Computer-Assisted Query Formulation

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

Armando Solar-Lezama  
MIT CSAIL  
asolar@csail.mit.edu
5.4 Combining Different Inference Algorithms ........ 42

References .......... 43
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, but 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 systems 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.
We interact with database management systems (DBMSs) on a daily basis, from financial transactions to online shopping. 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, some of the most 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 DBMSs.

The first of these two classes of applications are extensions of business data processing applications to those with complex business logic, such as social network websites, online shopping applications, etc. The queries issued in these applications are usually generated from pre-designed parameterized query templates that are written by developers, with the actual parameter values (e.g., account number, item names) provided by end users. Unfortunately, existing 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, forcing developers to learn a new language—and often a new programming paradigm. For example, an application developer 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. This “impedance mismatch” problem has plagued application developers for decades. This mismatch is often addressed by application frameworks that eliminate the need to think in terms of two distinct programming models, but do so at a significant performance cost.

The second class of applications are analytical tools designed to help end users understand large and complex datasets. Examples of such applications include visualization generators for large datasets, data exploration tools for data scientists, and outlier detectors for business analysts. These tools often provide an interface for end users to express their query needs by typing their queries, and the tools will then issue the queries against the underlying DBMSs and display the results on the console. The end users of such tools are often data scientists or business analysts, not developers. While query languages such as SQL abstract away many of the DBMS internals, the end users still need to understand concepts from relational algebra such as joins and aggregates, which is often not an easy task for individuals without the proper background. Such “abstraction mismatch” forces end users to re-express their queries using relational algebraic means. For example, scientists who are used to loading and exporting their data as a single raw file will need to learn how to manipulate relational algebraic expressions in order to merge multiple normalized relational tables into a single result set that can be exported. Finally, given that database query optimization is still an ongoing research problem [Lohman, 2014], users need to consider the intricacies of DBMS implementations when writing SQL queries, as a slight syntactic change can have a dramatic impact on execution efficiency. All these issues make existing query interfaces difficult to use for the target users of such tools.

In this tutorial, we survey a set of technologies whose aim is to assist users in bridging the gap between the abstractions they would
like to use to convey their intent and the interfaces supported by the DBMS. We focus on structured (often relational) database systems as that has been the target DBMS of many prior systems (rather than NoSQL systems or key-value stores for instance). These systems focus on raising the level of abstraction for users when they interact with DBMSs and can be categorized using four aspects:

- The intended users of the system: whether it is for application developers or end-users, and how much prior knowledge about query languages and DBMS does the system assumes.

- The usage model of the system: some systems require explicit setup prior to first use, while some rely on continuous interactions with the user when ambiguities arise.

- The algorithms used to translate user inputs to the underlying query language: these techniques range from domain-specific languages that can be compiled to the underlying query language to explicit search for translations to the query language with heuristics to reduce the search space. In addition, the systems also differ from each other in terms of the techniques used to rank the potential queries when multiple possibilities arise.

- Refinement mechanisms: for systems that are intended for end-users, many systems provide ways for users to provide additional inputs when the system inferred multiple candidate queries, or when it fails to understand the user’s intentions.

Many systems that help users specify database queries have used techniques from machine learning and natural language processing in the past. In recent years, with the advances in inductive program synthesis [Bodík and Jobstmann 2013, Gulwani 2010], there has been recent work in applying program synthesis techniques in helping users express their query intentions. Using synthesis techniques, researchers have built systems for both end-users and application developers to specify their queries, and these systems represent new mechanisms to learn queries. Synthesis can be used to automatically bridge the gap between different programming abstractions, and developers no longer
need to learn new query languages. In addition, the generated queries are guaranteed to preserve the semantics of user’s input. Finally, framing query inference as a synthesis problem has given us new insights and perspectives on previous approaches that are based on other techniques. We describe synthesis techniques and their applications in subsequent sections.

In tutorial is structured as follows, we first review how database systems process queries in §2, followed by an introduction to program synthesis in §3. We then discuss each aspect in query specification systems in §4: the intended users of each system §4.1, how users interact with such systems §4.2, the algorithms that different systems have used in matching user’s intentions to actual queries §4.3, and the means that users can provide additional input after the initial one §4.4. We then discuss potential future work in this direction in §5.
In this section we describe how DBMSs process queries and explain why it is difficult for users to express their query needs. We focus on query processing of SQL queries in relational database systems. We break down query processing into a number of components: the query interface, the query language, and query parsing and optimization. Figure 2.1 depicts this characterization.

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. 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, similar to the rows in a spreadsheet table. SQL is an implementation of relational algebra [Codd 1970, Date 2000], except that it models
2.2. Interfaces to DBMS

Since the initial development, relational DBMSs have been designed to be stand-alone systems instead of application libraries to be linked in with the application. The early DBMS implementations 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 process-
Query Processing

ing 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 allow application developers to embed query statements within their application source code as if they were using the command-line interface. During compilation, developers would link their application binaries with connector libraries. These libraries are completely separated from the DBMS implementation itself in order to support functionalities such as issuing queries from a remotely other than the machine that hosts the DBMS. 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 in turn return them to the application by serializing the results using programmatic data structures such as lists or arrays of primitive types.

On the one hand, connector libraries greatly ease DBMS developers as they cleanly abstract away the application, and the DBMS can process such 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 in the application. Worse yet, embedding query statements as raw strings in the application makes debugging difficult, as developers are not aware of errors in the queries until execution time since the raw strings are not parsed or type-checked by the application compiler. Not only that, embedding raw query strings in application code is often the source of various security loopholes such as SQL injection.

In recent years, there has been new frameworks that provide integration with DBMS through application libraries. Such libraries can be separated into two categories. The first category includes query language integrated libraries [Microsoft 5, Squeryl, jOOQ] that provide
2.3. Query Execution

Typical DBMS implementation uses a number of steps to process an incoming SQL query. 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 involved, and in this section we review each of the steps in detail.

Page 9

stylized library calls for relational operations. While developers still need to understand query concepts (such as selections and joins), they no longer need have knowledge about query language syntax, and using such libraries does not incur the same security issues as in 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. Not only do they allow developers to write code that interacts with the DBMS using the language for application logic, they also provide capabilities (e.g., through annotations or configuration files) for developers to label certain classes as persistent. During compilation, the framework would parse each persistent class and generate query statements to populate appropriate tables in the DBMS. In addition, such framework 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 would either return cached objects or translate the request into SQL queries executed by the DBMS. Writing to persistent objects are handled similarly. Using ORM frameworks greatly ease the application developer from the need to understand SQL. Unfortunately, applications written using ORM frameworks are often not efficient, as 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].

2.3 Query Execution

1Similar capabilities have been explored in distributed object research, for instance CORBA [Object Management Group 2012].
2.3.1 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 constructed as a directed dataflow graph. Each node in the graph is either a relational algebra operator, an UDF invocation, an output operator (i.e., prints the received tuples on the console), or a table scan operator of a relation (i.e., retrieve all tuples from the corresponding relation). The translation from SQL syntax to logical query plan is relatively straightforward. The logical query plan is typically represented as a tree with the output operator being the root and the table scan operators as the leaves. Query evaluation starts from the bottom, with each edge in the graph representing the intermediate query results that are forwarded from the flow of tuples from one operator to the next.

The generated logical plan is subsequently optimized by performing a number of rewrites. For instance, constant propagation, evaluating redundant Boolean formulas, etc. 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), combining predicates from two adjacent selection operators as a conjunction. In addition, the plan generator also performs a number of heuristics-based optimization. For example, “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 [Pirahesh et al., 1992, Seshadri et al., 1996], as many query optimization algorithms are operate on flattened expressions. As an illustration, consider three relations student, grade, and course that store information about students, course grades, and course descriptions respectively. The query:

```sql
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'
```
2.3. Query Execution

2.3.1 Query Execution

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 \( \pi \) represents relational projection, \( \Join \) represents a relational join while \( \sigma \) represents a relational selection.

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

2.3.2 Query Optimization

The processed 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. DBMS optimizers typically choose among different query plan implementations by first estimating the cost of plan, where cost is an estimate for the amount of time that would need to execute the query and is computed as a function of the following different aspects:

- 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) would greatly cut down 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, Fig. 2.2(right) shows two possible physical execution plans for the query shown in Fig. 2.2(left). Choosing which plan to execute depends on the factors discussed above along with those listed in the next section.

2.3.3 Issues in Query Optimization

Query optimization has been an active area of research in the database research community for decades. In this section we outline some of the issues that make the problem a challenging one.

Plan enumeration. The number of possible query plans grow exponentially as the number of relations and operators increases. Because of that, it is simply not possible to enumerate each of the query plan during query optimization in order to determine the one with the lowest cost, especially when joins are involved. There has been a lot of work in making plan enumeration tractable. For instance, by considering only a subset of the plans (e.g., “left-deep” trees in which the right child of each join operator is a base relation, or “bushy” trees that resemble a balanced tree) [Graefe 1993, Vance and Maier 1996, Ioannidis and
Kang [1991], using probabilistic sampling [Galindo-Legaria et al. 1994, Waas and Galindo-Legaria 2000], and by using dynamic programming to cut down the cost in plan enumeration [Selinger et al. 1979]. Unfortunately, the inability of query optimizers to enumerate all query plans means that the ultimate plan found is often not the optimal one in terms of cost.

**Algorithms for query execution.** Classical query execution algorithms assume a disk-oriented architecture where persistent data are stored on spin disks, and the DBMS is hosted on a machine with small amount of main memory compared on disk space [Graefe 1993, Graefe et al. 1998]. New algorithms has been devised for new architectures, such as parallel execution for multi-node implementations [De-witt et al. 1990], and more recently for machines with large amount of main memory [Diaconu et al. 2013]. Unfortunately, having many different algorithms to choose for the same query operator makes the optimizer’s job even more difficult.

**Cost estimation.** As discussed in §2.3.2, the query optimizer relies on having accurate cost estimations for each of the query operators when considering different query plans. Unfortunately, the cost associated with executing a query operator is often dependent on the underlying distribution of the input data, which cannot be determined during query compilation. For instance, the cost of a join operator would be close to zero if none of the input tuples satisfies the join predicate. Common approaches for this problem includes sampling the data to estimate cost [Haas and Swami 1995, Lipton et al. 1990], and maintaining various statistics about the input data using histograms for query optimization purposes [Stillger et al. 2001].

In summary, query optimization is an unsolved problem by far [Lohman 2014]. As a result, two semantically equivalent queries can have drastically different performance if they are expressed using slightly different syntax [White 2014], and manually choosing the best syntax to write a query is often tedious and error-prone process.
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. First, we describe deductive systems, which use a set of predefined rules to find the solution given a specification. Next, we discuss inductive systems that uses counter-example guided techniques to solve the problem instead. Such techniques are used in several query inference systems, to be discussed in §4.

3.1 The Problem

Program synthesis aims to automatically generate the implementation of a program given its specification. In the most general form, let \( \phi \) be a functional specification of the program that we would like to generate, and \( \phi \) is a formula expressed in a theory \( T \), \( \phi \) essentially represents a set of constraints that the generated program need to satisfy. The goal of program synthesis is to find a program \( p \), drawn from a space of all possible programs \( P \) written using language \( L \), such that the functional specification is valid for all possible program inputs \( I \), in other words,
we want to find a $p$ such that:

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

is valid. Note that the size of $I$ can be infinite in general. While one possible approach to solving the problem is to exhaustively generate all possible $p$, and check against $I$ (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

Rather than exhaustive search, classical program synthesizers solve the synthesis problem using deductive theorem proving techniques [Manna and Waldinger, 1992]. In deductive systems, the synthesizer comes with a set of predefined $(r, e)$ pairs, where $r$ is a transformation rule, and $e$ is an expression in $L$. Given the specification $\phi$, the system chooses one of the rules $r$ from the set to transform $\phi$ into a logically equivalent specification $\phi'$. Each time when a rule is applied the corresponding $e$ is recorded as part of the synthesized program. The process continues until $\phi'$ reduces to True, which means that the synthesizer has found $p$. If $\phi'$ 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 has 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] Kupferman and Vardi [2001].

### 3.3 Counter-Example Guided Inductive Synthesis

Deductive synthesis techniques are best for domains where users have full knowledge about the program that she would like to be synthesized, a theory (namely $T$) is available to write the specification, and there are
relatively few number of rules that are applicable during each iteration of the synthesis procedure to reduce $\phi$ to True.

Unfortunately, such technique is not very applicable in cases where users only have partial functional specifications (e.g., the user does not provide specifications in $T$, but rather provide sample program inputs and their corresponding outputs), or there is a large number of rules that can are applicable to reduce the original specification.

In recent years, there has been a lot of interest in counter-example guided synthesis. Rather than searching for a program $p$ that satisfies all inputs $I$, Counter-Example Guided Inductive Synthesis (CEGIS) [Solar-Lezama et al., 2006, Alur et al.] is one of the recently developed algorithms for solving the synthesis problem by converting the problem into an iterative one, with the idea to find a candidate program that works for an increasing subset of inputs from $I$ during each step. CEGIS is closely related to similar concepts in model checking [Clarke et al., 2000]. First, the synthesizer chooses a candidate program $p_1$ from $P$, and consults an oracle (e.g., by encoding the problem into a SMT formula) if $p_1$ satisfies the functional specification provided by the user (either in terms of a logical formula in $T$ or satisfying all input-output examples provided by the user). If it does, then we have found the solution. Otherwise, the oracle will return a counter-example $c_1$ (i.e., an input-output pair) that falsifies $p_1$. $c_1$ is added to the original list of input-output 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.

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], or the length of synthesized program [Phothilimthana et al., 2014, Perelman et al., 2014]. Other
3.3. Counter-Example Guided Inductive Synthesis

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. Inductive program synthesizers can be categorized into three different types according to the technique used to perform the search for candidate programs \( p \) during each iterative step of synthesis:

**Constraint-based.** These systems translate the functional specification from the user into a number of constraints on the candidate program. Synthesizing candidate programs is then formulated as a search for \( p \) that satisfies all given 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 aims to find candidate programs using different stochastic search algorithms, such as random or Monte Carlo sampling. Such technique 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.** Explicitly 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. 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 candidate programs, where two candidate programs are deemed as semantically equivalent if they behave the same on the set of counter-examples. If both candidates pass the counter-examples found thus far, then one of them is eliminated in the beginning of the next synthesis iteration. Otherwise, both of them are removed.

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 used to reduce the number of iterations required for synthesis.
In this section we give an overview of recent 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. In the following sections we discuss each of the aspects in detail.

4.1 Intended Users

As discussed earlier, most prior systems that help users formulate database queries focus on direct interaction with the end users who possess little or no prior knowledge of the query language (such as SQL). Such systems tend to help users formulate their intended queries [Khoussainova et al., 2010], suggest potential queries that users might like to issue [Chatzopoulou et al., 2009], for debugging / query plan space exploration [Tran et al., 2009], or simply to retrieve the data from persistent storage [Cheung et al., 2013].

However, a few systems [Khoussainova et al., 2010, Cheung et al., 2013, Iu and Zwaenepoel, 2010, Wiedermann et al., 2008, Iu et al.,
are intended for developers, who may have some knowledge about DBMS implementations. For instance, SnipSuggest [Khoussainova et al., 2010] is intended to interact with developers directly by suggesting SQL query fragments that might match the developers’ intentions. Another set of systems [Cheung et al., 2013, Iu and Zwaenepoel, 2010, Wiedermann et al., 2008, Iu et al., 2010] resemble traditional compilers, which transform portions of the application written by developers into a semantically equivalent representation using a query language. Unlike end-users (e.g., data scientists) who might export the retrieved query results for further analysis, these systems aim to help developers who integrate the retrieved query results programmatically within a larger general-purpose application, such as web applications. 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. Figure 4.1 lists the intended users for each system.

4.2 Usage Model

In this section, we describe the usage model of each of the systems. Here usage model refers to setting up each system before initial use, how users provide inputs to the system, and the output of each. We broadly categorize the usage models of the various systems. Figure 4.2 summarizes the various usage models to be discussed.

4.2.1 System Setup

A few of the systems 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.
### 4.2. Usage Model

<table>
<thead>
<tr>
<th>System</th>
<th>Intended Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Das Sarma et al. [Sarma et al., 2010]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>DataPlay [Abouzied et al., 2012, 2013]</td>
<td>knowledge about quantified queries</td>
</tr>
<tr>
<td>Explore-by-Example [Dimitriadou et al., 2014, Çetintemel et al., 2013]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>GestureQuery [Jiang et al., 2013]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>HadoopToSQL [Ju and Zwaenepoel, 2010]</td>
<td>Java developers</td>
</tr>
<tr>
<td>JReq [Ju et al., 2010]</td>
<td>Java developers</td>
</tr>
<tr>
<td>LifeJoin [Cheung et al., 2012, 2011]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>NaLIR [Li and Jagadish, 2014a, b]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>NaLIX [Li et al., 2007]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>NLyze [Gulwani and Marron, 2014]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>Precise [Popescu et al., 2003, 2004]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>Query By Example [Zloof, 1975]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>Query By Synthesis [Cheung et al., 2013]</td>
<td>Java developers</td>
</tr>
<tr>
<td>QueRIE [Chatzopoulou et al., 2009]</td>
<td>knowledge of SQL</td>
</tr>
<tr>
<td>Query by Output [Tran et al., 2009]</td>
<td>knowledge of SQL</td>
</tr>
<tr>
<td>Quicksilver [Lu and Bodik, 2013]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>SketchStory [Lee et al., 2013]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>SnipSuggest [Khoussainova et al., 2010, 2009]</td>
<td>knowledge of SQL</td>
</tr>
<tr>
<td>SQLSynthesizer [Zhang and Sun, 2013]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>Tableau [Tableau Software]</td>
<td>users with no knowledge of query languages</td>
</tr>
<tr>
<td>Wiedermann et al. [Wiedermann et al., 2008]</td>
<td>Java developers</td>
</tr>
</tbody>
</table>

**Figure 4.1:** Intended users of different systems
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.

4.2.2 User Input

Next, we consider the method of interaction between the intended users and the system. Besides obvious inputs such as the relations that are accessible in the database, the systems need to provide a means for users to specify their queries. We find that the interaction mode is often reflected by the type of intended user, and we categorize them into three types 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 \( O \), with the intention that the user’s intention is to retrieve \( O \), and in cases where inputs are not solicited then \( I \) is assumed to be all the accessible relations in the database.

There is a variety of ways that users can provide them 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
4.2. Usage Model

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 setting 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 able to 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 Bodik, 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 [Androtou-soulos et al., 1995]. There has been a number of systems built recent 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, 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.
4.3. Search Algorithms

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 applications (such as Java). Like visual query interfaces, such mechanism also allows users to provide full query specification. As discussed in §4.1, 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 in order to identify the code fragments that should be expressed as a database.

4.2.3 System Output

The outputs generated by the systems described above can be put into two categories. First, for systems that are targeted to end-users, they simply retrieve the data that are intended by the user and return such data. While this is the most direct means to return data to the user, one drawback of this is that in case where the inferred query does not match with the user’s original intention, it would be difficult for users to provide feedback to the system for refinement purposes (see §4.4 for details), especially when a large number of incorrect tuples is returned or missing from the output.

On the other hand, systems that target developers return the inferred query to the users. However, since the users have already provided full specification of their intended queries, the inferred queries are guaranteed to be semantically equivalent to the user’s input. As such, users do not need to interpret the inferred queries as in the other first case.

4.3 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 Fig. 4.5.
<table>
<thead>
<tr>
<th>System</th>
<th>Initial Input</th>
<th>Final system output</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataPlay [Abouzied et al., 2012]</td>
<td>query log</td>
<td>view definitions</td>
</tr>
<tr>
<td>Das Sarma et al. [Sarma et al., 2010]</td>
<td>graphical query tree</td>
<td>requested data</td>
</tr>
<tr>
<td>Explore-by-Example [Dimitriadou et al., 2014]</td>
<td>labels on initial set of tuples</td>
<td>requested data with SQL query used</td>
</tr>
<tr>
<td>GestureQuery [Jiang et al., 2013]</td>
<td>screen gestures</td>
<td>requested data</td>
</tr>
<tr>
<td>HadoopToSQL [Zwaenepoel, 2010]</td>
<td>Java code</td>
<td>SQL query</td>
</tr>
<tr>
<td>JReq [Jiang et al., 2010]</td>
<td>Java code</td>
<td>SQL query</td>
</tr>
<tr>
<td>LifeJoin [Cheung et al., 2012]</td>
<td>input-output examples</td>
<td>program to be executed on phones</td>
</tr>
<tr>
<td>NaLIR [Li and Jagadish, 2014a,b]</td>
<td>natural language utterance</td>
<td>SQL query</td>
</tr>
<tr>
<td>NaLIX [Li et al., 2007]</td>
<td>natural language utterance</td>
<td>Xquery</td>
</tr>
<tr>
<td>NLyze [Gulwani and Marron, 2014]</td>
<td>natural language utterance</td>
<td>requested data</td>
</tr>
<tr>
<td>Precise [Popescu et al., 2003]</td>
<td>natural language utterance</td>
<td>requested data</td>
</tr>
<tr>
<td>Query By Examples [Zloof, 1975]</td>
<td>graphical query tree</td>
<td>requested data</td>
</tr>
<tr>
<td>Query By Synthesis [Cheung et al., 2013]</td>
<td>Java code</td>
<td>SQL query</td>
</tr>
<tr>
<td>QueRIE [Chatzopoulou et al., 2009]</td>
<td>query log from user</td>
<td>recommended queries</td>
</tr>
<tr>
<td>Query by Output [Tran et al., 2009]</td>
<td>query and database instance</td>
<td>SQL queries that are instance equivalent</td>
</tr>
<tr>
<td>Quicksilver [Lu and Bodík, 2013]</td>
<td>input-output examples</td>
<td>requested data</td>
</tr>
<tr>
<td>SketchStory [Lee et al., 2013]</td>
<td>visualizations (e.g., graphs)</td>
<td>SQL query</td>
</tr>
<tr>
<td>SnipSuggest [Khoussainova et al., 2010]</td>
<td>query logs from users and partial query</td>
<td>SQL query</td>
</tr>
<tr>
<td>SQLSynthesizer [Zhang and Sun, 2009]</td>
<td>input-output examples</td>
<td>requested data</td>
</tr>
<tr>
<td>Tableau [Tableau Software, 2013]</td>
<td>graphical query tree</td>
<td>data visualization</td>
</tr>
<tr>
<td>Wiedermann et al. [Wiedermann et al., 2008]</td>
<td>Java code</td>
<td>SQL query</td>
</tr>
</tbody>
</table>

**Figure 4.2:** Usage models of different systems
4.3. Search Algorithms

4.3.1 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, 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 corresponds 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.

4.3.2 Artificial Intelligence Approaches

Many prior systems use algorithms from classification and statistical machine learning to solve the query inference problem. Such systems
Assisting Users Specify Database Queries

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 usually 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].

**Statistical techniques.** There are other artificial intelligence tech-
4.3. Search Algorithms

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 §4.2.1). 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

\[
\sum_{i=0}^{k} P(e_i|q_1, \ldots, q_n)
\]

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 §4.2.1 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].

### 4.3.3 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] 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 applica-
4.3. Search Algorithms

tion 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 $\forall o \in output. o.age > 21 \land o.salary > 100,000$ 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 query (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
Assisting Users Specify Database Queries

that her input contains ambiguities and asking for revision (see §4.4 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.

4.3.4 Program Synthesis

With the recent advances in program synthesis (as discussed in §3, researchers have used such technique to build a new suite of tools to assist users specify database queries [Lu and Bodík, 2013, Cheung et al., 2012, 2013, Gulwani and Marron, 2014]. Query By Synthesis (QBS) [Cheung et al., 2013] is an example of such system. 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. Fig. 4.3 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 lan-
4.3. Search Algorithms

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 4.3: Java code fragment (top), its representation in the QBS kernel language (middle), and the translated code fragment into SQL (bottom).
Assisting Users Specify Database Queries

The language used to express postcondition (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.

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 function 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, Fig. 4.4(top) shows the constraints generated by the code analyzer for the code fragment shown in Fig. 4.3(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 Fig. 4.4(bottom).

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
4.3. Search Algorithms

Constraints generated for the outer loop

<table>
<thead>
<tr>
<th>Initialization</th>
<th>outerInvariant(0, users, roles, [])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop exit</td>
<td>i ≥ size(users) ∧ outerInvariant(i, users, roles, listUsers) → postcondition(listUsers, users, roles)</td>
</tr>
<tr>
<td>Preservation</td>
<td>(same as inner loop initialization)</td>
</tr>
</tbody>
</table>

Constraints generated for the inner loop

<table>
<thead>
<tr>
<th>Initialization</th>
<th>i &lt; size(users) ∧ outerInvariant(i, users, roles, listUsers) → innerInvariant(i, 0, users, roles, listUsers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop exit</td>
<td>j ≥ size(roles) ∧ innerInvariant(i, j, users, roles, listUsers) → outerInvariant(i + 1, users, roles, listUsers)</td>
</tr>
<tr>
<td>Preservation</td>
<td>j &lt; size(roles) ∧ innerInvariant(i, j, users, roles, listUsers) → (get_i(users).id = get_j(roles).id ∧ innerInvariant(i, j + 1, users, roles, listUsers) append(listUsers, get_i(users))) ∨ (get_i(users).id ≠ get_j(roles).id ∧ innerInvariant(i, j + 1, users, roles, listUsers))</td>
</tr>
</tbody>
</table>

Function name | Synthesized definition

<table>
<thead>
<tr>
<th>outerInvariant(i, users, roles, listUsers)</th>
<th>i ≤ size(users) ∧ listUsers = π_ℓ(ϕ(top_i(users), roles))</th>
</tr>
</thead>
<tbody>
<tr>
<td>innerInvariant(i, j, users, roles, listUsers)</td>
<td>i &lt; size(users) ∧ j ≤ size(roles) ∧ listUsers = append( π_ℓ(ϕ(top_i(users), roles)), π_ℓ(ϕ(get_i(users), top_j(roles))))</td>
</tr>
<tr>
<td>postcondition(listUsers, users, roles)</td>
<td>listUsers = π_ℓ(ϕ(users, roles))</td>
</tr>
</tbody>
</table>

where ϕ(e\_users, e\_roles) := e\_users.roleId = e\_roles.roleId, ℓ contains all the fields from the User class

Figure 4.4: Sample constraints generated by QBS for the code fragment shown in Fig. 4.3.
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 §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.

4.4 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 the intended query within given 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 Fig. 4.6.

4.4.1 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 reissue 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
### 4.4. Query Refinement

<table>
<thead>
<tr>
<th>System</th>
<th>Search Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Das Sarma et al.</td>
<td>explicit search with pruning heuristics</td>
</tr>
<tr>
<td>DataPlay</td>
<td>generate all queries from input</td>
</tr>
<tr>
<td>Explore-by-Example</td>
<td>decision tree</td>
</tr>
<tr>
<td>GestureQuery</td>
<td>classifier and gesture driven rules</td>
</tr>
<tr>
<td>HadoopToSQL</td>
<td>syntax driven rules</td>
</tr>
<tr>
<td>JReq</td>
<td>program synthesis</td>
</tr>
<tr>
<td>LifeJoin</td>
<td>natural language syntax tree driven rules</td>
</tr>
<tr>
<td>NaLIR</td>
<td>natural language syntax tree driven rules</td>
</tr>
<tr>
<td>NaLIX</td>
<td>machine learning and program synthesis</td>
</tr>
<tr>
<td>NLyze</td>
<td>explicit search modeled as a graph matching problem</td>
</tr>
<tr>
<td>Precise</td>
<td>syntax driven rules</td>
</tr>
<tr>
<td>Query By Examples</td>
<td>program synthesis</td>
</tr>
<tr>
<td>Query By Synthesis</td>
<td>machine learning</td>
</tr>
<tr>
<td>QueRIE</td>
<td>decision tree</td>
</tr>
<tr>
<td>Query by Output</td>
<td>program synthesis and ranking</td>
</tr>
<tr>
<td>Quicksilver</td>
<td>gesture driven rules</td>
</tr>
<tr>
<td>SketchStory</td>
<td>machine learning</td>
</tr>
<tr>
<td>SnipSuggest</td>
<td>decision tree and ranking</td>
</tr>
<tr>
<td>SQLSynthesizer</td>
<td>syntax driven rules</td>
</tr>
<tr>
<td>Tableau</td>
<td>syntax driven rules</td>
</tr>
<tr>
<td>Wiedermann et al.</td>
<td>syntax driven rules</td>
</tr>
</tbody>
</table>

*Figure 4.5: Search algorithms of different systems*
Assisting Users Specify Database Queries

4.4.2 Labeling Additional Tuples

A number of systems allow users to provide additional labeled tuples for query refinement purposes. For instance, explore-by-example [Dimtriadou 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 Bodik, 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].

4.4.3 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 §4.3.3, 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 §4.2.2). 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.
4.4. Query Refinement

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

**Figure 4.6:** Refinement mechanisms (only systems that support refinement are shown)
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.

5.1 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 §4.2.2 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). An interesting topic is to allow the user design a lightweight logic language for users to express relational constraints. Given that language, the
5.2. Extending System Capabilities

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.

5.2 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.

5.3 Refinement Techniques

Many previous systems allow users to use tuple labeling as a means for user 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 “par-
Conclusion and Future Work

The output tuple is 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 “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 §4.4.1), other means of explanations include show 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.

5.4 Combining Different Inference Algorithms

As discussed in §4.3, 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 algorithms such as CEGIS (as discussed in §3.3) in reducing the number of user interaction rounds before being able to infer the intended query.


