

Holistic Approaches for Robustness and Optimization of database applications

Seminar Report

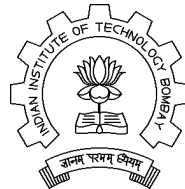
By

Karthik S. Ramachandra

Roll No : 09405002

Advisor

Prof. S. Sudarshan



Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai
November 2009

Abstract

Traditional query optimization and compiler optimization techniques have evolved independently over a long time thanks to the extensive research that has been happening in these areas. Though there are still many hard unsolved problems in those areas, we can safely claim that the techniques that have evolved are mature and robust. However, in practice, we see that there is a large class of applications that fall under the category of ‘database applications’ i.e., the applications which operate and maintain the data using a relational database. Now, the optimization of such applications is typically done using the above independent techniques. Clearly, these techniques do not guarantee the global optimal execution of the application. Finding such a global optimum requires a holistic viewpoint of the database application as a single entity. Achieving this goal involves the hard task of bringing together two independent areas of research under one roof. We present a survey of literature in this area and delve into some of the details of promising leads. We then conclude by proposing some directions for future work in this area.

1 Introduction

Relational databases are the standard storage choice for a lot of applications. Applications written in various programming languages access data from these relational databases primarily using SQL. The typical pattern of data access used is to dynamically construct and manipulate SQL queries using the application programming language (like Java/C# etc). Then the queries are dispatched to the database for execution. Subsequently, the results of the query from the database are mapped back to objects of the programming language for further usage.

The nature of such interactions between an application and the database get more and more complicated as the scale of the data increases and as the complexity of applications increase. This can lead to a large number of correctness, security and performance issues which can go unnoticed during development time. Traditionally, a lot of the attempts that have been made to identify such issues have been in two directions: (a) They focus on the DBMS - involving query optimizations, caching, validations etc., or (b) They focus on the application - involving source code optimizations, data caching, data authorization, etc.

From an application design and engineering viewpoint, maintaining this encapsulation between the data management activities and usage of the data is a good practice. But this doesn’t imply the optimal in terms of the issues identified above. So in order to overcome them and also retain the design, there have to be approaches which consider the application and the database as a unit. In essence, this is what we mean by a *holistic approach*. There has been some research which approach this problem in a holistic way, and try to reduce the barrier between the database and the application.

In this report, we do a survey of various such holistic approaches and try to understand the current state of research in this area. We also point out the limitations of the existing approaches and scope for future work. In this survey, we limit our study to techniques which work statically i.e. on the source code/byte code level.

2 Motivation for holistic approaches

Dasgupta et al.[4], Gould et al.[8] Garrod et al. [6] provide various examples which motivate holistic approaches. A simple yet very common example is that of unused data. Consider the C# code

```

cmd.CommandText = "SELECT sku, price, description FROM products";
SqlDataReader rdr = cmd.ExecuteReader();
while(rdr.read()){
    sku = rdr[0];
    price = rdr[1];
}

```

Figure 1: Code snippet showing unused data from a query[4]

```

ResultSet getTypeNames(String code) {
    String query = "SELECT type_name FROM TYPES WHERE type_code = " + code;
    return statement.executeQuery(query);
}

```

Figure 2: Code snippet showing possibly wrong data type mapping

snippet[4] shown in Figure 1.

Observe that only two of the three projection columns are used by the application code, though the query actually retrieves three columns. In fact, it is not very uncommon to encounter queries of the form “SELECT * FROM ...” where they are not really needed. As the schema evolves, these queries might end up being both memory as well as a performance bottleneck. Also, changes to the underlying table structure at a later point in time will break the application if the columns required are not enumerated in the project clause of a query.

Type checking is another area where a lot of runtime errors occur. For example, consider the java method in Figure 2. In this case, the parameter code is declared as a string and the column type_code is of type integer. As long as code is a string representing a number, there are no type errors. But, the Java type system cannot identify the potential risk of the string code containing non numeric characters.

From these examples, it is clear that any kind of

- (a) query validations or optimizations without knowledge of the application context
- (b) application code validation/optimizations without the knowledge of the database interactions

will not be able to identify such issues.

Holistic approaches therefore, exploit the knowledge of both the application program source code as well as the metadata of the database and try to make sure that the interactions between these two systems are optimal, correct, and secure. This is primarily done by applying source code transformations and query transformations which guarantee equivalence with the previous state while fixing the identified issues.

3 Early approaches

Applying program analysis concepts along with knowledge of data access patterns for performance improvements has been suggested as early as 1981 by Abu-Sufah et al.[1]. They consider virtual memory based systems and apply source to source transformations in order to improve data access locality. Though this does not deal with relational databases, they do use both program analysis as well as memory management details in order to optimize performance.

Shopiro[16] argues that “A considerable effort has been expended on the design of interactive query languages for relational databases, but there has been rather less work on programming languages for relational databases...We are particularly interested in **global program optimization**, rather than optimization of individual retrieval requests.” - clearly pointing towards holistic approaches. A high level programming language called Theseus, was built by Shopiro as early as 1979 in order to facilitate research in ‘automatic program optimization’. Theseus is an extension to Euclid, itself and extension to the Pascal programming language. Theseus has facilities for building and manipulating relations i.e. all the relational algebra features. It can express all the operations that are required in database applications. For simple retrievals, SQL queries may be shorter than the corresponding Theseus programs. But for more complex retrievals the Theseus programs are shorter and clearer. The main claim they make is that, in the usual database applications, there is a lack of communication between the SQL compiler and the programming language compiler. And this lack of communication inhibits many of the optimizations which can easily be done in Theseus.

4 Approaches for Robustness

We now analyze the various approaches that improve the robustness of a database application. The following aspects of the application are covered:

1. Syntactic checking of the database interaction (SQL)
2. Type checking and type casting
3. Identifying security vulnerabilities
4. Identifying data integrity violations

The prerequisite for any of the above tasks would be to first analyze the application source code/binary and the database in order to collect all the necessary information. This analysis turns out to be a common infrastructure for any such holistic approach and forms an important part of the same. Hence we cover this aspect in Section 6. In this section, we assume that we have the necessary information available and try to design the above robustness aspects.

4.1 Syntactic checking of the database interaction (SQL)

Gould et al.[8] present a program analysis technique to verify the correctness of dynamically generated query strings. They use Java as the ‘meta’ language - the language used to build and manipulate SQL programs as string data. This technique is implemented in a prototype tool that takes java bytecode as input, and after the analysis, reports any such errors that might occur during runtime. It works on top of a java bytecode string analysis procedure (which we explain in Section 6) to build a conservative

representation of the potential query strings that could be generated. This representation is a finite state automaton. This string analysis also performs syntax checking while building the automaton. So any syntax errors are reported at this stage. This string analysis differs from the standard SQL lexical analyzer, which analyzes a single query. This procedure statically analyzes the potential query set. But a limitation of this approach, as Dasgupta et al.[4] point out, is that this cannot guarantee that all possible SQL that can be executed by the application will be extracted. Refer to Section 6 for details.

4.2 Type checking and type casting

One of the main advantages of statically typed languages over dynamically typed languages is that type checking happens at compile time and lot of errors are caught early. But since SQL queries are just treated as strings by the host language (Java/C#), this type checking doesn't happen and is a common cause of runtime errors. Figure 2 shows one such scenario. Another such potential error is shown in Figure 3.

```
SELECT (discount * 100) || '%' as disc_percent FROM products WHERE price > 1000;
```

Figure 3: Query showing implicit type casts

The expression $(discount * 100) || \text{'\%'}$ concatenates a character with a numerical value. This assumes that the database implicitly type-casts the numeric result into a string before concatenation. On some databases this results in a runtime error.

In order to identify such cases, Gould et al.[8] propose a detailed approach based on a combination of automata-theoretic techniques and a variant of the context free language (CFL) reachability problem. The first step is to build an automaton by analyzing strings in the java bytecode. Then they perform a non trivial operation which they call as reconstruction of type environments. Once the type environments are constructed, type checking can be performed using the SQL type system grammar. This procedure is explained in detail in Section 6 as it forms a common preprocessing step for other approaches as well.

The other useful case is to highlight wrong usage of the query results in the application. Gould et al.[8] mentions this as a future enhancement to their work. An example of the same is shown in Figure 4. The first attribute in the result set is the name which is a string, but the code as shown tries to use it as an integer. Similarly with the salary attribute. These wrong usages can be reported during compile time.

A related error is also regarding the number of columns retrieved vs. used. Dasgupta et al. [4] enumerates the information that needs to be extracted to identify such errors and more: (1) The SQL string itself (wherever possible) (2) Number and database types of columns in the result of the SQL (applies only for SELECT statements) (3) Tables and columns referenced in the statement. They also add a fourth point (4) Optimizer estimated cost of the statement. We will look at it when we discuss performance and future work. Another bit of useful information would be (5) The parameters of the SQL query and the program variable/expression which is bound to those parameters, along with their data type in the application. The assumption here is that this analysis process has access to the database schema.

```

query = "SELECT name, salary FROM employee where id = 5";
rset = stmt.executeQuery(query);
...
salary = rset.getInt(1); // should be getInt(2)
name = rset.getString(2); // should be getString(1)
...

```

Figure 4: Code snippet showing wrong usage of query results

4.3 Identifying Security vulnerabilities

Given an application binary, Dasgupta et al.[4] suggests a way to report a potential SQL injection attack. A SQL injection attack occurs when user input is used to build a query string and execute it without validation. An attacker can inject malicious SQL code in the user input that gets executed on the database causing various kinds of damage.

In order to identify such a vulnerability, the important piece of information necessary is the set of points in the program where user input is used to build queries. To begin with, every function that supplies user input into the program is marked UNSAFE. For example, consider a HTTP form submission which has a text field. The submitted text appears in the HTTP Servlet as a Map of name and values. Suppose the text field is named 'description'. then, the following value is marked UNSAFE:

```
String desc = (String)request.getParameter("description");
```

If this variable goes through some functions, it might be that there is some validation that is happening. In this case, it is marked as MAYBE UNSAFE. This "safety" attribute is propagated through the data flow analysis from the source (the above statement) till it reaches the sink (the place where the SQL is executed). At this point, the resulting SQL is marked UNSAFE. If the string is added to the SQL query using the API functions PreparedStatement and setInt, setFloat etc, then that string is marked SAFE.

```

PreparedStatement updateSales = con.prepareStatement("UPDATE coffees
SET sales = ? WHERE cof_desc LIKE ?");
...
updateSales.setString(2, desc);

```

Finally, all the UNSAFE and MAYBE UNSAFE queries are reported as vulnerable.

4.4 Identifying data integrity violations

Various integrity constraints are enforced in the database. Not null constraints are one of them. Using the query strings and their parameters, in some cases it is possible to identify if running that query would violate this constraint. This applies to mainly INSERT and UPDATE queries. Also all cases where the constraints might be violated cannot be found as they depend on the data which might be entered by the user. For example, in Figure 5, if we know that there is a not null constraint on the date column, this can be highlighted as an integrity violation.

```

String product = getProduct();
Date date = null;
...//statements that do not modify date

PreparedStatement insertStock = con.prepareStatement(
"INSERT INTO inventory(product, date, qty) VALUES(?, ?, ?)");
insertStock.setString(1, product);
insertStock.setDate(2, date);
insertStock.setInt(3, 100);

```

Figure 5: Integrity constraint violation

There is another interesting application related to identifying data integrity violations shown by Dasgupta et al.[4]. Since constraint checking in the database can be expensive sometimes, the developer might want to enforce a constraint in the application code. They take an example where they want to ensure that the price column of the Products table always has a value > 0 . The input is a constraint of the form $[\text{Products}].[\text{price}] > 0$. Given such a constraint as input, the goal is to identify all the statements in the application where the price column is being inserted or updated. Once such statements are found, an assertion like *Assert(price > 0)*; is added before these statements in the code which validates the given constraint. This could be a program transformation which operates on the binary. Achieving this involves (a) Identifying the DML statements that affect the price column in the application code, (b) Identify the program variable/expression that is bound to the the price column in the statement.

5 Approaches for Optimization

Performance optimization of applications has been a long standing topic which has interested many people over a long period of time. Lots of ideas have been tried and are being tried and adopted. As we discussed earlier, there have been approaches which argue that a typical database application design enables a more holistic analysis that maintains the relationship between the database and application data.

5.1 Merging related queries

Liewen and DeWitt[12]’s work has covered a lot of ground in the area of query merging. They address the problem of transforming loops to joins which we demonstrate below. They use languages like PASCAL/R, O2, E, and O++ in their paper. Manjhi et al.[13] explain the merging transformation using the php code snippet shown in Figure 6 and Figure 7.

The above example is merging the loop into one join query. This way of optimizing is a known way, but here the important point is that this is being automated. i.e., patterns like Figure 6 are transformed automatically. The tricky part is of course to recognize the pattern. In Figure 6, suppose in between the first query and the loop, there is an update to the database. Then it might not be possible to merge these queries. Two code patterns are suggested where such transformations can be automated:

Loop-to-join: This is the example shown above. The application issues one query to get multiple values, and then executes another database query for each value (Figure 6). This can be transformed

```

$template := SELECT from_user_id FROM comments WHERE to_user_id = ?;
$query := set_params ($template, $to_id);
$result := execute ($query);
foreach($row in $result) {
    $from_id := get_user_id($row);
    $template := SELECT user_name FROM users WHERE user_id = ?;
    $query := set_params($template, $from_id);
    $result2 := execute($query);
}

```

Figure 6: Original Code

```

$template := SELECT from_user_id, user_name FROM comments, users
              WHERE from_user_id = user_id AND to_user_id = ?;
$query := set_params($template, $to_id);
$result := execute($query);

```

Figure 7: Transformed Code

into a single database join query as shown in Figure 7.

Merge-projection-predicates: The application issues multiple queries in succession that are identical except in the attributes they project. This can be replaced by a query where the projection clause is a union of the projection clauses of the original queries. An example given by Manjhi et al.[7] is shown in Figure 8 and the transformed code results as shown in Figure 9.

5.2 Prefetching query results

The other performance improvement suggested by Manjhi et al.[13] is the NON BLOCKING transformation, also referred to as prefetching query results. Consider the situation where a web application performs multiple queries within a HTTP request. The standard approach is that the application executes the queries sequentially, with some intermediate business logic between them. But after issuing a database query, the application waits for the query results. This wait might be unnecessary in some cases if for example the next database query that needs to be issued does not depend on the answer to the current query. In such cases, the client latency can be greatly reduced by overlapping the query executions. If the query execution is converted to a non blocking mode, then the application can issue multiple queries together. If the latencies of queries $1...N$ is $T1...TN$, then, ideally this transformation reduces the overall latency from $(T1 + T2 + ... + TN)$ to $\max\{T1, T2, ..., TN\}$.

However, this is not possible in cases where: (i) Some of the parameters of a subsequent query is a result of a previous one, (ii) There might be updates/inserts between queries which might impact query results. (iii) The query maybe conditionally issued. In the third case, we can go ahead and do a “speculative” prefetch, if we have the necessary parameters. This means that we are doing a trade off between doing extra work which might be unnecessary, and reduction of latency.

This optimization is formally explained as a query dependency graph. This is a directed acyclic graph(DAG) where the nodes are database accesses, and there is a directed edge between two nodes

```

$template1 := SELECT MAX(bid) from bids WHERE item_id = ?;
$query1 := set_params($template1, $item_id);
$result1 := execute($query1);
$template2 := SELECT COUNT(*) from bids WHERE item_id = ?;
$query2 := set_params($template2, $item_id);
$result2 := execute($query2);

```

Figure 8: Original code

```

$template := SELECT MAX(bid), COUNT(*) from bids WHERE item_id = ?;
$query := set_params($template, $item_id);
$result := execute($query);

```

Figure 9: Transformed code

if they have to be executed in that order. Given this graph, a database access can be issued as soon as all database accesses that it depends on (all nodes incident on this node) have been completed.

Their experimental results show that these two transformations had almost no impact on the scalability in a centralized setting. However, in a distributed setting, these transformations increase scalability by over 10% and reduce latency by over 50% in many cases [7].

5.3 Batched execution of iteratively invoked queries

Guravannavar et al.[10] deal with improving performance of iteratively invoked database queries and procedures. Several data retrieval and update tasks need more expressive power than what standard SQL offers. Therefore, queries, or calls to stored procedures/user-defined functions are often invoked multiple times, either from within a loop in an application program, or from the where/select clause of an outer query.

When the invoked query/procedure/function involves database access, a naive implementation can result in very poor performance, due to random I/O. Query decorrelation addresses this problem in the special case of nested sub-queries, but is not applicable otherwise. This problem is traditionally addressed by manually rewriting the application to make it set-oriented, by creating a batch of parameters, and by rewriting the query/procedure to work on the batch instead of one parameter at a time. Such manual rewriting is time-consuming and error prone. Guravannavar et al.[10] provide solutions to: (a) Automatically rewrite programs to replace multiple calls to a query by a batched call to a correspondingly rewritten query. (b) Rewrite a stored procedure/function to accept a batch of bindings, instead of a single binding.

Thereby, for example, a query which would have been invoked many times from different invocations of a stored procedure would be automatically replaced by one (or a few) invocations of a batched version of the query. Let us take a simple example to demonstrate this. Suppose we have a csv file with each row containing information of one product. The following snippet(Figure 10) of ruby code reads each line, and computes the price of the product using a database lookup for discount using the product type. As a last step, it inserts the product into the database. This will be rewritten as shown

```

File.open("products.csv") do | file |
  file.each_line do | line |
    p = new_product(line) // instantiate a product by parsing the csv
    disc = conn.execute "SELECT discount FROM product_types
                        WHERE type_code = #{p.type}"
    p.price = p.cost * disc
    conn.execute "INSERT INTO products(name, price, desc)
                VALUES(#{p.name}, #{p.price}, #{p.desc})"
  end
end

```

Figure 10: Iterative CSV import process

```

batch = Table.new(:type_code, :discount, :name, :price) //memory
File.open("products.csv") do | file |
  file.each_line do | line |
    p = new_product(line) // instantiate a product by parsing csv
    r = Record.new
    r.attributes = p.attributes
    batch.add(r)
  end
end

DBI.store("SELECT DISTINCT type_code FROM batch", "temp_table1")
rs = conn.execute "SELECT t.type_code, discount
                  FROM temp_table1 t, product_types pt
                  WHERE t.type_code = pt.type_code"
DBI.merge(batch, rs, "type_code") // update discount column

for b in batch do
  b.price = b.cost * b.discount
end

DBI.store("SELECT name, price, description FROM batch", "temp_table2")
conn.execute "INSERT INTO products SELECT name, price, description
              FROM temp_table_2"

```

Figure 11: Batched CSV import process

in Figure 11. It can be observed that the queries running for each line in the CSV have been moved outside the loop. The assumption here is that the batch fits in memory.

Parameter batching increases the plan space for the optimizer by allowing set-oriented plans for queries and updates, reducing random IO. For inserts and updates, batching allows for more efficient integrity checks and index maintenance. It can also reduce network round trip delays in the case of a distributed setting. This batching is achieved as follows:

First, batched forms of existing operations are defined. The batched form of an operation takes a set of parameters and returns a set comprising of all the results. The parameter value is also returned along with each result in order to correlate between the parameters and the results. The same definition

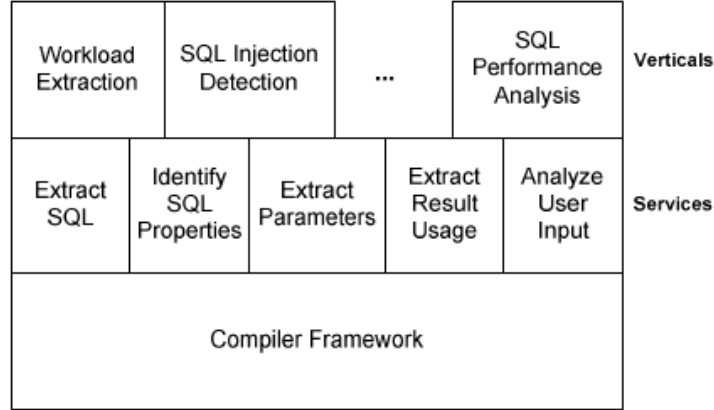


Figure 12: Static analysis services and verticals for database applications (Dasgupta et al.[4])

is extended for parameterized relational queries. This definition is straightforward in the case of side-effect free functions. For operations having side effects, there might not always be an equivalent batched form. In order to restrict the class of side-effect causing operations which can be batched, they introduce a notion of 'batch-safety' of an operation. This ensures that the return values and the final system state are independent of the order in which the parameters are processed.

Then, they use the data dependence graph and come up with a set of program transformation rules which enables batching whenever the program satisfies certain conditions. These rules continuously refine the program, while always maintaining equivalence with the original version.

6 Common infrastructure

Presently, the available compilers have almost no understanding of the data access patterns or any external entity. This is one of the reasons why most of the problems are detected at a later point in the application lifecycle. Also, a lot of the approaches described above have significant commonality. For example, extracting the possible SQL statements that can execute forms a basis for most of them. Keeping this in mind, Dasgupta et al. [4], Chaudhuri et al.[3] build a framework that adapts traditional program analysis techniques by leveraging understanding of data access APIs in order to identify such problems early on during application development. Though the framework is built specific to the ADO.NET data access APIs, the idea is quite general and can be applied to other platforms as well.

This framework, which sits on top of existing compiler frameworks, provides the following services: (a)Extract SQL (b) Identify SQL properties - like the types of the columns returned, the estimated cost etc. (c) Extract SQL parameters (d) Analyze result usage etc Figure 12. They choose the Phoenix compiler framework and work on the MSIL to provide these services [14]. In order to extract all this information, it is necessary to perform a global data flow analysis on the binary. This analysis must account for all the control paths which can reach a data access API like ADO.NET's ExecuteReader method.

```

String query = "SELECT name, salary FROM employee
                WHERE department = " + dept + ";";
if(SortEmployees()) {
    query += " ORDER BY name";
}
SqlCommand cmd = new SqlCommand(query, dbConnection);
SQLDataReader rdr = cmd.ExecuteReader();
...

```

Figure 13: Query which is sorted conditionally

Flow analysis for a basic block: A flow analysis for all such statements is performed after converting the statements to an intermediate representation(IR). Each IR statement can be mapped to a destination operand, opcode and source operands. During this flow analysis, a hash table is maintained. This hash table has the destination operand as the key, and a ‘data flow tree’ as the value. The tree contains nodes that hold the operand and the opcode, along with the referenced symbols, similar to algebraic expression trees. The Figure 14 shows the results of running a flow analysis for the cmd object in the code shown in Figure 13. This tree has two sub trees, one corresponding to the command text and one to the database connection. The leaves of the left subtree correspond to the static parts of the SQL. Hence it is possible to extract the SQL by a traversal of this subtree and concatenating the leaf nodes.

Global flow analysis: The above analysis is for a basic block of IR instructions. In order to track the definition of operands which span beyond a basic block, a global flow analysis has to be performed. This algorithm iterates over the predecessors of the current block under analysis, in order to resolve the nodes whose scope spans beyond the current basic block. The algorithm is shown in Figure 6. The UNION node is used if there are multiple resolutions for a node.

Running the algorithm on example given in Figure 13 will lead to two SQL queries as shown.

```

"SELECT name, salary FROM employee WHERE department = ' " + dept + " "
"SELECT name, salary FROM employee WHERE department = ' " + dept + " ORDER BY name"

```

Once this analysis is done, this information is used for building the other services for robustness, security and optimization, on top of this static analysis framework.

Gould et al.[8] propose a different approach for the identifying the set of potential SQL queries that could be generated. They use a control flow analysis technique which approximates the set of possible strings(queries) that the program may generate for a particular string variable at a particular program location of interest. These are called hotspots. For example, the JDBC API

```
statement.executeQuery(query);
```

which was shown in previous examples, is one such hotspot.

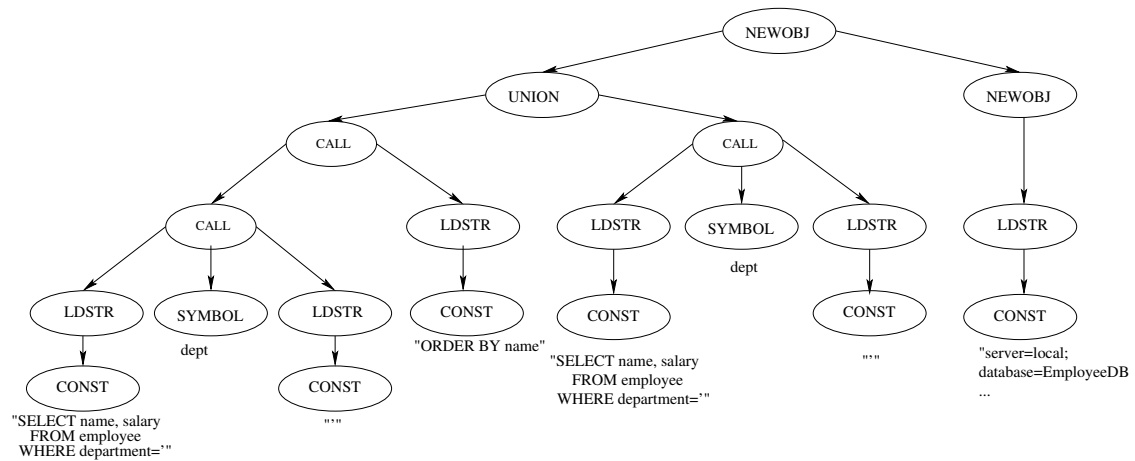


Figure 14: Expression tree for the SqlCommand object

ResolveNode

Input: Block C, Unresolved node N

Output: Resolved flow tree for N

```

for each B in predecessors(C)
  if N is defined in B
    if N's tree has undefined nodes
      call ResolveNode() for each undefined node
    else
      replace N with its tree in the flow tree of C
      mark N as resolved
    end if
  else //N is not defined in B
    ResolveNode(B,N) // look at predecessors(B)
  end if
  if N has multiple definitions
    add a UNION node with children as the different trees
  end if
end for

```

Figure 15: Algorithm for resolving operands across basic blocks(Dasgupta et al.[4])

The main steps of their analysis is as follows:

(a) Automaton generation and transformation

1. Find hotspots in the bytecode.
2. Create a flow graph for the strings present in the hotspot: the nodes correspond to variables or expressions in the program, edges represent def-use relationships.
For example: query = string1 + string2;
3. Reduce this flow graph to a grammar: treat the nodes as terminals and non terminals. For example:

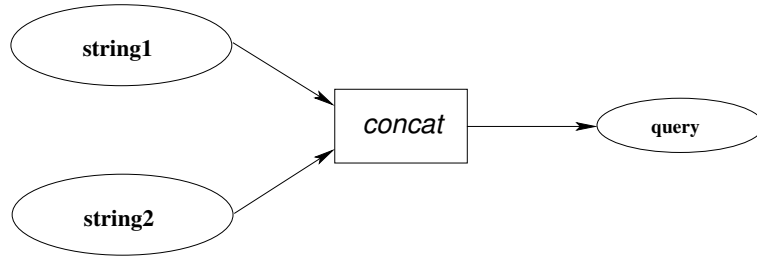


Figure 16: Steps in construction of a flow graph

$$C := S_1 S_2$$

$$Q := C$$

Here, S_1 , S_2 , C , and Q correspond to the respective nodes for string1, string2, concat, and query. This is not a regular grammar (and may not even be context free).

4. 'Widen' this grammar to a regular language. During this widening, syntax checking of the generated strings against a grammar, is done. This will result in the Finite State Automaton (FSA).
5. Do a depth first traversal of the FSA and group letters into tokens, remove whitespaces, and create an equivalent FSA with transitions over the SQL keywords, literals, and delimiters.

Syntax errors, if any, in the SQL, would be caught at this stage.

(b) Reconstruction of type environments

This step uses a variant of the CFL reachability algorithm. The CFL-reachability problem can be explained as follows:

The problem takes as inputs a context free grammar G with terminals T and non terminals N , and a directed graph D with edges labeled with symbols from $T \cup N$. Let S be the start symbol of G , and $\Sigma = T \cup N$. A path in the graph is called an S -path if its word is derived from the start symbol S . The CFL-reachability problem is to find all pairs of vertices and such that there is an S -path between s and t .

The algorithm to solve the CFL-reachability problem uses dynamic programming, and also relates to dynamic transitive closure. The problem is to maintain transitive closure of a graph while new basic graph edges can be added during graph closure. This is used in many standard program analysis algorithms such as type systems based on subtyping, alias analysis, and control-flow analysis.

The algorithm first normalizes the grammar such that each productions right-hand side contains at most two symbols. Then, new derived edges are added to D based on the productions of G . The algorithm repeatedly applies the above transformation to the graph until no more new edges can be added. Any pair of nodes and with an edge labeled X in the final graph has an X -path from s to t in the original graph D .

The steps involved in reconstruction of type environments are as follows:

1. Invoke a CFL reachability algorithm with the FSA and the SQL grammar as inputs. This will keep adding transitive edges to the automaton according to the rules of the grammar till there is an edge from the Start node to the terminal. The execution steps are shown in Figure 17 for a simple query.

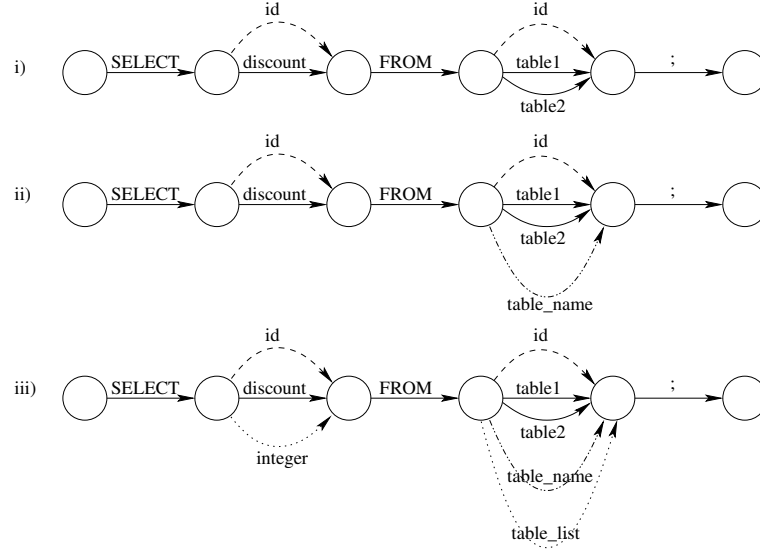


Figure 17: Steps in running the CFL reachability algorithm on the FSA

2. Once this Transitive closure graph is ready, do a top down traversal. During this, for every column, the type can be determined.
 - (a) Add new type edges to the graph. The point to note is that one column can have multiple types because of the possibility of multiple contexts(Figure 17(iii)).
 - (b) Similarly, for literals, the types are identified and edges are added at this point.

At this point, some errors could be detected. For example, a non existent column, duplicate table names, duplicate aliases etc.
3. This is the final step where type checking is done. We have the type-annotated automaton at this stage as shown in Figure 17(iii). SQL's simple type system allows the treatment of the type system as a context free grammar. So the same CFL reachability algorithm is applied using the type grammar. If during this process, there is an expression that does not match any one of the right hand side of the rules, then an error is discovered. But one limitation is that since one column can have multiple types, even if one of them does not match the right hand side, an error is reported. This is conservative, in the sense that there might not be a runtime error in some cases where an error is reported by this procedure.

Gould et al.[8] prove that if this analysis does not report any errors, then the generated object programs are type safe. Also, since this process is a one time activity, the performance is not a big concern. The complexity of the CFL reachability algorithm is cubic in the size of the graph.

7 Directions for future work

There are interesting directions for future work in this area:

7.1 Choosing the optimal holistic execution plan:

As Shopiro[16] mentions, the popular conception was that a human programmer with knowledge of the physical implementation underlying the data source (relational database) could write a much more efficient program than would be expected from a compiler of a relational language. Automatic program optimization (particularly flow analysis and data-structure selection) has reached the state at which this no longer need be true. The compiler can take more informed decisions and generate highly efficient code given the necessary information.

So far we have discussed the approaches for improving performance by performing program and query transformations. This encourages a declarative style of programming where the developer just specifies the 'what' and not the 'how'. This means that the execution plan space becomes even more wider as we consider the application code along with the database queries. The number of available execution plans is far more than just the number of query execution plans (which is itself quite large). Therefore, the problem of choosing the optimal execution plan becomes an interesting one. This will have to consider the cost of the queries along with the cost incurred in the program and find the global optimal for the application as opposed to the local optimal choices made by the program compiler and the SQL query optimizer.

As an example, consider the optimization approach presented by Guravannavar et al.[10]. Earlier, we discussed the batching of iteratively invoked database queries as an efficient execution plan. The work of Guravannavar et al.[10] shows that batching can improve performance by reducing random IO, reducing network round trips, etc (Figure 11). But, as we had observed earlier, the entire batch was assumed to be in memory. This might not always be possible. If the batch size is very large, a spill over to disk would have to be considered.

Batching comes with its own cost over and above the iterative approach. This cost includes the costs of (a) Creating the batch (in memory, with possible spill over to disk), (b) Forming the distinct parameter set, (c) Fetching the key columns along with the required data, (d) Merging the batch in memory, (e) Storing parameters in a temporary table. On the other hand, some of the benefits gained by batching can be enumerated as (a) Reduction in Network roundtrips - parameters are sent at once, (b) Single set oriented query as against N individual queries (implies sequential IO, grouped constraint checks etc). We reproduce the transformed code here, annotated with some cost factors (Figure 18). These costs (and benefits) can be classified as CPU cost, network cost, memory, and IO cost as indicated in Figure 18. So, the choice of an optimal execution plan should consider the costs of both the batched version and the iterative version and choose the plan with the minimum cost. In order to come up with an overall cost of the batched version, the parameters that are required are:

1. **Trip count(N):** In the above example, this represents the number of lines in the csv file. In general, this is the number of times a loop is executed. This is hard to determine, or even reliably estimate by analyzing the source code. For a single query plan, the traditional query optimizers use the estimates of the table sizes since it has access to table statistics during query compilation. There have been some efforts in predicting trip counts in the area of parallelizing compilers which might be of aid [15].
2. **Network latency:** This can be defined as the time taken by a packet to make a trip from a designated source to destination and back (round trip delay). High latency is often a cause of

```

// Creating the batch (possible Disk IO)
batch = Table.new(:type_code, :discount, :name, :price)

File.open("products.csv") do | file |
  file.each_line do | line |
    p = new_product(line)
    r = Record.new
    r.attributes = p.attributes
    batch.add(r)
  end
end

// Forming the distinct parameter set(CPU cost)
// Storing parameters in a temporary table(IO Cost)
// Network roundtrips reduce - parameters are sent at once (Network benefit)
DBI.store("SELECT DISTINCT type_code FROM batch", "temp_table1")

// Fetching the key columns along with the required data (Network cost, Disk IO cost)
// Single join query as against N lookup queries (Benefit)
rs = conn.execute "SELECT t.type_code, discount
                  FROM temp_table1 t, product_types pt
                  WHERE t.type_code = pt.type_code"

// Merging the batch in memory (CPU cost)
DBI.merge(batch, rs, "type_code")

for b in batch do
  b.price = b.cost * b.discount
end

// Storing parameters in a temporary table(IO Cost)
DBI.store("SELECT name, price, description FROM batch", "temp_table2")

// Network roundtrips reduce - parameters are sent at once (Network benefit)
// Single insert query as against N (Benefit)
conn.execute "INSERT INTO products SELECT name, price, description
            FROM temp_table_2"

```

Figure 18: Batched CSV import process(with costs-benefits indicated)

bad performance of applications. This delay is commonly observed in web applications which generally use a distributed server infrastructure to meet their scalability needs. Again, this parameter is tough to estimate at compile time as it depends on the deployment architecture. However, it can be determined once at runtime and used for computing costs.

3. **Disk IO:** This overhead can occur due to usage of temporary tables and also due to the batch creation which might have to be stored on disk. In order to minimize this overhead, [9] suggest rules to reduce the usage of temporary tables wherever possible. The overhead due to batch creation can be reduced by limiting the batch size and flushing the batch periodically if the entire batch cannot fit in memory.

```

N = findTripCount();
L = getNetworkLatency(); // Runtime constant
D = computeIOCost(N);
C = computeBatchQueryCost();
...
if(cost of Batching(N, L, D, C...) < cost of Iteration) {
    // execute batched version
} else {
    // execute iterative version
}

```

Figure 19: Choice of batch vs. iterative execution deferred to runtime

4. **Processor Time:** Merging the batch with the looked up parameter(‘discount’ in our example) based on the key parameters(type_code) involves additional processing overhead which can be likened to a join operation done in query processing. This cost indirectly depends on the trip count(N).

Along with the above, the query cost of the batched version should also be included. Considering the above cost-benefit analysis, we can observe that making this choice is not trivial. In fact, in a lot of cases, it is not possible to arrive at a good cost estimate in a static context. i.e. by analyzing the source code.

A possible approach could be to perform the cost estimation as much as possible statically, but deferring the choice of the plan to runtime based on the knowledge of N. Specifically, our example program could be transformed to a structure similar to Figure 19. We can compute the various parameters necessary at runtime as shown. This can give reliable cost estimates of various plans. It would be interesting to explore the learnings from *adaptive query processing* techniques (Deshpande et al.[5]) where these holistic plans could be switched at runtime. But enough care needs to be taken so that the runtime overheads introduced by these techniques, are minimized. Few other challenges involved in effective cost estimation are: (a) Conditional query execution, (b) Loops whose counts are not based on iteration over a collection e.g., while loops (c) Unknowns about the production deployment model - distributed settings might impact the costs drastically.

7.2 Holistic approaches for other external entities:

The ideas or approaches so far have focussed on interactions of an application with a relational DBMS. From the application program viewpoint, the database can be seen as an external entity with which interactions happen. A lot of these ideas are generalizable for other external entities like web services. In fact, extending the ideas presented above for non relational databases like Cassandra, SimpleDB, PNUTS, Dynamo, etc. itself poses some good problems. In a broad sense, this problem generalizes to any application in a distributed setting. The disparate nature of the various entities that may involve in a distributed application will bring in lots of challenges. [17] deal with a related problem where they optimize SELECT-PROJECT-JOIN queries spanning multiple web services.

7.3 Identifying queries in nested method invocations

The transformations proposed by Guravannavar et al.[10] considered the queries to be available as strings within the iterative loop execution. But this is rarely the case in real world applications. Application programs can be typically expected to have multiple nested method invocations and object instantiations eventually leading to a method which executes a query. Recognizing that a given loop is batchable, in the midst of the complex object model is vital for such transformations to be applicable. Also, there exist a large class of applications which do not use JDBC directly, but instead use object relational mappers like hibernate[11] which provide a higher level of abstraction.

In order to handle such complex code, a possible approach could be (a) first, identify the hotspots i.e., the statements in the code which invoke queries, (b) Build up the graph of method calls (similar to control flow analysis), (c) Generate a batched version of all the ancestors of the hotspot. This assumes that all the ancestors are batchable, which might not always be the case. Another approach could be to just collect the query parameters while the loop executes, and execute the batched version using these collected parameters when the results of the queries are used. This is the lazy evaluation strategy coupled with batching. This approach also has its limitations. If there are dependent queries within a loop, this approach would be useless as it would end up evaluating the queries iteratively. The goal therefore, is to find an elegant and simple solution to this problem.

7.4 Synthesizing Concurrency control for database applications:

Identifying the right isolation requirements of a concurrent program is a problem on which currently research is in progress. It can be shown that these requirements can be systematically derived from sequential proofs (proofs that a program satisfies certain properties in the absence of concurrent interleaving). They solve it for simple programs without procedure calls as they introduce interesting complications. In the context of a database application, if the procedure calls are actually the hotspots (statements which make a database query), then, using some of the holistic techniques, we might be able to come up with optimal isolation requirements.

8 Conclusion

In this report we have made an effort to survey existing literature on this topic of holistic approaches. This essentially involves the merging of extensive research work done both in the areas of traditional query optimization as well as in the area of compiler design and program analysis. By observing the trends in terms of the research work, we could possibly claim that the area of query optimization seems to be naturally evolving into considering factors beyond the database.

The main reason for this is probably the exponential growth of large scale distributed asynchronous systems with heterogeneous constraints. In order to effectively build software to handle the scale and the complexity, trends also show the increase in tendency towards declarative styles of programming [2]. The Claremont report on database research 2008[2] highlights this as a key research area where work needs to be done. The following statements from the report make it clear:

“Although developing new programming paradigms is not a database problem per se, ideas of **data independence**, **declarative programming** and **cost-based optimization**

provide a promising angle of attack. There is significant evidence that data-centric approaches can have major impact on programming in the near term.”

“For enterprise applications, a key distributed design decision is the partitioning of logic and data across multiple tiers: web clients, web servers, application servers, and a back-end DBMS. Data independence is particularly valuable here, to allow programs to be specified without making a priori, permanent decisions about physical deployment across tiers. **Automatic optimization** processes could make these decisions, and move data and code as needed to achieve efficiency and correctness.”

“Among the research questions we face are language design, efficient compilers and run-times, and techniques to **optimize code automatically across both the horizontal distribution of parallel processors, and the vertical distribution of tiers**. It seems natural that the techniques behind parallel and distributed databases partitioned dataflow, cost-based query optimization should extend to new environments.”

From our survey, we can conclude that a lot of this is yet to be achieved, though there have been promising efforts in these directions. Also, most of the existing research literature talks about approaches dealing with traditional relational databases. This will have to be extended for “non-relational databases”, such as Cassandra, SimpleDB, PNUTS, Dynamo, etc., which offer a subset of traditional relational database functionality, in exchange for improved scalability, performance, and/or simplicity [18].

Overall, our study says that this is a promising area for future research as it (a) is very relevant and (b) has many challenging hard problems that need to be solved, before it can be successfully applied.

References

- [1] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Trans. Comput.*, 30(5):341–356, 1981.
- [2] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O’Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. The claremont report on database research. *Commun. ACM*, 52(6):56–65, 2009.
- [3] S. Chaudhuri, V. Narasayya, and M. Syamala. Bridging the application and dbms divide using static analysis and dynamic profiling. In *SIGMOD ’09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 1039–1042, New York, NY, USA, 2009. ACM.
- [4] A. Dasgupta, V. Narasayya, and M. Syamala. A static analysis framework for database applications. In *ICDE ’09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1403–1414, Washington, DC, USA, 2009. IEEE Computer Society.
- [5] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, 2007.

- [6] C. Garrod, A. Manjhi, B. Maggs, T. Mowry, and A. Tomasic. Holistic application analysis for update-independence. In *Proceedings of HotWeb*, 2008.
- [7] C. Garrod, A. Manjhi, B. Maggs, T. Mowry, and A. Tomasic. Holistic query transformations for dynamic web applications. Technical Report CMU-CS-08-160, Carnegie Mellon University, 2008.
- [8] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, Washington, DC, USA, 2004. IEEE Computer Society.
- [9] R. Guravannavar. *Optimization and Evaluation of Nested Queries and Procedures*. PhD thesis, Indian Institute of Technology, Bombay, 2009.
- [10] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proc. VLDB Endow.*, 1(1):1107–1123, 2008.
- [11] The hibernate OR mapping tool <http://www.hibernate.org>.
- [12] D. F. Lieuwen and D. J. DeWitt. A transformation-based approach to optimizing loops in database programming languages. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 91–100, New York, NY, USA, 1992. ACM.
- [13] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic query transformations for dynamic web applications. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1175–1178, Washington, DC, USA, 2009. IEEE Computer Society.
- [14] The Phoenix compiler framework <http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx>.
- [15] L. Rauchwerger and D. Padua. Parallelizing while loops for multiprocessor systems. *Parallel Processing Symposium, International*, 0:347, 1995.
- [16] J. E. Shopiro. Theseus - a programming language for relational databases. *ACM Trans. Database Syst.*, 4(4):493–517, 1979.
- [17] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 355–366. VLDB Endowment, 2006.
- [18] Varley, Thomas. No Relation: The mixed blessings of non-relational databases. Masters' Thesis, University of Texas, Austin 2009.