QAGen: Generating Query-Aware Test Databases

Carsten Binnig‡ Donald Kossmann† Eric Lo†
Marc Nunkesser† Tamer Ozsu§
† ETH Zurich, Switzerland
‡ University of Heidelberg, Germany
§ University of Waterloo, Canada
Contact Email: eric.lo@inf.ethz.ch

Abstract

Today, a common methodology for testing a database management system (DBMS) is to generate a set of test databases and then execute queries on top of them. However, for DBMS testing, it would be a big advantage if we can control the input and/or the output (e.g., the cardinality) of each individual operator of a test query for a particular test case. Unfortunately, current database generators generate databases independent of queries. As a result, it is hard to guarantee that executing the test query on the generated test databases can obtain the desired (intermediate) query results that match the test case. In this paper, we propose a novel way for DBMS testing. Instead of first generating a test database and then seeing how well it matches a particular test case (or otherwise use a trial-and-error approach to generate another test database), we propose to generate a query-aware database for each test case. To that end, we designed a query-aware test database generator called QAGen. In addition to the database schema and the set of basic constraints defined on the base tables, QAGen takes the query and the set of constraints defined on the query as input, and generates a query-aware test database as output. The generated database guarantees that the test query can get the desired (intermediate) query results as defined in the test case. This approach of testing facilitates a wide range of DBMS testing tasks such as testing of memory managers and testing the cardinality estimation components of query optimizers.

1 Introduction

When introducing a new component or technique into a DBMS, it is often necessary to validate its correctness and evaluate the relative system improvements under a wide range of test cases and workloads. Today, a common methodology is to first generate a comprehensive set of test databases and then execute many test queries over the generated databases to compare the system behavior before and after the new component is incorporated. For generating test databases, current database generation tools allow a user to define the sizes and the data characteristics (e.g., value distributions and
inter/intra-table correlations) of the base tables. Examples include IBM DB2 Database Generator [2], DTM Data Generator [1], MUDD [24], or some research prototypes such as [16], [17] and [7]. Based on the generated test databases, the next step is to either create test queries manually, or stochastically generate many valid test queries by query generation tools such as RAGS [23] or QGEN [22] and execute them to test the system.

Unfortunately, the current testing approach is inadequate to test individual DBMS components because very often it is necessary to control the input/output of the intermediate operators of a query during a test. For example, assume that we want to test how a newly designed memory manager in a DBMS influences the correctness and/or the performance of multi-way hash-join queries (i.e., how the per-operator memory allocation strategy affects the resulting execution plans). Figure 1 shows a sample test case (figure extracted from [8]) that is used for testing. A test case is a parametric query $Q_p$ with a set of operator constraints. In Figure 1, the test query of the test case first joins a large filtered table $S$ with a filtered table $R$ to get a small join result. Then the small intermediate join result is joined with a filtered table $T$ to obtain a small final result. Since the memory requirement of a hash join is determined by the size of its inputs, it would be beneficial if we can control the input/output of each individual operator in the query tree. For example, the memory allocated to $\bowtie R\.attr_4 = S\.attr_5$ by the memory manager, can be studied by defining the output cardinality constraint on the join $\sigma(R) \bowtie \sigma(S)$ and the output cardinality constraint on $\sigma(T)$ in the test case. However, although we can instruct the database engine to evaluate the test query by a specific physical evaluation plan (e.g., fixing the join order and forcing the use of hash-join as the join algorithm), there is currently no easy way to control the intermediate results of a query.

1.1 The DBMS Testing Problem

The DBMS testing problem is to guarantee that executing a test query on a test database can obtain the desired intermediate query results (e.g., output cardinalities, join distributions) that are defined by the test case. Figure 2 illustrates the problem in a
conceptual way. In Figure 2(a)-(d), there are two test cases, $T_1$ and $T_2$ (denoted by dots); and there are three generated test database instances (denoted by squares) in Figure 2(a)-(c) respectively. In general, a good test database should cover the test cases (i.e., one can choose the parameter value set $P$ such that $Q_P$ meets the constraint set of the test case). However, in traditional test database generation, the generation process does not take the queries into account. Therefore, it is hard to guarantee that executing the test query $Q_P$ on the generated database can obtain the desired (intermediate) query results defined by the test case. Figure 2a shows this problem. The three generated test databases (Database 1, 2, and 3) do not cover the test case $T_2$ at all (i.e., executing the test query of $T_2$ on Database 1, 2, and 3 can never fulfill the constraints defined by $T_2$). Furthermore, even if a test database covers a test case (e.g., Database 1 covers $T_1$), it is difficult to manually find the correct parameter values $P$ of the test query such that the resulting query matches the constraints defined by the test case. For instance, it is unlikely that instantiating the test query $Q_P$ in Figure 2a manually with three sets of parameter values $P', P'', P'''$ can match the requirements of test case $T_1$.

Given a test database, query generation tools such as RAGS and QGEN generate many queries in order to cover a variety of test cases. However, RAGS and QGEN were not designed for testing an individual DBMS component. To test an individual DBMS component, the desired test query is usually given by a tester (e.g., the query in Figure 1). In this situation, RAGS and QGEN would require an extremely large amount of time to generate a query that matches the test query and the requirements of the test case (see Figure 2b). In addition, RAGS and QGEN also rely on what databases they are working on or otherwise they never can generate a test query that matches the test case (e.g., $T_2$).

The importance of this DBMS testing problem has been pointed out by Bruno et al. [8]. Given a test database $D$, a parametric conjunctive query $Q_P$, and cardinality constraints $C$ over the sub-expressions of $Q_P$, they studied how to find the parameter values $P$ of $Q_P$ such that the output cardinality of each operator in $Q_P$ fulfills $C$. In their pioneering work, they found that their problem is $NP$-hard. Their approach is illustrated in Figure 2c. Given the predefined test databases (e.g., Database 1, 2, and 3), it may be possible that there are no parameter values that can let the test query $Q_P$ match the requirements defined by the test case $T_2$. Furthermore, even if a test
database covers a test case (e.g., Database 1), since the solution space is too large, they can only search for approximate solutions (i.e., finding parameter values that make the resulting query with cardinalities on operators close to those specified by the test case) for select-project-join queries with only single-sided predicates (e.g., \( p_1 \leq a \) or \( a \leq p_2 \)) or double-sided predicates (e.g., \( p_1 \leq a \leq p_2 \)) (where \( a \) is an attribute and \( p_1 \) and \( p_2 \) are parameter values).

We observe that the test database generation process is the main culprit of ineffective DBMS testing. Currently, test databases are generated without taking the test query into account. Thus the generated databases cannot guarantee that executing the test query on them can obtain the desired (intermediate) query results defined by the test case. Therefore, the only way for meaningful testing is to do a painful trial-and-error test database generation process (e.g., generate Database 3, 2 and then 1 to find a database that matches \( T_1 \) in Figure 2(a)-(c)), and execute queries generated by RAGS/QGEN, or execute test queries with parameters instantiated by [8].

1.2 Contributions

In this paper, we address the DBMS testing problem in a different and novel way: instead of first generating a test database and then seeing if it is possible for the test query to obtain the desired query results that match the test case (otherwise use a trial-and-error approach to generate another test database), we propose to generate a specific test database tailored for each test case (see Figure 2d). To that end, we present a Query-Aware database Generator (QAGen). Given a database schema \( M \), a parametric query \( Q_P \), and a set of user-defined constraints on each query operator, QAGen directly generates a database \( D \) and query parameter values \( P \) such that executing \( Q_P \) with parameter values \( P \) on \( D \) guarantees that the user requirements imposed on the query operators are fulfilled. QAGen can generate test databases for a variety of complex queries such as TPC-H queries (although we will show that some rare cases are still \( \mathcal{NP} \)-hard for which we give efficient heuristics to solve them). The test databases generated by QAGen can be used in a number of ways in DBMS testing. For example, in addition to testing the memory manager, we can use QAGen to generate a test database that guarantees the size of the intermediate join results to test the accuracy of the cardinality estimation components (e.g., histograms) inside a query optimizer by fixing the join order\(^1\). As another example, we can use QAGen to generate a test database that guarantees the input and the output sizes (the number of groups) for an aggregation operator (GROUP-BY) in order to evaluate the performance of the aggregation algorithm under a variety of cases (e.g., in multi-way join queries or in nested queries).

Another contribution of QAGen is the novel way of defining the test database.

\(^1\)However, it is inapplicable to test the join reordering feature of a query optimizer directly because in this case the physical join ordering should not be fixed by the tester; and the intermediate cardinalities guaranteed by QAGen may affect the optimizer that resulting in a different physical evaluation plan with different intermediate results.
Traditional database generators (e.g., [7, 4, 17, 1, 2]) allow constraints to be defined only on the base tables (e.g., a join distribution is defined on the base tables). As a result, a tester cannot specify operator constraints (e.g., the output cardinality of a join) in an intrinsic way. QAGen allows a user to annotate constraints on each operator and base tables directly. Thus, the users can easily get a meaningful test database for a distinct test case.

Sometimes it would be advantageous to add new kinds of constraints to an operator in addition to the cardinality constraint during testing. For instance, the aggregation (group-by) operator may not only need to control the output size (i.e., the number of groups), but may also need to control how to distribute the input to the predefined output groups (i.e., some groups have more tuples while others have fewer). QAGen is designed to be extensible in order to incorporate new operator constraints easily.

1.3 Roadmap

The remainder of this paper is organized as follows: Section 2 gives an overview of QAGen. Section 3, 4, 5 and Section 6 show the details of QAGen. Section 7 presents the experimental results. Section 8 discusses related work. Section 9 contains conclusions and suggestions for future work.
2 QAGen Overview

The data generation process of QAGen consists of two phases: (1) the symbolic query processing (SQP) phase, and (2) the data instantiation phase. The goal of the symbolic query processing phase is to capture the user-defined constraints on the query into the target database. To process a query without concrete data, QAGen integrates the concept of symbolic execution [19] from software engineering into traditional query processing. Symbolic execution is a well known program verification technique, which represents values of program variables with symbolic values instead of concrete data and manipulates expressions based on those symbolic values. Borrowing this concept, QAGen first instantiates a database which contains a set of symbols instead of concrete data (thus the generated database in this phase is called a symbolic database). Figure 5 shows an example of a symbolic database with three symbolic relations $R$, $S$ and $T$. Essentially, a symbolic relation is just a normal relational table which consists of a set of symbolic tuples. Inside each symbolic tuple, the values are represented by symbols rather than by concrete values. For example, the symbol $a_1$ in the symbolic relation $R$ in Figure 5 represents any value under the domain of attribute $a$. The formal definition of these symbolic database related terms will be given in Section 5. For the moment, let us just treat the symbolic relations as normal relations and imagine the symbols as variables. Since the symbolic database is a generalization of relational databases and provides an abstract representation for concrete data, this gives room to QAGen to control the output of each operator of the query.

The symbolic query processing phase leverages the concept of traditional query processing. First, the input query is analyzed by a query analyzer. Then, the user specifies her desired requirements on the operators of the query tree. Afterwards, the input query is executed by a symbolic query engine just like in traditional query processing; i.e., each operator is implemented as an iterator, and the data flows from the base tables up to the root of the query tree [15]. However, unlike in traditional query processing, the symbolic execution of operators deals with symbolic data rather than concrete data. Each operator manipulates the input symbolic data according to the operator’s semantics and the user-defined constraints, and incrementally imposes the constraints defined on the operators to the symbolic database. After this phase, the symbolic database is then a query-aware database that captures all requirements defined by the test case of the input query (but without concrete data).

The data instantiation phase follows the symbolic query processing phase. This phase reads in tuples from the symbolic database that are prepared by the symbolic query processing phase and instantiates the symbols in the tuples by a constraint solver. The instantiated tuples are then inserted into the target database.

To allow a user to define different test cases for the same query, the input query of QAGen is in the form of a relational algebra expression. For example, if the input query is a 2-way join query $(\sigma_{\text{age}>p_1} \text{Customer} \bowtie \text{Orders}) \bowtie \text{Lineitem}$, then the user can specify a distribution (e.g., a Zipf distribution) between the lineitems and the orders that join with customers with an age greater than $p_1$. On the other hand, if the input query is $(\text{Orders} \bowtie \text{Lineitem}) \bowtie \sigma_{\text{age}>p_1} \text{Customer}$, then the user can specify the join
distribution between all orders and all lineitems.

Figure 3 shows the general architecture of QAGen. It consists of the following components: a Query Analyzer, a Symbolic Query Engine, a Symbolic Database and a Data Instantiator.

2.1 Query Analyzer

At the beginning of the symbolic query processing phase, QAGen first takes a parametric query $Q_P$, the database schema $M$ as input. The query $Q_P$ is then analyzed by the query analyzer component in QAGen. The query analyzer has two functionalities:

1) Correct knob selections. It analyzes the input query and determines which knob(s) are available for each operator. A knob can be regarded as a parameter of an operator that controls the output. A basic knob that is offered by QAGen is the output cardinality constraint². This knob allows a user to control the output size of an operator. However, whether such a knob is applicable depends on the operator and its input characteristics.

Figure 4 shows the knobs of each operator offered by QAGen under different cases. As an example, for a simple aggregation query $\text{SELECT MAX}(a) \text{ FROM } R$, the cardinality constraint knob should not be available for the aggregation operator ($\chi$), because the output cardinality is always one if $R$ is not empty or is zero if $R$ is empty (Figure 4 case (f)). As another example, the available knob(s) of an equi-join ($\bowtie$) that joins two relations (with foreign key relationship) depend on whether the input is pre-grouped or not on the join keys. If the input is pre-grouped, the equi-join can only offer the output cardinality as single knob (Figure 4 case (d)). If the input is not pre-grouped, the...

²The output cardinality of an operator can be specified as an absolute value or as a selectivity. Essentially they are equivalent.
user is allowed to tune the join distribution as well (Figure 4 case (c)). The input of an operator is pre-grouped on an attribute \( a \) if and only if there is at least one symbol which is not distinct in \( a \) (Section 3 will give formal definitions of all input characteristics). Consider a 2-way join query \((R \Join_{b=c} S) \Join_{a=e} T\) on the three symbolic relations \( R, S, \) and \( T \) in Figure 5. When the symbolic relation \( R \) first joins with the symbolic relation \( S \) on attribute \( b \) and \( c \), it is possible to specify the join distribution such as joining the first tuple \( t_1 \) of \( R \) with the first three tuples of \( S \) (i.e., \( t_3, t_4, t_5 \)); and the last tuple \( t_2 \) of \( R \) joins with the last tuple \( t_6 \) of \( S \) (kind of like Zipf distribution [25]). However, after the first join, the intermediate join result of \( R \Join S \) is pre-grouped on attribute \( a, b \), and \( c \) (e.g., the symbol \( a_1 \) is not distinct on the attribute \( a \) in the join result). Therefore, if this intermediate join result further joins with the symbolic relation \( T \) on attribute \( a \) and \( e \), then the distribution cannot be freely specified by a user, because if the first tuple \( t_{11} \) of \( T \) joins with the first tuple \( t_7 \) of the intermediate results, this implies that \( e_1 = a_1 \) and thus \( t_{11} \) must join with \( t_8 \) and \( t_9 \) as well.

The above example shows that it is necessary to analyze the query in order to offer the right knobs to the users. For this purpose, the query analyzer parses the input query in a bottom-up manner (i.e., starting from the input schema \( M \)) and incrementally pre-computes the output characteristics of each operator (e.g., annotates an attribute of the output of an operator as pre-grouped if necessary). In the example, the query analyzer annotates the attributes \( a, b, \) and \( c \) as pre-grouped in the output of \( R \Join S \). Based on this information, the query analyzer disables the join distribution knob on the next equi-join that joins with \( T \).

This step is fairly simple and the query analyzer can do that without analyzing the input data of each operator. Thus, the query analyzer essentially annotates the appropriate knob(s) to each operator according to Figure 4. As a result, the output of the query analyzer is an annotated query tree with the appropriate knob(s) on each operator. Section 3 will present the details of this step.

(2) Assign physical implementations to operators. As shown above, different knobs are available under different input characteristics. In general, different (combinations of) knobs of the same operator need separate implementation algorithms. Moreover, even for the same (combination of) knobs of the same operator, different implementation algorithms are conceivable (this is akin to traditional query processing where an equi-join operation can be done by hash-join or sort-merge join). Consequently, the other function of the query analyzer is to assign the correct (knob-supported) implemen-
ation to an operator. As a result, the output of the query analyzer is a knob-annotated query execution plan. Section 5 will present the implementation algorithms for each (knob-supported) operator in QAGen.

2.2 Symbolic Query Engine and Database

The symbolic query engine of QAGen is the heart of the symbolic query processing phase and it is similar to a normal query engine. It interprets the knob-annotated query execution plan given by the query analyzer. Symbolic query execution is also based on the iterator model [15]. That is, an operator reads in symbolic tuples from its child operator(s) one-by-one, processes each tuple, and returns the resulting tuple to the parent operator.

Before the symbolic query engine starts execution, the user can specify the knob value(s) on the available knob(s) of each operator in the knob-annotated execution plan. It is fine for a user to fill up values for some but not all knobs. In this case, the symbolic query engine will evaluate those operators by using default knob values.

Similar to traditional query processing, most of the operators in symbolic query processing can be processed in a pipelining mode but some cannot. For example, the aggregation operator with multiple group-by attributes needs to be implemented as a blocking operator, and under a special case, the equi-join operator is also a blocking operator. In these cases, the symbolic query engine materializes the intermediate results into the symbolic database if necessary. In symbolic query processing, a table in a query tree is regarded as an operator. During its open() method, the table operator initializes a symbolic relation based on the input schema $M$ and the user-defined constraints (e.g., table sizes) on the base tables.

An operator evaluates the input tuples according to its own semantics. On the one hand, it imposes additional constraints to each input tuple in order to reflect the constraints defined on the operator on the data level. On the other hand, it controls its output to its parent operator so that the parent operator can work on the right tuples. As a simple example, assume the input query is a simple selection query $\sigma_{a \geq p_1} R$ on the symbolic relation $R$ in Figure 5 and the user specifies the output cardinality as 1 tuple. Then, if the getNext() method of the selection operator iterator is invoked by its parent operator, the selection operator reads in tuple $t_1$ from $R$, annotates a positive constraint $[a_1 \geq p_1]$ to the symbol $a_1$ and returns the tuple $\langle a_1, b_1 \rangle$ to its parent. When the getNext() method of the selection operator iterator is invoked the second time, the selection operator reads in the next tuple $t_2$ from $R$, annotates a negative constraint $[a_2 < p_1]$ to the symbol $a_2$. However, this time it does not return this tuple to its parent, because the cardinality constraint (1 tuple) is already fulfilled.

It is worth noting that sometimes a user may specify some contradicting knob values on the knob-annotated query tree given by the query analyzer. For instance, a user may specify the output cardinality of the selection in the above example as 10 tuples even if she specified the table $R$ to have two tuples only. During runtime, when an operator cannot fulfill its output requirements even though it consumed all the input tuples, then the symbolic query engine stops processing and returns the corresponding error.
message to the user. By referring to the error message, the user can re-tune the knob values of the corresponding operator(s).

2.3 Data Instantiator

The data instantiation phase starts after the symbolic query engine of QAGen has finished processing. The data instantiator reads in the symbolic tuples from the symbolic database and instantiates the symbols inside each symbolic tuple by a constraint solver. In QAGen, we treat the constraint solver as an external black box component where it takes a constraint formula (in propositional logic) as input and returns a possible instantiation on each variable as output. For example, if the input constraint formula is $40 < a_1 + b_1 < 100$, then the constraint solver may return $a_1 = 55, b_1 = 11$ as output (or any other possible instantiation). Once the data instantiator has collected all the concrete values for a symbolic tuple, it inserts a corresponding tuple (with concrete values) into the target database.

3 Query Analyzer

The query analyzer has two functionalities: (1) the correct knob selection for the operators of a given relational algebra expression, and (2) the assignment of the physical implementation to each operator of the relational algebra expression. QAGen currently supports only one physical implementation for each possible combination of knobs per relational algebra operator. As a result, (2) is a straightforward job and we do not cover it for brevity. This section focuses on (1), which describes how to analyze the query and determine the available knob(s) for each operator in the input query.

First, the query analyzer parses the input SQL query into a query tree which consists of operators of the relational algebra. This parsing is carried out in exactly the same way as in a traditional SQL processor.

Next, the query analyzer determines the input characteristics of each operator of the relational algebra expression in order to decide what kind of knobs are available for each operator. In symbolic query processing, there are four types of input characteristics: pre-grouped, not pre-grouped, tree-structure, and graph-structure. Let $A$ be the set of attributes of the input of an operator. The input characteristic definitions are as follows:

Definition. **Pre-grouped** / **Not pre-grouped**: The input of an operator is **not pre-grouped** with respect to an attribute $a \in A$, iff there is a functional dependency $a \rightarrow \{ A - a \}$ (which means that $a$ is distinct) holds in the input. Otherwise, the input of the operator is **pre-grouped** with respect to the attribute $a$.

Definition. **Tree-structure** / **Graph-structure**: A set of attributes $A' \subset A$ of the input of an operator is in **tree-structure**, iff either the functional dependency $a_i \rightarrow a_j$ or $a_j \rightarrow a_i$ holds in the input of the operator (where $a_i, a_j \in A'$ and $a_i \neq a_j$). Otherwise, the set of attributes $A' \subset A$ of the input of the operator is in **graph-structure**.

10
Depending on the input characteristics of each operator in the query tree, the query analyzer annotates the correct knob(s) according to Figure 4. To determine the input characteristics, the query analyzer computes the set of functional dependencies hold on each intermediate result of the input query in a bottom-up fashion. Computing functional dependencies of a relational algebra expression has been studied by [20]. In our own work on reverse query processing [5, 6], we presented how to compute the integrity constraints and dependencies for SQL queries in detail. The followings are extracted from [6] and show how to compute the set of functional dependencies hold on the intermediate result of an SQL query.

Let $F_{in}$ be the set of functional dependencies hold in the input of an unary operator, and let $F_{left}$ and $F_{right}$ be the set of functional dependencies hold in the inputs of a binary operator. Furthermore, for a functional dependency $f \in F$, let $f.A_{left}$ denotes the attributes on the left-hand-side of $f$ and let $f.A_{right}$ denotes the attributes on the right-hand-side of $f$.

The following shows how to compute the functional dependencies $F_{out}$ for an SQL operator:

- **Selection.** $F_{out} = F_{in}$.
- **Projection.** $F_{out} = \text{cleanFD}(F_{in}, A_{\pi})$. $A_{\pi}$ is the set of attributes remain after the projection. The function $\text{cleanFD}(F_{in}, A_{\pi})$ removes the functional dependencies that do not participate in the output after the projection. Figure 6 shows the pseudo-code of the $\text{cleanFD}$ function.
- **Join.** $F_{out} = \text{closure}(F_{left} \cup F_{right} \cup \text{createFD}(p))$. Figure 7 shows the $\text{createFD}$ function and $p$ is the join predicate. The $\text{createFD}$ function takes a predicate $p$ as input and outputs a set of derivable functional dependencies. The function deals with arbitrary predicates by transforming the given predicate into conjunctive normal form (line 3 in Figure 7). The conjunctive normal form of a predicate consists of one or more conjuncts, each of which is a disjunction ($\lor$) of one or more literals (simple predicates without boolean operator). Afterwards, each conjunct is processed separately (line 4 in Figure 7). In case that the conjunct only consists of a simple predicate expressing the equality, it is transformed into a set of functional dependencies (line 6 to 10). The outmost function $\text{closure}$ is a function to compute the transitive closure of any given set of functional dependencies. It can be referred to any textbook or [20].
- **Aggregation.** $F_{out} = \text{closure}(\text{cleanFD}(F_{in}, A_{gr}) \cup \{A_{gr} \rightarrow A_{agg}\})$. $A_{gr}$ denotes the set of GROUP-BY attributes and $A_{agg}$ denotes the attribute representing the aggregation function.
- **Union.** $F_{out} = F_{left} \cap F_{right}$.
- **Minus.** $F_{out} = F_{left}$.
- **Intersection.** $F_{out} = F_{left} \cap F_{right}$.
**cleanFD** (Functional dependencies \( F_{in} \), Attributes \( A \))

1. \( F_{out} = \emptyset \)
2. FOREACH \( f \) in \( F_{in} \)
3. IF (\( f.A_{left} \cap A \neq \emptyset \))
4. \( f.A'_{right} = f.A_{right} \cap A \)
5. \( F_{out} = F_{out} \cup (f.A_{left} \rightarrow f.A'_{right}) \)
6. END IF
7. END FOR
8. RETURN \( F_{out} \)

**createFD** (Predicate \( p \))

1. \( F_{out} = \emptyset \)
2. // transform \( p \) to conjunctive normal form
3. \( p' = \text{transformToCNF}(p) // p' = p_1 \land \ldots \land p_n \)
4. FOREACH \( p \) in \( p' \)
5. IF (\( p \) is in the form of \( a_i \) op \( a_j \)) // op is a logical operator
6. \( F_{out} = F_{out} \cup \{a_i \rightarrow a_j\} \)
7. \( F_{out} = F_{out} \cup \{a_j \rightarrow a_i\} \)
8. ELSE IF (\( p \) is in the form of \( a_i \) op \( c \)) // \( c \) is a constant
9. \( F_{out} = F_{out} \cup \{\emptyset \rightarrow a_i\} \)
10. END IF
11. END FOR
12. RETURN \( F_{out} \)
Starting from the base tables, the query analyzer computes the set of functional dependencies hold on each intermediate results in a bottom-up fashion. Since the definition of the input characteristics of an operator solely depends on the functional dependencies, the type of knobs available for an operator can be easily determined according to Figure 4.

As we will see in the next section, all symbols in the base tables are distinct initially (see tables R and S in Figure 5 as an example). As a result, the initial set of functional dependencies for the base tables can be determined easily. For example, the base table $R$ in Figure 5 contains two functional dependencies: $\{a\} \rightarrow \{b\}$, and $\{b\} \rightarrow \{a\}$.

In Figure 5, the intermediate result $R \bowtie b=c S$ of the query $(R \bowtie b=c S) \bowtie a=e T$ has three pre-grouped attributes $a$, $b$ and $c$ (where $b = c$) and has one not pre-grouped attribute $d$. It is because:

- Initially, the set of functional dependencies of $R$ is $\{a \rightarrow b, b \rightarrow a\}$ and the set of functional dependencies of $S$ is $\{c \rightarrow d, d \rightarrow c\}$.
- According to the functional dependency calculation rule for joining (see above), the createFD function adds two more functional dependencies $b \rightarrow c$ and $c \rightarrow b$ for the join predicate $b = c$. The final set of functional dependencies of $F_{out}$ of the intermediate join result $R \bowtie b=c S$ is $\{a \rightarrow b, b \rightarrow a, c \rightarrow d, d \rightarrow c, b \rightarrow c, c \rightarrow b\}$.
- Among the set of attributes $A = \{a, b, c, d\}$ in the intermediate result, attribute $d$ functionally determines all attributes in $A$ whereas the others cannot. As a result, according to the definition of pre-grouping, $d$ is not pre-grouped and $a$, $b$, and $c$ are pre-grouped in the intermediate result.

We use another example to illustrate the concept of tree and graph input characteristics. Assume the following table is an intermediate result of a query:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
</tr>
<tr>
<td>a2</td>
<td>b1</td>
<td>c1</td>
<td>d2</td>
</tr>
<tr>
<td>a3</td>
<td>b2</td>
<td>c1</td>
<td>d2</td>
</tr>
<tr>
<td>a4</td>
<td>b3</td>
<td>c2</td>
<td>d1</td>
</tr>
</tbody>
</table>

Assume the following functional dependencies hold on the above intermediate result: $a \rightarrow \{b, c, d\}$, and $b \rightarrow \{c\}$. Following the definitions of tree and graph structure, the attribute set $A = \{a, b, c\}$ is in tree-structure because all attributes are functional dependent on each other. On the other hand, the attribute set $A = \{a, b, d\}$ is in graph-structure because there is no functional dependency between $b$ and $d$ (i.e., neither $\{a\} \rightarrow \{d\}$, nor $\{d\} \rightarrow \{a\}$ holds on the intermediate result).

4 Class of SQL queries considered

In this paper, we consider a subset of SQL queries for the QAGen system. We consider SQL queries with the following SQL operators: selection ($\sigma$), projection ($\pi$), equi-join
(\times), aggregation (\chi), union (\cup), minus (−) and intersection (\cap). The set of comparison operators is restricted to =, \leq, \geq, <, >. In addition to the above class of SQL queries, we assume there are no CHECK constraints defined on the base tables. Furthermore, we assume there are no constraints (including the constraints added by the operators in the query) imposed on the group-by attribute(s) on an aggregation operator, and the number of possible values in the domain of a group-by attribute is greater than the number of tuples to be output.

As shown in the previous section, in many cases, the available knobs for an operator depend on its input characteristics. In this paper, only the equi-join operator and the aggregation operator with single group-by attribute support pre-grouped input. The aggregation operator with multiple group-by attributes also supports pre-grouped input when the set of group-by attributes has a tree-structure.

Nevertheless, the cases supported by QAGen already suffices to cover 13 out of 22 complex TPC-H [4] queries. In general, supporting new operators (e.g., theta join), or adding new knobs (which may depend on new input characteristics) to an operator is straightforward in QAGen. For example, adding a new knob to an operator simply means incorporating the new QAGen implementation of that operator into the symbolic query engine and then updating the query analyzer about the input characteristics that this new knob depends on.

5 Symbolic Query Processing

In this section, we first define the data model of symbolic data and discuss how to physically store the symbolic data. Then we present the algorithms for the operators in symbolic query processing through a running example.

5.1 Symbolic Data Model

5.1.1 Definitions

A symbolic relation consists of a relation schema and a symbolic relation instance. The definition of a relation schema is exactly the same as the classical definition of a relation schema in [11]. Let \( R(a_1:\text{dom}(a_1), \ldots, a_i: \text{dom}(a_i), \ldots, a_n: \text{dom}(a_n)) \) be a relation schema with \( n \) attributes; and for each attribute \( a_i \), let \( \text{dom}(a_i) \) be the domain of the attribute \( a_i \).

A symbolic relation instance is a set of symbolic tuples \( T \). Each symbolic tuple \( t \in T \) is a \( n \)-tuple with \( n \) symbols: \( (s_1, s_2, \ldots, s_n) \). As a shorthand, the symbol \( s_i \) in tuple \( t \) can be referred by \( t.a_i \). A symbol \( s_i \) is associated with a set of predicates \( P_{s_i} \) (where \( P_{s_i} \) can be empty). The value of symbol \( s_i \) represents any one of the values in the domain of attribute \( a_i \) that satisfies all predicates in \( P_{s_i} \). A predicate \( p \in P_{s_i} \) of a symbol \( s_i \) is a propositional formula that involves at least \( s_i \), and zero or more other symbols that appear in different symbolic relation instances. Therefore, a symbol \( s_i \) with its predicates \( P_{s_i} \) can be represented by a conjunction of propositional logic formulas. A
symbolic database is defined as a set of symbolic relations and there is a one-to-many mapping between one symbolic database and many relational databases.

5.1.2 Data Storage

Symbolic databases are a generalization of relational databases and provide an abstract representation of concrete data. Given the close relationship between relational databases and symbolic databases, and the fact of the maturity of relational database technology, it may not pay off to re-invent the wheel and design another physical model for storing symbolic data. QAGen opts to leverage existing relational databases to implement the symbolic database concept. To that end, a natural idea for storing symbolic data is to store the data in columns of tables, introduce a user-defined type [3] to describe the columns, and use SQL user-defined functions to implement the symbolic operations. However, symbolic operations (e.g., a join that controls the output size and distribution) are too complex to be implemented by SQL user-defined functions. As a result, we propose to store symbols (and associated predicates) in relational databases by simply using the varchar SQL data type and let the QAGen symbolic query engine operate on a relational database directly. This way, we integrate the power of various access methods brought by the relational database engine into symbolic query processing.

The next interesting question is how to normalize a basic symbolic relation for efficient symbolic query processing. From the definition of a symbol, we know that a symbol may be associated with a set of predicates. For example, the symbol \(a_1\) may have a predicate \([a_1 \geq p_1]\) associated with it. As we will see later, most of the symbolic operations impose some predicates (from now on, we use the term predicate instead of constraints) on the symbols. Therefore, a symbol may associate with many predicates. As a result, QAGen stores the predicates of a symbol in a separate relational table called PTable. Reusing the simple selection query \(\sigma_{a \geq p} R\) from Figure 5 again, the symbolic relation \(R\) can be represented by a normal table in a RDBMS named \(R\) with the schema: \(R(a: \text{varchar}, b: \text{varchar})\) and a table named PTable with the schema: \(\text{PTable(symbol: varchar, predicate: varchar)}\). After the selection on \(R\), the relational representation of the symbolic table \(R\) is:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>symbol</th>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_1)</td>
<td>(b_1)</td>
<td>(a_1)</td>
<td>([a_1 \geq p_1])</td>
</tr>
<tr>
<td>(a_2)</td>
<td>(b_2)</td>
<td>(a_2)</td>
<td>([a_2 &lt; p_1])</td>
</tr>
</tbody>
</table>

Table \(R\) (2 tuples)

Table \(\text{PTable}\) (2 tuples)

5.2 Symbolic Query Evaluation

The major difference between symbolic query execution and traditional query processing is that the input (and thus the output) of each operator is symbolic data but not concrete data. The flexibility of symbolic data allows an operator to control its internal operation and thus its output. Same as traditional processing, an operator is implemented as an iterator. Therefore the interface of an operator is the same as traditional query
Figure 8: Running Example
processing which consists of three methods: open(), getNext() and close(). During query processing, if the operator has problems due to there are some contradicting knob values defined by the user, that operator should return an error message to the user. For example, user specifies the output cardinality of a selection operator is bigger than its input size. For brevity, error detection during symbolic query processing is not discussed in this paper; it is straightforward to implement. In the remainder of this section, we assume that no contradicting knob values are given by the user.

Next, we present the knobs and the algorithms for each operator through a running example. Unless stated otherwise, the following sub-sections only show the details of the getNext() method of each operator. All other aspects (e.g., open() and close()) are straightforward so that they are omitted for brevity. The running example is a 2-way join query which can demonstrate the details of the symbolic execution of selection, equi-join, aggregation and projection. We will also discuss some special cases of these operators. Figure 8a shows the input query tree (with all knobs and their values given). The example is based on the following simplified TPC-H schema (primary keys are underlined):

```
Customer (c_id int, c_acctbal float)
Orders (o_id int, o_date date, o_cid REFERENCE Customer)
Lineitem (l_id int, l_price float, l_oid REFERENCE Orders)
```

### 5.2.1 Symbolic Execution of Table Operator

**Knob:** Table Size (compulsory)

In QAGen, a base table in a query tree is regarded as an operator. During the open() method, it creates a relational table in a RDBMS with the attributes specified on the input schema $M$. According to the designed storage model, all attributes are in the SQL data type varchar. Next, it fills up the table by creating new symbolic tuples until it reaches the defined table size. Each symbol in the newly created tuples is named using the attribute name as prefix and a unique identification number. Therefore, at the beginning of symbolic query processing, each symbol in the base table should be unique. Figure 8b shows the relational representation of the three symbolic relations Customer, Orders and Lineitem for the running example. The getNext() method of the table operator is the same as the traditional Table-Scan operator that returns a tuple to its parent or returns null (an end-of-result message) if all tuples have been returned. Note that if the same table is used multiple times in the query, then the table operator only creates and fills the base symbolic table once.

Primary keys, unique and not null constraints are enforced already because all symbols are initially unique. Foreign key constraints related to the query are taken care by the join operator directly.

### 5.2.2 Symbolic Execution of Selection Operator

**Knob:** Output Cardinality $c$ (optional; default value = input size)
Let $I$ be the input and $O$ be the output of the selection operator $\sigma$ and let $p$ be the selection predicate. The symbolic execution of the selection operator controls the size of the output as $c$. Depending on the input characteristics, the problem hardness and solutions are completely different. Generally, there are two different cases.

**Case 1: Input is not pre-grouped on the selection attribute(s)** This is case (a) in Figure 4 and the selections in the running example (Figure 8a operator (ii) and (vi)) are in this case. This implementation is chosen by the query analyzer when the input is not pre-grouped on the selection attribute(s) and it is the usual case for most queries. In this case, the selection operator controls the output by:

1. During its getNext() method, read in a tuple $t$ by invoking getNext() on its child operator and process with [Positive Tuple Annotation] if the output cardinality has not reached $c$. Else proceed to [Negative Tuple Post Processing] and then return null to its parent.

2. [Positive Tuple Annotation] If the output cardinality has not reached $c$, then (a) for each symbol $s$ in $t$ that participates in the selection predicate $p$, insert a corresponding tuple $\langle s, p \rangle$ to the PTable and (b) return this tuple $t$ to its parent.

3. [Negative Tuple Post Processing] However, if the output cardinality has reached $c$, then fetch all the remaining tuples $I^\sim$ from input $I$. For each symbol $s$ of tuple $t$ in $I^\sim$ that participates in the selection predicate $p$, insert a corresponding tuple $\langle s, \neg p \rangle$ to the PTable, and repeat this step until calling getNext() on its child has no more tuples (returns null).

<table>
<thead>
<tr>
<th>symbol</th>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_id1</td>
<td>c_acctbal1</td>
</tr>
<tr>
<td>c_id2</td>
<td>c_acctbal2</td>
</tr>
</tbody>
</table>

(i) Output of $\sigma$; 2 tuples

<table>
<thead>
<tr>
<th>symbol</th>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_acctbal1</td>
<td>$[c_{acctbal1} \geq p_1]$</td>
</tr>
<tr>
<td>c_acctbal2</td>
<td>$[c_{acctbal2} \geq p_1]$</td>
</tr>
<tr>
<td>c_acctbal3</td>
<td>$[c_{acctbal3} &lt; p_1]$</td>
</tr>
<tr>
<td>c_acctbal4</td>
<td>$[c_{acctbal4} &lt; p_1]$</td>
</tr>
</tbody>
</table>

(ii) PTable

**Table A. After selection**

Each getNext() call on the selection operator returns a positive tuple that satisfies the selection predicate $p$ to its parent until the output cardinality has been reached. Moreover, to ensure all negative tuples (i.e., tuples got from the child operator after the output cardinality has been reached) would not get some instantiated values later in the data instantiation phase that ends up passing the selection predicate, the selection operator associates the negation of predicate $p$ to those negative tuples. In the running example, the attribute $c_{acctbal}$ in the selection predicate $[c_{acctbal} \geq p_1]$ of operator (ii) is not pre-grouped, because the data comes directly from the base Customer table. Since the output cardinality $c$ of the selection operator is 2, the selection operator associates the positive predicate $[c_{acctbal} \geq p_1]$ to the symbol $c_{acctbal1}$ and $c_{acctbal2}$ of the first two input tuples and associates the negated predicate $[c_{acctbal} < p_1]$ to the symbol $c_{acctbal3}$ and $c_{acctbal4}$ of the rest of the tuples. Table A(i) shows the
output of the selection operator and Table A(ii) shows the content of the PTable after the selection.

Case 2: Input is pre-grouped on the selection attribute(s)
This is a special case of selection, and only happens when a selection is on top of a join and there is an attribute \( a \) in the selection predicate \( p \) pre-grouped. In this paper, we do not consider this type of queries. Nevertheless, this case rarely happens because most selection operators can be pushed down by the user who gives the input.

5.2.3 Symbolic Execution of Equi-Join Operator

<table>
<thead>
<tr>
<th>Knob:</th>
<th>Output Cardinality ( c ) (optional; default value = size of the non-distinct input)</th>
</tr>
</thead>
</table>

Let \( R \) and \( S \) be the inputs, \( O \) be the output, and \( p \) be the simple equality predicate \( j = k \) where \( j \) is the (non-pre-grouped) join attribute on \( R \), and \( k \) is the join attribute on \( S \) that refers to \( j \) by a foreign key relationship. The symbolic execution of the equi-join operator ensures the join result size is \( c \). Again, depending on whether the input is pre-grouped or not, the solutions are different, too.

Case 1: Input is not pre-grouped on the join attribute \( k \).
This is case (c) in Figure 4, where the join attribute \( k \) in the input \( S \) is not pre-grouped. In this case, it is possible to support one more knob on the equi-join operation:

<table>
<thead>
<tr>
<th>Knob:</th>
<th>Join Distribution ( b ) (optional; choices = [Uniform or Zipf]; default = Uniform)</th>
</tr>
</thead>
</table>

The join distribution \( b \) defines how many tuples of input \( S \) join with each individual tuple in input \( R \). For example, if the join distribution is uniform, then each tuple in \( R \) joins with roughly the same number of tuples in \( S \). Both join operators in Figure 8a fall into this case. In this case, the equi-join operator (which supports both output cardinality \( c \) and distribution \( b \)) controls the output by:

1. [Distribution instantiation] During its open() method, instantiate a distribution generator \( D \), with the size of \( R \) as domain (denoted by \( n \)), the output cardinality \( c \) as frequency, and the distribution type \( b \) as input. This distribution generator \( D \) can be the one in [16] or [9] or any statistical packages that generate \( n \) numbers \( m_1, m_2, \ldots, m_n \) which follow Uniform or Zipf [25] distribution with a total frequency of \( c \). The distribution generator \( D \) is an iterator with a getNext() method. For the \( i \)-th call on the getNext() method (\( 0 \leq i \leq n \)), it returns the expected frequency \( m_i \) of the \( i \)-th number under distribution \( b \).

2. During its getNext() call, if the output cardinality has not yet reached \( c \), then (a) check if \( m_i = 0 \) or if \( m_i \) has not yet initialized, if yes, initialize \( m_i \) by calling getNext on \( D \) and get a tuple \( r^+ \) from \( R \) (\( m_i \) is the total number of tuples from \( S \) that should join with \( r^+ \)). (b) Get a tuple \( s^+ \) from \( S \) and decrease \( m_i \) by one. (c) Join tuple \( r^+ \) with \( s^+ \) according to [Positive Tuple Joining] below. (d) Return the joined tuple to its parent. However, during the getNext() call, if the output cardinality has reached \( c \) already, then process [Negative Tuple Joining] below, and return null to its parent.

3. [Positive Tuple Joining] If the output cardinality has not reached \( c \), then (a) for the tuple \( s^+ \), replace the symbol \( s^+.k \), which is the symbol of the join key attribute \( k \) of tuple \( s^+ \), by the symbol \( r^+.j \), which is the symbol of the join key attribute \( j \) of tuple \( r^+ \). After this, the
tuples $r^+$ and the tuple $s^+$ should share exactly the same symbol on their join attributes. Note that the replacement of symbols in this step is done on both the tuples loaded in the memory and the related tuples in base table as well (using an SQL statement like “Update k.BaseTable Set k=r^+.j WHERE k=s^+.k” to update the symbols on the base table where the join attribute $k$ comes from). (b) Perform an equi-join on tuple $r^+$ and $s^+$.

4. [Negative Tuple Joining] However, if the output cardinality has reached $c$, then fetch all the remaining tuples $S^-$ from input $S$. For each tuple $s^-$ in $S^-$, randomly look up a symbol $j^-$ on the join key $j$ in the set minus between the base table where the join attribute $j$ originates from and $R$ (using an SQL statement with the MINUS keyword), replace $s^- . k$ with the symbol $j^-$. This replacement is done on the base tables only because these tuples are not returned to the parent.

In the running example (Figure 8), after the selection on table Customer (operator ii), the next operator is a join between the selection output (Table A(i) in Section 5.2.2) and table Orders. The output cardinality $c$ of that join (operator iii) is 4 and the join distribution is uniform. Since the input of the join on the join key $o_{cid}$ is not pre-grouped, the query analyzer uses the algorithm above to perform the equi-join. First, the distribution generator $D$ generates 2 numbers (which is the size of the input $R$), with total frequency of 4 (output cardinality), and distribution as uniform. Assume $D$ returns a sequence: {2, 2}. This means that the first customer $c_{id}1$ should take 2 orders ($o_{id}1$ and $o_{id}2$) and the second customer $c_{id}2$ should also take 2 orders ($o_{id}3$ and $o_{id}4$). As a result, the symbols $o_{cid}1$ and $o_{cid}2$ from the Orders table should be replaced by $c_{id}1$ and the symbols $o_{cid}3$ and $o_{cid}4$ from the Orders table should be replaced by $c_{id}2$ (Step 3 above). In order to fulfill the foreign key constraint on those tuples which do not join, Step 4 above (Negative Tuple Joining) replaces $o_{cid}5$ and $o_{cid}6$ by customers that did not pass through the selection filter (i.e., customer $c_{id}3$ and $c_{id}4$) randomly. Table B(i) below shows the output of the join and Table B(ii) shows the updated Orders table (join keys are **bold**).

<table>
<thead>
<tr>
<th>c_acctbal</th>
<th>o_id</th>
<th>o_date</th>
<th>c_id=O_cid</th>
<th>o_id</th>
<th>o_date</th>
<th>o_cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_acctbal1</td>
<td>o_id1</td>
<td>o_date1</td>
<td>c_id1</td>
<td>o_id1</td>
<td>o_date1</td>
<td>c_id1</td>
</tr>
<tr>
<td>c_acctbal1</td>
<td>o_id2</td>
<td>o_date2</td>
<td>c_id1</td>
<td>o_id2</td>
<td>o_date2</td>
<td>c_id1</td>
</tr>
<tr>
<td>c_acctbal2</td>
<td>o_id3</td>
<td>o_date3</td>
<td>c_id2</td>
<td>o_id3</td>
<td>o_date3</td>
<td>c_id2</td>
</tr>
<tr>
<td>c_acctbal2</td>
<td>o_id4</td>
<td>o_date4</td>
<td>c_id2</td>
<td>o_id4</td>
<td>o_date4</td>
<td>c_id2</td>
</tr>
</tbody>
</table>

(i) Output of (σ(Customer) ∘ Order); 4 tuples

<table>
<thead>
<tr>
<th>o_id</th>
<th>o_date</th>
<th>o_cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>o_id1</td>
<td>o_date1</td>
<td>c_id1</td>
</tr>
<tr>
<td>o_id2</td>
<td>o_date2</td>
<td>c_id1</td>
</tr>
<tr>
<td>o_id3</td>
<td>o_date3</td>
<td>c_id2</td>
</tr>
<tr>
<td>o_id4</td>
<td>o_date4</td>
<td>c_id2</td>
</tr>
</tbody>
</table>

(ii) Orders (4 pos, 2 neg)

Table B. After Joining

After the join operation above, the next operator in the running example is another join between the above join results (Table B(i)) and the base Lineitem table (Figure 8b(iii)) in Zipf distribution. Again, the input of the join on the join key $l_{oid}$ of the Lineitem table is not pre-grouped and thus the above equi-join algorithm is chosen by the query analyzer. Assume that the distribution generator generates a Zipf sequence \{4,2,1,1\} for the four tuples in Table B(i) to join with 8 out of 10 line-items (where 8 is the user-specified output cardinality of this join operation). Therefore it produces the following output (join keys are **bold**):
Finally, note that if the two inputs of an equi-join are base tables (with foreign key constraint), then the output cardinality knob is disabled by the query analyzer. It is because in that case, all tuples from input \( R \) must join with input \( S \) and thus the output cardinality must be same as the size of \( S \).

Case 2: Input is pre-grouped on the join attribute \( k \).

This is a special case of equi-join when the input \( S \) is pre-grouped on the join attribute \( k \). This sometimes happens when a preceding join introduced a distribution on \( k \) as in the example in Figure 5. In the following we show that if the input is pre-grouped on the join attribute \( k \) of an equi-join, then the problem of controlling the output cardinality (even without the join distribution) is reducible to the subset-sum problem:

<table>
<thead>
<tr>
<th>( j )</th>
<th>( k )</th>
<th>( c_1 )</th>
<th>( c_2 )</th>
<th>( c_3 )</th>
<th>( c_4 )</th>
<th>( \ldots )</th>
<th>( c_m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j_1 )</td>
<td>( k_1 )</td>
<td>e.g. 5 times</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( j_2 )</td>
<td>( k_2 )</td>
<td>e.g. 4 times</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( j_3 )</td>
<td>( k_3 )</td>
<td>e.g. 2 times</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( k_4 )</td>
<td>e.g. 1 times</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( j_l )</td>
<td>( k_m )</td>
<td>( c_m ) times</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The subset-sum problem [14] takes as input an integer sum \( c \) and a set of integers \( C = \{c_1, c_2, \ldots, c_m\} \), and outputs whether there exists a subset \( C^+ \subseteq C \) such that \( \sum_{c_i \in C^+} c_i = c \). Consider the tables \( R \) and \( S \) in the figure on the right hand side, which are the inputs of such a join. Table \( R \) has one attribute \( j \) with \( l \) tuples all using distinct symbolic values \( ji \) (\( i \leq l \)). Table \( S \) also defines only one attribute \( k \) and has in total \( \sum c_i \) rows. The rows in \( S \) are clustered in \( m \) groups, where the \( i \)-th group has exactly \( c_i \) tuples using the same symbolic value \( ki \) (\( i \leq m \)). We now search for a subset of those \( m \) groups in \( S \) which join with arbitrary tuples in \( R \) so that the output has the size \( c \). Assume, we find such a subset, i.e., the symbolic values of those groups which result in the output with size \( c \). The groups returned by such a search induce a solution for the original subset-sum problem.

The subset-sum problem is a weakly \( NP \)-complete problem and there exists a pseudopolynomial algorithm which uses dynamic programming to solve it [14]. The complexity of that dynamic programming algorithm is \( O(\min(c, \sum c_i) \times m) \), where \( c \) is...
the desired output cardinality, $c_i$ is the size of the $i$-th group in $S$, and $m$ the number of different groups in $S$. If one of the input parameters is in binary (e.g., $m$ is encoded as a $n$-bit digit and thus has the size $2^n$), then the running time would be exponential in the input size. Fortunately, this means the special case of the equi-join operator (with pre-grouped input on the attribute $k$) is solvable in polynomial time because all the input parameters are given in unary. Since this case happens more often, we propose a dynamic programming version of equi-join for this special case.

The equi-join algorithm uses dynamic programming to compute a subset of the pre-groups with a total count that matches the output cardinality. This is a blocking operator because it needs to read all the input from $S$ first (for dynamic programming to solve the subset-sum problem). For memory reason, all the input tuples from $S$ are materialized in the symbolic database. One optimization for this case is that if $c$ is equal to the input size of $S$, then all tuples of $S$ must be joined with $R$ and the dynamic programming function can be skipped even the data is pre-grouped.

We reuse the figure above to illustrate the algorithm. Assume the join is on Table $R$ and Table $S$ and the join predicate is $j = k$. Assume Table $R$ has three tuples $(j1, j2, j3)$, and Table $S$ has 12 tuples which are clustered into 4 groups with symbol $k1, k2, k3, k4$ respectively. Furthermore, assume the join on $R$ and $S$ is specified with an output cardinality as $c = 7$. The dynamic programming equi-join controls the output as follows:

1. [Dynamic programming] During its open() method, (a) materialize the input $S$ of the join operator. (b) Extract the pre-group size (e.g., $c_1 = 5$, $c_2 = 4$, $c_3 = 2$, $c_4 = 1$) of each symbol $ki$ by executing “Select Count(k) From S Group By k Order By Count(k) Desc” on the materialized input. (c) Invoke a dynamic programming (dp) function with the pre-group sizes and the output cardinality (e.g., $c = 7$) as input. The dp function (omitted here because of space) finds a subset of symbols $K^+$ in $S$ which results in the desired total output cardinality (e.g., $K^+ = \{k1, k3\}$ because $c_1 + c_3 = 5 + 2 = 7 = c$). If the dp function cannot find any solution, stop processing and report this problem to the user.

2. [Positive Tuple Joining] During getNext(), (a) for each symbol $ki$ in $K^+$, read all tuples $S^+$ from the materialized input of $S$ which have $ki$ as the value of attribute $k$. (b) Afterwards, call getNext() on $R$ once and get a tuple $r$, join all tuples in $S^+$ with $r$ by replacing the join key symbols in $S^+$ with the join key symbols in $r$. For example, the first five $k1$ symbols in $S$ are replaced with $j1$ and the two $k3$ symbols in $S$ are replaced with $j2$ (again, these replacements are done on symbols loaded in the memory and the changes are propagated to the base tables of where $j$ and $k$ originate from). (c) Return the joined tuples to the parent.

3. [Negative Tuple Joining] This step is the same as the Negative Tuple Joining step in the simple case (Section 5.2.3 case 1) that joins the negative tuples in input $R$ with the negative tuples in input $S$.

### 5.2.4 Symbolic Execution of Aggregation Operator

| Knob: Output Cardinality $c$ (optional; default value = input size) |

Let $I$ be the input and $O$ be the output of the aggregation operator and $f$ be the
aggregation function. The symbolic execution of the aggregation operator controls the size of the output as $c$.  

5.2.4.1 Simple Aggregation This is the simplest case of aggregation where there is no grouping (i.e., no GROUP-BY keyword) defined on the query. In this case, the query analyzer disables the output cardinality knob because the output cardinality is either 1 (not-empty input) or 0 (empty input). In SQL, there are five aggregation functions: SUM, MIN, MAX, AVG, COUNT. For simple aggregation, the solutions are very similar for both pre-grouped or non-pre-grouped input on the attribute(s) in $f$. The following shows the case of non-pre-grouped input:

Let $expr$ be the expression in the aggregation function $f$ which consists of at least a non-empty set of symbol $S$ in $expr$ and let the size of the input $I$ be $n$.

1. **SUM($expr$).** During its getNext() method, (a) the aggregation operator consumes all $n$ tuples from $I$, (b) for each symbol $s$ in $S$, add a tuple $\langle s, [aggsum = expr_1 + expr_2 + \ldots + expr_n]\rangle$ to the PTable, where $expr_i$ is the corresponding expression on the $i$-th input tuple; and (c) return the symbolic tuple ($aggsum$) as output. As an example, assume there is aggregation function SUM(l_price) on top of the join result in Table C(i) of the previous section. Then, this operator returns one tuple $\langle aggsum \rangle$ to its parent and adds 8 tuples (e.g., the 2nd inserted tuple is $\langle l\_price2, [aggsum = l\_price1 + \ldots + l\_price8]\rangle$) to the PTable.

In fact, the above is a base case only. Instead, the aggregation operator optimizes the number and the size of the above predicates by inserting only one tuple $\langle l\_price1, [aggsum = l\_price1 \times 8]\rangle$ to the PTable and replacing the symbols $l\_price2, \ldots, l\_price8$ by the symbol $l\_price1$ on the base table. One reason for doing that is the size of the input may be very big, if that is the case, the extremely long predicate may exceed the SQL varchar size upper bound. Another reason is to insert fewer tuples in the PTable. And the most important reason is that the cost of of a constraint solver call is exponential to the size of the input formula in the worst case. Therefore, this optimization reduces the time of the later data instantiation phase. However, there is a trade-off: for each input tuple, the operator has to update the corresponding symbol in the base table where this symbol originates from.

2. **MIN($expr$).** The MIN aggregation operator also uses similar predicate optimization as SUM aggregation. During its getNext() method, (a) it regards the first expression $expr_1$ as the minimum value and returns $\langle expr_1 \rangle$ as output; and (b) replaces the expression $expr_1$ in the remaining tuples (where $2 < i \leq n$) by the second expression $expr_2$ and inserts two tuples $\langle expr_1, [expr_1 < expr_2] \rangle$ and $\langle expr_2, [expr_1 < expr_2] \rangle$ to the PTable.

As an example, assume there is aggregation function MIN(l_price) on top of the join result in Table C(i). Then, this operator returns $\langle l\_price1 \rangle$ as output and inserts 2 tuples to the PTable: $\langle l\_price1, [l\_price1 < l\_price2] \rangle$ and $\langle l\_price2, [l\_price1 < l\_price2] \rangle$ to the PTable. Moreover, $l\_price3, l\_price4, \ldots, l\_price8$ are replaced by $l\_price2$ on the base table.

3. **MAX($expr$).** During its getNext() method, (a) it regards the first expression $expr_1$ as the maximum value and returns $\langle expr_1 \rangle$ as output; and (b) replaces the expression $expr_1$ in the remaining tuples (where $2 < i \leq n$) by the second expression $expr_2$ and inserts two tuples $\langle expr_1, [expr_1 > expr_2] \rangle$ and $\langle expr_2, [expr_1 > expr_2] \rangle$ to the PTable.

4. **AVG($expr$).** It is the same as handling the SUM aggregation (with optimization) as described above; except it does not insert the two PTable entries.
5. COUNT(expr). The aggregation operator handles the COUNT aggregation function nearly as same as traditional query processing. During its getNext() method, (a) it counts the number of input tuples, \( n \), (b) add a tuple \( \langle \text{agccount} = n \rangle \), and (c) returns a symbolic tuple \( \langle \text{agccount} \rangle \) as output.

In general, combinations of different aggregation functions in one operator (e.g. \( \text{MIN}(\text{expr}_1) + \text{MAX}(\text{expr}_2) \)) need different but similar solutions. Their solutions are straightforward and we do not cover them here.

5.2.4.2 Single GROUP-BY Attribute When the aggregation operator has one group-by attribute, the output cardinality \( c \) defines how to assign the input tuples into \( c \) output groups. Let \( g \) be the single grouping attribute. For all algorithms we assume that \( g \) has no unique constraint in the database schema. Otherwise, the grouping is predefined by the input already and the query analyzer disables all knobs on the aggregation operator for the user. Again, this symbolic operation of aggregation can be divided into two cases:

Case 1: Input is not pre-grouped on the grouping attribute

In addition to the cardinality knob, when the symbols of the grouping attribute \( g \) in the input are not pre-grouped, it is possible to support one more knob:

| Knob: | Group Distribution \( b \) (optional; choices = [Uniform or Zipf]; default = Uniform) |

The group distribution \( b \) defines how to distribute the input tuples into the \( c \) pre-defined output groups. In this case, the aggregation operator controls the output by:

1. [Distribution instantiation] During its open() method, instantiate a distribution generator \( D \), with the size of \( I \) (denoted by \( n \)) as frequency, the output cardinality \( c \) as domain, and the distribution type \( b \) as input. The distribution generator is the same one as the one for doing equi-join (Section 5.2.3). It generates \( c \) numbers \( m_1, m_2, \ldots, m_c \), and the \( i \)-th call on its getNext() method \( (0 \leq i \leq c) \) returns the expected frequency \( m_i \) of the \( i \)-th number under distribution \( b \).

2. During getNext(), call \( D.getNext() \) to get a frequency \( m_i \), fetch \( m_i \) tuples (let them be \( I_i \)) from \( I \) and execute the following steps. If there are no more tuples from its child operator, return null to its parent.

3. [Group assignment] For each tuple \( t \) in \( I_i \), except the first tuple \( t' \) in \( I_i \), replace the symbol \( t.g \), which is the symbol of the grouping attribute \( g \) of tuple \( t \), by the symbol \( t'.g \). \( t'.g \) is the symbol of the grouping attribute \( g \) of the first tuple \( t' \) in the \( i \)-th group. Note that, the replacement of symbols in this step is done on both the tuple loaded in the memory and the related tuples in the base table as well.

4. [Aggregating] Invoke the Simple Aggregation Operator in the previous section (Section 5.2.4.1) with all the symbols participated in the aggregation function in \( I_i \) as input.

5. [Result Returning] Construct a new symbolic tuple \( \langle t'.g, \text{agg}_i \rangle \) to its parent where \( \text{agg}_i \) is the symbolic tuple returned by the Simple Aggregation Operator for the \( i \)-th group. Return the constructed tuple to its parent.
Sometimes, during the open() method, the distribution generator \( D \) may return 0 when the distribution is very skew (e.g., Zipf distribution with high skew factor). In this case, it may happen that an output group does not get any input tuple and the final number of output groups may less than the output cardinality requirement. There are several ways to handle this case. One way is to regard this as an runtime error which let the user know that she should not specify such a highly skew distribution when she asks for many output groups. Another way is to adjust the distribution generator \( D \) such that it first assigns one tuple to each output group (which consumes \( c \) tuples), and then it starts assigning the rest \( n - c \) tuples according to the distribution generation algorithm. This way, it ensures the cardinality requirement is fulfilled but the final distribution may not strictly adhere to the original distribution. In this paper, we assume the user does not specify any contradicting requirements, therefore QAGen uses the first approach.

Case 2: Input is pre-grouped on the grouping attribute

When the input on the grouping attribute is pre-grouped, it is understandable that this operation does not support the group distribution knob as in the above case. But if the input is pre-grouped on the grouping attribute and the output cardinality is the only specified knob, it is not a hard problem.

<table>
<thead>
<tr>
<th>( o_date )</th>
<th>SUM(l_price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>o_date1</td>
<td>aggsum_1</td>
</tr>
<tr>
<td>o_date2</td>
<td>aggsum_2</td>
</tr>
</tbody>
</table>

(i) Output of \( \chi \) (2 tuples)

<table>
<thead>
<tr>
<th>symbol</th>
<th>predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_acctbal1</td>
<td>( c_acctbal1 \geq p_i )</td>
</tr>
<tr>
<td>c_acctbal2</td>
<td>( c_acctbal2 \geq p_i )</td>
</tr>
<tr>
<td>c_acctbal3</td>
<td>( c_acctbal3 &lt; p_i )</td>
</tr>
<tr>
<td>c_acctbal4</td>
<td>( c_acctbal4 &lt; p_i )</td>
</tr>
<tr>
<td>l_price1</td>
<td>( \text{aggsum_1 = 5 \times l_price1} )</td>
</tr>
<tr>
<td>l_price5</td>
<td>( \text{aggsum_2 = 3 \times l_price5} )</td>
</tr>
</tbody>
</table>

(ii) PTable

Table D. After Aggregation

The aggregation operator (\( v \)) in the running example (Figure 8) falls into this case. Referring to Table C(i), which is the input of the aggregation operator in the example. The grouping attribute in the example is \( o\_date \), after several joins, the data in \( o\_date \) is pre-grouped into 4 pre-groups (\( o\_date1 \times 4; o\_date2 \times 2; o\_date3 \times 1; o\_date4 \times 1 \)). In this case, the aggregation operator controls the output by assigning tuples from the same pre-group to the same output group and each pre-group is assigned into \( c \) output groups in a round-robin fashion. In the example, the output cardinality of the aggregation operator is 2. The aggregation operator assigns the first pre-group (with \( o\_date1 \)) which includes 4 tuples into the first output group. Then the second pre-group (with \( o\_date2 \)) which includes 2 tuples to the second output group. When the third pre-group (with \( o\_date3 \)) which includes 1 tuple is being assigned to the first output group (because of round-robin), the aggregation operator replaces \( o\_date3 \) with \( o\_date1 \) in order to put the 5 tuples into the same group. Similarly, the aggregation
operator replaces \( o_{-}date^4 \) from the input tuple with \( o_{-}date^2 \). For the aggregation function, each output group \( g_i \) invokes the Simple Aggregation Operator in Section 5.2.4.1 with all the symbols participated in the aggregation function as input, and gets a new symbol \( agg_{g_i} \) as output. Finally, for each group, the operator constructs a new symbolic tuple \( \langle g_i, agg_{g_i} \rangle \) and returns it to the parent. Table D(i) shows the output of the aggregation operator, and Table D(ii) shows the updated PTable after the aggregation in the running example. Furthermore, since the aggregation involves the attribute \( o_{-}date \) and \( l_{-}price \), the Orders table and the Lineitem table are also updated (Figure 8c shows the updated tables).

5.2.4.3 HAVING and Single GROUP-BY Attribute  In most cases, dealing with a HAVING clause is the same as having a selection operator on top of the aggregation result.

<table>
<thead>
<tr>
<th>( o_{-}date )</th>
<th>( SUM(l_price) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o_{-}date^1 )</td>
<td>( aggsum^1 )</td>
</tr>
</tbody>
</table>

Table E. Output of HAVING clause (1 tuple)

Figure 8c shows the PTable content after the HAVING clause. It imposes two more constraints: \( [aggsum^1 \geq p^2] \) which is the positive tuple and \( [aggsum^2 < p^2] \) which is the negative tuple, and it returns Table E to the parent.

Special case: Parameters controlling the number of tuples of a group

There is a special case for the aggregation operator together with the HAVING clause. When there are more than one parameter in the query which influences the number of tuples of each output group implicitly, it is necessary to ask the user to define the count of each output group explicitly. The following is an example:

```sql
SELECT o_{-}date, SUM(l_{-}price)
FROM Orders, Lineitem
WHERE o_{-}id = l_{-}oid
AND l_{-}price\geq:p_1
GROUP BY o_{-}date
HAVING SUM(l_{-}price)\leq:p_2
```

In this query, the parameter \( p_1 \) and \( p_2 \) implicitly affect the number of tuples that can pass through the HAVING clause. For example, during data instantiation phase, if \( p_1 \) gets a value of 50 and \( p_2 \) gets a value of 200, then only groups with less than 4 tuples is possible to pass through the HAVING clause. In other words, if the user wants to control the output cardinality of the HAVING clause, she has to first control the number of tuples of each group. When the query analyzer detects this case, it prepares the following knobs for the user:

<table>
<thead>
<tr>
<th>Knob:</th>
<th>(a) positive group-count ( gc^+ ) (b) number of positive output groups ( c^+ )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(optional; default: ( gc^+ \geq 1, c^+ = \text{input size} ))</td>
</tr>
</tbody>
</table>
Knob: 
(a) negative group-count $gc^-$ 
(b) number of negative output groups $c^-$
(optional; default: $gc^- \geq 1$, $c^+ = 0$)

The knob $c^+$ defines the number of groups which should pass through the HAVING selection and its coexist knob $gc^+$ defines the number of tuples for every positive group. The knob $c^-$ defines the number of groups which should not pass through the HAVING selection and its coexist knob $gc^-$ defines the number of tuples for every negative group. The positive group-count ($gc^+$) and the negative group-count ($gc^-$) can be given in terms of a lower or an upper bound. The number of positive and negative groups together must be the same as the output cardinality of the aggregation operator (i.e. $c^+ + c^- = c$).

In the following, we discuss the algorithms to implement that knob for the symbolic aggregation. The hardness of the problem depends on whether the input is pre-grouped on the group-by attribute or not. Note, in both cases the user cannot define the distribution on how to assign the input tuples into different output groups because this would conflict with the above knobs.

Special case of GROUP-BY with HAVING, sub-case 1: Input is not pre-grouped on the group-by attribute

Assume that the aggregation operator of the query above gets an input of 10 tuples which is not pre-grouped on the group-by attribute $o\_date$. Furthermore, the user defines the following knob values: $gc^+ \geq 2$, $c^+ = 3$, $gc^- \leq 1$, $c^- = 2$. Thus the output cardinality of the aggregation operator is $c = 5$ in the example. The following shows the example input:

\[
\begin{align*}
\frac{gc^+ \geq 2}{gc^+ \geq 2} \\
\frac{gc^- \leq 1}{gc^- \leq 1}
\end{align*}
\]

The aggregation operator controls the output by:

1. [Assign tuples to output groups with a upper bound group-count] During its open() method, it first assigns one tuple to each output group with upper bound of number of tuples in the group as knob.
2. [Assign tuples to output groups with a lower bound group-count] Assign the minimum number of tuples to each output group with lower bound of number of tuples in the group as knob.
3. [Post-processing] If there are still some tuples in the input which are not assigned to an output group, then assign these input tuples to the output groups as follows: (a) if there are some output groups with lower bound group-count, then assign all remaining tuples to one of these output groups. (b) if there are only output groups with upper bound group-count, then assign tuples to those output groups until its upper-bound has been reached.
4. [Aggregating] During each getNext() call, get an output group $O_i$, invoke the Simple Aggregation Operator (Section 5.2.4.1) like the normal case does.
5. [Result Returning] Construct a new symbolic tuple \( \langle t.g, agg_i \rangle \) to its parent, where \( agg_i \) is the symbolic tuple returned by the Simple Aggregation Operator for the group \( O_i \) and \( t.g \) is the symbol of the group-by attribute of \( O_i \). Return the constructed tuple to its parent.

In the example, the negative output groups uses \( gc^- \leq 1 \) as the knob value. Therefore, each of the two negative group gets one tuple during Step 1. The positive output groups uses \( gc^+ \geq 2 \) as the knob value. Thus each of the three positive output groups gets two tuples during Step 2. For the two remaining tuples out of the 10 input tuples, they are distributed to the first positive output group.

Special case of GROUP-BY with HAVING, sub-case 2: Input is pre-grouped on the group-by attribute

This subcase contains the NP-complete Group Assignment Problem defined in Appendix A and is therefore NP-hard. We present an efficient heuristic that solves most of the instances that arise in practice. For the remaining instances it can in general provide a solution that meets most of the group constraints. The heuristic is inspired by the best fit decreasing algorithm (BFD) for the bin packing problem [18]. The basic idea of the BFD algorithm is that it considers the items in the order of non-increasing item sizes. Among the possible bins for an item, the algorithm always chooses the one that would have minimum leftover space after addition of that item. If an item fits in no bin, a new bin is opened.

In this special case of GROUP-BY with HAVING, we treat a resulting output group as a bin and a pre-group an item.

When all group constraints are upper bound constraints, we have such a classical bin packing problem with different bin sizes and a fixed number of bins. Basically, the resulting problem asks for a feasible packing for the given bin sizes. For this case we propose to execute the BFD algorithm as sketched above (with all bins being initially open). If this does not lead to a feasible solution, it is easy to provide a solution that fulfills all but one of the lower equal constraints and assigning the leftover items to one particular bin.

When the group constraints consist of mixed greater equal and lower equal constraints, we have a bin packing and filling problem with \( p \) packing (lower equal constraints) and \( c \) covering (greater equal constraints) bins. It becomes trivial to fulfill all lower equal constraints. Without loss of generality, for the \( p \) lower equal constraints we can assign the \( i \)-th smallest item (pre-group) to the \( i \)-th smallest packing bin (output group) for \( 1 \leq i \leq p \). If a packing bin is overfilled in this process the problem instance must be infeasible. In this case we have to accept the bin being overfilled.

It remains to clarify how to deal with the covering bins. For this problem we propose to iteratively search for a solution that satisfies as many constraints as possible. To this end we search for solutions that cover the \( c' \leq c \) w.l.o.g. smallest covering bins, starting at \( c' = c \). Although theoretically a binary search would be faster for finding the maximum \( c' \) we expect that for real instances \( c' \) will be very close to \( c \), which justifies a linear search. For a given \( c' \) the algorithm relies on the observation that a good cover of the bins overpacks these as little as possible. Therefore, we propose an analogous
approach to best fit decreasing. Execute the best fit decreasing algorithm to fill the bins as good as possible. If the addition of an item leads for all bins to an overfill, assign it to the one where it causes the minimum overfill. This is basically the same idea as for best fit decreasing except that now also negative leftover space (overfill) is considered.

5.2.4.4 Multiple GROUP-BY Attributes  If there is a set of group-by attributes $G$ (with multiple attributes), then the implementation of the aggregation operator depends not only on whether the input is pre-grouped, but also depends on whether the group-by attributes in the input are in tree-structure or in graph-structure (see Section 3). QAGen currently supports queries with tree-structure (see Figure 4). Studying the problem of controlling the output cardinality of an aggregation operator with group-by attributes which are in graph structure is part of the future work.

The aggregation operator treats aggregation with multiple group-by attributes as same as the case of a single group by attribute (Section 5.2.4.2). Assume attribute $a_n$ is the attribute in $G$ which is functional dependent on the least number of other attributes in $G$. The aggregation operator treats $a_n$ as the single group-by attribute and set the rest of attributes in $A$ to a constant value $v$ (attribute $a_n$ is selected because it has the largest number of distinct symbols in the input comparing to the other attributes).

As an example, assume the following table is an input to an aggregation operator.

<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$c_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$c_2$</td>
<td>$d_1$</td>
</tr>
</tbody>
</table>

Assume the set of group-by attributes $A$ is \{b, c, d\}, and the functional dependencies which hold on the input of the aggregation operator are: \{b\} → \{c, d\} and \{c\} → \{d\}. According to the definition in Section 3, the set of group-by attributes $G$ is in tree-structure.

In the input above, attribute $b$ is functional dependent on least other attributes in $G$ ($b$ is functional dependent on no attributes where $d$ is functional dependent on $b$ and $c$. As a result, the aggregation operator treats attribute $b$ as the single group-by attribute invoke the single group-by aggregation implementation and set the other attributes to use the same symbol for all input tuples (e.g., set all symbols for attribute $b$ to be $b_1$).

Since the aggregation operator with multiple-group attributes essentially is handled by the aggregation operator that supports single group-by attribute, it shares the same special cases (HAVING clause on top on an aggregation where the parameter values control the group count) as the single group-by case.

5.2.5 Symbolic Execution of Projection Operator

\[
\begin{array}{c|c|c|c}
\hline
\text{SUM(1, price)} & \hline
\text{aggsum1} & \hline
\end{array}
\]

Table F. Output of $\pi(1 \text{ tuple})$
Symbolic execution on a projection operator is exactly the same as the traditional query processing; it projects the specified attributes and no additional constraints are added. As a result, the final projection operator in the running example takes in the input from Table E and ends with the result shown in Table F.

5.2.6 Symbolic Execution of Union Operator

In SQL, the UNION operator eliminates the duplicates if they exist. On the other hand, the UNION ALL operator does not eliminate the duplicates. In SQP, the query analyzer does not offer any knob to the user to tune the UNION ALL operation. This operation is straightforward to implement: it reuses the UNION ALL operator in RDBMS and union the two inputs into one.

Let $R$ and $S$ be the inputs of the UNION operation which are not pre-grouped. In this case, the query analyzer offers the following knob to the user:

**Knob:** Output Cardinality $c$ (optional; default value = size of $R +$ size of $S$)

The UNION operator controls the output as follows:

1. During its getNext() call, if the output cardinality has not yet reached $c$, then (a) get a tuple $t$ from $R$ (or from $S$ alternatively), and (b) return $t$ to its parent. However, during the getNext() call, if the output cardinality has reached $c$ already, then process [Post-processing] below, and return null to its parent.

2. [Post-processing] Fetch the remaining tuples $T^-$ from inputs $R$ and $S$, set the symbols in tuple $t^- \in T$ to have the same symbol as one of the returned tuple $t$ in the previous step.

5.2.7 Symbolic Execution of MINUS Operator

In SQL, the MINUS operator selects all distinct rows from the first query but not the second.

Let $R$ and $S$ be the non-pregrouped inputs of the MINUS operation. In this case, the query analyzer offers the following knob to the user:

**Knob:** Output Cardinality $c$ (optional; default value = size of $R$)

The MINUS operator controls the output as follows:

1. During its getNext() call, if the output cardinality has not yet reached $c$, then (a) get a tuple $r^+$ from $R$ and (b) return $r^+$ to its parent. However, during the getNext() call, if the output cardinality has reached $c$ already, then process [Post-processing] below, and return null to its parent.

2. [Post-processing] Fetch a tuple $r^-$ from $R$, fetch all tuples $S^-$ from $S$, set the symbols in tuple $s^- \in S^-$ to have the same symbol as $r^-$. 

5.2.8 Symbolic Execution of INTERSECT Operator

**Knob:** Output Cardinality $c$ (optional; default value = size of $R$)

30
In SQL, the INTERSECT operator returns all distinct rows selected by both queries. Currently, QAGen supports INTERSECT with non-pregrouped inputs. Let \( R \) and \( S \) be the input of the INTERSECT operator, the symbolic execution of the INTERSECT operator is as follows:

1. During its getNext() call, if the output cardinality has not yet reached \( c \), then (a) get a tuple \( r^+ \) from \( R \), and get a tuple \( s^+ \) from \( S \), (b) set the symbols of \( s^+ \) as same as \( r^+ \). and return \( r^+ \) to its parent. However, during the getNext() call, if the output cardinality has reached \( c \) already, return null to its parent.

5.2.9 Symbolic Execution of Nested Query

Nested queries in symbolic query processing reuses the techniques in traditional query processing because queries can be unnested by using join operators [13]. In order to allow a user to have full control on the input, the user should give the input query in its unnested format. If the inner query and the outer query refer to the same table(s), then the query analyzer disables some knobs on operators that may allow a user to specify different constraints on the operators that work on the same table in both inner and outer query.

6 Data Instantiation

The final phase of the data generation process is the data instantiation phase. The data instantiator fetches the symbolic tuples from the symbolic database and uses a constraint solver (strictly speaking, the constraint solver is the decision procedure of a model checker [10]) to instantiate concrete values for them. The constraint solver takes a propositional formula (remember that all predicates can be represented by propositional formula) as input and returns a set of concrete values for the symbols in the formula that satisfies all the input predicates and the actual data types of the symbols. If the input formula is unsatisfiable, the constraint solver returns error. Such errors, however, cannot occur in this phase because contradicting knob values are handled during symbolic query processing. A constraint solver call is an expensive operation. In the worst case, the cost of a constraint solver call is exponential to the size of the input formula [10]. As a result, the objective of the data instantiator is to minimize the number of calls to the constraint solver if possible. Indeed, the predicate size optimizations during symbolic query processing (e.g. reducing \( aggsum = l\_price1 + \ldots + l\_price8 \) to \( aggsum = l\_price1 \times 8 \)) are designed for this purpose. After the data instantiator has collected all the concrete values of a symbolic tuple, it inserts the instantiated tuple into the final test database. The details of the data instantiator is as follows:

1. The process starts from any one of the symbolic tables.
2. It reads in a tuple \( t \), say \( \langle c\_id1, c\_acctbal1 \rangle \), from the symbolic tables.
3. [Look up symbol-to-value cache] For each symbol \( s \) in tuple \( t \), (a) it first looks up \( s \) in a table called SymbolValueCache in the symbolic database. The SymbolValueCache is a
table in the symbolic database that stores the concrete values of the symbols that have been instantiated by the constraint solver. (b) If the symbol \( s \) has been instantiated with a concrete value, then the symbol is initialized with the same cached value and then proceeds to the next symbol in \( t \).

In the running example, assume the constraint solver randomly instantiates the Customer (4 tuples) table first. Since the symbol \( c\_id1 \) is the first symbol to be instantiated, it has no instantiated value stored in the SymbolValueCache table. However, assume later when instantiating the first two tuples of Orders table (with \( o\_id1, o\_id2 \)), their \( o\_cid \) values will use the same value as instantiated for \( c\_id1 \) by looking up the SymbolValueCache.

4. \[\text{Instantiate values}\] Look up the predicates \( P \) of \( s \) from PTable. (a) If there are no predicates associated with \( s \), then instantiate \( s \) by a unique value that matches the actual domain of \( s \) in the input schema \( M \).

In the example, \( c\_id1 \) does not have any predicates associated with it (see PTable in Figure 8). Therefore, the data instantiator does not instantiate \( s \) with a constraint solver but instantiates a unique value \( v \) (because \( c\_id \) is a primary key), say, 1, to \( c\_id1 \). Afterwards, insert a tuple \( \langle s,v \rangle \) (e.g., \( \langle c\_id1,1 \rangle \)) to the SymbolValueCache.

(b) However, if \( s \) has some predicates \( P \) in PTable, then compute the \textit{predicate closure} of \( s \). The predicate closure of \( s \) is computed by recursively looking up all the directly correlated or indirectly correlated predicates of \( s \).

For example, the predicate closure of \( l\_price1 \) is \([\text{aggsum}1 = 5 \times \text{l\_price1} \text{ AND aggsum}1 \geq p2]\). Then the predicate closure (which is in the form of conjunctive propositional formula) is sent to the constraint solver (symbols exist in the SymbolValueCache are replaced by their instantiated values first). The constraint solver instantiates all symbols in the formula in a row (e.g., \( l\_price1 = 10, \text{aggsum}1 = 50, p2 = 18 \)).

For efficiency purposes, before a predicate closure is sent to the constraint solver, the data instantiator looks up another cache table called PredicateValuesCache in the symbolic database. This table caches the instantiated values of predicates. Since many predicates in the PTable essentially share the same pattern, the predicates stored in PredicateValuesCache are in the predicate pattern format. For example, the predicates \([c\_acctbal1 \geq p1]\) and \([c\_acctbal2 \geq p1]\) in Figure 8(c) share the same pattern: \([c\_acctbal \geq p1]\). As a result, after the instantiation of the predicate \([c\_acctbal1 \geq p1]\), the data instantiator inserts an entry \([\langle c\_acctbal \geq p1, c\_acctbal1, p1 \rangle]\) into the PredicateValuesCache table. When the next predicate closure \([c\_acctbal2 \geq p1]\) needs to be instantiated, the data instantiator looks up the predicate in PredicateValuesCache by its pattern; if the same predicate pattern is in PredicateValuesCache, then the data instantiator skips the instantiation of this predicate and reuses the instantiated value of \( c\_acctbal1 \) in the SymbolValueCache table for the symbol \( c\_acctbal2 \) (same for \( p1 \)).

The number of constraint solver calls is minimized by the introduction of the SymbolValueCache and PredicateValuesCache tables. Experiments show that this feature is crucial or otherwise generating a 1G query-aware database takes weeks instead of hours. Finally, note that in Step 4a, if a symbol \( s \) has no predicate associated with it, the data instantiator assigns a value to \( s \) according to its domain. Except for attributes with integrity constraints (e.g., primary keys), those values can be assigned randomly or always use the same value. It is unnecessary to instantiate any extra data characteristics (e.g., distribution) for those symbols because they do not participate in the query at all (i.e., their values do not affect the query results anyway).
7 Experiments

This section shows the results of the experiments with our prototype system QAGen. QAGen was implemented in Java and installed on a Linux AMD Opteron 2.2 GHz Server with 4 GB of main memory. The symbolic database and the target database used PostgreSQL 7.4.8 and they were installed on the same machine. As a constraint solver, a publicly available constraint solver called Cogent [12] was used.

We executed three sets of experiments with the following objectives: The first experiment (Section 7.1) studied the efficiency of the symbolic execution of individual operators. The second experiment (Section 7.2) generated different test databases for the same query but with different knob values in order to study if the knob values could affect the physical execution plan. The third experiment (Section 7.3) studied the scalability of QAGen for generating different database sizes for different queries.

7.1 Efficiency of Symbolic Operations

The objective of this experiment is to evaluate (1) the running time of individual symbolic operators; (2) their scalability, and (3) the running time of the data instantiation phase by generating three query-aware databases in different scales (10M, 100M, and 1G). The input query was query 8 in the TPC-H benchmark [4]. Its logical query plan input to QAGen is shown in Figure 9a. We chose TPC-H query 8 because it is one of the most complex queries in TPC-H with 7-way joins and aggregations. This query has various input characteristics to the operators enabling us to evaluate the performance of different operator implementations (e.g., the normal equi-join and the special case of equi-join that needs dynamic programming). The experiments were carried out in the following way: First, three benchmark databases were generated using dbgen from TPC-H benchmark. As a scaling factor, we used 10 MB, 100 MB, 1GB. Then, we executed query 8 on top of the three TPC-H databases, and collected the base table sizes and the cardinality of each intermediate result under the three scaling factors. The extracted cardinality of each intermediate result of query 8 is shown in Table 1 (Output-size) columns. Next, we generated three TPCH-query-8-aware databases with the collected base table sizes and output cardinalities as input and measured the efficiency of QAGen for generating databases that produced the same cardinality results. For this experiment, the join distribution was uniform.

Table 1 shows the cost breakdown of generating query-aware databases for TPC-H query 8 in detail. QAGen only took about 10 minutes for generating a 10MB query-aware database. The symbolic query processing phase was fast and scaled linearly. It took about 1 minute for 10MB and less than 3 hours for 1G database. The longest SQP operations were the initialization of the big symbolic table Lineitem (Line 10 in Table 1), and the join between the intermediate result $R5$ and Lineitem (Line 11). That join needed long time because it accessed the large Lineitem table frequently to update the symbolic values of the join attributes. In query 8, the input was pre-grouped on the last join (line 17 in Table 1 and operator (17) in Figure 9). However, the dynamic programming equi-join finished quickly because the input and output sizes were not
Table 1: QAGen Execution Time for TPC-H Query 8

<table>
<thead>
<tr>
<th>#</th>
<th>Symbolic operation</th>
<th>$size = 10M$</th>
<th></th>
<th>$size = 100M$</th>
<th></th>
<th>$size = 1G$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Output-size</td>
<td>Time</td>
<td>Output-size</td>
<td>Time</td>
<td>Output-size</td>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Region</td>
<td>5</td>
<td>&lt; 1s</td>
<td>5</td>
<td>&lt; 1s</td>
<td>5</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>2</td>
<td>$\sigma(Region) \Join R1$</td>
<td>1</td>
<td>&lt; 1s</td>
<td>1</td>
<td>&lt; 1s</td>
<td>1</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>3</td>
<td>Nation</td>
<td>25</td>
<td>&lt; 1s</td>
<td>25</td>
<td>&lt; 1s</td>
<td>25</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>4</td>
<td>$(R1 \Join Nation) \Join R2$</td>
<td>5</td>
<td>&lt; 1s</td>
<td>5</td>
<td>&lt; 1s</td>
<td>5</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>5</td>
<td>Customer</td>
<td>1.5k</td>
<td>&lt; 1s</td>
<td>15.0k</td>
<td>5s</td>
<td>150k</td>
<td>49s</td>
</tr>
<tr>
<td>6</td>
<td>$(R2 \Join Customer) \Join R3$</td>
<td>0.3k</td>
<td>1s</td>
<td>3.0k</td>
<td>7s</td>
<td>299.5k</td>
<td>75s</td>
</tr>
<tr>
<td>7</td>
<td>Orders</td>
<td>15.0k</td>
<td>4s</td>
<td>150.0k</td>
<td>45s</td>
<td>1.5m</td>
<td>553s</td>
</tr>
<tr>
<td>8</td>
<td>$\sigma(Orders) = R4$</td>
<td>4.5k</td>
<td>8s</td>
<td>45.0k</td>
<td>67s</td>
<td>457.2k</td>
<td>709s</td>
</tr>
<tr>
<td>9</td>
<td>$(R3 \Join R4) = R5$</td>
<td>0.9k</td>
<td>3s</td>
<td>9.0k</td>
<td>22s</td>
<td>91.2k</td>
<td>277s</td>
</tr>
<tr>
<td>10</td>
<td>Lineitem</td>
<td>60.0k</td>
<td>26s</td>
<td>600.5k</td>
<td>237s</td>
<td>6001.2k</td>
<td>2629s</td>
</tr>
<tr>
<td>11</td>
<td>$(R5 \Join Lineitem) \Join R6$</td>
<td>3.6k</td>
<td>34s</td>
<td>35.7k</td>
<td>348s</td>
<td>365.1k</td>
<td>4694s</td>
</tr>
<tr>
<td>12</td>
<td>Part</td>
<td>2.0k</td>
<td>&lt; 1s</td>
<td>20.0k</td>
<td>5s</td>
<td>200k</td>
<td>60s</td>
</tr>
<tr>
<td>13</td>
<td>$\sigma(Part) = R7$</td>
<td>12</td>
<td>1s</td>
<td>147</td>
<td>8s</td>
<td>1451</td>
<td>72s</td>
</tr>
<tr>
<td>14</td>
<td>$(R7 \Join R6) \Join R8$</td>
<td>29</td>
<td>3s</td>
<td>282</td>
<td>27s</td>
<td>2603</td>
<td>533s</td>
</tr>
<tr>
<td>15</td>
<td>Supplier</td>
<td>0.1k</td>
<td>&lt; 1s</td>
<td>1k</td>
<td>&lt; 1s</td>
<td>10k</td>
<td>3s</td>
</tr>
<tr>
<td>16</td>
<td>$(Supplier \Join R8) = R9$</td>
<td>29</td>
<td>&lt; 1s</td>
<td>282</td>
<td>1s</td>
<td>2603</td>
<td>6s</td>
</tr>
<tr>
<td>17</td>
<td>$(Nation \Join R9) = R10$</td>
<td>29</td>
<td>&lt; 1s</td>
<td>282</td>
<td>&lt; 1s</td>
<td>2603</td>
<td>3s</td>
</tr>
<tr>
<td>18</td>
<td>$\chi(R8) = R11$</td>
<td>2</td>
<td>&lt; 1s</td>
<td>2</td>
<td>1s</td>
<td>2</td>
<td>10s</td>
</tr>
<tr>
<td></td>
<td>Symbolic Query Processing</td>
<td>01m : 20s</td>
<td></td>
<td>12m : 53s</td>
<td></td>
<td>161m : 13s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data Instantiation (# Cogent-call)</td>
<td>09m : 31s (14)</td>
<td></td>
<td>96m : 03s (14)</td>
<td></td>
<td>1062m : 54s (14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>10m : 51s</td>
<td></td>
<td>108m : 56s</td>
<td></td>
<td>1224m : 07s</td>
<td></td>
</tr>
</tbody>
</table>

The data instantiation phase dominated the whole data generation process. It took about 9 minutes to instantiate a 10M query aware database and about 17 hours to instantiate a 1G query aware database. Nevertheless, about 40% of time were the overhead of reading symbolic tuples and inserting concrete tuples (not shown in the Table). In the experiments, the number of constraint solver (cogent) calls was small — there were only 14 calls for 3 scaling factors. The number of calls was constant because the data instantiator cached the pattern of the predicates but not the concrete predicates. We indeed repeated the same experiment by turning off the caching feature of QAGen, but it ended up that the data instantiation phase for a 1G database could not finish within 2 weeks because the constraint solver took a lot of time. It proved that the predicate optimization in SQP and the caching in the data instantiator work effectively.

### 7.2 Effects of Knob Values

The objective of this experiment is to show that by setting different cardinalities and distributions on the operators, QAGen can generate test databases that give different physical execution plans for the same query. In this experiment, the target database
<table>
<thead>
<tr>
<th>Result</th>
<th>TPC-H</th>
<th>MIN</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>R2</td>
<td>5</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>R3</td>
<td>3k</td>
<td>1</td>
<td>15k</td>
</tr>
<tr>
<td>R4</td>
<td>45k</td>
<td>1</td>
<td>150k</td>
</tr>
<tr>
<td>R5</td>
<td>9k</td>
<td>1</td>
<td>150k</td>
</tr>
<tr>
<td>R6</td>
<td>36k</td>
<td>1</td>
<td>600k</td>
</tr>
<tr>
<td>R7</td>
<td>147</td>
<td>1</td>
<td>20k</td>
</tr>
<tr>
<td>R8</td>
<td>282</td>
<td>1</td>
<td>600k</td>
</tr>
<tr>
<td>R9</td>
<td>282</td>
<td>1</td>
<td>600k</td>
</tr>
<tr>
<td>R10</td>
<td>282</td>
<td>1</td>
<td>600k</td>
</tr>
<tr>
<td>R11</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Uniform</td>
<td>Fig. 9b, Fig. 9c, Fig. 9e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zipf</td>
<td>Fig. 9b, Fig. 9d, Fig. 9e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Knob Values and Resulting Query Plans

size was fixed at 100MB and the input query was query 8 in TPC-H. The experiments were carried out in the following way: First, we generated 6 query-aware databases for query 8 with different knob values. Then, we executed query 8 on the 6 generated databases (on PostgreSQL) and studied their physical execution plans. The first database [TPCH-Uniform] was generated using the intermediate result sizes extracted from executing query 8 on TPC-H dbgen database (as in the first experiment above). The second database [TPCH-Zipf] was generated using the same intermediate result sizes as above but with Zipf distribution (skew factor = 1). The third [MIN-Uniform] and the fourth database [MIN-Zipf] were generated using the minimum possible value on each intermediate result size but with different distributions. The fifth [MAX-Uniform] and the last database [MAX-Zipf] were generated using the maximum possible value on each intermediate result size but with different distributions. Table 2 shows the intermediate result sizes of the above setup.

Figure 9 shows the physical evaluation plans of executing TPC-H query 8 on the generated query-aware databases. By controlling the output cardinalities of the operators, it caused PostgreSQL to use different join strategies. For example, when the cardinality of each output was set to minimum [MIN-Uniform], PostgreSQL tended to use left-deep-join order (Figure 9c). When the cardinality of each output was set to maximum [MAX-Uniform], PostgreSQL tended to use bushy-tree join order (Figure 9e). The output cardinalities also strongly influenced the choice of physical operators; when the output cardinality was big, PostgreSQL tended to use hash join (Figure 9e). However, when the output cardinality was small, PostgreSQL tended to use fewer hash join but used sort-merge-join and nested-loop-join (Figure 9b,c,d). The input and output cardinality also influenced the choice of physical aggregation operators. When the input to the aggregation (i.e., R10 in Table 2) was set as minimum or TPC-H size, then PostgreSQL tended to use group aggregation (Figure 9b,c,d). However, when the input to was set as maximum, then PostgreSQL tended to first do a hash aggregation and then sort it (Figure 9e). Controlling the distributions of the query operators also influenced
PostgreSQL, but it is less significant. For example, for the same cardinality setting, changing the join distribution influenced the join plans only when the cardinality was small (Figure 9c and d). When the cardinality was set with TPCH size (Figure 9b) or maximum size (Figure 9e), the distribution setting did not influence the join plans. Moreover, the distribution setting also had less influences on the choice of operators.

In this experiment, we attempted to use other database generation tools to generate the same set of test databases which can produce the same intermediate query results. We tried with two commercial test database generators, DTM [1] and IBM Test Database Generator [2] and one research tool [17][3]. However, these tools only allow constraining the base tables properties and we failed to manually control the intermediate result sizes for the purpose of this experiment. Another attempt was to use the query parameter generation tool from [8] to generate query parameters on top of the generated databases. However, [8] can only support SPJ queries (with single-sided or double-side predicates) which is not suitable for the complex TPC-H queries (which include aggregation and complex predicates) in this experiment.

### 7.3 Scalability of QAGen

The last experiment is to evaluate the scalability of QAGen for generating a variety of query-aware test databases. Currently, QAGen supports 13 out of 22 TPC-H queries. It does not support some queries because those queries either fall into the special cases of QAGen (e.g., query 5 (Q5) falls into the special case of selection in Section 5.2.2

[3] We also attempted to evaluate DGL from Microsoft [7], however their tool is not publicly available.
<table>
<thead>
<tr>
<th>Query</th>
<th>Phase</th>
<th>10M</th>
<th>100M</th>
<th>1G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQP</td>
<td>02m:48s</td>
<td>26m:45s</td>
<td>321m:27s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>07m:42s</td>
<td>78m:35s</td>
<td>844m:52s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>10m:22s</td>
<td>105m:10s</td>
<td>1166m:19s</td>
</tr>
<tr>
<td>2</td>
<td>SQP</td>
<td>00m:09s</td>
<td>01m:32s</td>
<td>16m:47s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>02m:27s</td>
<td>24m:55s</td>
<td>249m:50s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>02m:36s</td>
<td>26m:27s</td>
<td>256m:37s</td>
</tr>
<tr>
<td>3</td>
<td>SQP</td>
<td>01m:35s</td>
<td>16m:18s</td>
<td>185m:21s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>09m:34s</td>
<td>97m:07s</td>
<td>1016m:59s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>11m:09s</td>
<td>113m:25s</td>
<td>1202m:20s</td>
</tr>
<tr>
<td>4</td>
<td>SQP</td>
<td>02m:32s</td>
<td>23m:23s</td>
<td>221m:17s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>06m:10s</td>
<td>67m:22s</td>
<td>627m:11s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>08m:42s</td>
<td>80m:45s</td>
<td>848m:28s</td>
</tr>
<tr>
<td>6</td>
<td>SQP</td>
<td>01m:52s</td>
<td>64m:36s</td>
<td>180m:22s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>10m:36s</td>
<td>333m:31s</td>
<td>1121m:06s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>12m:28s</td>
<td>398m:07s</td>
<td>1301m:28s</td>
</tr>
<tr>
<td>10</td>
<td>SQP</td>
<td>01m:16s</td>
<td>12m:56s</td>
<td>156m:22s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>09m:42s</td>
<td>98m:13s</td>
<td>1107m:10s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>10m:58s</td>
<td>111m:09s</td>
<td>1263m:32s</td>
</tr>
<tr>
<td>12</td>
<td>SQP</td>
<td>02m:11s</td>
<td>21m:32s</td>
<td>244m:07s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>12m:01s</td>
<td>123m:04s</td>
<td>1387m:27s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>14m:12s</td>
<td>144m:36s</td>
<td>1631m:34s</td>
</tr>
<tr>
<td>14</td>
<td>SQP</td>
<td>01m:39s</td>
<td>08m:47s</td>
<td>95m:49s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>17m:15s</td>
<td>94m:50s</td>
<td>1023m:39s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>18m:54s</td>
<td>103m:27s</td>
<td>1119m:28s</td>
</tr>
<tr>
<td>15</td>
<td>SQP</td>
<td>00m:58s</td>
<td>09m:10s</td>
<td>98m:07s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>05m:40s</td>
<td>92m:24s</td>
<td>966m:10s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>06m:38s</td>
<td>101m:34s</td>
<td>1064m:17s</td>
</tr>
<tr>
<td>16</td>
<td>SQP</td>
<td>00m:14s</td>
<td>01m:42s</td>
<td>27m:01s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>05m:38s</td>
<td>05m:19s</td>
<td>52m:40s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>06m:52s</td>
<td>07m:01s</td>
<td>79m:41s</td>
</tr>
<tr>
<td>18</td>
<td>SQP</td>
<td>00m:55s</td>
<td>08m:20s</td>
<td>86m:30s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>08m:41s</td>
<td>86m:53s</td>
<td>861m:11s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>09m:36s</td>
<td>95m:13s</td>
<td>947m:41s</td>
</tr>
<tr>
<td>19</td>
<td>SQP</td>
<td>04m:14s</td>
<td>41m:45s</td>
<td>411m:12s</td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td>97m:23s</td>
<td>973m:03s</td>
<td>9707m:11s</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>101m:37s</td>
<td>1014m:48s</td>
<td>10118m:23s</td>
</tr>
</tbody>
</table>

Table 3: QAGen Scalability
case 2); or because some of them use non-equi-join (e.g., Q16, Q22). Nevertheless, we generated query-aware databases for the rest of the queries in three different scaling factors 10M, 100M, and 1G. Table 3 shows the detailed results. Experimental results show that both phases scaled well for all 13 TPC-H queries and the data instantiation (DI) phase was still the time dominating phase.

8 Related Work

The closest related work in DBMS testing is the work of [8] which studied how to generate query parameters for test queries with the test databases given. However, existing database generation tools such as [2, 1, 16, 17, 7] were designed to generate general-purpose test databases without concerning the test queries, and thus the generated databases cannot guarantee they cover specific test cases. As a consequence, [8] can hardly find a good database to work on and eventually only approximate solutions are feasible.

QAGen extends symbolic execution [19] and proposes the concept of symbolic query processing (SQP) to generate query-aware databases. SQP is related to constraint databases (e.g., [21]); however, constraint databases focus on constraints that represent infinite concrete data (e.g., spatial-temporal data) whereas SQP works on finite but abstract data. Recently, [5, 6] also studied test database generation which takes a query as input. Their system takes the query and the query result as input and processes the query reversely, in order to generate a database that can return the given query result. Their focus is to generate minimal size test databases for functional testing of database applications, rather than testing the DBMS itself. That work is orthogonal to our work.

9 Conclusions and Future Work

This work presented QAGen, a system which generates tailor-made test databases for different DBMS test cases. QAGen is based on symbolic query processing, a technique that combines traditional query processing and symbolic execution from software engineering. It could be shown that QAGen is able to generate query-aware databases for complex queries and it scales linearly.

One of our most important avenues for future work is to support more operators and to support more special cases in QAGen. Another important piece of future work is to exploit the possibility of instantiate many symbolic tuples in parallel during the data instantiation phase. In terms of DBMS testing, we would also like to investigate the possibility of generating many meaningful test databases, without asking the user to specifying the knob values (e.g., cardinality) explicitly, but automatically generates many sets of knob values that can test the system thoroughly. Finally, we believe the work of SQP can be integrated with traditional symbolic execution so as to extend program verification and test case generation techniques to support database applications.
A Complexity of the Group Assignment Problem

Problem Definition: The combinatorial problem that we call the Group Assignment problem is non-trivial to define. Therefore, we begin by defining the major entities of the input and then define the problem itself.

Definition 1 (Group Assignment Problem (GA), input). The input \((G, C, r)\) consists of a ground set \(G\) of \(n\) items where each item represents a pre-group of the input of an aggregation operator, as well as a constraint set \(C\) of \(m\) group-count constraints which result from the positive and negative group-count knob values (and \(m\) is the output cardinality of the aggregation operator).

We first describe the ground set and the associated variable set: Each item \(a \in G\) has an item size \(s(a) \in \mathbb{N}\) and an associated \(d\)-dimensional variable vector \(v(a)\). A variable vector \(v = (v_1, \ldots, v_d)\) is taken from the Cartesian product of \(d\) disjoint variable sets \(\Sigma_1, \ldots, \Sigma_d\). Thus, the variable vector \(v(a)\) represents the symbolic values of the group-by attributes of the pre-group which is represented by \(a\) and the size \(s(a)\) represents the size of that pre-group. In case that the input is not pre-grouped on the group-by attributes, the size is \(s(a) = 1\) for all items \(a \in G\).

The goal of the GA is to partition the items into \(m\) groups \((\gamma_1, \ldots, \gamma_m)\), where \(\bigcup_i \gamma_i = G\), such that each group \(\gamma\) meets the associated group constraint \(c(\gamma) \in C\). Each group constraint \(c(\gamma_i)\) is of one of the following three types:

\[
\sum_{a \in \gamma_i} s(a) \leq \text{rhs}(i) \quad \sum_{a \in \gamma_i} s(a) = \text{rhs}(i) \quad \sum_{a \in \gamma_i} s(a) \geq \text{rhs}(i)
\]

Definition 2 (GA problem). Given an input \((G, C, r)\) to the GA problem, partition the ground set into \(m\) groups \((\gamma_1, \ldots, \gamma_m)\) such that the constraints given by \(C\) hold.

It is not surprising that the GA-problem is NP-complete.

Lemma 3. The GA-problem is strongly NP-complete for varying sizes of ground set (single and multiple group-by attributes) and constraint set, even for 1 dimensional variable vectors, an arbitrary single constraint type and a single fixed right hand side for the constraints.

Proof. The problem is obviously in NP, as one can guess and verify a solution. For the reduction we reduce to the 3-partition problem, problem SP15 in [14]. 3-partition asks for given set \(A\) of \(3m'\) elements of sizes \(\sigma(a')\) for \(a' \in A\) and a bound \(B\) whether \(A\) can be partitioned into \(m'\) triplets such that for each such triplet \((a'_{i1}, a'_{i2}, a'_{i3})\) we have that \(\sigma(a'_{i1}) + \sigma(a'_{i2}) + \sigma(a'_{i3}) = B\). As a precondition to the problem we require that \(\sum_{a' \in A} \sigma(a') = m'B\). This problem can be formulated by the grouping part of the above problem alone: Each element \(a' \in A\) maps to an item \(a \in G\) with \(s(a) = \sigma(a')\) and \(v(a) = \alpha_a\). In this transformation each item gets a separate variable. The constraint set consists of \(m\) constraints that impose \(\sum_{a \in \gamma_i} s(a) = B\). Note that it is possible to replace all equal constraints together by lower or greater equal constraints. \(\square\)
References


