, most such coincidences should be discovered and eliminated from consideration. However, there is no guarantee that correlations inferred by the pattern detector will actually hold at run-time. This may cause Scalpel to generate semantic prefetches that are not useful. Since Scalpel can recognize such prefetches at run-time, this may impact the system's performance but it will not cause Scalpel to return incorrect query results to the application.

5.2.1 Client Predicate Selectivity

If Scalpel's pattern detector produces a candidate context/query pair (C, Q), this means that whenever Q occurs within context C, Q is correlated to C. However, this does not imply that Q occurs every time C occurs. Predicates within the client application may dictate that Q occurs in some cases but not in others. For example, in Figure 1, an application predicate (`currency != defaultcurrency`) determines whether the correlated inner query will or will not occur inside the outer query. The selectivity of client predicates is important to Scalpel because it affects the costs of the various semantic prefetching strategies (Section 6) that Scalpel's optimizer will consider. During the training phase, Scalpel estimates client predicate selectivities and then uses these estimates during cost-based optimization.

Scalpel uses two parameters to model the client predicate selectivity for each candidate context/query pair (C, Q). The first parameter, which we call P_0, represents the probability that Q will be opened at least once within context C. Thus, if the pattern detector reports (C, Q), and C occurs 5 times during the training phase, and Q's occurred within C on only two of these occasions, Scalpel will estimate $P_0 = 0.4$ for that context/query pair. The second parameter, which we call P_1, represents the probability that Q will be opened after each fetch from the innermost query of C, assuming that Q will occur at least once within C. To continue with the example, if, during the two occasions on which Q occurred within C, there were a total of 200 fetches from the innermost query of C and a total of 120 opens of Q, Scalpel will estimate $P_1 = 120/200 = 0.6$. Section 7.2 describes how Scalpel uses the selectivity parameters P_0 and P_1 for each context/query pair to estimate the costs of various execution strategies.

6. EXECUTION ALTERNATIVES

The Scalpel pattern detector finds a set of context/query pairs that are candidates for rewriting. For the simple example application shown in Figure 1, the training phase would ideally produce a single such pair, given a sufficiently long training period. For more complex applications the training phase may produce many such pairs. As an illustration of this, consider the example application fragment shown in Figure 8. In this case, Scalpel's training mechanism would ideally identify three context query pairs: `([Q_RST],Q_abcdef)`, `([Q_RST],Q_XYZ)`, and `([Q_RST,Q_XYZ],Q_1234)`. For each pair the context is listed first, in square brackets, followed by the query.

In general, the contexts identified during the training phase may be related to one other. We can consider these contexts to form a forest, where a context `[C,Q]` is a child of context `[C]` if `([C],Q)` is a candidate context/query pair. For example, the three context/query pairs that Scalpel would learn for the application in Figure 8 would form a single tree as

```
c1 = open( Q_RST );
while( r1 = fetch(c1) )
    c2 = open( Q_abcdef, r1 );
    while( r2 = fetch(c2) )
      ...
    close( c2 );
    c2 = open( Q_XYZ, r1 );
    while( r2 = fetch(c2) )
        c3 = open( Q_1234, r2 );
        while( r3 = fetch(c3) )
          ...
        close( c3 );
    close( c2 );
close( c1 );
```

Figure 8: Multiple Query Example. Each query is labeled with a suffix denoting the rows returned by that query.

illustrated in Figure 9.

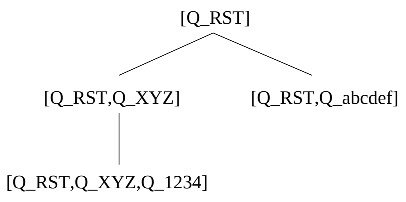

Figure 9: Context Tree Example

Each edge in the context forest represents a correlated, nested query, which provides an optimization opportunity for Scalpel. There are three fundamental approaches to the execution of such nested queries.

Nested Execution: The nested query can be executed in a nested fashion, as it was originally executed by the client program.

Partitioned Execution: Partitioned execution combines all of the executions of the inner query (within a particular outer query) into a single query. A rewritten version of the inner query is submitted once to the server. The results of the rewritten inner query are merged at the client with the results for the outer query. This effectively executes the nested query like a distributed join in which the inner table is moved to the outer table's location and joined there.

Unified Execution: The third approach, called *unified* by Fernández *et al.* [3], combines the inner and outer queries into a single query, the result of which encodes the results of both the outer and inner queries.

In the remainder of this section, we describe these execution strategies in more detail.

6.1 Nested Execution

Figure 10(a) illustrates the nested execution strategy for the application fragment of Figure 8. Each column corresponds to a query, and rectangles are used to denote separately opened and fetched cursors.

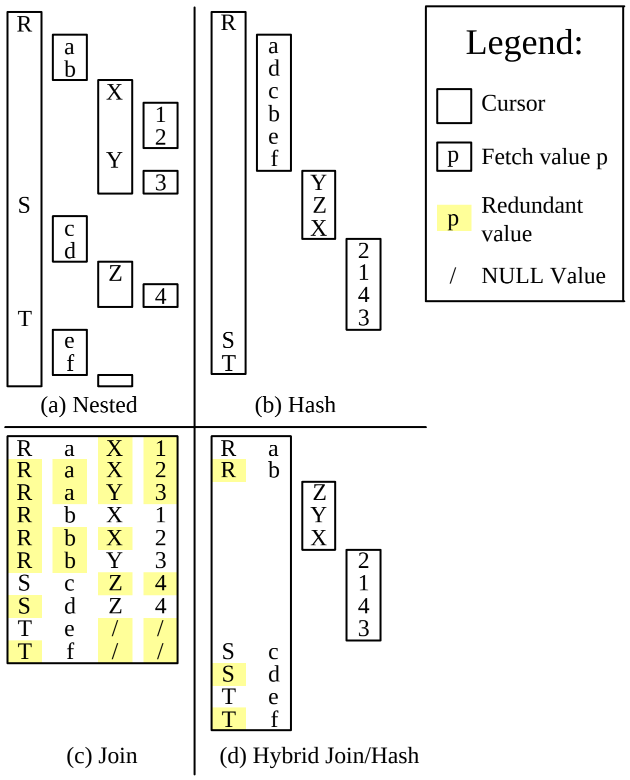

Figure 10: Fetch Traces

Although fixed per-request costs are associated with each invocation of the inner query, the nested execution strategy is appropriate when the selectivity of the inner query is expected to be very low, i.e., when the inner query is often not executed at all. For example, this will be true for the `get_openinvoices` function in Figure 1 if the report currency is usually the same as the default currency.

6.2 Lateral Derived Tables

If the nested execution strategy is used, Scalpel does not rewrite the application's queries. However, the partitioned and unified strategies do require query rewrites. Lateral derived tables provide a convenient way to specify these rewrites.

Lateral derived tables were introduced in SQL/99 [5] to allow nesting in the `FROM` clause of a SQL query. The syntax:

```
FROM  <table reference list>,
      LATERAL (<query expression>) <correlation name>
```

has the following semantics. Let `TRL` be the `<table reference list>`. Let `QE` be the `<query expression>`. The SQL within `QE` can contain references to attributes of `TRL`; these are called *outer references*. Let `TRLR` be the multi-set of rows resulting from `TRL`. Let `QE(r)` represent the multi-set resulting from evaluating `QE` with attributes of `r` supplied as actual parameters to the corresponding outer references.

The result of the `FROM` clause above is the following multi-set:

$$\{|\langle r, q\rangle \mid r \in \texttt{TRLR}, q \in \texttt{QE(r)}|\}$$

where $\langle r, q\rangle$ denotes the formation of a tuple by concatenating row q with row r.

The lateral derived table construct allows Scalpel to generate a single SQL query that directly matches the application semantics that it infers from monitoring the request stream. For example, the query in Figure 11 returns the rows fetched from both the `get_openinvoices` function and the `get_exchangerate` function of Figure 1.

```
SELECT O.id, O.curr, O.transdate, I.exchangerate
FROM ( SELECT id, curr, transdate FROM ar
       WHERE  customer_id=:cust_id AND ar.curr=:currency
         AND  NOT ar.amount=ar.paid ) O,
 LATERAL ( SELECT exchangerate FROM exchangerate
    WHERE  curr=O.curr AND transdate=O.transdate ) I
```

Figure 11: Combined Query for `get_exchangerate`

This query directly expresses the nested execution strategy that was implicit in the application code. Since it is expressed as a single request, it only incurs per-query overhead once. In many cases, is possible to decorrelate the nested query [8] (for example, Figure 2 is a manually decorrelated version of Figure 11). Decorrelation is best performed by the DBMS query optimizer because it can consider the costs of alternative rewrites. By using lateral derived tables, Scalpel allows the DBMS optimizer to select the best execution strategy.

The query in Figure 11 has a shortcoming: rows from the outer query will not be included in the result unless there is at least one row from the inner with matching currency and transaction date. This can be solved by adding an outer join. The `R1` derived table returns a single row; combined with the outer join, lateral derived table `I2` will always return at least one row for each invocation with a row from `O`. In this way, all rows of `O` are preserved.

```
SELECT O.id, O.curr, O.transdate, I2.exchangerate
FROM ( SELECT id, curr, transdate FROM ar
       WHERE  customer_id=:cust_id AND ar.curr=:currency
         AND  NOT ar.amount=ar.paid ) O,
 LATERAL ( SELECT I1.exchangerate FROM
  ( SELECT exchangerate FROM exchangerate
    WHERE  curr=O.curr AND transdate=O.transdate ) I1
   RIGHT OUTER JOIN (SELECT 1 AS x ) R1 ON 1=1 ) I2
```

Figure 12: Combined Query Using Outer Join

The query in Figure 12 is suitable for unified execution strategies: it includes the attributes and rows necessary to answer outer and inner requests. For partitioned execution, only the attributes of the outer query that serve as parameter values are included in the result. Furthermore, each distinct set of inner query parameter values need only be included once in the outer table. Figure 13 shows how a DISTINCT keyword can be used to produce a rewrite that is appropriate for partitioned execution.

```
SELECT O.curr, O.transdate, I2.exchangerate
FROM ( SELECT DISTINCT O1.curr, O1.transdate
       FROM ( SELECT id, curr, transdate FROM ar
       WHERE  customer_id=:cust_id AND ar.curr=:currency
         AND  NOT ar.amount=ar.paid ) O1 ) O,
 LATERAL ( SELECT I1.exchangerate FROM
  ( SELECT exchangerate FROM exchangerate
    WHERE  curr=O.curr AND transdate=O.transdate ) I1
   RIGHT OUTER JOIN (SELECT 1 AS x ) R1 ON 1=1 ) I2
```

Figure 13: Combined Query for Partitioned Execution

6.3 Partitioned Execution

There are many possible partitioned execution strategies for nested queries. Scalpel's optimizer currently considers one such strategy, which we call the *client hash join* strategy. Under this strategy, the inner query is combined with the outer using a lateral derived table like the one shown in Figure 13. This gives a single statement that retrieves *all* of the desired rows from the inner query for all possible outer rows. The first time that the inner query is executed by the application, the combined query is submitted instead to the server. All result rows are fetched and stored in a hash table at the client using the parameters of the inner query from the result set (`O2.curr` and `O2.transdate` in Figure 13) as the key. When the application opens the inner query, the hash table is consulted to see if the results associated with the inner query's parameter values have been cached. If so, the hash table is used to answer the inner query without sending a request to the server. If the result is not stored in the hash table, then the inner query is sent to the server unmodified. When the outer query is closed, the hash table is discarded.

Figure 10(b) illustrates the client hash join strategy. While the nested strategy opens 10 cursors, the partitioned client hash join strategy only opens 4 cursors. Further, the number of opened cursors in the partitioned execution strategy does not depend on the number of rows returned from outer queries. However, this strategy does require sufficient memory at the client to hold the hash table. Further, the CPU of the client machine may make the hash lookups slower than the original, nested strategy.

Although the client hash join strategy is the only partitioned strategy that is considered by Scalpel's optimizer, it would be relatively simple to consider other partitioned strategies within Scalpel's optimization framework. One example is the *client merge join* strategy, which does not require a hash table at the client. Under this approach, the combined query is ordered first by the ordering attributes of the outer query then by the ordering columns requested in the original inner query. This ordering ensures that all of the inner rows for a given outer row are partitioned together, relatively ordered by the ORDER BY of the original query. When an inner query is opened, the combined query's cursor is advanced until either the matching result rows are found, or the rows can not match because the current row is higher than the current ordering attributes of the outer context. In this latter case, a miss has occurred and the original inner query is submitted to the server unmodified.

6.4 Unified Execution

Under the partitioned strategy, the rewritten, combined query is issued when the application first opens the original *inner* query of a query/context pair. In contrast, under the unified execution strategy the combined query is issued when the application first opens the original *outer* query. In addition, the unified strategy may combine more than two queries from a single context tree (Figure 9) into a single query. When the Rewriter component observes the outermost context, it submits instead the combined query to return all of the desired outer and inner rows. The Scalpel system uses the cursor opened over the combined query to respond to the application's requests to fetch rows from the original queries that were combined.

As was the case with the partitioned strategies, many unified strategies are possible. Scalpel's optimizer currently considers one representative unified strategy in which the combined query is a join of the inner query with its context. Figure 10(c) illustrates the result of using this strategy to combine all of the queries from Figure 8. In addition, Figure 10(d) illustrates a hybrid strategy, in which the unified approach has been used to combine application queries `Q_RST` and `Q_abcdef`, while the partitioned approach is used for queries `Q_XYZ` and `Q_1234`.

Scalpel can form a unified join query corresponding to any connected subset of nodes from the context graph. Queries are joined using a lateral derived table expression similar to the one shown in Figure 13, except that additional `ORDER BY` clauses are added to control the order in which the query result is delivered to the client site. Scalpel determines the required ordering clauses by making a depth-first traversal of the relevant portion of the context graph. For each node, the original `ORDER BY` clause for that node's query is appended to the `ORDER BY` clause of the combined query, followed by a key for that query. This ensures that ordering of the result of the combined query corresponds to the nesting of the original queries, i.e., the inner query rows corresponding to each outer query row are grouped together.

When the application performs FETCH on one of the queries that has been combined by Scalpel to form the join query, Scalpel must extract the proper values for the application's FETCH from the result of the join query. Each of the original application queries is associated with a set of key columns in the combined query's result. If the next row in the join query result has the same values as the current row for the fetched query's key columns, then the next row is a duplicate which must be skipped. For example, in Figure 10(c), this occurs with the $(\mathrm{R}, \mathrm{a}, \mathrm{X}, 1)$ and $(\mathrm{R}, \mathrm{a}, \mathrm{X}, 2)$ tuples when satisfying a fetch for Q_{abcdef}. Both of these tuples have the same key-column values ('a'), so the second of these tuples will be skipped. Furthermore, if the parent query's key column values for the next row differ from those in the current row, that indicates the end of a logical result set. For example, in Figure 10(c), the tuple $(\mathrm{S}, \mathrm{c}, \mathrm{Z}, 4)$ indicates the end of the first logical result set for Q_{abcdef}. In Figure 10(c), the shaded parts of the join query result show the parts of the result that will not be returned to the application when it performs FETCH operations on its open cursors.

In general, it is possible that Scalpel would have to fetch backwards on the combined query's cursor in order to provide the correct return value for an application's FETCH. In the example from Figure 10(c), after returning EOF for Q_{abcdef} because the tuple $(\mathrm{S}, \mathrm{c}, \mathrm{Z}, 4)$ signaled the end of the logical result-set, the run-time fetches backwards on the combined cursor to the point $(\mathrm{R}, \mathrm{b}, \mathrm{X}, 1)$ in order to be positioned at the appropriate point for query Q_{XYZ}. This would require a scrollable cursor for the combined query. Where supported, scrollable cursors are often more expensive than forward-only cursors. Furthermore, fetching backward may reintroduce some of the the per-request latency that the unified strategy is designed to avoid, since fetching backward may require that rows be re-fetched from the server. For this reason, Scalpel avoids using the join strategy if scrollable cursors would be required. Backwards moves do not arise if the original application queries return at most one row each time they are opened, e.g., in case of a fetch based on a primary key. Scalpel's optimizer will not consider the join execution strategy if more than one of the original queries might return more than a single row when it is opened. This is sufficient to avoid backwards moves of the join query cursor.

The join-based unified execution strategy can significantly reduce per-query overhead as compared to the nested execution strategy. It also increases the scope of the server's optimizer by exposing join operations to it. A drawback of this strategy is that the join query result includes redundant data, as can be seen from the example in Figure 8(c). This led Shanmugasundaram *et al.* [9] to call this approach *redundant relations*. As was the case with the partitioned strategies, there are many other unified strategies that Scalpel currently does not consider. For example, instead of ordering the join result, an unordered join query result could be read into a hash table at the client. Another possibility is to reduce the redundancy in the unified query result by unifying with outer unions rather than joins, as suggested by Shanmugasundaram *et al.* [9] and Fernández *et al.* [3]. Again, although Scalpel's optimizer does not currently consider these options, it would not be difficult to make it do so.

7. OPTIMIZATION

After the training phase has found a forest of candidate query/context pairs, an optimization step is used to determine which of the execution strategies described in Section 6 will be used. The Pattern Optimizer component is divided into two modules: a plan generation module that generates candidate execution plans, and a ranking module that assigns a relative ordering to the generated plans. The generated plan with the best rank is selected for execution at run-time.

7.1 Plan Generation

The plan generation module works with one tree of contexts at a time. An optimal plan is selected by exhaustively enumerating all possible execution strategies, then selecting the one with the best ranking. More complicated enumeration schemes can produce an optimal plan without exhaustively enumerating all possibilities; for simplicity, we present only an exhaustive process in this paper.

Consider a node $[C, Q]$ that appears in the context tree (e.g., Figure 9). Each rewrite candidate (C, Q) is associated with a context `[C,Q]` that appears as a node in the tree. When the application executes OPEN(Q) while in context C, Scalpel can execute Q using either a nested (N), joined (J), or hash-based (H) execution strategy. For the joined strategy, we distinguish the query at the root of the joined subtree (JR) from the remaining queries to be joined (JD). A plan is described by a context tree in which each node is annotated with N, H, JR, or JD. For example, Figure 14 shows four plans for the context tree example of Figure 9. These strategies correspond to the execution traces shown in Figure 10.

With 4 possible annotations per node, there are up to 4^n possible execution strategies for a tree with n contexts. Not all of these 4^n strategies are legal. A node can be marked as JD only if its parent is also marked JD or if its parent is the join root JR. A further restriction limits the queries that can be combined in a joined strategy. Recall that we allow only one query in a joined strategy to return more than one row per invocation. We rely on a support function AT-MOST-ONE(Q) provided by the RDBMS to indicate

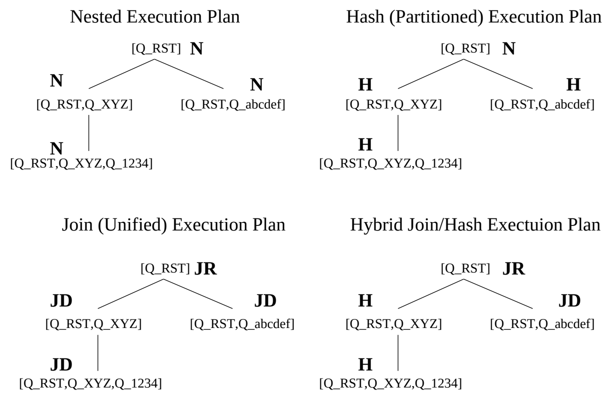

Figure 14: Examples of Generated Strategies

whether a query Q returns at most one row. This function detects when the database schema implies that only one row can be returned (for example, due to primary key lookup). A join root Q_{JR} can only have one descendant query Q_{JD} where $\textsc{At-Most-One}(Q_{\mathrm{JD}})$ is not true.

The enumeration algorithm iterates through all of the plans that are permitted by the above rules. For each plan, the algorithm associates rewritten queries with the H and JR nodes. For a node C that is annotated with H, an alternate query Q' is built by combining the original query Q with the query from the context's parent using the rules for combining queries described in Section 6. The enumeration algorithm combines the queries of the root and all descendants using joins as described in Section 6. This combined query is assigned to the JR node.

7.2 Costs and Ranks

We rank execution costs using the response time (in seconds) experienced by the client application. This latency can be estimated using the sum of estimated server, communication, and client latencies. Clearly, other ranking functions can be used instead; for example, we could choose to rank based only on server execution costs, or we could attempt to create a more precise model of latency by estimating the amount of overlap for server, network, and client processing costs. In this paper, we concentrate only on ranking by total latency.

Scalpel's optimizer relies on a combination of measured quantities and support routines provided by the RDBMS in order to estimate the cost of execution strategies. Table 1 summarizes the various quantities used for ranking execution strategies.

The function $\textsc{Server-Cost}(Q)$ gives the server cost of a query as estimated by the RDBMS. The RDBMS also provides $|Q|$, an estimate of the cardinality of Q, and $\textsc{Bytes}(Q)$, the estimated average row size (in bytes).

In addition to estimates from the RDBMS, Scalpel itself estimates some parameters that are not usually estimated by the RDBMS. $\textsc{Comm-Cost}(N, B)$ estimates the network latency for communicating N rows of B bytes. Scalpel determines this estimate based on measurements with the network configuration of interest. The overhead associated with a single $\textsc{Open}$ request is also measured and represented by U_0. Finally, Scalpel uses the client predicate selectivity estimates described in Section 5.2.1. We use $P_0(C)$ and $P_1(C)$ to denote the selectivities associated with context C, and we define $P(C) = P_0(C)P_1(C)$.

Overall, the cost of opening a query and fetching all of the rows is given by the following:

$$\begin{aligned}\textsc{Cost}(Q) = \;& U_0 + \textsc{Server-Cost}(Q)\\ & + \textsc{Comm-Cost}(|Q|, \textsc{Bytes}(Q))\end{aligned}$$

In addition to server and communication estimates, the optimizer estimates the cost of client operations associated with Scalpel's rewrites. H_{add} is an estimate of the cost of adding one row to a hash table, and H_{find} estimates the cost of finding a result set in a hash table.

Table 1: Estimated Quantities

Quantity	Source	Description
$\textsc{Server-Cost}(Q)$	RDBMS	Server costs for Q in seconds
$\lvert Q \rvert$	RDBMS	Rows returned by Q
$\textsc{Bytes}(Q)$	RDBMS	Average row length for Q
$\textsc{Comm-Cost}(N, B)$	Scalpel	Communication latency for N rows of B bytes
U_0	Scalpel	Overhead of a single request
H_{add}	Scalpel	Cost of adding to hash table
H_{find}	Scalpel	Cost of finding in hash table
P_0, P_1	Scalpel	Selectivity of client predicates

7.2.1 Ranking Strategies

For each strategy given by the plan generation component, the optimizer estimates the response time for the tree of contexts. For each context $C_1 = [C_0, Q]$, we estimate how many times Q is opened in context C_0 using the $\textsc{Opens}(C)$ recursive function:

$$\textsc{Opens}(C) = \begin{cases} 1 & : \quad C = [Q] \\ P(C)\,|Q_0|\,\textsc{Opens}(C_0) & : \quad C = [C_0, Q_0] \end{cases}$$

For a context C annotated with either nested (N) or joined root (JR), we estimate the cost as follows:

$$\textsc{Context-Cost}(C) = \textsc{Opens}(C) \times \textsc{Cost}(Q')$$

The Q' is the query opened at run-time; for an N context, this is the original query; for a JR context, it is the combined join query. This formula estimates the cost of opening the query and fetching all of the rows. The join descendant contexts (JD) therefore have a cost estimate of zero, because the fetch cost is estimated at the join root.

A context $C = (C_0, Q)$ with an H annotation will open its alternate query Q' the first time Q is opened in the parent context. If the Q is subsequently opened before the cursor associated with the parent context has been closed, the hash table is consulted to find prefetched results for the requested parameters. We estimate the cost of an H context as follows:

$$\begin{aligned}&\textsc{Context-Cost}(C) = \\ &P_0(C)\textsc{Opens}(C_0)\,[\textsc{Cost}(Q') + |Q'|H_{\mathrm{add}}] + H_{\mathrm{find}}\textsc{Opens}(C)\end{aligned}$$

The cost for a tree of contexts is given by the sum of the node costs. The strategy with lowest estimated cost is saved for use at run-time.

8. EXPERIMENTS

The costs associated with execution strategies depend on a number of factors described in the earlier sections. This section presents experiments that give a sense of how these factors combine to affect system performance. Table 2 shows the computers used in the experiments, and Table 3 shows the configurations of these computers. We ran our tests with two commercial DBMS products, S1 and S2. The license agreements prevent us from identifying them. All tests were run with JDK 1.4.2 and JDBC drivers provided by the DBMS vendors. The database instance was fully cached. A prototype implementation of Scalpel was used for the experiments; combined queries were manually de-correlated because the lateral derived table construct was not supported in one of the database servers.

Table 2: Available Computers

Computer	Processor	O/S
A	1.8 GHz Pentium 4	Windows XP
B	$2 \times$ 2.2GHz Pentium XEON	Win2K Server
C	$2 \times$ 500MHz Pentium 2	Win2K Server

Table 3: Tested Configurations

Config	Client	Connection	Server
LCL	A	Local shared memory	A
LAN	A	100Mbps LAN	B
WiFi	A	11Mbps 802.11b WiFi	B
WAN	A	1Mbps Cable modem + WAN	C

8.1 Effects of Client Predicate Selectivity

Scalpel uses two parameters, P_0 and P_1, to model the selectivity of client predicates, as discussed in Section 5.2.1. Figure 15 shows the run-time of the various execution alternatively as a function of the selectivity of the client predicates. The independent parameter is $P = P_0 P_1$, the product of the two selectivity parameters. Highly selective client predicates correspond to low values of P. Results are shown for an outer query Q_1 returning 1000 rows, and an inner query Q_2 returning 4 rows for each outer row.

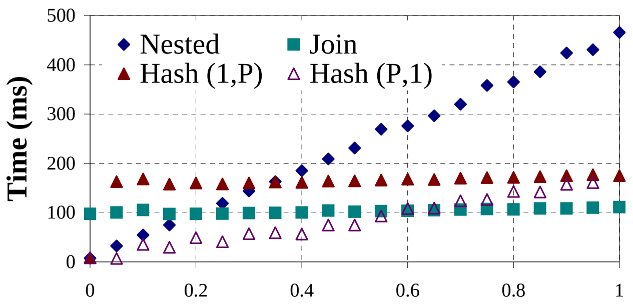

Figure 15: Run-Times for Selectivity $P = P_0 P_1$ (Configuration LCL)

For the original nested execution strategy, execution time is proportional to the product P. The outer query is always executed one time, and the inner query is executed an average of $|Q_1|P$ times. For the joined execution strategy, the execution time is largely independent of the selectivities of P_0 and P_1. Regardless of the results of these predicates, the join query is executed and all rows are fetched by the client. When P is small, nested execution is the more efficient strategy, as expected.

The hash execution strategy fetches all possible rows from the inner query when the first inner row is fetched (P_0 is true). This gives the hash strategy a strong dependence on the P_0 selectivity. The P_1 selectivity also has a small effect, as it changes the number of lookups performed in the hash table. Figure 15 shows two series for hash strategies: one with $(P_0, P_1) = (1, P)$, and the other with $(P_0, P_1) = (P, 1)$.

8.2 Execution Costs

Figure 16 shows the CPU costs and total time for each of the strategies. Results are shown for an outer query Q_1 returning 1000 rows, and an inner query Q_2 returning 4 rows for each outer row. The product $P = P_0 P_1$ was set to selectivity of 0.15, and the hash-based variant was executed with both $(P, 1)$ and $(1, P)$ selectivity.

Client and server CPU costs were measured using O/S functions and DBMS-specific requests respectively. The difference between total execution time and the measured CPU costs is labeled *latency*. Results for both database servers were measured; as they are consistent, only one set of results is presented here.

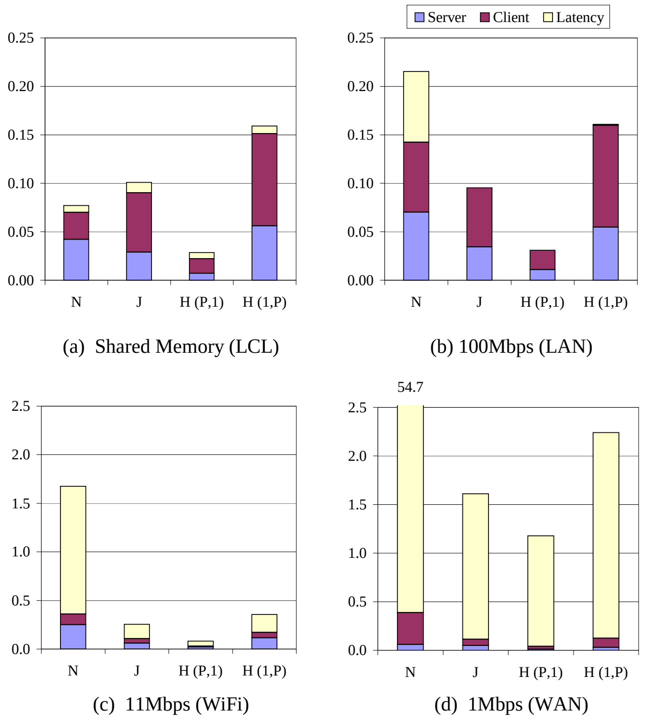

Figure 16: Run Time (s) for Net Configurations ($P = 0.15$)

There are several interesting observations that we can draw from Figure 16. First, the original nested execution strategy outperforms the joined and hash variants in the LCL configuration with $(1, P)$ predicates; however, the joined strategy has lower server costs. The joined and hash-based strategies are *faster* in the LAN configuration than in the LCL configuration because the LAN configuration allows for overlap between server processing and client processing. The nested strategy, on the other hand, takes more than twice as long in the LAN configuration because of the increased network latency. Second, the hash and join execution strategies reduce not only latency but also the client and server

CPU costs. This cost savings results from fewer messages that need to be formatted, sent, and interpreted. Finally, while the savings for the LAN configuration are low in absolute terms (about 125ms), the savings are very significant for WiFi and WAN; for example, with a wireless configuration the joined strategy saves nearly 1.5s of elapsed time, and the savings grows to 54s with the WAN setup.

The original nested execution strategy is optimal for a local connection and selectivity of $(P_0, P_1) = (1, 0.15)$; even if a LAN configuration is considered, absolute benefits are moderately low so system developers might decide to use the simpler nested implementation. However, the server costs are 50% lower with the joined variant. Further, if other deployments with higher network latency are used in the future, the nested implementation will be unsatisfactory.

8.3 Scalpel Overhead

The Scalpel system monitors all database requests during training. The correlation detection algorithm we have presented is $O(n^2)$ in n the number of attributes for all opened cursors. A slightly more complicated $O(n)$ algorithm exists. However, we have found that even the quadratic approach has reasonable overhead for the actual systems we examined. Table 4 shows the open-time for a query with varying numbers of outer queries opened (Depth), varying number of columns for each query (Cols), and varying number of parameters for the innermost query (Parms). Testing was performed with a local client (LCL); results were consistent for both DBMS systems tested (S1 and S2), so only one set of results are shown.

Table 4: Training Overhead

Depth	Cols	Parms	Original (ms)	Training (ms)
1	10	1	2.64	3.15
10	10	1	2.62	3.31
10	10	10	2.88	3.34
40	100	100	14.95	198,368.17

The overhead of the correlation detection algorithm is reasonable so long as the product of Depth, Cols, and Parms is not greater than 1000. In the actual systems we examined, we did not find any cases where this product exceeded 100. However, the poor scalability of the quadratic algorithm indicates that a linear algorithm may be preferable in a practical implementation.

At run-time, all Open requests are intercepted. If a query is not being re-written, the original query is submitted to the DBMS and the result set is wrapped in a monitor object that can detect when the cursor is closed. This monitoring is needed to maintain the current context for triggering rewrites.

We measured the overhead Scalpel adds to opening a query with S1. With a local configuration (LCL), the overhead is 27μs, or 7.0%. The overhead is the same for all network setups, and drops to 1.6% in the LAN configuration due to the higher base cost.

The overhead would be reduced if Scalpel were integrated with the vendor-provided JDBC driver (for example, eliminating the need to use wrapper objects to track the current context). However, the overhead seems acceptable as it is low in absolute terms, and consists entirely of client CPU time. Even with current overhead levels, significant gains can be made without significantly impacting user interaction due to the benefits of rewrites performed by the Scalpel system.

9. CONCURRENT UPDATES

The Scalpel system prefetches results before the associated cursor is opened. This can lead to a data consistency problem where prefetched data does not contain updates that would have been observed if the data were not prefetched. There are two possibilities for these updates: either they are performed by other connections or the current connection.

Updates performed by the current connection are monitored by the Scalpel system. If an update could possibly affect prefetched results, the associated prefetched data is released and further requests are submitted directly to the DBMS. If this situation is detected at training time, it is recorded in order to avoid rewrites that might be susceptible to this problem.

While Scalpel can detect changes from the monitored connection, it can not observe changes made by other transactions after data has been prefetched. If the *serializable* isolation level is used, then this does not introduce a correctness problem. The serializability requirement means that subsequent fetches would observe the same results as a prefetched result. However, depending on the implementation of the serializable isolation, prefetching can cause extra data contention because Scalpel prefetches some data that would not have been fetched by the original application (because a local P_0 or P_1 predicate failed). Locking-based implementations will prevent updates to these unneeded rows until the transaction ends. On the other hand, when using rewriting strategies the Scalpel system significantly reduces the run-time of a transaction, which can act to reduce data contention because locks are held for a shorter period.

When an isolation level weaker than serializable is used, prefetching may introduce anomalies. These anomalies are likely acceptable to application developers that accept the anomalies associated with weak isolation, but they need to be carefully considered.

10. RELATED WORK

A number of researchers have studied how to effectively execute queries that contain various forms of nesting [2, 8]. The approaches developed in that work are effective at choosing efficient evaluation plans for the correlated combined queries that we generate. However, the techniques are not directly applicable to the problem we consider because the nesting appears in the application, not the queries.

Shanmugasundaram *et al.* [9] and Fernández *et al.* [3] studied efficient mechanisms to generate nested results from relational data sources. Their work differs from the current research in three ways. First, they assumed that the query nesting was known due to the presence of an XML view definition, while we detect nesting in a client application. Second, our work extends the *view tree reduction* of Fernández *et al.*, a heuristic that combines all at-most-one-row queries with their outer query using joins. In our work, we choose the queries to join together on the basis of cost. Further, we permit up to one query that returns multiple rows to be included in the join provided that the savings outweighs the associated data redundancy. Finally, we consider the effects of local predicates on execution strategies.

Mayr and Seshadri [6] also considered effective mechanisms to execute queries that contain local predicates. However, they did not consider the case of nested queries, and assumed that the local predicate was identified in the query. The rewrites they employ can be applied to our case, but some are difficult unless we also rewrite the client application.

Florescu *et al.* [4] discussed *query simplification under preconditions*, an approach to optimizing parameterized queries based on knowledge of the source of the binding parameters. In their work, they considered queries that were accessed with actual parameter values drawn from different outer queries, and exploited rewrite optimizations that were safe in certain contexts. Their work relied on having a declarative representation of the queries and nesting, so they could be assured of knowing when a query was truly correlated.

Bernstein *et al.*[1] suggested that the *context* in which an object was fetched is an important factor to consider when deciding which related objects should be prefetched. For example, if two objects, A and B were both fetched from the same query, then we fetch A's x attribute, it is a reasonable assumption that we will soon fetch B's x attribute. The solution proposed by the authors attempts to discover operations that should be applied to multiple objects, and uses this as the basis for prefetching. In contrast to Scalpel, their system examined contexts of objects not of queries.

Sellis investigated mechanisms for choosing an execution strategy for a stream of queries that are known ahead of time [7]. Results for multi-query optimization complement our results well. Scalpel derives a prediction of queries that will be executed together, and these could be optimized by a query optimizer using multi-query optimization.

11. CONCLUSIONS AND FUTURE WORK

A number of client applications generate patterns of nested, correlated queries in the stream of requests they submit to a DBMS. This nesting pattern generates a large number of small queries that contribute to high latency and server processing costs. While it is possible in some cases to rewrite applications manually to avoid generating these patterns, such rewrites are complicated by a number of factors. One of these is the fact that a nested approach is in fact optimal for some configurations of the application program (for example, see Figure 15). In the cases where such optimal configurations are expected to occur in the majority of client deployments, it is prudent to choose the nested implementation, which is in any case easier to implement.

Although optimal in some configurations, the nested strategy can lead to poor execution performance for deployments that do not match the expected configuration. Developers are thus faced with an unfortunate decision: either use a more complicated implementation that is actually slower in the majority of predicted deployments, duplicate development effort to use multiple implementations that choose the optimal strategy at deployment time, or accept sub-optimal performance in deployments that do not match the expected case.

We have presented Scalpel, a system for detecting and optimizing patterns of repeated requests within a query stream. Scalpel uses a deployment-time training period that monitors a request stream to automatically detect nesting of queries and correlations between query parameters and attributes of outer queries.

Once Scalpel detects nested requests and finds predicted correlation sources for parameters of inner queries, it uses a cost-based optimizer to choose the optimal execution strategy for the particular deployed configuration. The cost-based optimizer minimizes response time by estimating the selectivity of two types of local predicates (P_0 and P_1) and by comparing the server, communication, and client costs of three execution strategies: nested, joined, or hash-based.

The execution strategies that we have presented give correct results even in the case that the training phase predicted correlations that do not hold at run-time. Further, these strategies give correct results in the face of concurrent updates provided that serializable transactions are used. If weaker isolation levels are used, the presented strategies may introduce anomalies that may be acceptable to application developers, although they must be carefully considered.

Rewriting nested patterns is only a first step toward optimizing the query streams applications present to servers: data structure correlations and batches also occur frequently. Further, additional strategies such as outer unions and merge-based rewrites may perform better in some configurations. Finally, the patterns detected by Scalpel may be useful for tuning a database servers. For example, it may provide insight into appropriate materialized views that can be used to answer multiple requests in a stream. Alternatively, knowledge of requests likely to occur in the future may aid the DBMS query optimizer to select an appropriate execution strategy using techniques such as multi-query optimization [7]. We plan to address these issues in future work.

12. ACKNOWLEDGMENTS

Our thanks to the anonymous reviewers for their helpful suggestions. This research was supported in part by iAnywhere Solutions and by the National Sciences and Engineering Research Council of Canada.

13. REFERENCES

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

[2] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *TODS*, 25(4), 2000.

[3] M. F. Fernández, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *SIGMOD*, 2001.

[4] D. Florescu, A. Y. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web-sites. In *VLDB*, pages 627–638, 1999.

[5] International Standards Organization. Database language SQL—Part 2: Foundation (SQL / Foundation). ISO/IEC 9075-2:1999, Sept. 1999.

[6] T. Mayr and P. Seshadri. Client-site query extensions. In *SIGMOD*, 1999.

[7] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

[8] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *ICDE*, Feb. 1996.

[9] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB Journal*, 10(2–3), 2001.