Database-Aware Program Optimizations via Static Analysis

Karthik Ramachandra
I.I.T. Bombay
karthiksr@cse.iitb.ac.in

Ravindra Guravannavar
Independent Consultant
ravig@acm.org

Abstract

Recent years have seen growing interest in bringing together independently developed techniques in the areas of optimizing compilers and relational query optimization, to improve performance of database applications. These approaches cut across the boundaries of general purpose programming languages and SQL, thereby exploiting many optimization opportunities that lie hidden both from the database query optimizer and the programming language compiler working in isolation. Such optimizations can yield significant performance benefits for many applications involving database access. In this article, we present a set of related optimization techniques that rely on static analysis of programs containing database calls, and highlight some of the key challenges and opportunities in this area.

1 Introduction

Most database applications are written using a mix of imperative language constructs and SQL. For example, database applications written in Java can execute SQL queries through interfaces such as JDBC or Hibernate. Database stored procedures, written using languages such as PL/SQL and T-SQL, contain SQL queries embedded in imperative program code. Such procedures, in addition to SQL, make use of assignment statements, conditional control transfer (IF-ELSE), looping and calls to subroutines. Further, SQL queries can make calls to user-defined functions (UDFs). User-defined functions can in turn make use of both imperative language constructs and SQL.

Typically the imperative program logic is executed outside the database query processor. Queries embedded in the program are submitted (typically synchronously, and over the network) at runtime, to the query processor. The query processor explores the space of alternative plans for a given query, chooses the plan with the least estimated cost and executes it. The query result is then sent back to the application layer for further processing. A database application would typically have many such interactions with the database while processing a single user request. Such interactions between the application and the database can lead to many performance issues that go unnoticed during development time. Traditional optimization techniques are either database centric (such as query optimization and caching), or application centric (such as optimizations performed by compilers), and do not optimize the interactions between the application and the database. This is because neither the programming language compiler nor the database query processor gets a global view of the application. The language compiler treats calls to the database as black-box function calls. It cannot explore alternative plans for executing a single query or a set of queries. Similarly, the database system has no knowledge of the context in

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Example 1 An opportunity for database aware optimizations

```
int sum = 0;
stmt = con.prepareStatement("select count(partkey) from part where p_category=?");
while(!categoryList.isEmpty()) {
    category = categoryList.removeFirst();
    stmt.setInt(1, category);
    ResultSet rs = stmt.executeQuery();
    int count = rs.getInt("count");
    sum += count;
}
```

which a query is being executed and how the results are processed by the application logic, and has to execute queries as submitted by the application.

For instance, in database applications, loops with query execution statements such as the Java program snippet in Example 1 are quite commonly encountered. Such loops result in repeated and synchronous execution of queries, which is a common cause for performance issues in database applications. This leads to a lot of latency at the application due to the many network round trips. At the database end, this results in a lot of random IO and redundant computations. A program compiler or a database query execution engine working independent of each other cannot reduce the latency or random IO in Example 1. Usually such loops are manually optimized by rewriting them to use set oriented query execution or asynchronous prefetching of results. Set oriented execution reduces random disk IO at the database, and also reduces network round trips. The effect of network and IO latency can be reduced by overlapping requests using asynchronous prefetching.

However, manually performing such transformations is tedious and error prone. It is very difficult to identify such opportunities in complex programs. These transformations can be automated by performing a combined analysis of the program along with the queries that it executes. In this paper, we give an overview of the techniques that we have developed as part of the DBridge project at IIT Bombay. This is joint work with Mahendra Chavan and S Sudarshan. Our techniques automatically optimize applications by rewriting both the application code and queries executed by the application together, while preserving equivalence with the original program. Such transformations can either be implemented as part of a *database-aware* optimizing compiler or as a plug-in for any integrated development environment (IDE) providing a visual, source-to-source application rewrite feature.

The paper is organized as follows. In Sections 2, 3 and 4, we describe various program transformation techniques, and show how they work together in order to optimize the program in Example 1. In Section 5, we briefly describe the design of DBridge, a research prototype in which we have implemented the techniques presented. Related work is briefly discussed in Section 6. In Section 7, we describe challenges and open problems in this area, and conclude in Section 8.

2 Set-Oriented Query Execution

Loops containing query execution statements, such as the one in Example 1 are often found to be performance bottlenecks. Such loops suffer poor performance due to the following reasons: (a) they perform multiple round trips to the database, (b) the database system is required to process one query at a time, as a result of which, it cannot use efficient set-oriented algorithms (such as hash or merge joins), which would compute answers for all the queries in one go and (c) per query overheads, such as parameter validation and authorization checks, are incurred multiple times. Due to these reasons, loops containing query execution statements are often the most promising targets for program transformations to improve performance.

Example 2 Transformed program for Example 1 after loop fission [11]

```
int sum = 0;
PreparedStatement stmt = con.prepare("select count(partkey) from part where category=?");
LoopContextTable lct = new LoopContextTable();
while(!categoryList.isEmpty()) {
    LoopContext ctx = lct.createContext();
    category = categoryList.removeFirst();
    ctx.setInt("category", category);
    stmt.setInt(1, category);
    stmt.addBatch(ctx);
}
stmt.executeBatch();
for(LoopContext ctx : lct) {
    category = ctx.getInt("category");
    ResultSet rs = stmt.getResultSet(ctx);
    int count = rs.getInt("count");
    sum += count;
}
```

Repeated execution of a query by a loop can often be avoided by rewriting the query into its *set-oriented form* and moving it outside the loop. Given a parameterized query $q(r)$, its set-oriented form $q_b(rs)$ is a query whose result contains the result of $q(r)$ for every r in the parameter set rs [11]. Using set-oriented forms of queries avoids loss of performance due to afore-mentioned reasons. However, automating such a loop transformation requires analyzing the data dependences between statements in the loop body. Query parameters are often values produced by other statements in the loop, and query results are used by other statements in the loop. Such statements must be suitably rewritten. Loop transformation for set-oriented query execution is described in [11, 10], and consists of two key steps: (i) loop distribution (loop fission) to form a canonical query execution loop, and (ii) replacing the canonical query execution loop with the set-oriented form of the query.

Loop distribution splits a loop into multiple loops, each containing a subset of the statements in the original loop. It is applied so that query execution statements inside the loop are isolated into separate canonical query execution loops. A canonical query execution loop is a loop that can be entirely replaced by a single query execution statement. In [11] a canonical query execution loop is a loop that (a) executes a query $q(r)$ for each record r in set rs, (b) contains no statements other than the query execution statement, and (c) results of the query $q(r)$ are stored along with the record r in rs. A canonical query execution loop containing query $q(r)$ is then replaced with a statement that executes the set-oriented query $q_b(rs)$.

The final rewritten program for Example 1 after performing loop fission is shown in Example 2. The first loop in the transformed program builds a temporary table with all the parameter bindings (using the *addBatch* API). Next, a rewritten form of the query is executed (using the *executeBatch* API) to obtain results for all the parameter bindings together. Then, the second loop executes statements that depend on the query results. The scalar aggregate query in Example 1 would be transformed into the following query, where *pb* is a temporary table in which the parameter bindings are materialized.

```
SELECT pb.category, le.c1 FROM pbatch pb,
    OUTER APPLY (SELECT count(partkey) as c1
            FROM part WHERE category=pb.category) le;
```

The rewritten query uses the OUTER APPLY construct of Microsoft SQL Server but can also be written using a left outer join combined with the LATERAL construct of SQL:99. Most widely used database systems

can decorrelate such a query into a form that uses joins or outer joins [9].

Queries, being side-effect free, can be executed in any order of the parameter bindings. However, the loop can contain other order-sensitive operations, which must be executed in the same order as in the original program. The *LoopContextTable* data structure ensures the following: (i) it preserves the order of execution between the two loops and (ii) for each iteration of the first loop, it captures the values of all variables updated, and restores those values in the corresponding iteration of the second loop. The rewritten program not only avoids the overheads of multiple round-trips, but also enables the database system to employ an efficient algorithm (such as hash/sort based grouping) to evaluate the result more efficiently. This loop transformation for set-oriented execution can be applied even when a query is conditionally executed inside an *if-then-else* statement. Queries inside multiple levels of nested loops can also be pulled out and replaced with their set-oriented forms. Details of the transformations can be found in [11, 10].

Statement Reordering: Data dependencies [15] between statements inside the loop play a crucial role in the loop distribution transformation. Loop distribution cannot be directly applied when there exist *loop-carried* flow dependencies (also known as write-read or true dependencies) between statements across the loop split boundaries. However, in many cases, by introducing additional variables and reordering the statements, it is possible to eliminate loop-carried dependencies that hinder the transformation. The algorithm for reordering statements so as to eliminate loop-carried flow dependencies crossing the loop split boundaries can be found in [10]. The desired reordering is possible if the query execution statement of interest does not belong to a cycle of true data dependencies. After reordering the statements, the loop fission transformation can be applied.

3 Asynchronous and Batched Asynchronous Query Submission

As described in Section 2, batching can provide significant benefits because it reduces the delay due to multiple round trips to the database and allows more efficient query processing techniques to be used at the database. Although batching is very beneficial, it does not overlap client computation with that of the server, as the client blocks after submitting the batch. Batching also results in a delayed response time, since the initial results from a loop appear only after the complete execution of the batch. Also, batching may not be applicable altogether when there is no efficient set-oriented interface for the request invoked, as is the case for many Web services.

As compared to batching, asynchronous prefetching of queries can allow overlap of client computation with computation at the server; it can also allow initial results to be processed early, instead of waiting for an entire batch to be processed at the database, which can lead to better response times for initial results. Asynchronous submission is also applicable if the query executed varies in each iteration of a loop. In this section, we focus on performing asynchronous prefetching in loops; Section 4 additionally describes a technique to handle procedure calls and straight line code. Opportunities for asynchronous submission are often not very explicit in code. For the program given in Example 1, the result of the query, assigned to the variable *count*, is needed by the statement that immediately follows the assignment. For the code in its present form there would be no gain in replacing the blocking query execution call by a non-blocking call, as the execution will have to block immediately after making a non blocking submission.

However, it is possible to automatically transform the given loop to enable beneficial asynchronous query submission. The transformations described in the context of batching in Section 2 can be extended to exploit asynchronous submission, as presented in [5]. As shown in Example 2, the loop in Example 1 is split at the point of query execution. However, instead of building a parameter batch and executing a set-oriented query, the program in Example 2 can instead perform asynchronous query submissions as follows.

The first loop of Example 2 invokes the *addBatch* method in each iteration. In the asynchronous mode, this *addBatch* method is modeled as a non blocking function that submits a request onto a queue and returns immediately (this method could be called *submitQuery* instead, but we stick to *addBatch* so that the decision to perform batching or asynchronous submission can be deferred till runtime). This queue is monitored by a thread

pool, and requests are picked up by free threads which execute the query in a synchronous manner. The results are then placed in a cache keyed by the loop context(ctx). The *executeBatch()* invocation does nothing in the case of asynchronous submission and can be omitted. In the second loop of Example 2, the program invokes *getResultSet*, which is a blocking function. It first checks the cache if the results are already available, and if not, blocks till they become available. More details regarding this transformation can be found in [5], and the design of the asynchronous API is discussed in Section 5. There are further extensions and optimizations to this technique, and we now discuss one such extension briefly.

Asynchronous Batching: Although asynchronous submission can lead to significant performance gains, it can result in higher network overheads, and extra cost at the database, as compared to batching. Batching and Asynchronous submission can be seen as two ends of a spectrum. Batching, at one end, combines all requests in a loop into one big request with no overlapping execution, where as asynchronous submission retains individual requests as is, while completely overlapping their execution. There is a range of possibilities between these two, that can be achieved by *asynchronous* submission of multiple, smaller *batches* of queries. This approach, called *asynchronous batching*, retains the advantages of batching and asynchronous submission, while avoiding their drawbacks. This is described in detail in [16], and we summarize the key ideas here.

As done in pure asynchronous submission, the non-blocking *addBatch* function places requests onto a queue. In pure asynchronous submission however, each free thread picks up one pending request from the queue. Instead, we now allow a free thread to pick up multiple requests, which are then sent as a batch, by rewriting the queries as done in batching. The size of these batches, and the number of threads to use, are parameters that can be adaptively tuned at runtime, based on metrics such as the arrival rate of requests onto the queue, and the request processing rate [16].

Further, observe that in Example 2, the processing of query results (the second loop) starts only after all asynchronous submissions are completed i.e, after the first loop completes. Although this transformation significantly reduces the total execution time, it results in a situation where results start appearing much later than in the original program. In other words, for a loop of n iterations, the time to k-th response ($1 \leq k \leq n$) for small k is more as compared to the original program, even though the time may be less for larger k. This could be a limitation for applications that need to show some results early, or that only fetch the first few results and discard the rest. This limitation can be overcome by overlapping the consumption of query results with the submission of requests. The transformation can be extended to run the producer loop (the loop that makes asynchronous submissions) as a separate thread. That is, the main program spawns a thread to execute the producer loop, and continues onto the second loop immediately. The details of this extension are given in [16].

Asynchronous batching can achieve the best of batching and asynchronous submission, since it has the following characteristics.

- Like batching, it reduces network round trips, since multiple requests may be batched together.
- Like asynchronous submission, it overlaps client computation with that of the server, since batches are submitted asynchronously.
- Like batching, it reduces random IO at the database, due to use of set oriented plans.
- Although the total execution time of this approach might be comparable to that of batching, this approach results in a much better response time comparable to asynchronous submission, since the results of queries become available much earlier than in batching.
- Memory requirements do not grow as much as with pure batching, since we deal with smaller batches.

4 Prefetching of query results

Consider a loop that invokes a procedure in every iteration, such as the one in Example 3. The loop invokes the procedure *computePartCount*, which in turn executes a query. Both asynchronous query submission and

Example 3 Opportunity for prefetching across procedure invocations

```
rs = executeQuery("select category_id from categories where cat_group=?", group_id);
while(rs.next()) {
    int category = rs.getInt("category_id");
    partCount = computePartCount(category, flag);
    sum += partCount;
}
int computePartCount(int category, boolean flag) {
    int count = 0;
    limit = DEFAULT;
    if(flag) limit = DEFAULT * 2;
    // some computations
    rs2 = executeQuery("select count(partkey) as part_count from part where p_category=?", category);
    if (rs2.next() {
       count = rs2.getString("part_count");
       if (count > limit) count = limit;
    }
    return count;
}
```

Example 4 The program of Example 3 after inserting a prefetch submission

```
rs = executeQuery("select category_id from categories where cat_group=?", group_id);
while(rs.next()) {
    category = rs.getInt("category_id");
    submitQuery("select count(partkey) as part_count from part where p_category=?", category);
    partCount = computePartCount(category); // this function remains unchanged as in Example 3 except that
                        // the executeQuery() first looks up a cache, and blocks if results are not yet available
    sum += partCount;
}
```

batching depend on the loop fission transformation which is effective within a procedure, but not effective in optimizing iterative execution of procedures containing queries. In general, cases where a query execution is deeply nested within a procedure call chain with a loop in the outermost procedure, are quite common in database applications, and they restrict the applicability of asynchronous submission and batching. Such cases are found especially in applications that use object relational mapping tools such as Hibernate.

Consider a query which is executed in procedure M (like the *computePartCount* procedure in Example 3), which is invoked from within a loop in procedure N. Performing loop fission to enable batching (or asynchronous submission) requires one of the following transformations to M: (i) a set-oriented (or asynchronous) version of M, (ii) fission of procedure M into two at the point of query execution, (iii) inlining of M in N. All these transformations are very intrusive and complex.

A more elegant solution would be to issue an asynchronous request for the query in advance (in procedure N). Once a prefetch request for the query is placed directly within the loop, the loop can be transformed to enable batching or asynchronous submission as described earlier. Manually identifying the best points in the code to perform prefetching is hard due to the presence of loops and conditional branches; it is even harder in the presence of nested procedure invocations. Manually inserted prefetching is also hard to maintain as code changes occur.

A technique to automatically insert prefetch requests for queries at the earliest possible points in a the

Example 5 Chaining and Rewriting prefetch requests for Example 3

```
submitChain("select category_id from categories where cat_group=?",
        "select count(partkey) as part_count from part where p_category=?", group_id, "q1.category_id")
    // the program remains unchanged as in Example 3
```

Example 6 The final program of Example 3 after loop fission

```
rs = executeQuery("select category_id from categories where cat_group=?", group_id);
while(rs.next()) {
    category = rs.getInt("category_id");
    bstmt.addBatch(category);
}
bstmt.submitBatch();
for(LoopContext ctx: lct) {
    category = ctx.getInt("category");
    partCount = computePartCount(category); // code for this function remains as in Example 3
    sum += partCount;
}
```

program across procedure calls is presented in [18]. In general, the goal of prefetching is to insert asynchronous query requests at the earliest possible points in the program so that the latency of network and query execution can be maximally overlapped with local computation. Suppose a query q is executed with parameter values v at point p in the program. The earliest possible points e where query q could be issued are the set of points where the following conditions hold: (a) all the parameters of q are available, (b) the results of executing q at points e and p are the same, and (c) conditions (a) and (b) do not hold for predecessors of e. For efficiency reasons, we impose an additional constraint that no prefetch request should be wasted. In other words, a prefetch request for query q with parameters v should only be inserted at earliest points where it can be guaranteed that q will be executed subsequently with parameters v.

Detecting earliest possible points for queries in the presence of multiple query execution statements, while satisfying the above constraints, requires a detailed analysis of the program. The presence of conditional branching, loops and procedure invocations lead to complex interstatement data and control dependences which are often not explicit in the program. We approach this problem using a data flow analysis framework called *anticipable expressions analysis* and extend it to compute *query anticipability* [18].

The transformed program after inserting the prefetch request in the calling procedure of Example 3 is shown in Example 4. The prefetch request for query in the procedure *computePartCount* is placed directly in the loop just before the procedure invocation. The *submitQuery* API is a non-blocking call that submits the query to a queue and returns immediately. The queue is monitored by a thread pool as done for asynchronous submission (Section 3). Note that the procedure *computePartCount* remains unchanged. This is because when the prefetch request initiated by *submitQuery* completes, the results are placed in a cache. The *executeQuery* API within the *computePartCount* procedure will first look up this cache, and block if the results have not yet arrived.

Chaining and rewriting prefetch requests: A commonly encountered situation in practice is the case where the output of one query feeds into another, such as the case in Example 4. This is an example of a *data dependence barrier* [18], where the dependence arises due to another query. For example say a query $q1$ forms a barrier for submission of $q2$, but $q1$ itself has been submitted for prefetch as the first statement of the method. As soon as the results of $q1$ become available in the cache, the prefetch request for $q2$ can be issued. This way of connecting dependent prefetch requests is called chaining. In DBridge, this is implemented using the *submitChain* API, and for our example, the *submitChain* invocation is shown in Example 5.

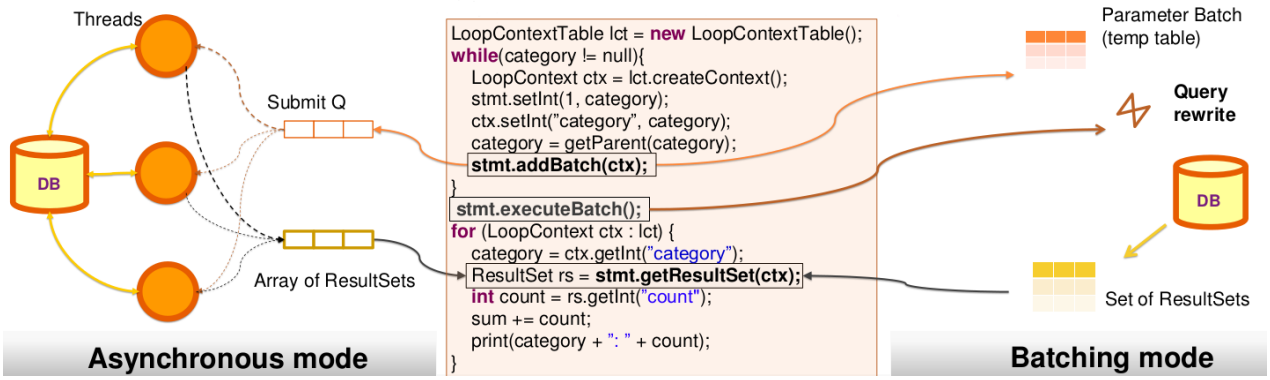

Figure 1: DBridge Batching and Asynchronous query submission API

Chaining by itself can lead to substantial performance gains, especially in the context of iterative query execution whose parameters are from a result of a previous query. Chaining collects prefetch requests together, resulting in a set of queries with correlations between them. Such queries can be combined and rewritten using known query decorrelation techniques [9]. In order to preserve the structure of the program, the results of the merged rewritten query are then split into individual result sets and stored in the cache according to the individual queries. In our implementation, the *submitChain* API internally performs query rewriting.

Integration with loop fission: Observe that in Example 4, we could alternatively perform batching or asynchronous submission by loop fission, as described in Sections 2 and 3. The rewritten program would then look as shown in Example 6. If the loop is over the results of a query, and the result attribute(s) of the query feed into the query within the loop, rewriting using the *submitChain* API is more beneficial since it achieves set oriented execution while avoiding the construction of a temporary batch table. Also, the *submitChain* API is less intrusive to the original program. The loop fission approach is beneficial in cases where the preconditions for *submitChain* [18] are not satisfied.

5 System Design and Implementation

We have implemented the above techniques and incorporated them into DBridge [4, 17], a tool that optimizes Java programs that use JDBC. The techniques however, are general, and can be adapted to other languages and data access APIs. Our system includes two components (i) a source-to-source program transformer, and (ii) a runtime batching and asynchronous submission framework. We now briefly describe these components.

Program Transformer: The program transformations are identical for both batching and asynchronous query submission. The analyses and the transformation rules have been built using the SOOT optimization framework [19], with Java as the target language and JDBC as the database access API. SOOT uses an intermediate code representation called Jimple and provides dependency information on Jimple statements. Our implementation analyzes and transforms Jimple code and finally, the Jimple code is translated back into a Java program.

Runtime Batching/Asynchronous Submission Framework: The DBridge runtime library works as a layer between the actual data access API and the application code. In addition to wrapping the underlying API, this library provides batching and asynchronous submission functions, and manages threads and caches. The library can be configured to either use batching, asynchronous submission, or asynchronous batching.

Set oriented execution API: The right side of Figure 1 shows the behaviour of the DBridge library in the batching mode. The first loop in the transformed program generates all the parameter bindings. Next, a rewritten form of the query is executed to obtain results for all the parameter bindings together. Then, the second loop executes statements that depend on the query results. Query rewrite is performed at runtime within DBridge's

implementation of the *executeBatch* method, which internally transforms the query statement into its set oriented form, as described in Section 2.

Asynchronous Query Submission API: The left side of Figure 1 shows the behaviour of the asynchronous submission API. The first loop in the transformed program submits the query to a queue in every iteration by invoking the *stmt.addBatch(ctx)* function. The queue is monitored by a thread pool which manages a configurable number of threads. The requests are picked up either individually or in batches, by free threads which maintain open connections to the database. The threads execute the query in a synchronous manner i.e., they block till the query results are returned. The results are then placed in a cache keyed by the loop context(ctx). In the second loop, the program invokes *getResultSet*, which is a blocking function. It first checks the cache if the results are already available, and if not, blocks till they become available.

6 Related Work

In the past, database researchers have explored the use of program analysis to achieve different objectives. Early approaches presented in [1, 12, 8, 13] combine program analysis and transformations in different ways to improve performance of database applications. Recently Manjhi et al. [14] describe program transformations for improving application performance by query merging and non-blocking calls. Chaudhuri et al. [2, 3] propose an architecture and techniques for a static analysis framework to analyze database application binaries that use the ADO.NET API. There has also been recent work on inferring SQL queries from procedural code using program synthesis by Cheung et al. [7, 6]. In this article, we present a consolidated summary of our work [11, 5, 18, 16] on program optimizations enabled by static analysis of the code. The techniques presented are applicable for general purpose programming languages such as Java, with embedded queries or Web service calls.

7 Open Challenges

Although many approaches and techniques for database aware program optimizations have been proposed, there are more opportunities that remain to be explored. We now discuss some open challenges and directions for future work in this area.

The techniques we have described in this paper are purely based on static analysis. There have been other approaches that use logs, traces and other runtime information for optimization. An interesting area to explore is a combination of these complementary approaches to achieve more benefits. Also, in this paper we have described techniques in the context of interactions between an application and a database. However these techniques are more general and can be extended to optimize interactions in other client server environments. Consider applications running on mobile devices or Web browsers. They interact with services typically over HTTP. These interactions are currently manually optimized using techniques similar to prefetching and batching optimizations. Adapting our techniques to automate these scenarios is an interesting and important area.

8 Conclusions

Taking a global view of database applications, by considering both queries and imperative program logic, opens up a variety of opportunities to improve performance. To harness such opportunities, program and query transformations must be applied together. Many program analyses and transformations well known in the field of compilers can profitably be made use of to optimize database access. In this article, we have given an overview of various techniques to optimize database applications, and also described some directions in which this work can be extended. We believe that this problem has the potential to attract more interest from both the database and the compilers community in future.

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] S. Chaudhuri, V. Narasayya, and M. Syamala. Bridging the application and DBMS divide using static analysis and dynamic profiling. In *SIGMOD*, pages 1039–1042, 2009.

[3] S. Chaudhuri, V. Narasayya, and M. Syamala. Database application developer tools using static analysis and dynamic profiling. In *IEEE Data Engineering Bulletin Vol 37, No 1*, March 2014.

[4] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. DBridge: A program rewrite tool for set-oriented query execution. In *ICDE*, pages 1284–1287, 2011.

[5] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *ICDE*, 2011.

[6] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. Using program analysis to improve database applications. In *IEEE Data Engineering Bulletin Vol 37, No 1*, March 2014.

[7] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. PLDI '13, pages 3–14, New York, NY, USA, 2013.

[8] G. B. Demo and S. Kundu. Analysis of the Context Dependency of CODASYL FIND-Statements with Application to a Database Program Conversion. In *ACM SIGMOD*, pages 354–361, 1985.

[9] C. A. Galindo-Legaria and M. M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *ACM SIGMOD*, 2001.

[10] R. Guravannavar. *Optimization and Evaluation of Nested Queries and Procedures*. Ph.D. thesis, Indian Institute of Technology, Bombay, 2009.

[11] R. Guravannavar and S. Sudarshan. Rewriting Procedures for Batched Bindings. In *Intl. Conf. on Very Large Databases*, 2008.

[12] R. H. Katz and E. Wong. Decompiling CODASYL DML into Retional Queries. *ACM Trans. on Database Systems*, 7(1):1–23, 1982.

[13] D. F. Lieuwen and D. J. DeWitt. A Transformation Based Approach to Optimizing Loops in Database Programming Languages. In *ACM SIGMOD*, 1992.

[14] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic Query Transformations for Dynamic Web Applications. In *ICDE*, 2009.

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[16] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan. Program transformations for asynchronous and batched query submission. *CoRR*, abs/1402.5781, January 2014.

[17] K. Ramachandra, R. Guravannavar, and S. Sudarshan. Program analysis and transformation for holistic optimization of database applications. In *Proc. of the ACM SIGPLAN SOAP*, pages 39–44, 2012.

[18] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *SIGMOD*, pages 133–144, 2012.

[19] Soot: A Java Optimization Framework: http://www.sable.mcgill.ca/soot.