Foundations and Trends® in Programming Languages
Vol. XX, No. XX (2016) 1–95

DOI: 10.1561/XXXXXXXXXX

now
the essence of knowledge

Computer-Assisted Query Formulation

Alvin Cheung
University of Washington
akcheung@cs.washington.edu

Armando Solar-Lezama
MIT CSAIL
asolar@csail.mit.edu

Contents

Abstract

Database management systems (DBMS) typically provide an application programming interface for users to issue queries using query languages such as SQL. Many such languages were originally designed for business data processing applications. While these applications are still relevant, two other classes of applications have become important users of data management systems: (a) web applications that issue queries programmatically to the DBMS, and (b) data analytics involving complex queries that allow data scientists to better understand their datasets. Unfortunately, existing query languages provided by database management systems are often far from ideal for these application domains.

In this tutorial, we describe a set of technologies that assist users in specifying database queries for different application domains. The goal of such systems is to bridge the gap between current query interfaces provided by database management systems and the needs of different usage scenarios that are not well served by existing query languages. We discuss the different interaction modes that such systems provide and the algorithms used to infer user queries. In particular, we focus on a new class of systems built using program synthesis techniques, and furthermore discuss opportunities in combining synthesis and other methods used in prior systems to infer user queries.

now Publishers Inc.. *Computer-Assisted Query Formulation.* Foundations and Trends® in Programming Languages, vol. XX, no. XX, pp. 1–95, 2016.
DOI: 10.1561/XXXXXXXXXX.

1

Introduction

From financial transactions to online shopping, we interact with database management systems (DBMSs) on a daily basis. Since the initial development of relational database systems, various query languages such as SQL have been developed for users to interact with the DBMS. Many of these languages proved very effective for what was originally their primary application: business data processing (*e.g.*, generating transaction reports at a financial institution). Unfortunately, many important applications of DBMSs that have emerged in recent decades have proven to be a less than ideal fit for the interaction models supported by traditional DBMSs.

One particularly important extension to the business data processing application space corresponds to applications with complex business logic, such as social network websites, online shopping applications, *etc.* Unfortunately, traditional query interfaces often make developing such applications difficult. First, the general-purpose languages in which these applications are usually written (*e.g.*, Java or Python) are quite different from the query languages supported by the DBMS. This forces developers to learn a new language—and often a new programming paradigm altogether. For example, an application developer who

is used to thinking about computation over objects stored in the program heap will need to recast her computation in terms of structured relations stored on disks when interacting with a DBMS. Moreover, in addition to being concerned about efficient memory layout for retrieving in-memory objects, she will also need to understand the costs associated with bringing objects into memory from the disk. This "impedance mismatch" [Copeland and Maier, 1984] problem has plagued application developers for decades. Today, this mismatch is often addressed by application frameworks known as Object Relational Mapping (ORM) Frameworks that eliminate the need to think in terms of two distinct programming models. Unfortunately, the use of ORMs often imposes significant performance costs [Subramanian].

There are many reasons for the performance cost of ORMs, but one that is especially significant is that they encourage a programming style where computation that could have been implemented with a single query and a single round trip to the database is instead implemented with several simpler queries connected together with imperative code that manipulates their results. This is problematic because in addition to increasing the number of round trips and the amount of data that needs to be transferred between the application and the DBMS, doing so also increases the cost of the computation, since the DBMS is in much better position to optimize queries compared to a general-purpose code compiler trying to optimize a block of imperative code that happens to implement a relational operation.

As an example, while a relational join between relations R and S can be implemented using a nested loop, with each loop processing tuples from the two respective relations fetched from the DBMS, it is much more efficient to implement the join as a single SQL query, as the DBMS can choose the best way to implement the join during query optimization.

In this tutorial we focus on a new approach based on verified lifting [Cheung et al., 2015] to reduce the performance cost of these application frameworks, allowing programmers to enjoy the benefits of the reduced impedance mismatch. The first step in this technique is to identify places in the application code where the programmer is using

imperative code to implement functionality that could be implemented as part of a query. The second and most important step is to use program synthesis technology to derive a query that is provably equivalent to the imperative code. Once that is done, the third step involves generating a new version of the code that uses the query in place of the original code.

The technology behind this work was originally published earlier [Cheung et al., 2013]. In this paper, we expand on the content of that original paper in order to make the technology more accessible to researchers without a strong background in program synthesis or verification, as well as to researchers who may not be as familiar with database concepts. In Section 2, we provide a quick primer on query execution and query processing, focusing on key concepts that will help the reader understand the reasons for the performance problems introduced by ORMs. Section 3 provides a comprehensive primer on program synthesis technology, focusing in particular on the techniques that are leveraged by QBS, and putting them in context of other synthesis technologies. Section 4 describes the details of the QBS approach, and finally Section 5 describes the state of the art in terms of applications of synthesis to interact with DBMS systems and promising directions for future work.

2

Query Processing

This section provides a high-level introduction to the different ways applications can interact with the DBMS. The section also provides some background on how the DBMS processes queries; this background will help the reader understand why issuing queries in different ways can have significant performance impacts for an application.

2.1 Relational DBMS and Query Languages

Since the development of relational database systems in the 1970s, SQL (Structure Query Language) has become the most popular query language for interacting with DBMS. SQL is based on the relational model, which models data as relation instances. A relation instance is similar to a spreadsheet table with rows and columns, except that columns are well-typed. Each column is a named and typed field, and the set of fields for each relation is known as the schema of that relation. An instance of a relation is a set of records (also called tuples), where all records share the same schema. SQL is an implementation of relational algebra [Codd, 1970, Date, 2000], except that it models relations as bags (*i.e.*, multisets) rather than sets, as in the original relational algebra

formulation. The language includes a subset of relational operators such as projections, selections, joins, and aggregations, but not others such as transitive closure. In addition to these operators, inLogical Plan ge practice most DBMSs allow users to execute arbitrary code by defining user defined functions (UDFs). To use UDFs, developers first implement the UDF using a domain-specific language such as PL/SQL (which is an implementation of the SQL/PSM standard [International Organization for Standardization, 2011]) and compile it using a custom compiler provided by the DBMS. The compiled binary is then linked to the DBMS kernel and the function is then available to the query executor, to be described in Section 2.4.

2.2 DBMS as a library

Since the initial development, relational DBMSs have been designed to be stand-alone systems rather than application libraries to be linked with the application during compilation. Early DBMS implementations did not support complex applications and only provided a command-line interface for end-users to interact with the system by typing queries on the console, with the DBMS returning results and displaying them to the user on the screen. As business data processing applications became popular, DBMS implementations started to provide language level abstractions (such as JDBC [JDBC 4.2 Expert Group, 2014] and ODBC [International Organization for Standardization, 2008]) for applications to interact with DBMSs programmatically by issuing SQL queries. Such abstractions are often implemented as connector libraries provided by the DBMS, and are linked by developers to their application binaries at compile time. These libraries allow application developers to embed query statements within their application source code as if they were using the command-line interface, and are furthermore completely separated from the DBMS implementation itself in order to support functionality such as issuing queries remotely via network. In these situations, the embedded query statements are sent to the connector libraries as the application executes, which in turn are forwarded to the DBMS for execution. The results are then sent back to

the libraries, and the libraries would return them to the application after serializing the results into data structures such as lists or arrays of primitive types.

On the one hand, connector libraries greatly ease DBMS development as they cleanly abstract away the application; the DBMS can process queries as if they were issued by end-users through the command-line interface. Unfortunately, this comes at a cost for the application developer. First, as applications are usually written using a general-purpose language, developers need to learn a new language in order to express their persistent data needs. Worse yet, embedding query statements as raw strings in the application makes debugging difficult; the raw strings are not parsed or type-checked by the application compiler, so errors in the queries only become apparent at execution time. Not only that, embedding raw query strings in application code is often the source of various security vulnerabilities such as SQL injection.

2.3 The ORM approach

In recent years, new frameworks and libraries have been developed to provide better integration between the application and the DBMS. Such frameworks can be separated into two categories. The first category includes query language integrated libraries [Microsoft, b, Squeryl, jOOQ] that provide stylized library calls for relational operations. While developers still need to understand query concepts (such as selections and joins), they no longer need to have knowledge about query language syntax, and using such libraries does not incur the same security issues as embedding raw strings in program code. In another category are object-relational mapping (ORM) frameworks [JBoss, Cooper et al., 2007, Microsoft, a, Django]. These frameworks go one step beyond language integrated libraries by giving developers the ability to interact with the DBMS using the language of the application logic. With ORMs, developers label certain classes as persistent—usually through annotations or configuration files—and the framework automatically manages all persistently stored objects for the application: when the application needs to retrieve objects that are stored in the

DBMS, it simply passes the requested object identifiers to the ORM framework, and the framework either returns cached objects or translates the request into SQL queries executed by the DBMS. Writes to persistent objects are handled similarly; the user never has to write queries, because the framework generates all queries automatically. [1]

Unfortunately, applications written using ORM frameworks are often not efficient for several reasons. First, they lack high-level information about the application, and as such they often issue unnecessary queries that slow down the application [Cheung et al., 2014]. Most importantly from the point of view of this work, applications written with ORMs often take what could have been a single query and break it down into simpler queries whose results are then processed in application code. At first, it may seem that the only effect of this would be that some computation that used to take place in the database server now takes place in the application, but as the following section will explain, the DBMS contains a sophisticated optimization engine that can dramatically improve the performance of a query. By moving some of the query logic to the application, the benefits of this optimization capability are lost.

2.4 Query Execution

The goal of query execution is to compile the SQL query into an executable, called the physical execution plan, that retrieves the requested data from the base relations. Figure 2.1 illustrates the steps used by a typical DBMS to process an incoming query; in this section we review each of the steps in detail.

Logical Plan Generation. Upon receiving the SQL query, the parser first converts it into a logical query plan after performing a number of validation steps (*e.g.*, syntax and type checks, verifying that the referred tables exist, *etc*). The logical plan serves as an intermediate representation of the SQL query and is typically represented as a tree with the output operator at the root and table scan operators (which retrieve

[1] Similar capabilities have been explored in distributed object research, for instance CORBA [Object Management Group, 2012].

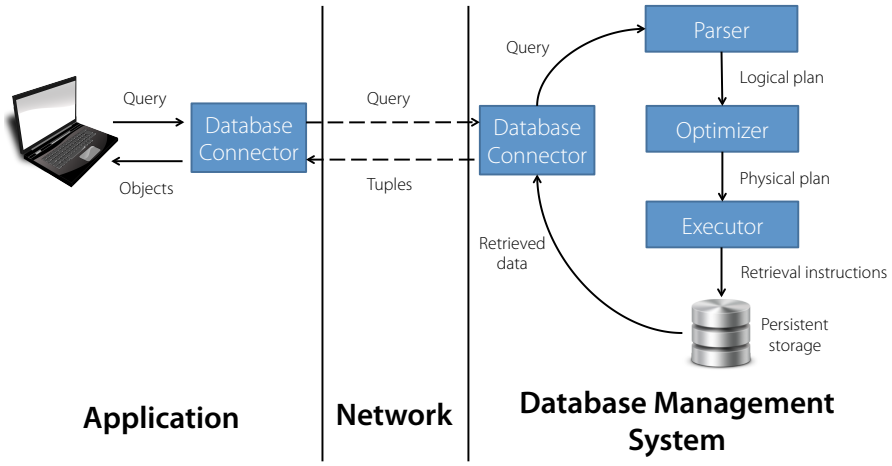


Figure 2.1: Query processing in a relational DBMS

all tuples from a given relation) at the leaves. Intermediate nodes in the tree represent either relational algebra operators or UDF invocations, and all edges represent dataflow relationships. Query evaluation starts at the bottom, with each edge in the graph representing the intermediate query results that are forwarded from one operator to the next.

Plan Optimization. The generated logical plan is then passed to the query optimizer. The goal of query optimization is to choose an implementation for each operator in the logical plan. This is done by performing a number of rewrites that include constant propagation and evaluation of redundant Boolean formulas. In addition, the query might also be rewritten using a number of relational algebraic properties. For example, inlining view definitions [Pirahesh et al., 1992] (views are pre-defined result sets of queries that are similar to let-bindings) or combining predicates from two adjacent selection operators as a conjunction. In addition, the plan generator also performs a number of heuristics-based optimizations. Two examples are "pushing down" selection operators into table scan operators—as doing so will decrease the number of tuples that need to be fetched from disks—and flattening nested queries to enable further optimization on the flattened expressions [Pirahesh et al., 1992, Seshadri et al., 1996].

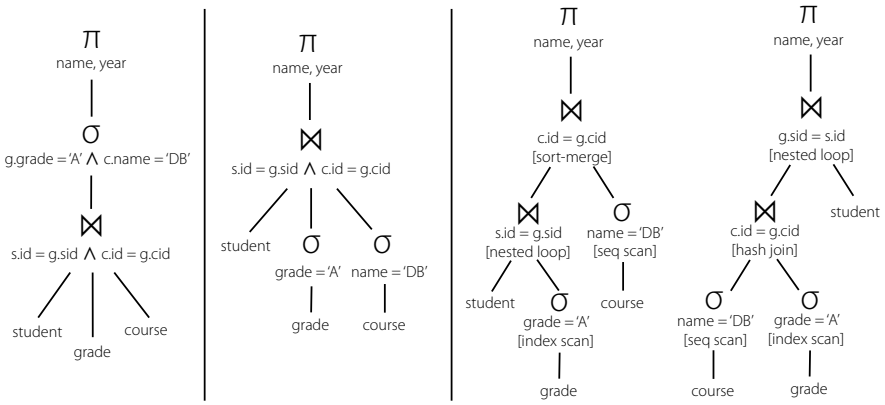


Figure 2.2: Query plans for the query that returns students who received 'A' in the database class: its logical query plan (left), logical plan after selection pushdown (middle), and two potential physical execution plans with chosen implementations for each query operator (right). Here π represents relational projection, $\bowtie$ represents a relational join while σ represents a relational selection.

As an illustration, consider three relations—`student`, `grade` and `course`—that store information about students, course grades and course descriptions respectively. The query:

```
SELECT s.name, s.year
FROM   student s, grade g, course c
WHERE  s.id = g.sid AND c.id = g.cid AND
       g.grade = `A' AND c.name = `DB'
```

retrieves student information for those who received 'A' in the database class. Figure 2.2 shows the logical query plan generated from this query along with potential physical query plans generated by the query optimizer.

Cost Estimation. The query optimizer relies on cost estimates of a plan's cost in order to select the most efficient query plan. In most query optimizers, the cost is usually computed as a function of each of the following quantities, some of which depend on the query being issued, and some are DBMS implementation-specific:

- The number of disk reads. Since accessing the disk can take multiple orders of magnitude more time when compared to accessing

main memory, reducing the number of disk reads (by pushing down selection operators, for instance) can greatly reduce the plan cost.

- The size of intermediate results. The intermediate results that are passed between each operator in the logical plan (represented by the edges in the graph) need to be stored in memory during query evaluation. As such, reducing the sizes of such intermediates can improve the execution time of the query.

- The data structure used. The way that the persistent data is stored on disk can greatly affect the query execution cost. For instance, storing data using row-major format is ideal for queries that project all fields from a relation (*e.g.*, `SELECT * FROM table`). Meanwhile, storing data in a column-major format [Stonebraker et al., 2005] makes evaluating aggregates efficient (*e.g.*, summing all values of a particular field) as the executor can avoid random disk seeks during query evaluation. The existence of auxiliary data structures, such as indices, can also affect the query evaluation cost as well.

As an example, Figure 2.2(right) shows two possible physical execution plans for the query shown in Figure 2.2(left). Choosing which plan to execute depends on the factors discussed above along with those listed in the next section.

Implications for ORMs. Query optimization is by no means a solved problem [Lohman, 2014]; for many queries, for example, writing the same query in slightly different ways can have a significant impact in the performance of a query. Nevertheless, thanks to verified lifting, the performance difference between a query implemented as Java or Python code and one executed inside the DBMS can be dramatic. For queries involving joins, the database is often able to use $\mathcal{O}(n \log n)$ or even constant time algorithms, significantly outperforming a naive join implemented with nested loops in the application code. For this reason, identifying those places where the application code is implementing queries and transforming that code to issue queries to the database di-

rectly can have dramatic performance improvements, as our evaluation section will corroborate.

3

Program Synthesis

Program synthesis is an emerging technology that could help address some of the challenges with existing query languages and interfaces that we outlined in the previous section. In this section we describe the basic concepts in program synthesis. We first discuss the synthesis problem, then outline two different approaches to solving the problem. The first approach is deductive, where an implementation is derived from a specification through the iterative application of deductive rules. The second approach is inductive, where the goal is to generalize from a given set of examples; counterexample-guided inductive synthesis can be used in cases where the examples are not provided a priori, but must be derived by the system from a higher level specification. Both techniques are used in several query inference systems, to be discussed in Section 5.

3.1 The Problem

There are a number of different approaches to formalizing the synthesis problem, but in general, the goal is to derive a program p drawn from some space of possible programs P that satisfies a set of semantic con-

straints. In the *syntax guided synthesis* formalism [Alur et al., 2013], for example, the space of functions is a grammar describing a language L defined on top of an underlying logical theory T, so that all the formulas in L are legal formulas in that theory. In this formalism, the specification is a formula ϕ in the theory T that involves the unknown function p_{un}. The goal is to find p such that

$$\forall i \in I \;.\; \phi(p, i)$$

is valid, where $\phi(p, i)$ is the formula obtained by substituting the concrete p for the unknown p_{un} and i for the free variables. Note that the size of the set I from which values for the free variables are drawn from can be infinite in general.

One class of problems that this formalism of syntax guided synthesis is a poor fit involves *reactive programs* which run in an infinite loop receiving inputs from the environment and produce outputs in response in addition to maintaining some internal state. For these programs, the most natural specifications usually describe properties of the infinite sequence of inputs and outputs to the program, and are most naturally expressed in some *temporal logic*. For the purposes of this tutorial, however, we will be focusing on the standard syntax guided synthesis formulation, since this is a good fit for learning database queries.

Obviously, one possible approach to solve the problem is to simply exhaustively iterate through each possible program p, and check against ϕ (*e.g.*, using a theorem prover). Such approach is only applicable to cases where p is highly constrained. Otherwise, the large number of possible candidates for p makes the approach infeasible.

3.2 Deductive Synthesis

Deductive theorem proving is one of the earliest techniques [Manna and Waldinger, 1992] used to combat the exhaustive search problem in synthesis. The process works by deriving a constructive proof that the program specification is satisfiable. In deductive systems, the synthesizer designer comes with a set of predefined (r, e) pairs, where r is derivation (as part of a deductive proof), and e is an expression in L that represents the result of applying r to ϕ. The combination of (r, e)

pairs forms a *tableau* and it records the deductive steps that were taken to synthesize p from ϕ. Starting from the original specification ϕ, the system chooses one of the rules r from the set to transform ϕ into a logically equivalent specification ϕ'. Each time when a rule is applied the corresponding e is recorded as part of the synthesized program in the tableau. The process continues until ϕ' reduces to **True**, which means that the synthesizer has found p. If ϕ' reduces to **False** instead, then the synthesizer might choose to backtrack and apply another rule. And if all applications have been explored, then the system will declare that no such p exists within the language L. There have been a number of synthesizers built using this deductive approach, for instance for synthesizing various types of algorithms [Traugott, 1989, Manna and Waldinger, 1981] and implementation of systems from specifications [Qian, 1993, Burstein et al., 2000, Bickford et al., 2001].

3.3 Inductive Synthesis

Deductive synthesis techniques are best for domains where the user has a clear idea of the program that she would like to be synthesized, a clean theory is available to write the specification, and there is a relatively small number of rules that are applicable during each iteration of the synthesis procedure to reduce ϕ to **True**. The technique is less applicable when specifications are only partial or insufficiently declarative (*e.g.*, the user only provides sample program inputs and their corresponding outputs), or there are a large number of rules that are applicable to reduce the original specification.

In inductive synthesis, the idea is to leverage efficient search mechanisms to find a program that satisfies a specification for a small given set of inputs. If the synthesis problem is not already an inductive synthesis problem, it can be reduced to a series of inductive synthesis problems through the use of *counterexample guided inductive synthesis* (CEGIS) [Solar-Lezama et al., 2006, Alur et al., 2013]. The idea of CEGIS is to find candidate programs that work for an increasing subset of inputs from I during each step.

In applying CEGIS, the first step is for the synthesizer chooses a candidate program p_1 from P, and consults an oracle to test its correctness—the oracle could be anything from a random tester to a full fledged verifier. If p_1 satisfies the functional specification provided by the user, then we have found the solution. Otherwise, the oracle will return a counter-example input c_1 that falsifies p_1. This input c_1 is added to the original list of examples that the candidate program needs to satisfy. The next iteration starts with the synthesizer coming up with another candidate program p_2 that satisfies all counter-examples found thus far (namely c_1). p_2 is again sent to the oracle, which returns with another counter-example c_2. The process repeats until the oracle fails to find further counter-examples, meaning that the synthesized program satisfies the functional specification provided by the user. CEGIS is closely related to similar concepts in model checking [Clarke et al., 2000]. In the worst case, CEGIS will reduce to checking all possible program inputs in the case where no such program from P exists that satisfies the functional specification.

To cut down the time to synthesize and verify candidate programs, in practice most systems bound the space of programs that will be searched by the synthesizer and subsequently checked by the oracle. For instance, the system might limit the synthesized programs to be loop-free [Jha et al., 2010, Gulwani et al., 2011] and bound the length of the synthesized program [Phothilimthana et al., 2014, Perelman et al., 2014]. Other types of constraints include limiting the size of integral types (and hence reducing the amount of time needed for verification), or simply by putting a time-out for the synthesis process to complete.

There are currently three major approaches to conduct the search required by inductive synthesizer: constraint-based, stochastic and explicit. We explain each of them below.

Constraint-based. These systems encode the space of possible functions as a parameterized function $p[c]$ that behaves differently depending on the values of the *control parameters*. The functional specification is then translated into a set of constraints on c. Inductive synthesis is framed as finding a control parameter that satisfies all the constraints. During inductive synthesis, each counter-example found is encoded as

further constraints to the system. The search process completes when the oracle cannot find further counter-examples (*i.e.*, we have found the solution), or when the search fails to find a candidate p that satisfies all the given constraints (*i.e.*, no such p exists). This technique has been used in synthesizing block ciphers [Solar-Lezama et al., 2006], homework graders [Singh et al., 2013], and program deobfuscation [Jha et al., 2010].

Stochastic search. Rather than constraints, another means to perform the search for candidate programs by using stochastic search. Such systems aim to find candidate programs using different stochastic search algorithms, such as random or Monte Carlo sampling. Such techniques rely on a cost function to guide the sampling (and thus search) procedure, and the found counter-examples are used to refine the space where sampling takes place. Compared to constraint-based algorithms, stochastic search technique might not be complete (*i.e.*, it might miss a candidate program even though one exists within the search space). However, the sampling procedure can be easily parallelized across multiple machines. This technique has been used in bit-manipulation programs [Schkufza et al., 2013] and floating-point programs [Schkufza et al., 2014].

Explicit enumeration with symbolic representation. Explicit search is another technique for searching for candidate programs. However, given the large space of possible programs, it is infeasible to represent all candidates explicitly and prune them until a solution is found. In practice, such systems represent the candidate programs symbolically. For instance, the space of all possible programs can be represented succinctly using a directed acyclic graph [Gulwani et al., 2012]. By removing edges from different nodes in the graph, the synthesizer effectively eliminates candidate programs from the search space. This approach is particularly effective when the correctness criteria can be factored such that different parts of the program can be synthesized independently. For instance, to learn a text-editing program (*e.g.*, capitalizing or abbreviating letters) that processes (first name, last name) pairs, the synthesizer can first synthesize a program fragment that pro-

cesses first names, and then learn another fragment for the last name, before combining them together to form the final output. As such, the two synthesis steps can proceed independently, and the synthesizer can immediately eliminates programs that fail in processing first names correctly regardless of the program fragment for processing last names.

As another example, Transit [Udupa et al., 2013] enumerates candidate programs of increasing size during each step in the synthesis process. The system uses the counter-examples that are found to prune away equivalent candidate programs, where two candidate programs are deemed as semantically equivalent if they behave the same on the set of counter-examples. Larger candidate programs are built from smaller candidate programs, so pruning has an exponential benefit since eliminating a single small program also eliminates all the larger programs that would have been constructed from it. Once a candidate is found that satisfies the specification, it is returned, possibly to be checked as part of CEGIS.

In the following sections we discuss how inductive synthesis has been applied in assisting users formulating database queries. In particular, we focus on different techniques that are used to reduce the number of iterations required for synthesis using CEGIS. These kinds of explicit search techniques are particularly effective when the search space has a lot of *symmetries*—*i.e.*, large number of classes of equivalent programs—because equivalent programs are pruned away early in the search process.

4

Using Verified Lifting to Rewrite Code into SQL

Verified lifting [Cheung et al., 2015, Kamil et al., 2016] is a technique that takes as input a block of potentially optimized code written in an imperative general-purpose language, and infers a summary of it expressed in a high-level predicate language that is provably equivalent to the semantics of the original program. The lifted summaries expressed using the predicate language are found automatically using inductive program synthesis. Once found, such summaries can be translated to different high-performance DSLs, and subsequently retargeted to execute on different architectures as needed.

In this section we discuss QBS, a system that uses verified lifting to automatically transforms fragments of application logic into SQL queries. QBS differs from traditional compiler optimizations as it relies on synthesis technology to generate invariants and postconditions for a code fragment. The postconditions and invariants are expressed using a new theory of ordered relations that allows us to reason precisely about both the contents and order of the records produced complex code fragments that compute joins and aggregates. The theory is close in expressiveness to SQL, so the synthesized postconditions can be readily translated to SQL queries.

In this section we discuss the enabling technologies behind QBS along with our initial prototype implementation. Using 75 code fragments automatically extracted from over 120k lines of open-source code written using the Java Hibernate ORM, our prototype can convert a variety of imperative constructs into relational specifications and significantly improve application performance asymptotically by orders of magnitude.[1]

4.1 Interacting with the DBMS

QBS (Query By Synthesis) is a new code analysis algorithm designed to make database-backed applications more efficient. Specifically, QBS identifies places where application logic can be converted into SQL queries issued by the application, and automatically transforms the code to do this. By doing so, QBS move functionalities that are implemented in the server component that is hosted on the application server to the query component to be executed in the DBMS. This reduces the amount of data sent from the database to the application, and it also allows the database query optimizer to choose more efficient implementations of some operations—for instance, using indices to evaluate predicates or selecting efficient join algorithms.

One specific target of QBS is programs that interact with the database through ORM libraries such as Hibernate for Java. Our optimizations are particularly important for such programs because ORM layers often lead programmers to write code that iterates over collections of database records, performing operations like filters and joins that could be better done inside of the database. Such ORM layers are becoming increasingly popular; for example, as of August, 2015, on the job board `dice.com` 13% of the 17,000 Java developer jobs are for programmers with Hibernate experience.

We are not the first researchers to address this problem; Wiedermann et al. [Wiedermann and Cook, 2007, Wiedermann et al., 2008] identified this as the *query extraction problem*. However, our work is

[1] Materials in this chapter are based on work published as Cheung, Solar-Lezama, and Madden, "Optimizing Database-Backed Applications with Query Synthesis," in proceedings of PLDI 13 [Cheung et al., 2013].

able to analyze a significantly larger class of source programs and generate a more expressive set of SQL queries than this prior work. Specifically, to the best of our knowledge, our work is the first that is able to identify joins and aggregates in general purpose application logic and convert them to SQL queries. Our analysis ensures that the generated queries are *precise* in that both the contents and the order of records in the generated queries are the same as those produced by the original code.

At a more foundational level, this work is the first to demonstrate the use of constraint-based synthesis technology to attack a challenging compiler optimization problem. Our approach builds on the observation by Iu et al. [Iu et al., 2010] that if we can express the postcondition for an imperative code block in relational algebra, then we can translate that code block into SQL. Our approach uses constraint-based synthesis to automatically derive loop invariants and postconditions, and then uses an SMT solver to check the resulting verification conditions. In order to make synthesis and verification tractable, we define a new *theory of ordered relations* (TOR) that is close in expressiveness to SQL, while being expressive enough to concisely describe the loop invariants necessary to verify the codes of interest. The postconditions expressed in TOR can be readily translated to SQL, allowing them to be optimized by the database query planner and leading in some cases to orders of magnitude performance improvements.

At a high level, QBS makes the following contributions:

- We demonstrate a new approach to compiler optimization based on constraint-based synthesis of loop invariants and apply it to the problem of transforming low-level loop nests into high-level SQL queries.
- We define a theory of ordered relations that allows us to concisely represent loop invariants and postconditions for code fragments that implement SQL queries, and to efficiently translate those postconditions into SQL.
- We define a program analysis algorithm that identifies candidate code blocks that can potentially be transformed by QBS.

```
List<User> getRoleUser () {
  List<User> listUsers = new ArrayList<User>();
  List<User> users = this.userDao.getUsers();
  List<Role> roles = this.roleDao.getRoles();
  for (User u : users) {
    for (Roles r : roles) {
      if (u.roleId().equals(r.roleId())) {
        User userok = u;
        listUsers.add(userok);
      }
    }
  }
  return listUsers;
}
```

Figure 4.1: Sample code that implements join operation in application code, abridged from actual source for clarity

- We demonstrate our full implementation of QBS and the candidate identification analysis for Java programs by automatically identifying and transforming 75 code fragments in two large open source projects. These transformations result in order-of-magnitude performance improvements. Although those projects use ORM libraries to retrieve persistent data, our analysis is not specific to ORM libraries and is applicable to programs with embedded SQL queries.

4.2 QBS Overview

This section gives an overview of our compilation infrastructure and the QBS algorithm to translate imperative code fragments to SQL. We use as a running example a block of code extracted from an open source project management application [Wilos Orchestration Software] written using the Hibernate framework. The original code was distributed across several methods which our system automatically collapsed into a single continuous block of code as shown in Figure 4.1. The code retrieves the list of users from the database and produces a list containing a subset of users with matching roles.

```
List listUsers := [ ];
int i, j = 0;
List users := Query(SELECT * FROM users);
List roles = Query(SELECT * FROM roles);
while (i < users.size()) {
  while (j < roles.size()) {
    if (users[i].roleId = roles[j].roleId)
      listUsers := append(listUsers, users[i]);
    ++j;
  }
  ++i;
}
```

Figure 4.2: Sample code expressed in kernel language

Postcondition

$listUsers = \pi_{\ell}(\bowtie_{\varphi}(users, roles))$
where
$\varphi(e_{users}, e_{roles}) := e_{users}.roleId = e_{roles}.roleId$
ℓ contains all the fields from the *User* class

Translated code

```
List<User> getRoleUser () {
  List<User> listUsers = db.executeQuery(
       "SELECT u
        FROM users u, roles r
        WHERE u.roleId == r.roleId
        ORDER BY u.roleId, r.roleId");

  return listUsers;
}
```

Figure 4.3: Postcondition as inferred from Figure 4.1 and code after query transformation

The example implements the desired functionality but performs poorly. Semantically, the code performs a relational join and projection. Unfortunately, due to the lack of global program information, the ORM library can only fetch all the users and roles from the database and perform the join in application code, without utilizing indices or efficient join algorithms the database system has access to. QBS fixes this problem by compiling the sample code to that shown at the bottom of Figure 4.3. The nested loop is converted to an SQL query that implements the same functionality in the database where it can be executed more efficiently, and the results from the query are assigned to `listUsers` as in the original code. Note that the query imposes an order on the retrieved records; this is because in general, nested loops can constraint the ordering of the output records in ways that need to be captured by the query.

One way to convert the code fragment into SQL is to implement a syntax-driven compiler that identifies specific imperative constructs (e.g., certain loop idioms) and converts them into SQL based on pre-designed rules. Unfortunately, capturing all such constructs is extremely difficult and does not scale well to handle different variety of input programs. Instead, QBS solves the problem in three steps. First, it synthesizes a postcondition for a given code fragment, then it computes the necessary verification conditions to establish the validity of the synthesized postcondition, and it finally converts the validated postcondition into SQL. A postcondition is a Boolean predicate on the program state that is true after a piece of code is executed, and verification conditions are Boolean predicates that guarantees the correctness of a piece of code with respect to a given postcondition.

As an example, given a program statement `s`: `x = y;`, if we want to show that the postcondition `x = 10` holds after `s` is executed, then one way to do so is to check whether the predicate `y = 10` is true before executing the statement. In other words, if it is true that `y = 10` before executing `s`, then we can prove that the postcondition holds, assuming we know the semantics of `s`. In this case `y = 10` is the verification condition with respect to the postcondition `x = 10`. Note that there can be many postconditions for the same piece of code (e.g., `True` is

an obvious but uninteresting one), and the corresponding verification conditions are different.

While a typical problem in program verification is to find the logically strongest postcondition[2] for a piece of code, in QBS our goal is to find a postcondition of the form `v = Q()`, where `Q` is a SQL query to be executed by the database, along with the verification conditions that establish the validity of the postcondition. Limiting the form of the postcondition greatly simplifies the search, and doing so also guarantees that the synthesized postcondition can be converted into SQL. Compared to syntax-driven approaches, solving the code conversion problem in this way allows us to handle a wide variety of code idioms. We are unaware of any prior work that uses both program synthesis and program verification to solve this problem, and we discuss the steps involved in the sections below.

4.2.1 Qbs Architecture

We now discuss the architecture of QBS and describe the steps in inferring SQL queries from imperative code. The architecture of QBS is shown in Figure 4.4.

Identify code fragments to transform. Given a database application written in Java, QBS first finds the persistent data methods in the application, which are those that fetch persistent data via ORM library calls. It also locates all entry points to the application such as servlet handlers. From each persistent data method that is reachable from the entry points, the system inlines a neighborhood of calls, i.e., a few of the parent methods that called the persistent data method and a few of the methods called by them. If there is ambiguity as to the target of a call, all potential targets are considered up to a budget. A series of analyses is then performed on each inlined method body to identify a continuous code fragment that can be potentially transformed to SQL; ruling out, for example, code fragments with side effects. For each candidate code fragment, our system automatically detects the

[2] p is the strongest postcondition if there does not exist another postcondition p' such that $p' \rightarrow p$.

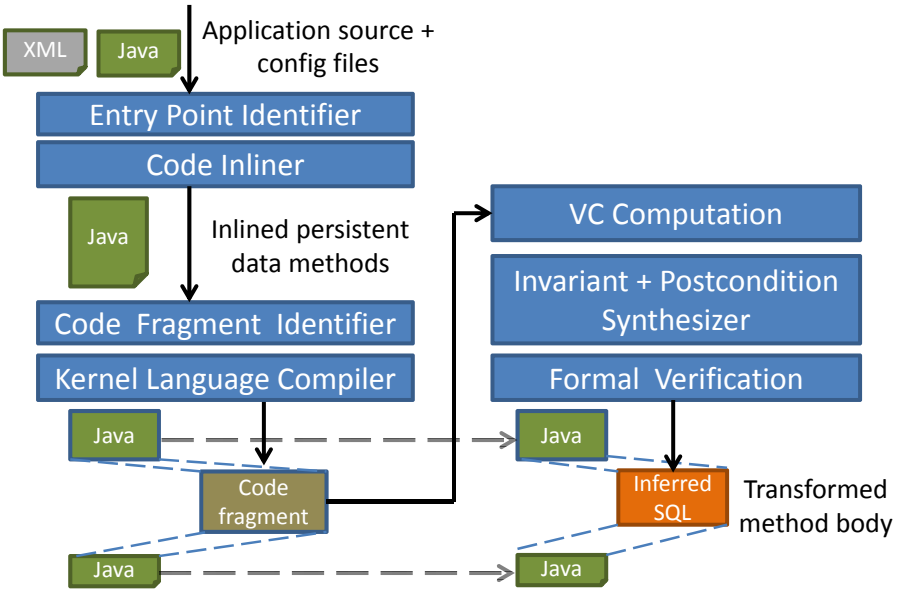


Figure 4.4: QBS architecture

program variable that will contain the results from the inferred query (in the case of the running example it is `listUsers`) — we refer this as the "result variable."

In order to apply the QBS algorithm to perform the desired conversion, our system must be able to cope with the complexities of real-world Java code such as aliasing and method calls, which obscure opportunities for transformations. For example, it would not be possible to transform the code fragment in Figure 4.1 without knowing that `getUsers` and `getRoles` execute specific queries on the database and return non-aliased lists of results, so the first step of the system is to identify promising code fragments and translate them into a simpler kernel language shown in Figure 4.5.

The kernel language operates on three types of values: scalars, immutable records, and immutable lists. Lists represent the collections of records and are used to model the results that are returned from database retrieval operations. Lists store either scalar values or records constructed with scalars, and nested lists are assumed to be appropri-

$$
\begin{array}{rcl}
c \in \text{constant} & ::= & \text{True} \mid \text{False} \mid \text{number literal} \mid \text{string literal} \\
e \in \text{expression} & ::= & c \mid [\,] \mid \mathit{var} \mid e.f \mid \{f_i = e_i\} \mid e_1 \ \mathit{op}\ e_2 \mid \neg\, e \\
& \mid & \text{Query}(\ldots) \mid \text{size}(e) \mid \text{get}_{e_r}(e_s) \\
& \mid & \text{append}(e_r, e_s) \mid \text{unique}(e) \\
c \in \text{command} & ::= & \text{skip} \mid \mathit{var} := e \mid \text{if}(e)\ \text{then}\ c_1\ \text{else}\ c_2 \\
& \mid & \text{while}(e)\ \text{do}\ c \mid c_1\ ;\ c_2 \mid \text{assert}\ e \\
\mathit{op} \in \text{binary op} & ::= & \wedge \mid \vee \mid > \mid =
\end{array}
$$

Figure 4.5: Abstract syntax of the kernel language

ately flattened. The language currently does not model the three-valued logic of null values in SQL, and does not model updates to the database. The semantics of the constructs in the kernel language are mostly standard, with a few new ones introduced for record retrievals. Query(...) retrieves records from the database and the results are returned as a list. The records of a list can be randomly accessed using get, and records can be appended to a list using append. Finally, unique takes in a list and creates a new list with all duplicate records removed. Figure 4.2 shows the example translated to the kernel language. At the end of the code identification process, QBS would have selected a set of code fragments that are candidates to be converted into SQL, and furthermore compile each of them into the kernel language as discussed.

Compute verification conditions. As the next step, the system computes the verification conditions of the code fragment expressed in the kernel language. The verification conditions are written using the predicate language derived from the theory of ordered relations to be discussed in Section 4.3. The procedure used to compute verification conditions is a fairly standard one [Dijkstra, 1975, Gries, 1987]; the only twist is that the verification condition must be computed in terms of an unknown postcondition and loop invariants. The process of computing verification conditions is discussed in more detail in Section 4.4.1.

Synthesize loop invariants and postconditions. The definitions

of the postcondition and invariants need to be filled in and validated before translation can proceed. QBS does this using a synthesis-based approach that is similar to prior work [Srivastava and Gulwani, 2009], where a synthesizer is used to come up with a postcondition and invariants that satisfy the computed verification conditions. The synthesizer uses a symbolic representation of the space of candidate postconditions and invariants, and efficiently identifies candidates within the space that are correct according to a bounded verification procedure. It then uses a theorem prover (Z3 [Microsoft], specifically) to check if those candidates can be proven correct. The space of candidate invariants and postconditions is described by a template generated automatically by the compiler. To prevent the synthesizer from generating trivial postconditions (such as True), the template limits the synthesizer to only generate postconditions that can be translated to SQL as defined by our theory of ordered relations, such as that shown at the top of Figure 4.3.

As mentioned earlier, we observe that it is not necessary to determine the strongest invariants or postconditions: we are only interested in finding postconditions that allow us transform the input code fragment into SQL. In the case of the running example, we are only interested in finding a postcondition of the form `listUsers = Query(...)`, where *Query*(...) is an expression translatable to SQL. Similarly, we only need to discover loop invariants that are strong enough to prove the postcondition of interest. From the example shown in Figure 4.1, our system infers the postcondition shown at the top of Figure 4.3, where π, σ, and $\bowtie$ are ordered versions of relational projection, selection, and join, respectively to be defined in Section 4.3. The process of automatic template generation from the input code fragment and synthesis of the postcondition from the template are discussed in Section 4.4.

Unfortunately, determining loop invariants is undecidable for arbitrary programs [Blass and Gurevich, 2001], so there will be programs for which the necessary invariants fall outside the space defined by the templates. However, our system is significantly more expressive than the state of the art as demonstrated by our experiments in Section 4.7.

Convert to SQL. After the theorem prover verifies that the com-

puted invariants and postcondition are correct, the input code fragment is translated to SQL, as shown in the bottom of Figure 4.3. The predicate language defines syntax-driven rules to translate any expressions in the language into valid SQL. The details of validation is discussed in Section 4.5 while the rules for SQL conversion are introduced in Section 4.3.2. The converted SQL queries are patched back into the original code fragments and compiled as Java code.

4.3 Theory of Finite Ordered Relations

As discussed in Section 4.2, QBS synthesizes a postcondition and the corresponding verification conditions for a given code fragment before converting it into SQL. To do so, we need a language to express these predicates, and axioms that allow us to reason about terms in this language. For this purpose, QBS uses a theory of finite ordered relations. The theory is defined to satisfy four main requirements: precision, expressiveness, conciseness, and ease of translation to SQL. For precision, we want to be able to reason about *both* the contents and order of records retrieved from the database. This is important because in the presence of joins, the order of the result list will not be arbitrary even when the original list was arbitrary, and we do not know what assumptions the rest of the program makes on the order of records. The theory must also be expressive enough not just to express queries but also to express invariants, which must often refer to partially constructed lists. For instance, the loop invariants for the sample code fragment in Figure 4.1 must express the fact that `listUsers` is computed from the first `i` and `j` records of `users` and `roles` respectively. Conciseness, e.g., the number of relational operators involved, is important because the complexity of synthesis grows with the size of the synthesized expressions, so if we can express invariants succinctly, we will be able to synthesize them more efficiently. Finally, the inferred postconditions must be translatable to standard SQL.

There are many ways to model relational operations, but we are not aware of any that fulfills all of the criteria above. For example, relational algebra is not expressive enough to describe sufficiently precise loop

$$
\begin{array}{rcl}
c \in \text{constant} & ::= & \text{True} \mid \text{False} \mid \text{number literal} \mid \text{string literal} \\
e \in \text{expression} & ::= & c \mid [\,] \mid \text{program var} \mid \{f_i = e_i\} \mid e_1 \ op \ e_2 \mid \neg\, e \\
& \mid & \text{Query}(\ldots) \mid \text{size}(e) \mid \text{get}_{e_s}(e_r) \mid \text{top}_{e_s}(e_r) \\
& \mid & \pi_{[f_{i_1},\ldots,f_{i_N}]}(e) \mid \sigma_{\varphi_\sigma}(e) \mid \bowtie_{\varphi_\bowtie}(e_1, e_2) \\
& \mid & \text{sum}(e) \mid \max(e) \mid \min(e) \\
& \mid & \text{append}(e_r, e_s) \mid \text{sort}_{[f_{i_1},\ldots,f_{i_N}]}(e) \mid \text{unique}(e) \\
op \in \text{binary op} & ::= & \wedge \mid \vee \mid > \mid = \\
\varphi_\sigma \in \text{select func} & ::= & p_{\sigma_1} \wedge \ldots \wedge p_{\sigma_N} \\
p_\sigma \in \text{select pred} & ::= & e.f_i \ op \ c \mid e.f_i \ op \ e.f_j \mid \text{contains}(e, e_r) \\
\varphi_\bowtie \in \text{join func} & ::= & p_{\bowtie_1} \wedge \ldots \wedge p_{\bowtie_N} \\
p_\bowtie \in \text{join pred} & ::= & e_1.f_i \ op \ e_2.f_j
\end{array}
$$

Figure 4.6: Abstract syntax for the predicate language based on the theory of ordered relations

invariants. Defined in terms of sets, relational algebra cannot naturally express concepts such as "the first `i` elements of the list." First order logic (FOL), on the other hand, is very expressive, but it would be hard to translate arbitrary FOL expressions into SQL.

4.3.1 Basics

Our theory of finite ordered relations is essentially relational algebra defined in terms of lists instead of sets. The theory operates on three types of values: scalars, records, and ordered relations of finite length. Records are collections of named fields, and an ordered relation is a finite list of records. Each record in the relation is labeled with an integer index that can be used to fetch the record. Figure 4.6 presents the abstract syntax of the theory and shows how to combine operators to form expressions.

$$\boxed{\text{size}}$$

$$\frac{r = [\,]}{\text{size}(r) = 0} \quad \frac{r = h : t}{\text{size}(r) = 1 + \text{size}(t)}$$

$$\boxed{\text{get}}$$

$$\frac{i = 0 \quad r = h : t}{\text{get}_i(r) = h} \quad \frac{i > 0 \quad r = h : t}{\text{get}_i(r) = \text{get}_{i-1}(t)}$$

$$\boxed{\text{append}}$$

$$\frac{r = [\,]}{\text{append}(r, t) = [t]} \quad \frac{r = h : t}{\text{append}(r, t') = h : \text{append}(t, t')}$$

$$\boxed{\text{top}}$$

$$\frac{r = [\,]}{\text{top}_r(i) = [\,]} \quad \frac{i = 0}{\text{top}_r(i) = [\,]} \quad \frac{i > 0 \quad r = h : t}{\text{top}_r(i) = h : \text{top}_t(i-1)}$$

$$\boxed{\text{join } (\bowtie)}$$

$$\frac{r_1 = [\,]}{\bowtie_\varphi (r_1, r_2) = [\,]} \quad \frac{r_2 = [\,]}{\bowtie_\varphi (r_1, r_2) = [\,]}$$

$$\frac{r_1 = h : t}{\bowtie_\varphi (r_1, r_2) = \ \text{cat}(\bowtie'_\varphi (h, r_2),\ \bowtie_\varphi (t, r_2))}$$

$$\frac{r_2 = h : t \quad \varphi(e, h) = \text{True}}{\bowtie'_\varphi (e, r_2) = (e, h) : \bowtie'_\varphi (e, t)} \quad \frac{r_2 = h : t \quad \varphi(e, h) = \text{False}}{\bowtie'_\varphi (e, r_2) = \bowtie'_\varphi (e, t)}$$

$$\boxed{\text{projection } (\pi)}$$

$$\frac{r = [\,]}{\pi_\ell(r) = [\,]} \quad \frac{r = h : t \quad f_i \in \ell \quad h.f_i = e_i}{\pi_\ell(r) = \{f_i = e_i\} : \pi_\ell(t)}$$

Figure 4.7: Axioms that define the theory of ordered relations

$$\boxed{\text{selection } (\sigma)}$$

$$\frac{r = [\,]}{\sigma_\varphi(r) = [\,]}$$

$$\frac{r = h : t \quad \varphi(h) = \text{True}}{\sigma_\varphi(r) = h : \sigma_\varphi(t)} \quad \frac{r = h : t \quad \varphi(h) = \text{False}}{\sigma_\varphi(r) = \sigma_\varphi(t)}$$

$$\boxed{\text{sum}}$$

$$\frac{r = [\,]}{\text{sum}(r) = 0} \quad \frac{r = h : t}{\text{sum}(r) = h + \text{sum}(t)}$$

$$\boxed{\text{max}}$$

$$\frac{r = [\,]}{\text{max}(r) = -\infty}$$

$$\frac{r = h : t \quad h > \text{max}(t)}{\text{max}(r) = h} \quad \frac{r = h : t \quad h \leq \text{max}(t)}{\text{max}(r) = \text{max}(t)}$$

$$\boxed{\text{min}}$$

$$\frac{r = [\,]}{\text{min}(r) = \infty}$$

$$\frac{r = h : t \quad h < \text{min}(t)}{\text{min}(r) = h} \quad \frac{r = h : t \quad h \geq \text{min}(t)}{\text{min}(r) = \text{min}(t)}$$

$$\boxed{\text{contains}}$$

$$\frac{r = [\,]}{\text{contains}(e, r) = \text{False}}$$

$$\frac{e = h \quad r = h : t}{\text{contains}(e, r) = \text{True}} \quad \frac{e \neq h \quad r = h : t}{\text{contains}(e, r) = \text{contains}(e, t)}$$

Figure 4.7 continued

The semantics of the operators in the theory are defined recursively by a set of axioms; a sample of which is shown in Figure 4.7. get and top take in an ordered relation e_r and return the record stored at index e_s or all the records from index 0 up to index e_s respectively. The definitions for π, σ and $\bowtie$ are modeled after relational projection, selection, and join respectively, but they also define an order for the records in the output relation relative to those in the input relations. The projection operator π creates new copies of each record, except that for each record only those fields listed in $[f_{i_1}, \ldots, f_{i_N}]$ are retained. Like projection in relational algebra, the same field can be replicated multiple times. The σ operator uses a selection function φ_σ to filter records from the input relation. φ_σ is defined as a conjunction of predicates, where each predicate can compare the value of a record field and a constant, the values of two record fields, or check if the record is contained in another relation e_r using contains. Records are added to the resulting relation if the function returns True. The $\bowtie$ operator iterates over each record from the first relation and pairs it with each record from the second relation. The two records are passed to the join function $\varphi_\bowtie$. Join functions are similar to selection functions, except that predicates in join functions compare the values of the fields from the input ordered relations. The axioms that define the aggregate operators max, min, and sum assume that the input relation contains only one numeric field, namely the field to aggregate upon.

The definitions of unique and sort are standard; in the case of sort, $[f_{i_1}, \ldots, f_{i_N}]$ contains the list of fields to sort the relation by. QBS does not actually reason about these two operations in terms of their definitions; instead it treats them as uninterpreted functions with a few algebraic properties, such as

$$\bowtie_\varphi(\mathsf{sort}_{\ell_1}(r_1), \mathsf{sort}_{\ell_2}(r_2)) = \mathsf{sort}_{\mathsf{cat}(\ell_1,\ell_2)}(\bowtie_\varphi(r_1, r_2)).$$

(where cat concatenates two lists together) Because of this, there are some formulas involving sort and unique that we cannot prove, but we have not found this to be significant in practice (see Section 4.7 for details).

4.3.2 Translating to SQL

The expressions defined in the predicate grammar can be converted into semantically equivalent SQL queries. In this section we prove that any expression that does not use append or unique can be compiled into an equivalent SQL query. We prove this in three steps; first, we define base and sorted expressions, which are formulated based on SQL expressions without and with ORDER BY clauses respectively. Next, we define *translatable expressions* and show that any expression that does not use append or unique can be converted into a translatable expression. Then we show how to produce SQL from translatable expressions.

Definition 4.1 (Translatable Expressions). Any transExp as defined below can be translated into SQL:

$$\begin{aligned} b \in \mathsf{baseExp} &::= \mathsf{Query}(...) \mid \mathsf{top}_e(s) \mid \bowtie_{\mathsf{True}}(b_1, b_2) \mid \mathsf{agg}(t) \\ s \in \mathsf{sortedExp} &::= \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b))) \\ t \in \mathsf{transExp} &::= s \mid \mathsf{top}_e(s) \end{aligned}$$

where the term agg in the grammar denotes any of the aggregation operators (min, max, sum, size).

Theorem 1 (Completeness of Translation Rules). All expressions in the predicate grammar in Figure 4.6, except for those that contain append or unique, can be converted into translatable expressions.

The theorem is proved by defining a function Trans that maps any expression to a translatable expression and showing that the mapping is semantics preserving. The definition of Trans relies on a number of TOR expression equivalences:

Theorem 2 (Operator Equivalence). The following equivalences hold, both in terms of the contents of the relations and also the ordering of the records in the relations:

- $\sigma_\varphi(\pi_\ell(r)) = \pi_\ell(\sigma_\varphi(r))$
- $\sigma_{\varphi_2}(\sigma_{\varphi_1}(r)) = \sigma_{\varphi'}(r)$, where $\varphi' = \varphi_2 \wedge \varphi_1$
- $\pi_{\ell_2}(\pi_{\ell_1}(r)) = \pi_{\ell'}(r)$, where ℓ' is the concatenation of all the fields in ℓ_1 and ℓ_2.

- $\mathsf{top}_e(\pi_\ell(r)) = \pi_\ell(\mathsf{top}_e(r))$
- $\mathsf{top}_{e_2}(\mathsf{top}_{e_1}(r)) = \mathsf{top}_{\mathsf{max}(e_1,e_2)}(r)$
- $\bowtie_\varphi (r_1, r_2) = \sigma_{\varphi'}(\bowtie_{\mathsf{True}} (r_1, r_2))$, i.e., joins can be converted into cross products with selections with proper renaming of fields.
- $\bowtie_\varphi (\mathsf{sort}_{\ell_1}(r_1), \mathsf{sort}_{\ell_2}(r_2)) = \mathsf{sort}_{\ell_1:\ell_2}(\bowtie_\varphi (r_1, r_2))$
- $\bowtie_\varphi (\pi_{\ell_1}(r_1), \pi_{\ell_2}(r_2)) = \pi_{\ell'}(\bowtie_\varphi (r_1, r_2))$, where ℓ' is the concatenation of all the fields in ℓ_1 and ℓ_2.

Except for the equivalences involving sort, the other ones can be proven easily from the axiomatic definitions.

Given the theorem above, the definition of Trans is shown in Figure 4.8. Semantic equivalence between the original and the translated expression is proved using the expression equivalences listed in Thm. 2. Using those equivalences, for example, we can show that for $s \in \mathsf{sortedExp}$ and $b \in \mathsf{baseExp}$:

$$\begin{aligned}
\mathsf{Trans}(\sigma_{\varphi'_\sigma}(s)) &= \mathsf{Trans}(\sigma_{\varphi'}(\pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b))))) && [\mathsf{sortedExp}\text{ def.}] \\
&= \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi_\sigma \wedge \varphi'_\sigma}(b))) && [\mathsf{Trans}\text{ def.}] \\
&= \pi_{\ell_\pi}(\sigma_{\varphi'_\sigma}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi_\sigma}(b)))) && [\text{expression equiv.}] \\
&= \sigma_{\varphi'_\sigma}(\pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi_\sigma}(b)))) && [\text{expression equiv.}] \\
&= \sigma_{\varphi'_\sigma}(s) && [\mathsf{sortedExp}\text{ def.}]
\end{aligned}$$

Thus the semantics of the original TOR expression is preserved.

Translatable expressions to SQL. Following the syntax-directed rules in Figure 4.9, any translatable expression can be converted into an equivalent SQL expression. Most rules in Figure 4.9 are direct translations from the operators in the theory into their SQL equivalents.

One important aspect of the translation is the way that ordering of records is preserved. Ordering is problematic because although the operators in the theory define the order of the output in terms of the order of their inputs, SQL queries are not guaranteed to preserve the order of records from nested sub-queries; e.g., the ordering imposed by an ORDER BY clause in a nested query is not guaranteed to be

$$\boxed{\mathsf{Query}(...)}$$

$$\mathsf{Trans}(\mathsf{Query}(...))) = \pi_{\ell}(\mathsf{sort}_{[\ ]}(\sigma_{\mathsf{True}}(\mathsf{Query}(...))))$$

where ℓ projects all the fields from the input relation.

$$\boxed{\pi_{\ell_2}(t)}$$

$$\begin{aligned}\mathsf{Trans}(\pi_{\ell_2}(s)) &= \pi_{\ell'}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi}(b)))\\ \mathsf{Trans}(\pi_{\ell_2}(\mathsf{top}_e(s))) &= \mathsf{top}_e(\pi_{\ell'}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi}(b))))\end{aligned}$$

where ℓ' is the composition of ℓ_π and ℓ_2.

$$\boxed{\sigma_{\varphi_2}(t)}$$

$$\mathsf{Trans}(\sigma_{\varphi_2}(s)) = \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi\wedge\varphi_2}(b)))$$

$$\begin{aligned}&\mathsf{Trans}(\sigma_{\varphi_2}(\mathsf{top}_e(s)))\\ &\quad= \mathsf{top}_e(\pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_{\varphi\wedge\varphi_2}(b))))\end{aligned}$$

$$\boxed{\bowtie_{\varphi_{\bowtie}}(t_1, t_2))}$$

$$\begin{aligned}&\mathsf{Trans}(\bowtie_{\varphi_{\bowtie}}(s_1, s_2))\\ &\quad= \pi_{\ell'_\pi}(\mathsf{sort}_{\ell'_s}(\sigma_{\varphi'_\sigma}(\bowtie_{\mathsf{True}}(b_1, b_2))))\end{aligned}$$

where $\varphi'_\sigma = \varphi_{\sigma_1} \wedge \varphi_{\sigma_2} \wedge \varphi_{\bowtie}$ with field names properly renamed,
$\ell'_s = \mathsf{cat}(\ell_{s_1}, \ell_{s_2})$, and $\ell'_\pi = \mathsf{cat}(\ell_{\pi_1}, \ell_{\pi_2})$.

$$\begin{aligned}&\mathsf{Trans}(\bowtie_{\varphi}(\mathsf{top}_e(s_1), \mathsf{top}_e(s_2)))\\ &\quad= \pi_{\ell}(\mathsf{sort}_{[\ ]}(\sigma_{\varphi}(\bowtie_{\mathsf{True}}(\mathsf{top}_e(s_1), \mathsf{top}_e(s_2)))))\end{aligned}$$

where ℓ contains all the fields from s_1 and s_2.

Figure 4.8: Definition of Trans

$$\boxed{\mathsf{top}_{e_2}(t)}$$

$$\begin{aligned} \mathsf{Trans}(\mathsf{top}_{e_2}(s)) &= \mathsf{top}_{e_2}(s) \\ \mathsf{Trans}(\mathsf{top}_{e_2}(\mathsf{top}_{e_1}(s))) &= \mathsf{top}_{e'}(s) \end{aligned}$$

where e' is the minimum value of e_1 and e_2.

$$\boxed{\mathsf{agg}(t)}$$

$$\begin{aligned} \mathsf{Trans}(\mathsf{agg}(s)) &= \pi_\ell(\mathsf{sort}_{[\ ]}(\sigma_{\mathsf{True}}(\mathsf{agg}(s)))) \\ \mathsf{Trans}(\mathsf{agg}(\mathsf{top}_e(s))) &= \pi_\ell(\mathsf{sort}_{[\ ]}(\sigma_{\mathsf{True}}(\mathsf{agg}(s)))) \end{aligned}$$

where ℓ contains all the fields from s.

$$\boxed{\mathsf{sort}_{\ell_{s_2}}(t)}$$

$$\begin{aligned} \mathsf{Trans}(\mathsf{sort}_{\ell_{s_2}}(s)) &= \pi_{\ell_\pi}(\mathsf{sort}_{\ell'_s}(\sigma_\varphi(b))) \\ \mathsf{Trans}(\mathsf{sort}_{\ell_{s_2}}(\mathsf{top}_e(s))) &= \mathsf{top}_e(\pi_{\ell_\pi}(\mathsf{sort}_{\ell'_s}(\sigma_\varphi(b)))) \end{aligned}$$

where $\ell'_s = \mathsf{cat}(\ell_s, \ell_{s_2})$.

Let $s = \pi_{\ell_\pi}(\mathsf{sort}_{\ell_s}(\sigma_\varphi(b)))$. Trans is defined on expressions whose subexpressions (if any) are in translatable form, so we have to consider cases where the sub-expressions are either s or $\mathsf{top}_e(s)$. Each case is defined above.

Figure 4.8 continued

$$
\begin{array}{rcl}
[\![\mathsf{Query}(\mathit{string})]\!] & = & (\ \mathit{string}\ ) \\
[\![\mathsf{top}_e(s)]\!] & = & \mathsf{SELECT}\ *\ \mathsf{FROM}\ [\![s]\!]\ \mathsf{LIMIT}\ [\![e]\!] \\
[\![\bowtie_{\mathsf{True}}(t_1, t_2)]\!] & = & \mathsf{SELECT}\ *\ \mathsf{FROM}\ [\![t_1]\!], [\![t_2]\!] \\
[\![\mathsf{agg}(t)]\!] & = & \mathsf{SELECT}\ \mathsf{agg}(\mathit{field})\ \mathsf{FROM}\ [\![t]\!] \\
[\![\pi_{\ell_1}(\mathsf{sort}_{\ell_2}(\sigma_{\varphi_\sigma}(t)))]\!] & = & \mathsf{SELECT}\ [\![\ell_1]\!]\ \mathsf{FROM}\ [\![t]\!]\ \mathsf{WHERE}\ [\![\varphi_\sigma]\!] \\
 & & \mathsf{ORDER\ BY}\ [\![\ell_2]\!], \mathsf{Order}(t) \\
[\![\mathsf{unique}(t)]\!] & = & \mathsf{SELECT\ DISTINCT}\ *\ \mathsf{FROM}\ [\![t]\!] \\
 & & \mathsf{ORDER\ BY}\ \mathsf{Order}(t) \\
[\![\varphi_\sigma(e)]\!] & = & [\![e]\!].f_1\ \mathit{op}\ [\![e]\!]\ \mathsf{AND}\ \ldots\ \mathsf{AND}\ [\![e]\!].f_N\ \mathit{op}\ [\![e]\!] \\
[\![\mathsf{contains}(e, t)]\!] & = & [\![e]\!]\ \mathsf{IN}\ [\![t]\!] \\
[\![[f_{i_1}, \ldots, f_{i_N}]]\!] & = & f_{i_1}, \ldots, f_{i_N}
\end{array}
$$

Figure 4.9: Syntactic rules to convert translatable expressions to SQL

respected by an outer query that does not impose any ordering on the records.

To solve this problem, the translation rules introduce a function Order—defined in Figure 4.10—which scans a translatable expression t and returns a list of fields that are used to order the subexpressions in t. The list is then used to impose an ordering on the outer SQL query with an ORDER BY clause. One detail of the algorithm not shown in the figure is that some projections in the inner queries need to be modified so they do not eliminate fields that will be needed by the outer ORDER BY clause, and that we assume Query(...) is ordered by the order in which the records are stored in the database (unless the query expression already includes an ORDER BY clause).

Append and Unique. The append operation is not included in translatable expressions because there is no simple means to combine two relations in SQL that preserves the ordering of records in the resulting relation.[3] We can still translate unique, however, using

[3]One way to preserve record ordering in list append is to use case expressions in SQL, although some database systems such as SQL Server limit the number of

$$\begin{aligned}
\mathsf{Order}(\mathsf{Query}(...)) &= [\text{record order in DB}] & \mathsf{Order}(\mathsf{agg}(e)) &= [\,] \\
\mathsf{Order}(\mathsf{top}_i(e)) &= \mathsf{Order}(e) & \mathsf{Order}(\mathsf{unique}(e)) &= \mathsf{Order}(e) \\
\mathsf{Order}(\pi_\ell(e)) &= \mathsf{Order}(e) & \mathsf{Order}(\sigma_\varphi(e)) &= \mathsf{Order}(e)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Order}(\bowtie_\varphi (e_1, e_2)) &= \mathsf{cat}(\mathsf{Order}(e_1), \mathsf{Order}(e_2)) \\
\mathsf{Order}(\mathsf{sort}_\ell(e)) &= \mathsf{cat}(\ell, \mathsf{Order}(e))
\end{aligned}$$

Figure 4.10: Definition of Order

the SELECT DISTINCT construct at the outermost level, as Figure 4.9 shows. Using unique in nested expressions, however, can change the semantics of the results in ways that are difficult to reason about (e.g., $\mathsf{unique}(\mathsf{top}_e(r))$ is not equivalent to $\mathsf{top}_e(\mathsf{unique}(r))$). Thus, the only expressions with unique that we translate to SQL are those that use it at the outermost level. In our experiments, we found that omitting those two operators did not significantly limit the expressiveness of the theory.

With the theory in mind, we now turn to the process of computing verification conditions of the input code fragments.

nested case expressions.

$$i \ op \left\{ \begin{array}{c} i \mid \mathsf{size}(\mathit{users}) \mid \mathsf{size}(\mathit{roles}) \mid \mathsf{size}(\mathit{listUsers}) \mid \\ \mathsf{sum}(\pi_\ell(\mathit{users}) \mid \mathsf{sum}(\pi_\ell(\mathit{roles}) \mid \mathsf{max}(\pi_\ell(\mathit{users}) \mid \\ [\text{other relational expressions that return a scalar value}] \end{array} \right\} \wedge$$

$$\mathit{listUsers} = \left\{ \begin{array}{c} \mathit{listUsers} \mid \sigma_\varphi(\mathit{users}) \mid \\ \pi_\ell(\bowtie_\varphi (\mathsf{top}_{e_1}(\mathit{users}), \mathsf{top}_{e_2}(\mathit{roles}))) \mid \\ \pi_\ell(\bowtie_{\varphi_3} (\sigma_{\varphi_1}(\mathsf{top}_{e_1}(\mathit{users}), \sigma_{\varphi_2}(\mathsf{top}_{e_2}(\mathit{roles}))))) \mid \\ [\text{other relational expressions that return an ordered list}] \end{array} \right\}$$

Figure 4.11: Space of possible invariants for the outer loop of the running example.

4.4 Synthesis of Invariants and Postconditions

Given an input code fragment in the kernel language, the next step in QBS is to come up with an expression for the result variable of the form $\mathit{resultVar} = e$, where e is a translatable expression as defined in Section 4.3.2.

4.4.1 Computing Verification Conditions

In order to infer the postcondition, we compute *verification conditions* for the input code fragment using standard techniques from axiomatic semantics [Hoare, 1969]. As in traditional Hoare style verification, computing the verification condition of the while statements involves a loop invariant. Unlike traditional computation of verification conditions, however, both the postcondition and the loop invariants are unknown when the conditions are generated. This does not pose problems for QBS as we simply treat invariants (and the postcondition) as unknown predicates over the program variables that are currently in scope when the loop is entered.

As an example, Table 4.1 shows the verification conditions that are generated for the running example. In this case, the verification conditions are split into two parts, with invariants defined for both loops.

The first two assertions describe the behavior of the outer loop on line 5, with the first one asserting that the outer loop invariant must be true on entry of the loop (after applying the rule for the

Verification conditions for the outer loop (oInv = outerLoopInvariant, iInv = innerLoopInvariant, pcon = postCondition)	
initialization	$\mathsf{oInv}(0, \mathit{users}, \mathit{roles}, [\,])$
loop exit	$i \geq \mathsf{size}(\mathit{users}) \wedge \mathsf{oInv}(i, \mathit{users}, \mathit{roles}, \mathit{listUsers})$ $\rightarrow \mathsf{pcon}(\mathit{listUsers}, \mathit{users}, \mathit{roles})$
perservation	(same as inner loop initialization)
Verification conditions for the inner loop	
initialization	$i < \mathsf{size}(\mathit{users}) \wedge \mathsf{oInv}(i, \mathit{users}, \mathit{roles}, \mathit{listUsers})$ $\rightarrow \mathsf{iInv}(i, 0, \mathit{users}, \mathit{roles}, \mathit{listUsers})$
loop exit	$j \geq \mathsf{size}(\mathit{roles}) \wedge \mathsf{iInv}(i, j, \mathit{users}, \mathit{roles}, \mathit{listUsers})$ $\rightarrow \mathsf{oInv}(i+1, \mathit{users}, \mathit{roles}, \mathit{listUsers})$
preservation	$j < \mathsf{size}(\mathit{roles}) \wedge \mathsf{iInv}(i, j, \mathit{users}, \mathit{roles}, \mathit{listUsers})$ $\rightarrow (\mathsf{get}_i(\mathit{users}).\mathit{id} = \mathsf{get}_j(\mathit{roles}).\mathit{id} \;\wedge$ $\mathsf{iInv}(i, j+1, \mathit{users}, \mathit{roles}, \mathsf{append}(\mathit{listUsers}, \mathsf{get}_i(\mathit{users})))) \vee$ $(\mathsf{get}_i(\mathit{users}).\mathit{id} \neq \mathsf{get}_j(\mathit{roles}).\mathit{id} \;\wedge$ $\mathsf{iInv}(i, j+1, \mathit{users}, \mathit{roles}, \mathit{listUsers}))$

Table 4.1: Verification conditions for the running example

assignments prior to loop entry), and the second one asserting that the postcondition for the loop is true when the loop terminates. The third assertion states that the inner loop invariant is true when it is first entered, given that the outer loop condition and loop invariant are true. The preservation assertion is the inductive argument that the inner loop invariant is preserved after executing one iteration of the loop body. The list *listUsers* is either appended with a record from $\mathsf{get}_i(\mathit{users})$, or remains unchanged, depending on whether the condition for the if statement, $\mathsf{get}_i(\mathit{users}).\mathit{id} = \mathsf{get}_j(\mathit{roles}).\mathit{id}$, is true or not. Finally, the loop exit assertion states that the outer loop invariant is valid when the inner loop terminates.

4.4.2 Constraint based synthesis

The goal of the synthesis step is to derive postcondition and loop invariants that satisfy the verification conditions generated in the previous step. We synthesize these predicates using the SKETCH constraint-based synthesis system [Solar-Lezama et al., 2006]. In general, SKETCH takes as input a program with "holes" and uses a counterexample guided synthesis algorithm (CEGIS) to efficiently search the space of all possible completions to the holes for one that is correct according to a bounded model checking procedure. For QBS, the program is a simple procedure that asserts that the verification conditions hold for all possible values of the free variables within a certain bound. For each of the unknown predicates, the synthesizer is given a *sketch* (i.e., a template) that defines a space of possible predicates which the synthesizer will search. The sketches are automatically generated by QBS from the kernel language representation.

4.4.3 Inferring the Space of Possible Invariants

Recall that each invariant is parameterized by the current program variables that are in scope. Our system assumes that each loop invariant is a conjunction of predicates, with each predicate having the form $\mathit{lv} = e$, where lv is a program variable that is modified within the loop, and e is an expression in TOR.

The space of expressions e is restricted to expressions of the same static type as lv involving the variables that are in scope. The system limits the size of expressions that the synthesizer can consider, and incrementally increases this limit if the synthesizer fails to find any candidate solutions (to be explained in Section 4.4.5).

Figure 4.11 shows a stylized representation of the set of predicates that our system considers for the outer loop in the running example. The figure shows the potential expressions for the program variable i and *listUsers*. One advantage of using the theory of ordered relations is that invariants can be relatively concise. This has a big impact for synthesis, because the space of expressions grows exponentially with respect to the size of the candidate expressions.

4.4.4 Creating Templates for Postconditions

The mechanism used to generate possible expressions for the result variable is similar to that for invariants, but we have stronger restrictions, since we know the postcondition must be of the form $resultVar = e$ in order to be translatable to SQL, and the form is further restricted by the set of translatable expressions discussed in Section 4.3.2.

For the running example, QBS considers the following possible set of postconditions:

$$listUsers = \left\{ \begin{array}{c} users \mid \sigma_{\varphi}(users) \mid \mathsf{top}_{e}(users) \mid \\ \pi_{\ell}(\bowtie_{\varphi}(\mathsf{top}_{e_1}(users), \mathsf{top}_{e_2}(roles))) \mid \\ \pi_{\ell}(\bowtie_{\varphi_3}(\sigma_{\varphi_1}(\mathsf{top}_{e_1}(users), \sigma_{\varphi_2}(\mathsf{top}_{e_2}(roles))))) \mid \\ [\text{other relational expressions that return an ordered list}] \end{array} \right\}$$

4.4.5 Optimizations

The basic algorithm presented above for generating invariant and postcondition templates is sufficient but not efficient for synthesis. In this section we describe two optimizations that improve the synthesis efficiency.

Incremental solving. As an optimization, the generation of templates for invariants and postconditions is done in an iterative manner: QBS initially scans the input code fragment for specific patterns and

creates simple templates using the production rules from the predicate grammar, such as considering expressions with only one relational operator, and functions that contains one Boolean clause. If the synthesizer is able to generate a candidate that can be used to prove the validity of the verification conditions, then our job is done. Otherwise, the system repeats the template generation process, but increases the complexity of the template that is generated by considering expressions consisting of more relational operators, and more complicated Boolean functions. Our evaluation using real-world examples shows that most code examples require only a few (< 3) iterations before finding a candidate solution. Additionally, the incremental solving process can be run in parallel.

Breaking symmetries. Symmetries have been shown to be one of sources of inefficiency in constraint solvers [Torlak and Jackson, 2007, Déharbe et al., 2011]. Unfortunately, the template generation algorithm presented above can generate highly symmetrical expressions. For instance, it can generate the following potential candidates for the postcondition:

$$\sigma_{\varphi_2}(\sigma_{\varphi_1}(\mathit{users}))$$
$$\sigma_{\varphi_1}(\sigma_{\varphi_2}(\mathit{users}))$$

Notice that the two expressions are semantically equivalent to the expression $\sigma_{\varphi_1 \wedge \varphi_2}(\mathit{users})$. These are the kind of symmetries that are known to affect solution time dramatically. The template generation algorithm leverages known algebraic relationships between expressions to reduce the search space of possible expressions. For example, our algebraic relationships tell us that it is unnecessary to consider expressions with nested σ like the ones above. Also, when generating templates for postconditions, we only need to consider translatable expressions as defined in Section 4.3.2 as potential candidates. Our experiments have shown that applying these symmetric breaking optimizations can reduce the amount of solving time by half.

Even with these optimizations, the spaces of invariants considered are still astronomically large; on the order of 2^{300} possible combinations of invariants and postconditions for some problems. Thanks to these

Type	Expression inferred
outer loop invariant	$i \leq \mathsf{size}(\mathit{users}) \ \wedge \ \mathit{listUsers} = \pi_\ell(\bowtie_\varphi (\mathsf{top}_i(\mathit{users}), \mathit{roles}))$
inner loop invariant	$i < \mathsf{size}(\mathit{users}) \wedge j \leq \mathsf{size}(\mathit{roles}) \wedge$ $\mathit{listUsers} = \mathsf{append}(\pi_\ell(\bowtie_\varphi (\mathsf{top}_i(\mathit{users}), \mathit{roles})),$ $\pi_\ell(\bowtie_\varphi (\mathsf{get}_i(\mathit{users}), \mathsf{top}_j(\mathit{roles})))$
postcondition	$\mathit{listUsers} = \pi_\ell(\bowtie_\varphi (\mathit{users}, \mathit{roles}))$

where $\varphi(e_{\mathit{users}}, e_{\mathit{roles}}) := e_{\mathit{users}}.\mathit{roleId} = e_{\mathit{roles}}.\mathit{roleId}$,
ℓ contains all the fields from the *User* class

Figure 4.12: Inferred expressions for the running example

optimizations, however, the spaces can be searched very efficiently by the constraint based synthesis procedure.

4.5 Formal Validation and Source Transformation

After the synthesizer comes up with candidate invariants and postconditions, they need to be validated using a theorem prover, since the synthesizer used in our prototype is only able to perform bounded reasoning as discussed earlier. We have implemented the theory of ordered relations in the Z3 [Microsoft] prover for this purpose. Since the theory of lists is not decidable as it uses universal quantifiers, the theory of ordered relations is not decidable as well. However, for practical purposes we have not found that to be limiting in our experiments. In fact, given the appropriate invariants and postconditions, the prover is able to validate them within seconds by making use of the axioms that are provided.

If the prover can establish the validity of the invariants and postcondition candidates, the postcondition is then converted into SQL according to the rules discussed in Section 4.3.2. For instance, for the running example our algorithm found the invariants and postcondition as shown in Figure 4.12, and the input code is transformed into the results in Figure 4.3.

If the prover is unable to establish validity of the candidates (detected via a timeout), QBS asks the synthesizer to generate other candidate invariants and postconditions after increasing the space of possible solutions as described in Section 4.4.5. One reason that the prover may not be able to establish validity is because the maximum size of the relations set for the synthesizer was not large enough. For instance, if the code returns the first 100 elements from the relation but the synthesizer only considers relations up to size 10, then it will incorrectly generate candidates that claim that the code was performing a full selection of the entire relation. In such cases our algorithm will repeat the synthesis process after increasing the maximum relation size. If the verification is successful, the inferred queries are merged back into the code fragment. Otherwise QBS will invoke the synthesizer to generate another candidate. The process continues until a verified one is found or a time out happens.

4.5.1 Object Aliases

Implementations of ORM libraries typically create new objects from the records that are fetched, and our current implementation will only transform the input source into SQL if all the objects involved in the code fragment are freshly fetched from the database, as in the running example. In some cases this may not be true, as in the code fragment in Figure 4.13.

Here, the final contents of `results1` and `results2` can be aliases to those in `objs1`. In that case, rewriting `results1` and `results2` into two SQL queries with freshly created objects will not preserve the alias relationships in the original code. Our current implementation will not transform the code fragment in that case, and we leave sharing record results among multiple queries as future work.

4.6 Preprocessing of Input Programs

In order to handle real-world Java programs, QBS performs a number of initial passes to identify the code fragments to be transformed to kernel language representation before query inference. The code identification

```
List objs1 = fetchRecordsFromDB();
List results1 = new ArrayList();

for (Object o : objs1) {
  if (f(o))
    results1.add(o);
}

List results2 = new ArrayList();
for (Object o : objs1) {
  if (g(o))
    results2.add(o);
}
```

Figure 4.13: Code fragment with alias in results

process makes use of several standard analysis techniques, and in this section we describe them in detail.

4.6.1 Generating initial code fragments

As discussed in Section 4.2, code identification first involves locating application entry point methods and data persistent methods. From each data persistent method, our system currently inlines a neighborhood of 5 callers and callees. QBS only inline callees that are defined in the application, and provide models for native Java API calls. For callers QBS only inline those that can be potentially invoked from an entry point method. The inlined method bodies are passed to the next step of the process. Inlining improves the precision of the points-to information for our analysis. While there are other algorithms that can be used to obtain such information [Xie and Aiken, 2005, Whaley and Rinard, 1999], we chose inlining for ease of implementation and is sufficient in processing the code fragments used in the experiments.

4.6.2 Identifying code fragments for query inference

Given a candidate inlined method for query inference, QBS next identifies the code fragment to transform to the kernel language representation. While QBS can simply use the entire body of the inlined method for this purpose, we would like to limit the amount of code to be analyzed, since including code that does not manipulate persistent data will increase the difficulty in synthesizing invariants and postconditions with no actual benefit. QBS accomplishes this goal using a series of analyses. First, QBS runs a flow-sensitive pointer analysis [Sagiv et al., 1999] on the body of the inlined method. The results of this analysis is a set of points-to graphs that map each reference variable to one or more abstract memory locations at each program point. Using the points-to information, QBS performs two further analyses on the inlined method.

Location tainting. QBS runs a dataflow analysis that conservatively marks values that are derived from persistent data retrieved via ORM library calls. This analysis is similar to taint analysis [Tripp et al., 2009], and the obtained information allows the system to remove regions of code that do not manipulate persistent data and thus can be ignored for our purpose. For instance, all reference variables and list contents in Figure 4.1 will be tainted as they are derived from persistent data. The results from this analysis are used to identify the boundaries of the code fragment to be analyzed and converted.

Type analysis. As Java uses dynamic dispatch to resolve targets of method calls, we implemented class analysis to determine potential classes for each object in a given code fragment. If there are multiple implementations of the same method depending on the target's runtime type, then QBS will inline all implementations into the code fragment, with each one guarded by a runtime type lookup. For instance, if object `o` can be of type `Bar` or a subtype `Baz`, and that method `foo` is implemented by both classes, then inlining `o.foo()` will result in:

```
if (o instanceof Baz) {
  // implementation of Baz.foo()
} else if (o instanceof Bar) {
  // implementation of Bar.foo()
} else throw new RuntimeException(); // should not reach here
```

This process is applied recursively to all inlined methods.

Def-use analysis. For each identified code fragment, QBS runs a definition-use analysis to determine the relationship among program variables. The results are used after ensure that none of the variables that are defined in the code fragment to be replaced is used in the rest of the inlined method after replacement. And the checking is done before replacing the code fragment with a verified SQL query.

Value escapement. After that, QBS performs another dataflow analysis to check if any abstract memory locations are reachable from references that are outside of the inlined method body. This analysis is needed because if an abstract memory location `m` is accessible from the external environment (e.g., via a global variable) after program point `p`, then converting `m` might break the semantics of the original code, as there can be external references to `m` that rely on the contents of `m` before the conversion. This analysis is similar to classical escape analysis [Whaley and Rinard, 1999]. Specifically, we define an abstract memory location `m` as having escaped at program point `p` if any of the following is true:

- It is returned from the entry point method.
- It is assigned to a global variable that persists after the entry point method returns (in the web application context, these can be variables that maintain session state, for instance).
- It is assigned to a `Runnable` object, meaning that it can be accessed by other threads.
- It is passed in as a parameter into the entry point method.
- It can be transitively reached from an escaped location.

With that in mind, we define the beginning of the code fragment to pass to the QBS algorithm as the program point `p` in the inlined method where tainted data is first retrieved from the database, and the end as the program point `p'` where tainted data first escapes, where `p'` appears after `p` in terms of control flow. For instance, in Figure 4.1 the `return` statement marks the end of the code fragment, with the result variable being the value returned.

4.6.3 Compilation to kernel language

Each code fragment that is identified by the previous analysis is compiled to our kernel language. Since the kernel language is based on value semantics and does not model heap updates for lists, during the compilation process QBS translates list references to the abstract memory locations that they point to, using the results from earlier analysis. In general, there are cases where the preprocessing step fails to identify a code fragment from an inlined method (e.g., persistent data values escape to multiple result variables under different branches, code involves operations not supported by the kernel language, etc.), and QBS will simply skip such cases. However, the number of such cases is relatively small as our experiments show.

4.7 Experiments

In this section we report our experimental results. The goal of the experiments is twofold: first, to quantify the ability of our algorithm to convert Java code into real-world applications and measure the performance of the converted code fragments, and second to explore the limitations of the current implementation.

We have implemented a prototype of QBS. The source code analysis and computation of verification conditions are implemented using the Polyglot compiler framework [Nystrom et al., 2003]. We use Sketch as the synthesizer for invariants and postconditions, and Z3 for validating the invariants and postconditions.

Application	# persistent data code fragments	translated	failed
Wilos	33	29	4
itracker	16	12	4
Total	**49**	**41**	**7**

Table 4.2: Real-world code fragments experiment

4.7.1 Real-World Evaluation

In the first set of experiments, we evaluated QBS using real-world examples from two large-scale open-source applications, Wilos and itracker, written in Java. Wilos (rev. 1196) [Wilos Orchestration Software] is a project management application with 62k LOC, and itracker (ver. 3.0.1) [itracker Issue Management System] is a software issue management system with 61k LOC. Both applications have multiple contributors with different coding styles, and use the Hibernate ORM library for data persistence operations. We passed in the entire source code of these applications to QBS to identify code fragments. The preprocessor initially found 120 unique code fragments that invoke ORM operations. Of those, it failed to convert 21 of them into the kernel language representation, as they use data structures that are not supported by our prototype (such as Java arrays), or access persistent objects that can escape from multiple control flow points and hence cannot be converted.

Meanwhile, upon manual inspection, we found that those 120 code fragments correspond to 49 distinct code fragments inlined in different contexts. For instance, if A and C both call method B, our system automatically inlines B into the bodies of A and C, and those become two different code fragments. But if all persistent data manipulation happens in B, then we only count one of the two as part of the 49 distinct code fragments. QBS successfully translated 33 out of the 49 distinct code fragments (and those 33 distinct code fragments correspond to 75 original code fragments). The results are summarized in Table 4.2, and the details can be found in Table 4.3.

This experiment shows that QBS can infer relational specifications from a large fraction of candidate fragments and convert them into SQL equivalents. For the candidate fragments that are reported as translat-

wilos code fragments

#	Java Class Name	Line	Oper.	Status	Time (s)
17	ActivityService	401	A	†	–
18	ActivityService	328	A	†	–
19	AffectedtoDao	13	B	✓	72
20	ConcreteActivityDao	139	C	*	–
21	ConcreteActivityService	133	D	†	–
22	ConcreteRoleAffectationService	55	E	✓	310
23	ConcreteRoleDescriptorService	181	F	✓	290
24	ConcreteWorkBreakdown-ElementService	55	G	†	–
25	ConcreteWorkProduct-DescriptorService	236	F	✓	284
26	GuidanceService	140	A	†	–
27	GuidanceService	154	A	†	–
28	IterationService	103	A	†	–
29	LoginService	103	H	✓	125
30	LoginService	83	H	✓	164
31	ParticipantBean	1079	B	✓	31
32	ParticipantBean	681	H	✓	121
33	ParticipantService	146	E	✓	281
34	ParticipantService	119	E	✓	301
35	ParticipantService	266	F	✓	260
36	PhaseService	98	A	†	–
37	ProcessBean	248	H	✓	82
38	ProcessManagerBean	243	B	✓	50
39	ProjectService	266	K	*	–
40	ProjectService	297	A	✓	19
41	ProjectService	338	G	†	–
42	ProjectService	394	A	✓	21
43	ProjectService	410	A	✓	39
44	ProjectService	248	H	✓	150
45	RoleDao	15	I	*	–
46	RoleService	15	E	✓	150
47	WilosUserBean	717	B	✓	23
48	WorkProductsExpTableBean	990	B	✓	52
49	WorkProductsExpTableBean	974	J	✓	50

itracker code fragments

#	Java Class Name	Line	Operation	Status	Time (s)
1	EditProjectFormActionUtil	219	F	✓	289
2	IssueServiceImpl	1437	D	✓	30
3	IssueServiceImpl	1456	L	*	–
4	IssueServiceImpl	1567	C	*	–
5	IssueServiceImpl	1583	M	✓	130
6	IssueServiceImpl	1592	M	✓	133
7	IssueServiceImpl	1601	M	✓	128
8	IssueServiceImpl	1422	D	✓	34
9	ListProjectsAction	77	N	*	–
10	MoveIssueFormAction	144	K	*	–
11	NotificationServiceImpl	568	O	✓	57
12	NotificationServiceImpl	848	A	✓	132
13	NotificationServiceImpl	941	H	✓	160
14	NotificationServiceImpl	244	O	✓	72
15	UserServiceImpl	155	M	✓	146
16	UserServiceImpl	412	A	✓	142

where:
A: selection of records
B: return literal based on result size
C: retrieve the max / min record by first sorting and then returning the last element
D: projection / selection of records and return results as a set
E: nested-loop join followed by projection
F: join using contains
G: type-based record selection
H: check for record existence in list
I: record selection and only return the one of the records if multiple ones fulfill the selection criteria
J: record selection followed by count
K: sort records using a custom comparator
L: projection of records and return results as an array
M: return result set size
N: record selection and in-place removal of records
O: retrieve the max / min record

✓ indicates those that are translated by. QBS
* indicates those that QBS failed to find invariants for.
† indicates those that are rejected by QBS due to TOR / pre-processing limitations.

Table 4.3: Details of the 49 distinct code fragments. The times reported correspond to the time required to synthesize the invariants and postconditions. The time taken for the other initial analysis and SQL translation steps are negligible.

able by QBS, our prototype was able to synthesize postconditions and invariants, and also validate them using the prover. Furthermore, the maximum time that QBS takes to process any one code fragment is under 5 minutes (with an average of 2.1 minutes). In the following, we broadly describe the common types of relational operations that our QBS prototype inferred from the fragments, along with some limitations of the current implementation.

Projections and Selections. A number of identified fragments perform relational projections and selections in imperative code. Typical projections include selecting specific fields from the list of records that are fetched from the database, and selections include filtering a subset of objects using field values from each object (e.g., user ID equals to some numerical constant), and a few use criteria that involve program variables that are passed into the method.

One special case is worth mentioning. In some cases only a single field is projected out and loaded into a set data structure, such as a set of integer values. One way to translate such cases is to generate SQL that fetches the field from all the records (including duplicates) into a list, and eliminate the duplicates and return the set to the user code. Our prototype, however, improves upon that scheme by detecting the type of the result variable and inferring a postcondition involving the unique operator, which is then translated to a SELECT DISTINCT query that avoids fetching duplicate records from the database.

Joins. Another set of code fragments involve join operations. we summarize the join operations in the application code into two categories. The first involves obtaining two lists of objects from two base queries and looping through each pair of objects in a nested for or while loop. The pairs are filtered and (typically) one of the objects from each pair is retained. The running example in Figure 4.1 represents such a case. For these cases, QBS translates the code fragment into a relational join of the two base queries with the appropriate join predicate, projection list, and sort operations that preserve the ordering of records in the results.

Another type of join also involves obtaining two lists of objects from two base queries. Instead of a nested loop join, however, the code iterates through each object e from the first list, and searches if e (or one of e's fields) is contained in the second. If true, then e (or some of its fields) is appended to the resulting list. For these cases QBS converts the search operation into a contains expression in the predicate language, after which the expression is translated into a correlated subquery in the form of SELECT * FROM r1, r2 WHERE r1 IN r2, with r1 and r2 being the base queries.

QBS handles both join idioms mentioned above. However, the loop invariants and postconditions involved in such cases tend to be more complex as compared to selections and projections, as illustrated by the running example in Figure 4.12. Thus, they require more iterations of synthesis and formal validation before finding a valid solution, with up to 5 minutes in the longest case. Here, the majority of the time is spent in synthesis and bounded verification. We are not aware of any prior techniques that can be used to infer join queries from imperative code, and we believe that more optimizations can be devised to speed up the synthesis process for such cases.

Aggregations. Aggregations are used in fragments in a number of ways. The most straightforward ones are those that return the length of the list that is returned from an ORM query, which are translated into COUNT queries. More sophisticated uses of aggregates include iterating through all records in a list to find the max or min values, or searching if a record exists in a list. Aggregates such as maximum and minimum are interesting as they introduce loop-carried dependencies [Allen and Kennedy, 1984], where the running value of the aggregate is updated conditionally based on the value of the current record as compared to previous ones. By using the top operator from the theory of ordered relations, QBS is able to generate a loop invariant of the form $\mathsf{v} = \mathsf{agg}(\mathsf{top}_i(\mathsf{r}))$, where agg represents an aggregate operation, and then translate the postcondition into the appropriate SQL query.

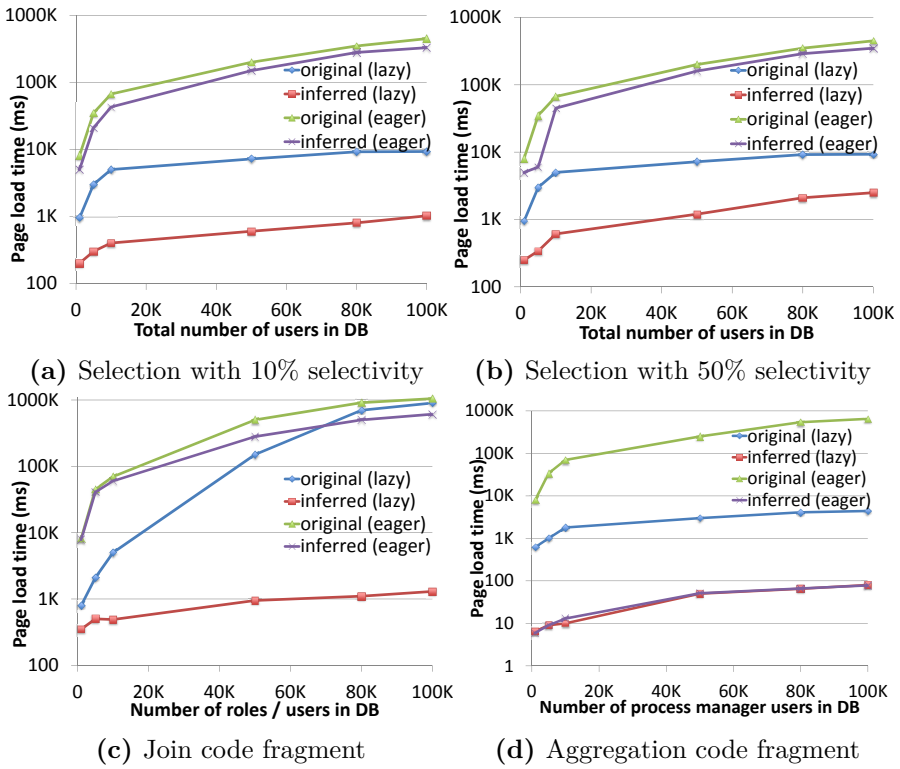
As a special case, a number of fragments check for the existence of a particular record in a relation by iterating over all records and setting a result Boolean variable to be true if it exists. In such cases,

the generated invariants are similar to other aggregate invariants, and our prototype translates such code fragments into `SELECT COUNT(*) > 0 FROM ... WHERE e`, where `e` is the expression to check for existence in the relation. We rely on the database query optimizer to further rewrite this query into the more efficient form using EXISTS.

Limitations. We have verified that in all cases where the generated template is expressive enough for the invariants and postconditions, our prototype does indeed find the solution within a preset timeout of 10 minutes. However, there are a few examples from the two applications where our prototype either rejects the input code fragment or fails to find an equivalent SQL expression from the kernel language representation. Fragments are rejected because they involve relational update operations that are not handled by TOR. Another set of fragments include advanced use of types, such as storing polymorphic records in the database, and performing different operations based on the type of records retrieved. Incorporating type information in the theory of ordered relations is an interesting area for future work. There are also a few that QBS fails to translate into SQL, even though we believe that there is an equivalent SQL query without updates. For instance, some fragments involve sorting the input list by `Collections.sort`, followed by retrieving the last record from the sorted list, which is equivalent to max or min depending on the sort order. Including extra axioms in the theory would allow us to reason about such cases.

4.7.2 Performance Comparisons

Next, we quantify the amount of performance improvement as a result of query inference. To do so, we took a few representative code fragments for selection, joins, and aggregation, and populated databases with different number of persistent objects. We then compared the performance between the original code and our transformed versions of the code with queries inferred by QBS. Since Hibernate can either retrieve all nested references from the database (eager) when an object is fetched, or only retrieve the top level references (lazy), we measured the execution times for both modes (the original application is config-



(a) Selection with 10% selectivity

(b) Selection with 50% selectivity

(c) Join code fragment

(d) Aggregation code fragment

Figure 4.14: Webpage load times comparison of representative code fragments

ured to use the lazy retrieval mode). The results shown in Figure 4.14 compare the time taken to completely load the webpages containing the queries between the original and the QBS inferred versions of the code.

Selection Code Fragment. Figure 4.14a and Figure 4.14b show the results from running a code fragment that includes persistent data manipulations from fragment #40 in Table 4.3. The fragment returns the list of unfinished projects. Figure 4.14a shows the results where 10% of the projects stored are unfinished, and Figure 4.14b shows the results with 50% unfinished projects. While the original version performs the selection in Java by first retrieving all projects from the database, QBS inferred a selection query in this case. As expected, the query inferred

by QBS outperforms the original fragments in all cases as it only needs to retrieve a portion (specifically 10 and 50%) of all persistent objects from the database.

Join Code Fragment. Figure 4.14c shows the results from a code fragment with contents from fragment #46 in Table 4.3 (which is the same as the example from Figure 4.1). The fragment returns the projection of `User` objects after a join of `Roles` and `Users` in the database on the `roleId` field. The original version performs the join by retrieving all `User` and `Role` objects and joining them in a nested loop fashion as discussed in Section 4.2. The query inferred by QBS, however, pushes the join and projection into the database. To isolate the effect of performance improvement due to query selectivity (as in the selection code fragment), we purposefully constructed the dataset so that the query returns all `User` objects in the database in all cases, and the results show that the query inferred by QBS still has much better performance than the original query. This is due to two reasons. First, even though the number of `User` objects returned in both versions are the same, the QBS version does not need to retrieve any `Role` objects since the projection is pushed into the database, unlike the the original version. Secondly, thanks to the automatically created indices on the `Role` and `User` tables by Hibernate, the QBS version essentially transforms the join implementation from a nested loop join into a hash join, i.e., from an $O(n^2)$ to an $O(n)$ implementation, thus improving performance asymptotically.

Aggregation Code Fragment. Finally, Figure 4.14d shows the results from running code with contents from fragment #38, which returns the number of users who are process managers. In this case, the original version performs the counting by bringing in all users who are process managers from the database, and then returning the size of the resulting list. QBS, however, inferred a COUNT query on the selection results. This results in multiple orders of magnitude performance improvement, since the QBS version does not need to retrieve any objects from the database beyond the resulting count.

4.7.3 Advanced Idioms

In the final set of experiments, we used synthetic code fragments to demonstrate the ability of our prototype to translate more complex expressions into SQL. Although we did not find such examples in either of our two real-world applications, we believe that these can occur in real applications.

Hash Joins. Beyond the join operations that we encountered in the applications, we wrote two synthetic test cases for joins that join relations `r` and `s` using the predicate `r.a = s.b`, where `a` and `b` are integer fields. In the first case, the join is done via hashing, where we first iterate through records in `r` and build a hashtable, whose keys are the values of the `a` field, and where each key maps to a list of records from `r` that has that corresponding value of `a`. We then loop through each record in `s` to find the relevant records from `r` to join with, using the `b` field as the look up key. QBS currently models hashtables using lists, and with that our prototype is able recognize this process as a join operation and convert the fragment accordingly, similar to the join code fragments mentioned above.

Sort-Merge Joins. Our second synthetic test case joins two lists by first sorting `r` and `s` on fields `a` and `b` respectively, and then iterating through both lists simultaneously. We advance the scan of `r` as long as the current record from `r` is less than (in terms of fields `a` and `b`) the current record from `s`, and similarly advance the scan of `s` as long as the current `s` record is less than the current `r` record. Records that represent the join results are created when the current record from `r` equals to that from `s` on the respective fields. Unfortunately, our current prototype fails to translate the code fragment into SQL, as the invariants for the loop cannot be expressed using the current the predicate language, since that involves expressing the relationship between the current record from `r` and `s` with all the records that have been previously processed.

Iterating over Sorted Relations. We next tested our prototype with two usages of sorted lists. We created a relation with one unsigned

integer field `id` as primary key, and sorted the list using the `sort` method from Java. We subsequently scanned through the sorted list as follows:

```
List records = Query("SELECT id FROM t");
List results = new ArrayList();
Collections.sort(records); // sort by id
for (int i = 0; i < 10; ++i) {
  results.add(records.get(i));
}
```

Our prototype correctly processes this code fragment by translating it into `SELECT id FROM t ORDER BY id LIMIT 10`. However, if the loop is instead written as follows:

```
List records = Query("SELECT id FROM t");
List results = new ArrayList();
Collections.sort(records); // sort by id
int i = 0;
while (records.get(i).id < 10) {
  results.add(records.get(i));
  ++i;
}
```

The two loops are equivalent since the `id` field is a primary key of the relation, and thus there can at most be 10 records retrieved. However, our prototype is not able to reason about the second code fragment, as that requires an understanding of the schema of the relation, and that iterating over `id` in this case is equivalent to iterating over `i` in the first code fragment. Both of which require additional axioms to be added to the theory before such cases can be converted.

4.8 Summary

In this section, we discussed QBS, a system for inferring relational specifications from imperative code that retrieves data using ORM libraries. Our system automatically infers loop invariants and postconditions associated with the source program, and converts the validated postcondition into SQL queries. Our approach is both sound and precise in preserving the ordering of records. We developed a theory of ordered relations that allows efficient encoding of relational operations

into a predicate language, and we demonstrated the applicability using a set of real-world code examples. The techniques developed in this system represent an instance of verified lifting, which is applicable to the general problem of inferring high-level structure from low-level code representations.

5

Assisting Users Specify Database Queries

In this section we give an overview of systems that help users write database queries. We organize the systems using four different aspects: the intended users of the system, the usage model, the algorithms used to infer the user's intended query, and whether the system includes a refinement mechanism for iterative interactions with the user. We discuss each of the aspects in detail in the following sections.

5.1 Intended Users

Most of the systems we consider in this tutorial focus on helping users who possess little or no prior knowledge of traditional query languages such as SQL. The target users for each system are listed in Table 5.1 and can be categorized into the following:

Users with no programming knowledge. These systems aim to help end-users (*e.g.*, data scientists as mentioned in Section 1) find answers from or manipulate datasets that they have collected. Since the target users have no programming knowledge, such systems provide alternative interfaces for users to express their query needs (*e.g.*, via natural language [Li et al., 2007, Gulwani and Marron, 2014], or

graphically [Tableau Software]), translate user inputs into queries, and return the query results to the users. The intermediate queries that are generated by the system are usually not shown to the users.

Users with application programming experience. Unlike end-users, these users are often developers who have knowledge about application programming, but not necessarily query languages. The goal of such systems is to help users in translating their application code into queries [Cheung et al., 2013, Iu and Zwaenepoel, 2010, Wiedermann et al., 2008], and integrate the retrieved query results programmatically within a larger general-purpose application, such as a web application. As such, these systems resemble traditional compilers that convert application code fragments into semantically equivalent queries. The intention is that transforming code fragments that interact with DBMS into queries will improve application performance, since DBMSs come with different optimization strategies and specialized implementations of relational operators, and the developer might not be aware of those when writing the code.

Expert developers. This group of users are experts in query languages who might need help in formulating their queries from one query language to another [Abouzied et al., 2012], debugging their queries [Tran et al., 2009], or suggesting further queries that they might be interested in issuing given prior queries [Khoussainova et al., 2010]. Such systems resemble programming tools that users might use during application development.

System	**Intended Users**
Das Sarma et al [Sarma et al., 2010]	1.
DataPlay [Abouzied et al., 2012, 2013]	2.
Explore-by-Example [Dimitriadou et al., 2014, Çetintemel et al., 2013]	1.
GestureQuery [Jiang et al., 2013]	1.
HadoopToSQL [Iu and Zwaenepoel, 2010]	3.
JReq [Iu et al., 2010]	3.
LifeJoin [Cheung et al., 2012, 2011]	1.
NaLIR [Li and Jagadish, 2014a,b]	1.
NaLIX [Li et al., 2007]	1.
NLyze [Gulwani and Marron, 2014]	1.
Precise [Popescu et al., 2003, 2004]	1.
Query By Example [Zloof, 1975]	1.
Query By Synthesis [Cheung et al., 2013]	3.
QueRIE [Chatzopoulou et al., 2009]	4.
Query by Output [Tran et al., 2009]	4.
Quicksilver [Lu and Bodík, 2013]	1.
SketchStory [Lee et al., 2013]	1.
SnipSuggest [Khoussainova et al., 2010, 2009]	4.
SQLSynthesizer [Zhang and Sun, 2013]	1.
Tableau [Tableau Software]	1.
Wiedermann et al [Wiedermann et al., 2008]	3.

where:

1: users with no knowledge of query languages

2: knowledge about quantified queries

3: Java developers

4: knowledge of SQL

Table 5.1: Intended users of different systems

5.2 Usage Model

The systems we describe reflect a significant degree of experimentation with regards to the kind of interface that a system should present to its users. The ideal usage model will depend on several factors such as the level of expertise of its expected users, and the context in which the system will be used. Additionally, the underlying technologies used by the different systems impose constraints on the usage models and may require additional setup or maintenance tasks from database administrators in order to fulfill their role.

The classification for each system is summarized in Table 5.2. The four main types of user input accepted by these systems are outlined below.

Input-output examples. Systems that are intended for users with little or no knowledge of query languages tend to allow end users to provide input and output examples. Such examples are essentially *partial* specifications for the intended behavior of the system, Formally, such systems solicit a list of input tuples I and output tuples o from the user, with the goal to formulate a query Q such that $O \in Q(I)$, where $o \in O$. Note that the system is free to infer a query Q that retrieves a superset of the outputs provided by the user, with the understanding that the user's intention is to retrieve O; in cases where inputs are not solicited then I is assumed to be all the accessible relations in the database.

There are a variety of ways that users can provide examples to the system. In LifeJoin [Cheung et al., 2012], the system first generates a list of potential output tuples, and the user is presented with an interface that asks her to label the ones that should be in the output set. Explore-by-example [Dimitriadou et al., 2014] follows a similar approach, where the tool presents an initial set of tuples that the user might be interested in, and the user is asked to select those of interest in order for the tool to retrieve more tuples that are similar to those selected.

One drawback of this approach is that the set of queries that can be generated is limited in expressivity to those represented by the tuples

that the system initially presents to the user. For instance, if the user would like to compute the average employee salary but that tuple does not show up in the list of potential output, then the user will not be able to ask the system to generate that query. Furthermore, even if the system is expressive enough to generate tuples with aggregate values, the user might still need to interpret the meaning of each tuple shown before being she can decide whether to include it in the output results or not. While the user might be able to do so for queries that involve simple selection or projection, it might be difficult to do so for queries that involve complex expressions. As an extreme example, the system might generate a query that sum up all values of a given column and present that to the user. In that case it will be very difficult for the user to interpret how that number was derived without any explanation. Finally, the system might overwhelm the user by asking her to manually select from too many potential tuples.

A similar technique is used in SQLSynthesizer [Zhang and Sun, 2013]. Unlike LifeJoin, the user is asked to provide *both* the input and output examples. In SQLSynthesizer, the user defines the input and output relations, and provide a small number of sample tuples for both. While the user no longer needs to manually select from a long list of tuples, it might be difficult for her to come up with the appropriate list of input and output tuples that is constraining enough for the system to find the intended query.

Visual query interface. The second type of interaction mode is by providing the user with a graphical interface. For instance, Query-by-Example [Zloof, 1975] and Quicksilver [Lu and Bodík, 2013] illustrate an interesting combination of graphical interface and soliciting input-output examples from the user. In Query-by-Example, the user is presented with a spreadsheet interface and is asked to provide names for each column in the spreadsheet, where each name is supposed to be the name of some relation's field. Subsequently, the user is asked to provide sample output tuples using the spreadsheet interface, and the tool then proceeds to generate a query. Similarly, Quicksilver allows the user to provide sample tuples using a drag-and-drop interface by directly dragging tuples from the input tables. Both of these systems

retain the same problem formulation as the previous category in terms of a providing partial specification of the intended query.

On the other hand, there are a number of tools that allow users to directly construct queries using a graphical query language. Unlike providing input-output examples, these systems allow users to provide a *full* functional specification of their intended query. For instance, the Dataplay [Abouzied et al., 2012] system provides a graphical interface for users to directly construct and manipulate query trees. The tool assumes users have knowledge about quantified queries and provides an development environment for constructing queries.

A number of tools such as Tableau [Tableau Software] is modeled after traditional data exploration tools [Sarawagi et al., 1998]. These tools allow users to visualize the input data using a spreadsheet interface, and defines a number of visual operators for the user to express their queries in a graphically manner. SketchStory [Lee et al., 2013], on the other hand, allows users to draw free-form visualizations and label them using field names from persistent relations, and the tool proceeds to generate queries given the field names and the structure of the visualization. Similarly, the system described in [Jiang et al., 2013] allows users to specify queries using finger gestures on a touchscreen.

Natural language. Given the advances in natural language processing, the database research community has explored using natural language utterances to directly pose questions to the DBMS [Androutsopoulos et al., 1995]. There has been a number of systems built recently due to the advances in statistical natural language processing [Popescu et al., 2003, Gulwani and Marron, 2014, Li and Jagadish, 2014a, Li et al., 2007]. Unfortunately, due to the ambiguities in natural languages, and the lack of refinement mechanisms, the main difficulty in building such system is how to limit the expressivity of the input so that the system can parse and translate the input into a valid query.

Using other programming languages. A few systems [Cheung et al., 2013, Iu and Zwaenepoel, 2010, Wiedermann et al., 2008, Iu et al., 2010] allow users to provide inputs in terms of a program fragment written in other general-purpose programming languages for ap-

plications (such as Java). Like visual query interfaces, such mechanism also allows users to provide full query specification. As discussed in Section 5.1, these systems target developers rather than end-users who might not have any knowledge about query languages. The goal is to free developers from the need to understand DBMS implementation details, and to automatically identify the application code fragments that should be expressed as a database queries for efficient execution.

5.2.1 Interaction

For some of the interaction models outlined above, the input provided by the user already contains a complete description of the intended queries, so the systems can guarantee that the output fully satisfies the intended query without the need for further interaction. On the other hand, systems where the intended user input is expected to be incomplete or ambiguous require mechanisms to help users understand what is the query that the system inferred and potentially provide additional input if the results are unsatisfactory. The issue of interaction is described in more detail in Section 5.4.

5.2.2 System Setup

Some of the input modes described above, particularly those that rely on machine learning, require an explicit setup step which usually involves collecting some prior data for training purposes. For instance, SnipSuggest [Khoussainova et al., 2010] uses query logs from other users in order to produce query fragment suggestions. Such system is most beneficial when deployed on a large-scale DBMS that is shared among multiple users (*e.g.*, when the DBMS is deployed on the cloud), and that users are willing to contribute their query logs to train the system. Other system setup examples include collecting history of prior interactions between the system and the user [Chatzopoulou et al., 2009], and collecting labeled instances of (query, intention) pairs in order to train a model [Gulwani and Marron, 2014].

The common theme among all such systems is that they all use statistical-based machine learning as search algorithm, and hence they

need to acquire a large number of labeled query instances to bootstrap the algorithm. Because of that, it might take some time before the system has accumulated enough prior data. From the end-user's perspective, it might be difficult for her to tell when the system has become fully functional.

5.3 Search Algorithms

Given user's input, the next step is to infer the query that the user had in mind. In this section we discuss the different search algorithms that have been devised for this purpose, with the results summarized in Table 5.4.

5.3.1 Explicit Search

The simplest way to infer user's query is to exhaustively search through the space of possible queries. Unfortunately, the search space is prohibitively large except for trivial queries. To make the search tractable, systems that use brute force search all come with a number of heuristics to reduce the search space. Das Sarma et al [Sarma et al., 2010] studied the complexity of finding a query Q that describes the relationships between a database D and an existing view V. In this case V can be viewed as a full specification of a result set that the user would like to retrieve, and the task is to infer a query Q such that $Q(D)$ equals (or approximately equals to) V. The work studies the complexity of finding solutions to the problem using explicit search for different types of queries (unsurprisingly most of the interesting types of queries are NP-hard), along with approximation algorithms for each type, in terms of closeness between the original result set V and the one returned by Q.

Other systems use different strategies to prune down the search space. In NaLIR [Li and Jagadish, 2014a], given a natural language input, the system generates all parse trees up to a certain depth. All syntactically valid parse trees are then ranked. Ranking is based on a measure that compares the the syntactic category that each word in the input utterance is assigned to, and whether the assigned category cor-

System	Initial Input	Final system output
Das Sarma et al [Sarma et al., 2010]	query log	view definitions
DataPlay [Abouzied et al., 2012, 2013]	graphical query tree	requested data
Explore-by-Example [Dimitriadou et al., 2014, Çetintemel et al., 2013]	labels on initial set of tuples	requested data with SQL query used
GestureQuery [Jiang et al., 2013]	screen gestures	requested data
HadoopToSQL [Iu and Zwaenepoel, 2010]	Java code	SQL query
JReq [Iu et al., 2010]	Java code	SQL query
LifeJoin [Cheung et al., 2012, 2011]	input-output examples	program to be executed on phones
NaLIR [Li and Jagadish, 2014a,b]	natural language	SQL query
NaLIX [Li et al., 2007]	natural language	Xquery
NLyze [Gulwani and Marron, 2014]	natural language	requested data
Precise [Popescu et al., 2003, 2004]	natural language	requested data
Query By Examples [Zloof, 1975]	graphical query tree	requested data
Query By Synthesis [Cheung et al., 2013]	Java code	SQL query
QueRIE [Chatzopoulou et al., 2009]	query log from user	recommended queries
Query by Output [Tran et al., 2009]	query and database instance	SQL queries that are instance equivalent
Quicksilver [Lu and Bodík, 2013]	input-output examples	requested data
SketchStory [Lee et al., 2013]	visualizations (*e.g.*, graphs)	SQL query
SnipSuggest [Khoussainova et al., 2010, 2009]	query logs from users and partial query	SQL query
SQLSynthesizer [Zhang and Sun, 2013]	input-output examples	requested data
Tableau [Tableau Software]	graphical query tree	data visualization
Wiedermann et al [Wiedermann et al., 2008]	Java code	SQL query

Table 5.2: Usage models of different systems

responds to the information extracted from the database schema. On the other hand, the PRECISE system [Popescu et al., 2003] takes in natural language utterance as input and generates all possible parses of the input by solving a max-flow problem. To differentiate among the different parses, the system uses a query equivalence checker to eliminate those that are deemed as semantically equivalent. The system returns the retrieved answers to the user if it is able to reduce the number of queries to one, otherwise it flags the user input as ambiguous and returns an error. In summary, while using explicit search is guaranteed to find the intended query, the prohibitive cost of enumeration often causes systems to restrict the search space to make the search tractable, as illustrated by the systems described.

5.3.2 Artificial Intelligence Approaches

Many prior systems use algorithms from classification and statistical machine learning to solve the query inference problem. Such systems often fall under two different categories:

Decision tree learning. A number of systems use decision trees to infer the user's query [Dimitriadou et al., 2014, Zhang and Sun, 2013, Tran et al., 2009]. The typical setting is as follows. Given the user inputs (say, a number of desired output tuples), the system first generates a universal relation U by performing a cross product (or a join based on primary key and foreign key relations) on the set of relations that are involved. The system determines the involved relations by examining the field names that are mentioned in the user input. After that, the query inference problem is reduced to inferring the projection and Boolean selection predicates on U. Learning selection predicates is formulated as a classification problem, where the system uses the provided inputs to categorize all tuples into two groups: those that are indicated by the users as belong to the output set, and those that are not. Different systems use different decision tree learning techniques to solve the problem (*e.g.*, using Classification and Regression Tree (CART) [Breiman et al., 1984] or other learning techniques [Frank and Witten, 1998]) to determine the selection predicates. A heuristic is usu-

ally applied to limit the number of predicates involved. The predicates are then translated into filtering conditions in the query. Finally, the list of fields to be projected are computed by comparing the fields in U and the fields included in the user input. If multiple decision trees (*i.e.*, queries) are generated, then the system would rank them according to a predefined metric, such as query complexity.

While decision trees can be used to learned a variety of queries, there are a number of issues associated with them. First, if the database contains a large number of tuples, then physically generating and enumerating each of the tuple in the universal relation U will be expensive. Also, unless the user provided a large number of positive examples, otherwise the number of negative examples can easily overwhelm that of the positive examples due to the number of tuples stored in the database. Such heavily skewed dataset can affect the quality of the learned decision tree [Cieslak and Chawla, 2008]. Finally, it is difficult to learn queries beyond selections and projections (*e.g.*, those that involve aggregates or user-defined functions), unless the system generates all possible means that aggregates can be applied to the input tables and asks users to label the results, but doing so will incur a prohibitive cost.

Statistical techniques. There are other artificial intelligence techniques that are used besides learning decision trees. These techniques tend to be derived from statistical machine learning that are commonly used in recommendation systems. Such systems usually require collecting prior data before making predictions (as discussed in Section 5.2.2). In the general setting, the system first defines a number of "features" associated with each collected data. For instance, features about previous queries from the query log, such as the relations that are involved in each query, the aggregate functions that are used (if any); a summary of previous queries issued by the same user, *etc.* Given such features, the user input is mapped to the same feature space, and the system then infers the query by finding those that are closest to the user input in the feature space. As expected, different systems have proposed various techniques in computing distances and evaluating closeness between the user's input and the candidate solutions.

As a concrete example, given a query fragment, SnipSuggest [Khoussainova et al., 2010] recommends different ways to fill in the rest of the fragment, based on similar queries that other users have previously issued. In order to make recommendations, the system uses aspects such as the relations and fields that are involved in previous queries as features. Given an input query fragment q, the system first maps the fragment to the feature space. Subsequently, it suggests to complete the query fragment by adding k features to it. The list of k features are chosen such that

$$\Sigma_{i=0}^{k} P(e_i|q_1, \ldots, q_n)$$

is maximized. Here $q_1, \ldots, q_n$ represents the set of features that are present in the query fragment q, and the conditional probability $P(e|q)$ is computed by computing from the number of previously issued queries that contain the features e and q, *i.e.*,

$$P(e|q) = \frac{|e \cup q|}{|q|}$$

where $|q|$ refers to the number of previously issued queries where feature q is present. The system also includes other metrics to define query distances and approximation algorithms to efficiently find the list of k features.

Other systems follow a similar approach. For instance, the system proposed in [Chatzopoulou et al., 2009] computes a summary of previously issued queries from a given user, and use that as the feature set in generating recommendations. GestureQuery [Jiang et al., 2013] uses various finger movements detected on the screen as features in order to determine the intended query operator that the user would like to issue.

In summary, statistical based learning techniques are able to infer highly expressive queries (assuming that the input data contain a variety of different query types). However, as discussed in Section 5.2.2, one issue with statistical approaches is that it is often unclear how much prior data is needed before the system can make accurate predictions. Furthermore, the set of features chosen to model the search space can greatly affect the prediction results, and feature selection has been an

active area of research in the machine learning community [Guyon and Elisseeff, 2003].

5.3.3 Domain Specific Languages

In this approach, the system defines a domain specific language (DSL) on top of the one provided by the database. The intention is that end-users might be more comfortable in expressing their data retrieval needs using an application-specific language rather than a general query language. After receiving a user's input in the DSL, the system translates the input into the underlying query language using syntax-driven rules that are typical in classical compilers. The key differences among the systems are the type of DSL (visual, gestural, written, *etc*), and the means that the DSL is compiled to the query language.

For instance, DSLs such as Linq [Microsoft, b] and Links [Cooper et al., 2007] embed query constructs as part of the application language. Developers can express their queries using such constructs, and the DSL compiler will compile such constructs into the underlying query language supported by the database. Similar ideas have been explored earlier in languages such as RIGEL [Rowe and Shoens, 1979] and extensions to Pascal [Schmidt, 1977].

One issue with embedding query language constructs in an application programming language is that developers still need to understand the semantics of such constructs, which are often very similar to relational algebra, in order to make use of them. Other systems use different interaction modes with users in order to understand their query needs. For example, Dataplay [Abouzied et al., 2012] defines a visual DSL based on quantifiers. The system allows the user to directly manipulate relations and other symbols using a graphical interface to construct a quantified logical formula describing the output relation (*e.g.*, to find all high-earning employees that are over 21, the user might write the formula $\forall o \in \mathit{output} \;.\; o.\mathit{age} > 21 \;\wedge\; o.\mathit{salary} > 100,000$ using the graphical interface). Given this declarative specification, the system then compiles the specification into SQL, and the retrieved data are displayed on the graphical interface to the user. Similar DSL are defined in other systems as well, for instance the graphical DSLs supported

by SketchStory [Lee et al., 2013] and Tableau [Tableau Software], the stylized templates that Query By Example [Zloof, 1975] provides users to specify the output relation, the gestural query language discussed in GestureQuery [Jiang et al., 2013] for users issuing queries from a touch-based device, and the restricted grammar in NaLIX [Li et al., 2007] that translates templates of natural language sentences into XML queries. Similarly, systems such as JReq [Iu et al., 2010], HadoopToSQL [Iu and Zwaenepoel, 2010], and the technique proposed in [Wiedermann et al., 2008] transform input code fragments into queries if they are expressed in stylized loop templates. The transformation works by analyzing the input code fragments and using predefined rules to convert various imperative program constructs into query expressions (*e.g.*, converting conditionals in imperative programming language into selection predicates in SQL).

Depending on the design of the DSL, the mapping between statements in the DSL and the underlying query language might not be one-to-one. For instance, in Dataplay, the input formula might be translatable to multiple SQL queries (*e.g.*, projecting extra fields that are not mentioned in the formula). As such, such systems typically provide means for users to provide further constraints, such as providing input-output examples as additional constraints, or warning the user that her input contains ambiguities and asking for revision (see Section 5.4 for details).

In comparison to artificial intelligence techniques, DSL-based systems are less expressive as the space of possible queries that can be inferred are pre-defined by the translation rules embedded in the system. On the other hand, such systems do not incur any learning / setup phrase, and can easily translate the user's input into a query if it is expressed using the DSL.

5.3.4 Program Synthesis

Program synthesis (as discussed in Section 3), is the newest technique available to researchers to assist users in specifying database queries [Lu and Bodík, 2013, Cheung et al., 2012, 2013, Gulwani and Marron, 2014]. Program synthesis allows users to provide constraints using a variety of

mechanisms (*e.g.*, input-output examples, explicit logical constraints, *etc*). Some synthesis techniques themselves rely on explicit search, but they also make use of various techniques to prune the size of the search space in inferring the user's intended query that satisfies all input constraints.

As an illustration, Query By Synthesis (QBS) [Cheung et al., 2013] is an example of such system, where it uses the semantics of the imperative code (*i.e.*, the application code) written by developers as the constraint to guide the search. Give source code written in Java, QBS automatically identifies code fragments that make use of persistent data (by analyzing calls to popular libraries for interacting with databases, such as JDBC [Java Persistence 2.0 Expert Group, 2009] and Hibernate [JBoss]. For each found code fragment, QBS tries to convert them into semantically equivalent SQL queries. Unlike the DSL approach, QBS does not rely on syntax-driven rules to find and convert the input code fragment into SQL. Instead, it compiles the input code fragment to a small kernel language. The kernel language is carefully designed to not model the entire semantics of Java, as many of the program constructs in Java has no semantic equivalents in SQL (*e.g.*, exceptions). Instead, the language includes standard constructs in an imperative language, along with common operations on lists. Figure 5.1 shows a sample Java code fragment, its representation in the kernel language, and the SQL query inferred by QBS.

Next, QBS formulates the problem of finding the SQL query to convert each kernel language code fragment as a search for postconditions (and loop invariants if needed). This allows the system to leverage standard program verification techniques [Hoare, 1969, Floyd, 1967] to validate the transformation once a postcondition is found. To facilitate easy transformation of the postcondition into SQL, the predicate language used to express postconditions (and loop invariants) is based on a theory of ordered relations. The theory itself closely resembles relational algebra (*i.e.*, it includes operators such as selection, projection, and join), except that relations are modeled as ordered lists rather than multisets. Operations on ordered relations are defined using a number of axioms.

```
List<User> getRoleUser () {
  List<User> listUsers = new ArrayList<User>();
  List<User> users = this.userDao.getUsers();
  List<Role> roles = this.roleDao.getRoles();
  for (User u : users) {
    for (Roles r : roles) {
      if (u.roleId().equals(r.roleId())) {
        User userok = u;
        listUsers.add(userok);
  }}}
  return listUsers;
}
```

```
List listUsers := [ ];
int i, j = 0;
List users := Query(SELECT * FROM users);
List roles = Query(SELECT * FROM roles);
while (i < users.size()) {
  while (j < roles.size()) {
    if (users[i].roleId = roles[j].roleId)
      listUsers := append(listUsers, users[i]);
    ++j;
  }
  ++i;
}
```

```
List<User> getRoleUser () {
  List<User> listUsers = db.executeQuery(
       "SELECT u
        FROM users u, roles r
        WHERE u.roleId == r.roleId
        ORDER BY u.roleId, r.roleId");
  return listUsers;
}
```

Figure 5.1: Java code fragment (top), its representation in the QBS kernel language (middle), and the translated code fragment into SQL (bottom).

The search for postcondition and invariants is done using a combination of lightweight code analysis and constraint-based synthesis. First, using Hoare logic, the analyzer first generates a number of logical constraints describing the relationship between the postcondition and each program expression in the code fragment (*e.g.*, if the loop terminates and its invariant is preserved, then the postcondition is true). During this process, the analyzer simply treats the postcondition and any loop invariants as an uninterpreted functions whose definitions are to be filled in later on. In addition, the analyzer also identifies potential "ingredients" of the postcondition and loop invariants (*e.g.*, if the code fragment includes a conditional, then the analysis will include relational selection as a possible candidate). After that, the logical constraints and identified potential components of the postcondition and loop invariants are sent to a program synthesizer. If the synthesizer is able to find a postcondition and loop invariants that satisfy the constraints, the code fragment is converted into its SQL equivalent. As an example, Table 5.3(top) shows the constraints generated by the code analyzer for the code fragment shown in Figure 5.1(top). In the figure outerInvariant, innerInvariant, and postcondition represent the loop invariants for the outer and inner loops, and the postcondition for the code fragment respectively. Each of them is treated as a function call whose definition needs to be synthesized. The synthesized definitions are shown in Table 5.3(bottom).

Quicksilver [Lu and Bodík, 2013] is another system that uses synthesis to infer user queries. Unlike QBS, Quicksilver targets end users who might not have knowledge about query languages. To use the system, the user first uploads her data via a web interface. The system then displays data in a spreadsheet format, and provides a programming-by-example interface for users to indicate tuples that should be in the output. Given the output samples, the system formulates the synthesis problem as:

$$\exists Q(\forall p \in P(\exists r \in R | Q(r) = p))$$

where P represents the set of examples provided by the user, and R is the universal relation (*i.e.*, cross product of all the input relations). The goal of the system is to find a query Q such that Q returns all

Constraints generated for the outer loop	
initialization	$\mathsf{outerInvariant}(0, \mathtt{users}, \mathtt{roles}, [\,])$
loop exit	$\mathtt{i} \geq \mathsf{size}(\mathtt{users}) \wedge \mathsf{outerInvariant}(\mathtt{i}, \mathtt{users}, \mathtt{roles}, \mathtt{listUsers})$ $\rightarrow \mathsf{postcondition}(\mathtt{listUsers}, \mathtt{users}, \mathtt{roles})$
preservation	(same as inner loop initialization)
Constraints generated for the inner loop	
initialization	$\mathtt{i} < \mathsf{size}(\mathtt{users}) \wedge \mathsf{outerInvariant}(\mathtt{i}, \mathtt{users}, \mathtt{roles}, \mathtt{listUsers})$ $\rightarrow \mathsf{innerInvariant}(\mathtt{i}, 0, \mathtt{users}, \mathtt{roles}, \mathtt{listUsers})$
loop exit	$\mathtt{j} \geq \mathsf{size}(\mathtt{roles}) \wedge \mathsf{innerInvariant}(\mathtt{i}, \mathtt{j}, \mathtt{users}, \mathtt{roles}, \mathtt{listUsers})$ $\rightarrow \mathsf{outerInvariant}(\mathtt{i}+1, \mathtt{users}, \mathtt{roles}, \mathtt{listUsers})$
preservation	$\mathtt{j} < \mathsf{size}(\mathtt{roles}) \wedge \mathsf{innerInvariant}(\mathtt{i}, \mathtt{j}, \mathtt{users}, \mathtt{roles}, \mathtt{listUsers})$ $\rightarrow (\mathsf{get}_{\mathtt{i}}(\mathtt{users}).\mathtt{id} = \mathsf{get}_{\mathtt{j}}(\mathtt{roles}).\mathtt{id} \wedge$ $\mathsf{innerInvariant}(\mathtt{i}, \mathtt{j}+1, \mathtt{users}, \mathtt{roles},$ $\mathsf{append}(\mathtt{listUsers}, \mathsf{get}_{\mathtt{i}}(\mathtt{users})))) \vee$ $(\mathsf{get}_{\mathtt{i}}(\mathtt{users}).\mathtt{id} \neq \mathsf{get}_{\mathtt{j}}(\mathtt{roles}).\mathtt{id} \wedge$ $\mathsf{innerInvariant}(\mathtt{i}, \mathtt{j}+1, \mathtt{users}, \mathtt{roles}, \mathtt{listUsers}))$

Function name	**Synthesized definition**
$\mathsf{outerInvariant}(\mathtt{i}, \mathtt{users}, \mathtt{roles}, \mathtt{listUsers})$	$\mathtt{i} \leq \mathsf{size}(\mathtt{users}) \wedge$ $\mathtt{listUsers} =$ $\pi_\ell(\bowtie_\varphi(\mathsf{top}_{\mathtt{i}}(\mathtt{users}), \mathtt{roles}))$
$\mathsf{innerInvariant}(\mathtt{i}, \mathtt{j}, \mathtt{users}, \mathtt{roles}, \mathtt{listUsers})$	$\mathtt{i} < \mathsf{size}(\mathtt{users})$ $\wedge\, j \leq \mathsf{size}(\mathtt{roles}) \wedge$ $\mathtt{listUsers} = \mathsf{append}($ $\pi_\ell(\bowtie_\varphi(\mathsf{top}_{\mathtt{i}}(\mathtt{users}), \mathtt{roles}))$, $\pi_\ell(\bowtie_\varphi(\mathsf{get}_{\mathtt{i}}(\mathtt{users}), \mathsf{top}_{\mathtt{j}}(\mathtt{roles})))$
$\mathsf{postcondition}(\mathtt{listUsers}, \mathtt{users}, \mathtt{roles})$	$\mathtt{listUsers} =$ $\pi_\ell(\bowtie_\varphi(\mathtt{users}, \mathtt{roles}))$

where $\varphi(e_{\mathtt{users}}, e_{\mathtt{roles}}) := e_{\mathtt{users}}.\mathtt{roleId} = e_{\mathtt{roles}}.\mathtt{roleId}$,
ℓ contains all the fields from the User class

Table 5.3: Sample constraints generated by QBS for the code fragment shown in Figure 5.1.

the input examples P when executed on some relation r constructed from the universal relation. Note that Q can return a superset of the examples from P. Like QBS, the system uses CEGIS to learn Q, and validation is performed using the data provided by the user rather than using a specialized theory as in QBS.

Finally, instead of using a synthesis-only approach, NLyze [Gulwani and Marron, 2014] combines synthesis and rule-based translations in learning user queries (as a spreadsheet manipulation script). Given an input utterance (a natural language sentence), the system first enumerates all possible queries based on types of the values that are mentioned in the input. Meanwhile, the system also generates another set of candidate queries using syntax-based translation rules, where the rules are learned using labeled data, as discussed in Section 5.2.2. The candidate queries are then ranked according to a scoring function defined on top of a number of features, among which is the number of times that the translation rules have been applied in generating the candidate query.

Using program synthesis to infer queries has a number of advantages. First, compared to the DSL approach, synthesis serves as a means to dynamically search for the query given the user's input, and does not require devising DSL, or needing to implement and maintain syntax-driven rules for conversion. This allows the general framework to be used in converting other source to target languages. In addition, unlike machine learning techniques, it does not require collecting extensive amount of prior data and devising complex models in order to infer queries of interest. However, as discussed in Section 3, since most synthesizers are based on logic, finding approximate matches (*e.g.*, recommending similar queries rather than finding semantic equivalents) is a challenging task. However, a few systems [Cheung et al., 2012, Gulwani and Marron, 2014] have demonstrated good results by combining program synthesis with other techniques.

5.4 Query Refinement

While many systems have been able to achieve high precision in inferring user's queries, there are occasions where the system fails to find

System	Search Algorithm
Das Sarma et al [Sarma et al., 2010]	explicit search with pruning heuristics
DataPlay [Abouzied et al., 2012, 2013]	generate all queries from input
Explore-by-Example [Dimitriadou et al., 2014, Çetintemel et al., 2013]	decision tree
GestureQuery [Jiang et al., 2013]	classifier and gesture driven rules
HadoopToSQL [Iu and Zwaenepoel, 2010]	syntax driven rules
JReq [Iu et al., 2010]	syntax driven rules
LifeJoin [Cheung et al., 2012, 2011]	program synthesis
NaLIR [Li and Jagadish, 2014a,b]	natural language syntax tree driven rules
NaLIX [Li et al., 2007]	natural language syntax tree driven rules
NLyze [Gulwani and Marron, 2014]	machine learning and program synthesis
Precise [Popescu et al., 2003, 2004]	explicit search modeled as a graph matching problem
Query By Examples [Zloof, 1975]	syntax driven rules
Query By Synthesis [Cheung et al., 2013]	program synthesis
QueRIE [Chatzopoulou et al., 2009]	machine learning
Query by Output [Tran et al., 2009]	decision tree
Quicksilver [Lu and Bodík, 2013]	program synthesis and ranking
SketchStory [Lee et al., 2013]	gesture driven rules
SnipSuggest [Khoussainova et al., 2010, 2009]	machine learning
SQLSynthesizer [Zhang and Sun, 2013]	decision tree and ranking
Tableau [Tableau Software]	syntax driven rules
Wiedermann et al [Wiedermann et al., 2008]	syntax driven rules

Table 5.4: Search algorithms of different systems

System	Refinement mechanism
DataPlay [Abouzied et al., 2012, 2013]	more labeled examples
Explore-by-Example [Dimitriadou et al., 2014, Çetintemel et al., 2013]	more labeled examples
GestureQuery [Jiang et al., 2013]	preview results for user to provide more gestures
LifeJoin [Cheung et al., 2012, 2011]	more labeled examples
NaLIX [Li et al., 2007]	suggest possible valid natural language sentences
SQLSynthesizer [Zhang and Sun, 2013]	more labeled examples

Figure 5.2: Refinement mechanisms (only systems that support refinement are shown)

the intended query using the initial input from the user. For instance, the initial input from the user might be under-specified, or it is ambiguous enough that the system found multiple possibilities. In this section, we discuss different mechanisms that systems have devised in helping users refine their initial inputs and provide additional feedback, with results shown in Figure 5.2.

5.4.1 Ranking Multiple Possibilities

When the system is able to infer multiple different queries given a user's input, one obvious mechanism is to generate an error message to the user, or return all found possibilities and let the user determine which (if any) matches her intention. In cases where the system follows the latter approach [Khoussainova et al., 2010, Li et al., 2007, Gulwani and Marron, 2014], the system usually provides a ranking of the potential queries, where ranking is determined by the complexity of the found queries [Gulwani and Marron, 2014, Li et al., 2007], or by similarities with previously issued queries [Khoussainova et al., 2010]. The user then has the option of selecting one of the queries from the ranked list, or reissuing a new request. One drawback of this approach is that if the user does not have knowledge about the query language, then showing her the inferred queries will not be helpful in helping the user refine her request, unless the system is able to formulate the inferred queries in the input language (*e.g.*, as natural language sentences).

5.4.2 Labeling Additional Tuples

A number of systems allow users to provide additional labeled tuples for query refinement purposes. For instance, explore-by-example [Dimitriadou et al., 2014] would initially show a number of tuples that it believes would be of interest to the user. The user has the option to provide feedback by labeling the list of tuples returned as either positive or negative examples. Based on the labels, the system will refine the set of tuples that is retrieved from the database. The tool is designed to be interactive with the user, until she is satisfied with the tuples returned, at which point the system will generate the final query that it used to retrieve such tuples. Similar techniques are employed in other systems [Zhang and Sun, 2013, Cheung et al., 2012, Lu and Bodík, 2013]. One challenge in using interactive sessions as refinement is that it might require a large number of rounds until the system is able to infer the query that the user has in mind, and that has been a research topic in computational learning theory [Abouzied et al., 2013, Angluin et al., 1992].

5.4.3 Other Techniques

Finally, a few systems have devised alternative techniques for user refinement. For instance, Dataplay [Abouzied et al., 2012] allows users to first construct her query using a visual language. As mentioned in Section 5.3.3, the constructed logical formula might be under-specified. As such, the system would retrieve tuples satisfying the initial user input. In addition, it provides an interface that allows users to label each retrieved tuple, in a manner that resembles input-output examples (as discussed in Section 5.2). The system that uses the additional input to refine the result. On the other hand, given an initial gestural specification, GestureQuery [Jiang et al., 2013] shows a preview of the tuples that would be retrieved (along with statistics about the retrieved tuples), and the user can make refinements by making additional gestures on the screen.

6

Conclusion and Future Work

We have described in previous sections different aspects of prior systems that aim to help users specify database queries. In this section we describe new research opportunities that are enabled by prior research and discuss the challenges involved in each topic.

6.1 Beyond Input-Output Examples

While providing input-output examples is an effective way to solicit initial and subsequent feedback from the user, this mechanism is limited to learning simple queries. As discussed in Section 5.2, it is highly unlikely that users are willing to label large number of tuples during the refinement process, besides the fact that the system might take a long time to enumerate all tuples that are contained in the candidate query (as an extreme case, imagine the user highlighting 5 relations but provide no further specification as initial input to the system, in that case the system will need to perform a cross product of all 5 relations, enumerate each of the resulting tuples, and ask the user to label each one). One way to reduce the number of input-output examples needed from the user is to make use of prior history of user queries,

for instance construct probabilities for different types of queries that the user might issue, and use them to bias the search [Gulwani and Marron, 2014]. An interesting topic is to allow the user design a high-level language to express partial constraints on the desired data. Given that language, the system can provide a *mixed-mode* interface where the user can provide specifications using both input-output examples and relational expressions to the system. DSL-based systems such as Dataplay [Abouzied et al., 2012] have done initial exploration in that aspect.

Besides letting users provide more concise constraints to the system, mixed-mode interfaces also allow systems to learn more complex queries. For instance, data scientists often issue queries that involve complex aggregates and user-defined algebraic functions. Such queries are very difficult for systems to learn using input-output examples. As an example, consider the user providing a tuple with a single numeric value that represents the sum of salaries of a certain department. It would be very difficult for any system to learn such query given the number of vast number of ways for which such number could be computed.

6.2 Extending System Capabilities

All of the systems described above focus on learn data retrieval tasks. It would be interesting to expand such systems to handle other data manipulation tasks as well. For instance, loading data from raw files into DBMS, exporting data from one relation and importing them into another, *etc*. While there has been work done in using usage models and algorithms discussed earlier in data transformations [Kandel et al., 2011], data cleaning [Stonebraker et al., 2013], and spreadsheet manipulations [Gulwani et al., 2012], it would be interesting to make use of such techniques for other data manipulation tasks as well.

6.3 Refinement Techniques

Many previous systems allow users to label tuples as a means to provide specifications to the system. However, all such deployments are limited

in that they only allow users to label each tuple as either positive (*i.e.*, it should be retained result set), or negative (*i.e.*, it should be removed from the result set). An interesting research question is to investigate semantically richer interfaces for the user to provide inputs. For instance, allowing the user to label tuples as "partially correct, " perhaps because it contains all the fields that the user would like to retrieve, but were padded with some extra fields. This feature would also be useful in recommendation systems where the user can assign a score to each returned tuple rather than labeling each with a concrete "yes" or "no."

On the other hand, another interesting topic is to enable the system to explain how each of the output tuples were derived. Seeing such derivations will help the user provide more appropriate examples for refinement purposes. While showing the raw database queries used to retrieve each tuple might not be beneficial (as the user might not understand the query language, as discussed in Section 5.4.1), other means of explanations include showing the *support* of each output tuple (*e.g.*, the set of positive tuples that the user previously labeled), and illustrating the effects of the user labeling an output tuple in terms of the set of tuples that will be added or removed.

6.4 Combining Different Inference Algorithms

As discussed in Section 5.3, each of the algorithms has its own advantages and disadvantages. An exciting area of research has been combining different algorithms in improving the precision in query inference. For instance, NLyze [Gulwani and Marron, 2014] combines natural language processing and program synthesis techniques in inferring user queries. Other combinations are also possible. For example, since classical program synthesizers are not good in situations where the user provides conflicting specifications (*e.g.*, due to input errors or changes in interests), one possibility is to incorporate techniques from machine learning research (such as support vector machines) to handle such uncertainties. Furthermore, techniques such as active learning [Settles, 2010] can also be used in conjunction with program synthesis algo-

rithms such as CEGIS (as discussed in Section 3.3) in reducing the number of user interaction rounds before being able to infer the intended query.

In conclusion, modern data applications have been underserved by traditional query languages, which were designed with different applications in mind. Recent research in alternative means for users to specify queries have resulted in many systems and techniques for that purpose, and synthesis in particular has shown promise in bridging the gap between the user's intention and the need to write queries that are highly efficient. However, there is still a lot of exciting research to be done in this area, and it would be interesting to leverage techniques from other fields (*e.g.*, artificial intelligence) in improving the accuracy of these systems.

References

Azza Abouzied, Joseph M. Hellerstein, and Avi Silberschatz. Playful query specification with DataPlay. *VLDB Endowment*, 5(12):1938–1941, 2012.

Azza Abouzied, Dana Angluin, Christos H. Papadimitriou, Joseph M. Hellerstein, and Avi Silberschatz. Learning and verifying quantified boolean queries by example. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 49–60, 2013.

John R. Allen and Ken Kennedy. Automatic loop interchange. In *International Conference on Compiler Construction*, pages 233–246, 1984.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*, pages 1–17, 2013.

Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

Dana Angluin, Michael Frazier, and Leonard Pitt. Learning conjunctions of horn clauses. *Machine Learning*, 9:147–164, 1992.

Mark Bickford, Christoph Kreitz, Robbert Van Renesse, and Robert Constable. An experiment in formal design using meta-properties. In *DISCEX-II: The 2nd DARPA Information Survivability Conference and Exposition. IEEE*, 2001.

Andreas Blass and Yuri Gurevich. Inadequacy of computable loop invariants. 2(1):1–11, 2001.

Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

Mark Burstein, Drew Mcdermott, Douglas R. Smith, and Stephen J. Westfold. Derivation of glue code for agent interoperation. In *4th Intl. Conf. on Autonomous Agents*, pages 277–284, 2000.

Ugur Çetintemel, Mitch Cherniack, Justin DeBrabant, Yanlei Diao, Kyriaki Dimitriadou, Alexander Kalinin, Olga Papaemmanouil, and Stanley B. Zdonik. Query steering for interactive data exploration. In *Biennial Conference on Innovative Data Systems Research*, 2013.

Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. Query recommendations for interactive database exploration. In *International Conference on Scientific and Statistical Database Management*, pages 3–18, 2009.

Alvin Cheung, Arvind Thiagarajan, and Samuel Madden. Automatically generating interesting events with LifeJoin. In *International Conference on Embedded Networked Sensor Systems*, pages 411–412, 2011.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. In *International Conference on Information and Knowledge Management*, pages 1732–1736, 2012.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 3–14, 2013.

Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: being lazy is a virtue (when issuing database queries). In *ACM SIGMOD International Conference on Management of Data*, pages 931–942, 2014.

Alvin Cheung, Shoaib Kamil, and Armando Solar-Lezama. Bridging the gap between general-purpose and domain-specific compilers with synthesis. In *1st Summit on Advances in Programming Languages*, pages 51–62, 2015.

David A. Cieslak and Nitesh V. Chawla. Learning decision trees for unbalanced data. In *European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 241–256, 2008.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *International Conference on Formal Methods for Components and Objects*, pages 266–296, 2007.

George Copeland and David Maier. Making smalltalk a database system. In *ACM SIGMOD International Conference on Management of Data*, pages 316–325, 1984.

Chris J. Date. *An introduction to database systems (7th ed.)*. Addison-Wesley-Longman, 2000.

David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In *International Conference on Automated Deduction*, pages 222–236, 2011.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-example: an automatic query steering framework for interactive data exploration. In *ACM SIGMOD International Conference on Management of Data*, pages 517–528, 2014.

Django. `http://www.djangoproject.com`. Accessed: 2014-11-22.

Robert Floyd. Assigning meanings to programs. *American Mathematical Society Symposia on Applied Mathematics*, 19:19–31, 1967.

Eibe Frank and Ian H. Witten. Generating accurate rule sets without global optimization. In *International Conference on Machine Learning*, pages 144–151, 1998.

David Gries. *The Science of Programming*. Springer, 1987.

Sumit Gulwani and Mark Marron. NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *ACM SIGMOD International Conference on Management of Data*, pages 803–814, 2014.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 62–73, 2011.

Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, August 2012.

Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

International Organization for Standardization. Information technology – Database languages – SQL – Part 3: Call-Level Interface (SQL/CLI). ISO ISO/IEC 9075-3:2008, Geneva, Switzerland, 2008.

International Organization for Standardization. Information technology – Database languages – SQL – Part 4: Persistent Stored Modules (SQL/PSM). ISO ISO/IEC 9075-4:2011, Geneva, Switzerland, 2011.

itracker Issue Management System. `http://itracker.sourceforge.net/index.html`. Accessed: Apr 5, 2016.

Ming-Yee Iu and Willy Zwaenepoel. HadoopToSQL: a mapReduce query optimizer. In *European Conference on Computer systems*, pages 251–264, 2010.

Ming-Yee Iu, Emmanuel Cecchet, and Willy Zwaenepoel. JReq: Database queries in imperative languages. In *International Conference on Compiler Construction*, pages 84–103, 2010.

Java Persistence 2.0 Expert Group. Java Persistence API. `http://jcp.org/aboutJava/communityprocess/final/jsr317`, 2009.

JBoss. Hibernate documentation. `http://www.hibernate.org`. Accessed: 2014-11-22.

JDBC 4.2 Expert Group. JDBC 4.2 API. `http://jcp.org/aboutJava/communityprocess/mrel/jsr221`, 2014.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering*, pages 215–224, 2010.

Lilong Jiang, Michael Mandel, and Arnab Nandi. GestureQuery: A multi-touch database query interface. *VLDB Endowment*, 6(12):1342–1345, August 2013.

jOOQ. `http://www.jooq.org`. Accessed: 2014-11-22.

Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2016.

Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372, 2011.

Nodira Khoussainova, Magdalena Balazinska, Wolfgang Gatterbauer, YongChul Kwon, and Dan Suciu. A case for a collaborative query management system. In *Biennial Conference on Innovative Data Systems Research*, 2009.

Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. SnipSuggest: Context-aware autocompletion for SQL. *VLDB Endowment*, 4(1):22–33, October 2010.

Bongshin Lee, Rubaiat Habib Kazi, and Greg Smith. SketchStory: Telling more engaging stories with data through freeform sketching. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2416–2425, 2013.

Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *VLDB Endowment*, 8(1):73–84, 2014a.

Fei Li and Hosagrahar V. Jagadish. NaLIR: an interactive natural language interface for querying relational databases. In *ACM SIGMOD International Conference on Management of Data*, pages 709–712, 2014b.

Yunyao Li, Huahai Yang, and H. V. Jagadish. NaLIX: A generic natural language search environment for XML data. *ACM Transactions on Database Systems*, 32(4), 2007.

Guy Lohman. Is Query Optimization a "Solved" Problem? `http://wp.sigmod.org/?author=20`, 2014. Accessed: 2014-11-22.

Edward Lu and Rastislav Bodík. Quicksilver: Automatic synthesis of relational queries. Technical Report UCB/EECS-2013-68, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, May 2013.

Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, August 1992.

Zohar Manna and Richard J. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1(1-2):5–48, 1981.

Microsoft. Entity Framework. `http://msdn.microsoft.com/en-us/data/ef.aspx`, a. Accessed: 2014-11-22.

Microsoft. Language-integrated query. `http://msdn.microsoft.com/en-us/library/bb397926.aspx`, b. Accessed: 2014-11-22.

Microsoft. z3 Theorem Prover. `http://research.microsoft.com/en-us/um/redmond/projects/z3`. Accessed: Apr 5, 2016.

Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *International Conference on Compiler Construction*, pages 138–152, 2003.

Object Management Group. Common Object Request Broker Architecture (CORBA) Specification, Version 3.3. `http://www.omg.org/spec/CORBA/3.3`, 2012.

Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 408–418, 2014.

Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 396–407, 2014.

Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in Starburst. In *ACM SIGMOD International Conference on Management of Data*, pages 39–48, 1992.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. Towards a theory of natural language interfaces to databases. In *International Conference on Intelligent User Interfaces*, pages 149–157, 2003.

Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *International Conference on Computational Linguistics*, 2004.

Xiaolei Qian. The deductive synthesis of database transactions. *ACM Transactions on Database Systems*, 18(4):626–677, December 1993.

Lawrence A. Rowe and Kurt A. Shoens. Data abstraction, views and updates in RIGEL. In *ACM SIGMOD International Conference on Management of Data*, pages 71–81, 1979.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Symposium on Principles of Programming Languages*, pages 105–118, 1999.

Sunita Sarawagi, Rakesh Agrawal, and Nimrod Megiddo. Discovery-driven exploration of olap data cubes. In *International Conference on Extending Database Technology*, pages 168–182, 1998.

Anish Das Sarma, Aditya G. Parameswaran, Hector Garcia-Molina, and Jennifer Widom. Synthesizing view definitions from data. In *International Conference on Database Theory*, pages 89–103, 2010.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 305–316, 2013.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, page 9, 2014.

Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions Database Systems*, 2(3):247–261, September 1977.

Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. Complex query decorrelation. In *International Conference on Data Engineering*, pages 450–458, 1996.

Burr Settles. Active learning literature survey. Technical Report 1648, Department of Computer Sciences, University of Wisconsin, Madison, 2010.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 15–26, 2013.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006.

Squeryl. Squeryl: A Scala ORM for SQL Databases. `http://squeryl.org`. Accessed: 2014-11-22.

Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 223–234, 2009.

Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, Stanley B. Zdonik, Alexander Pagan, and Shan Xu. Data curation at scale: The data tamer system. In *Biennial Conference on Innovative Data Systems Research*, 2013.

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented DBMS. In *International Conference on Very Large Data Bases*, pages 553–564, 2005.

Singaram Subramanian. How to Identify and Resolve Hibernate N+1 SELECT's Problems. `http://architects.dzone.com/articles/how-identify-and-resilve-n1`. Accessed: 2015-01-12.

Tableau Software. `http://www.tableausoftware.com`. Accessed: 2014-11-22.

Emina Torlak and Daniel Jackson. Kodkod: a relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *ACM SIGMOD International Conference on Management of Data*, pages 535–548, 2009.

Jonathan Traugott. Deductive synthesis of sorting programs. *Journal of Symbolic Computation*, 7(6):533–572, June 1989.

Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 87–97, 2009.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 287–296, 2013.

John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 187–206, 1999.

Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *ACM Symposium on Principles of Programming Languages*, pages 199–210, 2007.

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

Wilos Orchestration Software. `http://www.ohloh.net/p/6390`. Accessed: Apr 5, 2016.

Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *ACM Symposium on Principles of Programming Languages*, pages 351–363, 2005.

Sai Zhang and Yuyin Sun. Automatically synthesizing SQL queries from input-output examples. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 224–234, 2013.

Moshé M. Zloof. Query-by-example: The invocation and definition of tables and forms. In *International Conference on Very Large Data Bases*, pages 1–24, 1975.