# Understanding the Behavior of Database Operations under Program Control

Juan M. Tamayo  Alex Aiken
Nathan Bronson
Stanford University
{jtamayo,aiken,nbronson}@cs.stanford.edu

Mooly Sagiv
Tel-Aviv University
sagiv@math.tau.ac.il

## Abstract

Applications that combine general program logic with persistent databases (e.g., three-tier applications) often suffer large performance penalties from poor use of the database. We introduce a program analysis technique that combines information flow in the program with commutativity analysis of its database operations to produce a unified dependency graph for database statements, which provides programmers with a high-level view of how costly database operations are and how they are connected in the program. As an example application of our analysis we describe three optimizations that can be discovered by examining the structure of the dependency graph; each helps remove communication latency from the critical path of a multi-tier system. We implement our technique in a tool for Java applications using JDBC and experimentally validate it using the multi-tier component of the Dacapo benchmark.

***Categories and Subject Descriptors*** D.2.3 [*Software Engineering*]: Coding Tools and Techniques

## 1. Introduction

Database access tends to be a performance bottleneck for three-tier applications. In production settings the middle-tier software and the database are often physically separated and roundtrip communication latency between the program and the database can become the dominant factor governing overall system performance. However, it is currently very difficult for programmers to gain an understanding of the bottlenecks in system performance. While profilers can give information about where delays occur in the system (where one component must wait on the results of another), this low-level information does not necessarily suggest what higher-level reorganization will lead to improved performance.

We are interested in providing programmers with a high-level picture of program behavior useful for understanding the performance of a three-tier application. The problem is difficult because the database operations in an application can depend on each other in complex ways, both in how they interact within the program (e.g., is the output of one query potentially used as input to another query?) and in how they interact through the database (e.g., does one database operation read tuples that another database operation writes?). The relevant database statements might be far apart in the source code, they may be connected by complicated control-flow structures or layers of abstraction, or they might be surrounded by many other database operations that make it hard to reason about dependencies between them. Classical database and compiler tools are not able to solve this problem, because each only sees at most half of the picture. Standard database performance optimizations, such as indexing or query optimization, can give information about individual database operations, but are not able to understand or exploit properties of multiple queries run under program control. A standard compiler has some sense of the data- and control-flow properties of a program, but has no model of database operations and the interactions they may have through the persistent store.

Thus, any systematic approach to understanding and improving database usage must involve analysis of the flow of information both within the program and through the database. We present a tool that dynamically collects dependency information between database statements through the middle tier (Section 3.1) and through the database (Section 3.2). Our tool aggregates the data collected and presents a dependency graph between database statements (Section 4) like the ones shown in Figures 1 and 6.

As an example application, we show how the dependency graph can be used to locate optimization opportunities, as described in Sections 2 and 6. Because we are focused on program understanding, our tool generates the graphs automatically, but it does not automatically apply

*OOPSLA'12,* October 19–26, 2012, Tuscon, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10. . . $10.00

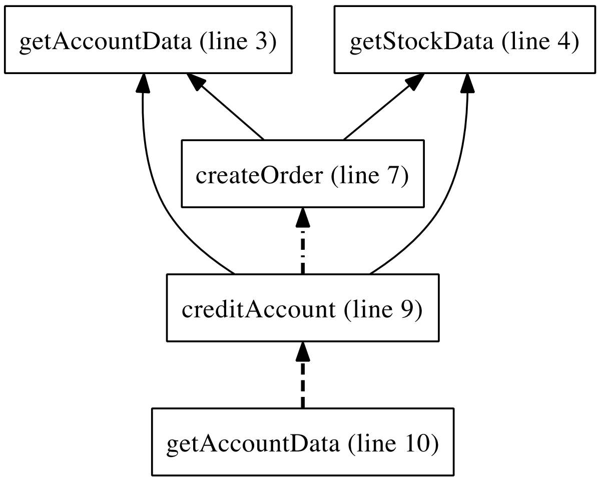


**Figure 1.** The dependency graph produced by our tool for the database statements executed in the code of Figure 2. Nodes are database statements, edges are dependencies between statements. There are four kinds of dependency: (1) solid edges are dependencies of data flowing through the middle-tier (the client program), (2) dashed edges represent read-after-write dependencies through the database, (3) dash-dot edges represent the original sequence between database writes. Write-after-read database dependencies do not occur in this example.

optimizations. (Because we rely on dynamic analysis techniques, our approach cannot prove that it is safe to perform the optimizations.) Instead, our method suggests to programmers specific, likely places to look for opportunities to improve performance and which optimization to try. We have identified three relatively simple optimizations that can improve database usage: *statement batching*, in which multiple database write statements are submitted as one; *asynchronous query execution*, where the application avoids blocking on database reads; and *removing redundant statements*, where a second read of some database tuples is replaced by simply reusing the results of a previous query. We show through a significant case study (Section 6) that the cumulative effect of applying all three optimizations can be substantial.

Our specific contributions are:

- We identify four different kinds of dependencies that are important for programmers to understand in restructuring applications that use a database, and give a dynamic analysis method for capturing the different kinds of dependencies.
- We present a novel method for summarizing the output of the dynamic analysis, grouping database statements and transactions together based on their calling context to produce concise but context-sensitive graphs illustrating the frequency, cost, and dependencies between database statements within a transaction.
- We show that opportunities for three simple program optimizations can be identified using our dependency graphs: statement batching, asynchronous query execution, and removing redundant statements. Each of these transformations improves system performance by avoiding or hiding network latency; some also reduce the total work performed.
- We present the results of a case study on a substantial three-tier application, in which our analysis techniques identify opportunities to apply all three optimizations, resulting in an overall reduction of 8 database roundtrips in the most important operation. Every multi-statement transaction exercised by this benchmark contained an optimization opportunity.

## 2. The Problem

Consider the code in Figure 2, which illustrates a "buy" operation in a stock trading application. The operation (a single method in this case) takes as arguments the stock to buy, the quantity, and the buyer. It then queries the database for details about the user and the stock, computes the total purchase price, creates a new order, calculates the buyer's new account balance, and returns an `Account` object with the buyer's balance updated.

Our goal is to improve the performance of this example. The method executes five database statements, each requiring at least one roundtrip to the database. We can remove or hide some of these delays by applying the following optimizations:

- *Statement batching*: Instead of emitting one statement at a time and waiting for its result, multiple statements can be batched and sent together to the database engine, saving one roundtrip per statement batched. For example, the statements in `createOrder [line 11]` and `creditAccount [line 13]` could be batched together, saving one roundtrip.
- *Asynchronous queries*: By emitting queries without immediately using their results it is possible to execute multiple queries simultaneously, thereby hiding the latency of some. In our example, the statements in `getAccountData [line 4]` and `getStockData [line 6]` could be executed concurrently, reducing the total operation latency by one roundtrip.
- *Removing redundant queries*: The best way to improve performance is to do less work. In our example, the execution of `getAccountData [line 15]` is unnecessary, since knowing the behavior of `creditAccount [line 13]` is enough to predict the new account balance. We can remove this query, save one more roundtrip and also reduce the load on the database.

```
Account buy(Connection con, String userID,     Account buy(Connection con, String userID,
        String symbol, int quantity) {                  String symbol, int quantity) {
  Account account =                        (A)   Future<Account> account =
      getAccountData(userID);                        getAccountData(userID);
  Stock stock =                                  Future<Stock> stock =
      getStockData(symbol);                          getStockData(symbol);
  double total =                                 double total =
      stock.getPrice()*quantity                      stock.get().getPrice()*quantity
      + account.getOrderFee();                       + account.get().getOrderFee();
                                                 Statement s = con.createStatement();
  createOrder(account, stock,              (B)   s.addBatch(createOrder(account, stock,
              quantity, total);                                quantity, total));
  creditAccount(account, total);                 s.addBatch(creditAccount(account, total));
                                                 s.executeBatch(); s.close();
  return getAccountData(userID);           (C)   account.setBalance(account.getBalance()
                                                       + total);
                                                 return account;
}                                              }
```

**Figure 2.** Example code illustrating the optimizations described in this paper. (A) Independent statements can be executed concurrently, hiding the latency for some of them. (B) Multiple write statements can be batched, sending them as one to the database. (C) Unnecessary queries can be removed.

Applying these optimizations requires a high-level view of all queries executed, combined with information about the dependencies between them. Finding these dependencies is not straightforward. In real applications operations are more complex than the illustrative example shown here, and there can be several layers of abstraction that hide the specific queries executed. Manually finding all queries and their dependencies is tedious at best. Furthermore, even a complete understanding of all the dependencies within the program is insufficient, because it is unclear what is going on in the database. In particular, do the queries touch the same or distinct tuples in the database?

To solve these problems, we propose a high-level view of the dependencies between all database statements in a transaction. Four kinds of dependencies are considered:

- *Dataflow* dependencies result when the output of one database statement flows through the program (to part of) the input of another database statement.
- *Database write-write* dependencies result when two database statements write at least one common tuple in the database. We conservatively assume any pair of database operations that write the database have a write-write dependency (see discussion in Section 3.2).
- *Database write-after-read* dependencies arise when one database operation reads a tuple $t$ and some subsequent database operation writes $t$.
- *Database read-after-write* dependencies arise when one database operation writes a tuple $t$ and some subsequent database operation reads $t$.

The dependency graph for the code in Figure 2 is given in Figure 1. Note that this dependency graph with the multiple, distinct kinds of edges, makes it easy to spot the possible opportunities to apply the three optimizations:

- Batching opportunities appear as sequences of database writes with nothing but database write-write dependencies connecting them.
- Parallelizable reads appear as sets of independent nodes (i.e., with no connecting paths of any kind of dependency).
- Duplicate reads appear as multiple independent instances of the same database statement or as database read-after-write dependencies.

Finding these optimization opportunities is reasonably straightforward in graphs like the ones presented in Section 6. For larger graphs an automated tool could scan the graph definition and automatically spot the improvement opportunities.

## 3. Finding Dependencies

To generate graphs like the one in Figure 1 we require, for every database statement executed, the set of statements on which it depends. We use dynamic information flow tracking (Section 3.1) to infer dependencies caused by data flowing through the application code, and a commutativity analysis (Section 3.2) to infer dependencies through data stored in the database.

## 3.1 Dynamic Information Flow Tracking

We use dynamic taint analysis, a widely-used technique [21], to find data dependencies in the application. In its original form [18], dynamic taint analysis prevents security attacks by labeling data coming from untrusted inputs as tainted, keeping track of the propagation of tainted data as the program executes, and detecting when tainted data is used in dangerous ways.

More generally, a dynamic taint analysis algorithm specifies a domain of *labels* used to taint data, a *join* operator for combining labels, and three policies: which data should be tainted, how taint should be tracked through the program, and where the taint of values should be checked. For finding dependencies between database statements in a program we use the following policies:

*Taint injection:* Each database statement execution is assigned a unique identifier $i$ (described further below) and all data returned from that statement is tainted with the set $\{i\}$.

*Taint tracking:* Taint propagates through all explicit data flows in the program: roughly speaking, the label on the outputs of an operation is the union of the labels of its inputs. We do not, however, track implicit flows through conditional branches (i.e., the taint of the predicate of an if-statement is not propagated to the branches). We observed that the majority of the conditionals in the program tested for error conditions. While a strict interpretation of the taint flow is that any information computed following an error check encodes the fact that the computation did not fail, we found that this increased the complexity of the resulting dependence graph without providing useful information to the user. In our benchmark this analysis choice did not suggest any false optimizations.

*Taint detection:* Taint is detected at the execution of every database statement: before a database statement $s$ is executed we record the set of labels used to build $s$.

Thus one component of a label is a set of database statement identifiers, the join for which is set union. Two main difficulties remain. First, because every execution instance of a database statement has a unique identifier, we must somehow deal with label sets of potentially unbounded size. Second, in practice this approach can yield dependencies between any pair of database statement instances, and the overwhelming majority of these dependencies are not interesting.

We take advantage of the following insight to deal with both problems. The interesting relationships are all within a single transaction; intuitively, database statements within separate transactions are already likely to be concurrent—that is exactly why they are wrapped in a transaction—and therefore not the source of performance bottlenecks. Thus, the dependencies that are useful are just those between database statements within a transaction, which is a small subset of all the dependencies in a program execution. We exclude dependencies between transactions by refining the taint labels and the join operation:

- *Identifiers* Every transaction is assigned an integer `transactionId`. Transaction ids monotonically increase in the order in which transactions are started by the program. Within a transaction, each database statement is assigned a `queryId`. Thus, every database operation executed by the program is associated with a ⟨`transactionId`,`queryId`⟩ pair. Queries executed outside an explicit transaction are assigned a singleton `transactionId`.
- *Labels* A label is a pair $\langle t, S\rangle$ consisting of a `transactionId` and a set of `queryId`'s.
- *Join.* The join of two labels $\langle t_1, S_1\rangle$ and $\langle t_2, S_2\rangle$ is

$$\langle t_1, S_1\rangle \sqcup \langle t_2, S_2\rangle = \begin{cases} \langle t_1, S_1 \cup S_2\rangle & \text{if } t_1 = t_2 \\ \langle t_2, S_2\rangle & \text{if } t_1 < t_2 \\ \langle t_1, S_1\rangle & \text{if } t_1 > t_2 \end{cases}$$

If both labels have the same `transactionId` we return a label with same `transactionId` and the union of the two sets, which is easily computed by taking the bitwise or of the two labels. If the `transactionId`'s are different, we return the label of the one with the larger `transactionId`, discarding information for earlier transactions.

This taint analysis captures the dataflow dependencies of the code in Figure 2. For example, consider the dependency between the query in `getStockData [line 6]` and the query in `creditAccount [line 13]`. This dependency arises because the `total` value used by `creditAccount [line 13]` is computed from the price in the `stock` object, which is populated from data queried in `getStockData [line 6]`. By tainting the data queried in `getStockData [line 6]` and propagating taint as the program executes, we detect the dependency between `getStockData [line 6]` and the query in `creditAccount [line 13]`. The full dependency chain is:

| | |
|---|---|
| `ResultSet` | (database driver, in `getStockData [line 6]`) |
| ↓ | |
| `stock.price` | (object field) |
| ↓ | |
| `total` | (local variable) |
| ↓ | |
| `creditAccount` | (method parameter) |
| ↓ | |
| `PreparedStatement` | (database driver, in `creditAccount [line 13]`) |

## 3.2 Finding Database Dependencies

As discussed in Section 1, data flow through the client program is insufficient to fully characterize the dependency graph on database statements. Tuples modified by a database

```
-- Read 1 (r1)
SELECT * FROM STOCKS
    WHERE SYMBOL = ?;
-- Write 1 (w1)
INSERT INTO ORDERS (USERID, AMMOUNT)
    VALUES (?, ?);
-- Read 2 (r2)
SELECT * FROM ACCOUNTS
    WHERE USERID = ?;
-- Write 2 (w2)
UPDATE ACCOUNTS
    SET BALANCE = BALANCE + ?
    WHERE ACCOUNTID = ?;
```

**Figure 3.** Dependent database statements. Question marks represent placeholders that are dynamically populated by the Java application. There is a dependency between $r_2$ and $w_2$ if the accounts identified by `USERID` (in $r_2$) and `ACCOUNTID` (in $w_2$) are the same.

statement might be used by some other statement later in the transaction, creating a dependency between the two. Because this dependency arises through data stored in the database, it is not possible to discover it without analyzing the side effects of database statements.

For example, consider the database statements in Figure 3. Using the optimizations described in Section 2, there are two possible improvements to this code:

Option 1: We could move $r_2$ after $w_2$ and batch $w_1$ and $w_2$. This optimization saves one roundtrip to the database. However, if the accounts identified by `USERID` (in $r_2$) and `ACCOUNTID` (in $w_2$) were the same, this optimization would be incorrect: since $w_2$ writes to a tuple read by $r_2$, executing $r_2$ after $w_2$ would change the result of $r_2$.

Option 2: We could move $r_2$ before $w_1$, execute $r_1$ and $r_2$ concurrently, and batch $w_1$ and $w_2$. This optimization would save two roundtrips to the database.

While Option 2 is always preferable in this example, Option 1 illustrates a difficulty when finding dependencies between database statements: the parameters to $r_2$ and $w_2$ are not known until runtime, and even then the dependencies arise through the content of the database. Without more information, a static analysis would have no choice but to conservatively assume a dependency exists.

In our dynamic analysis we classify database statements as either reads, which do not modify the database, or writes, which do modify the database's contents. Consequently, there are three types of database dependencies: *read-after-write*, *write-after-read*, and *write-after-write*. Two reads are never dependent through the database, even though they might be through the client application.

We conservatively assume there is always a dependency between two consecutive writes. Little is lost, because neither of the optimizations involving write statements (statement batching or redundant query elimination) require changing the order of writes. Furthermore, finding write-write dependencies via dynamic analysis is prohibitively expensive, as it requires either knowing the set of tuples modified by each write—which is impossible without either access to the database implementation or effectively simulating it—or executing the writes in different orders and comparing the entire state of the different databases that result to check whether they are the same. Both options are infeasible in practice.

To find dependencies between database reads and writes, we use the fact that if a write changes the result of a read then the statements are dependent. Consider a single read in a sequence of writes:

$$w_1 \quad w_2 \quad \dots \quad w_i \quad r \quad w_{i+1} \quad \dots \quad w_n$$

Since we assume that consecutive writes are dependent, the sequence of writes is fixed. We are interested in two particular writes:

- The first write $w$ after $r$ that changes the result of $r$. This write indicates how far to the right we can move $r$ without altering the result of the program. There is a write-after-read dependency between $r$ and $w$.
- The last write $w'$ before $r$ to change the result of $r$. This write limits how far to the left we can move $r$. There is a read-after-write dependency between $w'$ and $r$.

These two dependencies per read, together with the assumption that consecutive writes are dependent, characterize all dependencies through the database.

In the example of Figure 3, the sequence of reads and writes for $r_2$ is:

$$w_1 \quad r_2 \quad w_2$$

$w_2$ limits how far to the right we can move $r_2$ before obtaining the wrong result, so there is a write-after-read dependency between $w_2$ and $r_2$. In this case $r_2$ can move freely to the left so there are no read-after-write dependencies.

We implement this database dependency analysis using a proxy JDBC driver that logs all statements executed in a database transaction. Before the transaction is committed, the driver executes the following procedure:
1. Roll back the current transaction. 2. Separate the statements executed into a list of reads and a list of writes.
3. Replay the transaction by executing all writes in order. Before and after every write execute each read and compute a fingerprint of the read's result.
4. Commit the transaction.

For a transaction with $w$ writes the above procedure generates $w + 1$ fingerprints per read. To determine whether a write changes the result of a read, we compare the read's fingerprints before and after executing the write. If the fingerprints differ, the write changed the result of the read.

## 4. Graph Construction

For each database statement, the dynamic analysis algorithms of Section 3 output a *runtime statement* containing:

- the context in which the database statement was executed, in the form of a stack trace,
- the `transactionId` of the transaction of which it is a part, and
- the set of runtime statements it depends on.

All runtime statements are part of a transaction, which is the set of runtime statements with the same `transactionId`.

The major difficulty in presenting the results of the dynamic analysis to a user is that the number of runtime statements is normally overwhelming. Even for applications of moderate complexity many thousands of runtime statements are produced for realistic workloads. In this section we describe our techniques for summarizing this information in a form that is both concise and useful.

A *summary graph* groups sets of runtime statements together. Specifically, the nodes of the summary graph are sets of runtime statements with the same stack trace (i.e., a node of the graph corresponds to a stack trace). There is an edge between two nodes $n_1, n_2$ if a runtime statement in $n_2$ depends on a runtime statement in $n_1$. Nodes and edges also have *weight*, which is just the number of runtime statements in a node and the number of underlying dependencies between two nodes, respectively.

If the program contains a loop, a node may represent more than one execution of a statement within a single transaction. This will be reflected by the node's weight; if there is a loop carried dependence this will result in a self-edge or cycle.

For example, if the code of Figure 2 is executed 1000 times, a total of 1000 transactions are logged, with a total of 5000 runtime statements. These 5000 runtime statements are represented by five nodes in a summary graph, identified by the following five stack traces—each stack consists in this example consists of just one functionc call:

1. `getAccountData [line 4]`
2. `getStockData [line 6]`
3. `createOrder [line 11]`
4. `creditAccount [line 13]`
5. `getAccountData [line 15]`

These are exactly the nodes presented in Figure 1. As can be seen from this example, grouping runtime statements by stack trace dramatically reduces the quantity of data a user needs to inspect while still retaining distinctions based on calling context—i.e., the same database statement invoked in two completely different calling contexts is regarded as belonging to distinct nodes of the dependency graph. Note that `getAccountData` is called twice in different places in Figure 2 and thus occurs in two distinct nodes in Figure 1.

```
public interface TradeServices {
  (...)
  public OrderDataBean buy(String userID,
        String symbol,
        double quantity,
        int orderProcessingMode)
        throws Exception, RemoteException;
  public OrderDataBean sell(String userID,
        Integer holdingID,
        int orderProcessingMode)
        throws Exception, RemoteException;
  (...)
  public QuoteDataBean createQuote(
        String symbol,
        String companyName,
        BigDecimal price)
        throws Exception, RemoteException;
  public QuoteDataBean getQuote(
        String symbol)
        throws Exception, RemoteException;
  public Collection getAllQuotes()
        throws Exception, RemoteException;
  (...)
}
```

**Figure 4.** Simplified version of the TradeServices interface, after removing thrown exceptions and comments. This interface lists all business operations implemented by the middle tier. Each operation runs a different database transaction.

Our method produces a set of dependency graphs, where each graph corresponds to a set of executions of the "same" transaction. Of course, we have a similar problem with transactions as with runtime statements, namely that we must decide which runtime transactions belong to the same group. It is not obvious, however, how to group transactions. We observe that even in languages without explicit syntax for beginning and ending transactions, programmers still deliberately organize transactions under a particular lexical scope. For example, Figure 4 shows some of the services exposed by the business layer of Daytrader, an example online stock trading system we return to in Section 6. Implementations for the methods in the `TradeServices` interface create a transaction on method entry, which they commit before returning. If all transactions are coded in a similar manner, then all queries run within a transaction share a common calling context, namely the method that implements the service exposed by the middle layer. We can use this lexical scope to automatically group transactions into meaningful sets, and create an output graph for each set of transactions.

Now, the lexical scope for a transaction is not given to us—we must infer it from the execution instances of the runtime statements. Consider all the runtime statements $R_n$ with `transactionId` $n$. The *identifier* for transaction $n$ is

|  |  |  |
|---|---|---|
| `DaCapoTrader.run [line 107]` | `DaCapoTrader.run [line 107]` |  |
| `DaCapoTrader.runTradeSession [line 293]` | `DaCapoTrader.runTradeSession [line 293]` |  |
| `DaCapoTrader.doSell [line 477]` | `DaCapoTrader.doSell [line 477]` | ... |
| `TradeDirect.sell [line 354]` | `TradeDirect.sell [line 354]` |  |
| `TradeDirect.completeOrder [line 551]` | `TradeDirect.getAccountData [line 987]` |  |
| `TradeDirect.updateOrderStatus [line 1288]` |  |  |
| Statement 1 | Statement 2 | ... |

**Figure 5.** A set of runtime statements forming a transaction. Each column represents a runtime statement, identified by its stack trace. The identifier for the transaction (shown in gray) is the common prefix of all stack traces.

the common prefix of the stack traces of $R_n$; an example is given in Figure 5. Transactions with the same identifier are grouped, and a single graph is generated for all transactions with the same identifier.

Thus, there are two levels of grouping: sets of runtime statements are grouped into graph nodes, and a second level of grouping organizes nodes into transactions. It is actually more convenient to compute the transaction grouping first. The full algorithm for constructing graphs from runtime statements is to perform the following steps in order:

1. Compute the identifier of each runtime transaction; i.e., for all runtime statements with the same `transactionId`, compute the greatest common prefix of the runtime statements' stack traces.
2. Group runtime statements by the identifier of their `transactionId`; let these groups be $G_1, G_2, \ldots$. Each $G_i$ produces one dependency graph in the output.
3. The nodes of $G_i$ are the sets of runtime statements in $G_i$ with the same stack trace. There is an edge between two nodes if two of their runtime statements are dependent, as described above.

Figure 6 gives an example graph for Daytrader's `sell` method in the TradeServices interface listed in Figure 4. Node weights (the number of runtime statements represented by each node) and edge weights (the number of runtime statement dependencies represented by each edge) are shown.

## 5. Implementation of the Analysis

We implemented the graph construction algorithm of Section 4 as a tool that uses bytecode rewriting to transparently analyze Java programs that use JDBC directly or indirectly. Our tool can target programs hosted by a pure-Java application container such as Geronimo, since the underlying layer uses JDBC directly. Our tool uses the ASM library [7] to add information flow tracking to Java, and adds a JDBC shim to capture queries and values as they cross the boundary to the database.

Our tool provides the full functionality of our algorithm, except that we merge the taint for *queryId*s larger than 31. To avoid excess object creation we encode each label in a single 64-bit value, with a 32 bit `transactionId` and 32 bits to record the presence or absence of the corresponding `queryId` in the label's set $S$.

## 6. Case Study

In this section we report on our experience applying an implementation of our method to Daytrader, a sample application included with the Apache Geronimo application server. Daytrader is "built around the paradigm of an online stock trading system" [10] and is designed to be a realistic and sophisticated three-tier application. The Dacapo Benchmarks [5], version 2009, include a substantial workload for Daytrader. Our case study is based on this workload, with a few deployment modifications. In particular, when evaluating performance improvements we use a higher-performance database engine running outside the application server, instead of the default embedded database. We did not modify the benchmark workload (i.e., the sample data and execution script).

It is worth mentioning that a difficulty in evaluating our work is the lack of appropriate benchmarks; while three-tier applications are ubiquitous in practice, they are scarce in the research community. Besides DaCapo, other candidates include the TORPEDO [16] and 007 [15] benchmarks. Unfortunately, TORPEDO is too small to provide an interesting test of our approach, which is most useful for complex applications with large databases, and 007 focuses on a hierarchical object-relational database, while we address the issues associated with traditional RDBMSs. However, we believe that DaCapo is sufficiently involved to be a reasonable representative of three-tier applications found in practice.

The core operations implemented by the business logic layer are defined in the `TradeServices` interface, a simplified version of which is shown in Figure 4. We focus here on the `sell` business operation, because it is central to the application's performance, has one of the highest latencies, and illustrates all of the optimization strategies of Section 2. The statement dependency graph for the `sell` operation is shown in Figure 6. Our analysis tools caused an order-of-magnitude slowdown over running the benchmark without instrumentation, which is typical and acceptable for an off-line dynamic technique. Note that the output of our technique does not de-

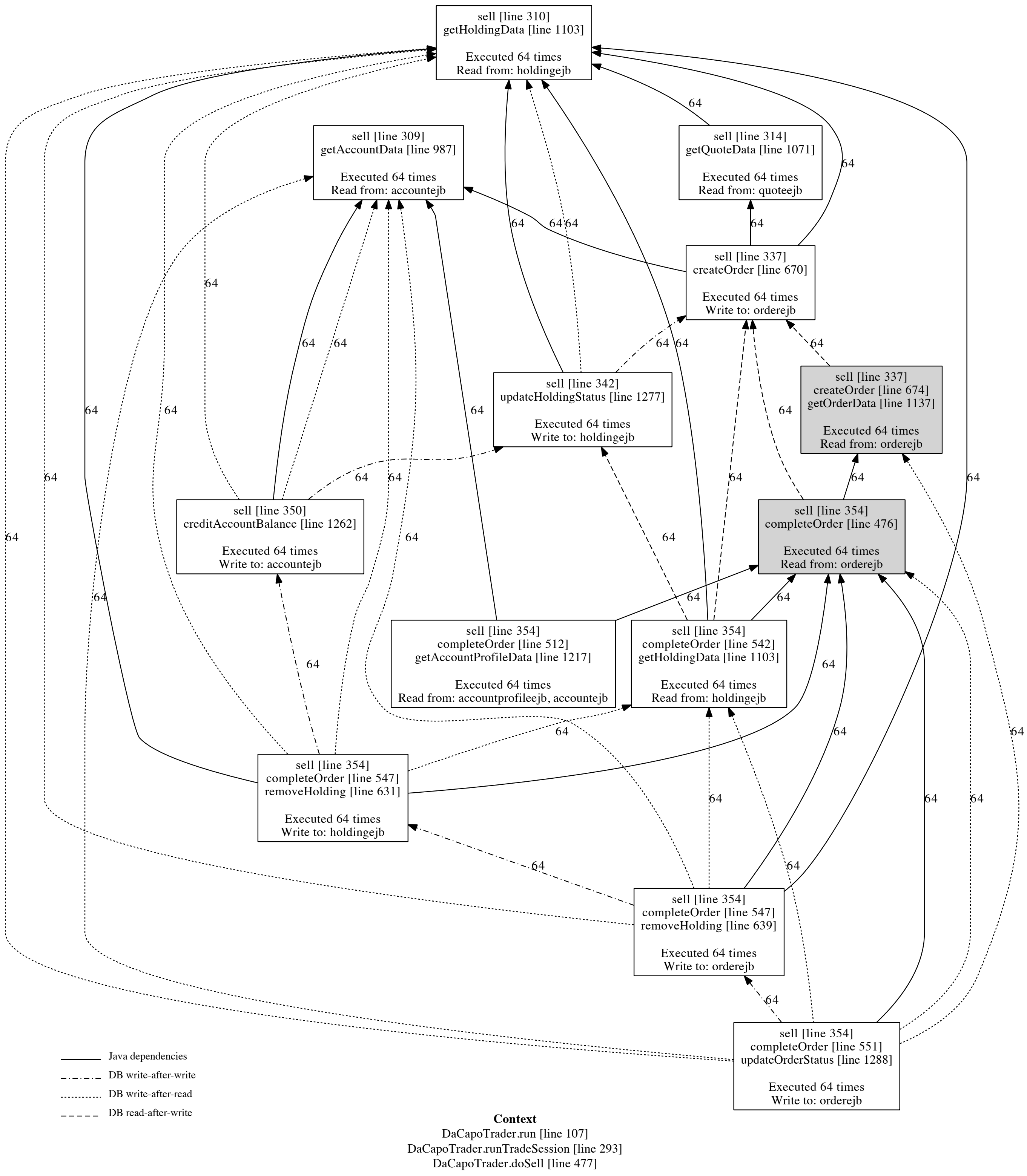


**Figure 6.** Dependency graph for the `sell` operation in the Daytrader benchmark. Gray nodes indicate repeated queries; solid edges are dependencies of data flowing through the middle-tier; dashed edges represent read-after-write dependencies through the database; dash-dot edges represent the original sequence between writes; dotted edges represent write-after-read dependencies through the database.

pend on the running time in any way, so the slowdown does not affect the output of our tool.

In the following subsections we illustrate how a programmer can use the information in the dependency graph to identify opportunities for each of the optimizations, and we also present the more important details of our implementation. Ultimately we are able to eliminate 8 round trips from the `sell` operation, including 4 from statement batching and 2 each from identifying opportunities for asynchronous execution and removing redundant queries.

Before applying our system to Daytrader we attempted to analyze the application by hand to establish an upper bound on the possible improvements. Our manual search missed several of the optimizations, and did not find any database optimizations that were not identified by our tool.

## 6.1 Statement Batching

As described in Section 2, JDBC allows multiple write statements to be submitted together to the database, instead of submitting them one at a time. Because we assume there is a dependency between consecutive writes, writes form an ordered sequence. Two writes can be batched if the only path between them in the dependency graph consists of only write-write dependencies. We want the longest sequence of writes such that each pair of writes in sequence satisfies this condition. The longer the sequences, the better, because we pay one roundtrip per batch execution, regardless of how many write statements are included.

Write statements $w_1$ and $w_2$ can be batched if no data written by $w_1$ is used to construct statement $w_2$. For example, Daytrader's `sell` operation executes the following six writes:

1. `createOrder [line 670]`
2. `updateHoldingStatus [line 1277]`
3. `creditAccountBalance [line 1262]`
4. `completeOrder [line 547]` + `removeHolding [line 631]`
5. `completeOrder [line 547]` + `removeHolding [line 639]`
6. `completeOrder [line 551]` + `updateOrderStatus [line 1288]`

As Figure 6 shows, writes 3, 4, 5 and 6 do not have any paths between them except write-write dependencies, so they should be batched if possible. Batching will reducing the number of roundtrips by three, as it will allow one statement to be sent to the database instead of four. Writes 1 and 2 can also potentially be batched, saving another roundtrip.

Implementing statement batching can require some code restructuring by the programmer. For example, JDBC does not allow different *prepared statements*—precompiled database statements used to avoid injection attacks and duplicate statement compilation—to be executed in the same batch. Thus, the programmer is forced to convert any prepared statements into regular database statements before executing them in batch.

In addition, although statement batching is a standard interface exposed by almost all database drivers, not all drivers implement batching more efficiently than regular statements. We found this to be the case for several database drivers, which synchronously executed batched statements one at a time from inside the client-side JDBC driver. In our experiments we used IBM's DB2, which properly implements server-side batching.

Finally, statement batching requires the programmer to maintain the list of statements to execute until all the statements are ready to execute together. In our case study we simply stored each statement to be batched in a local variable until it was needed. More sophisticated implementations could use a statement queue per connection that is flushed by the programmer.

In some cases statement batching could be at least partially automated. In particular, the queue could be flushed only when the application executes a read to one of the tables to be written by the batched statements, when the transaction commits, or when the programmer needs to do so manually. The main limitation would be API compatibility: JDBC drivers must return the number of rows updated after executing a write, even if the application has no immediate need for that information. Extending the API to allow programmers to submit a statement without waiting for it to execute would solve this problem.

## 6.2 Asynchronous Query Execution

JDBC provides a synchronous API: Java programs emit queries and block on the result. This is a problem when the result of multiple independent queries are required for a computation. In the statement dependency graph, reads with no paths between them are candidates for concurrent issuing. For example, in the graph of Figure 6 the following two pairs of queries have no known dependencies:

1. `getQuoteData [line 1071]` and `getAccountData [line 987]`.
2. `completeOrder [line 512]` + `getAccountProfileData [line 1217]` and `completeOrder [line 542]` + `getHoldingData [line 1103]`.

Read statement $r$ can be delayed as long as there is no statement $s$ that requires its result. A brief inspection of the code confirms that it is safe to issue both pairs concurrently. Applying these optimizations saves two roundtrips, one for each pair of queries.

As with statement batching, there are some important details that must be considered. JDBC drivers are not designed for executing concurrent queries. Database connections use locks extensively to guarantee thread-safety and block on network calls. In addition, there is a one-to-one mapping between a database connection and its transaction, making it impossible to open multiple database connections within the same transaction.

To provide the illusion of asynchronous query execution, our implementation maintains a pool of threads, each with an open database connection, that execute queries outside the main database connection. Many applications, and Daytrader in particular, wrap queries in methods that execute the database statement, iterate over the result set, and return the data in wrapper objects. This design encapsulates each database statement, making it reusable across the application. However, this design also requires the entire result set to be read before the method can return. To minimize the number of changes made to the application, we modify such methods so that they submit the query to the thread pool and return a *future* to the wrapper object, instead of the object itself. Futures are a standard way of representing the result of an asynchronous computation in Java. Changing the method's type signature to return a future type creates compiler errors exactly where the return value of the method is used. At those locations the programmer can either force the future immediately (if there is no point in delaying the execution of the query), or keep a reference to the future instead of a reference to the wrapper object, delaying the query evaluation until it is required.

DB2's default isolation level (Read Committed) can be satisfied by queries in our thread pool as long as the main transaction has not updated any of the accessed tables. In principle, all connections to the database should be auto-commit, because they never execute more than a single statement. However, we found that executing a single statement in an auto-commit connection requires two roundtrips: one to obtain the result, and one to commit the transaction. Our implementation avoids this latency by arranging for the worker threads to return the result of their read prior to a manual commit of the reading transaction. Worker threads only return themselves to the pool after this cleanup is completed.

Our concurrent query execution implements a *read committed* isolation level, even if the underlying database statements are executed at *serializable* isolation. All of our other optimizations preserve read committed or serializable isolation. This strategy is correct because DayTrader on DB2 already uses read committed isolation.

In general, program transformations may actually strengthen a weaker isolation level such as *read uncommitted*. If the application relies on repeating a query as an ad-hoc form of inter-transaction communication, for example, merging duplicate queries may result in livelock. In this particular example examining the data dependence graph will reveal the cycle, but there may be other scenarios where the effect of explicitly weak isolation levels is not apparent in the graph. This is an advantage of an approach that guides the programmer, rather than adding an extra automatic layer that alters and complicates the existing behavior.

## 6.3 Removing Redundant Queries

A straightforward way to improve the performance of any application is to do less work. In our context, unnecessary database statements should be removed. It is natural to ask the question, why would there be unnecessary queries in an application in the first place?

Object-oriented programmers build programs by composing abstractions. They try to make these abstractions self-contained, so they can be reused or extended by other people. For example, in Daytrader the order creation process is encapsulated in the `createOrder` method, which returns an `Order` object used by different business operations. Unnecessary statements can appear when two of these abstractions are put together.

Two signs in the graph point to redundant queries. First, queries that always return the same result are likely redundant. Second, read-after-write dependencies through the database are always suspicious: Why would an application execute an expensive database query to retrieve data it wrote earlier? If the data is already in memory it might not be necessary to retrieve it from the database.

In the `sell` operation of Figure 6 two sets of statements appear to be redundant:

1. `createOrder [line 670]`;
   `createOrder [line 674]` +
   `getOrderData [line 1137]`;
   and `completeOrder [line 476]`.
2. `updateHoldingStatus [line 1277]` and
   `completeOrder [line 542]` +
   `getHoldingData [line 110]`.

These sets of statements point to specific places in the source code where a programmer can look to determine if any of the queries is redundant.

Removing redundant queries is more application dependent that either statement batching or concurrent query execution. For Daytrader, we analyzed the two sets of queries given above and found the following:

1. Both the `createOrder` method and the `completeOrder` methods unnecessarily query the database for an order that is already in memory. Removing these queries saves two roundtrips to the database.
2. The `completeOrder [line 542]`, `getHoldingData [line 1103]` query reads data inserted by `updateHoldingStatus [line 1277]`. However, this update is used to "signify the sell is inflight", according to the developer's comments. We decided not to remove this signaling mechanism.

|  | `login` | `register` | `buy` | `sell` | `update` |
|---|---|---|---|---|---|
| Statement batching | 0 | 1 | 2 | 4 | 0 |
| Concurrent queries | 1 | 0 | 2 | 2 | 0 |
| Removing queries | 0 | 0 | 2 | 2 | 1 |
| Total | 1 | 1 | 6 | 8 | 1 |
| Original statements | 3 | 2 | 11 | 13 | 2 |

**Figure 7.** Improvements for the five multi-statement operations exercised by the Dacapo benchmark. In all multi-statement operations our tool found optimization opportunities.

### 6.4 Additional Operations

In addition to the `sell` operation, we applied our tool to all operations exercised by the Dacapo benchmark. Figure 7 shows the number of roundtrips saved in each operation. In five out of nine operations we found optimization opportunities, which we implemented as explained earlier this section. The remaining four operations executed a single read. In summary, our tool found optimization opportunities in every multi-statement database transaction exercised by Dacapo.

## 7. Performance Results

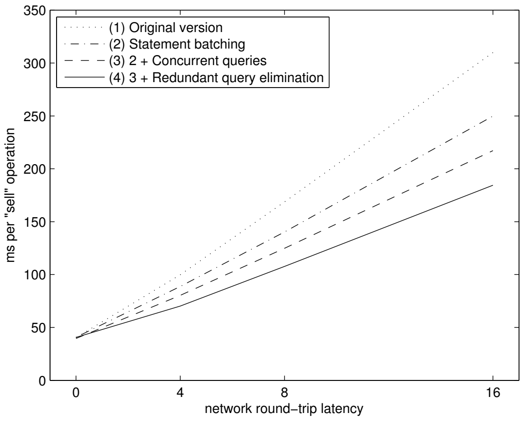
Figure 9 shows the results of implementing all the optimizations described in previous section for the `sell` operation. Experiments were run on a Dell Precision T7500n with two quad-core 2.66Ghz Intel Xeon X5550 processors and 24GB of RAM. Linux kernel version 2.6.28-16-server was used, and hyper-threading was enabled, yielding a total of 16 hardware thread contexts. We ran our experiments in Sun's Java(TM) SE Runtime Environment, build 1.6.0 21-b06, using the HotSpot 64-Bit Server VM. For the database backend we used DB2 Enterprise Server v9.7.0.0, running in the same machine as the application server.

Dacapo's workload for Daytrader exercises the business layer directly. To simulate concurrent users it keeps a set of client threads that take operations from a shared queue and execute them against the business layer. We measure the latency of the `sell` operation as seen from one of these client threads.

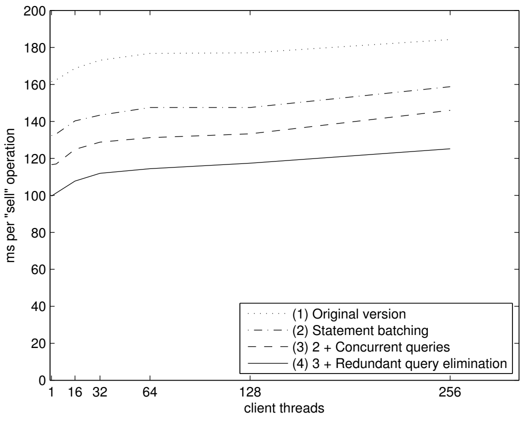
Optimizations were implemented one after the other, even though they can be implemented independently, to make the results of each optimization clearer. Figure 8 summarizes the number of roundtrips saved for each optimization. Figure 9 (top) shows the latency of the `sell` operation as the network delay increases. Reducing the number of roundtrips to the database reduces the latency of the entire operation, as expected. Figure 9 (bottom) shows the latency of the `sell` operation versus the number of client threads. The curves are reasonably flat, indicating that the machines have not saturated and the network latency dominates the overall latency of the operation.

|  |  | Estimated from measured improvement |  |  |
|---|---|---|---|---|
|  | Expected | 16 ms | 8 ms | 4 ms |
| Statement batching | 4 | 3.75 | 3.55 | 2.75 |
| Concurrent execution | 2 | 2.04 | 1.92 | 2.13 |
| Removing queries | 2 | 2.04 | 2.15 | 2.53 |
| Total | 8 |  |  |  |

**Figure 8.** Number of roundtrips eliminated for the `sell` operation, by type of optimization. We reduce up to eight round-trips to the database. As the network delay increases the measured improvement more closely matches the predicted improvements.



Sell latency vs. network delay, 16 client threads



Sell latency vs client threads, 8 ms network delay

**Figure 9.** Performance results for the `sell` operation after implementing the optimizations described in Section 6.

## 8. Related work

Previous work on optimizing three-tier applications has focused on automatic techniques for caching and prefetching the results of database queries [12, 19, 23]. There is overlap with the optimizations we have proposed as example applications of our dependency graphs: caching can hide the cost of redundant queries and prefetching effectively saves roundtrips to the database by eliminating the latency of correctly predicted queries. However, while automatic techniques will work well in many typical situations, they have a limited view of the program and will not by themselves achieve acceptable levels of performance in every situation. There is still a need for programmers to understand what is happening in a three-tier program so that they can restructure the program to improve performance, either entirely by hand or just enough that the automatic optimizations work as intended. Thus, our approach complements the work on automatic optimization of three-tier applications.

Several efforts have looked at more tightly integrating the programming language and the database, providing a better programming model than SQL statements bolted on to a stock language [3, 13]. Besides the better abstractions for the programmer, any such language system would presumably have stronger built-in semantics for the persistent store and thus begin in a better position for tools to reason about the performance of multi-tier applications. However, the need for programmers to understand the performance of such applications would not be eliminated, and we expect that dependency graphs much like we propose would be a natural medium of communication with programmers even in such higher-level languages.

Some previous efforts examine different aspects of analyzing multi-language systems, although we are not aware of any that specifically target performance in multi-tier systems. Moise and Wong [17] describe a system to infer source dependencies between multi-language systems, e.g. for a mixed Java/C++ program they determine which Java classes call a particular C++ function. Strein et al. [22] describe a prototype IDE that allows refactoring of mixed-language programs (specifically, C and Visual Basic). Salah et al. [20] describe a system that tags execution traces with a user-defined mark, which is useful for finding which parts of the source code implement which user-facing functionality.

Dynamic taint analysis is widely-used in security research. Schwartz et al. [21] present a good overview of the technique and formalize it for a simple intermediate language. They also discuss the trade-offs and challenges when choosing taint injection, propagation and checking policies. As discussed in [8], techniques for taint tracking in one language do not necessarily port well to other languages: there are many intricacies involved in building a complete, accurate taint propagation mechanism. For example, we make essential use of a trick due to [4] to instrument the entire JDK in the presence of Java's dynamic loading.

RoadRunner [9] is a Java instrumentation framework for rapid prototyping of dynamic analysis tools. RoadRunner has been successfully used to implement data race and atomicity violation detection tools. RoadRunner's framework is not general enough to implement our taint flow algorithm, primarily because it lacks shadow values for local variables and stack operands and it is not possible to propagate taint on arithmetic operations and assignments.

Luo et al. [14] describe a system in which multiple database statements that specify associative and commutative operations (e.g., an increment operation) over a set of tuples are merged into a single statement. They also propose grouping transactions that contain operations over the same set of tuples, thus increasing the number of database statements that can be merged. Their method is limited by the fact that most database drivers (and JDBC in particular) provide a synchronous interface to their clients: One query cannot be emitted before the previous one has finished. Therefore, the database engine will receive only one query at a time. Their method would be well suited for improving the performance of batched database statements like the ones produced after refactoring an application using our tool.

Bogle and Liskov [6] propose batched futures, a mechanism for reducing the cost of database calls. Instead of executing calls when the client requests them, they delay queries until their value is needed. At that point several calls may have been requested, and they can be executed in batch. Our implementation of concurrent queries has the same purpose, but eagerly executes queries in multiple database connections instead of batching them. The speedup of batched futures is limited by the number of operations that can be deferred; our tool helps the programmer reorganize the code to maximize the number of deferred operations.

For Java applications Heath [11] proposes an asynchronous database driver using both thread-pools and non-blocking socket I/O. There are asynchronous database drivers for MySQL and Drizzle [2]. Python's event-driven networking engine Twisted has an asynchronous wrapper over the standard (synchronous) Python database API [1]. The dependency information produced by our method would help a programmer port an existing application to use any of these asynchronous database drivers.

## 9. Conclusion

Understanding the performance of database operations in a three-tier system is difficult because they may be far apart in the source code, hidden behind abstraction barriers, connected by complicated control-flow structures, or they might interact only through the database. As the number of database statements in the application increases, the number of possibilities and the number of ways in which connections can be obfuscated grows super-linearly. Our dependency graph solves most of these problems.

Our goal in this work is to examine how program analysis technology can be used to understand and optimize long latency database operations, which are often a performance bottleneck in commercial three-tier applications. We have identified three simple transformations that reduce the number of, or hide the latency of, database roundtrips: statement batching, asynchronous query execution, and redundant query elimination. Identifying where these transformations may be applied requires dependency information that spans both the application and the database.

We use dynamic taint analysis to find dependencies through the middle-tier, and database statement commutativity to infer dependencies through the database. In addition, we automatically group similar transactions to present an easier-to-understand result to the programmer. We have also presented an extended case study where we applied an implementation of our methods to a three-tier Java application, and found it to be useful in improving its performance by reducing the latency of its operations.

## References

[1] Twisted Documentation: twisted.enterprise.adbapi: Twisted RDBMS support. `http://twistedmatrix.com/documents/current/core/howto/rdbms.html`, 2011.

[2] Drizzle client & protocol library. `https://launchpad.net/libdrizzle`, 2011.

[3] Ali Ibrahim Ben Wiedermann and William R. Cook. Interprocedural query extraction for transparent persistence. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 19–36, 2008.

[4] Walter Binder, Jarle Hulaas, and Philippe Moret. Reengineering standard Java runtime systems through dynamic bytecode instrumentation. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pages 91 –100, Sep 2007.

[5] S. Blackburn, R. Garner, C. Hoffmann, A. Khang, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190. ACM, 2006.

[6] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, pages 341–354. ACM, 1994.

[7] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems, Technical report, France Telecom R&D, 2002.

[8] Erika Chin and David Wagner. Efficient character-level taint tracking for Java. In *Proceedings of the 2009 ACM workshop on Secure web services*, SWS '09, pages 3–12. ACM, 2009.

[9] Cormac Flanagan and Stephen N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '10, pages 1–8. ACM, 2010.

[10] The Apache Software Foundation. Apache Geronimo v2.0 Documentation : Daytrader. `https://cwiki.apache.org/GMOxDOC20/daytrader.html/`, 2011.

[11] Mike Heat. Asynchronous database drivers. Master's thesis, Brigham Young University, April 2011.

[12] Ali Ibrahim and William R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *Proceedings of the European Conference on Object Oriented Programming*, pages 50–73, 2006.

[13] Ming-Yee Iu and Willy Zwaenepoel. Queryll: Java database queries through bytecode rewriting. In *Proceedings of the International Conference on Middleware*, pages 201–218, 2006.

[14] G Luo and M Watzke.... Grouping database queries and/or transactions, October 2010.

[15] D. DeWitt M. Carey and J. Naughton. The 007 benchmark. *SIGMOD Record*, 22(2):12–21, 1993.

[16] B. E. Martin. Uncovering database access optimizations in the middle tier with torpedo. In *Proceedings of the International Conference on Data Engineering*, pages 916–929, 2005.

[17] Daniel L. Moise and Kenny Wong. Extracting and representing cross-language dependencies in diverse software systems. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 209–218. IEEE Computer Society, 2005.

[18] J Newsome and D Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, NDSS '05, page 18. Internet Society, 2005.

[19] A. Rama, G. Yorsh, M. Vechev, and E. Yahav. Spring: Speculative prefetching of remote data. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

[20] Maher Salah, Spiros Mancoridis, Giuliano Antoniol, and Massimiliano Di Penta. Scenario-driven dynamic analysis for comprehending large software systems. In *Proceedings of the Conference on Software Maintenance and Reengineering*, pages 71–80. IEEE Computer Society, 2006.

[21] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331. IEEE Computer Society, 2010.

[22] Dennis Strein, Hans Kratz, and Welf Lowe. Cross-language program analysis and refactoring. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 207–216. IEEE Computer Society, 2006.

[23] Woong-Kee Loh Wook-Shin Han and Kyu-Young Whang. Type-level access pattern view: Enhancing prefetching performance using the iterative and recursive patterns. *Information Science*, 180(21), November 2010.