Database Queries in Java

THÈSE N^{O} 4913 (2010)

PRÉSENTÉE LE 20 DÉCEMBRE 2010

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE SYSTÈMES D'EXPLOITATION

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Christopher Ming-Yee IU

acceptée sur proposition du jury:

Prof. R. Guerraoui, président du jury
Prof. W. Zwaenepoel, directeur de thèse
Dr E. Cecchet, rapporteur
Prof. W. Cook, rapporteur
Prof. D. Kostic, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2010

Résumé

Dans les langages de programmation conventionnels comme Java, les interfaces pour accéder aux bases de données sont souvent inélégantes. Typiquement, un langage de requête doit être intégré dans un langage de programmation pour que les programmeurs puissent accéder à toute la puissance et la vitesse d'une base de données. Les programmeurs, eux, ils préfèrent utiliser un seul langage de programmation à usage général pour les calculs généraux ainsi que l'accès aux bases de données.

Cette thèse explore comment les opérations sur les bases de données peuvent être exprimée avec la syntaxe existante des langages de programmations. Les programmeurs peuvent écrire tout leur code—pour les calculs généraux ainsi que l'accès aux bases de données—dans un seul langage. Pour exécuter ces opérations sur une base de données avec des performances acceptables, des algorithmes sont nécessaires pour trouver ces opérations et les optimiser. Cette thèse s'occupe des techniques qui peuvent être facilement adoptées parce qu'elles ne nécessitent pas de changements aux compilateurs existants.

Trois systèmes ont été développés: Queryll, JReq, et HadoopToSQL. Chaque système étudie le problème selon un contexte respectivement du code en style fonctionnel, du code en style impératif, et du code en style MapReduce.

Mots-clés: bases de données, MapReduce, exécution symbolique, langages de requêtes, Java, réécriture de bytecode

Abstract

In conventional programming languages like Java, the interface for accessing databases is often inelegant. Typically, an entire separate database query language must be embedded inside a conventional programming languages for programmers to access the full power and speed of a database. Programmers, though, prefer working entirely from within their conventional programming languages, both for general-purpose computation and for database access.

This thesis explores how database operations can be expressed using the existing syntax of conventional programming languages. Programmers are able to write all their code—both general purpose code and database access code—in a single language. To run these database operations efficiently though, algorithms are needed for finding these database operations and optimizing them. This thesis focuses on techniques that can be easily adopted because they do not require changes to existing compilers.

Three systems have been developed: Queryll, JReq, and HadoopToSQL. Each system examines the problem from the context of functional-style code, imperative-style code, and MapReduce-style code respectively.

Keywords: databases, MapReduce, symbolic execution, query languages, Java, bytecode rewriting

Contents

Chapter 1

Introduction

Databases are an important component of many computer systems. They are used because they make it easier to work with large amounts of data. Usually, working with large datasets involves complex algorithms and data structures, but databases are able to abstract away these details, allowing users to focus on the data and the higher-level operations they want to perform with this data. A key component in making these abstractions practical is query languages like SQL. These languages allow users to express how they want to manipulate data in abstract high-level terms. The languages can then be translated to lower-level data structures and algorithms that can be run efficiently.

Unfortunately, accessing a database using a conventional programming language, like Java, is often much more difficult than using a database query language like SQL. Conventional programming languages are general-purpose and not narrowly focused on database access. Because conventional programming languages need to support a wide-variety of domains, the languages are more general, more complex, and less restrictive than database query languages. This generality causes the following problems when trying to program database operations with these languages:

- Their syntax is often too verbose when used to describe database operations
- They rarely perform the types of code optimizations needed to achieve good database performance because these optimizations are not useful outside the database domain
- The increased expressiveness of these languages are harder to analyze and optimize than database query languages

Currently, the most common approach for solving this problem is to embed database query languages into conventional programming languages. Typically, code for the database query language is stored in strings (Figure 1.1), so that they can be manipulated by conventional programming languages. These strings can then be passed to a library which contains a proper database language compiler and optimizer for executing the code stored in the strings. Unfortunately, this separation means that programmers must learn two completely different programming languages and that the conventional programming language compiler is not aware of the database query language and hence cannot error-check

```
PreparedStatement stmt = con.prepareStatement(
    "SELECT A.address FROM Apartments A WHERE A.rent < ?");
stmt.setInt(1, 900);
ResultSet rs = stmt.executeQuery();
Vector<String> toReturn = new Vector<String>();
while (rs.next()) toReturn.add(rs.getString(1));
return toReturn;
```

Figure 1.1: Database query languages can be embedded into conventional programming languages

it at compile-time. Additionally, programmers often must deal with integration issues like needing to manually marshal data between the database query language and the conventional programming language.

Another alternative is to modify conventional programming languages to include specific support for database features. For example, a conventional programming language can be modified to include syntax specifically for database queries (Figure 1.2). The compiler of the language then needs to be modified to recognize this query syntax and to apply appropriate database optimizations to them. Although this approach is somewhat inelegant because it encumbers a general purpose language with a narrow feature that might only be used by a minority of programmers, it is an effective solution to the problem. Modifying existing programming languages to include special syntax for database queries does have some unintended consequences though.

```
var rs = from a in apartments
         where a.rent < 900
         select a.address;
return rs.ToList();
```

Figure 1.2: Creating domain-specific languages where database features are directly supported in a general purpose programming language can cause maintenance and evolution problems

Firstly, this approach has a high barrier to adoption. A compiler is not the only component of the programmer toolchain. If the syntax of a language is altered to include support for queries, then IDEs, profilers, refactoring tools, debuggers, and all other tools from the programmer toolchain have to be adapted to handle the new syntax. This ecosystem of tools can be quite large, and programmers may be reluctant to accept a modified language syntax if they need to replace all their programmer tools with new ones.

Secondly, modifying a programming language to include query support may inhibit language and database evolution. By making a programming language more complex by including explicit support for database queries, future changes to the language become more difficult because new features may have complex interactions with these query features. Hence, language evolution is inhibited. Also, if databases are modified to include support for new types of queries, then the syntax of the programming language must be modified as well to support these new types of queries. This change requires the creation of a new language specification, the modification of the language compiler, and changes to all the other tools of the programmer toolchain as well. As a result, evolution of the database to support new features becomes inhibited.

Ideally, support for queries should be well-integrated into programming languages yet still be isolated in a separate component so that it can be maintained and evolved separately from the programming language. This thesis examines how to merge the functionality of the database query language SQL into the conventional programming language Java without requiring changes to the Java language or compiler. It uses a technique called bytecode rewriting where the output of the Java compiler, a low-level intermediate representation of code known as bytecode, is rewritten to transform Java code into queries that can be executed efficiently on a SQL database. This bytecode rewriting component is an independent component in the programmer toolchain, but its functionality can be merged into existing compilers or language runtimes. Although this thesis specifically deals with SQL and Java, the techniques and algorithms should be applicable to other modern declarative query languages and other language intermediate representations.

The techniques proposed by this thesis are an advancement over existing techniques in that

- They allow programmers to write database operations using a subset of existing Java syntax that is consistent with existing coding conventions. No changes to the Java language are required

- They include algorithms for finding and efficiently executing database operations that are written in expressive general-purpose languages instead of restricted domain-specific languages. This thesis uses symbolic execution as the basis for its algorithms

- They move all optimizations and other database functionality outside of the compiler and into a separate tool that can be maintained and evolved separately from the main language. This approach also allows for easier adoption of the system since no changes to existing tools such as compilers are needed

This thesis explores the hypothesis that bytecode rewriting is a practical apporach for supporting database queries in Java by studying three different systems: Queryll, JReq, and HadoopToSQL. For each system, programmers write database queries in Java but with a different programming style. By showing how each system uses bytecode rewriting to successfully translate these different styles of code into database queries, this thesis demonstrates the versatility and robustness of the bytecode rewriting approach to supporting database queries.

Queryll

Queryll [IZ06] speculates on how the addition of support for functional programming to conventional programming languages can lead to a simpler and more concise approach for supporting database operations in those languages. Although other systems have examined how purely functional code can be translated into database queries, imperative languages with functional features have different characteristics and require their own algorithms. Queryll demonstrates how the restrictive nature of functional-style code means it can be analyzed and translated into database operations using simple and robust algorithms.

JReq

The JReq system [ICZ10] takes existing conventions for processing large amounts of data in object-oriented imperative languages and adapts those conventions so that they can be used for performing database operations. JReq defines a syntax called the JReq Query Syntax (JQS) for writing database queries. In JQS, queries are written as loops iterating over datasets. The queries are normal Java code that can be compiled, error-checked, and even run by existing Java compilers and virtual machines. Unlike functional-style code, imperative JQS queries do contain loops and side-effects, so a more complex algorithm than Queryll is needed to translate JQS queries into efficient SQL.

Although there are existing tools such as object-relational mapping tools that can translate simple navigational queries written in object-oriented imperative code to SQL, JReq is notable in that it can handle complex query operations such as aggregation and nesting.

HadoopToSQL

MapReduce is a popular framework for working with large datasets in computing clusters. Due to the popularity of this framework, programmers would like to use this style of code to access data stored in databases. HadoopToSQL [IZ10] automatically transforms MapReduce-style queries to use the indexing, aggregation, and grouping features provided by SQL databases. MapReduce queries are distinctive in that they can contain arbitrary code that might not be expressible in SQL. Whenever possible, HadoopToSQL will translate MapReduce code into equivalent SQL queries, allowing the computation to take advantage of SQL grouping and aggregation features, but if there are no SQL equivalents, HadoopToSQL can still generate input set restrictions, optimizing the computation by allowing it to avoid scanning entire datasets.

Thesis Organization

Chapter 2 begins the thesis with an overview of related work and of some common infrastructure used by all of the systems. The thesis then examines each system in a separate chapter. Chapters 3, 4, and 5 each deal with the Queryll, JReq, HadoopToSQL systems respectively. Conclusions are discussed in Chapter 6.

Chapter 2

Background

This chapter contains two sections. The first section describes existing research in the area of databases and programming languages. The second section describes common infrastructure used by the systems of this thesis.

2.1 Related Work

2.1.1 SQL

SQL [Ame92] is currently the most popular database language for expressing database operations. To extract information from a database, programmers write queries in a declarative style. In a declarative query, the properties of the desired result are described while the exact procedures for calculating that result are unexpressed. The database can then choose the most efficient algorithm for finding the result. By contrast, in imperative languages, programmers describe the exact algorithm for calculating a result, which makes it more difficult for a database to optimize since it must understand the algorithm first before being able to replace it with a more efficient one.

To support database access, most conventional object-oriented languages provide some sort of API where database queries are stored in strings and passed to a library which interprets the strings and executes the query on a database. The Java language uses the Java Database Connectivity (JDBC) API [Sunb] for this purpose. Database operations are written in SQL, so the language for queries is completely unrelated to Java. Since SQL code is stored in strings, the Java compiler treats the SQL code as opaque data, meaning that it cannot be checked for errors until runtime when the strings are given to a library for processing. This separation of the two languages extends to the underlying data models, so programmers must manually marshal data between Java and SQL. Although JDBC provides some helper methods to help with data marshaling, programmers must still manually pack parameters into queries and then manually read out and interpret individual fields from the query results. Figure 2.1 shows an example of a JDBC query.

Because of the problems with APIs like JDBC, database vendors have created variants of common conventional programming languages that include direct database support. Embedded SQL is the name for embedding SQL into languages in this way. Embedded

```
PreparedStatement stmt = con.prepareStatement(
    "SELECT A.address FROM Apartments A WHERE A.rent < ?");
stmt.setInt(1, 900);
ResultSet rs = stmt.executeQuery();
Vector<String> toReturn = new Vector<String>();
while (rs.next()) toReturn.add(rs.getString(1));
return toReturn;
```

Figure 2.1: A sample SQL query written using JDBC

```
int rent = 900;
#sql iterator Addresses(String address);
Addresses a = null;
#sql a = {select address
            into
            from Apartments
           where rent < :rent };
Vector<String> toReturn = new Vector<String>();
while (a.next()) toReturn.add(a.address());
return toReturn;
```

Figure 2.2: A sample SQL query written using embedded SQL in Java

SQL for Java is often known as SQLJ [EM98], and it allows SQL statements to be intermixed directly with Java code. The SQL code can access Java variables for parameters and results can be stored directly into Java data structures without explicit marshaling. Because SQL is no longer treated as opaque strings, it can be error-checked by the compiler. Embedded SQL does not, however, hide the differences between the relational model of SQL and the object-oriented model of Java and between the large syntax differences between SQL statements and Java statements. There are also problems with tools and tool evolution. Typically, a precompiler is used to compile SQLJ into Java code that uses JDBC. Every time the database or Java evolves, the precompiler must be adapted to these changes. Since most programmers make use of IDEs, these IDEs must be made aware of the syntax changes, of needing to use the precompiler, and of potential debugging issues. Embedded SQL tools must constantly be updated in order to keep up with the latest changes in tools. Less common IDEs or more obscure tools in a programmer's toolchain may simply be unsupported. Figure 2.2 shows an example of a query written in embedded SQL for Java.

2.1.2 Navigational Databases

Although SQL databases are currently the most common type of database, navigational databases predate SQL databases and are still used in some applications. One of the earliest types of navigational database are the databases based on the CODASYL [TF76] standard. These databases are based on a network model for databases, in which data is modeled as entities with navigational links to sets of related entities. CODASYL queries are typically written in an imperative style where a programmer must specify how and when to move between related entities. Since navigational paths are fundamental components of network databases, it is unsurprising that these databases excel at navigational-style queries. Given a reference to an entity in the database, one can easily navigate to references of related entities. For queries which involve ad hoc relationships between entities and which involve complex filtering of these entities, CODASYL databases tend to be verbose and difficult to optimize. Though there is much research into query optimization for CODASYL [KW82], the imperative nature of the queries limits the types of optimizations possible. This thesis shows how modern languages (CODASYL is usually embedded in COBOL) can support more concise imperative queries and is able to optimize them to achieve good performance despite complex filtering or the use of ad hoc entity relationships.

2.1.3 Object-Oriented Databases

The modern incarnation of navigational databases is the object-oriented database (OODB) [MSOP86]. In modern object-oriented programming languages, in-memory data is represented as objects. OODBs extend this representation to persisted data, providing programmers with a single abstraction for data regardless of storage location. Programmers do not need to manually translate data between different formats or to have different mental models for data. In fact, OODBs strive to achieve the goal of transparent persistence, where programmers use data without having to think about how it is stored or accessed because persistence issues are completely abstracted away. OODBs map well onto a navigational model of databases because objects are primarily accessed through manipulating their fields and navigating among related objects.

Although the most popular programming languages are object-oriented, the most popular and most mature databases are SQL databases. Unfortunately, SQL's table-oriented model for data is inconsistent with the object model of object-oriented programming languages. Object-Relational Mapping (ORM) tools such as Ruby on Rails, Hibernate [JBo], or EJB [Suna, DK06] attempt to bridge this difference. They provide an object-oriented abstraction layer on top of a SQL database. Programmers specify a mapping from SQL tables to an object representation, and the ORM tool then generates code that allows programmers to manipulate these objects and have these changes be persisted automatically to the corresponding SQL tables. For example, consider a simple database describing bank clients, each of whom may have multiple bank accounts. This database might be composed of two tables (Figure 2.3): Client and Account. Using the Queryll ORM tool, this database can be mapped to the class diagram in Figure 2.4.

An ORM abstraction layer essentially provides an OODB-like API for SQL data. Similar to when using an OODB, programmers can then manipulate objects without concerning themselves with data marshaling issues. Again, like an OODB, navigational queries are well-supported. ORM tools generate accessors on objects for manipulating the fields of a

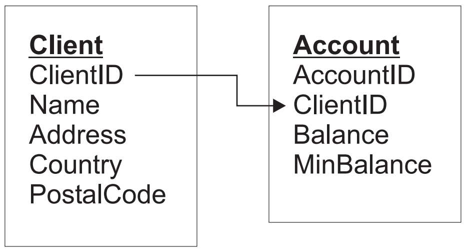


Figure 2.3: A simple database

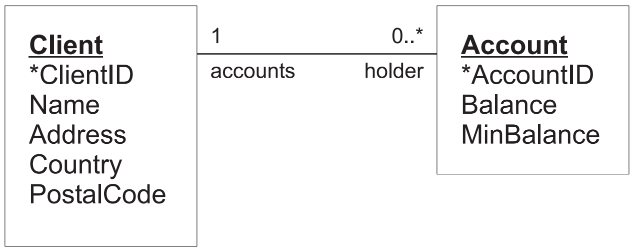


Figure 2.4: Class diagram of database entities (* denotes primary keys)

record and simple methods are provided for traversing related objects. They also provide abstractions for dealing with updates, error-handling, and transactions.

Both OODBs and ORM tools (which provide an OODB abstraction over SQL databases) face similar limitations when handling more complex queries. Although objects are well-suited for expressing navigational queries, queries involving complex filtering or ad hoc relationships cannot be expressed using a simple object API of methods and field accesses. General-purpose object-oriented languages do not have sufficient query support to handle these types of queries. To write complex queries, programmers can either switch to a domain-specific language with integrated query support or they need to use a separate query language. The common query languages for OODBs and ORMs are derived from the Object Query Language (OQL), and they have disadvantages that are similar to those of JDBC. Queries are encoded in strings that cannot be type-checked until runtime, programmers must manually encode parameters, and programmers must manually marshal data out of query results. Figure 2.5 shows an example of such a query written using the Java Persistence API [DK06].

```
List l = em.createQuery("SELECT a FROM Account a "
  + "WHERE 2 * a.balance < a.creditLimit AND a.country = :country")
  .setParameter("country", "Switzerland")
  .getResultList();
```

Figure 2.5: A sample query written in the Java Persistence Query Language (JPQL)

2.1.4 Complex Queries

Many researchers have studied how to support complex queries in general-purpose programming languages without changing the languages.

Functional Queries

Current query languages like SQL tend to be declarative. Since functional programming languages are also declarative, database queries can be easily expressed in functional languages. Kleisli [Won00] demonstrated that it was possible to translate queries written in a functional language into SQL.

Microsoft was able to add query support to object-oriented languages by extending them with declarative and functional extensions in a feature called Language INtegrated Query (LINQ) [Tor06]. LINQ adds a declarative syntax to .Net languages by allowing programmers to specify SQL-style SELECT...FROM...WHERE queries from within these languages (Figure 2.6). This syntax is then internally converted to a functional style in the form of lambda expressions, which is then translated to SQL at runtime. To support this runtime translation, the compilers for .Net languages compile lambda expressions into two forms: executable code and a data structure representation that can be inspected at runtime. Although there are similar proposals for languages such as Java [WPN06], LINQ has demonstrated that significant and invasive changes to the syntax and type system to the Java language would be required. Adding similar query support to an imperative programming language like Java without adding specific syntax support for declarative or functional programming results in verbose queries and requires meta-programming extensions to the language [CR05].

```
var rs = from a in apartments
         where a.rent < 900
         select a.address;
return rs.ToList();
```

Figure 2.6: A sample LINQ query

Scala [Ode06] is a language that combines object-oriented and functional programming. Although the language is not a derivative of Java, Scala is often associated with Java because Scala code is typically compiled to run on Java virtual machines and because Java's libraries are commonly used in Java programs. As such, research into supporting database queries in Scala is often used as an example of how query support can be added to existing object-oriented languages that have been augmented with some functional programming features. When database queries are expressed as functions in Scala, Scala must somehow manipulate the code of these functions to translate them into database queries. Although limited tricks with type inferencing can be used to support simple queries [SZ09], changes to the compiler are needed for more complex queries [GIS10].

Imperative Queries

In imperative languages like Java, the normal style for filtering and manipulating large datasets is for a programmer to use loops to iterate over the dataset. As a result, researchers have tried to develop systems that allow programmers to write database queries in imperative languages using such a syntax. Wiedermann, Ibrahim, and Cook [WC07, WIC08] have successfully translated queries written in an imperative style into declarative database queries. They use abstract interpretation and attribute grammars to translate queries written in Java into database queries. Their work focuses on gathering the objects and fields traversed by program code into a single query (similar to the optimizations performed by Katz and Wong [KW82]) and on recognizing filtering constraints. Their approach lacks a mechanism for inferring loop invariants and hence cannot handle queries involving aggregation or complex nesting since these operations span multiple loop iterations. Their approach does support inter-procedural optimization though and is particularly well-suited for optimizing code written in a transparently persistent style.

The difficulty of translating imperative program code to a declarative query language can potentially be avoided entirely by translating imperative program code to an imperative query language. The research of Liewen and DeWitt [LD92] or of Guravannavar and Sudarshan [GS08] demonstrate dataflow analysis techniques that could be used for such a system. Following such an approach is impractical though because all common query languages are declarative because declarative query languages are easier for databases to optimize.

MapReduce Queries

Programmers are increasingly using MapReduce [DG04] for performing queries over large datasets. With MapReduce, programmers write queries by defining two functions—map and reduce—for filtering, processing, and grouping records together. MapReduce is popular because it transparently handles many of the difficulties of processing data on clusters of commodity hardware, including issues such as fault tolerance, data transfer, and data partitioning.

Both MapReduce and databases are used for processing and querying large datasets stored in computing clusters. Because the two approaches have different processing models but are used in similar domains, researchers have been studying the relative merits of the two approaches. In fact, there has recently been many position papers comparing SQL-based approaches for querying data stored on a cluster of machines versus MapReduce-based approaches [PPR$^+$09, SAD$^+$10, DG10]. The two approaches show different strengths and weaknesses in areas such as scalability, fault tolerance, performance, and flexibility. As a result, some researchers have tried building hybrid systems that combine properties of both approaches.

This thesis examines the possibility of combining MapReduce and databases by using MapReduce as the interface for expressing data processing code, but to make use of database features such as indices to accelerate the computation. Programmers have started using the MapReduce abstraction with advanced storage engines that support database features [CDG+06] instead of cluster file systems. To make use of the database features though, programmers must write their database operations in a separate database query language instead of normal MapReduce code. This thesis focuses on automatically rewriting MapReduce code to use database operations. Unlike the functional and imperative query systems described previously, MapReduce programs are distinctive in that they not only use a syntax that is a mix of functional and imperative styles, but programs can also include arbitrary computation in their data processing code.

Another approach to combining MapReduce and databases involves layering a declarative query language on top of MapReduce, so that MapReduce exports a database-like interface. Hive [TSJ+09] and PigLatin [ORS+08] are examples of such an approach. These query languages are much less verbose than regular MapReduce, and their restricted structure can be analyzed with conventional techniques. Unfortunately, a programmer loses many of the benefits of MapReduce by using such query languages. One of the main advantages of MapReduce is that programmers can perform arbitrary computation at data nodes. This computation can save communication bandwidth by aggressively filtering, compressing, and transforming data before the data is transferred. The restricted syntax of query languages built on top of MapReduce is not rich enough to express such complex algorithms.

HadoopDB [ABPA+09] is another system that provides a database-like declarative query language as its interface. It uses a Hive-derived query language as its input. This query language is not merely translated to MapReduce but to a mix of MapReduce and SQL. Hence the resulting query execution uses the scaling features of MapReduce but can also take advantage of SQL features like indices. Although the queries are easier to analyze and optimize, they are not sufficiently expressive to describe complex performance-enhancing algorithms.

DryadLINQ [YIF+08] is a query language for the Dryad [IBY+07] distributed execution engine, which, like MapReduce, is designed for processing large datasets in large computing clusters. Instead of providing a simple declarative query language on top of Dryad, DryadLINQ uses a variation of LINQ. Consequently, in addition to writing simple declarative-style queries, programmers can also include arbitrary computation in their queries. DryadLINQ researchers are also studying how to adapt DryadLINQ to support using database features like indices on the back-end datastore.

2.1.5 Bytecode Rewriting and Symbolic Execution

The systems described in this thesis make heavy use of an approach to program transformation called bytecode rewriting that allows a tool to modify the behavior of a program without changing compilers or virtual machines. Because bytecode is a low-level representation of program code, symbolic execution is used to build higher-level representations of the code, which can be more easily manipulated.

All Java compilers compile Java programs into a machine independent intermediate representation known as bytecode. This bytecode is stored in files called class files. Java programs are distributed as class files which can be executed using a Java VM. Bytecode rewriting is a well-known Java technique for modifying the behavior of compiled Java code.

A typical example would be J-Orchestra [TS04] which can alter Java objects so that they can be invoked remotely without requiring changes to the original code. Many aspect-oriented programming [KLM$^+$97] tools also make use of bytecode rewriting to support dynamic aspect weaving [PSDF01]. And some ORM tools already make use of bytecode rewriting to transparently add persistence code to ordinary Java objects to enable those objects to be stored in databases. These uses of bytecode rewriting are limited to only modifying surface features of code such as intercepting method calls; the bytecode analysis used in this thesis requires a deeper understanding of the structure of code. The automatic parallelization program javab [BG97] is one example of a bytecode rewriting tool that performs similar detailed code analysis. One can consider class file decompilation [MH02], where bytecode is converted to Java source files, to be another form of bytecode rewriting involving deep code analysis.

The type of symbolic execution used by the algorithms in this thesis is similar to work done in the software verification community, especially work on translation validation and credible compilation [Rin99, Nec00]. With translation validation, a compiler not only translates an input program into an output program, it also generates a proof that the output program implements the input program. A proof-checker can then be used to verify that the proof and hence the compilation is correct. Proofs are usually composed of simulation relations, which describe the relationship between variables and execution points in the input and output programs. Proof-checkers will use symbolic execution to execute both the input program and output program. The preconditions and postconditions of executing the code will be gathered and often stored as Hoare triples. A proof-checker will then use the simulation relations to verify that the postconditions that hold at various points in the code are equivalent, thus proving the equivalences of the input and output programs. For complex compiler optimizations, it is often difficult to prove the correctness of a compiler for all inputs, but it is feasible for a compiler to automatically generate proofs showing the correctness of a particular run of the compiler. In the situation that a compiler is not correct for all inputs, when the compiler processes a problematic input, its outputted correctness proof will not hold, and the proof-checker will catch that error.

2.2 Common Infrastructure

When programmers need to work with persisted data in a modern object-oriented language, it is now generally accepted that an object representation of this data is highly desirable. All of the systems described in this thesis work with persisted data, so an object mapping layer has been written that provides an object representation of data for these systems. This layer essentially serves as an ORM although it is not restricted to relational data; nevertheless, it will be referred to as an ORM, for lack of a better term. This ORM serves as a common piece of infrastructure for all of the systems in this thesis.

The ORM is written in a combination of Java and XSLT. Programmers provide an XML description of entities, their fields, and the relationships between these entities to the ORM (Figure 2.7). The ORM then generates a series of Java classes representing these entities as objects (Figure 2.8). The classes contain getter and setter accessor methods for

```
<entity name="Customer" table="Customers">
   <field name="CustomerId" type="int" key="true" column="CustomerId"/>
   <field name="Name" type="String" column="Name"/>
</entity>

<link map="1:N">
   <from entity="Customer" field="Accounts"/>
   <to entity="Account" field="Customer"/>
   <column from="CustomerId" to="CustomerId"/>
</link>
```

Figure 2.7: An example of an XML description of a Customer entity

```
class Customer {
   ...
   String getName();
   int getCustomerId();
   Collection<Account> getAccounts();
}
```

Figure 2.8: The ORM will generate a class to represent each entity in the database

manipulating the fields of the entities. The ORM also generates a special EntityManager class that ensures that when the object representation of entities are manipulated, the database versions remain updated and consistent. For MapReduce programs, it is not possible to alter data that has been persisted, so the ORM instead provides simple classes for reading and writing entities from a database or text file.

In addition to generating Java classes that provide an object representation of entities, the ORM also parses the information about entities into a form that can be understood by later bytecode analysis tools. Figure 2.9 shows how the ORM tool fits into the programmer toolchain. The programmer first provides a description of their entities to the ORM tool, which generates some entity classes. The programmer can then write a Java program that uses these entity classes. All of this Java code is compiled by a Java compiler into Java bytecode. Before the code is run in a VM, a bytecode analysis tool can analyze the program's bytecode by using information about the generated entity classes.

More modern ORMs do not require a separate stage in the programmer pipeline for generating entity classes because they allow a programmer to write their entity classes directly in Java themselves (augmented with some annotations describing how they should be mapped to a database). When this Java code is compiled, the annotations describing the mapping between a class's methods and database fields is embedded into the Java bytecode. A bytecode analysis tool can then read this mapping information directly from the bytecode. The ORM used in this thesis serves as an easily-modifiable prototyping tool; hence, its basic design as a code generator. Nothing precludes a more advanced

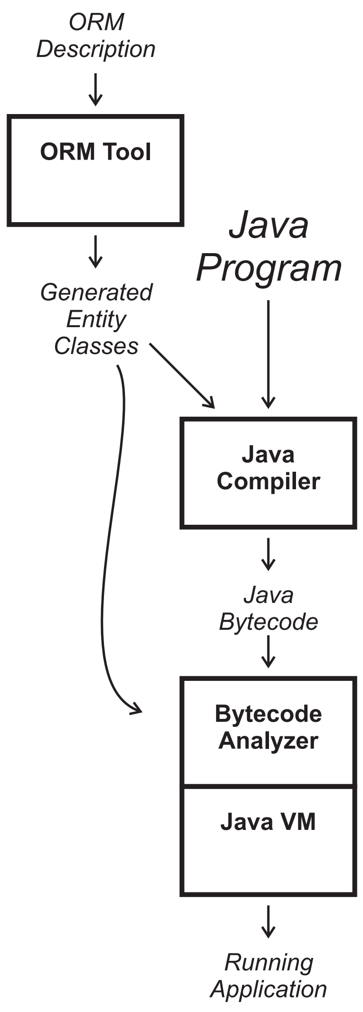


Figure 2.9: The ORM used in this thesis behaves as a code generator in the programmer toolchain

ORM from being used in its place.

Chapter 3

Queryll: Functional-Style Queries

Functional-style queries are often desired even in imperative languages because the syntax of functional-style queries is similar to that of declarative query languages like SQL. This familiarity of this syntax also means programmers can more easily reason about the behavior of their queries and compiler writers can more easily design mappings from functional-style queries to declarative query languages.

Traditionally, functional-style queries have been problematic in Java due to insufficient language support for functional programming, resulting in extremely verbose queries. However, there are many proposals for adding improved support for functional programming to Java like CICE [LLB], FCM [CS], and BGGA [BGGvdA]. When one of these proposals is eventually adopted, it will become possible to write functional-style queries in Java much more compactly. This improvement in syntax makes a functional approach to database queries in Java much more practical.

This chapter describes how functional-style queries might eventually look in Java, and it describes an algorithm called Queryll for translating these queries into SQL. The primary research contribution of this chapter is the Queryll algorithm, which translates imperative Java code into declarative SQL. The algorithm is able to take advantage of the fact that the code is written in a functional-style, resulting in a simple and robust algorithm.

3.1 Challenges and Motivation

Although Kleisli [Won00] demonstrated how one could translate functional code into relational queries, and Microsoft's LINQ [Tor06] provides a commercial implementation of such a system, Java has peculiarities that require a distinct algorithm. In full functional languages, functions have a high-level representation that can be easily analyzed and manipulated. As an imperative language, Java does not provide a nice high-level representation of functions like a functional language.

The Java compiler can be enhanced with support for database queries, thereby allowing the query framework to access the high-level abstract syntax tree of a program. As mentioned in the thesis introduction, this thesis specifically avoids this approach. Putting database query facilities in a separate tool results in easier maintenance, easier evolution, and faster adoption.

In LINQ, the compiler automatically annotates functions with a high-level intermediate representation. At runtime, the query system can read and manipulate this high-level representation to generate queries. Unfortunately, this high-level representation is limited to expressions only and not general functions. There are no proposals for adding such a compiler annotation to Java.

Instead, Queryll uses bytecode analysis to analyze already compiled Java code. It requires no changes to the Java compiler or Java VM. Queryll uses this analysis to create a high-level representation of the behavior of the code, which allows it to generate database queries at runtime. Because code written in a functional-style has no side-effects, Queryll can use a straight-forward algorithm based on symbolic execution to trace through the effects of running low-level bytecode instructions. Queryll can support complex queries and exhibits good performance.

3.2 Syntax

The key to making functional-style database queries feasible in Java is the addition of anonymous functions to the Java language. At present, there are many different proposals for how this can be done. Fortunately, in all of these proposals, the functions are compiled down to a similar representation. Since Queryll operates on already compiled Java code, the exact syntax of these lambda expressions is not relevant to the design of the algorithm. Nevertheless, to understand how a functional-style query system might work in Java, it is useful to see a possible syntax. This chapter uses the BGGA v0.6 anonymous function syntax [GvdA] to illustrate a possible query syntax. Other proposals for adding functional programming features to Java will result in database query systems with similar characteristics though slightly different syntaxes.

With the BGGA syntax (Figure 3.1), functions are denoted with the hash symbol (`#`), followed by a list of parameters, and then the code of the function. In many situations, a function will simply evaluate an expression and return the result. For these cases, the BGGA syntax allows for a shorter *lambda expression* syntax which is denoted with a hash symbol, followed by function parameters, and ended with the expression to be evaluated and returned by the function. In BGGA, functions can be stored in variables and passed around. The data type of a variable that holds a function is denoted with a hash symbol, followed by the return type of the function, and then the parameters of the function.

Once support for functional programming features is added to Java, it becomes possible to use standard functional syntax for manipulating large collections of data. In standard functional languages, collections can be manipulated with operations such as `map`, which remaps each collection entry into a different value, or `filter`/`find_all`, which filters out collection entries that satisfy certain restrictions. These operations take a collection and a function as parameters. Each item in the collection is iterated over, and a new collection is created by evaluating each item using the supplied function. The new collection then becomes the result of the operation.

This convention can be used to make database query operations in Java that directly

Anonymous Function Syntax
#(parameters) {statements}
e.g. `#(int x, int y) { return x+y; }`

Lambda Expression Syntax
#(parameters) expression
e.g. `#(int x, int y) x+y`

Function Type Syntax
#returnType(parameters)
e.g. `#int(int x, int y) variable = #(int x, int y) x+y;`

Figure 3.1: An overview of the BGGA v0.6 [GvdA] syntax used in this chapter

```
class QueryList<T> implements List<T> {
   ...
   public <U> QueryList<U> select(#U(T value) f) ...
   public QueryList<T> where(#boolean(T value) f) ...
}
```

Figure 3.2: Method signatures for the select and where methods of a QueryList

correspond to existing SQL operations. Special `Collection` classes can be created that have extra methods for manipulating the collection data. For example, one could define a QueryList class with `select` and `where` methods, corresponding to SQL's SELECT and WHERE operations. These methods take an anonymous function as a parameter and depending on the semantics one wants, these methods can either return a new collection or an iterator. Figure 3.2 shows the method signatures for `select` and `where` methods. The `select` method iterates over all the elements of the collection. Each element is, in turn, passed as a parameter to the supplied function `f`, and the results are stored in a new collection. Finally, the `select` method returns the new collection. Similarly, the `where` method iterates over all the elements of the collection, but it only adds elements to the new collection if the function `f` returns true.

Figure 3.3 shows how a simple database query could be expressed using these `select` and `where` methods. Figure 3.4 shows an equivalent SQL query. The query uses an object `db`, which needs to be generated by an ORM tool. This object has methods such as `getCustomers()` that returns a QueryList of all the Customer records in a database. Programmers can then invoke `select` and other methods on this QueryList to define their query.

In this syntax for queries, the anonymous functions passed to the `select` and `where` methods should not contain any complex control-flow structures such as loops. The control flow graph can be in the form of an arbitrary directed-acyclic graph though. The functions also cannot have any side-effects since it is not possible to recreate this side-effect behavior

```
QueryList<String> results =
   db.getCustomers()
      .where(#(Customer c) c.getCountry().equals("UK"))
      .select(#(Customer c) c.getName() );
```

Figure 3.3: Simple database query in Java using lambda expressions

```
SELECT C.Name
  FROM Customer C
 WHERE C.Country = 'UK'
```

Figure 3.4: This SQL query is equivalent to the query in Figure 3.3

using a database query. The only changes in program state caused by execuating a function should be for the function to return a value. In particular, functions ...

- Can call other methods, but only those from a restricted list with known side-effects
- Can read and modify local variables (since these changes will be discarded once the function exits)
- Can read but not modify non-local variables
- Can instantiate certain known classes if their constructors are known to be safe

These restrictions result in queries with reasonable expressiveness while being fairly straight-forward. As such, programmers can easily determine whether their queries satisfy the syntax and will be translated in database queries correctly. The restrictions also simplify the translation process.

This syntax for queries in Java also supports query parameters. This is expressed by having the anonymous functions make use of variables defined outside of their scope. These variables can be fields of other objects or final local variables (Figure 3.5).

3.2.1 Complex Queries

The syntax can be extended to handle complex queries. The important operations that a relational query system must support are selection, projection, join, aggregation, duplicate

```
final String country = "UK";
QueryList<Customer> results =
   db.getCustomers()
      .where(#(Customer c) c.getCountry().equals(country));
```

Figure 3.5: A database query in Java that makes use of parameters

removal, nested queries, set operations, sorting, and limiting (Appendix A). Other relational operations can then be expressed using combinations or simple variations of these basic operations. A convenient syntax for grouping operations is also desirable, given the frequency of their use. Queryll supports these operations by adding methods to the QueryList collection class.

Selection

The basic operation of most queries involves selecting a subset of a dataset to examine. As demonstrated earlier, this can be expressed by taking an initial set of data and then filtering the data with a boolean expression.

```
db.getCustomers().where(#(Customer c) (c.getName().equals("Bob")));
```

Projection

A query may only need certain fields from a record. It may also be necessary to construct new data structures to hold these fields. Queryll supplies a Pair object that can hold two arbitrary values. Similar to a LISP list which also holds only two values (car and cdr), Pair objects can be chained together to construct simple data structures during a query. Queryll also provides Tuple objects as a convenience for programmers who want to create simple fixed size n-tuples. This ability to create new data structures is equivalent to using projection operations to create new columns for database relations or to remove columns from database relations. The example below iterates over the Customer entities in a database and creates a new Collection consisting of only the first names and last names of these customers.

```
db.getCustomers().select(#(Customer c) (
   new Pair<String, String>(c.getFirstName(), c.getLastName())));
```

Projection operations themselves are not directly expressible in Queryll, as doing so would mean that Queryll would have to support the creation of new classes at runtime. Java only allows classes to be created at runtime through complicated bytecode rewriting schemes, and forcing programmers to statically declare special classes for holding their query results is quite verbose and cumbersome. Queryll's use of Pair objects to provide power equivalent to projection is much more consistent with existing Java syntax.

Another important aspect of the expressiveness of SQL is its CASE WHEN...ELSE...-END statements. These statements allow SQL to conditionally return different values from a query. Queryll's `select` method may contain control-flow statements, which give equivalent expressiveness to SQL's CASE WHEN...ELSE...END statements.

```
   db.getCustomers().select(#(Customer c) {
      if (c.getCountry().equals("US"))
         return "US";
      else
         return "Other";
      });
```

Join

Arbitrary full cross-joins between different tables can be expressed by taking one QueryList and calling a `join` method. This join method iterates through each element of the list and passes this element to a supplied function. This function returns a QueryList of objects that should be joined with that element. This will generate a new QueryList filled with Pair objects of all combinations of elements. This new QueryList can then be further queried.

```
db.getCustomers().join(#(Customer c){db.getAccounts()})
   .where(#(Pair<Customer, Account> p) (p.getFirst().getID() == 10))
   .where(#(Pair<Customer, Account> p)
      (p.getFirst().getID() == p.getSecond().getCustomerID()))
   .select(#(Pair<Customer, Account> p) (p.getSecond()));
```

In the above example, each customer is arbitrarily joined with all of the accounts in the database. This results in a collection of Customer-Account pairs. These pairs are then filtered. The method `getFirst()` is called on the Pair object, returning the first element of the pair, the Customer object. Those pairs where the customer does not have an id of 10 are filtered out. Then the pairs are filtered a second time. This filtering produces a result set of only those Customer-Account pairs where the account belongs to the corresponding customer. Finally, a projection operation is performed that restricts the result set to only the account information of the Pair objects.

Although this syntax for joins can be used to express arbitrary joins, it can be verbose, especially for common joins. Fortunately, since the programmer must describe the relationship between entities to the underlying ORM tool of Queryll, Queryll is able to generate methods for navigating among objects, and these methods can be used during queries. These methods can simplify common join operations. When a query navigates over a 1:1 or N:1 relationship between entities, Queryll translates the query into a cross join, a selection constraint on the join, and then the operation described by the query.

When a query navigates 1:N or N:M relationships between entities, the programmer must either use an aggregation operation to reduce the multiple related entities to a single value or the programmer must use the `join` method for this purpose.

```
   // an aggregation operation over a 1:N relationship returning a scalar
   // value
   db.getCustomers()
      .where(#(Customer c) (c.getAccounts().size() > 3));

   // a 1:N join expressed using a join operation
   db.getCustomers()
      .join(#(Customer c) (c.getAccounts()));
```

Aggregation

To support common SQL aggregation operations, the QueryList collection has methods for calculating aggregate values over the objects in the collection. For example, in the query below, the `sumDouble()` method iterates over a collection of Order objects and calculates a sum of double-precision floating point values. The method takes a function which takes an Order object and returns the double value to be summed.

```
db.getOrders()
   .sumDouble(#(Order o) (o.getTotalValue()));
```

To calculate multiple aggregate values, a special selection method is available. The `selectAggregates()` method iterates over a collection and returns a pair or other tuple, where each value of the tuple is the result of an aggregation operation.

```
db.getOrders()
   .where(#(Order o)( o.getTotalValue() > 1000))
   .selectAggregates(#(QueryList<Order> oo)
      (new Pair<Integer, Double>
                (oo.size(),
                 oo.SumDouble(#(Order o) (o.getTotalValue())))));
```

Duplicate Removal

A method called `unique()` returns a copy of the list with all duplicate entries removed.

```
db.getCustomers()
   .select(#(Customer c) (c.getCountry()))
   .unique();
```

Nested Queries

Since operations on QueryList collections return new QueryList objects, operations can be chained together or joined together to provide one form of nesting.

```
db.getOrders()
   .where(#(Order o) (o.getTotalValue() > 1000))
   .select(#(Order o) (o.getCustomer())
   .asSet()
   .where(#(Customer c) (c.getAccounts().size() > 5));
```

Calculating aggregate values can convert a QueryList into a scalar value, which allows queries to be nested inside operations that take single values. For example, the nested query below counts the number of accounts belonging to each customer in the UK. It uses a nested aggregation operation inside a projection operation to calculate the number of accounts belonging to each customer.

```
db.getCustomers()
      .where(#(Customer c) c.getCountry().equals("UK"));
      .select(#(Customer c) c.getAccounts().sum(#(Account a) 1) );
```

Grouping

Although grouping operations can be expressed using nested queries, the frequency of grouping operations in queries demands some syntactic sugar to make such operations easier to express. In Queryll, a grouping operation takes two parameters, one is a function that returns the keys to group by, and the other is a function that returns aggregates on the keys and associated values, as in `selectAggregates`.

```
db.getCustomers()
   .group( #(Customer c) (c.getCountry()),
           #(String country, QueryList<Customer> cc) (cc.Count()));
```

Set Operations

To support set operations, the QueryList has method corresponding to SQL's UNION, INTERSECT, and EXCEPT set operations. In the example below, the `except()` method is used to subtract the set of customers from the UK from the full set of customers. As a result, it returns the set of customers who are not from the UK.

```
db.getCustomers()
   .except(db.getCustomers()
              .where( #(Customer c) (c.getCountry().equals("UK")) );
```

Sorting and Limiting

Finally, a query may want its results sorted or to have only partial results returned. Sorting is supported by letting programmers pass in a Comparator function which describes which fields should be compared. Returning partial results can be support using a method where programmers can pass in the number of results they desire.

```
db.getOrders()
   .select(#(Order o) (o.getCustomer());
```

Figure 3.6: If `o.getCustomer()` can throw an exception, the exception will propagate out of the anonymous function and will be handled inside the `select` method

```
try {
   QueryList<Customer> results = db.getOrders()
      .select(#(Order o) (o.getCustomer());
   for (Customer c: results) { ... }
} catch (QueryException e) {}
```

Figure 3.7: If this query is run directly, then if `o.getCustomer()` throws an exception, the exception will propagate outwards to the outer exception handler. Conveniently, if the query is translated the SQL, exceptions from the generated SQL can be caught with the same exception handler

```
top10Accounts = db.allAccounts
   .sortedByDoubleDescending(#(Account a) (a.getBalance()));
   .firstN(10);
```

3.2.2 Exceptions

Since queries need to access a database, communication and database exceptions may occur, and these exceptions need to be signaled to the program. The issue of exceptions must be addressed at two levels: at the ORM level and at the generated query level.

Generated ORM objects may need to access the database when certain fields are accessed or when certain navigational links are followed. These database accesses may throw exceptions. As a result, a query may involve invoking methods on ORM objects that throw exceptions. Handling these exceptions inside the query itself is verbose, and the exception handling code has no meaning if the entire query is translated to SQL. As such, the query API should let ORM exceptions propagate out of anonymous functions and into the collection methods (Figure 3.6).

If a query is translated into SQL, an exception may occur when executing the generated SQL. To allow the programmer to handle this situation, the query methods should be marked as potentially throwing exceptions. This is convenient because it provides a single, consistent place where programmers can catch exceptions when writing queries, regardless of whether the query is translated into SQL or simply run in-memory (Figure 3.7). If a query is translated to SQL, then query methods can throw exceptions signaling problems with this SQL. If a query is not translated to SQL and the anonymous functions passed to a query method are executed directly, any exceptions from these anonymous functions can be caught inside the query method and rethrown to be handled outside the query. The same exception handler can be used for both cases.

3.2.3 Iterators vs. Collections

So far, the syntax description has shown how collections of database records can be represented and manipulated as Java `Collections`. This syntax is consistent with the existing conventions used in functional programming for working with large sets of data in memory. One alternative is to represent collections of database records as iterators instead. Query methods like `select` and `where` for manipulating these records can be added to the iterators instead of a `Collection` object. Such an approach is used by LINQ. Although manipulating iterators instead of collections deviates from standard functional programming conventions, it does have some advantages. Some database queries return results which are too large to fit into memory and can only be streamed through. When using a collections approach, large result sets must either be disallowed or the need to stream them must somehow be hidden behind an abstraction. When using a iterator approach, all result sets are represented as a stream, so no special handling is needed for large result sets.

3.2.4 Limitations

One important aspect of SQL that is not addressed by Queryll is support for NULL values and related operators. Since Java is a Turing-complete language, nothing precludes Java from supporting NULL values. Unfortunately, since Java does not support three-value logic or operator overloading, providing the same semantics for NULL as SQL does would be extremely verbose. Solving this problem is outside the scope of Queryll, but if a solution to this problem is eventually found, Queryll can be easily adapted to support it.

3.3 Translation Algorithm

The main challenge in translating these Java queries into SQL is in deciphering the operations performed by the anonymous functions. In all the proposals for adding support for functional programming to Java, anonymous functions are compiled down into separate classes at the bytecode level[1] [LLB, CS, BGGvdA]. For example, the sample query in Figure 3.3 can be compiled down into the classes shown in Figure 3.8. The different syntaxes for functions then become irrelevant because all anonymous functions are compiled down to normal Java classes and methods regardless of syntax.

The translation algorithm operates at the bytecode level. This design allows it to be independent of Java compilers, IDEs, and virtual machines. Queryll can be added to an existing software project without requiring programmers to adopt a new compiler or to use a special debugger. Programmers are free to adopt new tools without worrying if these tools are compatible with Queryll. Queryll is also designed to use only bytecode analysis. By not using any bytecode rewriting, the Queryll implementation becomes vastly simpler

[1]More recent proposals for anonymous functions in Java have suggested extending the Java virtual machine with new instructions that support direct references to methods of classes [Goe10]. The translation algorithm can also handle Java code that has been compiled to use this functionality.

```
class Where1 implements Lambda {
   public boolean call(Customer c) { return c.getCountry().equals("UK"); }
}
class Select1 implements Lambda {
   public String call(Customer c) { return c.getName(); }
}

QueryList<String> results =
   db.getCustomers()
      .where( new Where1() )
      .select( new Select1() );
```

Figure 3.8: Simple database query in Java with lambda expressions expanded into lower-level classes

and all the components of Queryll can be traced through in a debugger (unlike bytecode generated by a bytecode rewriter).

This section describes

- How Queryll finds anonymous functions to analyze
- The bytecode analysis algorithm
- Query generation
- How query parameters and nested queries are handled

3.3.1 Finding Anonymous Functions

The translation must first choose which pieces of code to analyze. In the worst case, Queryll can simply analyze the bytecode of every class file used by a program, but all this analysis would slow down the startup time of the program.

Depending on how anonymous functions are eventually implemented in Java, Queryll can use different approaches for narrowing down the number of classes it must analyze:

- Since functions do not currently exist in Java, a common substitute is to define an interface containing only a single method. If a programmer wants a method to take a function as an argument, they can use an interface as an argument instead. It has been proposed that future versions of Java will allow programmers to pass anonymous functions to methods that accept such interfaces, and Java will automatically convert the function into a class implementing the appropriate method. If this occurs, Queryll can define its query methods to accept interfaces, and it can narrow down its bytecode analysis to only those classes that implement one of these special interfaces

```
QueryList<Office> results =
   db.getOffices()
      .where(#(Office o) o.getName().equals("UK")
         || o.getName().equals("US"));
```

Figure 3.9: A simple query that will be translated into SQL

- Anonymous functions may support programmer annotations, so programmers can annotate their functions to flag them for analysis by Queryll
- Queryll can cache a copy of all the code of all the classes of a program. It can then perform its bytecode analysis when actual classes are created and passed to Queryll for building queries.
- It's possible that anonymous functions may be implemented in such a way that anonymous functions are obscured by opaque proxy objects. To handle such a situation, Queryll would require static dataflow analysis of all code that makes use of Queryll to understand where functions are proxied and how these proxies eventually propagate to Queryll.

3.3.2 Anonymous Function Analysis

Once the anonymous functions used in a query are found, these functions can then be analyzed. These anonymous functions are compiled down to classes with a method containing the code of the function. For example, the anonymous function from the query in Figure 3.9 might be translated into the bytecode shown in Figure 3.10. Different compilers may generate slightly different bytecode from the same Java code. Since Queryll operates at the bytecode level, it must be tolerant of these variations. It employs symbolic execution to convert low-level bytecode instructions back into high-level expressions. Since the anonymous functions accepted by Queryll do not contain loops and are not supposed to have any side-effects, this conversion can be done using a fast and efficient algorithm.

Firstly, Queryll verifies that the code does not contain any complex control flow nor contain any side-effects. Checking for the presence of loops can be done by simply performing a depth-first search walk of the control flow graph from the head of the function and noticing if there are any backwards edges. The detection of side-effects can be performed by ensuring that each instruction of the code does not have side-effects (i.e. modification of non-local variables, calls to unknown methods, etc.).

Queryll then interprets what sort of query is being performed in the code. Since the code might contain many variables and branching instructions, it can be difficult to understand the code. To avoid this problem, the code is broken down into straight paths during the analysis. The control flow graph can be walked, and every path leading from the code entry point to a return statement are noted. The instructions that form a path are then treated as a straight-line piece of code. Analyzing straight-line code is much easier because it is easy to calculate both the values of variables at any point in the code

```
 1:  aload_1
 2:  invokevirtual Office.getName:()Ljava/lang/String;
 3:  ldc "US"
 4:  invokevirtual java/lang/String.equals:(Ljava/lang/Object;)Z
 5:  ifne 11
 6:  aload_1
 7:  invokevirtual Office.getName:()Ljava/lang/String;
 8:  ldc "UK"
 9:  invokevirtual java/lang/String.equals:(Ljava/lang/Object;)Z
10:  ifeq 13
11:  iconst_1
12:  goto 14
13:  iconst_0
14:  ireturn
```

Figure 3.10: Java bytecode instructions of the query from Figure 3.9

and dependencies between any instructions. Table 3.1 shows the three paths that exist in the bytecode from Figure 3.10.

Symbolic execution is then used for converting the instructions along each path into a higher level representation. Queryll starts at the first instruction of a path, and then executes each instruction of the path using abstract values instead of real concrete values. For example, if it sees an instruction for adding values `a` and `b` together, instead of actually adding those two values, Queryll will use the expression `a + b` as the result of the operation. Similarly, instead of storing numbers and objects on the execution stack and in local variables, Queryll will store symbolic expressions there. When branch instructions are encountered, they are encoded as conditions for the path. When the symbolic execution reaches the last instruction along a path, it will have generated an expression representing the value returned by the anonymous function. Table 3.2 shows the process of symbolically executing the first path from Table 3.1. Since this path is used by a `where` method, Queryll is primarily interested in when a path returns true (i.e. which records are not filtered out). Queryll represents this by generating an expression for when the return value is `1` and the path conditions are true.

Because Java bytecode instructions for conditional GOTOs can only work with conditions involving integers (Java bytecode does not have a boolean data type), the resulting expression may contain redundant comparisons. These extra comparisons can confuse some SQL implementations, so Queryll always performs a simplification step on the final expression to remove them.

Alternate Formulation

The previous algorithm works well for raw Java bytecode, but there are bytecode frameworks that work with other code representations. For example, many optimizations are

Table 3.1: There are three paths through anonymous function

Path 1		Path 2		Path 3	
1:	aload_1	1:	aload_1	1:	aload_1
2:	Office.getName()	2:	Office.getName()	2:	Office.getName()
3:	ldc “US”	3:	ldc “US”	3:	ldc “US”
4:	String.equals(...)	4:	String.equals(...)	4:	String.equals(...)
5:	ifne 11	5:	ifne 11	5:	ifne 11
	branch taken		branch not taken		branch not taken
11:	iconst_1	6:	aload_1	6:	aload_1
12:	goto 14	7:	Office.getName()	7:	Office.getName()
14:	ireturn	8:	ldc “UK”	8:	ldc “UK”
		9:	String.equals(...)	9:	String.equals(...)
		10:	ifeq 13	10:	ifeq 13
			branch taken		branch not taken
		13:	iconst_0	11:	iconst_1
		14:	ireturn	12:	goto 14
				14:	ireturn

Table 3.2: State of the execution stack and of path conditions when Path 1 from Figure 3.1 is symbolically executed

Path 1	Stack	Conditions
1: aload_1	0: $arg0	
2: Office.getName()	0: $arg0.getName()	
3: ldc “US”	0: “US”	
	-1: $arg0.getName()	
4: String.equals(...)	0: $arg0.getName() = “US”	
5: ifne 11		($arg0.getName() = “US”) != 0
branch taken		($arg0.getName() = “US”) != 0
11: iconst_1	0: 1	($arg0.getName() = “US”) != 0
12: goto 14	0: 1	($arg0.getName() = “US”) != 0
14: ireturn	Returned Value: 1	($arg0.getName() = “US”) != 0
Final Expression	1=1 AND ($arg0.getName() = “US”) != 0	
Simplification	$arg0.getName() = “US”	

```
            1:  $r2 = $o.<Office: String getName()>();
            2:  $z0 = $r2.<String: boolean equals(Object)>("UK");
            3:  if $z0 != 0 goto label0;

            4:  $r3 = $o.<Office: String getName()>();
            5:  $z1 = $r3.<String: boolean equals(Object)>("US");
            6:  return $z1;

label0:  7:  return 1;
```

Figure 3.11: A possible Jimple representation of the query from Figure 3.9

Table 3.3: There are two paths through anonymous function

Path 1		Path 2	
1:	$r2 = $o.getName()	1:	$r2 = $o.getName()
2:	$z0 = $r2.equals("UK")	2:	$z0 = $r2.equals("UK")
3:	if $z0 != 0 goto label0 (branch not taken)	3:	if $z0 != 0 goto label0 (branch taken)
4:	$r3 = $o.getName()	7:	return 1
5:	$z1 = $r3.equals("US")		
6:	return $z1		

easier to implement when using a three-address form. Although normal symbolic execution still works when instructions are represented in these alternate forms, a slightly different formulation may be more efficient and easier to implement.

This section will now describe an alternate formulation of the function analysis algorithm. This formulation is appropriate for code in a three-address form, such as Jimple [VRCG$^+$99], a three-address form of Java bytecode. Figure 3.11 shows a possible Jimple representation of the anonymous function used in the query shown in Figure 3.9.

In this alternate formulation, the code of the anonymous function is still broken down into different paths, and each path is analyzed separately. Table 3.3 shows the two paths through the function from Figure 3.11.

For each path, Queryll needs to determine the value returned by the function if that path is followed and the conditions that need to hold for that path to be followed. In the case of the function passed to the `where` method in Figure 3.11, Queryll is interested in determining when the function returns true. Instead of symbolically executing the path to determine this, this formulation involves iterating backwards over the instructions. For each path, Queryll starts at the last instruction and walks backwards over each instruction. As Queryll performs this walk, it reconstructs expressions representing the returned value and path conditions.

If Queryll encounters an instruction returning a value, it stores which value is returned. If it encounters a conditional branch instruction, it merges this branch condition into the path condition expression with an AND operation. The variables in these expressions will be made up of mostly local variables. If Queryll encounters an instruction that makes an assignment to one of these local variables, it goes through the returned value expression and path condition expression, and it replaces all of the instances of the local variables with the value assigned to the local variable in the instruction. Unlike with symbolic execution, this formulation only needs to keep track of the returned value and path conditions; it does not need to store the value of all local variables or an execution stack.

When Queryll finishes walking through all the instructions, the resulting expressions should be made up of operations acting on constants, outside variables, or entries from the source collection. For example, if Queryll was trying to construct an expression to describe the paths of Table 3.3, it would go through the steps shown in Table 3.4. The expressions for the returned value and path conditions can then be merged into a final expression that can then be used in query generation.

Table 3.4: For a given path, Queryll can construct an expression that describes when the path is executed

Instruction	Returned Value	Conditions
Initial		
6: return $z1	$z1	
5: $z1 = $r3.equals("US")	($r3 = "US")	
4: $r3 = $o.getName()	($o.Name = "US")	
3: if $z0 != 0 goto label0 (branch not taken)	($o.Name = "US")	$z0 = 0
2: $z0 = $r2.equals("UK")	($o.Name = "US")	($r2 = "UK") = 0
1: $r2 = $o.getName()	($o.Name = "US")	($o.Name = "UK") = 0
Final Expression	($o.Name = "US") AND ($o.Name = "UK") = 0	
Simplification	(entry.Name = "US") AND (entry.Name != "UK")	

Instruction	Returned Value	Conditions
Initial		
7: return 1	1	
3: if $z0 != 0 goto label0 (branch taken)	1	$z0 != 0
2: $z0 = $r2.equals("UK")	1	($r2 = "UK") != 0
1: $r2 = $o.getName()	1	($o.Name = "UK") != 0
Final Expression	(1 = 1) AND ($o.Name = "UK") != 0	
Simplification	entry.Name = "UK"	

3.3.3 Runtime Query Construction

It is easier to translate functional-style Java queries into SQL code at runtime instead of statically. Although static SQL query generation is possible, it requires deeper code

$$Q \vdash \begin{array}{l}\texttt{db.getCustomers()}\\ \quad\texttt{.select(\#(Customer c)\{c.getName()\})}\\ \quad\texttt{.where(\#(String name)\{name.equals("Bob")\})}\end{array} \Downarrow ?$$

Figure 3.12: An example Java query that will be used to illustrate how runtime query generation in Queryll

analysis that is less flexible and hence more restrictive on how programmers write their queries. Also, statically inserting the generated SQL query into Java code requires a bytecode rewriting framework; whereas, runtime SQL query generation does not need to modify existing code, so only a much simpler bytecode analysis framework is needed.

With runtime query generation, queries are built up inside query methods, like `select` and `where`. When these query methods are invoked with anonymous functions as parameters, the query methods can look up the static bytecode analysis results for these anonymous functions and construct a SQL query. Since a programmer must call multiple query methods to build up a full SQL query, the generated SQL should not be executed on a database. The generated SQL should only be executed lazily when the programmer actually tries to access the data because then Queryll can be sure that the programmer has finished specifying their query. As a result, each of the special Queryll `Collection` objects will have an associated SQL query. Whenever data is accessed from the `Collection`, the associated SQL query will be executed and the `Collection` populated with the query result. Invoking query methods on the `Collection` will return a new `Collection` with a different associated SQL query.

The general approach used for runtime query construction will be illustrated using a simple example. Figure 3.12 shows a simple query. Translation mapping Q is used to denote the mapping of how Java code is translated into a SQL representation.

The query can be broken into three method calls (Figure 3.13). The first call to `db.getCustomers()` returns all of the Customer records from the database, `select()` discards everything except the name field of each record, and `where()` restricts the name field to only those called "Bob."

Because Queryll needs to store the SQL representations that underlie each query `Collection`, it requires different data structures for all the different types of SQL queries. All of the queries in the example can be expressed using SELECT...FROM...WHERE... SQL queries. To represent a SELECT...FROM...WHERE... query, Queryll needs a data structure that stores four pieces of information: the column values that should appear in the SELECT clause, the table being queried, where restrictions for filtering the table, and a description of how to convert the returned columns of a result set into Java objects. Queryll stores these four pieces of information using the 4-tuple $\mathsf{SFW(columns, from, where, reader)}$.

In the example, Customer records are assumed to have three fields—id, name, and address—so `db.getCustomers()` returns a $\mathsf{SFW()}$ tuple for reading these columns from a Customer table. `select()` takes this $\mathsf{SFW()}$ tuple, looks up the symbolic execution

$$\frac{}{\begin{array}{l}Q \vdash \texttt{db.getCustomers()} \Downarrow \\ \qquad \mathsf{SFW}(\langle \textbf{Id}, \textbf{Name}, \textbf{Address}\rangle, \textbf{Customer}, \textbf{1=1}, \textsc{CustomerReader})\end{array}}$$

$$\frac{S \vdash \langle query, \mathsf{@arg1.getName()}\rangle \Downarrow newquery}{Q \vdash query\texttt{.select(\#(Customer c)\{c.getName()\})} \Downarrow newquery}$$

$$\frac{W \vdash \langle query, \mathsf{@arg1} = \text{“}\mathsf{Bob}\text{”}\rangle \Downarrow newquery}{Q \vdash query\texttt{.where(\#(String name)\{name.equals("Bob")\})} \Downarrow newquery}$$

Figure 3.13: The query in Figure 3.12 is broken down into three method calls

$$\frac{\Sigma \vdash \langle fun, cols, reader\rangle \Downarrow \langle newcols, newreader\rangle}{\begin{array}{l}S \vdash \langle \mathsf{SFW}(cols, from, where, reader), fun\rangle \Downarrow \\ \qquad \mathsf{SFW}(newcols, from, where, newreader)\end{array}}$$

$$\frac{\Sigma \vdash \langle fun, cols, reader\rangle \Downarrow \langle\langle newwhere\rangle, \textsc{BoolReader}\rangle}{\begin{array}{l}W \vdash \langle \mathsf{SFW}(cols, from, where, reader), fun\rangle \Downarrow \\ \qquad \mathsf{SFW}(cols, from, where\ \textbf{AND}\ newwhere, reader)\end{array}}$$

Figure 3.14: S and W apply `select()` and `where()` operations respectively to a SELECT...FROM...WHERE... query by creating a new SELECT...FROM...WHERE... query with different columns or a modified WHERE clause

analysis of the given anonymous function, and passes everything to a S mapping for further analysis. Similarly, the `where()` query method takes the query generated by `select()`, looks up the symbolic execution analysis of the supplied anonymous function, and delegates further processing to a W mapping.

The mappings S and W both perform similar processing (Figure 3.14). They take a SELECT...FROM...WHERE... query and apply a projection or selection operation to the query, generating a new SELECT...FROM...WHERE... query. The S mapping, used by the `select()` query method, will generate new columns and a new reader for the new query. The W mapping, used by the `where()` query method, will generate a new WHERE clause for the new query. Both S and W make use of a mapping Σ. Σ takes as input the symbolic execution expression calculated for the anonymous function plus information about the original query being modified. It calculates the effect of applying the anonymous function to the original query and expresses the result in terms of a tuple of column values and a description of how to interpret these column values.

The Σ mapping simply finds SQL equivalents to the operators that appear in the previously calculated symbolic execution expressions (Figure 3.15). References to the argument of an anonymous function (i.e. records from the original query that are being iterated over) are replaced with appropriate values from the original query.

Finally, a SQL query represented as a $\mathsf{SFW}()$ can be mapped into an actual SQL query string using a mapping G (Figure 3.16).

$$\frac{\Sigma \vdash \langle left, cols, reader \rangle \Downarrow \langle \langle leftexpr \rangle, exprreader \rangle \quad \Sigma \vdash \langle right, cols, reader \rangle \Downarrow \langle \langle rightexpr \rangle, exprreader \rangle}{\Sigma \vdash \langle left{=}right, cols, reader \rangle \Downarrow \langle leftexpr{=}rightexpr, \textsc{BoolReader} \rangle}$$

$$\frac{}{\Sigma \vdash \langle \mathsf{@arg1}, cols, reader \rangle \Downarrow \langle cols, reader \rangle}$$

$$\frac{}{\Sigma \vdash \langle \text{“}\mathsf{Bob}\text{”}, cols, reader \rangle \Downarrow \langle \langle \textbf{“Bob”} \rangle, \textsc{StringReader} \rangle}$$

$$\frac{\Sigma \vdash \langle expr, cols, reader \rangle \Downarrow \langle \langle newcol_1, newcol_2, newcol_3 \rangle, \textsc{CustomerReader} \rangle}{\Sigma \vdash \langle expr.\mathsf{getName()}, cols, reader \rangle \Downarrow \langle \langle newcol_2 \rangle, \textsc{StringReader} \rangle}$$

Figure 3.15: Σ finds SQL equivalents to the expressions computed by symbolic execution

$$\frac{Q \vdash java \Downarrow \mathsf{SFW}(\langle col_1, col_2, \ldots \rangle, from, where, reader)}{G \vdash java \Downarrow \textbf{SELECT } col_1, col_2, \ldots \textbf{ FROM } from \textbf{ WHERE } where}$$

Figure 3.16: When a `Collection` is accessed, the underlying SFW() query will be converted into a SQL query string and executed on the database

3.3.4 Complex Queries

Query Parameters

The anonymous functions used in Queryll queries may refer to variables outside the scope of the function. These references are treated as query parameters by Queryll. Constructing queries at runtime allows for the easy handling of these query parameters. When an anonymous function makes a reference to a static variable, query methods like `where` and `select` can look-up the values of these static variables and store them in the generated query. For other types of variables, such as in the example shown in Figure 3.5, the Java compiler will store the values of these variables in the anonymous function objects themselves. Java will generate a constructor for these anonymous function objects that take a value for these variables and store them in a field (Figure 3.17). Query methods can simply read the values of these fields in the anonymous function object and store them in the generated query.

Nested Queries

Because the anonymous functions used in Queryll nested queries do not have complex control flow nor side-effects, they can be analyzed using the same techniques used for non-nested queries. There is one additional complication involving query parameters though. In the non-nested case, the query generator could rely on parameters being stored in the fields of anonymous function objects. This is not possible with nested queries because the inner-nested anonymous functions are only instantiated when the outside anonymous functions are run. When the code is translated into a database query, the outside anony-

```
class Where1 implements Lambda {
   final String country;
   public Where1(final String country) {
      this.country = country;
   }
   public boolean call(Customer c) {
      return c.getCountry().equals(country);
   }
}

final String country = "UK";
QueryList<Customer> results =
   db.getCustomers()
      .where( new Where1(country) );
```

Figure 3.17: The query parameter from Figure 3.5 is compiled by Java into a variable passed to a constructor where it is stored in a field. The `where` method can access this field to read the query parameter

mous functions are never executed, so the inner-nested functions are never instantiated and the values of fields extracted.

Instead, Queryll must separately analyze the constructors of these inner-nested anonymous function objects to see where parameters passed in to the anonymous function objects are stored in fields. Then Queryll can map the usage of fields in anonymous functions to parameters passed in to the constructors.

3.4 Implementation

A prototype implementation of Queryll has been constructed. Since the Queryll syntax requires support for anonymous functions, the experimental OpenJDK7 b105 release with Lambda patches from September 6, 2010 was used for the implementation. This version of Java contains some early support for anonymous functions. The Queryll prototype uses the ASM 3.3 [BLC02] library for its bytecode analysis. The prototype does not yet implement exceptions and set operations. It also only supports scalar nested queries without query parameters. It does not verify that there no side-effects in the constructors of those nested queries, and it does not include pointer aliasing support.

3.5 Experiments

For Queryll to be a practical query system, programmers must be able to encode real-life queries in the system, and these queries must exhibit reasonable performance when run. To evaluate these properties, the database queries from the TPC-W benchmark [Tra02] were taken and adapted to run using Queryll.

TPC-W emulates the behavior of database-driven websites by recreating a website for an online bookstore. The experiments use the Rice implementation of TPC-W [ACC+02], which uses JDBC/SQL to access a database. Queryll focuses on database queries only and not data manipulation, so only the database queries of the benchmark were used. In particular, the experiments do not include database updates, transactions, persistence lifecycle, or application server code. For each query, an equivalent query was written in Queryll. The SQL generated from the Queryll versions of the query were manually verified to be comparable to the SQL versions of the query. The performance of the JDBC version and Queryll version could then be compared.

A 600 MB database in PostgreSQL 8.3.0 [Pos] was created by populating the database with the number of items set to 10000. Each query was first executed 200 times with random valid parameters to warm the database cache, then the time needed to execute the query 3000 times with random valid parameters was measured, and finally the system was garbage collected. A single run of the benchmark consists of alternately running each query using both JDBC and Queryll. The benchmark was run 30 times, and the averages of only the last 10 runs were included in the final results. The database and the query code were both run on the same machine, a 2.5 GHz Pentium IV Celeron Windows machine with 1 GB of RAM. The symbolic execution component of Queryll is only run once at the start of the benchmark. This component required 766 milliseconds to scan through the 342 class files of the benchmark and process the 46 of them used in queries.

Table 3.5 shows the results of the experiment. All of the TPC-W database queries were successfully expressed as Queryll queries. This demonstrates that Queryll approach is capable of handling real-world database queries. Hand inspection of the SQL generated by Queryll shows the generated SQL to be structurally similar to the hand-written SQL. Overall, the performance of Queryll seems reasonable. The use of Queryll does impose some small overhead over hand-written SQL though. A deeper investigation into the causes of this overhead shows that it accumulates from many small inefficiencies such as

- Queryll generates SQL that is more verbose than hand-written SQL because it carefully provides aliases for every table and column to avoid ambiguity. This extra verbosity takes longer for the SQL driver to parse and process. This overhead can be reduced though PreparedStatement caching where the SQL driver parses queries into an intermediate form, and that intermediate form can be reused for subsequent queries. The Rice JDBC implementation of TPC-W does not use this optimization, so it is also not used in Queryll

- For some queries, extra fields are fetched from the database as compared to hand-written SQL because of inefficiencies in the ORM tool used by Queryll

- Because Queryll generates queries at runtime, it must use an abstraction to handle the setting of query parameters, which imposes some overhead over simply setting them directly

- Similarly, Queryll must use factory objects to read query results into objects whereas with hand-written SQL, the code for reading results can be executed directly

Table 3.5: The average execution time and standard deviation of TPC-W queries are shown in milliseconds. The Queryll with Analysis column includes the time required by Queryll to fully rebuild a SQL query each time a query is executed, thereby giving an indication of the overhead required for runtime query construction. The columns for differences in execution time compares the performance of JDBC and normal Queryll, which caches and reuses its analysis and constructed queries

	JDBC		Queryll		Δ		Queryll with Analysis	
Query	Time	σ	Time	σ	Time	%	Time	σ
getName	3652	38.4	4041	66.9	389	11%	4920	105.1
getCustomer	8441	40.9	9222	61.7	781	9%	11263	189.1
getMostRecentOrder	29147	1626.3	33131	1580.1	3984	14%	42769	7747.4
getBook	6436	60.2	6909	110.8	473	7%	9602	164.8
doAuthorSearch	10442	58.9	10406	181.8	-36	(0%)	12252	196.9
doSubjectSearch	16841	132.9	17067	72.5	227	1%	18447	185.7
getIDandPassword	3873	83.1	4189	87.8	316	8%	5077	74.9
getBestSellers	53135	587.0	53741	403.2	606	1%	57702	349.7
doTitleSearch	26833	231.6	27286	208.1	453	2%	29073	315.1
getNewProducts	23096	308.2	25161	385.8	2065	9%	26747	211.9
getRelated	6381	217.7	8059	164.4	1678	26%	12098	207.6
getUserName	3681	68.6	4005	105.8	324	9%	4769	127.3

- Query generation, factory objects, etc. result in extra memory objects that may reduce cache locality and impose extra garbage collection overhead

Table 3.5 also includes a column Queryll with Analysis which shows the time needed for Queryll to construct its queries at runtime and then to execute them. Although the symbolic execution of anonymous functions is done statically, the actual composition and transformation of these functions into SQL queries occurs at runtime. Most database applications execute the same queries often and repeatedly, so Queryll normally caches and reuses the queries it constructs. To generate the Queryll with Analysis results, Queryll's caching of constructed queries is disabled. These results give an indication of the overhead of runtime query construction for ad hoc queries.

Overall, the TPC-W experiment demonstrates that Queryll can handle real database queries used in real applications. Although there is some inevitable overhead due to the use of a middleware abstraction for executing queries, for the most part, Queryll offers comparable performance to hand-written SQL.

3.6 Summary

Adding support for functional programming to traditional object-oriented languages like Java makes it possible to write database queries in those languages using a syntax similar to common declarative query languages like SQL. This functional-style for writing database queries does not have complex control flow such as loops and the functions describing the query itself do not contain any side-effects. As a result, it is possible to write a simple, robust algorithm for translating Java code written in this style into SQL. Queryll is able to do this translation by building an expression representing the return values of the functions used in the Java code.

Chapter 4

JReq: Imperative-Style Queries

The most popular general purpose programming languages today are object-oriented languages like Java. Because of the imperative nature of these languages, it is difficult to embed database query languages, which tend to be declarative, into these languages in a consistent way. This chapter describes an approach for allowing programmers to write database queries in an imperative style inside the imperative language Java. Queries can be written using the normal imperative Java style for working with large datasets—programmers use loops to iterate over the dataset. The queries are valid Java code, so no changes are needed to the Java language to support these complex queries. To run these queries efficiently on common databases, the queries are translated into SQL using an algorithm based on symbolic execution. These algorithms have been implemented in a system called JReq.

Current techniques for integrating database query support into imperative languages are not yet able to handle complex database queries involving aggregation and nesting. Support for aggregation is important because it allows a program to calculate totals and averages across a large dataset without needing to transfer the entire dataset out of a database. Similarly, support for nesting one query inside another significantly increases the expressiveness of queries, allowing a program to group and filter data at the database instead of transferring the data to the program for processing. JReq is able to handle these constructs.

These are the main technical contributions of this work:

- An approach for expressing complex queries in Java code using loops and iterators is demonstrated. This programming style is called the JReq Query Syntax (JQS).

- An algorithm that can robustly translate complex imperative queries involving aggregation and nesting into SQL is described.

- This algorithm is implemented in JReq and its performance is evaluated.

```
QueryList<String> results = new QueryList<String>();
for (Account a: db.allAccounts())
   if (a.getCountry().equals("UK"))
      results.add(a.getName());
```

Figure 4.1: A more natural Java query syntax

4.1 JReq Query Syntax

The JReq system allows programmers to write queries using normal Java code. JReq is not able to translate arbitrary Java code into database queries, but queries written in a certain style. This subset of Java code that can be translated by JReq into SQL code is called the JReq Query Syntax (JQS). Although this style does impose limitations on how code must be written, it is designed to be as unrestrictive as possible.

4.1.1 General Approach and Syntax Examples

Databases are used to store large amounts of structured data, and the most common coding convention used for examining large amounts of data in Java is to iterate over collections. As such, JReq uses this syntax for expressing its queries. JQS queries are generally composed of Java code that iterates over a collection of objects from a database, finds the ones of interest, and adds these objects to a new collection (Figure 4.1). For each table of the database, a method exists that returns all the data from that table, and a special collection class called a QueryList is provided that has extra methods to support database operations like set operations and sorting.

JQS is designed to be extremely lenient in what it accepts as queries. For simple queries composed of a single loop, arbitrary control-flow is allowed inside the loop as long as there are no premature loop exits nor nested loops (nested loops are allowed if they follow certain restrictions), arbitrary creation and modification of variables are allowed as long as they are scoped to the loop, and methods from a long list of safe methods can be called. At most one value can be added to the result-set per loop iteration, and the result-set can only contain numbers, strings, entities, or tuples. Since JReq translates its queries into SQL, the restrictions for more complex queries, such as how queries can be nested or how variables should be scoped, are essentially the same as those of SQL.

One interesting property of the JQS syntax for queries is that the code can be executed directly, and executing the code will produce the correct query result. Of course, since one might be iterating over the entire contents of a database in such a query, executing the code directly might be unreasonably slow. To run the query efficiently, the query must eventually be rewritten in a database query language like SQL instead. This rewriting essentially acts as an optional optimization on the existing code. Since no changes to the Java language are made, all the code can compile in a normal Java compiler, and the compiler will be able to type-check the query statically. No verbose, type-unsafe data marshaling into and out of the query is used in JQS.

In JQS, queries can be nested, values can be aggregated, and results can be filtered in more complex ways. JQS also supports navigational queries where an object may have references to various related objects. For example, to find the customers with a total balance in their accounts of over one million, one could first iterate over all customers. For each customer, one could then use a navigational query to iterate over his or her accounts and sum up the balance.

```
QueryList results = new QueryList();
for (Customer c: db.allCustomer()) {
   double sum = 0;
   for (Account a: c.getAccounts())
      sum += a.getBalance();
   if (sum > 1000000) results.add(c);
}
```

Intermediate results can be stored in local variables and results can be put into groups. In the example below, a map is used to track (key, value) pairs of the number of students in each department. In the query, local variables are freely used.

```
QueryMap<String, Integer> students =
      new QueryMap<String, Integer>(0);
for (Student s: db.allStudent()) {
   String dept = s.getDepartment();
   int count = students.get(dept) + 1;
   students.put(dept, count);
}
```

Although Java does not have a succinct syntax for creating new database entities, programmers can use tuple objects to store multiple result values from a query (these tuples are of fixed size, so query result can still be mapped from flat relations and do not require nested relations). Results can also be stored in sets instead of lists in order to query for unique elements only, such as in the example below where only unique teacher names (stored in a tuple) are kept.

```
QuerySet teachers = new QuerySet();
for (Student s: db.allStudent()) {
   teachers.add(new Pair(
         s.getTeacher().getFirstName(),
         s.getTeacher().getLastName()));
}
```

In order to handle sorting and limiting the size of result sets, the collection classes used in JQS queries have extra methods for sorting and limiting. The JQS sorting syntax is similar to Java syntax for sorting in its use of a separate comparison object. In the query below, a list of supervisors is sorted by name and all but the first 20 entries are discarded.

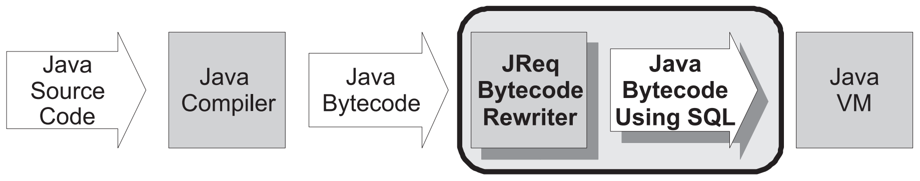


Figure 4.2: JReq inserts itself in the middle of the Java toolchain and does not require changes to existing tools

```
QuerySet<Supervisor> supervisors = new QuerySet<Supervisor>();
for (Student s: db.allStudent())
   supervisors.add(s.getSupervisor());
supervisors
   .sortedByStringAscending(new StringSorter<Supervisor>() {
       public String value(Supervisor s) {return s.getName();}})
   .firstN(20);
```

For certain database operations that have no Java equivalent (such as SQL regular expressions or date arithmetic), utility methods are provided that support this functionality.

4.2 Translating JQS using JReq

For imperative JQS code to execute efficiently on a database, it must be translated into a declarative form that a database can optimize. This section explains this translation process using the query from Figure 4.1 as an example.

Since JQS queries are written using actual Java code, the JReq system cannot be implemented as a simple Java library. JReq must be able to inspect and modify Java code in order to identify queries and translate them to SQL. A simple Java library cannot do that. One of the goals of JReq, though, is for it to be non-intrusive and for it to be easily adopted or removed from a development process like a normal library. To do this, the JReq system is implemented as a bytecode rewriter that is able to take a compiled program outputted by the Java compiler and then transform the bytecode to use SQL. It can be added to the toolchain as an independent module, with no changes needed to existing IDEs, compilers, virtual machines, or other such tools (Figure 4.2). Although the current implementation has JReq acting as an independent code transformation tool, JReq can also be implemented as a postprocessing stage of a compiler, as a classloader that modifies code at runtime, or as part of a virtual machine.

The translation algorithm in JReq is divided into a number of stages. It first preprocesses the bytecode to make the bytecode easier to manipulate. The code is then broken up into loops, and each loop is transformed using symbolic execution into a new representation that preserves the semantics of the original code but removes many secondary features of the code, such as variations in instruction ordering, convoluted interactions between different instructions, or unusual control flow, thereby making it easier to identify

```
                $accounts = $db.allAccounts()
                $iter = $accounts.iterator()
                goto loopCondition
loopBody:       $next = $iter.next()
                $a = (Account) $next
                $country = $a.getCountry()
                $cmp0 = $country.equals("UK")
                if $cmp0==0 goto loopCondition
loopAdd:        $name = a$.getName()
                $results.add($name)
loopCondition:  $cmp1 = $iter.hasNext()
                if $cmp1!=0 goto loopBody
exit:
```

Figure 4.3: Jimple code of a query

queries in the code. This final representation is tree-structured, so bottom-up parsing is used to match the code with general query structures, from which the final SQL queries can then be generated.

4.2.1 Preprocessing

Although JReq inputs and outputs Java bytecode, its internal processing is not based on bytecode. Java bytecode is difficult to process because of its large instruction set and the need to keep track of the state of the operand stack. To avoid this problem, JReq uses the SOOT framework [VRCG^{+}99] from Sable to convert Java bytecode into a representation known as Jimple, a three-address code version of Java bytecode. In Jimple, there is no operand stack, only local variables, meaning that JReq can use one consistent abstraction for working with values and that JReq can rearrange instruction sequences without having to worry about stack consistency. Figure 4.3 shows the code of the query from Figure 4.1 after conversion to Jimple form.

4.2.2 Transformation of Loops

Since all JQS queries are expressed as loops iterating over collections, JReq needs to add some structure to the control-flow graph of the code. It breaks down the control flow graph into nested strongly-connected components (i.e. loops), and from there, it transforms and analyzes each component in turn. Since there is no useful mapping from individual instructions to SQL queries, the analysis operates on entire loops. Conceptually, JReq calculates the postconditions of executing all of the instructions of the loop and then tries to find SQL queries that, when executed, produce the same set of postconditions. If it can find such a match, JReq can replace the original code with the SQL query. Since the result of executing the original series of instructions from the original code gives the

Type	**Path**
Exiting	loopCondition $\rightarrow$ exit
Looping	loopCondition $\rightarrow$ loopBody $\rightarrow \circlearrowleft$
Looping	loopCondition $\rightarrow$ loopBody $\rightarrow$ loopAdd $\rightarrow \circlearrowleft$

Figure 4.4: Paths through the loop

same result as executing the query, the translation is safe. Unfortunately, because of the difficulty of generating useful loop invariants for loops [BM07], JReq is not able to calculate postconditions for a loop directly.

Loop Paths

To understand the behavior of loops, JReq will examine all the different execution paths through the loop. It can then combine the behaviors of these different paths to determine the behavior of an arbitrary iteration of a loop. To find these paths, JReq starts at the entry point to the loop and walks the control flow graph of the loop until it arrives back at the loop entry point or exits the loop. As it walks through the control flow graph, JReq enumerates all possible paths through the loop. The possible paths through the query code from Figure 4.3 are listed in Figure 4.4. Theoretically, there can be an exponential number of different paths through a loop since each `if` statement can result in a new path. In practice, such an exponential explosion in paths is rare. JReq's Java query syntax has an interesting property where when an `if` statement appears in the code, one of the branches of the statement usually ends that iteration of the loop, meaning that the number of paths generally grows linearly. The only types of queries that seem to lead to an exponential number of paths are ones that try to generate "CASE WHEN...THEN" SQL code, and these types of queries are rarely used. Although exponential path explosion is not thought to be a problem for JReq, such a situation can be avoided by using techniques developed by the verification community for dealing with similar problems [FS01].

For each path, JReq generates a Hoare triple. A Hoare triple describes the effect of executing a path in terms of the preconditions, code, and postconditions of the path. JReq knows what branches need to be taken for each path to be traversed, and the conditions on these branches form the preconditions for the paths. Method calls and modifications of variables become the postconditions of the paths.

Symbolic Execution

Symbolic execution is used when calculating these preconditions and postconditions. The use of symbolic execution means that all preconditions and postconditions are expressed in terms of the values of variables from the start of the loop iteration and that minor changes to the code like simple instruction reordering will not affect the derived postconditions. There are many different styles of symbolic execution, and JReq's use of symbolic execution to calculate Hoare triples is analogous to techniques used in the software verification

```
1: $cmp1 = $iter.hasNext()
2: if $cmp1 != 0 goto loopBody (branch taken)
3: $next = $iter.next()
4: $a = (Account) $next
5: $country = $a.getCountry()
6: $cmp0 = $country.equals("UK")
7: if $cmp0 == 0 goto loopCondition (branch skipped)
8: $name = a$.getName()
9: $results.add($name)
```

Figure 4.5: Instructions of the last path from Figure 4.4

community, particularly work on translation validation and credible compilation [Rin99, Nec00].

JReq's symbolic execution begins at the first instruction of a path and then traces through the execution of each instruction along the path. Instead of working with real concrete values for variables, which may differ each time a path is executed, JReq uses symbolic values for variables when executing the instructions. As it symbolically executes each instruction, JReq will gather preconditions and postconditions. On reaching the last instruction of the path, it will have computed the preconditions and postconditions for executing the entire path.

For each instruction, JReq essentially performs three steps:

- It will propagate any preconditions and postconditions from the previous instruction to the current instruction since any changes in the state of the program made by the previous instruction will continue to hold in the following instruction
- Any changes in state such as method calls or assignments to variables will be recorded as postconditions while any conditional branches will be noted as preconditions
- The new preconditions and postconditions may make use of variables that are known to contain other values, so those values are substituted in for those variables

As an example, consider the instructions (Figure 4.5) from the last path from Figure 4.4. If JReq applies the three steps of its symbolic execution algorithm to the first instruction, it generates the results shown in Figure 4.6. Because there are no previous instructions, there are no preconditions or postconditions to propagate during the first step. In the second step, the call to the `hasNext()` method and the assignment of the result to the variable `$cmp1` are both added to the list of postconditions. In the final step, there are no variables that need to be substituted, so the list of postconditions remains the same.

When symbolic execution to the second instruction (Figure 4.7), the postconditions calculated after the first instruction are propagated first. The second instruction is a conditional branch, so the condition becomes a precondition. Finally, this new precondition

Instruction

```
    1:  $cmp1 = $iter.hasNext()
```

After Propagation

None

After Gathering

Preconditions	
Postconditions	**$iter.hasNext()**
	$cmp1 = $iter.hasNext()

After Substitution

Preconditions	
Postconditions	$iter.hasNext()
	$cmp1 = $iter.hasNext()

Figure 4.6: The effect of applying the three steps of JReq's symbolic execution to the first instruction in Figure 4.5

references the `$cmp1` variable, and the list of postconditions shows that `$cmp1` has been assigned a certain value, so this value can be substituted into the precondition expression.

If symbolic execution is applied to all the instructions in the path, JReq will calculate the final preconditions and postconditions for the path.

Simplification

Figure 4.4 shows the final preconditions and postconditions for the path. Not all of the postconditions gathered are significant though, so JReq uses variable liveness information to prune assignments that are not used outside of a loop iteration and uses a list of methods known not to have side-effects to prune safe method calls. Figure 4.9 shows the final Hoare triples of all paths after pruning.

Basically, JReq has transformed the loop instructions into a new tree representation where the loop is expressed in terms of paths and various precondition and postcondition expressions. The semantics of the original code are preserved in that all the effects of running the original code are encoded as postconditions in the representation, but problems with instruction ordering or tracking instruction side-effects, etc. have been filtered out.

In general, JReq can perform this transformation of loops into a tree representation in a mechanical fashion, but JReq does make some small optimizations to simplify processing in later stages. For example, constructors in Java are methods with no return type. In JReq, constructors are represented as returning the object itself, and JReq reassigns the result of the constructor to the variable on which the constructor was invoked. This change means that JReq does not have to keep track of a separate method invocation postcondition for each constructor used in a loop.

Instruction

```
    2:  if $cmp1 != 0 goto loopBody (branch taken)
```

After Propagation

Postconditions	\$iter.hasNext()
	\$cmp1 = \$iter.hasNext()

After Gathering

Preconditions	**\$cmp1 != 0**
Postconditions	\$iter.hasNext()
	\$cmp1 = \$iter.hasNext()

After Substitution

Preconditions	**\$iter.hasNext()** != 0
Postconditions	\$iter.hasNext()
	\$cmp1 = \$iter.hasNext()

Figure 4.7: The effect of applying the three steps of JReq's symbolic execution to the second instruction in Figure 4.5

Path: loopCondition $\rightarrow$ loopBody $\rightarrow$ loopAdd $\rightarrow\circlearrowleft$

Preconditions	\$iter.hasNext() != 0
	((Account)\$iter.next()).getCountry().equals(”UK”) != 0
Postconditions	*\$iter.hasNext()*
	\$cmp1 = \$iter.hasNext()
	\$iter.next()
	\$next = \$iter.next()
	\$a = (Account) \$iter.next()
	((Account)\$iter.next()).getCountry()
	\$country = ((Account)\$iter.next()).getCountry()
	((Account)\$iter.next()).getCountry().equals(”UK”)
	\$cmp0 = ((Account)\$iter.next()).getCountry().equals(”UK”)
	((Account)\$iter.next()).getName()
	\$name = ((Account)\$iter.next()).getName()
	\$results.add(((Account)\$iter.next()).getName())

Figure 4.8: Hoare triple expressing the result of a path (expressions that will be pruned by liveness analysis are indented)

Exiting Path	
Preconditions	$iter.hasNext() == 0
Postconditions	
Looping Path	
Preconditions	$iter.hasNext() != 0
	((Account)$iter.next()).getCountry().equals(”UK”) == 0
Postconditions	$iter.next()
Looping Path	
Preconditions	$iter.hasNext() != 0
	((Account)$iter.next()).getCountry().equals(”UK”) != 0
Postconditions	$iter.next()
	$results.add(((Account)$iter.next()).getName())

Figure 4.9: Final Hoare triples generated from Figure 4.3 after pruning

4.2.3 Query Identification and Generation

Once the code has been transformed into Hoare triple form, traditional translation techniques can be used to identify and generate SQL queries. For example, Figure 4.10 shows how one general Hoare triple representation can be translated into a corresponding SQL form. That particular Hoare triple template is sufficient to match all non-nested SELECT...FROM...WHERE queries without aggregation functions. In fact, because the transformation of Java code into Hoare triple form removes much of the syntactic variation between code fragments with identical semantics, a small number of templates is sufficient to handle most queries.

Since the Hoare triple representation is in a nice tree form, bottom-up parsing can be used to classify and translate the tree into SQL. When using bottom-up parsing to match path Hoare triples to a template, one does have to be careful that each path add the same number and same types of data to the result collection (e.g. in Figure 4.10, one needs to check that the types of the various $valA_n$ being added to $\$results$ is consistent across the looping paths). One can use a unification algorithm across the different paths of the loop to ensure that these consistency constraints hold.

One further issue complicating query identification and generation is the fact that a full JQS query is actually composed of both a loop portion and some code before and after the loop. For example, the creation of the object holding the result set occurs before the loop, and when a loop uses an iterator object to iterate over a collection, the definition of the collection being iterated over can only be found outside of the loop. To find these non-loop portions of the query, the JReq transformation is recursively applied to the code outside of the loop at a higher level of nesting. Since the JReq transformation breaks down a segment of code into a finite number of paths to which symbolic execution is applied, the loop needs to be treated as a single indivisible “instruction” whose postconditions are the same as the loop’s postconditions during this recursion. This recursive application of the JReq transformation is also used for converting nested loops into nested SQL queries.

Exiting Path	
Preconditions	$iter.hasNext() == 0
Postconditions	*exit loop*
Looping Path$_i$	
Preconditions	$iter.hasNext() != 0
	...
Postconditions	$iter.next()
...etc.	
Looping Path$_n$	
Preconditions	$iter.hasNext() != 0
	pred$_n$
Postconditions	$iter.next()
	$results.add(***valA***$_n$, ***valB***$_n$, ...)
...etc.	

```
SELECT
   CASE WHEN pred₁ THEN valA₁
        WHEN pred₂ THEN valA₂
        ...
   END,
   CASE WHEN pred₁ THEN valB₁
        WHEN pred₂ THEN valB₂
        ...
   END,
   ...
FROM ?
WHERE pred₁ OR pred₂ OR ...
```

Figure 4.10: Code with a Hoare triple representation matching this template can be translated into a SQL query in a straight-forward way

Figure 4.11 shows the Hoare triples of the loop and non-loop portions of the query from Figure 4.1.

Figure 4.12 shows some sample operational semantics that illustrate how the example query could be translated to SQL. In the interest of space, these operational semantics do not contain any error-checking and show only how to match the specific query from Figure 4.1 (as opposed to the general queries supported by JReq). The query needs to be processed three times using mappings S, F, and W to generate SQL select, from, and where expressions respectively. σ holds information about variables defined outside of a loop. In this example, σ describes the table being iterated over, and Σ describes how to look up fields of this table.

JReq currently generates SQL queries statically by replacing the bytecode for the JQS query with bytecode that uses SQL instead. Static query generation allows JReq to apply more optimizations to its generated SQL output and makes debugging easier because one can examine generated queries without running the program. During this stage, JReq can also optimize the generated SQL queries for specific databases though the prototype currently does not contain such an optimizer. In a previous version of JReq, SQL queries were constructed at runtime and evaluated lazily. Although this results in slower queries, it allows the system to support a limited form of inter-procedural query generation. A query can be created in one method, and the query result can later be refined in another method.

During query generation, JReq uses line number debug information from the bytecode to show which lines of the original source files were translated into SQL queries and what they were translated into. IDEs can potentially use this information to highlight which lines of code can be translated by JReq as a programmer types them. Combined with the type error and syntax error feedback given by the Java compiler at compile-time, this

```
Hoaretriples(
  Exit(
    Pre($iter.hasNext() == 0),
    Post()
  ),
  Looping(
    Pre($iter.hasNext() != 0,
        ((Account)$iter.next()).getCountry().equals("UK") == 0),
    Post(Method($iter.next()))
  ),

  Looping(
    Pre($iter.hasNext() != 0,
        ((Account)$iter.next()).getCountry().equals("UK") != 0),
    Post(Method($iter.next()),
         Method($uk.add(((Account)$iter.next()).getName())))))
```

```
PathHoareTriple(
   Pre(),
   Post($results = (new QueryList()).addAll(
            $db.allAccounts().iterator().AddQuery())))
```

Figure 4.11: The Hoare triples of the loop and non-loop portion of the query from Figure 4.1. The loop Hoare triples are identical to those from Figure 4.9, except they have been rewritten so as to emphasize the parsability and tree-like structure of the Hoare triple form

$$\frac{\begin{array}{l} a = \texttt{Exit(Pre(\$iter.hasNext()==0), Post())} \\ b = \texttt{Looping(Pre(\$iter.hasNext()!=0, ...),} \\ \qquad\qquad\quad \texttt{Post(Method(\$iter.next())))} \\ c = \texttt{Looping(Pre(\$iter.hasNext()!=0, } d\texttt{),} \\ \qquad\qquad\quad \texttt{Post(Method(\$iter.next()), } e\texttt{))} \\ e = \texttt{Method(}resultset\texttt{.add(}child\texttt{))} \\ S \vdash \langle child, \sigma \rangle \Downarrow select \\ W \vdash \langle d, \sigma \rangle \Downarrow where \end{array}}{\begin{array}{l} S \vdash \langle \texttt{Hoaretriples}(a, b, c), \sigma \rangle \Downarrow select \\ W \vdash \langle \texttt{Hoaretriples}(a, b, c), \sigma \rangle \Downarrow where \end{array}}$$

$$\frac{\begin{array}{l} W \vdash \langle left, \sigma \rangle \Downarrow where_l \\ W \vdash \langle right, \sigma \rangle \Downarrow where_r \end{array}}{W \vdash \langle left\texttt{.equals(}right\texttt{)==0}, \sigma \rangle \Downarrow where_l \texttt{<>} where_r}$$

$$\frac{\begin{array}{l} W \vdash \langle left, \sigma \rangle \Downarrow where_l \\ W \vdash \langle right, \sigma \rangle \Downarrow where_r \end{array}}{W \vdash \langle left\texttt{.equals(}right\texttt{)!=0}, \sigma \rangle \Downarrow where_l \texttt{=} where_r} \qquad \frac{}{\begin{array}{l} S \vdash \langle \texttt{"UK"}, \sigma \rangle \Downarrow \textbf{“UK”} \\ W \vdash \langle \texttt{"UK"}, \sigma \rangle \Downarrow \textbf{“UK”} \end{array}}$$

$$\frac{\Sigma \vdash \langle child, \sigma, \textsc{Name} \rangle \Downarrow val}{\begin{array}{l} S \vdash \langle child\texttt{.getName()}, \sigma \rangle \Downarrow val \\ W \vdash \langle child\texttt{.getName()}, \sigma \rangle \Downarrow val \end{array}} \qquad \frac{\Sigma \vdash \langle child, \sigma, \textsc{Country} \rangle \Downarrow val}{\begin{array}{l} S \vdash \langle child\texttt{.getCountry()}, \sigma \rangle \Downarrow val \\ W \vdash \langle child\texttt{.getCountry()}, \sigma \rangle \Downarrow val \end{array}}$$

$$\frac{}{\begin{array}{l} \Sigma \vdash \langle \texttt{(Account)\$iter.next()}, \sigma, \textsc{Country} \rangle \Downarrow \sigma(\textsc{next})\textbf{.Country} \\ \Sigma \vdash \langle \texttt{(Account)\$iter.next()}, \sigma, \textsc{Name} \rangle \Downarrow \sigma(\textsc{next})\textbf{.Name} \end{array}}$$

$$\frac{}{F \vdash \langle \texttt{\$db.allAccounts().iterator()}, \sigma \rangle \Downarrow \textbf{Account}}$$

$$\frac{\begin{array}{l} S \vdash \langle \texttt{HoareTriples(...)}, \sigma[\textsc{next} := \mathbf{A}] \rangle \Downarrow select \\ W \vdash \langle \texttt{HoareTriples(...)}, \sigma[\textsc{next} := \mathbf{A}] \rangle \Downarrow where \\ F \vdash \langle iterator, \sigma \rangle \Downarrow from \end{array}}{\begin{array}{l} \langle resultset\texttt{.addAll(}iterator\texttt{.AddQuery())}, \sigma \rangle \Downarrow \\ \qquad \textbf{SELECT } select \textbf{ FROM } from \textbf{ AS A WHERE } where \end{array}}$$

Figure 4.12: Sample operational semantics for translating Figure 4.11 to SQL

feedback helps programmers write correct queries and optimize query performance.

4.2.4 Implementation Expressiveness and Limitations

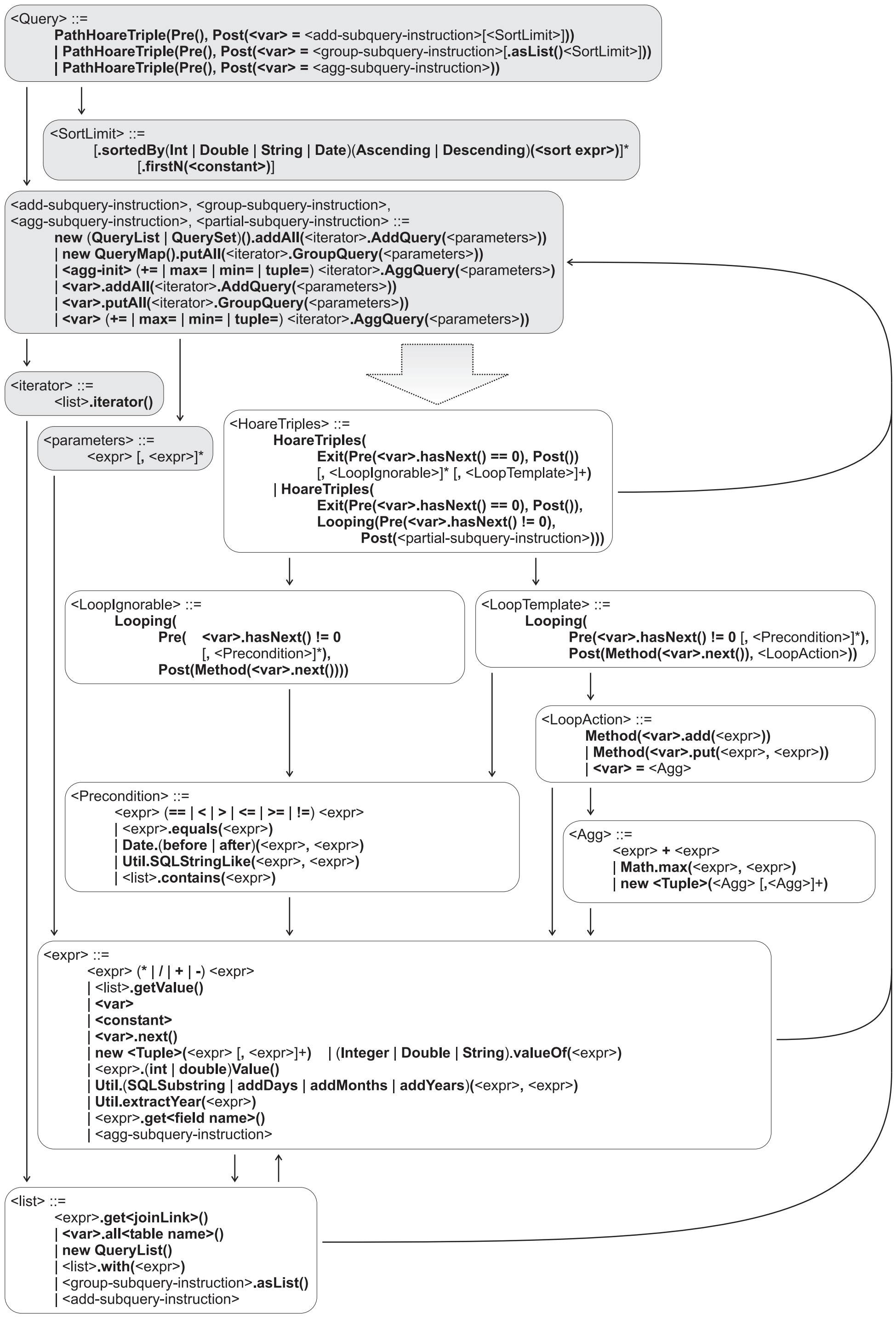
The translation algorithm behind JReq is designed to be able to recognize queries with the complexity of SQL92 [Ame92]. This implementation, though, focuses on the subset of operations used in typical SQL database queries. Figure 4.13 shows a grammar of JQS, the Java code that JReq can translate into SQL. JQS is specified using the grammar of Hoare triples from after the symbolic execution stage of JReq. This approach is used because it is concise and closely describes what queries will be accepted. Specifying JQS using a traditional grammar directly describing a Java subset was found to be too imprecise or too narrow to be useful. Because JReq uses symbolic execution, for each query, any Java code variant with the same semantic meaning will be recognized by JReq as being the same query. This large number of variants cannot be captured using a direct specification of a Java grammar subset.

In the figure, the white boxes refer to grammar rules used for classifying loops. The gray boxes are used for combining loops with context from outside of the loop. There are four primary templates for classifying a loop: one for adding elements to a collection, one for adding elements to a map, one for aggregating values, and another for nested loops resulting in a join. Most SQL operations can be expressed using the functionality described by this grammar.

Some SQL functionality that is not currently supported by JQS include set operations, intervals, and internationalization because the queries used in this thesis did not require this functionality. Support for NULL and related operators was also left out of this iteration of JQS. Because Java does not support three-value logic or operator overloading, special objects and methods to emulate the behavior of NULL would have been necessary, resulting in a verbose and complicated design. Operations related to NULL values such as OUTER JOINs are not supported as well.

JQS also currently offers only basic support for update operations since it focuses only on the query aspects of SQL. SQL's more advanced data manipulation operations are rarely used and not too powerful, so it would be fairly straight-forward to extend JQS to support these operations. Most of these operations are simply composed of a normal query followed by some sort of INSERT, DELETE, or UPDATE involving the result set of the query.

In the end, the JReq system comprises approximately 20 thousand lines of Java and XSLT code. Although JReq translations can be applied to an entire codebase, annotations are used to direct JReq into applying its transformations only to specific methods known to contain queries. Additionally, some planned features were never implemented because the experiments did not require them: the handling of non-local variables, type-checking or unification to check for errors in queries, and pointer aliasing support.



```
<Query> ::=
        PathHoareTriple(Pre(), Post(<var> = <add-subquery-instruction>[<SortLimit>]))
        | PathHoareTriple(Pre(), Post(<var> = <group-subquery-instruction>[.asList()<SortLimit>]))
        | PathHoareTriple(Pre(), Post(<var> = <agg-subquery-instruction>))

<SortLimit> ::=
        [.sortedBy(Int | Double | String | Date)(Ascending | Descending)(<sort expr>)]*
                [.firstN(<constant>)]

<add-subquery-instruction>, <group-subquery-instruction>,
<agg-subquery-instruction>, <partial-subquery-instruction> ::=
        new (QueryList | QuerySet)().addAll(<iterator>.AddQuery(<parameters>))
        | new QueryMap().putAll(<iterator>.GroupQuery(<parameters>))
        | <agg-init> (+= | max= | min= | tuple=) <iterator>.AggQuery(<parameters>)
        | <var>.addAll(<iterator>.AddQuery(<parameters>))
        | <var>.putAll(<iterator>.GroupQuery(<parameters>))
        | <var> (+= | max= | min= | tuple=) <iterator>.AggQuery(<parameters>))

<iterator> ::=
        <list>.iterator()

<parameters> ::=
        <expr> [, <expr>]*

<HoareTriples> ::=
        HoareTriples(
                Exit(Pre(<var>.hasNext() == 0), Post())
                [, <LoopIgnorable>]* [, <LoopTemplate>]+)
        | HoareTriples(
                Exit(Pre(<var>.hasNext() == 0), Post()),
                Looping(Pre(<var>.hasNext() != 0),
                        Post(<partial-subquery-instruction>)))

<LoopIgnorable> ::=
        Looping(
                Pre(    <var>.hasNext() != 0
                        [, <Precondition>]*),
                Post(Method(<var>.next())))

<LoopTemplate> ::=
        Looping(
                Pre(<var>.hasNext() != 0 [, <Precondition>]*),
                Post(Method(<var>.next()), <LoopAction>))

<LoopAction> ::=
        Method(<var>.add(<expr>))
        | Method(<var>.put(<expr>, <expr>))
        | <var> = <Agg>

<Precondition> ::=
        <expr> (== | < | > | <= | >= | !=) <expr>
        | <expr>.equals(<expr>)
        | Date.(before | after)(<expr>, <expr>)
        | Util.SQLStringLike(<expr>, <expr>)
        | <list>.contains(<expr>)

<Agg> ::=
        <expr> + <expr>
        | Math.max(<expr>, <expr>)
        | new <Tuple>(<Agg> [,<Agg>]+)

<expr> ::=
        <expr> (* | / | + | -) <expr>
        | <list>.getValue()
        | <var>
        | <constant>
        | <var>.next()
        | new <Tuple>(<expr> [, <expr>]+)    | (Integer | Double | String).valueOf(<expr>)
        | <expr>.(int | double)Value()
        | Util.(SQLSubstring | addDays | addMonths | addYears)(<expr>, <expr>)
        | Util.extractYear(<expr>)
        | <expr>.get<field name>()
        | <agg-subquery-instruction>

<list> ::=
        <expr>.get<joinLink>()
        | <var>.all<table name>()
        | new QueryList()
        | <list>.with(<expr>)
        | <group-subquery-instruction>.asList()
        | <add-subquery-instruction>
```

Figure 4.13: JQS grammar

4.3 Evaluation

4.4 Syntax Usability

Although JQS provides a syntax for database queries that is consistent with existing Java conventions for processing collections, it is unclear whether programmers would prefer this syntax. Intuitively, having a single common syntax for both general purpose computation and for database queries should benefit programmers by eliminating the "semantic gap." Modern object-oriented languages are written in an imperative style and use an object abstraction to model data. Database query languages like SQL are written in a declarative style and use a relational abstraction to model data. Programmers supposedly require extra training and expertise to handle this difference in the semantics of these languages. Even then, their productivity may be reduced by having to mentally use both models simultaneously when programming a database.

Alternatively, one could believe that the effect of this semantic gap is small and that there are more important factors that should be considered in designing a query language. Because query languages like SQL are designed specifically for accessing databases, it is possible that their syntax is more intuitive than what can be achieved using a more general-purpose syntax. Although programmers may be more familiar with the object-oriented imperative syntax of languages like Java, the declarative nature of SQL queries might be inherently better-suited to the database domain.

To gain some insight into whether JQS provides a reasonable syntax for describing database queries, a small user study has been conducted into how people understand database queries written in either JQS or in SQL using JDBC. The user study involves observing users as they interacted with database queries in order to see how users approached the queries and to see what difficulties they encountered.

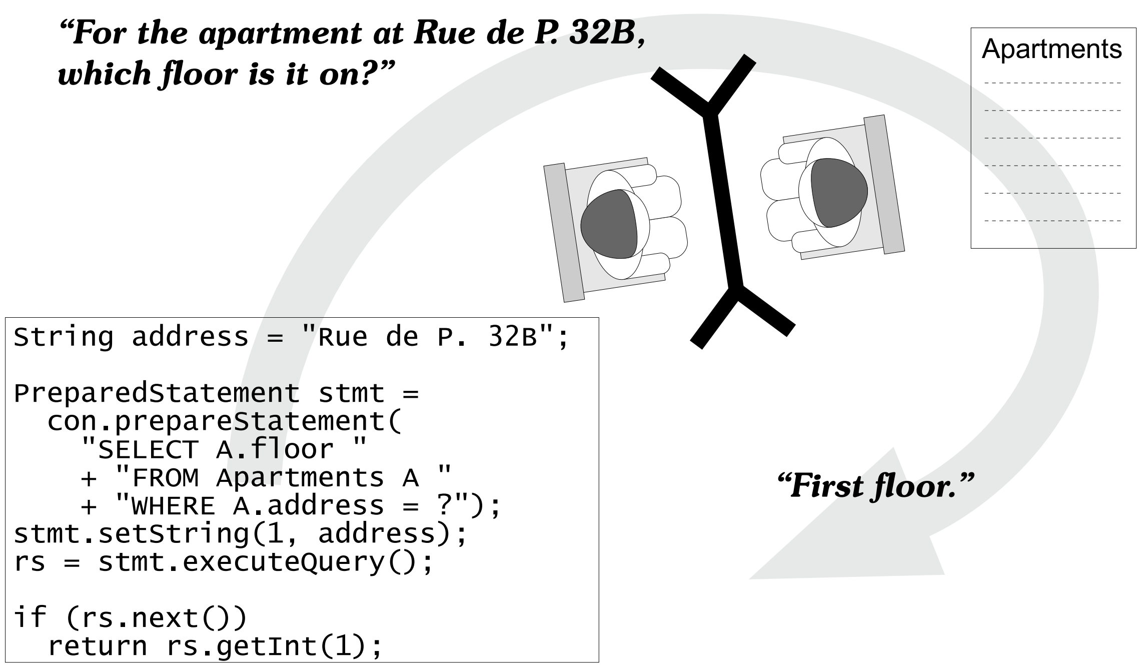
4.4.1 Question and Experiment

The user study was designed to focus on the task of understanding database queries rather than the task of writing database queries. This was done because

- Studies that involve the writing of program code are time-consuming, requiring a larger time commitment from study participants and resulting in less data to be analyzed

- Before study participants can write database queries, they must be given instruction in the corresponding query languages, and this can introduce a potential source of bias in the experiments

The user study experiment was performed in groups of two people. The participants were told to imagine a scenario where they have a computer program that queries a database of apartment listings, but that the database is currently unavailable. Instead, they have to phone up someone with printouts of all the apartment listings and ask them to look up the data instead. Unlike a study design where participants can simply try to



```
String address = "Rue de P. 32B";

PreparedStatement stmt =
  con.prepareStatement(
    "SELECT A.floor "
    + "FROM Apartments A "
    + "WHERE A.address = ?");
stmt.setString(1, address);
rs = stmt.executeQuery();

if (rs.next())
  return rs.getInt(1);
```

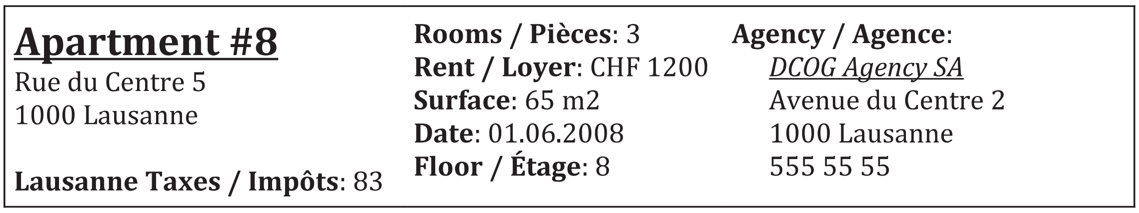
Figure 4.14: In the user study, one study participant must interpret a database query and ask the other study participant to look up the answer to the query from a printout of the database contents

interpret database queries on their own, this study design provides a much richer set of data because the interaction between the two subjects can provide some insight into their thought processes. By forcing subjects to actually describe queries orally, one can see how they converge to their chosen answers and what sort of difficulties they encounter along the way.

During the experiment, one person sits in front of a computer, while the other sits with a two page printout of apartment listings (Figure 4.14). A short Styrofoam wall separates the two persons. The person in front of the computer sees the program code of a computer program that queries a database of apartment listings. This person then needs to ask the other person to lookup the answer to the query in the printouts. The participants are told to complete the task correctly but also as quickly as possible.

The person in front of the computer has to find the answers to six queries, which will all be written either in SQL or JQS. They have a five minute time-limit to find the answer to each query. After answering the six queries, the experiment participants switch roles. If the first set of programs were written using SQL, then the next set of programs will be written in JQS and vice versa. When the experiment is completed, both participants fill in a short questionnaire about their experiences.

The experiment is designed to assess the difficulty that people have in understanding the queries written in SQL and JQS based on whether they correctly interpret the meanings of each query. It also provides some insight into the mental thought processes that people



Apartment #8
Rue du Centre 5
1000 Lausanne

Lausanne Taxes / Impôts: 83

Rooms / Pièces: 3
Rent / Loyer: CHF 1200
Surface: 65 m2
Date: 01.06.2008
Floor / Étage: 8

Agency / Agence:
DCOG Agency SA
Avenue du Centre 2
1000 Lausanne
555 55 55

Figure 4.15: A sample apartment listing

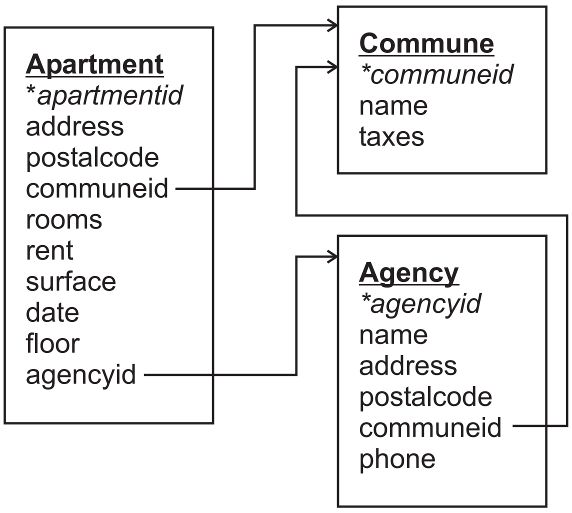


Figure 4.16: Relational schema envisioned for the apartment listings

may use in understanding the queries since they must orally describe the queries to others during the course of the experiment.

The Database

The queries used in the experiment are intended to act on a database of apartment listings. There are twelve apartments in the listings, and the data for each apartment is printed in the format shown in Figure 4.15. When participants are given instructions about the experiment, they are also given a sample listing of five apartments that they can study so that they can familiarize themselves with the data available. This was intended to reduce the variability in results between the first half of the experiment and the second half of the experiment where the roles of the study participants are reversed and the study participants have more experience with the queries and the data schema.

The queries in the experiment assume that the apartment data is represented using the relational schema shown in Figure 4.16 (or in a corresponding object-oriented schema in the case of JQS). Fields marked with an asterisk are primary keys for the relation.

	Query Description	Variation A	Variation B
1	Single table query	The rents of apartments with 1.5 rooms or less	The rents of apartments with a surface of 30 square meters or less
2	Single table query with aggregation and two conditions	The number of apartments available before 15.8.2008 and with a surface greater than 100 square meters	The number of apartments with a rent of less than 1500 and with more than 2 rooms
3	Single table query	The floor of the apartment at the address Avenue de A.	The address of the apartment with postal code 1025
4	Natural join between two tables	The taxes of the commune for the apartment at the address Rue de A.	The name of the agency responsible for the apartment at the address Rue de A.
5	Single table query	The addresses of apartments with a rent of below 900	The addresses of apartments available before 15.5.2008
6	Join of a table with itself	The name of communes with taxes higher than the Renens commune	The name of the agencies with the same phone number as the agency named A.

Table 4.1: Descriptions of queries from the user study

The Queries

In total there are twelve queries in the experiment, divided into two groups of six queries. Queries from between the two groups are designed to have comparable difficulty and structure. Table 4.1 gives descriptions of all the queries used in the experiment. Within each group of six queries, the queries alternate between fairly simple queries and queries of moderate difficulty. Overall, the task of understanding these queries is supposed to be comparable to what a programmer might face if they had to learn the codebase of a new web application that uses a database.

Each pair of participants in the study alternated between starting with the first six queries being written in SQL or with the first six queries being written in JQS. Participants were allowed to choose amongst themselves who would start with the queries and who would start with the printouts of apartments.

The Questionnaire

The questionnaire asked participants to rank their knowledge of SQL into one of four levels of experience—none, beginner, intermediate, and expert—based on whether they have no experience with SQL, have taken a course or read a book on SQL, taken multiple courses on SQL or have worked on a SQL databases project, or have used SQL extensively in multiple databases projects. Participants' programming experience was also classified into one of four categories—none, beginner, intermediate, and expert—based on whether they do not know how to program, have 4 years or less of programming experience, over 4 years of programming experience, or over 4 years of experience and experience with programming projects not related to courses. Participants were also asked to rank their understanding of queries on a scale of one to five, to describe what they found difficult in understanding the queries and to describe which queries were difficult to understand and why.

4.4.2 Results

After a small pilot test involving two participants to find potential problems with the study design, the user study was run with twelve participants drawn from various graduate students and interns doing systems research at the computer science department of EPFL.

The audio of the conversations between the two participants was recorded and transcribed. Two timing measurements were gathered for each query. The first measurement is of the time between when a query is first shown and when the participant trying to understand the query gives their first instruction to the participant with the printouts. This is supposed to capture the time needed for someone to understand some part of a query. The data is potentially noisy though because the timing data includes the time that participants sometimes spent discussing the previous query and because some participants start giving instructions as soon as they understand even only a small part of the query while others wait until they fully understand the query before speaking. The second measurement is of the time between when a query is first shown and when the participant enters their answer to the query and clicks on a button to move on to the next query. This is intended to capture the total time needed to understand a query, but, again, the data is noisy for the same reasons listed before and also due to the variability in the amount of time needed for the other study participant to look up data and the variability in the amount of time needed to type in answers at the computer.

The audio transcripts were also analyzed to judge the correctness of the participants' interpretations of the queries. The user study's correctness criteria was that participants needed to correctly identify the fields returned by the query, the subset of data selected by the query, and the relationship between the different entities used in the query. Participants' final formulation of the query could not refer to unnecessary elements (so participants could not simply read the query verbatim, for example) though participants could give instructions asking for more data than is strictly necessary to answer the query if they later filter this data themselves when entering the answer to the query.

Overall, the experiment unfolded without incident, though some study participants still expressed some confusion about the data schema during the experiment despite being able to study it before the experiment commenced. Also, participants who had to understand queries written in JQS tended to have more programming experience and slightly more SQL experience than the participants who had to understand queries written in SQL.

	JQS		SQL		
	Mean	S.D.	Mean	S.D.	F
Overall	22s	17s	28s	31s	F(1, 60) = 1.16, p<0.28
Simple Queries (Q1, Q3, Q5)	22s	11s	29s	38s	F(1, 30) = 0.72, p<0.40
Moderate Queries (Q2, Q4, Q6)	22s	22s	27s	23s	F(1, 30) = 0.45, p<0.51
Q1	31s	10s	60s	53s	F(1, 10) = 1.79, p<0.21
Q2	13s	3s	30s	34s	F(1, 10) = 1.51, p<0.25
Q3	20s	9s	10s	7s	F(1, 10) = 5.64, p<0.04
Q4	11s	5s	29s	18s	F(1, 10) = 5.07, p<0.05
Q5	15s	7s	16s	19s	F(1, 10) = 0.03, p<0.86
Q6	43s	30s	22s	17s	F(1, 10) = 2.08, p<0.18

Table 4.2: ANOVA results of the time needed for a participant to give his or her first instruction after seeing a query. Interaction effects are excluded because they do not have a meaningful interpretation in this experiment

	JQS	SQL
Important details missed	2	3
Confusion about multiple entities	1	1
General misunderstanding		1
Problems with joins and entity keys		4

Table 4.3: A summary of the errors in query understanding that occurred during the user study

Analysis

For completeness, a table with the analysis of variance of the times needed for study participants to understand queries is included (Table 4.2). The user study was not designed to generate quantitative conclusions given its small size, and this fact is reflected in the results.

Table 4.3 shows the number of incorrectly interpreted queries from the user study and the reasons behind each error. Although the results seem to suggest that participants had more difficulty understanding SQL queries than JQS queries, this conclusion cannot be definitively drawn due to the small size of the study. Nonetheless, this error data provides some insights into query language features that can cause confusion among programmers.

In terms of errors caused by important details missed, it seems likely that the verbosity of a query language might obscure important information or its syntax may emphasize unimportant query features rather than important ones. For example, two of the JQS errors were caused by participants asking for the wrong field to be returned in the result. Figure 4.17 shows such an example. With SQL, there was only one such error. But JDBC's syntax for passing parameters to SQL did result in two errors. Participants mistook the parameter index as being the parameter itself. Figure 4.18 shows the query in question. The specific mistake made was that participants asked for apartments with less than a single room instead of less than 1.5 rooms. No such problems occurred with the equivalent

QUERY: OK, give me all the apartments that . . . ah, number of rooms is less or equal or 1.5
PRINTOUTS: Number of apartments?
QUERY: Number of rooms.
PRINTOUTS: No, you want to know number of apartments that has this?
QUERY: Uh, no, each of them.
. . .
PRINTOUTS: Apartment four, five, uh, that's it.

Figure 4.17: A participant fails to ask for the rent field of apartments

```
double numRooms = 1.5;

PreparedStatement stmt = con.prepareStatement(
    "SELECT A.rent "
  + "FROM Apartments A "
  + "WHERE A.rooms <= ?");
stmt.setDouble(1, numRooms);
rs = stmt.executeQuery();

Vector<Integer> toReturn = new Vector<Integer>();
while (rs.next())
  toReturn.add(rs.getInt(1));
return toReturn;
```

Figure 4.18: Potential SQL parameter confusion in a JDBC query

JQS query (Figure 4.19).

For SQL, problems understanding the use of keys in joins resulted in four errors. Figure 4.20 shows a typical example of the confusion that occurs. SQL uses keys to identify entities and to relate different entities with each other. These keys are usually not inherent to the entity but are artificial constructs needed to model entities in the database. In an object-oriented query language, this sort of confusion is rarer because entity keys are rarely exposed, and the relationship between entities are exposed as methods (Figure 4.21).

The query from the same SQL example in Figure 4.21 also caused some confusion about the multiple entities (apartments and agencies) involved. Participants thought solely in terms of apartment entities. One of the errors for JQS was caused by a similar misunderstanding (Figure 4.22).

Finally, one participant using JQS and one participant using SQL experienced general confusion in trying to understand the queries (Figure 4.23).

```
double numRooms = 1.5;

DBSet<Integer> toReturn = new DBSet$<$Integer$>$();
for (Apartment a: db.allApartments()) {
   if (a.rooms() <= numRooms)
      toReturn.add(a.rent());
}
return toReturn;
```

Figure 4.19: This JQS query is equivalent to the JDBC query in Figure 4.18, but did not lead to confusion about parameters

QUERY: Uh, you have to select, uh, you have to select the name of an apartment
PRINTOUTS: Mm.
QUERY: Whose ... agency id same as the id of Rue du [A].
PRINTOUTS: So the apartment, uh, where the agency is on this address?
QUERY: Uh, you first look at the agency id of, Rue du [A].
PRINTOUTS: There is no agency there
QUERY: What is this? Agency Id?...
...

Figure 4.20: A participant cannot understand how foreign and primary keys describe a relationship between two tables

```
SELECT B.name                          for (Apartment a: db.allApartments())
  FROM Apartments A, Agencies B          if (a.address().equals(address))
  WHERE A.address = ?                      toReturn.add(a.agency().name());
  AND A.agencyid = B.agencyid
```

Figure 4.21: These query excerpts demonstrate how in an object-oriented query language like JQS, the relationship between entities is explicit, unlike in SQL

QUERY: Agency [B] S dot A.
PRINTOUTS: Wait, wait, wait ... yeah?
QUERY: One second, I'll tell you ... OK. Find the apartments with that, ah, agency [B] S dot A
PRINTOUTS: Got it.
QUERY: Got it, nah? ... Now... and find all other apartments with the phone number, with the same phone number as this.
PRINTOUTS: With the same phone number?
QUERY: Yeah
...

Figure 4.22: Participants became confused between apartment and agency entities

QUERY: Yeah, um . . . Oh my god. Give me two different communes . . . that have the same name. Uh. Wait. Two different communes that have the same name but somehow the taxes are . . . in one of them is higher than in the other.
PRINTOUTS: Mmm.
QUERY: Uh, so . . . Hmm. Find two communes that have the same name, but, somehow, y'know, different taxes.
PRINTOUTS: There's none.
QUERY: Hmm? Nothing
PRINTOUTS: No.

Figure 4.23: General confusion over a join

4.4.3 Discussion

Overall, although a user study of this size does not allow one to make definitive statements about whether one language is superior to another, the results do suggest that JQS compares favorably with SQL using JDBC and that JQS avoids characteristics such as joins and entity keys that can cause problems in SQL queries.

Furthermore, the study provides some insight into ways in which query languages ease of use can be improved in general:

- Requiring programmers to manually marshal parameters into a query can cause confusion
- Explicitly encoding the relationship between entities and hiding the use of keys will result in more easily understood queries
- Programmers sometimes have trouble identifying the entities being examined by a query and the fields returned by a query, so a query language should try to make these elements of a query more clear

The concept of the "semantic gap" was evident in the study results in the form of users having difficulty understanding joins between relations whereas they had little difficulty understanding the explicit links between objects.

Interestingly, the user study did not find any indication that these query language characteristics caused any difficulties:

- Declarative-style vs. imperative-style queries
- Syntax differences between a query language and the object-oriented language it is embedded inside

This may be explained by the fact that the user study focused solely on understanding queries, and these language characteristics may primarily be useful for programmers trying to write new queries.

4.4.4 TPC-W

The behavior of JReq was evaluated by testing the ability for the JReq system to handle the database queries used in the TPC-W benchmark [Tra02]. TPC-W emulates the behavior of database-driven websites by recreating a website for an online bookstore.

The Rice implementation of TPC-W [ACC$^+$02], which uses JDBC to access its database, was used as a starting point. For each query in the TPC-W benchmark, an equivalent query using JQS was written. The SQL generated from JQS was manually verified to be semantically equivalent to the original SQL. The performance of each query when using the original JDBC and when using the JReq system could then be compared. The JReq prototype does not provide support for database updates, so queries involving updates were not tested. Since this experiment is intended to examine the queries generated by JReq as compared to hand-written SQL, some of the extra features of JReq such as transaction and persistence lifecycle management were also disabled.

A 600 MB database in PostgreSQL 8.3.0 [Pos] was created by populating the database with the number of items set to 10000. The complete TPC-W benchmark, which tests the complete system performance of web servers, application servers, and database servers, was not run. Instead, the experiment focused on measuring the performance of individual queries instead. Each query was first executed the query 200 times with random valid parameters to warm the database cache, then the time needed to execute the query 3000 times with random valid parameters was measured, and finally the system was garbage collected. Because of the poor performance of the getBestSellers query, it was only executed it for 50 times to warm the cache and the performance of executing the query only 250 times was measured. The experiment first took the JQS version of the queries, measured the performance of each query consecutively, and repeated the benchmark 50 times. The averages of only the last 10 runs are recorded to avoid the overhead of Java dynamic compilation. The experiment then repeated this experiment using the original JDBC implementation instead of JQS. The database and the query code were both run on the same machine, a 2.5 GHz Pentium IV Celeron Windows machine with 1 GB of RAM. The benchmark harness was run using Sun's 1.5.0 Update 12 JVM. JReq required approximately 7 seconds to translate the 12 JQS queries into SQL.

The performance of each of the queries is shown in Table 4.4. In all cases, JReq is faster than hand-written SQL. These results are a little curious because one usually expects hand-written code to be faster than machine-generated code. If one looks at the one query in Figure 4.24 that shows the code of the original hand-written JDBC code and compares it to the comparable JQS query and the JDBC generated from that query, one can see that the original JDBC code is essentially the same as the JDBC generated by JReq. In particular, the SQL queries are structurally the same though the JReq-generated version is more verbose. What makes the JReq version faster though is that JReq is able to take advantage of small runtime optimizations that are cumbersome to implement when writing JDBC by hand. For example, all JDBC drivers allow programmers to parse SQL queries into an intermediate form. Whenever the same SQL query is executed but with different parameters, programmers can supply the intermediate form of the query to the SQL driver instead of the original SQL query text, thereby allowing the SQL driver to

Table 4.4: The average execution time, standard deviation, and difference from hand-written JDBC/SQL (all in milliseconds) of the TPC-W benchmark are shown in this table with the column JReq NoOpt referring to JReq with runtime optimizations disabled. One can see that JReq offers better performance than the hand-written SQL queries

	JDBC		JReq NoOpt			JReq		
Query	Time	σ	Time	σ	Δ	Time	σ	Δ
getName	3592	112	3633	24	1%	2241	15	(38%)
getCustomer	8424	79	8944	57	6%	3939	24	(53%)
getMostRecentOrder	29108	731	88831	644	205%	8009	57	(72%)
getBook	6392	30	7347	55	15%	3491	27	(45%)
doAuthorSearch	10216	24	10414	559	2%	7306	46	(28%)
doSubjectSearch	16999	128	16898	86	(1%)	13667	120	(20%)
getIDandPassword	3706	33	3820	41	3%	2375	25	(36%)
getBestSellers	4472	50	4455	51	(0%)	3936	39	(12%)
doTitleSearch	27302	203	26979	418	(1%)	23985	61	(12%)
getNewProducts	23111	68	24447	128	6%	21086	70	(9%)
getRelated	6162	52	7731	92	25%	2690	34	(56%)
getUserName	3506	57	3569	13	2%	2214	11	(37%)

skip repeatedly reparsing and reanalyzing the same SQL query text. Taking advantage of this optimization in hand-written JDBC code is cumbersome because the program must be structured in a certain way and a certain amount of bookkeeping is involved, but this is all automated by JReq.

Table 4.4 also shows the performance of code generated by JReq if these runtime optimizations are disabled (denoted as JReq NoOpt). Of the 12 queries, the performance of JReq and hand-written JDBC is identical for six of them. The other six queries show slower performance in JReq than with hand-written JDBC for a variety of different reasons:

- Three queries (getBook, getCustomer, and getMostRecentOrder) are slower because they fetch too much data. The original queries fetched most of the fields of certain entities but not all of them, whereas the Queryll version of the query was written in such a way as to read in the whole entity with all of its fields.

- One query (getNewProducts) also fetched more data than the original query. This is caused by a limitation of the current Queryll syntax for sorting, which only allows results to be sorted based on data in the results. The original query sorted its results based on a field not in the final results. This field had to be fetched in the Queryll version to allow it to be sorted properly. A better syntax for sorting would resolve this issue.

- One of the queries (getRelated) was slower because the generated SQL was much longer than the original SQL. The query involves ORing together five expressions,

Original hand-written JDBC query

```
PreparedStatement getUserName = con.prepareStatement(
   "SELECT c_uname FROM customer WHERE c_id = ?");
getUserName.setInt(1, C_ID);
ResultSet rs=getUserName.executeQuery();
if (!rs.next()) throw new Exception();
u_name = rs.getString("c_uname");
rs.close(); stmt.close();
```

Comparable JQS query

```
EntityManager em = db.begin();
DBSet<String> matches = new QueryList<String>();
for (DBCustomer c: em.allDBCustomer())
   if (c.getCustomerId()==C_ID) matches.add(c.getUserName());
u_name = matches.get();
db.end(em, true);
```

JDBC generated by JReq

```
PreparedStatement stmt = null; ResultSet rs = null;
try { stmt = stmtCache.poll();
      if (stmt == null) stmt = em.db.con.prepareStatement(
          "SELECT (A.C_UNAME) AS COL0 "
        + "FROM Customer AS A WHERE (((A.C_ID)=?))");
      stmt.setInt(1, param0);
      rs = stmt.executeQuery();
      QueryList toReturn = new QueryList();
      while(rs.next()) { Object value = rs.getString(1);
                         toReturn.bulkAdd(value); }
      return toReturn;
} catch (SQLException e) { ... } finally {
   if (rs != null) try { rs.close(); } catch...
   stmtCache.add(stmt); }
```

Figure 4.24: Comparison of JDBC vs. JReq on the getUserName query

and due to JReq's ORing together of paths, this gets translated into a long query (Figure 4.25). This is not an actual problem with exponential path explosion since there is only one path for each OR, but each path is translated into a long conjunction of terms. The use of boolean algebra minimization techniques (like those used in IC circuit simplification) could solve this problem.

- Finally, one query (getIDandPassword) resulted in a query that was structurally identical to the original query, but it was more verbose due to the fact that it was machine-generated, resulting in slightly longer times to parse the query. This excess

Queryll query

```
i.getItemId() == i_id &&
  (i.getRelatedItem1() == j.getItemId() || i.getRelatedItem2() == j.getItemId()
    || i.getRelatedItem3() == j.getItemId() || i.getRelatedItem4() == j.getItemId()
    || i.getRelatedItem5() == j.getItemId())
```

SQL generated by JReq

```
      (((A.i_id)=?)  AND ((A.i_related1)=(B.i_id)))
OR    (((A.i_id)=?)  AND ((A.i_related1)!=(B.i_id))  AND ((A.i_related1)=(B.i_id)))
OR    (((A.i_id)=?)  AND ((A.i_related1)!=(B.i_id))  AND ((A.i_related1)!=(B.i_id))
                     AND ((A.i_related1)=(B.i_id)))
OR    (((A.i_id)=?)  AND ((A.i_related1)!=(B.i_id))  AND ((A.i_related1)!=(B.i_id))
                     AND ((A.i_related1)!=(B.i_id))  AND ((A.i_related1)=(B.i_id)))
OR    (((A.i_id)=?)  AND ((A.i_related1)!=(B.i_id))  AND ((A.i_related1)!=(B.i_id))
                     AND ((A.i_related1)!=(B.i_id))  AND ((A.i_related1)!=(B.i_id))
                     AND ((A.i_related1)=(B.i_id)))
```

Figure 4.25: Although the 5 ORs in the getRelated query do not result in path explosion, the ORs are still not translated very efficiently

verbosity can be handled by filtering out extraneous elements from the outputted query.

Overall though, all the queries from the TPC-W benchmark, a benchmark that emulates the behavior of real application, can be expressed in JQS, and JReq can successfully translate these JQS queries into SQL. JReq generates SQL queries that are structurally similar to the original hand-written queries for all of the queries. Although the machine-generation of SQL queries may result in queries that are more verbose and less efficient than hand-written SQL queries, by taking advantage of various optimizations that a normal programmer may find cumbersome to implement, JReq can potentially exceed the performance of hand-written SQL.

4.4.5 TPC-H

Although TPC-W does capture the style of queries used in database-driven websites, these types of queries make little use of more advanced query functionality such as nested queries. To evaluate JReq's ability to handle more difficult queries, some benchmarks have been run involving TPC-H [Tra08]. The TPC-H benchmark tests a database's ability to handle decision support workloads. This workload is characterized by fairly long and difficult ad hoc queries that access large amounts of data. The purpose of this experiment is to verify that the expressiveness of the JQS query syntax and JReq's algorithms for generating SQL queries are sufficient to handle long and complex database queries.

The 22 SQL queries and parameter generator from the TPC-H benchmark were extracted and modified to run under JDBC in Java. MySQL 5.0.51 was chosen for the database instead of PostgreSQL in this experiment in order to demonstrate JReq's ability to work with different backends. The following changes were required to the TPC-H queries to run them on MySQL:

- Query 1 was altered to remove the precision indicator during mathematics on dates since this feature is not supported by MySQL.
- For Query 13, the method used for naming columns was altered to be compatible with MySQL.
- Query 15 used temporary tables. Since JReq focuses on queries only, query variant 15a, rewritten to use nested queries instead of temporary tables, was used instead.

The queries were rewritten using JQS syntax. All 22 of the queries could be expressed using JQS syntax except for query 13, which used a LEFT OUTER JOIN, which was not supported in this version of JQS, as described in Section 4.2.4. To verify that the JQS queries were indeed semantically equivalent to the original queries, the query results between JDBC and JReq when run on a small TPC-H database using a scale factor of 0.01 were compared, and the results matched. This shows the expressiveness of the JQS syntax in that 21 of the 22 queries from TPC-H can be expressed in the JQS syntax and be correctly translated into working SQL code. JReq required approximately 33 seconds to translate the 21 JQS queries into SQL.

A TPC-H database using a scale factor of 1 was generated, resulting in a database about 1GB in size. Each of the 21 JQS queries from TPC-H were executed in turn using random query parameters, with a garbage collection cycle run in-between each query. The corresponding JDBC queries using the same parameters were then executed. This was repeated six times, with the last five runs kept for the final results. Queries that ran longer than one hour were canceled. A 2.5 GHz Pentium IV Celeron machine with 1 GB of RAM running Fedora Linux 9, and Sun JDK 1.5.0 Update 16 was used for the experiment. Table 4.5 summarizes the results of the benchmarks.

Unlike TPC-W, the queries in TPC-H take several seconds each to execute, so runtime optimizations do not significantly affect the results. Since almost all the execution time occurs at the database and since the SQL generated from the JQS queries are semantically equivalent to the original SQL queries, differences in execution time are mostly caused by the inability of the database's query optimizer to find optimal execution plans. In order to execute the complex queries in TPC-H efficiently, query optimizers must be able to recognize certain patterns in a query and restructure them into more optimal forms. The particular SQL generated by JReq uses a SQL subset that may match different optimization patterns in database query optimizers than hand-written SQL code.

- For example, the original query 16 evaluates a COUNT(DISTINCT) operation inside of GROUP BY. This is written in Queryll using an equivalent triply nested query, but MySQL is not able to optimize the query correctly, and running the triply nested query directly results in extremely poor performance.

Table 4.5: TPC-H benchmark results showing average time, standard deviation, and time difference (all results in seconds)

	JDBC		JReq				JDBC		JReq		
Query	Time	σ	Time	σ	Δ	Query	Time	σ	Time	σ	Δ
q1	73.5	0.4	71.9	3.4	(2%)	q12	23.4	0.5	29.7	0.2	27%
q2	145.4	2.2	146.0	1.9	0%	q14	491.7	8.9	500.8	10.1	2%
q3	37.9	0.6	38.6	0.9	2%	q15	24.9	0.7	24.8	0.6	(0%)
q4	23.0	0.5	23.8	0.2	3%	q16	21.3	0.6	> 1 hr	-	-
q5	209.1	4.2	206.1	3.2	(1%)	q17	2.1	0.2	11.0	3.6	429%
q6	15.2	0.3	15.8	0.3	4%	q18	> 1 hr	-	349.3	4.0	-
q7	79.1	0.5	83.1	1.6	5%	q19	2.8	0.1	18.1	0.4	540%
q8	48.8	1.7	51.0	1.9	4%	q20	69.4	4.3	508.4	11.4	633%
q9	682.0	97.4	690.2	97.9	1%	q21	245.5	3.2	517.0	7.1	111%
q10	47.1	1.0	47.2	0.5	0%	q22	1.1	0.0	1.6	0.0	43%
q11	41.7	0.6	41.9	0.7	1%						

- Oddly, in query 18, JReq's use of deeply nested queries instead of a more specific SQL operation (in this case, GROUP BY...HAVING) fits a pattern that MySQL is able to execute efficiently, unlike the original hand-written SQL. Because of the sensitivity of MySQL's query optimizer to the structure of SQL queries, it will be important in the future for JReq to provide more flexibility to programmers in adjusting the final SQL generated by JReq.

- Queries 20 and 21 in Queryll are slower than the hand-written SQL because the queries use the IN and EXISTS keywords several times, but Queryll's syntax currently does not provide a way to express the meanings of these keyword directly, so instead they are expressed by counting the number of elements that match and checking if the count is greater than 0.

- In queries 17 and 22, the Queryll queries were slower than the original because Queryll does not currently have direct support for calculating averages, so averages are calculated indirectly by taking the total of the data and dividing by the number of elements. When calculating averages over large subqueries, this approach is slower.

- Finally, queries 7, 12, and 19 are slower in JReq because the queries make use of OR, which can result in a large number of long paths being generated by the JReq algorithm, resulting in longer queries.

Overall, 21 of the 22 queries from TPC-H could be successfully expressed using the JQS syntax and translated into SQL. Only one query, which used a LEFT OUTER JOIN, could not be handled because JQS and JReq do not currently support the operation yet. For most of the queries, the JQS queries executed with similar performance to the original queries. Where there are differences in execution time, most of these differences can be eliminated by either improving the MySQL query optimizer, adding special rules to the SQL generator to generate patterns that are better handled by MySQL, or extending the syntax of JQS to allow programmers to more directly specify those specific SQL keywords that are better handled by MySQL.

4.5 Summary

The JReq system translates database queries written in the imperative language Java into SQL. Unlike other systems, the algorithms underlying JReq are able to analyze code written in imperative programming languages and recognize complex query constructs like aggregation and nesting. In developing JReq, a syntax for database queries that can be written entirely with normal Java code was created, an algorithm based on symbolic execution to automatically translate these queries into SQL was designed, and a research prototype of the system that shows competitive performance to hand-written SQL was implemented.

Chapter 5

HadoopToSQL: MapReduce-Style Queries

In object-oriented imperative languages like Java, large datasets are typically processed by using a loop to iterate over the records of the dataset. The JReq system demonstrated how to build a query language using such a syntax. There are alternate approaches to processing large datasets in languages like Java though.

Programmers are increasingly using MapReduce [DG04] for performing queries over large datasets. With MapReduce, programmers write queries by defining two functions—map and reduce—for filtering, processing, and grouping records together. MapReduce is popular because it transparently handles many of the difficulties of processing data on clusters of commodity hardware, including issues such as fault tolerance, data transfer, and data partitioning. Although initially used for log-processing, it has now been applied to new workloads such as scientific computing [CS08] and business decision support systems [KJH+08].

Although MapReduce is used for processing large amounts of data, MapReduce code cannot automatically use database features like indices to improve its performance [PPR+09]. Programmers have started using the MapReduce abstraction with advanced storage engines that support database features [CDG+06] instead of cluster file systems, but to make use of the database features, programmers must write their database operations separately from their MapReduce code. These database operations are typically written in their own separate query language.

With HadoopToSQL, programmers can write their code for processing large datasets entirely within the MapReduce framework. HadoopToSQL can then analyze the code and automatically extract database operations that can be used to improve the performance of the code. It operates on MapReduce queries written for the Hadoop [Apa] open-source MapReduce implementation. Hadoop queries are written using normal Java code. Unlike JReq, which focuses on allowing programmers to describe database operations by working with a restricted subset of the Java language, HadoopToSQL cannot impose such restrictions. Much of the power and usefulness of MapReduce comes from the fact that it allows arbitrary computation inside the map and reduce functions. As a result, HadoopToSQL

```
function map(LogEntry, output):
     output.collect(LogEntry.Country, 1);

function reduce(Country, Iterator, output)
     int sum = 0;
  loop:
     if !Iterator.hasNext() goto end
     Iterator.next();
     sum += 1;
     goto loop
  end:
     output.collect(Country, sum);
```

Figure 5.1: Pseudocode for a MapReduce query that counts the LogEntries for each country.

```
  SELECT A.Country, COUNT(*)
    FROM LogEntry A
GROUP BY A.Country
```

Figure 5.2: HadoopToSQL is able to analyze the MapReduce query from Figure 5.1 and generate this equivalent SQL query.

acts more as a query optimizer that optimizes MapReduce code by finding code that can be more efficiently run as a database operation and rewriting them. In certain cases though, HadoopToSQL is able to translate a MapReduce query entirely to SQL. For example, the MapReduce query in Figure 5.1 can be translated to the equivalent SQL query in Figure 5.2. If HadoopToSQL is not capable of generating an equivalent SQL query, it tries to find input restrictions for the query so that the query can take advantage of indexing features of SQL storage engines.

This work makes the following research contributions:

- Algorithms are presented for analyzing and understanding MapReduce code.
- This understanding is shown to enhance MapReduce performance by using the database features of advanced storage engines.
- These algorithms have been implemented and evaluated to demonstrate the performance benefits of this approach.

5.1 Background and Motivation

MapReduce is a data processing model designed primarily for large clusters of machines. In a MapReduce cluster, all data is stored as (key, value) pairs. There may be multiple values

```
function map(key1, value1) : (key2, value2)*
function reduce(key2, value2*) : (key3, value3)*

foreach (key1, value1) in dataset
  temp.addAll(map(key1, value1))
temp.sort()
foreach (key2) in temp.keys()
  result.addAll(reduce(key2, temp[key2]))
```

Figure 5.3: A conceptual view of how a MapReduce query is executed.

per key. To perform a query across this data, programmers must define two functions: map and reduce. In Hadoop, map functions take a (key, value) pair as input and output zero or more new (key, value) pairs. Then, the system sorts these new (key, value) pairs. For each key, all the values that correspond to that key are passed as input to the reduce function, which then generates zero or more new (key, value) pairs as output. The final set of (key, value) pairs is saved as the result of the query. Figure 5.3 shows a conceptual view of how a MapReduce query is executed. Typically, programmers write the code for these two functions using a conventional imperative programming language.

MapReduce is popular because it provides a powerful yet simple-to-understand abstraction that hides many of the difficulties of performing queries on large computer clusters such as dealing with inter-machine communication bottlenecks and machine failure. In practice, MapReduce queries scale well to giant datasets stored across large machine clusters. Since MapReduce is designed to handle failure-prone hardware, it works well with clusters built using commodity hardware, hence providing excellent scalability to large datasets at a reasonable cost.

It is possible to use a traditional declarative query language like SQL or Hive for the same domain [PPR+09, SAD+10]. However, queries that need to perform complex computation are ill-suited for declarative query languages but are easily expressed in MapReduce. MapReduce programs are written in conventional imperative programming languages such as Java. Therefore, it is easy to include arbitrary computation such as a complicated AI classifier or mathematical computation. Such computation cannot be expressed directly in declarative query languages but must be programmed externally and then imported into the query language using user-defined functions and stored procedures.

In the research literature, MapReduce has traditionally been used within the context of log-processing workloads [PDGQ05, ORS+08]. For example, a MapReduce query needs to examine all the log entries of visits to a website to find the most popular web pages on that site. Since these workloads typically require that every record in a dataset be examined, MapReduce is usually paired with a basic cluster file system as a storage engine. All record entries can then easily be streamed off the file system and into the map function.

There are workloads, though, that access only subsets of a dataset. For example, a business might want to analyze their sales in a certain region within a specific date range. For these workloads, streaming through every record in a dataset is extremely inefficient.

Indexing the dataset in advance and then using the index to restrict which records are examined is potentially much faster and more efficient. In order to support this possibility, MapReduce needs to be run using an advanced storage engine that supports indexing, and MapReduce queries must be rewritten to take advantage of these storage engine features. Instead, MapReduce code can be analyzed in order to automatically extract information about the subset being accessed.

This analysis is not straight-forward because MapReduce supports arbitrary computation in its map and reduce functions. As a result, any MapReduce query optimizer must be able to analyze arbitrary code in order to extract possible optimizations. HadoopToSQL is designed to optimize Hadoop MapReduce code, in which map and reduce functions are expressed using Java. Since there are at present no advanced storage engines purpose-built for MapReduce, the optimizations have been targeted towards an SQL storage engine.

There already exist possible scenarios where programmers may want to run MapReduce on top of SQL databases. For example, some firms horizontally partition large SQL datasets across many small commodity machines [Per]. In such a configuration, queries that access data on only a single machine are fast, but more complex queries that aggregate data across the machines require the use of a distributed SQL database [DGS88, PPR+09] or distributed middleware layer [ST , Spo]. These firms may choose to use MapReduce for this purpose. Even if a company has an SQL database that fits entirely on a single server, it might decide to write its queries using MapReduce if it believes it will eventually build a MapReduce cluster for data warehousing.

Ultimately, though, HadoopToSQL targets SQL storage engines because they are readily available. The main purpose of HadoopToSQL is to demonstrate that static analysis can be used to better understand MapReduce queries. This understanding can be used to adapt MapReduce code automatically to take advantage of advanced storage engines.

5.2 Transformations

The key innovation in HadoopToSQL is a static analysis component that uses symbolic execution to analyze the Java code of a MapReduce query. It transforms queries to make use of SQL's indexing, aggregation, and grouping features. HadoopToSQL offers two algorithms that generate SQL code from MapReduce queries. One algorithm can extract input set restrictions from MapReduce queries, and the other can translate entire MapReduce queries into equivalent SQL queries. Both are intra-procedural algorithms. They function by finding all control flow paths through map and reduce functions, using symbolic execution to determine the behavior of each path, and then mapping this behavior onto possible SQL queries. HadoopToSQL analyzes all MapReduce queries using both techniques. Since translating entire queries into SQL offers more performance benefits than simply finding input set restrictions, that optimization is preferred if both can be applied to a particular query. If none are applicable, then the query is run without optimization.

5.2.1 Input Set Restrictions in the Map Function

Since database queries tend to be very data-intensive, one of the most important optimizations that can be performed is to reduce the amount of data that needs to be processed. MapReduce queries that operate on only a subset of a dataset can be greatly optimized if HadoopToSQL is able to extract the shape of this subset from the query code and apply this shape as a constraint on the input set of the queries. For example, given a database of a company's sales, a query that analyzes the sales of a certain region only needs to be supplied with data from that region.

Conceptually, HadoopToSQL's algorithm for finding input set restrictions works by tracing through different possible execution paths of the map function. As HadoopToSQL follows the paths of these traces, it records the constraints on variables that need to hold for each trace to occur. If a trace does not result in output being generated, then the trace is ignored. If a trace does result in output being generated, then the input constraints that trigger the trace are included in the input set. There can also be traces that HadoopToSQL cannot fully analyze such as traces with calls to unknown methods. When faced with such imprecise knowledge, HadoopToSQL must make the conservative assumption that this trace generates output. As such, the input constraints that trigger the trace are also included in the input set. The resulting restrictions are not "tight" but do not exclude any data unintentionally.

HadoopToSQL generates these traces by performing a depth-first walk of all paths through the control flow graph of the map function, starting at the entry point and ending at the function exit. It stops traversing along a path upon encountering a loop (which can lead to infinitely long paths) or a statement with unknown side-effects. It then labels that path as not fully analyzable. Statements with unknown side-effects include essentially all method calls, but HadoopToSQL knows about common methods with no side-effects like `String.equals()`, methods of automatically generated entity objects, and methods that are necessary for MapReduce such as `Output.collect()`. This approach to path traversal leads to HadoopToSQL being most effective at finding input constraints in programs that filter their input as early as possible.

To calculate the constraints on variables that need to hold for a trace to occur, HadoopToSQL uses symbolic execution to calculate the preconditions and postconditions of executing the statements of a path. Essentially, each branch on a path becomes a precondition of the path, and each method call and variable assignment to a non-local variable becomes a postcondition. For *each* path that might generate output, HadoopToSQL takes the various preconditions of the path and creates a single precondition expression for the path by ANDing them together. This expression describes the input that triggers the execution of the path. HadoopToSQL then takes these expressions for each path and ORs them all together. This results in a boolean expression that can be used to restrict the input set to the query.

Figure 5.4 shows an example map function, which will be used to illustrate how the input set restriction algorithm works. The function includes a call to a `classify()` method that potentially contains a complicated algorithm for classifying sales into different categories and sizes. HadoopToSQL first enumerates all paths through the method, truncating

```
function map(Sale, Output):
     if Sale.Region() == "East" goto end
     if Sale.Region() != "North" goto output
     Classification = classify(Sale)
     if Classification.Size() <= 5 goto end
  output:
     Output.collect(
        Classification.SalesCategory(), Sale)
  end:
     return
```

Figure 5.4: Pseudocode of a map function that analyzes the sales in a certain region.

paths that include the `classify()` method since the method has unknown side-effects (Figure 5.5).

HadoopToSQL then uses symbolic execution on each path to determine the preconditions and postconditions of each path. Figure 5.6 shows the preconditions and postconditions of the two paths from Figure 5.5.

HadoopToSQL knows that the method `Sale.Region()` has no side-effects because it is an accessor method of an automatically generated entity object. It can thus determine that path 3 does not generate output, that path 2 obviously does generate output, and that path 1 is not fully analyzable. As a result, it uses the input constraints of path 1 and path 2 to generate input set restrictions for the query. The individual preconditions of each path are ANDed together to form input constraints for the path. These expressions are then ORed together, resulting in the final input set restrictions, which may contain redundant terms (Figure 5.7). The code for reading data into the map function can then be modified to include a WHERE clause with these input set constraints (Figure 5.8). Although the final WHERE clause may be amenable to further simplification, this task is left to the SQL query engine.

5.2.2 Complete Translation to SQL

HadoopToSQL's second transformation algorithm can translate entire MapReduce queries into a single SQL query. Such a query can be more efficient than a normal MapReduce query by reading only the fields of a record that are used by the query. It can also make use of aggregation optimizations in SQL databases. For example, a query might divide its data into a large number of categories based on whether the value of a field fits within certain ranges. It might then calculate aggregates for each category. If a database has sorted its dataset by the same field, it can calculate these aggregates with a single pass through the data. Finally, a query that is fully translated to SQL can also make use of input constraints.

Unfortunately, since the query model supported by MapReduce cannot be mapped directly onto the SQL query model, this transformation is only feasible for certain classes

```
Path 1:
if Sale.Region() == "East"      (branch not taken)
if Sale.Region() != "North"     (branch not taken)
Classification = classify(Sale) (path traversal
                                 aborted)

Path 2:
if Sale.Region() == "East"      (branch not taken)
if Sale.Region() != "North"     (branch is taken)
    goto output
Output.collect(
  Classification.SalesCategory(),
  Sale)

Path 3:
if Sale.Region() == "East"      (branch is taken)
    goto end
```

Figure 5.5: HadoopToSQL finds three paths through the map function.

of MapReduce queries. HadoopToSQL can only translate MapReduce queries fulfilling these general properties into SQL queries:

For the map function:

- Any execution of the map function can emit at most one (key, value) pair.
- The function can make arbitrary use of `if` statements but it cannot contain any loops.
- The function can only use operators and functions that exist in SQL.
- The function can create and modify only local variables. These variables must have types that are compatible with SQL.

For the reduce function:

- The reduce function must emit exactly one (key, value) pair.
- The (key, value) pair output by the function must use same key that is used for its input (key, value) pairs.
- The function can only use operators and functions that exist in SQL.
- The function can create and modify only local variables. These variables must have types that are compatible with SQL.

```
Path 1 Preconditions:
   Sale.Region() != "East"
   Sale.Region() == "North"

Path 1 Postconditions:
   Sale.Region()
   (traversal aborted)

Path 2 Preconditions:
   Sale.Region() != "East"
   Sale.Region() != "North"

Path 2 Postconditions:
   Sale.Region()
   Output.collect(
      Classification.SalesCategory(), Sale)

Path 3 Preconditions:
   Sale.Region() == "East"

Path 3 Postconditions:
   Sale.Region()
   (exit function)
```

Figure 5.6: By using symbolic execution, HadoopToSQL is able to determine the preconditions and postconditions of executing each path through the code.

```
Path 1 Precondition Expression:
   Sale.Region() != "East"
   AND Sale.Region() == "North"

Path 2 Precondition Expression:
   Sale.Region() != "East"
   AND Sale.Region() != "North"

Final Boolean Expression:
      (Sale.Region() != "East"
           AND Sale.Region() == "North")
   OR (Sale.Region() != "East"
           AND Sale.Region() != "North")
```

Figure 5.7: From the preconditions of each path, HadoopToSQL is able to derive a boolean expression describing the input set restrictions.

```
ResultSet rs = execute(
      "SELECT * FROM Sale A"
      + " WHERE (A.Region <> 'East' "
      + "          AND A.Region = 'North')"
      + "    OR (A.Region <> 'East' "
      + "          AND A.Region <> 'North')";
while (rs.next())
  Sale s = new Sale(rs);
  apply map to s
```

Figure 5.8: Pseudocode for how the input set constraint appears in the WHERE clause of an SQL query for feeding data into a map function.

- The reduce function should either be the identity function, or it should iterate over its input values and compute some sort of aggregation that is compatible with SQL.

Most of these properties are the result of the inherent restrictions of the SQL query syntax and are not due to inflexibility in the transformation algorithm. For example, an SQL query can output at most one output row for each input row processed, so for a map function to be translated into an SQL query, it too can only output at most one (key, value) pair for each input (key, value) pair.

Conceptually, the transformation that HadoopToSQL performs is that it tries to fill in a stencil of a SELECT...FROM...WHERE...GROUP BY query based on the behavior of the map and reduce functions. HadoopToSQL extracts an input restriction from the map function, and uses it as the WHERE clause of the SQL query. The (key, value) pair generated by the map function is used as the SELECT clause of the SQL query. If the reduce function calculates an aggregation, then a GROUP BY clause is added to the query with a grouping based on the key, and the SELECT clause is modified to aggregate the values computed in the map.

The analysis of the map function is performed using the same method as described in Section 5.2.1, but HadoopToSQL needs to fully understand the behavior of the code instead of merely calculating a conservative approximation. Once the map code is broken up into paths and after the preconditions and postconditions have been calculated through symbolic execution, HadoopToSQL can use the path preconditions to compute an expression for the WHERE clause. There should be no ambiguous operations in the preconditions, so the resulting input set restrictions are exact. Since the data being output by the map function are encoded in the path postconditions, HadoopToSQL can simply extract the expressions being output and use them in the SELECT clause.

For example, consider the map and reduce functions in Figure 5.9. The program divides sales into two categories—one category for the "North" region and one category for the others—and calculates the total commission on sales for each category. There are two paths through the map function, both of which generate output. Figure 5.10 shows the preconditions and postconditions for the two paths, and it shows how to derive a single

```
function map(Sale, Output):
     if Sale.Region() != "North" goto L1
     Output.collect("North", Sale.Commission())
     goto mapend
  L1:
     Output.collect("NotNorth", Sale.Commission);
  mapend:
     return

function reduce(key, Iterator, Output):
     sum = 0
  loop:
     if !Iterator.hasNext() goto reduceend
     sum = sum + Iterator.next()
     goto loop
  reduceend:
     Output.collect(key, sum)
     return
```

Figure 5.9: Pseudocode for a MapReduce program that can be translated completely into SQL.

SELECT clause that is equivalent to the two paths.

The extraction of aggregation information from the reduce function is more involved because the reduce function must use a loop to iterate over its input. The loop in the function is found by using a strongly-connected components algorithm. All the paths through this loop are enumerated and the preconditions and postconditions for each path are calculated using symbolic execution (Figure 5.11). HadoopToSQL has a series of template patterns that describe how various SQL aggregation operations are expressed as a loop's preconditions and postconditions (Figure 5.12). By matching these templates against the loop, it is able to identify which SQL aggregation is being used. HadoopToSQL currently has templates for recognizing SQL's SUM, MIN, and MAX aggregation operations. The templates look for loops that iterate over a collection and that collect a result in a single variable.

HadoopToSQL can then analyze the non-loop code of the reduce function by again using symbolic execution to calculate path preconditions and postconditions. The symbolic execution engine treats the loop as a single statement that calculates an aggregation. If the rest of the reduce code satisfies all the reduce function properties described earlier in this section, then the key is added as a GROUP BY to the query, and aggregation operations are applied to the values in the SELECT clause. Since MapReduce results appear in sorted order, HadoopToSQL also adds an ORDER BY clause to the final query.

So in the example, if the loop is found to match a template for a SUM aggregation, then the loop of the reduce function is replaced by a single statement summarizing the effect of

```
Path 1 Preconditions:
     Sale.Region() == "North"
Path 1 Postconditions:
     Output.collect("North", Sale.Commission())

Path 2 Preconditions:
     Sale.Region() != "North"
Path 2 Postconditions:
     Output.collect("NotNorth", Sale.Commission())

SELECT CASE WHEN A.Region = "North" THEN "North"
            ELSE "NotNorth" END,
       A.Commission
  FROM Sale A
```

Figure 5.10: The SELECT clause of an SQL query can be computed based on the preconditions and postconditions of the paths through the map function. This SELECT clause calculates only two fields: one for the map function's key and one for the map function's value. Because the key differs based on the input, a CASE statement is required.

```
Path 1 Preconditions:
     !Iterator.hasNext()
Path 1 Postconditions:
     exit loop

Path 2 Preconditions:
     Iterator.hasNext()
Path 2 Postconditions:
     Iterator.next()
     sum = sum + Iterator.next()
```

Figure 5.11: The loop of the reduce function in Figure 5.9 has these preconditions and postconditions, which indicate that it calculates a SUM() aggregation.

```
Path i Preconditions:
     !Iterator.hasNext()
     ...
Path i Postconditions:
     exit loop

Path n Preconditions:
     Iterator.hasNext()
     ...
Path n Postconditions:
     Iterator.next()
     sum = sum + expression
```

Figure 5.12: Template pattern for identifying a loop as a SUM aggregation. In the template patterns for MAX and MIN aggregation, the addition operation is replaced by a max and min operation respectively.

```
function reduce(key, Iterator, Output):
     sum = 0
 loop:
     sum += SUM(Iterator values)
 reduceend:
     Output.collect(key, sum)
     return
```

Figure 5.13: The loop inside the reduce function is replaced by a statement summarizing the effect of the loop.

the loop (Figure 5.13). Symbolic execution is then applied to the entire reduce function to calculate postconditions, which reveals that the reduce function encodes a GROUP BY query (Figure 5.14). The SELECT clause used to calculate the outputted key and value of the map function can then be merged into a GROUP BY stencil to produce a final SQL query for the combined map and reduce functions (Figure 5.15). The final query may contain expressions that can be further simplified, but this task is left to the SQL query engine.

5.3 Implementation Details

The HadoopToSQL system consists of a static analysis component and a runtime component. The static analysis component is applied to the Java bytecode of a MapReduce query. It attempts to apply different transformations to the code to try to find an efficient way to execute the code on an SQL database. The runtime component provides a simple

```
Path Postconditions:
     Output.collect(key, 0 + SUM(value))

Matching GROUP BY stencil:
  SELECT key, SUM(value)
    FROM ...
GROUP BY key
ORDER BY key
```

Figure 5.14: If the postconditions for the non-loop portions of the reduce function show that the function satisfies the needed properties, the SQL query can be converted to use a GROUP BY and aggregation.

object-relational mapping tool to simplify access to database entities. It includes runtime libraries for mapping the SQL data model to fit the MapReduce data model.

5.3.1 Static Analysis Component

The static analysis component of HadoopToSQL is implemented as a bytecode rewriter. It is able to take a compiled MapReduce program generated by the Java compiler and analyze it to find ways to run it efficiently on an SQL database.

Although the HadoopToSQL bytecode rewriter accepts Java bytecode as input, its internal processing is actually based on a representation called Jimple, a three-address code version of Java bytecode. It uses the SOOT framework [VRCG$^+$99] from Sable to transform Java bytecode to this representation. Raw Java bytecode is difficult to process because of its large instruction set and the need to keep track of the state of the operand stack. In Jimple, there is no operand stack. There are only local variables, meaning that HadoopToSQL can use one consistent abstraction for working with values.

The static analysis component outputs a data structure that contains descriptions of how various map and reduce functions can be translated to SQL. The HadoopToSQL runtime component can then query this data structure when deciding on a procedure for executing MapReduce queries. Because of HadoopToSQL's design as a bytecode rewriter, it can be added to the toolchain as an independent module, with no changes needed to existing IDEs, compilers, virtual machines, or other such tools.

5.3.2 Runtime Component

HadoopToSQL contains various runtime libraries for allowing an SQL storage model to mix with a MapReduce approach to data.

For example, with MapReduce, data records are typically stored as text in files, whereas in SQL, data records are stored as relations in tables. Neither storage representation is particularly convenient for programmers, who prefer mapping these representations to an object representation inside their programs. HadoopToSQL includes a simple object-relational mapping (ORM) tool that can perform this mapping of either text or relations

```
Output of the map function:
SELECT
       /* Key */
       CASE
         WHEN A.Region = "North" THEN "North"
         ELSE "NotNorth" END,
       /* Value */
       A.Commission
  FROM Sale A
 WHERE /* Input restriction */
       A.Region = "North"
       OR A.Region <> "North"

Stencil for the reduce function's GROUP BY:
  SELECT key, SUM(value)
    FROM ...
   WHERE input restriction
GROUP BY key
ORDER BY key

Final SQL query:
  SELECT CASE
         WHEN A.Region = "North" THEN "North"
         ELSE "NotNorth" END,
         SUM(A.Commission)
    FROM Sale A
   WHERE A.Region = "North"
         OR A.Region <> "North"
GROUP BY CASE
         WHEN A.Region = "North" THEN "North"
         ELSE "NotNorth" END
ORDER BY CASE
         WHEN A.Region = "North" THEN "North"
         ELSE "NotNorth" END
```

Figure 5.15: Merging the SELECT clause of the map function with the GROUP BY stencil of the reduce function results in the final SQL query.

to entity objects. This hides the differences between the two storage models and provides a more convenient interface for programmers. Programmers provide an XML description of a schema, and the ORM tool creates corresponding entity object classes as well as code for reading these objects from either a text file or from an SQL database. Programmers can then express their MapReduce programs in terms of manipulating these objects instead of needing to write code for parsing text input or for querying databases.

The Hadoop implementation of MapReduce provides a `FileInputFormat` object for reading lines of text from files. The HadoopToSQL library provides alternate objects that can read their data from either databases or files and that can return ORM entity objects instead of lines of text. To switch between using an SQL database as storage engine as opposed to a MapReduce distributed file system, programmers merely have to change the configuration information of their MapReduce queries to use the HadoopToSQL libraries for managing their input.

5.4 Experimental Evaluation

To evaluate HadoopToSQL, some single-server experiments and one distributed experiment were run. The single server experiments allow the performance of SQL queries generated by HadoopToSQL to be compared directly with the performance of hand-written SQL queries run on a standard single-server database. The distributed experiment verifies that the performance benefits of HadoopToSQL still hold on a cluster. For all of the experiments, data is loaded into databases and indexed before the experiments are run.

5.4.1 Single-Server Experiments

The single-server experiments are run on a dual-processor Pentium IV Xeon machine with 4 GB of RAM running Linux, OpenJDK 1.6, Hadoop 0.20, and PostgreSQL 8.3. Hadoop is configured for stand-alone operation, with its input and output files stored on the local disk.

Stock benchmark

To illustrate the behavior of HadoopToSQL, a benchmark was created involving a database of synthetic stock market prices. The database consists of 10,000 different stocks. For each stock, the database tracks the daily closing price and trading volume. To examine the effect of database size, the number of days of historical stock data can be varied between 500 and 3,500 days. When stored in an SQL database, the historical data uses the stock symbol and date as a primary key. A text dump of a database with 3,500 days of data is 970 MB in size.

The benchmark executes a query that calculates sums of 15 different stocks over a period of five months. This query is inspired by the type of computation involved in calculating stock market indices like the Dow Jones Industrial Average. The performance of this query is measured in the following configurations:

- Hand-written SQL
- MapReduce running on a single machine
- MapReduce running on SQL without any optimizations by HadoopToSQL
- MapReduce running on SQL with input set restrictions calculated by HadoopToSQL
- MapReduce running on SQL with a full translation by HadoopToSQL to SQL

Figure 5.16 shows the query times for each of these variations. Each data point is the average of 10 query executions with each execution using a random set of 15 stocks. Both regular MapReduce and MapReduce on SQL without optimizations exhibit increasing query time as the database size increases. This is caused by the fact that both variations must scan through the entire database in order to find the stocks and days relevant to the query. As the database size increases, the queries must examine more data as well. For example, given 3,500 days of stock data, a full scan of the dataset needs to examine 35 million records. The performance of MapReduce on SQL without optimizations is approximately 50% worse than that of regular MapReduce. This is due to the fact that it must perform a table scan of an SQL database instead of reading its data from text files like regular MapReduce. Although an SQL database can theoretically store its data in a more compact representation than the textual representation used in MapReduce, SQL databases are rarely optimized for this sort of access pattern, so they do not necessarily fill disk blocks to the maximum extent or arrange data sequentially on disk. By contrast, linear traversals of files is a well-optimized access pattern for operating systems.

The granularity of the y-axis in Figure 5.16 hides significant detail, so Figure 5.17 is included in this chapter to show an enlarged view of the same data. Hand-written SQL, the restricted input set configuration, and the full translation configuration are all able to indicate to the underlying database that they only want a subset of the data. As such, the SQL database is able to make use of underlying indices to ignore the extra data in the database, meaning that these queries only need to examine approximately 2,000 records. Although all three configurations process the same number of records, the full translation configuration spends time creating XML configuration files, starting a Hadoop MapReduce engine, sending the configuration information to the MapReduce engine, and other non-query-related overhead. This configuration is thus half a second slower than hand-written SQL despite the fact that both configurations execute essentially the same SQL query against the database. Due to the short running time of the query, Figure 5.17 exaggerates the size of this overhead. The restricted input set configuration runs a full execution of MapReduce, applying the map and reduce functions to its data, so it has the worst performance of the three.

This benchmark shows the importance of using indices in order to extract the best performance for MapReduce queries running on an SQL database. The MapReduce query model has no notion of indices since the information about which data is used by a query is encoded in the program code itself, which cannot normally be inspected by a MapReduce runtime. The program analysis performed by HadoopToSQL is able to extract this information and hence take advantage of the indices available in SQL.

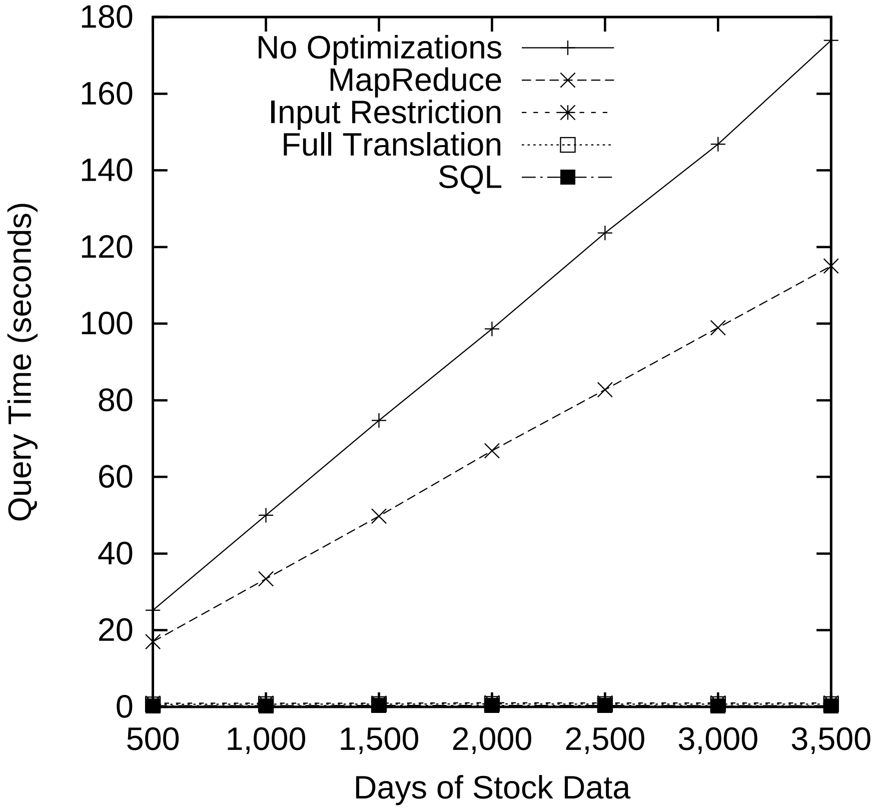


Figure 5.16: Query time on stock benchmark as database size increases.

TPC-H

TPC-H [Tra08] is a standard SQL database benchmark for decision-support workloads. This benchmark is included in the experiments because it allows for easy comparison of MapReduce results with SQL results and because it provides an interesting business-oriented workload. Nonetheless, the results must be interpreted with caution because a direct translation of TPC-H queries to MapReduce does not necessarily reflect how such queries would be written and how the schema would be designed if the benchmark specifically targeted a MapReduce query model.

This experiment examines queries Q1 and Q3 of TPC-H, which map well to MapReduce and have non-trivial running times. The benchmark is configured with a TPC-H scale factor of one, resulting in a dataset of approximately 1,100 MB in size. The experiment uses random query parameters as specified by TPC-H.

Query Q1 scans a single table of order line items within a certain date range and calculates aggregates for different categories. Figure 5.18 shows the query results for query Q1. Similar to the stock benchmark, HadoopToSQL is able to extract an equivalent SQL query from the MapReduce code. As a result, the translated query is able to make

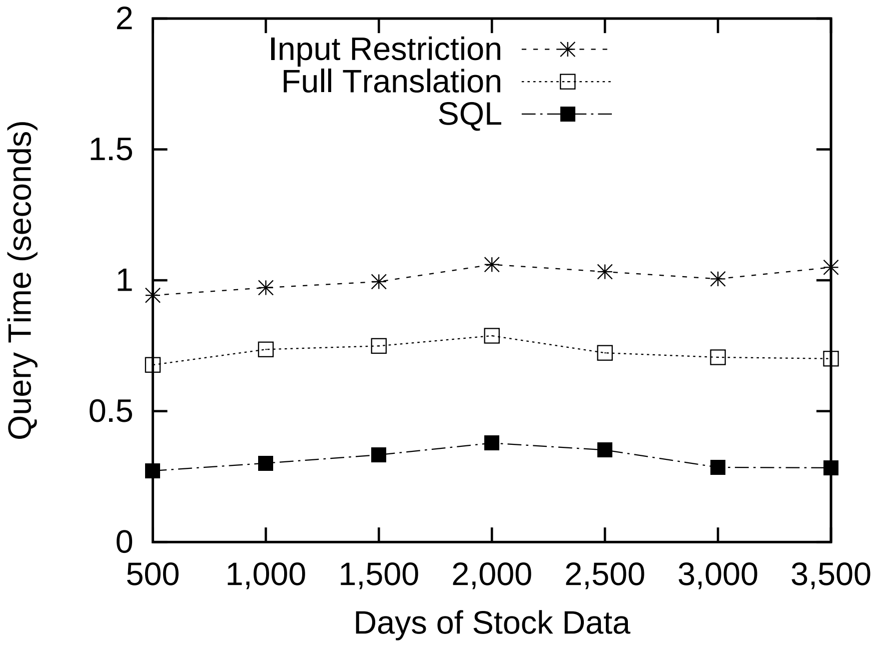


Figure 5.17: Query time on stock benchmark as database size increases (with zoomed y-axis).

use of database indices, resulting in much better performance than regular MapReduce, which must scan the entire contents of the text file of line order items. The translated query also exhibits performance that is almost as good as hand-written SQL.

Unlike query Q1, query Q3 involves a database join. Query Q3 examines customer, order, and order line item information to determine the 10 highest-valued orders with certain characteristics and that have not been shipped. It needs to join the customer, order, and line item entities in computing its result. Since MapReduce does not have any built-in support for joins (joins are fundamentally slow operations when applied to data in a cluster), programmers normally structure their data differently if they intend to query it with MapReduce. In particular, programmers denormalize their data in advance to avoid the poor performance of joins in MapReduce. To reflect this fact, different data layouts were used for each configuration.

The SQL query is run using separate tables for each of the three entities. HadoopToSQL also stores the data in three separate tables, but the tables are joined at runtime and presented to the map function as a single table, much like an SQL view. For regular MapReduce, the query is run against a file with the three entities joined in advance. The coding of the MapReduce and HadoopToSQL versions of the queries have one potential inefficiency as compared to the SQL query. TPC-H specifies that for query Q3, only the top 10 results are needed. In the MapReduce and HadoopToSQL versions of the queries, the top 10 results are found by calculating all the results and sorting them—it is potentially

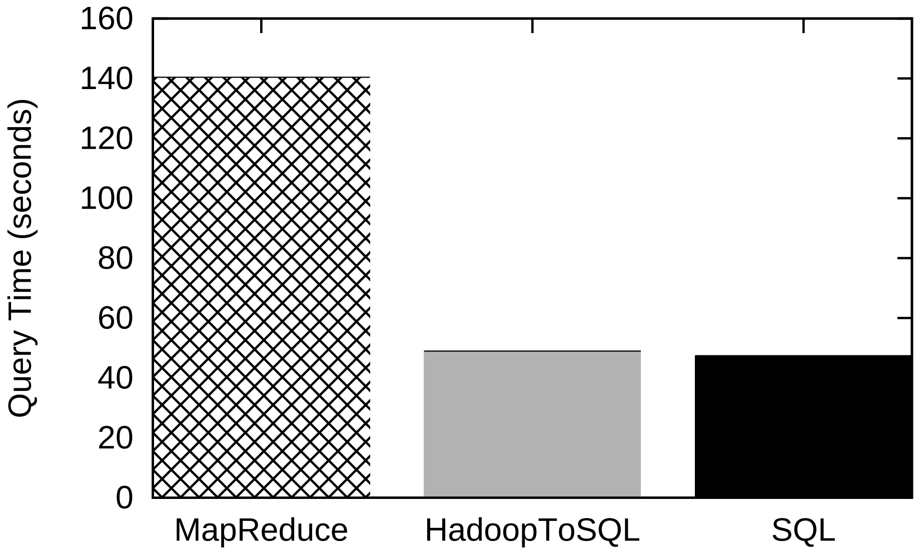


Figure 5.18: Query time for TPC-H query Q1.

more efficient to calculate the top 10 results directly.

Figure 5.19 shows the execution times for TPC-H Q3. Regular MapReduce is significantly slower than both HadoopToSQL and SQL. This is due to the fact that it must scan through all the records of the dataset without being able to restrict itself to only those orders that have not yet shipped and that satisfy the expected characteristics. HadoopToSQL is able to extract useful input constraints from the query and is hence able to achieve comparable performance to the hand-written SQL version of the query.

For completeness, versions of the MapReduce and HadoopToSQL queries that can operate on a dataset that has not been denormalized were also created. This requires that the Customer, Order, and LineItem records be joined during query execution, which are emulated using multiple MapReduce steps. The HadoopToSQL version of query Q3 uses six different applications of MapReduce to calculate its result: three stages filter and reformat input records, two stages join these records together and aggregate the results, while a final stage is used to sort the results.

Figure 5.20 shows the resulting execution times. The individual times of each of the six MapReduce stages are shown where applicable. HadoopToSQL is able to improve the performance of the MapReduce query when it is run on an SQL database, but it is not able to achieve performance comparable to that of a hand-written SQL query (unlike with the denormalized version of the query). The problem is that HadoopToSQL only optimizes within a single application of MapReduce. HadoopToSQL is not able to optimize across the six MapReduce stages of this version of the query.

The TPC-H benchmark shows that HadoopToSQL can be used to improve the performance of real queries. For a MapReduce query to achieve comparable performance to

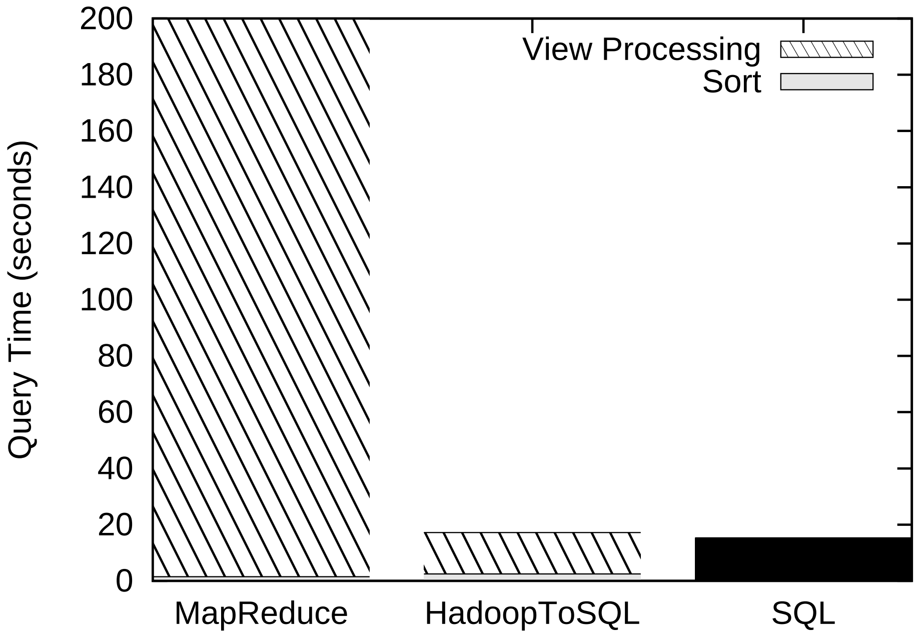


Figure 5.19: Query time for TPC-H query Q3 when the Customer, Order, and LineItem tables are joined in advance.

SQL on a single server, it is important to extract as many input constraints on the query as possible so as to reduce the amount of data that needs to be processed. HadoopToSQL is effective at extracting such constraints from within a single application of MapReduce, but it is currently not able to extract constraints across multiple MapReduce stages.

5.4.2 Distributed Behavior

To evaluate whether the benefits of HadoopToSQL still hold in the distributed case, where there is additional communication and coordination overhead, An experiment involving a small cluster of machines has been created. It uses the Selection Task from the paper of Pavlo et al. [PPR$^+$09]. This task involves scanning a list of PageRanks for the URLs of different web pages. The task outputs those URLs with a PageRank greater than the parameter 10.

Configured using the default parameters, the data generation code from the paper generates 5.6M ranking records per data node in the cluster, for a total size of about 300MB per node. For the SQL and HadoopToSQL configurations, the dataset is divided into equal-sized partitions. An SQL database is running on each data node, and each data node stores one of these partitions in its database. The records are stored with indices for URLs and for PageRank. For the MapReduce configuration, the dataset is stored in the Hadoop distributed file system, which automatically distributes the data among the data

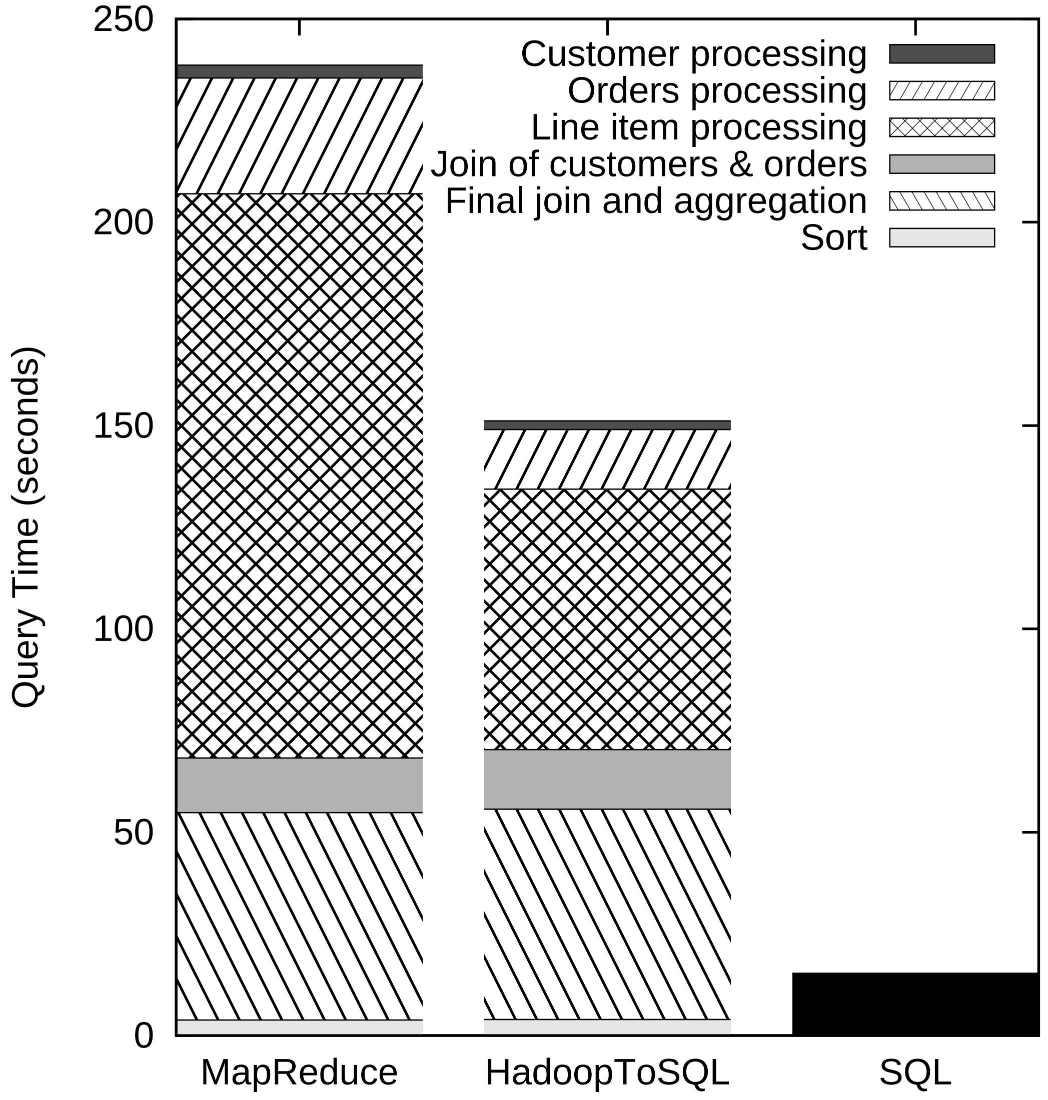


Figure 5.20: Query time for TPC-H query Q3 when the joins of the Customer, Order, and LineItem tables are performed by MapReduce.

nodes.

The experiment uses 10 data nodes running on Amazon's Elastic Compute Cloud (EC2). A "small " EC2 instance is used for each node, which are configured with a single virtual core, 1.7GB of RAM, and 160GB of local disk space. Two additional nodes are needed to run the Name Node and Job Tracker servers needed by Hadoop for tracking distributed file system metadata and coordinating MapReduce jobs. All the machines run Fedora 8, Hadoop 0.20, Java 1.6, and PostgreSQL 8.2.

The selection task can be completed by MapReduce in this way:

- During the map phase, each data node scans through ranking records, outputs those URLs that satisfy the query, and stores the results into the distributed file system.
- After the map phase has executed, the result of the query has been computed but is stored in multiple files distributed throughout the cluster.
- The reduce phase transfers these files to a single node, which combines them into a single sorted file.

Although HadoopToSQL can translate MapReduce programs into SQL queries, it currently does not contain code for running SQL queries on a cluster of SQL machines. As a result, for this experiment, HadoopToSQL is only able to use its transformations to find input set restrictions. To estimate the performance of this task on an SQL database, the experiment uses a small program that emulates the behavior of a distributed SQL database, due to the difficulty in gaining access to one. This program launches 10 threads that each queries one of the databases. The results are then transferred back to this program and stored to disk in no particular order.

Figure 5.21 shows the results of running the benchmark. Each data point is an average of three benchmark runs. For MapReduce and HadoopToSQL, two results are shown. The foreground bar shows the time needed to run the map phase of the MapReduce job only. The background bar includes the time needed to also run a reduce phase. Depending on how the user intends to use the data, they may or may not require the extra processing performed by the reduce phase.

In this experiment, HadoopToSQL is able to find an input set restriction successfully, resulting in better performance than MapReduce. Both HadoopToSQL and SQL are able to restrict their processing to only the 300,000 records of data per node that satisfied the query. HadoopToSQL's map phase is significantly faster than MapReduce's map phase, but the improvement is less when the reduce phase is included. This occurs because input set restrictions only help the map phase of a query and do not shorten the reduce phase. Although the total time of the HadoopToSQL query is longer than the estimated time for the SQL query, the map phase of the HadoopToSQL query takes less time than the SQL query. This occurs because the SQL program gathers all the query results on a single node, resulting in a potential communication and disk bottleneck on that one node. Although the results of the MapReduce and HadoopToSQL queries are known after the map phase, the query results are stored in multiple files spread out among the data nodes. These results are only merged together into a single file during the reduce phase. Because the

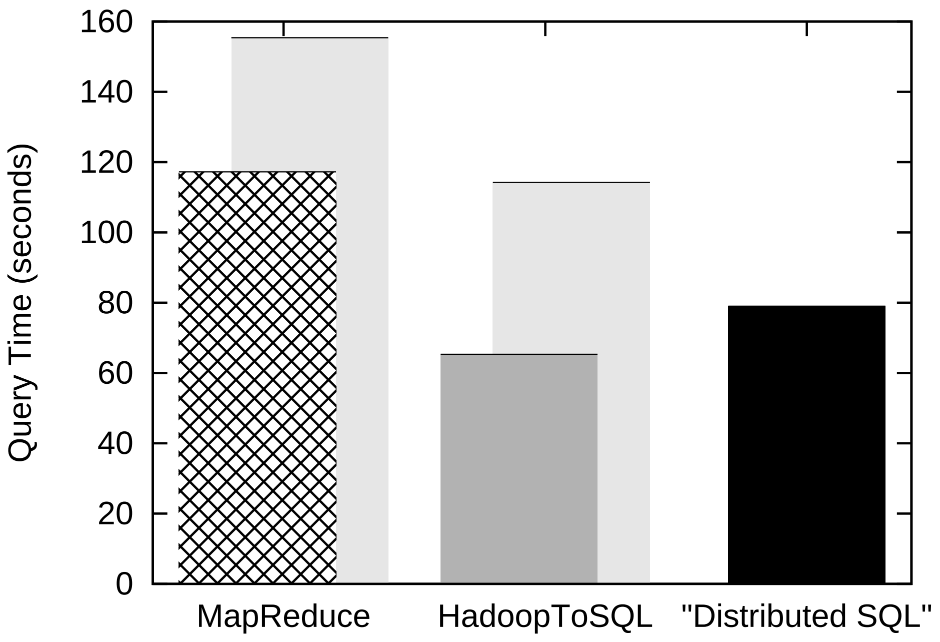


Figure 5.21: In this graph of execution time for the Selection Task, the results for MapReduce and HadoopToSQL are shown using two bars—the foreground bar shows the results of the map phase only, whereas the background bar includes the time of a reduce phase for gathering the results into single file.

reduce phase of a MapReduce program starts while the map phase is still running, it is not possible to determine the actual duration of a reduce phase from the graph. In fact, the reduce phase of the HadoopToSQL query has a shorter overlap with the map phase than the MapReduce query due to the shorter runtime of the HadoopToSQL query's map phase.

5.5 Extensions

Although HadoopToSQL is already very powerful, there are many ways to extend the work to increase its usefulness. In particular, the core static analysis algorithms can be made less restrictive, a traditional distributed database query optimizer can be added, and an advanced storage engine can be designed specifically for MapReduce.

HadoopToSQL's symbolic execution currently halts when it encounters loops or outside functions while searching for input set restrictions. Although exploring loops and outside functions can lead to an exponential explosion of paths, sometimes this explosion is manageable, so HadoopToSQL could undertake limited explorations of loops and outside functions. Loops and outside functions can also be separately analyzed in advance of path traversals. For example, a system can check if a function is free of side-effects by verifying that it neither modifies any non-local variables nor calls any other function with side-effects. Calls to these functions can then be used in HadoopToSQL's symbolic execution. The return value of the function may be ambiguous, but symbolic execution can handle such ambiguity. Alternately, other researchers have successfully used other approaches such as attribute grammars for finding input set restrictions [WIC08] .

HadoopToSQL also currently lacks the ability to optimize across multiple instances of MapReduce. Complex MapReduce programs sometimes consist of multiple stages or instances of MapReduce chained together. The static analysis of HadoopToSQL allows it understand the operations performed by individual instances of MapReduce but is not useful in analyzing the relationship between instances. To solve this problem, HadoopToSQL would first need to provide programmers a mechanism to describe the flow of data between different MapReduce instances. The system could then combine this information with its analysis of individual MapReduce stages to build a query plan describing the complete computation. Once a query plan is built, a traditional database query plan optimizer can be used to rearrange elements of the plan to produce a more optimal execution. HadoopDB [ABPA$^+$09] operates directly on MapReduce query plans generated from the Hive query language, and it demonstrates some of the possibilities of applying traditional database query optimization techniques to MapReduce.

Finally, additional performance gains can be achieved by building advanced storage engines specifically for use with MapReduce instead of relying on SQL databases. As noted in the experiments, traditional databases arrange their data to allow for random-access and updates instead of linear table scans. Therefore, on workloads that need to process their entire dataset, using these databases is slower than using files stored in a MapReduce distributed file system. A purpose-built storage engine for MapReduce could arrange its data in compressed flat files to allow for optimal linear table scans but also provide indices for random-access. An advanced storage engine purpose-built for MapReduce could also take advantage of the fact that intermediate MapReduce results are always saved on disk by reusing these intermediate results for other queries that calculate the same values or subsets of the same values.

5.6 Summary

HadoopToSQL allows MapReduce programmers to take advantage of database features in advanced storage engines without needing to use a separate database query language. It uses static analysis algorithms based on symbolic execution to understand MapReduce queries and optimize them to use database operations. On workloads that access only a subset of a dataset, the performance of MapReduce queries can be significantly improved through such optimizations.

The evaluation has shown that HadoopToSQL is indeed able to understand MapReduce queries and optimize them for an SQL storage engine. Because the resulting queries are able to take advantage of SQL facilities such as indices, the queries are able to execute much more efficiently using an SQL database than using traditional MapReduce files. In many cases, HadoopToSQL is able to generate SQL code from MapReduce programs whose performance approximates that of hand-written SQL.

HadoopToSQL currently has difficulty analyzing MapReduce programs with loops and unknown method calls, and it is also unable to analyze across multiple MapReduce instances. These limitations can be addressed by adding special analysis algorithms specifically for loops and function calls, and by incorporating a traditional distributed database query optimizer into HadoopToSQL.

Chapter 6

Conclusion

Many applications need to process and manage large datasets, and programmers prefer to use databases for working with this data. Unfortunately, performing database operations from within conventional programming languages is often difficult and error-prone. This thesis examines how to integrate support for database queries into the programming language Java using a bytecode rewriting approach. With this approach, queries are expressed using a syntax and style that conform with existing Java conventions. A bytecode rewriter translates this code into a form that can run efficiently on conventional databases. This bytecode rewriter exists as a separate component in the programmer toolchain, and can be maintained and evolved separately from the Java language and compiler. In this thesis, the practicality of using bytecode rewriting to support database queries in Java was explored through the design of three different query systems. Each system studied how a different style for expressing database queries can be supported in Java through bytecode rewriting.

Queryll

Current imperative object-oriented programming languages are increasingly being augmented with support for functional language features such as anonymous functions and closures. These features offer a rich syntax for expressing database operations. Programmers can write queries using list comprehensions where the contents of a dataset are passed into a function for processing. Although the resulting code is written in a functional style, the code is eventually compiled into an imperative form requiring specific algorithms for recognizing such code and reconstructing its meaning. Queryll demonstrates an algorithm that is suitable for such an environment. By taking advantage of the lack of loops or side-effects in functional-style code, the Queryll algorithm is simple and efficient.

JReq

The JReq system studies how database queries could be written using imperative object-oriented code. In such code, the standard convention for manipulating large datasets is to iterate over each record in a dataset. As a result, a consistent syntax for database operations should also involve iterating over records. Such a syntax was designed and developed to the point where it could support complex queries. JReq is able to translate code written in this way into database queries that can be efficiently run by databases. The translation algorithm involves decomposing code into nested loops, using symbolic execution to transform each loop into a canonical form that summarizes the preconditions and postconditions for each loop, and then matching this canonical form against templates of the query types supported by the database. Experiments show that queries written with JReq can achieve similar performance to hand-written SQL queries in standard database benchmarks.

HadoopToSQL

MapReduce is a widely used framework for allowing programmers to process large datasets stored in a computing cluster. Although the processing of large datasets can be significantly accelerated by making use of database features such as indices, MapReduce code rarely takes advantage of such functionality because of the difficulty of interfacing MapReduce and databases. HadoopToSQL shows how MapReduce code can be analyzed and automatically rewritten to take advantage of database features. MapReduce code is typically written in conventional imperative programming languages and may contain loops. As such, HadoopToSQL is able to take advantage of the general algorithm from the JReq system for decomposing and transforming code. Beyond needing to adapt the algorithm to support MapReduce syntax, the algorithm is also extended to handle code that it cannot fully understand. As a result, unlike the JReq system, HadoopToSQL not only translates entire pieces of code into equivalent database queries, but can accelerate code that is too complex to be translated into a database query by using optimizations such as input set restrictions. HadoopToSQL is able to significantly improve the performance of appropriate MapReduce queries.

Final Remarks

This thesis successfully demonstrates that bytecode rewriting is a practical approach for supporting database queries in Java. Database operations can be expressed entirely using syntax from conventional programming languages. This syntax can be analyzed by a separate bytecode rewriter tool, so that the language and compiler does not need to be burdened with domain-specific features. Despite the expressiveness and lack of structure in conventional programming languages, symbolic execution can be used to extract database operations from the code.

By examining how bytecode rewriting can support three different styles of queries, the generality, practicality, and usefulness of the bytecode rewriting approach have been shown. In the future, this approach will hopefully be taken into consideration when people integrate support for database queries into programming languages.

Appendix A

Visualizing SQL

SQL92 [Ame92] is a large specification, which makes it difficult to understand the scope and expressiveness of the language. In particular, it is difficult to compare the expressiveness of SQL with other query languages because it is difficult to find the main expressive structures in SQL.

To surmount this problem, visualization techniques are used to group related functionality and to expose the main structure of the SQL language. The visualization starts with the raw BNF grammar for SQL92 since the grammar provides an upper-bound of the expressiveness of the language. This grammar is transformed into a graph by treating each non-terminal symbol as a node. If a non-terminal symbol can be expanded into another non-terminal symbol, a directed edge is placed between the two corresponding nodes. Terminal symbols are ignored since they are not needed in determining the general structure of SQL.

The resulting graph is still too large for human comprehension, so redundant nodes need to be removed from it. This is primarily done by grouping related non-terminal symbols. The SQL92 specification is divided into a number of chapters, and each chapter includes a number of grammar rules along with detailed descriptions of when these rules apply. By machine-parsing a text file of the SQL92 specification, the non-terminal symbols in the grammar could be annotated with the chapter they appear in. In the resulting graph of the grammar, nodes representing non-terminal symbols appearing the same chapter are merged. Self-loops and multi-edges are removed from the merged nodes. As a result, nodes in the resulting graph represent chapters from the specification, and a directed edge between chapter nodes means that the corresponding chapter defines a non-terminal symbol that can be expanded to a non-terminal symbol that is defined in another chapter. Analysis of the resulting graph reveals that it is primarily DAG-like, though it contains two large strongly-connected components: one related to queries and another related to defining literal values.

This thesis is interested in the query language component of the SQL specification only. Since the chapter graph of the SQL specification is grouped by chapters, it then becomes easy to prune out those chapters related to schema definition, data manipulation, or integration with other programming languages. The directed edges allow one to verify that there are no unexpected dependencies on chapters being pruned out. Some chapters refer to features which are poorly supported or which do not substantially increase the expressiveness of SQL since they can be expressed using other SQL features. These features include collations, views, temporary tables, cursors, indicator variables, modules, and procedures. These chapters are also removed from the graph.

The resultant graph can be visualized using a graph visualization package such as graphviz. Although there are a manageable number of nodes in the graph, the relationships between the various nodes are too complex to be visually inspected. Part of the problem is that some nodes have high in-degree, meaning many parts of the grammar depend on them, because they act like "libraries." For example, a chapter of the specification is devoted to listing various terminal symbols used in the specification, and another chapter is focused on how to express number constants. These nodes are pruned out of the graph because they distort the shape of the graph while being trivially supported by other query languages. Another part of the problem is that some chapter nodes have high-interdependencies because the chapters referred to areas of the specification with similar and related functionality. These nodes can be merged together into a single node, thereby significantly reducing the number of edges.

Figure A.1 shows the nodes that remain after these graph transformations. The remaining chapters are essentially those that are reachable from the following two chapters: 20.2 <direct select statement: multiple rows> and 13.5 <select statement: single row>. Studying the chapters shown in this graph reveals that the key operations that SQL supports are selection, projection, join, aggregation, duplicate removal, nested queries, set operations, sorting, and limiting. The main feature of SQL that is not reflected in the graph is the NULL value and its associated three value logic.

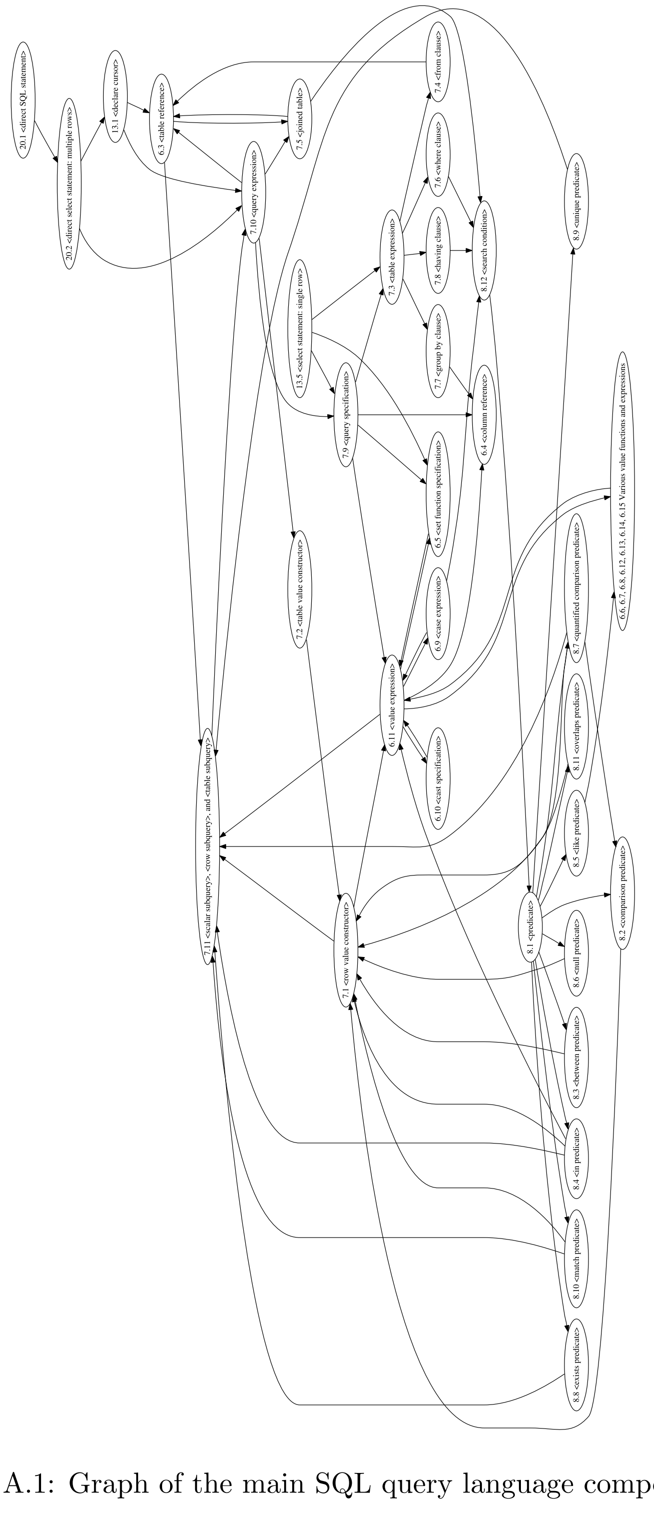

20.1 <direct SQL statement>
20.2 <direct select statement: multiple rows>
13.1 <declare cursor>
6.3 <table reference>
7.5 <joined table>
7.4 <from clause>
7.10 <query expression>
7.6 <where clause>
7.8 <having clause>
7.3 <table expression>
8.12 <search condition>
8.9 <unique predicate>
13.5 <select statement: single row>
7.7 <group by clause>
6.4 <column reference>
7.9 <query specification>
6.5 <set function specification>
6.6, 6.7, 6.8, 6.12, 6.13, 6.14, 6.15 Various value functions and expressions
8.7 <quantified comparison predicate>
7.2 <table value constructor>
6.9 <case expression>
6.11 <value expression>
8.11 <overlaps predicate>
6.10 <cast specification>
7.11 <scalar subquery>, <row subquery>, and <table subquery>
8.5 <like predicate>
8.2 <comparison predicate>
8.1 <predicate>
7.1 <row value constructor>
8.6 <null predicate>
8.3 <between predicate>
8.4 <in predicate>
8.10 <match predicate>
8.8 <exists predicate>

Figure A.1: Graph of the main SQL query language components

Bibliography

[ABPA$^+$09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[ACC$^+$02] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Alan Cox, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Bottleneck characterization of dynamic web site benchmarks. Technical Report TR02-389, Rice University, February 2002.

[Ame92] American National Standards Institute. *American National Standard for Information Systems—Database Language—SQL: ANSI INCITS 135-1992 (R1998)*. American National Standards Institute, 1992.

[Apa] Apache Software Foundation. Hadoop. *http://hadoop.apache.org/core/*.

[BG97] Aart J.C. Bik and Dennis B. Gannon. Javab—a prototype bytecode parallelization tool. Technical Report TR489, Indiana University, July 1997.

[BGGvdA] Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. Closures for the Java programming language (v0.5). `http://www.javac.info/closures-v05.html`. [accessed 2010-05-24].

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, 2002.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[CDG$^+$06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[CR05] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 97–106, 2005.

[CS] Stephen Colebourne and Stefan Schultz. First-class methods: Java-style closures. `http://docs.google.com/Doc?id=ddhp95vd_6hg3qhc`. [accessed 2010-05-24].

[CS08] Shimin Chen and Steven W. Schlosser. Map-Reduce meets wider varieties of applications. Technical Report IRP-TR-08-05, Pittsburgh, USA, 2008.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[DG10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[DGS88] D. J. DeWitt, S. Ghanderaizadeh, and D. Schneider. A performance analysis of the gamma database machine. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 350–360, New York, NY, USA, 1988. ACM.

[DK06] Linda DeMichiel and Michael Keith. JSR 220: Enterprise JavaBeans 3.0. `http://www.jcp.org/en/jsr/detail?id=220`, May 11 2006.

[EM98] Andrew Eisenberg and Jim Melton. SQLJ part 0, now known as SQL/OLB (object-language bindings). *SIGMOD Rec.*, 27(4):94–100, 1998.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 193–205, New York, NY, USA, 2001. ACM.

[GIS10] Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending Scala with database query capability. *Journal of Object Technology*, 9(4):45–68, July 2010.

[Goe10] Brian Goetz. Translation of lambda expressions in javac. `http://cr.openjdk.java.net/~mcimadamore/lambda_trans.pdf`, 2010. [accessed 2010-05-24].

[GS08] Ravindra Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proc. VLDB Endow.*, 1(1):1107–1123, 2008.

[GvdA] Neal Gafter and Peter von der Ahé. Closures for the Java programming language (v0.6a). `http://www.javac.info/closures-v06a.html`. [accessed 2010-05-24].

[IBY$^+$07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[ICZ10] Ming-Yee Iu, Emmanuel Cecchet, and Willy Zwaenepoel. JReq: Database queries in imperative languages. In *CC '10: Proceedings of the 19th International Conference on Compiler Construction*, Berlin, Heidelberg, 2010. Springer-Verlag.

[IZ06] Ming-Yee Iu and Willy Zwaenepoel. Queryll: Java database queries through bytecode rewriting. In Maarten van Steen and Michi Henning, editors, *Middleware*, volume 4290 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2006.

[IZ10] Ming-Yee Iu and Willy Zwaenepoel. HadoopToSQL: a MapReduce query optimizer. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 251–264, New York, NY, USA, 2010. ACM.

[JBo] JBoss. Hibernate. *http://www.hibernate.org/*.

[KJH$^+$08] Kiyoung Kim, Kyungho Jeon, Hyuck Han, Shin gyu Kim, Hyungsoo Jung, and Heon Y. Yeom. MRBench: A benchmark for MapReduce framework. *Parallel and Distributed Systems, International Conference on*, 0:11–18, 2008.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 - Proceedings European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KW82] R. H. Katz and E. Wong. Decompiling CODASYL DML into retional queries. *ACM Trans. Database Syst.*, 7(1):1–23, 1982.

[LD92] Daniel F. Lieuwen and David J. DeWitt. Optimizing loops in database programming languages. In *DBPL3: Proceedings of the third international workshop on Database programming languages : bulk types & persistent data*, pages 287–305, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[LLB] Bob Lee, Doug Lea, and Josh Bloch. Concise instance creation expressions: Closures without complexity. `http://docs.google.com/Doc.aspx?id=k73_1ggr36h`. [accessed 2010-05-24].

[MH02] Jerome Miecznikowski and Laurie Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *CC 2002*, pages 111–127. Springer-Verlag, 2002.

[MSOP86] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented DBMS. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 472–482, New York, NY, USA, 1986. ACM Press.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 83–94, New York, NY, USA, 2000. ACM.

[Ode06] Martin Odersky. The Scala experiment: can we provide better language support for component systems? In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 166–167, New York, NY, USA, 2006. ACM.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[PDGQ05] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[Per] Jurriaan Persyn. Database sharding at Netlog, with MySQL and PHP. *http://www.jurriaanpersyn.com/archives/2009/02/12/database-sharding-at-netlog-with-mysql-and-php/*.

[Pos] PostgreSQL Global Development Group. PostgreSQL. *http://www.postgresql.org/*.

[PPR+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.

[PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *REFLECTION '01*, volume 2192 of *LNCS*, pages 1–24, London, UK, 2001. Springer-Verlag.

[Rin99] Martin C. Rinard. Credible compilation. Technical Report MIT/LCS/TR-776, Cambridge, MA, USA, 1999.

[SAD+10] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[Spo] Spock Proxy. Spock proxy—a proxy for MySQL horizontal partitioning. `http://spockproxy.sourceforge.net/`.

[ST] ST Global. Spider storage engine. `http://spiderformysql.com/`.

[Suna] Sun Microsystems. Enterprise JavaBeans technology. `http://java.sun.com/products/ejb/`.

[Sunb] Sun Microsystems. JDBC technology. `http://java.sun.com/products/jdbc/`.

[SZ09] Daniel Spiewak and Tian Zhao. ScalaQL: Language-integrated database queries for Scala. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *SLE*, volume 5969 of *Lecture Notes in Computer Science*, pages 154–163. Springer, 2009.

[TF76] Robert W. Taylor and Randall L. Frank. CODASYL data-base management systems. *ACM Comput. Surv.*, 8(1):67–103, 1976.

[Tor06] Mads Torgersen. Language INtegrated Query: unified querying across data sources and programming languages. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 736–737, New York, NY, USA, 2006. ACM Press.

[Tra02] Transaction Processing Performance Council (TPC). *TPC Benchmark W (Web Commerce) Specification Version 1.8.* Transaction Processing Performance Council, 2002.

[Tra08] Transaction Processing Performance Council (TPC). *TPC Benchmark H (Decision Support) Standard Specification Version 2.8.0.* Transaction Processing Performance Council, 2008.

[TS04] Eli Tilevich and Yannis Smaragdakis. Portable and efficient distributed threads for Java. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 478–492, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a Map-Reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[WC07] Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 199–210, New York, NY, USA, 2007. ACM Press.

[WIC08] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 19–36, New York, NY, USA, 2008. ACM.

[Won00] Limsoon Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, 2000.

[WPN06] Darren Willis, David Pearce, and James Noble. Efficient object querying for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 1–14. USENIX Association, 2008.

Curriculum Vitae

Ming-Yee Iu was born in Ottawa, Canada in 1978. He graduated with a Bachelor of Mathematics with Honours in Computer Science from the University of Waterloo in 2000. He later completed a Master of Mathematics in Computer Science from the University of Waterloo in 2002. He joined EPFL in 2004 and started his PhD studies there in 2005 under the supervision of Professor Willy Zwaenepoel.