Generating Databases for Query Workloads (Full Version)

Eric Lo† Nick Cheng‡ Wing-Kai Hon†
†Hong Kong Polytechnic University ‡National Tsing Hua University
†{ericlo, csutcheng}@comp.polyu.edu.hk ‡wkhon@cs.nthu.edu.tw

ABSTRACT
To evaluate the performance of database applications and DBMSs, we usually execute workloads of queries on generated databases of different sizes and measure the response time. This paper introduces WAGEN, an offline data generation tool that takes a set of queries as input and generates database instances for which the users can control the characteristics of the resulting workload. Applications of WAGEN include database testing, database application testing, and application-driven benchmarking. We present the architecture and the implementation algorithms of WAGEN. We also present case studies of using WAGEN and experimental results on TPC benchmarks.

1. INTRODUCTION

Query performance is a key factor of a successful database (DB) application and DBMS. To evaluate the performance of DB applications and DBMSs, we usually execute workloads of queries on generated databases in different sizes and measure the response time. This paper presents a workload-aware data generator, WAGEN. Given a database schema \( H \) and a set of queries, WAGEN allows users to generate databases in different sizes with the power to control not only the characteristics of the generated data (e.g., value distribution) but also the characteristics of the workload (e.g., cardinality of intermediate query operators). The applications of WAGEN include the following:

- Testing DBMSs Recent papers [9, 24, 7, 22] have pointed out that controlling the cardinalities of query operators in a test query is very useful in DBMS testing. For example, testers can study the performance of a hash-join implementation by varying the input and output cardinalities of the join operator [9]. Recent data generation technology has made some progress in this respect. QAGen [7] is an offline test database generator designed for this purpose. It takes a test case and a database schema \( H \) as input. A test case is a parameterized query \( Q \) with operators and base tables annotated with cardinality and data distribution constraints. The output of QAGen is a query-aware database \( D \) that conforms to \( H \) and a set of parameter values \( P \). Executing query \( Q \) (with parameter values \( P \)) on \( D \) (denoted as \( Q_P(D) \)) guarantees that the constraints annotated on \( Q \) are satisfied. QAGen every time takes only one test case as input and generates an independent test database that is specific for that test case. To carry out a test of \( n \) test cases on a DBMS product, the test team needs to maintain \( n \) separate test databases, which require a prohibitively high storage cost [24] (imagine a test suite of 1000 test cases, where each test case demands a 10GB test database).

- Differing from QAGen, WAGEN takes a set of annotated parameterized queries (or in this context, a set of DBMS test cases) as input, and generates a minimal set of database instances with the same query cardinality and data distribution assurance as QAGen does. As such, tests on DBMSs can be carried out more space efficiently.

- Stress testing database applications Consider a DB application with \( n \) SQL queries. Developers of that application can use WAGEN to generate a variety of synthetic workloads to stress the application. For example, a developer may use WAGEN to generate a 1GB database that guarantees all the application queries return millions of rows. This functionality allows the developers to test the functional and performance limits of their applications.

- Application-driven benchmarking Benchmarking requires the generation of benchmark databases. Existing benchmarks such as TPC benchmarks, although comprehensive, may not 100% reflect the performance of a DBMS with respect to an enterprise’s environment because of the differences in the schemas between TPC benchmarks and the enterprise’s DB applications. By using WAGEN, an enterprise is able to study the performance of a DBMS with respect to its own DB applications. Suppose a new start-up wishes to purchase a DBMS. The start-up may wish to know which DBMS (e.g., Oracle, SQL Server) performs the best for its application when dealing with one billion records and selective user queries. The start-up can use WAGEN to generate the relevant data and evaluate the DBMSs using its own set of database application queries. These application-specific benchmark results can complement the TPC benchmark results and provide supplementary information to the company when purchasing its DBMS.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXX-XX-X/XX/XX ...$5.00.

1 In case WAGEN generates more than one test database, we may use a database testing framework (e.g., DbUnit [1], HTpar [16]) to automatically assign the generated databases to the test queries.

2 In this case, the developers need to specify only the output cardinalities of the final results and may leave the constraints of intermediate operators empty.
To the best of our knowledge, we are the first to study the generation of workload-aware data. Compared with the state-of-the-art (single)-query-aware data generation technology, workload-aware data generation is more general and has more applications, but is also more challenging. This paper contains our solution to the problem, including the architecture and algorithms of implementing WAGEN. This paper also contains case studies of using WAGEN to generate workload-aware data for real database applications and experimental results on TPC benchmarks.

The rest of this paper is organized as follows. Section 2 covers the background and related work. Section 3 presents the architecture and algorithms of WAGEN. Section 4 shows the experimental results. Section 5 concludes the paper with future research directions.

2. BACKGROUND AND RELATED WORK

2.1 Traditional Data Generation

Currently, there are a number of commercial and academic data generators (e.g., [3, 2, 27, 15, 8, 25, 10, 18, 17]) which generate synthetic databases for a given database schema. Beside the database schema, some tools also support the input of table sizes, data repositories, and additional data constraints (e.g., statistical distributions of individual attributes and value ranges). However, all these tools do not consider the queries during data generation. Unless tuning the generated data manually, queries extracted from real database applications usually obtain empty results from those data. As a result, the benchmark results are not useful.

2.2 Query-aware Data Generation

Query-aware data generation was first studied by [23] and has received more attention in recent years. In [23], the authors studied the generation of test data that complies with functional dependencies for simple relational queries. In [5], Binnig et al. studied the generation of test data for functional testing database applications. The crux of [5] is the concept of reverse query processing, which takes an application query and the corresponding query result as input, and returns a test database that allows the given query to obtain the desired query result. The focus of [5] is to generate minimal size test databases for a single application query. Binnig et al. extended their work to process multiple application queries in [6]. In [26], the authors discussed the generation of example data to facilitate dataflow (e.g., MapReduce) programming. All these projects, however, focused on generating the smallest amount of data as possible for the ease of human understanding.

The QAAGen system [7] is a predecessor of WAGEN. QAAGen is a query-aware test database generator designed for testing query optimizers. It takes an annotated parameterized query Q and a database schema H as input. Each operator or base table in Q is annotated with a set of constraints (usually cardinality and data distribution). Figure 1a shows an annotated selection query Q1 as an example. The SQL statement of Q1 is SELECT A FROM R WHERE R.A < :p1. Q1 specifies that table R should be populated with two tuples and the query should return one tuple (:p1 is a parameter). The output of QAAGen is a query-aware database D that conforms to H and a set of parameter values P. Executing query Q (with parameter values P) on D (denoted as Q(D)) guarantees that the constraints defined on Q are satisfied.

As a means to process a query like the one in Figure 1a before the data is generated, QAAGen integrates the concept of symbolic execution [19] into traditional query processing and develops a technique called symbolic query processing (SQP). In SQP, each operator is implemented as an iterator with methods (open(), get-Next(), and close()). SQP starts with the population of a symbolic database (SDB) according to the sizes of the base tables specified in the annotated query. Figure 1b shows the SDB initialized for query Q1. A symbolic database consists of a number of symbolic relations. A symbolic relation is a collection of symbolic tuples. Inside each symbolic tuple, the values are initially represented by symbols rather than by concrete values. For instance, tuple t1 in Figure 1b is a symbolic tuple of symbolic relation R and symbol $a1$ represents any value under the domain of attribute A.

Since a symbolic database provides an abstract representation for concrete data, SQP can control the output of each operator in accordance with the user-defined constraints. Specifically, an operator in SQP evaluates the input tuples according to its own semantics. It manipulates the symbols in each input tuple in order to reflect the constraints defined on the operator. At the same time, it controls its output to its parent operator so that the parent operator can work on the right tuples. Continuing with the example in Figure 1b, when the getNext() method of the selection operator $\sigma_{R.A < :p1}$ is first invoked, it reads tuple t1 from R, annotates a “positive” constraint [:<:p1] (i.e., the selection predicate) to symbol $a1$ and returns tuple $\langle$a1$<$:p1,$b1\rangle$ to its parent. When the getNext() method of the selection operator is invoked a second time, the selection operator reads the next tuple t2 from R, and annotates a “negative” constraint [>:<:p1] (i.e., the negation of the selection predicate) to symbol $b2$. However, this time it does not return t2 to its parent because the cardinality constraint (1 tuple) is already satisfied. After symbolic query processing, the set of symbolic relations capture all the constraints defined on the input query (see Figure 1c). In the final step, a constraint solver is used to instantiate the symbolic tuples and the parameters with concrete values.4 Figure 1d shows the instantiated table R and we can see that executing Q1 on R (with :p1=22) would get exactly one tuple as defined by the user. In SQP, joins and groupings are implemented by symbol replacements. For example, if a group-by query is annotated to return 1 group from table R, the same symbolic relation in Figure 1b will be initialized but the grouping operator will replace symbol $a2$ with $a1$ during getNext() on t2. Since both tuples t1 and t2 contain $a1$ in their attribute A, the data instantiate will instantiate them with the same concrete value. In SQP, the data distribution constraints are controlled by the cardinalities.

QAAGen is mainly composed of three components: a Query Analyzer, a Symbolic Query Processing (SQP) Engine, and a Data In-

---

4 A constraint solver takes as input a constraint formula and returns an instantiation on each variable as output. For example, if an input constraint formula is $40 < a1 + b1$, a constraint solver may return $a1 = 55$, $b1 = 11$ (or any other correct instantiation) as output. Although the constraint satisfaction problem on a finite domain is $NP$-complete, in practice, there are many (best-effort) constraint solvers that can efficiently solve many forms of constraints.
stantiator. The Query Analyzer is used to parse annotated-queries and determine the cardinality or the data distribution if they are not specified on some query operator. The Symbolic Query Processing Engine is used to symbolically process the query and the Data Instantiator uses an external constraint solver called Cogenti [13] to instantiate the processed SDBs and the parameters with real values according to the user-given data distributions. The SQP Engine in QAGen includes the SQP implementations of most SQL operators including selection, projection, join, grouping, and aggregation. QAGen is thus able to generate databases for a variety of SQL queries.

3. WAGEN

WAGEN uses the symbolic query processing technique developed in [7] as a building block. However, as we will discuss later, the generation of a single symbolic database for multiple queries is a difficult problem; thus, we do not restrict ourselves to find a single database instance \( D \) for all input queries. Instead, given a database schema \( H \), a set of annotated queries \( Q = \{ Q_1, Q_2, \ldots, Q_m \} \) (the operator(s) in \( Q_i \) are annotated with cardinality constraint(s) \( C_i \)), WAGEN generates \( mn \) (\( m \leq n \)) databases \( D_1, D_2, \ldots, D_m \) and \( m \) sets of parameter values \( P_1, P_2, \ldots, P_m \), such that (1) all databases \( D_j \) (\( 1 \leq j \leq m \)) conform to \( H \), and (2) the resulting cardinalities \( C'_i \) of executing \( Q_i \) on one of the generated databases \( D_j \), using the parameter values \( P_j \), approximately meet \( C_i \) (the degree of approximation defined is based on the relative error between actual cardinalities and annotated cardinalities; details are in Section 3.2). Approximate cardinalities are sufficient for applications such as DBMS testing [9, 24] and database application testing [5]. Assume that a DBMS test engineer wants to use WAGEN to generate a workload with a 1GB database and ten application queries, in which one of the queries, \( Q_1 \), is annotated by the tester as a highly selective query that returns one row. In this case, a generated database that returns five rows for \( Q_1 \) is still very acceptable. As SQP controls the data distributions through the operator cardinalities, we focus on the control of the operator cardinalities in this paper.

Of course, if \( m = n \), that essentially means WAGEN is the same as QAGen in which each query has to be executed on a separate generated database. Therefore, the goal of WAGEN is to minimize \( m \), the number of generated databases, in best effort.

3.1 System Architecture

SQP was designed to generate \( n \) separate databases for the \( n \) input annotated queries. If SQP is carried out on a "processed" symbolic database, SQP will generate many symbolic tuples with contradicting constraints (as different queries may impose different constraints on the same symbolic tuple) and they will be unable to be instantiated with concrete values.

To illustrate, assume that we need to generate a database for two (annotated) application queries \( Q_1 \) and \( Q_2 \). Let \( Q_1 \) be the query given in Figure 1a, and \( Q_2 \) be a selection query \( \text{SELECT} \ast \text{FROM} R \text{WHERE} R.a > p_1 \), which specifies that table \( R \) should have two tuples and the query should return one tuple (Figure 2a). Assume \( Q_1 \) is first symbolically processed by the SQP engine and we obtain the symbolic database \( D_1 \) (Figure 1c). If the second query \( Q_2 \) is directly processed on \( D_1 \), the selection operator of \( Q_2 \) may annotate the positive constraint \( [\text{p1}] \) to \( t_1 \) and the negative constraint \( [\text{p1}] \) to \( t_2 \). That will result in an SDB (Figure 2b) in which tuple \( t_1 \) is associated with a contradicting constraint \[\text{p1} \land \text{p1} \land \text{p1} \land \text{p1} \land \text{p1} \]. In fact, \( Q_2 \) must annotate consistent constraints with \( Q_1 \) (e.g., two tuples for table \( R \) or otherwise WAGEN will return an error to the user.

![Figure 2: Examples of SQP on a "processed" SDB](image)

Figure 3: WAGEN architecture

\[\text{p1} \land \text{p1} \land \text{p1} \land \text{p1} \land \text{p1} \]

Figure 3 shows our proposed architecture for WAGEN. To generate \( mn \) databases for \( n \) annotated queries \( Q_1, Q_2, \ldots, Q_n \), WAGEN first uses QAGen’s SQP engine as a black-box component to process each annotated query separately (without data instantiation) and generates \( n \) symbolic databases \( D_1, D_2, \ldots, D_n \). Each symbolic database \( D_i \) guarantees that \( Q_i(D_i) \) satisfies the constraints annotated on \( Q_i \). Then, a Symbolic Database Integrator is used to integrate the SDBs. The integration algorithms are designed to minimize the number of symbolic tuples with contradicting constraints (e.g., \( t'_1 \) in Figure 2b) and the number of generated databases. Finally, we use the Data Instantiator of QAGen to instantiate each integrated SDB with concrete values. The major advantage of this architecture is that we can fully utilize the capability of SQP in processing a variety of SQL queries. The Execution Planner is designed for integrating multiple SDBs and we defer its discussion until Section 3.2.2.

3.2 Symbolic Database Integration

We begin with the discussion of integrating two symbolic relations (with the same table definitions) that are separately generated by the SQP engine for two annotated queries. We discuss the integration of multiple symbolic relations in the end of this subsection and the integration of multiple symbolic databases in Section 3.2.2.

We use the annotated SQL queries \( Q_1 \) and \( Q_2 \) in Figures 4a and 4c as the running example. For ease of exposition, both \( Q_2 \) and \( Q_4 \) are simple selection queries posed on table \( S \). Figures 4b and 4d show the corresponding symbolic databases \( D_1 \) and \( D_2 \) that are generated by the SQP engine for \( Q_1 \) and \( Q_2 \). When only two symbolic relations are involved, the major challenge for the symbolic data integrator is to minimize the number of symbolic tuples with contradicting constraints. In other words, the integrator cannot simply merge \( t_1 \) with \( t_5 \), (i.e., treating symbols \( S_1 \) and \( S_4 \) as the same symbol and joining the constraints of \( t_1 \) and \( t_5 \) together to get \( S_1 \land S_1 \land S_1 \land S_1 \land S_1 \)), \( t_2 \) with \( t_6 \) and \( t_7 \) with \( t_8 \), and \( t_4 \) with \( t_9 \). Such a naive integration would result in an integrated symbolic database \( D \) as shown in Figure 5a. The problem with \( D \) is that many symbolic tuples are contradicting with each other: \( t'_5 \) induces a relationship \( \geq p_2 \geq p_1 \), but \( t'_1 \) and \( t'_2 \) induce a relationship \( \leq p_2 \leq p_1 \). As such, the integration algorithms should be designed to minimize the number of symbolic tuples with contradicting constraints in the
integrated SDB. For example, Figure 5b shows an ideal symbolic database that is integrated from $D_3$ and $D_4$ and does not contain any tuples with contradicting constraints.

To integrate two symbolic relations $S_r$ and $S_j$ (where $S_r$ and $S_j$ share the same table definition), we model the problem as a graph problem.

**Definition 1. (Constrained Node).** A node $n$ is constrained if it is associated with a propositional formula, $\phi_n$, composed of variables under a finite domain (SQL data types).

**Definition 2. (Satisfiable Edge).** An edge $e(u, v)$ is satisfiable if the conjunction of the propositional formula associated with constrained nodes $u$ and $v$ is satisfiable. That is, $\phi_u \land \phi_v$ is satisfiable.

As an example, consider an edge $e(u, v)$ connecting two constrained nodes $u$ and $v$. Assume $u$ is associated with a propositional formula $x > p$ and $v$ is associated with a propositional formula $x < q$, then $e$ is not satisfiable. On the contrary, if $u$ is associated with a propositional formula $x > p$ and $v$ is associated with a propositional formula $y < 10$, then $e$ is a satisfiable edge.

**Definition 3. (Constrained Bipartite Graph)** A graph $G = (U, V, E)$ with node sets $U$ and $V$ and edge set $E$ is a constrained bipartite graph (CBG) if $G$ is a bipartite graph, all nodes in $U$ and $V$ are constrained nodes, and all edges in $E$ are satisfiable edges.

Now, we can model a symbolic tuple $t_i$ of symbolic relation $S_i$ ($S_j$) as a constrained node $u_i$ ($v_j$) in a CBG $G$. For each pair of tuples $t_i \in S_i$ and $t_j \in S_j$, if the conjunction (of the constraints) of $t_i$ and $t_j$ is satisfiable (i.e., no contradiction), we add a satisfiable edge $e(u_i, v_j)$ to $G$. As a result, the two symbolic relations in Figures 4b and 4d can be modeled as a constrained bipartite graph $G_n$ shown in Figure 6a. Now, we can model the integration of $S_i$ and $S_j$ as finding a maximum satisfiable matching of a CBG.

**Definition 4. (Satisfiable Matching)** Given a constrained bipartite graph $G = (U, V, E)$, a matching $M$ is satisfiable if the conjunction of the propositional formulas associated with all constrained nodes in $M$ is satisfiable.

**Definition 5. (Maximum Satisfiable Matching)** Given a constrained bipartite graph $G = (U, V, E)$, a satisfiable matching $M$ is maximum satisfiable if the size of $M$ is largest among all satisfiable matchings in $G$.

The size of a maximum satisfiable matching (MSM) could be different from the size of a maximum matching. Figure 6b shows a maximum but not satisfiable matching $M_1$ of $G_5$. Edge $e_1$ in $M_1$ suggests that tuple $t_1$ of $D_3$ in Figure 4b should be integrated with tuple $t_2$ of $D_4$ shown in Figure 4d. Therefore, if the integration follows $M_1$, the resulting integrated database would become $D$ in Figure 5a. On the other hand, if the integration follows $M_2$ (see Figure 6c), which is a maximum satisfiable matching of $G_1$, the resulting integrated database would become the ideal integrated symbolic databases shown in Figure 5b.

We cast the problem of finding a MSM of a constrained bipartite graph as a decision problem:

**Definition 6. ($k$-SAT-MATCH Problem).** Given a constrained bipartite graph $G = (U, V, E)$ and an input integer $k$, the decision problem $k$-SAT-MATCH is to answer if there is a satisfiable matching $M$ of size that is at least $k$.

 Searching a maximum matching from a bipartite graph can be done in polynomial time. However, searching a maximum satisfiable matching from a CBG may require exponential time in the worst case (proof in Appendix A.1). The main difficulty lies in the requirement of “satisfiability” among the induced relationships of variables at run-time (e.g., in Figure 6b, adding edge $(t_1, t_2)$ to $M_1$ will induce a relationship that hinders adding edges $(t_4, t_7)$ and $(t_4, t_8)$ to $M_1$). This is also the main reason why applying SQP on a “processed” SDB online (mentioned in Section 3.1.1) is not a good idea. Nevertheless, we have many tricks to avoid the worst case in almost all circumstances. Specifically, we have developed a best-effort symbolic database integration algorithm that utilizes the special properties of SQP to reduce the search space. Our experiments show that $SI$ practically solves the problem and scales well under a variety of inputs.

### 3.2.1 The Symbolic Database Integration Algorithm

The symbolic database integration algorithm ($SI$) solves the maximum satisfiable matching by separating the induced relationship problem and the maximum matching problem. The main idea is as follows.

Given a constrained bipartite graph $G$ as input, (1) it first identifies all the total-order relationships that can be induced by the satisfiable edges and puts them in a set $R$. (2) For each possible subset $R_i$ of $R$, it constructs a new constrained bipartite graph $G_i$. $G_i$ includes the edges that induce total-order relationship(s) in $R_i$ and the edges that induce no total-order (edges that induce only partial-order relationships). (3) Find a maximum matching $M_i$ for each constructed bipartite graph $G_i$. (4) Finally, for all the maximum matchings found, follow (any) one that has the maximum matching size to perform tuple integration.

Assume that $SI$ takes $G$ (Figure 6a) as input. As a first step, a total-order relationship $r_1 = [p_2 > p_1]$ induced by edges $(t_1, t_3)$ and $(t_2, t_3)$ and a total-order relationship $r_2 = [p_2 < p_1]$ induced by edges $(t_4, t_6)$, $(t_4, t_7)$, $(t_4, t_8)$, $(t_4, t_9)$, $(t_4, t_10)$, and $(t_4, t_11)$ are added to $R$. Next, four constrained bipartite graphs $G_1$, $G_2$, $G_3$, and $G_4$ are constructed according to Step 2 above. Specifically, $G_1$ (shown in Figure 7a) includes the edges that induce the total-order $r_1$ and the edges that induce no total-order (e.g., $(t_1, t_5)$). $G_2$ (shown in Figure 7b) includes the edges that induce the total-order $r_2$ and the edges that induce no total-order. $G_3$ includes the edges that induce $r_1$ and $r_2$ and the edges that induce no total-order ($G_3$ is the same as the input graph). $G_4$ includes the edges that induce no relationships.

---

6Note that $G_n$ is not necessarily complete.

We do not show $G_4$ here for space reasons.
By following the basic idea illustrated above, \( SI \) has to search maximum matchings for \( 2^{|R|} \) constrained bipartite graphs. Although it looks a lot on the surface, \(|R|\) is actually a small number in practice. For example, in our experiments, the maximum values of \(|R|\) found in two real database applications and three TPC benchmarks (TPC-W, TPC-C, and TPC-H) are only 8, 6, 13, 24, and 17, respectively. Furthermore, we have incorporated four techniques into \( SI \) such that it actually visits at most 24 CBGs in all our experiments. We have also devised an approximation version of \( SI \) that runs in linear time. However, our experiments show that the exact version of \( SI \), in practice, scales well, and finds perfect matchings easily such that the most time consuming part is usually the preparation of SDBs using SQP. We put the approximation version of \( SI \) in Section 3.2.1.3.

**Algorithm SI**

**Trick 1.** Pruning CBGs that are constructed from contradicting relationships

The following lemma tells us that if a CBG \( G_t \) contains some contradicting relationships, \( SI \) can ignore \( G_t \) because there exists another CBG \( G_j \) with a larger MSM.

**Lemma 1.** Let \( r_j \) and \( r_k \) be two contradicting total-order relationships and let \( R_j, R_k, and R_h \) be three relationship sets. Assume \( \{r_j, r_k\} \subseteq R_j \cup R_k \cup R_h \). Let \( G_i, G_j, and G_k \) be the constrained bipartite graphs constructed from \( R_i, R_j, and R_k \), respectively. If \( M_i, M_j, \) and \( M_k \) are maximum satisfiable matchings of \( G_i, G_j, \) and \( G_k \), respectively, then \( |M_i| = \max(|M_j|, |M_k|) \).

**Proof.** Since \( r_j \) and \( r_k \) are contradicting, the maximum satisfiable matching \( M_j \) in \( G_j \) must not simultaneously contain edges inducing \( r_j \) and edges inducing \( r_k \). In other words, \( M_i \) must either be a maximum satisfiable matching in \( G_i \) or in \( G_j \), so that either \( |M_j| = |M_i| \) or \( |M_j| = |M_k| \). Since \( |M_i| \) is maximized, it follows that \( |M_i| = \max(|M_i|, |M_j|) \).

In our example, by Lemma 1, \( SI \) does not need to consider \( G_2 \) because \( r_2 \) and \( r_3 \) are contradicting and therefore the size of the MSM of \( G_2 \) would not be larger than the size of both the MSM of \( G_1 \) and the MSM of \( G_2 \).

**Trick 2.** Compressing the problem instances

\( SI \)'s efficiency can be further improved by compressing the symbolic tuples. For instance, in Figure 7a, tuples \( t_1 \) and \( t_2 \) are capturing the same selection predicate \( S.A > :p1 \) of query \( Q_3 \). Therefore, they are compressed into a single node. A maximum matching problem is often transformed into a maximum flow problem in network optimization [4, 20]. Here, \( SI \) compresses the input constrained bipartite graph \( G = (U, V, E) \) into a constrained flow network \( G' = (U', V', E') \):

i. (Build node sets) for every group of nodes \( N_u \) in \( U \) that captures the same predicate, add a node \( n'_{u} \) in \( U' \); similarly for \( V' \).

ii. (Build edge set) add an edge between \( n'_{u} \) and \( n'_{v} \) if there was an edge \((n_u, n_v)\) in \( G \) where \( n_u \in N_u \) and \( n_v \in N_v \).

iii. (Connecting source and sink) add an edge between source \( s' \) and \( n'_{u} \), and an edge between \( n'_{v} \) and sink \( t' \).

iv. (Calculate edge capacities) for edges of the form \( s', n'_{u} \), the capacity is set to \( |N_u| \); for edges of the form \( n'_{u}, t' \), the capacity is set to \( |N_u| \); for edges of the form \( n'_{v}, n'_{v}' \), the capacity is set to \( \min(|N_u|, |N_v|) \).

Figure 8a and 8b show the flow networks compressed from Figures 7a and 7b, respectively. We can see that the number of nodes is only half of the original constrained bipartite graph. The compression scheme described above covers cases where the inputs are simple queries without grouping and joining (SQP performs symbol replacements in those cases). We present the detailed compression scheme in Section 3.2.1.1.

**Trick 3.** DFS and subset pruning

Consider Figure 9, which is a more complicated constrained bipartite graph \( G = (U, V, E) \), as an example. \( G \) represents an instance of integrating two symbolic relations after several rounds of integration, which often happens when multiple queries are input to the system (see Section 3.2.2 for details). For the time being, we focus on an MSM search for Figure 9.

In Figure 9, the set of edges induces the following set of total-order relationships \( R = \{r_1 = [p1 > p2], r_2 = [p2 \geq p1], r_3 = [p2 > p1]\} \). For example, edge \((t_1, t_2)\) induces a total order \([p1 > p2]\) and edge \((t_4, t_6)\) induces a total order \([p2 > p1]\). There are no total-orders induced from symbols connecting parameters \( p3 \) and \( p4 \). For the example, in Figure 9 we can visualize its 3\(^2\) cases (all possible subsets) as a search tree (Figure 10). The left branch of the search tree denotes the inclusion of a relationship and the right branch of the search tree denotes the exclusion of a
relationship. As an example, leaf node 5 represents the case that we
need to construct a constrained bipartite graph by including edges
that induce relationship \(r_3\mid \{p2 \geq p1\}\) (e.g., \((t_4, t_6)\)), edges that
induce relationship \(r_3\mid \{p2 \geq p1\}\) (e.g., \((t_4, t_6)\)), and edges that
induce no total-order (e.g., \((t_4, t_6)\)). Looking at Figure 10, we see
that Lemma 1 prunes cases 1, 2, and 3, as those cases include edges
from contradicting relationships (\(r_1\) contradicts both \(r_2\) and \(r_3\)).

\(SI\) traverses the search tree in a depth-first manner because the
order of node traversal helps prune the search space by the fol-
lowing lemma.

**Lemma 2.** Given two non-empty relationship subsets \(\{R_i, R_j\} \subseteq \mathcal{R}\), if \(R_i \subseteq R_j\), the size of the MSM \(M_i\), of the constrained bipartite
graph constructed from \(R_i\), must be less than or equal to the
size of the MSM \(M_j\), of the constrained bipartite graph constructed
from \(R_j\) (i.e., \(|M_i| \leq |M_j|\)).

**Proof.** Since \(R_i \subseteq R_j\), the edges of \(M_i\) are all included in the
constrained bipartite graph \(G_j\) constructed from \(R_j\), so that \(M_i\) is
a satisfiable matching in \(G_j\). On the other hand, \(M_j\) is a maximum
satisfiable matching in \(G_j\), so we must have \(|M_i| \leq |M_j|\).

By Lemma 2, \(SI\) can prune cases 6, 7, and 8 because the MSM
obtained from these cases cannot be larger than the MSM obtained
from case 5. Up to this point, \(SI\) needs to consider only cases 4 and 5.

**Trick 4. Early Stopping.** Our goal is to find the largest MSM
among all the possible CBGs. The last trick is, if \(SI\) finds a perfect
satisfiable matching in a CBG, it can stop early. Although simple,
examples show that this trick is very useful since \(SI\) is often
able to find a perfect satisfiable matching very early in the process.

Algorithm 1 presents the pseudo-code of \(SI\). We have gone
through Steps (1) to (3) in the main discussion. As the relation-
ships are unweighted, so the search tree is constructed randomly.
In terms of implementation, Steps 4(a) and 4(b) are merged so that
we construct the flow network from the symbolic relations directly.
To implement Step 4(c), we use a push-relabel maximum flow al-
gorithm with complexity \(O(n^3)\) \([11]\) (\(n\) is the number of nodes
in the flow network). To implement Step 4(d), for each edge of
the form \((n'_a, n'_b)\) in the network flow \(G'_i\) with flow value \(f\), \(SI\)
matches \(f\) members of \(N_0\) to \(f\) members of \(N_0\). Finally, in Step
5, \(SI\) follows the largest maximum satisfiable matching that it has
found to perform tuple merging.

**Algorithm 1 SI**

(1) Identifies all the total-order relationships that can be induced by the
satisfiable edges and puts them in a set \(\mathcal{R}\).

(2) Construct a search tree \(T\) for each subset \(R_i\) of \(\mathcal{R}\) that
(i) contains no contradicting relationships and
(ii) \(R_i\) is not a subset of another subset \(R_j\).

(3) Initialize MAX-MSM=\(\emptyset\) to store the largest MSM discovered so far.

(4) Follow the search tree \(T\) in a depth-first order.
(a) construct a new constrained bipartite graph \(G_i\) which includes
(i) the edges that induce the relationships in \(R_i\)
appear as a left branch of a node (inclusion); and
(ii) the edges that induce no total-order (edges that induce only
partial-order relationships):
(b) transform \(G_i\) into its flow network counterpart \(G'_i\);
(c) find a maximum flow \(M'_i\) from \(G'_i\) by invoking a maximum
flow algorithm;
(d) transform the resulting maximum flow into maximum match-
ing MSM;
if a perfect satisfiable matching \(M\) is found, stop searching.
if \(|MSM| > \text{size-of MAX-MSM}\),
set \(\text{MAX-MSM} = \text{MSM}\).

(5) Follow MAX-MSM to perform the integration.

**Lemma 3.** Given a CBG \(G\), algorithm \(SI\) returns a maximum
satisfiable matching of \(G\) correctly.

**Proof.** If no pruning occurs, all relationship subsets with no
contradicting total-order relationships will be examined as in the
algorithm \(SI\), so that the matching reported in the end (which is
the one whose size is largest among all maximum matchings) must be
a maximum satisfiable matching of \(G\). \(\square\)

### 3.2.1 Detailed Compression Scheme.

We now discuss in more detail the compression scheme when the
queries involve grouping and joining. In SQP, two identical
symbols shall be instantiated by the same value. So, the group-
ing operator and the joining operator are designed to replace the
symbols. For example, Figure 11b shows a symbolic relation after
SQP, where the query contains a selection > :p1 which expects 6
tuples pass the filter and those tuples are grouped into two groups.
The original symbolic relation before SQP is shown in Figure 11a.
To control that only two groups are returned, the grouping oper-
ator replaces symbols \(s_a2, s_a3\) by symbol \(s_a1\), symbols \(s_a5, s_a6\)
by symbol \(s_a4\). (Other replacement scheme is doable, and that de-
pends on how the user wants the value distribution looks like; see
\([7]\) for details).

With symbol replacement, merging tuples in \(SI\) (Trick 2) shall
do not only base on the predicate, but also base on the symbol names.
For example, tuples \(t_1, t_2, \) and \(t_3\) shall be compressed into one
node, and tuples \(t_4, t_5, \) and \(t_6\) shall be compressed into another
node. Furthermore, to distinguish between compressing duplicated
symbols (e.g., the three \(s_a1\’s\) in Figure 11b) and original symbols
(e.g., \(s_b1, s_b2,\) and \(s_b3\) in Figure 11b), we name nodes obtained by
compressing duplicate symbols as atomic nodes and name the others
as normal nodes. In Figure 11b, after compression, we obtain
three compressed nodes with edge weights 3, 3, and 2, respectively,
as shown in Figure 11c.

Figure 12a shows a flow network which involves atomic com-
pressed nodes because of the existence of group-by or join oper-
We now discuss how to integrate multiple symbolic databases when each database is independently generated from a single input annotated query by SQP. Intuitively, to integrate $n$ symbolic relations $S_1, S_2, \ldots, S_n$ (which share the same table definition and are generated by SQP for $n$ queries), we can model the problem as finding an MSM of a constrained $n$-partite graph; however, that problem is obviously too difficult to be solved while maintaining both a good running time and a good matching size. Therefore, our method of integrating multiple symbolic databases resembles the concept of joining.

We use $SI(D_i, D_j)$ denote the integration of two SDBs $D_i$ and $D_j$, and use $D_k$ to denote the resulting SDB. The integration of three SDBs, $D_i, D_j,$ and $D_k,$ can then be achieved by one of two integration plans, either $SI(SI(D_i, D_j), D_k)$ or $SI(D_i, SI(D_j, D_k))$. 

### 3.2.1.3 Approximation version of $SI$.

Given a constrained bipartite graph $G = (U, V, E)$, we now describe an approximation algorithm for the MSM problem. For simplicity, we first assume that each edge in $E$ induces at most one total order relationship. Then, we show that our proposed algorithm will always return a satisfiable matching, whose size will be at least half of an MSM.

The proposed algorithm has the following steps.

1. Find any maximum matching $M$ in $G$.
2. Let $p_1, p_2, \ldots, p_k$ be the $k$ parameters involved in the induced total order relationships.
3. Select a $k$-permutation $\sigma: [1, k] \rightarrow [1, k]$ uniformly at random.
4. Order the parameters by $p_{\sigma(1)} < p_{\sigma(2)} < \ldots < p_{\sigma(k)}$.
5. Output all edges in $M$ whose induced relationships do not violate the above ordering.

**Theorem 1.** The above randomized algorithm returns a satisfiable matching, whose expected size is at least half of an MSM.

**Proof.** It is trivial that the edges returned form a satisfiable matching. For each edge in $M$, the probability that it is output is at least $1/2$, so that the expected size of the returned matching is $|M|/2$. Since the size of $M$ is at least the size of any MSM, the theorem thus follows.

We can de-randomize the above algorithm by the standard method of conditional expectations. Consequently, we have the following:

**Theorem 2.** There is a deterministic polynomial time algorithm that reports a satisfiable matching, whose expected size is at least half of an MSM.

In general, if each edge may induce more than 1 total order relationship, the above theorem is refined as follows:

**Theorem 3.** Let $M$ be a maximum matching of the constrained bipartite graph $G$, and let $k$ be the number of parameters in the induced total order relationships. Let $pr_j$ denote the probability that the $j$th edge in $M$ is satisfied, when the ordering of the parameters is selected uniformly at random. Then there is a deterministic polynomial-time algorithm that returns a satisfiable matching, whose expected size is $\sum_{j=1}^{|M|} pr_j$.

### 3.2.2 Multiple Queries

3.2.1.2 Multiple Attributes and Multiple Tables.

Generalizing $SI$ to handle multiple attributes is straightforward. In case a tuple contains multiple attributes, a single node is created for the conjunction of all the constraints in the attributes. In fact, Figure 9 is an example of said idea. Integrating two SDBs that contain more than one pair of symbolic relations is also straightforward. We can simply apply $SI$ on every pair of overlapping symbolic relations.
we first scale down the cardinalities requirements of the input
is set to the largest possible integer. To optimize this process,
mon table, that size of the MSM between a pair of SDBs
specifically, our approach is to first pre-build a summary about the
size of the MSM between a pair of SDBs
D


tation borrows ideas from there. Specifically, in traditional query
using some efficient algorithm. For WAGEN, we pre-build a sum-
agrams) on the data and exploit those to estimate the best pl
results in a database
D


Recall that given the SDBs of
annotated queries, our goal is to integrate the
SDBs into as few databases as possible. As the
MSM returned by an SI operation may not be a perfect matching,
the size of the MSM may get smaller and smaller when the inte-
gregation goes up to the root. In order to ensure the matching size, or
the quality, of an integrated database at a particular level of integra-
tion is acceptable, WAGEN stops integrating two SDBs when the
quality of an SI operation drops below the user-defined-threshold.
Since the size of an MSM is not readily known to the users, we
define the quality threshold (from the user perspective) as the rela-
tive error between the annotated cardinality and the actual cardinal-
ity (obtained by posing the query on the generated data). Consider
Figure 13a again. Assume that after
SI
(D1, D2),
SI
(D12, D3)
results in a database
D
123
in which posing a query (e.g.,
Q
2
) on it
finds some query operator with relative error exceeding the thresh-
old. Then, WAGEN will not further integrate
D
123
with
D
4
In-
stead, it discards
D
123
and integrates
D
3
with
D
4
and so on (see
Figure 13b). In the example, two databases
D
12
and
D
45
are gen-
erated to serve five queries.

3.2.2.1 Determining a good integration plan.
Since there are an exponential number of possible integration
plans, deducing an optimal one that returns a minimum set of databases,
which have the lowest error, is a challenging problem. In fact, it is
as hard as finding the optimal joining plan [12], which is \( NP \)-hard (see Appendix A.2). Traditional query optimization uses heuristics and estimation to solve the join plan selection problem. Our solu-
tion borrows ideas from there. Specifically, in traditional query
optimization, we usually pre-build certain summaries (e.g., his-
tograms) on the data and exploit those to estimate the best plan
using some efficient algorithm. For WAGEN, we pre-build a sum-
mary about the quality of some core SI operations and exploit that
to estimate the best plan using a simple graph algorithm. More
specifically, our approach is to first pre-build a summary about the
sizes of the MSM between every pair of input SDBs. To obtain the
size of the MSM between a pair of SDBs
D
i
and
D
j
, we have to carry out
SI
(D
i
, D
j
). Given
n
annotated queries (thus
n
SDBs), we have to execute \( C \) \( n \) \( 2 \) SI operations (if
SI
(D
i
, D
j
) has no com-
mon table, that
SI
is skipped and the MSM between
D
i
and
D
j
is set to the largest possible integer). To optimize this process,
we first scale down the the cardinalities requirements of the input
queries (e.g., from generating 1GB data to 1MB data) by a tool we
developed in [22]. For example, the input query in Figure 4a
can be automatically scaled-down to have table
S
annotated with
umples and the output annotated with one tuple (the tool will
make sure the scaling is meaningful and in proportion). This scale
down optimization is built upon the observation that (1) the num-
ber of total order relationships and more importantly (2) the ratio
between MSM size and the CBG size depend on the characteristics
of the input queries (e.g., the selection predicates) but not the size
of the databases to be generated. Thus, there would be no differ-
ce in (1), (2), and thus the number of resulting databases between
generating 1MB and 1GB data (our experiments confirmed this).
However, there would be a significant time difference between the
two. More specifically, the running time of an SI operation mainly
consists of: (T1) scanning the SDBs and constructing the flow net-
work, (T2) running the maximum flow algorithm, (T3) loading and
merging tuples according to the MSM and inserting them into a
new SDB, and (T4) the algorithmic overhead (e.g., checking con-
tradiciting total-orders). Using the four SI tricks, experiments show
that (T1) often is the most time consuming step because QAGen
(and thus WAGEN) store the symbolic/instantiated tuples in a Post-
greSQL (the data is usually too large to fit in memory). Therefore,
much time is spent on the overhead (e.g., JDBC) of reading sym-
bolinc tuples from the database. By running SI operations on the
scaled-down SDBs instead, we can obtain the summary about the
(proportionally scaled-down) sizes of the MSM between every pair
of SDBs (at the leaf level) more efficiently. As a note, this sum-
mary can be obtained efficiently because it is independent of the
annotated data size and operator cardinalities.
The summary obtained is represented as a graph. In the graph,
a node denotes an SDB, an edge denotes an SI operation between
a pair of SDBs
D
i
and
D
j
, and the edge weight denotes the MSM
size between
D
i
and
D
j
. Figure 14a shows an example of such a graph
for the five SDBs. Recall that, one additional database is
required whenever the quality of the resulting instantiated database
exceeds the user-threshold. Actually, that is directly related to the
size of the MSM obtained from each SI operation. Since our goal
is to minimize the number of generated databases, the best plan
should be the one that maximizes the MSM size of each SI opera-
tion. Therefore, we suggest that the best integration plan should be
derived from the Maximum Spanning Tree (MST) of the graph.
In Figure 14a, the MST of the graph is highlighted. Based on the
MST, the suggested integration plan (Figure 14b) follows a de-
creasingly ordered, by the edge weight (the MSM size), sequence of
the MST, i.e.,
SI
(SI
(SI
(D
2
, D
1
), D
3
), D
4
), D
5
).

Similar to any query plan selection algorithm, our approach is
also based on heuristics and estimation, which may not find the op-
timal plan. Nevertheless, experimental results about this approach
are quite encouraging. In our case studies on two real database ap-
plications, the plans suggested by our method are perfect—we can
generate a single database that serve all application queries with
perfect cardinality guarantees. In our experiments on TPC-C, TPC-
W, and TPC-H benchmarks, the plans suggested by our method successfully integrate all databases into only two databases.

3.2.3 Summary of the Methodology

Overall, the execution of WAGEN is composed of two steps:

Step A. Finding a good integration plan. This involves:

- $A_1$ Scale down the cardinalities in the input queries. This is done by our tool in [22] using negligible time.
- $A_2$ SQP the scaled down input queries to get the small SDBs. This step is done once by the SQP engine in [7] for each input query.\(^8\)
- $A_3$ Build a summary (graph) of MSM size by running SI on every pair of small SDBs.
- $A_4$ Suggest a plan $P$ by finding a Maximum Spanning Tree from the graph. We can use any Minimum Spanning Tree algorithm. In our implementation, we used Kruskal’s algorithm [21], which runs in $O(n^2 \log n)$ time. As $n$ is generally a small number for typical database applications (e.g., a TPC-W implementation has only about 20 parameterized queries), this step runs very fast.

Step B. Executing $P$ in the original scale. This involves:

- $B_1$ SQP the input queries to get the SDBs in original scale. This step is done once by the SQP engine in [7] for each input query.\(^8\)
- $B_2$ Run $n-1$ SI operations according to $P$.
  - a) After each SI, instantiate the resulting SDB.
  - b) Pose all the queries on the resulting database to check the quality, i.e., the relative error between the actual cardinality and the annotated cardinality of each operator. Add a new database if necessary.

4. CASE STUDIES AND EXPERIMENTS

We have carried out case studies on two real database applications and experiments on three TPC benchmarks (TPC-W, TPC-C, and TPC-H). QAGen uses PostgreSQL to manage the symbolic/instantiated database and uses Java to implement the SQP operations. For easy interacting with QAGen’s components, we also use Java and PostgreSQL to implement WAGEN. All experiments were carried out on a Pentium Dual-Core 2.5GHz PC with 8GB memory running Ubuntu. In all experiments, we set the relative error tolerance to be 100% for cardinalities in range $[1, 1000]$ (e.g., the acceptable range of cardinality 10 is $[1, 20]$; cardinality 0 is excluded), 10% for cardinalities in range $[1001, 10000]$ (e.g., the acceptable range of cardinality 5000 is $[4500, 5500]$), and 1% for cardinalities $> 10000$. In all experiments, we exclude UD (INSERT, UPDATE, and DELETE) SQL queries. We also exclude “independent” queries that share no common tables with the others (e.g., a SELECT query that accesses a table X is removed from consideration if no other SELECT queries also access X). That is because the symbolic databases generated for independent queries can be “perfectly integrated” with other SDBs without any effort.

We characterize the efficiency of WAGEN based on the items listed in Section 3.2.3. The quality of the generated databases is characterized by the error between the annotated and actual cardinalities of all queries and their sub-queries.

4.1 Case study — A security monitoring system

This real database application is a security monitoring system that records the staff and guests access in a company campus. Our goal is to generate a minimal number of databases to simulate the workload of this real application. Eight queries remained after removing IUD and independent queries. The details of the annotated queries are presented in Table 1. For queries having more than two operators (e.g., Query Q4 has selection and group-by operations), we report the intermediate cardinalities as well. For example, in the original data, Q4 returns 13 tuples. If discarding the group-by operation, Q4 returns 265 tuples. The scaled-down and scale-up cardinalities are calculated by the tool in [22] with scale factors 0.5 and 10, respectively. The semantics of the queries are self-explanatory. The queries mainly access the following tables (irrelevant attributes are omitted):

<table>
<thead>
<tr>
<th>Item Description / Sub-item</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$ scale-down the input queries</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>$A_2$ SQP all down-scaled queries once</td>
<td>9min30s</td>
</tr>
<tr>
<td>$A_3$ Build a summary (graph) of MSM size</td>
<td>8s</td>
</tr>
<tr>
<td>$A_4$ Suggest a plan $P$ by finding a MST from the graph</td>
<td>0.5s</td>
</tr>
</tbody>
</table>

Running SQP on all scaled-down input queries to generate the small SDBs ($A_2$) is more time consuming than the other parts ($A_1$, $A_3$, $A_4$). This makes sense because query-aware data generation is much more advanced than traditional query-unaware data generation technology and thus requires time to process symbolic data. The time spent on running SI on $C_2$ pairs of SDBs ($A_3$) takes 83s. The following table shows the experimental result for step B, i.e., executing the good integration plan in the original scale:

<table>
<thead>
<tr>
<th>Item Description / Sub-item</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_5$ all queries in original scale</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>$A_6$ Follow P to run SI operations</td>
<td>8s</td>
</tr>
<tr>
<td>$A_7$ After each SI, instantiate the resulting SDB</td>
<td>4min19s</td>
</tr>
<tr>
<td>$A_8$ Pose all the queries on the resulting database to check quality</td>
<td>0.41s</td>
</tr>
</tbody>
</table>

By following the suggested plan, all SDBs in the original scale are integrated into one. Although $SI$ is the key to minimizing the databases, the time spent on running the seven $SI$ operations (because this workload has eight queries) is only 8 seconds. The overall running time, $\sum_{A_i} + \sum_{H_i}$, is 57 minutes. When posing the original queries on the generated database, all queries obtain exact cardinalities as annotated in the input (i.e., 0 error).

Another experiment we have carried out is to study the efficiency of $SI$ and the pruning effectiveness of the tricks used by $SI$. The efficiency of the $SI$ algorithm can be studied through the four items...
(T1 to T4) we mentioned in Section 3.2.2. The effectiveness of an SI can be characterized by (i) the compression ratio: the total number of tuples (E1) vs. the total number of nodes in the flow network (E2); and (ii) the pruning effectiveness: the number of possible cases (E3), the number of cases actually examined (E4), and the number of cases pruned by Lemmas 1 and 2 (E5). In this experiment, we used the best plan found in the case study (i.e., step A) and measured the total values of the aforementioned items of all seven executed SI operations. We also repeated this experiment using annotated queries scaled down by 0.5 and also scaled up to 10 times. Figure 15 presents the results. We can see that T1 is the dominating step because it reads a large number of tuples from the SDBs. Thanks to compression, the number of nodes (E2) is significantly smaller than the original data (E1). Thus, the number of flow running time (T2), the time to integrate and instantiate the compressed tuples (T3), and the algorithmic overhead (T4) are small. As this workload is fairly simple, most SI operations involved very few unique relationships and the number of unique relationships is the same across different scales. The last SI operation involved 8 distinct relationships and thus, there were 256 possible cases (E3). However, only 3 cases were examined (E4). That is because 66 cases were pruned in the process (E5). After examining 3 cases, SI found a perfect matching and stopped early. In some SI operations (e.g., SI(D4, D1)), there were no common tables and SI thus examined 0 cases. Overall, we can see that the SI algorithm running time ∑ TSI scales linearly with the size of the generated data. In this workload, the number of generated databases remains constant when the requested data size increases. We verify the quality of the generated databases by posing the original queries on the generated database. The results show that all the resulting cardinalities perfectly match the ones in the annotated queries in all scales.

To study the performance of the overall methodology with respect to workloads of different sizes, we carried out an experiment that varies the number of annotated queries in the input. The table below summarizes the results (time is in seconds):

<table>
<thead>
<tr>
<th>Num. of Queries</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>∑ TSI</td>
<td>124</td>
<td>222</td>
<td>206</td>
<td>172</td>
<td>65</td>
<td>37</td>
<td>25</td>
</tr>
<tr>
<td>Num. of DB</td>
<td>348</td>
<td>492</td>
<td>546</td>
<td>590</td>
<td>634</td>
<td>678</td>
<td>722</td>
</tr>
</tbody>
</table>

The running time roughly scales linearly to the number of input queries. All the generated databases are perfect (i.e., no error).

4.2 Case study — A POS system

The second database application is a reporting system that built on top of a sales database. The database consists of about two million records. After removing 4 independent queries and 14 IUD queries, 10 queries remained. Again, we executed the twelve queries on the real data (with parameter values found in the real workloads) to prepare the annotated queries in the original scale. The details of the input queries are presented in Table 2. The queries mainly access the following tables (irrelevant attributes are omitted):

<table>
<thead>
<tr>
<th>Item Description/Sub-item</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>scale-down the input queries</td>
</tr>
<tr>
<td>A2</td>
<td>SQP all down-scaled queries once</td>
</tr>
<tr>
<td>A3</td>
<td>Build a summary (graph) of SDB size</td>
</tr>
<tr>
<td>A4</td>
<td>Suggest a plan P by finding a MST from the graph</td>
</tr>
</tbody>
</table>

Running SQP on all scaled-down input queries to generate the small SDBs (As) takes more time consuming than the other parts (A1, A3, A4). The time spent on running SI on C1.02 pairs of SDBs (As) takes (E6). The following table shows the result for step B, i.e., executing the good integration plan in the original scale:

<table>
<thead>
<tr>
<th>Item Description/Sub-item</th>
<th>Total Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>SQP all queries in original scale</td>
</tr>
<tr>
<td>B2</td>
<td>Follow P to run SI operations</td>
</tr>
<tr>
<td></td>
<td>(a) After each SI, instantiate the resulting SDB</td>
</tr>
<tr>
<td></td>
<td>(b) Pose all the queries on the resulting database to check quality</td>
</tr>
</tbody>
</table>

By following the suggested plan, all SDBs in the original scale are integrated into one. Although SI is the key to minimizing the databases, the time spent on running the seven SI operations (because this workload has eight queries) is only 9 minutes. The overall running time, ∑ TSI + ∑ TSI, is about 5 hours. When posing the original queries on the generated database, all queries obtain exact cardinalities as annotated in the input (i.e., 0 error).

We have also carried out experiments to study the efficiency of SI and the pruning effectiveness of the tricks used by SI. Figure 16 presents the results of the efficiency study. We can see that T1 is still the dominating step because it reads a large number of tuples from the SDBs. Since many more tuples are involved, the overhead on E3 is also quite significant. Overall, we can see that the SI algorithm running time ∑ TSI scales linearly with the size of the generated data. Since there are many tables involved in this workload, we give a summary of the effectiveness of SI instead of the full results for brevity: the maximum number of distinct relationships involves in this workload is 6, leading to a maximum of 64 cases to be examined. Same as the other experiments, the pruning effectiveness of SI is also very good. In all SI, only 1 to 2 cases were actually examined. In this workload, the number of generated databases remains constant when the requested data size increases. We verify the quality of the generated databases by posing the original queries on the generated database. The results show that all the resulting cardinalities perfectly match the ones in the annotated queries in all scales.

To study the performance of the overall methodology with respect to workloads of different sizes, we carried out an experiment that varies the number of annotated queries in the input. The table below summarizes the results (time is in seconds):
The running time roughly scales linearly to the number of input queries. All the generated databases are perfect (i.e., no error).

4.3 Experimental Result: TPC-C Benchmark

The TPC-C benchmark models a typical OLTP environment where users executes transactions against a database. We downloaded an open-source implementation of TPC-C from http://db.apache.org/daisy/index.html. For space reasons, we refer readers to the TPC-C specification for the details of the queries (we name the TPC-C queries according to their appearance order in the specification). After removing IUD and independent queries, 16 TPC-C queries remained. The expected cardinalities annotated on the operators of the queries are specified according to the actual cardinalities obtained by running the queries on the TPC-C data with scale factor 5.0 (they are scaled-down to a scale factor of 1.0 during the plan search process). The breakdown of the whole plan search process is as follows:

The overall running time, \( \sum A + \sum B \), is 11 hours 53 minutes. By following the suggested plan, two databases \( D_s \) and \( D_b \) were generated. When posing the original queries on the generated database, all queries obtain exact cardinalities as annotated in the input.

The following table shows the experimental result about step \( B \), i.e., executing the good integration plan in scale 5.0:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description/Sub-item</th>
<th>Total Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D_s )</td>
<td>SQP all queries in original scale ( 3 )</td>
<td>9.5 min</td>
</tr>
<tr>
<td>( A_4 )</td>
<td>Build a summary graph of ( B )</td>
<td>40.5 s</td>
</tr>
</tbody>
</table>

The overall running time, \( \sum A + \sum B \), is 11 hours 53 minutes. By following the suggested plan, two databases \( D_s \) and \( D_b \) were generated. When posing the original queries on the generated database, all queries obtain exact cardinalities as annotated in the input.

Figure 17 presents the time breakdown of all executed \( S_I \) operations in different scale factors (1.0, 5.0, and 10.0), using the best plan found in part A. Most items behave the same as the two real workloads. Under this bigger workload, we can see that the suggested plan favors the integration of SDBs without any common table (as their MSMs are set to be the largest possible integer in those cases), so the first few \( S_I \) operations do not integrate anything. Slightly different from the two real database applications, the time spent on the \( S_I \) algorithm (\( T_4 \)) dominates the overall running time. When we look at the number of cases (E3), we quickly find out that is related to the current implementation (not algorithmic issue) of WAGEN. Specifically, this TPC-C workload has one \( S_I \) operation that needs to deal with 24 distinct relationship, leading to 16+ million cases. That should not be an issue originally because only two cases (E4) were actually visited after a large number of cases were pruned (E5). However, the current WAGEN implementation is implemented in Java and uses an external constraint solver called Cogent [13], which is a C++ binary executable, to check Lemma 1. For each cross-language Cogent call, it took about 0.3s overhead (by JNI). All together there were 49632 Cogent calls (same for all scales; because that depends on the number of distinct relationships, not the data size). So, those calls used a total of 14850 seconds, which almost equals to time \( T_4 \). Indeed we tried to use some constraint solvers written in Java in our implementation and that bottleneck disappeared (the bottleneck is back to \( T_1 \)). However, we found that, in general, Java constraint solvers are not very stable. As an experimental prototype, we keep Cogent in our current implementation because it is more stable (most constraint solvers are written in C++).
it can work seamlessly with Cogent). As a side note, the above also explains why item $B_2$ becomes the bottleneck in part $B$. That is due to the large overhead spent on calling a non-Java external binary. Therefore, if we find a stable Java constraint solver, the running time of $B$ can be reduced by 4 hours.

Other than the above implementation issue, this experiment draws similar conclusions as the two real workload experiments. The running time scales linearly to the workload size. The experimental result about varying the number of annotated queries in the input is summarized in Figure 18. Overall, the running time scales roughly linearly to the number of input queries. All the generated databases are perfect (i.e., no error). When the 9-th query was added, one more database was required. As a note, this experiment could only be carried out by adding TPC-C queries. The number of queries cannot be scaled up even higher by adding randomly generated queries because they often return empty results (i.e., cardinality equals 0 in the output operator). Furthermore, TPC-C, TPC-W and TPC-H benchmarks are simulating realistic workloads, we believe that the number of queries we used is enough to reflect realistic applications.

### 4.4 Experimental Result: TPC-W Benchmark

The TPC-W benchmark models a typical web-commerce database application. We downloaded an open-source implementation of TPC-W from http://www.ece.wisc.edu/~pharm/tpcw.shtml. For space reasons, we refer readers to the TPC-W specification for the details of the queries (we name the TPC-W queries according to their appearance order in the specification). After removing IUD and independent queries, 15 TPC-W queries remained. The expected cardinalities annotated on the operators of the queries are specified according to the actual cardinalities obtained by running the queries on the TPC-W data with scale factor 10.0 (they are scaled-down to become a scale factor of 1.0 during the plan search process). The breakdown of the whole plan search process is as follows:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description / Sub-item</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>scale-down the input queries</td>
<td>$&lt;1s$</td>
</tr>
<tr>
<td>$A_2$</td>
<td>SQP all scale-down queries once</td>
<td>6min</td>
</tr>
<tr>
<td>$A_3$</td>
<td>Build a summary (graph) of MEM size</td>
<td>150s</td>
</tr>
<tr>
<td>$A_4$</td>
<td>Suggest a plan $P$ by finding a MST from the graph</td>
<td>$0.383s$</td>
</tr>
</tbody>
</table>

Again, the most time consuming part is running SQP on all scaled-down input queries in order to generate the small SDBs ($A_2$). The time spent on running $SI$ on $C_{17}^3$ pairs of SDBs ($A_3$) is 180s.

The following table shows the experimental result about step $B$, i.e., executing the good integration plan in scale 10.0:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description / Sub-item</th>
<th>Total Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1$</td>
<td>SQP all queries in original scale</td>
<td>$75min$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>Follow $P$ for run $SI$ operations (a) After each $SI$, instantiate the resulting SDB (b) Pose all the queries on the resulting database to check quality</td>
<td>$6min$</td>
</tr>
</tbody>
</table>

The overall running time, $\sum_{a} + \sum_{b}$, is 1 hour 55 minutes. By following the suggested plan, two databases $D_d$ and $D_s$ were generated. Figure 20 presents the time breakdown of all executed $SI$ operations in different scale factors (1.0, 10.0, and 100.0). Queries in TPC-W are more simple than TPC-C, the maximum number of distinct relationship is 13. As the number of cases is smaller than that in TPC-C, the overhead spent on calling the external constraint solver is less significant. Figure 19 shows that increasing the number of queries increases the running time linearly. When the 6-th query was added, one more database was required. Note that one of the two generated databases does not have perfect quality, the relative error of query Q15, when posed on the instantiated database $D_s$ (see Figure 20), got 90% and 98% cardinality errors on two operators (cardinalities in ranges [1, 1000]), respectively. Other queries got perfect cardinality guarantees.

### 4.5 Experimental Result: TPC-H Benchmark

The TPC-H benchmark models a typical decision support system, which contains 22 highly complex queries. QAGen can support up to 14 TPC-H queries. Our experiments were carried out based on integrating the SDBs generated by SQP the 14 TPC-H queries. However, we are now in the process of commercializing WAGEN, so we do not present the full report here. Instead, we present a partial result which takes six TPC-H queries as input. The expected cardinalities annotated on the operators of the queries are specified according to the actual cardinalities obtained by running the queries on the TPC-H data with scale factor 0.1 (they are scaled-down to become a scale factor of 0.02 during the plan search process). The breakdown of the whole plan search process is as follows:
Again, the most time consuming part is running SQP on all scaled-down input queries to generate the small SDBs \( \langle A_2 \rangle \). The time spent on running \( SI \) on \( C_2 \) pairs of SDBs (\( \times A_2 \)) is 41s.

The following table shows the experimental result about step 8, i.e., executing the good integration plan in scale 0.1.

<table>
<thead>
<tr>
<th>Item Description / Sub-item</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{E}_3 ) scale-down the input queries</td>
<td>&lt;1s</td>
</tr>
<tr>
<td>( \mathcal{E}_4 ) SQP all down-scaled queries once ( ^2 )</td>
<td>3min</td>
</tr>
<tr>
<td>( \mathcal{E}_5 ) Build a summary (graph) of MSS size</td>
<td>41s</td>
</tr>
<tr>
<td>( \mathcal{E}_6 ) Suggest a plan ( P ) by finding a MST from the graph ( 0.5s )</td>
<td></td>
</tr>
</tbody>
</table>

\( \sum A = 4.8\text{min} \)

The overall running time, \( \sum A + \sum B_i \), is 32 minutes. By following the suggested plan, two databases \( D_a \) and \( D_d \) were generated (0 error). Figure 21 presents the time breakdown of all executed \( SI \) operations in different scale factors (0.02, 0.1, and 1.0). We can see that the number of compressed nodes is not a constant. That is because TPC-H queries are very complex and involve a lot of grouping and joining operations. Nevertheless, the number of nodes increases linearly. Figure 22 also shows that increasing the number of queries increases the running time linearly.

### 4.6 Evaluating the goodness of the suggested plans

In order to evaluate how good the plans suggested by our algorithms are, we have executed thousands to millions of extra randomly generated integration plans (using a cluster of machines) to get a picture of how the best plans for each workload could be. As there are an exponential number of possible plans, we have developed another algorithm to do this evaluation more efficiently. First of all, we restrict to only left-deep-join plans which are often found in those random trials to further improve our integration plan searching method. As such, the algorithm adds \( \langle D_2, D_4 \rangle \) as a new conflict and tries another random (but not repeated) plan \( P \) (since \( P \) cannot collect full AEUs). Consider \( P \) in Figure 23a as the last example. \( P \) is directly pruned by APS because the collected conflicts \( \langle D_1, D_2 \rangle \) and \( \langle D_1, D_3 \rangle \) tell us that this plan has at least three databases, which is worse than \( P \).

We have spent 10 days running APS on 10 machines. We have exhausted all the plans for the real workloads and tried about ten millions of plans for the TPC benchmarks. We found that the integration plans suggested by our method have the same number of generated databases (and 0 error) as the best plan we could find in millions of plans, except in the experiment using TPC-C benchmark, where we found a plan that integrated all databases into one with little error, which is better than the one suggested by our method. As another future work, we will exploit the best plan found in those random trials to further improve our integration plan searching method.

### 5. CONCLUSION

WAGEN is a workload-aware data generator that takes as input a set of queries and generates database instances for which the users can control the characteristics of the resulting workload. Applications of WAGEN include database testing, database application testing, and application-driven benchmarking. Although the whole data generation process requires solving several difficult problems, our experiments show that our proposed methods are able to practically solve them. WAGEN is able to generate a minimal number of databases for real applications and TPC workloads. Our future work will focus on further improving our plan search methods.

### 6. REFERENCES


APPENDIX

A. PROOFS

A.1 Proof of $k$-SAT-MATCH is $NP$-complete.

THEOREM 4. Problem $k$-SAT-MATCH is $NP$-complete.

We begin with proving $k$-SAT-MATCH is in $NP$, and further show that it is $NP$-hard by a reduction from the $NP$-complete problem known as X3C (Exact Cover by 3-set).

COROLLARY 1. $k$-SAT-MATCH is in $NP$.

PROOF. Each “yes” instance has a polynomial-size proof, which consists of the set of edges in the matching, and the set of values for each variable. Thus, each “yes” instance can be verified in polynomial time. □

COROLLARY 2. $k$-SAT-MATCH is $NP$-hard.

PROOF. Obviously, if we solely focus on the constraint satisfaction problem (i.e., the condition on satisfiability required in Definition 4), $k$-SAT-MATCH is definitely $NP$-hard. However, as we want to show the difficulty of the matching problem itself (e.g., adding an edge to the matching set will induce some relationships that hinder the matching of the other nodes), we assume here the constraint satisfaction step is at no cost.

We are going to reduce X3C (Exact Cover by 3-set) to the $k$-SAT-MATCH problem. The X3C problem [14] takes as input a set of elements $\delta = \{S_1, S_2, \ldots, S_n\}$ and a collection of 3-element set $C = \{C_1, C_2, \ldots, C_m\}$ and asks whether there is a sub-collection of $C$ whose size is $n$, such that it exactly covers all elements of $\delta$. The reduction is to construct a constrained bipartite graph $G = (U, V, E)$ as follows.

1. For each 3-element set $C_i = \{S_j, S_k, S_l\}$, insert 3 constrained nodes $u_{i,j}$, $u_{i,k}$, and $u_{i,l}$ to constrained node set $U$. The propositional formulas that are associated with $u_{i,j}$, $u_{i,k}$, and $u_{i,l}$ would be $\{S_j \leq w_i\}$, $\{S_k \leq w_i\}$ and $\{S_l \leq w_i\}$, respectively ($S_j$, $S_k$, $S_l$, $w_i$ are symbols and $w_i$ is any unique value).
2. For each element \( S_j \), insert a constrained node \( v_j \) to constrained node set \( V \). The propositional formula that is associated with \( v_i \) would be \([S_j \geq w] \) (value \( w \) would be the same for all elements).

3. Connect the nodes in \( U \) and \( V \) if they are created from the same element \( S_j \).

For instance, assume a 3-element set \( C_2 = \{S_4, S_5, S_6\} \) has inserted 3 nodes \( u_{2,4}, u_{2,5}, \) and \( u_{2,6} \) to \( U \) in Step 1 and element \( S_4 \) has inserted a node \( v_4 \) to \( V \) in Step 2. Then, nodes \( u_{2,4} \) and \( v_4 \) should be connected as both of them are created from element \( S_4 \).

4. For each 3-element set \( C_i \), insert a node \( u_{C_i} \) with propositional formula \([S_{C_i} \geq w_{C_i}] \) to \( U \) and insert a node \( v_{C_i} \) with propositional formula \([S_{C_i} \leq w_{C_i}] \) to \( V \) and connect the two nodes with an edge.

The rest of the proof will establish:

**Proposition 1.** There is an exact cover of \( S \) if and only if the size of maximum satisfiable matching of \( G \) is exactly \( 3n + (m - n) \).

Firstly, if the node \( u_{i,j} \) appears in the MSM, it must be matched with the node \( v_j \), so that it will induce the total-order relationship \( w_i \geq w \). On the other hand, if \( u_{C_i} \) appears in the MSM, it must be matched with \( v_{C_i} \), so that it will induce the total-order relationship \( w > w \). Thus, if either \( u_{i,j} \), \( u_{i,k} \), or \( u_{C_i} \) appear in the MSM, we cannot have \( u_{C_i} \) in the MSM at the same time.

Suppose we denote \( z \) to be the number of \( i \)'s such that \( u_{i,j} \), \( u_{i,k} \), or \( u_{C_i} \) appear in the MSM. Then, the size of MSM is at most \( 3z + (m - z) \), which in turn is at most \( 3n + (m - n) \) since \( z \leq n \).

The "only-if" direction. Next, suppose there is an exact cover of \( S \). In that case, let \( C_{i_1}, C_{i_2}, \ldots, C_{i_n} \) be the 3-sets such that they exactly cover \( S \). This implies the elements in these 3-sets must be distinct from each other. Then, consider the following matching in \( G \):

1. For each \( i \in \{i_1, i_2, \ldots, i_n\} \), the corresponding nodes of \( C_{i_1} \), i.e., \( u_{i_1,j}, u_{i_1,k}, u_{i_1,l} \), are matched to \( v_{i_1} \), \( v_{i_2} \), and \( v_{i_3} \), respectively.
2. For each \( i \notin \{i_1, i_2, \ldots, i_n\} \), \( u_{C_i} \) is matched to \( v_{C_i} \).

The above matching is also satisfiable because the edges induce total-order relationships of the form \( w_i \geq w \) when \( i \in \{i_1, i_2, \ldots, i_n\} \), and of the form \( w_i < w \) for other choice of \( i \). Thus, all edges can be satisfied simultaneously. Finally, it is easy to check that the above matching has \( 3n + (m - n) \) edges, so that it is a maximum satisfiable matching.

The "if" direction. If the size of MSM is exactly \( 3n + (m - n) \), we claim that \( z \), which is the number of \( i \)'s such that \( u_{i,j}, u_{i,k}, \) or \( u_{i,l} \) appear in the MSM, must be exactly \( n \); in addition, for each such \( i \), all \( u_{i,j}, u_{i,k}, u_{i,l} \) must appear in the matching. If this claim is true, it will immediately imply the corresponding 3-sets \( C_i \)'s (in total \( n \) of them) will cover exactly \( S \).

Now, it remains to prove the claim. We first show that \( z = n \). If \( z < n \), then the matching can contain at most \( 3z \) edges connecting some \( u_{i,j} \) with \( v_{i,j} \), and at most \( m - z \) edges connecting some \( u_{C_i} \) with \( v_{C_i} \), so that the number of edges is at most \( 3z + (m - z) \), which is less than \( 3n + (m - n) \). On the other hand, if \( z > n \), then the matching can contain at most \( 3n \) edges connecting some \( u_{i,j} \) with \( v_{i,j} \) (because \( v_{i,j} \) is limited), and at most \( m - z \) edges connecting some \( u_{C_i} \) with \( v_{C_i} \), so that the number of edges is at most \( 3n + (m - z) \), which again is less than \( 3n + (m - n) \). Thus, if the size of MSM is \( 3n + (m - n) \), we must have \( z = n \).

Given \( z = n \), there are at most \( m - n \) edges connecting \( u_{C_i} \) with \( v_{C_i} \). Thus, at least \( 3n \) edges must be connecting some \( u_{i,r} \) with \( v_r \). However, since there are only \( n \) values of \( i \) with \( u_{i,j}, u_{i,k}, u_{i,l} \) appear in the MSM, the previous statement is possible unless for each such \( i \), all \( u_{i,j}, u_{i,k}, u_{i,l} \) appear in the matching. Thus, the proof of the claim completes, and so do the proofs of the Proposition 1 and Corollary 2.

A.2 Proof sketch of the optimal integration plan problem

Given an instance of a cross product optimization [12], we create a corresponding symbolic database such that the matching size between two databases is always equal to the size of the cartesian product of the databases plus the size of the two databases. Thus, finding the maximum (satisfiable) matching equals to finding the optimal join ordering, which is \( \mathcal{NP} \)-hard.