Reverse Query Processing

Carsten Binnig
University of Heidelberg
binnig@informatik.uni-heidelberg.de

Donald Kossmann
ETH Zurich
kossmann@inf.ethz.ch

Eric Lo
ETH Zurich
eric.lo@inf.ethz.ch

Abstract

Generating databases for testing database applications (e.g., OLAP or business objects) is a daunting task in practice. There are a number of commercial tools to automatically generate test databases. These tools take a database schema (table layouts plus integrity constraints) and table sizes as input in order to generate new tuples. However, the databases generated by these tools are not adequate for testing a database application. If an application query is executed against such a synthetic database, then the result of that application query is likely to be empty or contain weird results, such as a report on the performance of a sales person that contains negative sales. To solve this problem, this paper proposes a new technique called Reverse Query Processing (RQP). RQP gets a query and a result as input and returns a possible database instance that could have produced that result for that query. RQP also has other applications; most notably, testing the performance of DBMS and debugging SQL queries.

1. Introduction

When designing a completely new database application or a component of such an application (e.g., some reporting functionality) it is necessary to generate a test database in order to carry out functional tests on the new application logic. There are a number of commercial and academic tools [1, 4, 5, 21] which enable the generation of a test database for a given database schema. Beside the database schema, some tools support the input of the table sizes and additional rules used for data instantiation (e.g. statistical distributions, value ranges, data repositories).

However, these tools often generate test databases which do not reflect the semantics of the application logic that should be tested. In other words, if we pose the SQL queries of an application against the test database, the SQL queries often return no or non-meaningful results. An example of that can be shown by the following query which is extracted from a reporting application. The query lists the total sales of ordered line items per day, if the discounted price was less than a certain average and more than a certain sum (the database schema of the application is given in Figure 2a):

```
SELECT orderdate, SUM(price*(1-discount))
FROM Lineitem, Orders WHERE l_oid=oid
GROUP BY orderdate
HAVING AVG(price*(1-discount))<=100
AND SUM(price*(1-discount))>=150;
```

The following tables show a real excerpt of the test database generated by a commercial test database generation tool[1] for the example application:

lid	name	price	discount	l_oid
103132	Kc1cqZlf	810503883	0.7	1214077
126522	hcTpT8ud34	994781460	0.1	1214077
397457	5SwWn9q3	436001336	0.0	1297288
...	...	...	...	...

Table *Lineitem*

oid	orderdate
1214077	1983-01-23
1297288	1995-01-01
...	...

Table *Orders*

It is obvious that the query above returns an empty result for that test database because none of the generated tuples satisfies the complex `HAVING` clause (including different aggregations on arithmetic functions). Even though some tools allow the user to specify additional rules in order to constrain the generated databases (e.g., constraining the domain of `discount`), those constraints are defined on the base tables only and there are no means to control the query results directly. Therefore, those tools can hardly deal with the complexity of SQL and application programs; not even with the single SQL query above.

To generate meaningful test databases for applications, this paper proposes to take the application query and the desired query result (in addition to the database schema) as input and to generate a database accordingly. More formally, given a Query Q and a Table R, the goal is to find a Database D (a set of tables) such that $Q(D) = R$. We call this problem *reverse query processing* or RQP, for short. RQP turns traditional query processing around. For example, RQP is based on a reverse relational algebra (RRA). Logically, each operator of the relational algebra has a corresponding operator of the reverse relational algebra that implements its reverse function. All reverse algebra operators must respect the integrity constraints of the database schema in order to generate correct output. Furthermore, unlike traditional query processing, iterators in RQP are push-based. Thus the whole data processing is started by scanning the query result and pushing each tuple down to the leaves (i.e., the base tables) of the query tree.

RQP has several applications. In this paper we focus

[1] We do not disclose the name of the tool for legal reasons.

on using RQP to generate a test database based on an individual query of an application program in order to carry out functional tests (N.B. the generated database is not for testing the correctness of the queries in the application) . For example, consider an application with an *if-else* block where the *if* condition relies on the result R of a query Q. Given that query Q and different results R (e.g. one R for each branch of the *if-else* block), RQP can generate different databases to test all code paths of that application (R can be given by the testers manually or by some code analysis tools, such as [20]). For some applications, it would be beneficial to generate one database for all queries within a single application program and to consider other SQL statements such as `UPDATE` statements (which is not possible for a mock object which simulates the query results). Usually, RQP can then be applied to each query and the `WHERE` clauses of `UPDATE` statements individually and the union of all RQP results can be used as a test database for the whole application. For some complex applications, the union of RQP results may not be adequate and thus we have to merge the RQP generated databases in order to fully test all the facets of an application. Currently, based on the result of this work, we are devising a set of formal criteria for such database merging; however, presenting these criteria is beyond the scope of this paper.

Another application of RQP is to generate test databases in order to test the performance of a RDBMS for *any user defined benchmark queries*. In this application, in addition to allowing the user to specify the target database size like existing commercial test database generation tools, users can specify the *size of the query results* and the *selectivity* of each predicate. This way, the performance of a RDBMS can be studied thoroughly from a totally different angle.

There are a few more possible RQP applications. For example, we can use RQP to debug SQL queries, or to generate databases from different materialized views in order to test the confidentiality of the view data [19]. A detailed discussion of all RQP applications can be found in [3].

The remainder of this paper is organized as follows: Section 2 defines the problem and gives an overview of RQP. Section 3 describes the logical reverse relational algebra for RQP. Section 4 presents the physical implementation of RRA for functional testing. Section 5 describes the results of the experiments and Section 6 discusses related work. Section 7 contains conclusions and future work.

2. RQP Overview

2.1. Problem Statement

Given an SQL Query Q, the Schema S_D of a relational database (including integrity constraints), and a Table R (called RTable), find a database instance D such that:

$$R = Q(D)$$

and D is compliant with S_D and its integrity constraints.

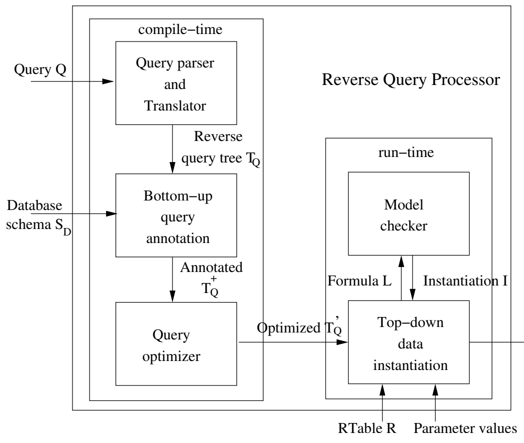


Figure 1. RQP Architecture

In general, there are many different database instances which can be generated for a given Q and R. Depending on the application, some of these instances might be better than others. For functional testing, RQP should generate a small D that satisfies the correctness criteria above, so that the running time of tests is reduced. Thus, the physical implementation of the operators presented in Section 4 tries to generate a minimal D.

2.2. RQP Architecture

Figure 1 gives an overview of the general architecture to implement reverse query processing. It is applicable to all RQP applications such as functional testing or performance testing. A query is (reverse) processed in four steps by the following components.

1. Parser: The SQL query is parsed into a query tree that consists of operators of the relational algebra. This parsing is carried out in exactly the same way as in a traditional SQL processor. What makes RQP special is, that this query tree is translated into a *reverse query tree*. In the reverse query tree, each operator of the relational algebra is translated into a corresponding operator of the *reverse relational algebra*. The reverse relational algebra is presented in more detail in Section 3. In fact, in a strict mathematical sense, the reverse relational algebra is not an algebra and its operators are not operators because they allow different outputs for the same input. Nevertheless, we use the terms *algebra* and *operator* in order to demonstrate the analogies between reverse and traditional query processing.

2. Bottom-up Query Annotation: The bottom-up query annotation phase in Figure 1 annotates each operator of a reverse query tree with an *input schema* S^{IN} and an *output schema* S^{OUT}. The input and output schema are defined by a set of attributes (names and data types), integrity constraints, and functional dependencies. RQP considers the integrity constraints of SQL (primary key, unique, foreign key, not null, and check) as well as aggregation constraints

[22]. By enhancing [16], this step annotates each operator with a set of these integrity constraints that its input and output must fulfill. It is carried out in a bottom-up fashion, i.e., the annotation starts from the leaves (tables) up to the top of the query. S^{IN} and S^{OUT} are necessary for the top-down data instantiation (Step 4): in this step, each operator of the reverse query tree can check, if its input satisfies the constraints of its subqueries and the database schema (by S^{IN}). Moreover, none of the operators *generates any output data that violates any constraints* (by S^{OUT}). Due to space constraints, we explain the details in [3].

3. Query Optimization: In the last step of compilation, the reverse query tree is transformed into an *equivalent* reverse query tree that is expected to satisfy a certain optimization goal (e.g., running time and/or database size). Traditional query optimization is based on result-equivalence: after a rewrite the same results should be produced. Query optimization for RQP can be much more aggressive: it is acceptable to generate a different D for the same input as long as the criterion $R = Q(D)$ is fulfilled. As a result, RQP allows more rewrites. More details on query optimization in RQP can be found in [3]. For example, [3] shows how to fully unnest nested SQL queries for optimization. Moreover, it shows that it is not important to carry out join reordering because joins in RQP are mostly cheap.

4. Top-down Data Instantiation: At run-time, the annotated reverse query tree is interpreted using the RTable R as input. Just as in traditional query processing, there is a physical implementation for each operator of the reverse relational algebra that is used for reverse query execution. The physical algebra in this paper is to generate a D for functional testing which tries to be minimal. Generating databases for performance testing needs a different physical algebra. As part of this step, a model checker (more precisely, the decision procedure of a model checker) [6] is used in order to generate data. The physical algebra in this paper is fully described in Section 4.

In many applications, queries have parameters (e.g., bound by a host variable). In order to process such queries, values for the query parameters must be provided as input to Top-down data instantiation. For functional testing, it is possible to generate several test databases with different parameter settings derived from the program code in order to test different code paths. In this case, the first three phases of query processing only have to be carried out once, and the Top-down data instantiation can use the same annotated reverse query tree for each set of parameter settings.

2.3. RQP Example

Figure 2 gives a detailed example of reverse query processing. Figure 2a shows the database schema (definition of the `Lineitem` and `Orders` tables with their integrity constraints) and an SQL query that asks for the sales (`SUM(price)`) by `orderdate`. The query is parsed and optimized and the result is a reverse query tree with operators of the reverse relational algebra (see Figure 2b). This tree is very similar to the query tree used in traditional query processors. The differences are that (1) operators of the reverse relational algebra (Section 3) are used and (2) that the data flow through this tree is from top to bottom (rather than from bottom to top).

The data flow at run-time is shown in Figure 2c. Starting with an RTable that specifies that two result tuples should be generated (Table (i) at the top of Figure 2c), each operator of the reverse relational algebra is interpreted by the Top-down data instantiation component in order to produce intermediate results of reverse query processing. In this phase, RQP uses the decision procedure of a model checker in order to guess appropriate values (e.g., possible `AVG(price)` values which are compliant with the predicate of the `HAVING` clause of the query). Of course, several solutions are possible and the decision procedure of the model checker chooses possible values that match all constraints discovered in the Bottom-up annotation step. The final result of RQP in this example are possible instantiations for the `Lineitem` and `Orders` tables. It is easy to see that these instantiations meet the integrity constraints of the database schema and that (forward) executing the SQL query using these instantiations gives the RTable as a result.

3. Reverse Relational Algebra

The Reverse Relational Algebra (RRA) is a reverse variant of the traditional relational algebra [7] and its extensions for group-by and aggregation [10]. Each operator of the relational algebra has a corresponding operator in the reverse relational algebra; the symbols are the same (e.g., σ for selection), but the operators of the RRA are marked as op^{-1} (e.g., σ^{-1}). Furthermore, the following equation holds for all operators and all valid tables R:

$$op(op^{-1}(R)) = R$$

However, reverse operators in RRA should not be confused with *inverse* operators because $op^{-1}(op(S)) = S$ is **not** necessarily true for some valid tables S.

In the traditional relational algebra, an operator has 0 or more inputs and produces exactly one output relation. Conversely, an operator of the RRA has exactly one input and produces 0 or more output relations. Just as in the traditional relational algebra, the operators of the RRA can be composed. As shown in Figure 2b, the composition is carried out according to the same rules as for the traditional relational algebra. As a result, it is very easy to construct a reverse query plan for RQP by using the same SQL parser as for traditional query processing.

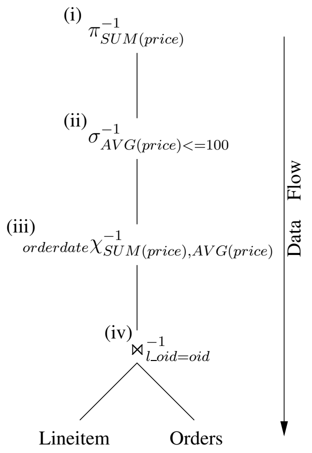
The close relationship between RRA and the traditional relational algebra has two consequences: (1) The reverse variants of the basic operators of the (extended) relational algebra (selection, projection, rename, Cartesian product,

```
CREATE TABLE Orders(
oid INTEGER PRIMARY KEY,
orderdate DATE);

CREATE TABLE Lineitem (
lid INTEGER PRIMARY KEY,
name VARCHAR(20),
price FLOAT,
discount FLOAT
  CHECK (1>= discount >=0),
l_oid INTEGER REFERENCES Orders);

SELECT SUM(price)
FROM Lineitem, Orders
WHERE l_oid=oid
GROUP BY orderdate
HAVING AVG(price)<=100;
```

(a) Example Schema and Query


(i) $\pi^{-1}_{SUM(price)}$
(ii) $\sigma^{-1}_{AVG(price)<=100}$
(iii) $_{orderdate}\chi^{-1}_{SUM(price),AVG(price)}$
(iv) $\bowtie^{-1}_{l\_oid=oid}$
Lineitem
Orders
Data Flow

(b) Reverse Relational Algebra Tree

SUM(price)
100
120

(i) RTable

orderdate	SUM(price)	AVG(price)
1990-01-02	100	100
2006-07-31	120	60

(ii) Output of π^{-1}; Input of σ^{-1}

orderdate	SUM(price)	AVG(price)
1990-01-02	100	100
2006-07-31	120	60

(iii) Output of σ^{-1}; Input of χ^{-1}

lid	name	price	discount	l_oid	oid	orderdate
1	productA	100.00	0.0	1	1	1990-01-02
2	productB	80.00	0.0	2	2	2006-07-31
3	productC	40.00	0.0	2	2	2006-07-31

(iv) Output of χ^{-1}; Input of $\bowtie^{-1}$

lid	name	price	discount	l_oid
1	productA	100.00	0.0	1
2	productB	80.00	0.0	2
3	productC	40.00	0.0	2

Lineitem

oid	orderdate
1	1990-01-02
2	2006-07-31

Orders

(c) Input and Output of Operators

Figure 2. Example Schema and Query for RRA

union, aggregation, and minus) form the basis of the RRA. All other operators of the RRA (e.g., reverse outer joins) can be expressed as compositions of these basic operators. (2) The relational algebra has laws on associativity, commutativity, etc. on many of its operators. Analogous versions of most of these laws apply to the RRA. Some laws are not applicable to the RRA (e.g., applying projections before joins). These laws are listed in [16] and must be respected for RQP optimization (see [3] for details).

The remainder of this section defines the seven basic operators of the RRA The definition of RRA operators is independent of the application. A physical implementation of RRA for generating test databases for functional testing is described in Section 4.

3.1. Reverse Projection (π^{-1})

The reverse projection operator π^{-1} generates new columns according to its *output schema* S^{OUT}. As for all operators of the reverse relational algebra, $\pi(\pi^{-1}(R)) = R$ must apply for all valid R.

In Figure 2, π^{-1} creates the `orderdate` and `AVG(price)` columns. In order to generate correct values for these columns, π^{-1} needs to be aware of the constraints imposed by the aggregations (`SUM` and `AVG`) and the `HAVING` clause of the query. That is, the values in the `AVG(price)` column must be smaller or equal to 100 so that the σ^{-1} does not fail. Furthermore, the value of the `orderdate` column must be unique and the values in the `AVG(price)` and `SUM(price)` columns must match so that the reverse aggregation does not fail (that is why we need the bottom-up phase to compute S^{OUT} that includes all the necessary constraints). In this example, there are no other integrity constraints from the database schema that must be respected as part of the reverse projection. In general, such constraints must also be respected in an implementation of the π^{-1} operator. If it is impossible to generate values that fulfill all the constraints, it returns *error*.

3.2. Reverse Selection (σ^{-1})

The simplest operator of the RRA is the reverse selection: It either returns *error* or a superset (or identity) of its input. The additional tuples not in the input must fulfill the negation of the selection predicate. *Error* is returned if the input of the reverse select operator does not match the selection predicate. For example, if the query asks for all employees with a salary greater than 10,000 and the RTable contains an employee with a salary of 1,000, then *error* is returned. Another example of σ^{-1} is given in Figure 2c. In that example no additional tuples are added to the output of σ^{-1} (Table (iii) in Figure 2c).

3.3. Reverse Aggregation (χ^{-1})

Like the π^{-1} operator, the reverse aggregation operator generates columns. Furthermore, the reverse aggregation operator possibly generates additional rows in order to meet all constraints of its aggregate functions. Again, as for all RRA operators, the goal is to make sure that $\chi(\chi^{-1}(R)) = R$ and that the output is compliant with all constraints of the output schema (functional dependencies, predicates, etc.). If this is not possible, then the reverse aggregation fails and returns *error*.

Tables (iii) and (iv) of Figure 2c show the input and output of reverse aggregation of the running example. In that example, the values of the `lid, name`, and `discount` columns are generated obeying the integrity constraints of the `Lineitem` table (see Figure 2a). The value of the `price` column is generated using the input (the result of the reverse selection) and the intrinsic mathematical properties of the aggregate functions. The values of `l_oid` and `oid` are generated obeying the constraints imposed by the join predicate of the query and the `primary key` constraint of the `Orders` table.

3.4. Other operators

The reverse join operator ($\bowtie^{-1}$) completes the running example. It takes one relation as input and generates two output relations. Like all other operators, the reverse join makes sure that its outputs meet the specified output schemas (the database schemas for the `Lineitem` and `Orders` tables in the example of Figure 2) and that the join of its outputs gives the correct result. If it is not possible to fulfill all these constraints, an *error* is raised. The only thing that is special about the $\bowtie^{-1}$ operator is that it has two outputs. The reverse Cartesian product is a variant of the reverse join with *true* as a join predicate.

The reverse union operator ($\cup^{-1}$) takes one relation as input and generates two output relations. According to the constraints of the two output schemas of the two output relations (computed during the bottom-up query annotation phase), the reverse union distributes the tuples of the input relation accordingly to one or even to both output relations. If the input of a reverse union involves a tuple that does not fulfill the constraints of any branch, then the reverse union fails and returns *error*.

The reverse minus operators ($-^{-1}$) always routes the input tuples to the left branch or returns an error, if this is not possible. Furthermore, it is possible that the $-^{-1}$ generates additional tuples for both branches.

The reverse rename operator has the same semantics as in the traditional relational model. Thus, only the output schema is affected; no data manipulation is carried out. Examples of all these operators are shown in [3].

4. Top-down Data Instantiation

The Top-down data instantiation component in Figure 1 interprets the optimized reverse query execution plan using an RTable R and possibly query parameters as input. It generates a database instance D as output. The generated database D fulfills the constraints of the database schema and the overall correctness criterion of RQP. If this is not possible, then *error* is returned.

The reverse query execution plan consists of a set of physical RRA operators. As in traditional query processing, each operator in the execution plan is implemented as an iterator [11]. Unlike traditional query processing, the iterators are push-based. That is, whenever an operator produces a tuple, it calls the *pushNext* method of the relevant child (output) operator and continues processing once the child operator is ready. Thus, the whole data instantiation is started by scanning the RTable and pushing each tuple of the RTable one at a time to the children operators of the reverse query plan. Such a push-based model is required because operators of the RRA can have multiple outputs; the alternative would be to implement a pull-based model with buffering which is significantly more complex [17]. All iterators have the same interface which contains the following three methods:

- *open()*: prepare the iterator for producing data as in traditional query processing;
- *pushNext(Tuple t)*: (a) receive a tuple t, (b) check if t satisfies the input schema S^{IN} of the operator, (c) produce zero or more output tuples, and (d) for each output tuple, call the *pushNext* method of the relevant children operators;
- *close()*: clean up everything as in traditional query processing.

As in traditional query processing, the set of physical RRA operators is called the physical reverse relational algebra. Each logical RRA operator may have different counterparts in the physical RRA. The choice is application dependent; for example, different physical implementations are used for SQL debugging and for performance testing. This section presents the physical algebra of SPQR which strictly follows the RRA in Section 3. SPQR is a RQP prototype for functional testing. The physical algebra of SPQR tries to keep the generated database as small as possible. The following subsections show how the operators produce tuples in their *pushNext* method. All other aspects (e.g., *open* and *close*) are straightforward so that the details are omitted for brevity.

4.1. Reverse Projection in SPQR

In SPQR, the reverse project operator produces exactly one output tuple for each input tuple. In order to generate values for new columns, the reverse project operator calls the decision procedure of a model checker. The idea is to create a formula which represents the constraints which have to be satisfied by the output. These constraints represent the values known from the input tuple on the one hand and the output schema on the other hand. For example, if the input schema has one column (A), the input tuple is $\langle 3 \rangle$, and the output schema has two columns (A and B) and an additional constraint that $A + B < 30$, then the following constraint formula is generated:

```
A = 3 & A+B < 30
```

This constraint formula is passed to the model checker. In SPQR, we treat the model checker as a black box. It takes a constraint formula as input and returns one of the possible data instantiations on all variables as output (if the constraint formula is satisfiable). In this example, the model checker would return, say, $A = 3, B = 20$ and these values would be used to generate an output tuple.

Figure 3 shows the pseudocode of how the π^{-1} operator generates an output tuple from an input tuple. The most important statement is the call of the *instantiateData* function (Line 2) which does the actual work. Since this function is also used by the implementation of the χ^{-1} operator, it has two return parameters: one which defines the instantiated data (variable, value pairs) and another which indicates how

```
$\pi^{-1}$.pushNext(Tuple $t$)
(1)  //Instantiate output data
(2)  ($I$,$count$):=instantiateData($t$,$S^{OUT}$)
(3)  IF($I$=NULL) //no instantiation found
(4)    RETURN $error$
(5)  ELSE
(6)    $t_{out}$:=createTuple($I$,$S^{OUT}$,1)
(7)    //push down the new tuple $t_{out}$
(8)    nextOperator.pushNext($t_{out}$)
(9)  END IF
```

Figure 3. Method `pushNext` **of** π^{-1}

```
instantiateData(Tuple $t$, Schema $S^{OUT}$)
Output:
 -instantiation $I$ //data instantiation
 -int $n$ //number of tuples for aggregation
 (1)  //number of tuples for aggregation
 (2)  IF $t$ includes COUNT of aggregation
 (3)    $count$,$maxcount$:=COUNT value in $t$
 (4)  ELSE //USER_THREHOLD=1 if no aggregation
 (5)    $count$:=1; $maxcount$:=USER_THRESHOLD
 (6)  END IF
 (7)  FOR($n$=$count$ TO $maxcount$)
 (8)    //Create constraint formula $L$
 (9)    $L$:=createConstraint($t$,$S^{OUT}$,$n$)
(10)    $I$:=decisionProcedure($L$)
(11)    IF($I$!=NULL) RETURN ($I$,n)
(12)  END FOR //Trial-and-error
(13)  RETURN (NULL,0)
```

Figure 4. Function `instantiateData`

many tuples are used to solve aggregations which might be part of the formula (see below). The second return value is only needed for the χ^{-1} operator so that it can be ignored for the moment. If the call to *instantiateData* was successful (i.e., $I \neq NULL$ in Line 3), then a new output tuple is created according to the output schema of the π^{-1} operator and passed to the next reverse operator (Lines 6 to 8). Otherwise, *error* is returned (Line 4).

The pseudocode of a simplified version of the *instantiateData* function is shown in Figure 4. This function creates a constraint formula L (Line 9) following the semantics of the reverse operator and executes the decision procedure of the model checker on L (Line 10). As part of the creation of the constraint formula, restrictions of the model checker need to be taken into account. For example, the model checker used in the performance experiments (Section 5) does not support SQL numbers and dates. As a result, all SQL numbers and dates must be converted into (long) integers and the constraints must be adjusted accordingly. Furthermore, arithmetic expressions (e.g., $A + B$) which might appear in the input and output schemas of the reverse projection must be taken into account.

The most complex part of the *instantiateData* function deals with the generation of columns that involve aggregations. In Figure 2, for example, the π^{-1} operator needs to generate values for the `AVG(price)` column. In order to generate correct values, the *instantiateData* function needs to guess how many tuples are aggregated by the aggregate function; for instance, two tuples are aggregated for the second tuple of the RTable in Figure 2. The two tuples are generated by the χ^{-1} operator, but the π^{-1} operator which only generates one output tuple per input tuple must be aware of this fact in order not to generate values that cannot be matched by the χ^{-1} operator. Unfortunately, today's publicly available model checkers have not been designed for aggregation so that this guessing must be carried out as part of the *instantiateData* function in a trial-and-error phase (Lines 7 to 12). The guessing iteratively tries different values of *n* (the number of tuples aggregated) and calls the decision procedure for each value until the decision procedure of the model checker was successful to instantiate data.

Continuing the example in Figure 2 for the second tuple of the RTable (`SUM(price) = 120`), the following formula is generated for $n = 1$:[2]

```
sum_price=120 &
orderdate!=19900102 & avg_price<=100 &
sum_price=price1 & avg_price=sum_price/1
```

This formula is given to the decision procedure of the model checker and obviously, the model checker cannot find values for the variables `price1` and `avg_price` that meet all constraints. In the second attempt for $n = 2$, the following formula is passed to the decision procedure:

```
sum_price=120 &
orderdate!=19900102 & avg_price<=100
sum_price=price1+price2 & avg_price=sum_price/2
```

This time, the decision procedure finds an instantiation:[3]

```
sum_price=120, avg_price=60,
price1=80, price2=40,
orderdate=20060731
```

From this instantiation, the values of `orderdate`, `avg_price`, and `sum_price` are used in order to generate the output tuple of the reverse project operator. In the SPQR prototype, the maximum number of attempts ($maxcount$ in Figure 4) can be constrained by the user in order to make sure that the whole process does not run for ever. Moreover, all the guessing is not necessary, if the query involves a `COUNT` aggregation, because the values (or constraints) of the corresponding `COUNT` column in the tuple (t) can be used (Lines 2 and 3 of Figure 4). Furthermore, in order to avoid the guessing, several optimizations can be applied (Section 4.5). These optimization techniques work very well so that in practice guessing is eliminated very often; in fact, the experimental results in Section 5 show that all guessing is eliminated by the proposed optimization for the whole TPC-H benchmark.

The pseudocode of Figure 4 is a simplification for the special case that there are no nested aggregations (e.g., `SUM(AVG(price))`) and no joins on aggregated values

[2]The constraint on `orderdate` is generated because `orderdate` is the primary key of the output schema and, thus, a different `orderdate` value must be generated for the `SUM(price) = 120` than for the `SUM(price) = 100` tuple. 19900102 is the integer representation for the date January 2, 1990, the `orderdate` value of the `SUM(price) = 100` tuple.

[3]20060731 is the integer representation of the date July 7, 2006.

```
$\chi^{-1}$.pushNext(Tuple $t$)
(1)  //Instantiate data
(2)  ($I$,$count$):=instantiateData($t$,$S^{OUT}$)
(3)  IF($I$=NULL) //no instantiation found
(4)   RETURN $error$;
(5)  ELSE
(6)   FOR($n$=1 TO $count$)
(7)    $t_{out}$ :=createTuple($I$,$S^{OUT}$,$n$)
(8)    nextOperator.pushNext($t_{out}$)
(9)   END FOR
(10) END IF
```

Figure 5. Method `pushNext` **of** χ^{-1}

(e.g., aggregations in several subqueries). However, the code can easily be generalized for all cases. This generalization is not shown because it is fairly straightforward. SPQR indeed implements such a generalized version of the *instantiateData* function.

4.2. Reverse Aggregation in SPQR

The reverse aggregation operator can be implemented in an analogous way to the reverse projection. The difference is that while the π^{-1} operator only guesses how many tuples are potentially involved in an aggregation, the χ^{-1} operator actually generates these tuples. The key idea to use the decision procedure of a model checker, however, is the same.

Figure 5 shows the pseudo-code. The *instantiateData* function is called in the same way as for π^{-1}. The only difference is that the return parameter *count* is now initialized (Line 2) which defines the number of output tuples. If the *instantiateData* function was successful, then *count* tuples are generated (Lines 6 to 9) using the values returned by the *instantiateData* function. If not, then *error* is generated (Lines 3 and 4). Again, an example that shows this code in action can be seen in Figure 2c (Tables (iii) and (iv)).

4.3. Other Operators in SPQR

The reverse join operator can be implemented in different ways, depending on the join predicate. The simplest (and cheapest) implementation is the implementation of an equi-join that involves a primary key or an attribute with a unique constraint in the join predicate. Such joins are the most frequent joins in practice. They can be implemented as a simple projection with duplicate elimination. The implementation of general joins and Cartesian products is more complex; the full algorithms are given in [3]. In any case, the implementation of reverse joins and Cartesian products does not involve calls to the model checker. So these operators are much cheaper than π^{-1} and χ^{-1}.

The other operators of the reverse relational algebra (reverse selection, rename, minus, and union) are trivial to implement. For example, the reverse selection can be implemented as the identity function. Due to space constraints, the implementation details for these operators are given in [3]. [3] also contains some fine points on the implementation of the reverse projection and aggregation operator. Moreover, [3] also shows that there is a limitation on implementing some physical RRA operators: If the same database table is referenced multiple times in a reverse query tree, then the physical implementations of σ^{-1}, $\bowtie^{-1}$ and $-^{-1}$ are not allowed to generate additional tuples for this table. This limitation does not affect the physical RRA in this paper as these operators generate no additional tuples in order to keep D as small as possible. But this limitation does affect physical algebras which generate additional tuples (e.g., the physical algebra for performance testing).

4.4. Processing Nested Queries

In order to reverse process a nested query, SPQR uses the concept of nested iterations (with special *apply* operators) which are known from traditional query processing [9], in a reverse way: The inner subquery can be thought of as a reverse query tree whose input is parameterized on values generated for correlation variables of the outer query. Just as in traditional query processing, reverse processing of nested queries is expensive: it has quadratic complexity with the size of the RTable.

4.5. Optimization of Data Instantiation

The previous subsections showed that reverse query processing heavily relies on calls to a model checker. Unfortunately, those calls are expensive. Furthermore, the cost of a call grows with the length of the formula; in the worst case, the cost is exponential to the size of the formula. The remainder of this section lists techniques in order to reduce the number of calls to the model checker and to reduce the size of the formulae (in particular, the number of variables in the formulae). The optimizations are illustrated using the example of Figure 2.

Definition - Independent attribute: An attribute a is *independent* with regard to an output schema S^{OUT} of an operator iff S^{OUT} has no integrity constraints limiting the domain of a and a is not correlated with another attribute a' (e.g. by $a > a'$) which is not independent.

Definition - Constrictive independent attribute: An attribute a is *constrictive independent*, if it is independent with regard to an output schema S^{OUT} disregarding certain optimization-dependent integrity constraints.

The following optimizations use these definitions:

OP 1 - Default-value Optimization: This optimization assigns a default (fixed) value to an independent attribute a. The default value assigned to a depends on the type of the attribute. Attributes which use this optimization are not included in the constraint formula. An example attribute which could use this optimization is the attribute `name` of `Lineitem`; it could use a default value; e.g., “product”.

OP 2 - Unique-value Optimization: This optimization assigns a unique increment counter value to a constrictive independent attribute a, which is only bound by unique or primary key constraints. Here, the optimization-dependent integrity constraints, which are disregarded by the constrictive independent attribute a, are unique and primary key constraints. Attributes which use this optimization are not included in the constraint formula. In the running example, values for the `lid` attribute could be generated using this optimization. If another attribute a' of the same schema exists which is correlated by equality (e.g. $a = a'$ from an equi-join) and a' is an independent or a constrictive independent attribute which is only bound by unique or primary key constraints, then attribute a' is set to the same unique value as a and constraints involving a' need not be included in calls to the model checker either.

OP 3 - Single-value Optimization: This optimization can be applied to a constrictive independent attribute a which is only bound by `CHECK` constraints. An example of such an attribute is the attribute `discount` of `Lineitem`. Such attributes are only included in a constraint formula, the first time the top-down phase needs to instantiate a value for them. Afterwards, the instantiated value is reused.

OP 4 - Aggregation-value Optimization: This optimization can be applied to constrictive independent attributes a which are involved in an aggregation. If the attribute a is used in an aggregation function, e.g., `SUM(a)` and a result value for the aggregation function is given, then we can use the following arithmetics to instantiate a directly:

1. If `SUM(a)` is an attribute in the operator's input schema, `MIN(a)` and `MAX(a)` are not in the operator's input schema, and a has type float: Instantiate a value for a by solving a=`SUM(a)/n` with n the number of tuples used to solve the aggregation in the `instantiateData` function. In this case, no variables $a_1, a_2, \ldots, a_n$ need to be generated and used in the constraint formula passed to the model checker.
2. Same as (1), but `MIN(a)` or `MAX(a)` are in the operator's input schema, and $n \geq 3$: Use values for `MIN(a)` or `MAX(a)` once to instantiate a. Instantiate the other values for a by solving a=`(SUM(a)-MIN(a)-MAX(a))/(n-2)`.
3. Same as (1), but a is of data type integer: Again, we can directly compute a by solving `SUM(a)`$=n_1 \times a_1 + n_2 \times a_2$, where a_1=$\lfloor$`sum(a)/n`$\rfloor$, a_2=$\lceil$`sum(a)/n`$\rceil$, n_1=$n - n_2$ and n_2=`(SUM(a) modulo n)`.
4. If only `COUNT(a)` is in the operator's input schema, a can be set using the Default-value optimization (OP 1) because a is independent in this case.

OP 5 - Count heuristics: Unlike the previous four optimizations, this optimization does not find instantiations. Instead, this optimization reduces the number of attempts for guessing the number of tuples (n in Figure 4) to reverse process an aggregation by constraining the value of n. The heuristics for this purpose are shown below. The theoretical foundations for these heuristics are given in [22].

1. If `SUM(a)` and `AVG(a)` are attributes of the operator's input schema, then n=`SUM(a)/AVG(a)`.
2. If `SUM(a)` and `MAX(a)` are attributes of the operator's input schema, then $n \geq$ `SUM(a)/MAX(a)` (if `SUM(a)` and `MAX(a)` ≥ 0; if `SUM(a)` and `MAX(a)` ≤ 0 use $n \leq$ `SUM(a)/MAX(a)`).
3. If `SUM(a)` and `MIN(a)` are attributes of the operator's input schema, then $n \leq$ `SUM(a)/MIN(a)` (if `SUM(a)` and `MIN(a)` ≥ 0; if `SUM(a)` and `MIN(a)` ≤ 0 use $n \geq$ `SUM(a)/MIN(a)`).

OP 6 - Tolerance on precision: Depending on the application of RQP, tolerances can be exploited in order to speed up model checking. That is, rather than, say, specifying `a = 100`, a more flexible constraint `90` $\leq$ `a` $\leq$ `110` can be used. Of course, this optimization is only legal for certain applications. Our prototype, SPQR has a user-defined tolerance range which is set to 0 percent by default.

OP 7 - Memoization: Another general optimization technique is to cache calls to the model checker. For example, π^{-1} and χ^{-1} often solve similar constraints and carry out the same kind of guessing. In Figure 2, for instance, the results of guessing for the π^{-1} operator can be re-used by the χ^{-1} operator. Memoization at run-time has been studied in [14] for traditional query processing; that work is directly applied in SPQR.

5. Performance Experiments and Results

This section presents the results of performance experiments with our prototype system SPQR and the TPC-H benchmark [2]. We used TPC-H in order to show that SPQR can reverse process complex SQL queries and that SPQR scales for different sizes of generated databases.

The SPQR system was implemented in Java 1.4 and installed on a Linux AMD Opteron 2.2 GHz Server with 4 GB of main memory. In all experiments reported here, SPQR was configured to allow 0 percent tolerance; that is, OP 6 of Section 4 was disabled. As a backend database system, PostgreSQL 7.4.8 was used and installed on the same machine. As a decision procedure, Cogent [8] was used. Cogent is a decision procedure that is publicly available and has been used in several projects world-wide. For our purposes, it was configured to generate *error*, if numerical overflows occurred.

The TPC-H benchmark is a decision support benchmark and consists of 22 business oriented queries and a database schema with eight tables. The queries have a high degree of complexity: all of them include at least one aggregate function with a complex formula, and many queries involve subqueries.

The experiments were carried out in the following way: First, a benchmark database was generated using the *dbgen* function as specified in the TPC-H benchmark. As scaling factors, we used 0.1 (100 MB database; 860K rows), 1

	100M		1G		10G	
Query	RTable	Generated	RTable	Generated	RTable	Generated
1	4	600,572	4	6,001,215	4	59,986,052
2	44	220	460	2,300	4,667	23,335
3	1216	3,648	11,620	34,860	114,003	342,009
4	5	10,186	5	105,046	5	1,052,080
5	5	30	5	30	5	30
6	1	1	1	1	1	1
7	4	24	4	24	4	24
8	2	32	2	32	2	32
9	175	1,050	175	1,050	175	1,050
10	3767	15,068	37,967	151,868	381,105	1,524,420
11	2541	7,623	1,048	3,144	289,022	867,066
12	2	6,310	2	61,976	2	621,606
13	38	162,576	42	1,629,964	46	16,298,997
14	1	4	1	4	1	4
15	1	2	1	2	1	2
16	2762	23,264	18,314	236,500	27,840	2,372,678
17	1	3	1	3	1	3
18	5	15	57	171	624	1,871
19	1	2	1	2	1	2
20	21	105	204	1,020	1,968	9,840
21	47	2,325	411	20,705	4,009	197,240
22	7	1,282	7	12,768	7	127,828

Table 1. Size of RTable/Generated D (rows)

Query	#M-Inv	MC	QP	DB	*Total*	*Total*	*Total*
1	4	6:06	12:01	8:42	*26:51*	*207:11*	*2054:19*
2	44	0:02	$<$ 1ms	0:21	*0:24*	*0:47*	*4:02*
3	1216	18:55	0:14	0:11	*19:20*	*183:49*	*1819:48*
4	5	$<$ 1ms	0:05	0:14	*0:20*	*2:26*	*24:15*
5	10	0:11	$<$ 1ms	$<$ 1ms	*0:12*	*0:12*	*0:12*
6	2	0:1	$<$ 1ms	$<$ 1ms	*0:02*	*0:01*	*0:01*
7	8	0:9	$<$ 1ms	0:01	*0:10*	*0:10*	*0:09*
8	12	0:13	$<$ 1ms	0:02	*0:15*	*0:17*	*0:14*
9	175	4:17	0:02	0:03	*4:23*	*4:33*	*10:20*
10	3767	55:13	0:42	0:37	*56:33*	*566:45*	*5639:13*
11	2541	41:43	0:13	0:14	*42:11*	*18:15*	*4472:00*
12	3155	6:57	0:16	0:11	*7:25*	*83:09*	*719:56*
13	21	$<$ 1ms	1:38	1:16	*2:56*	*27:47*	*276:05*
14	6	0:07	$<$ 1ms	0:01	*0:08*	*0:08*	*0:15*
15	3	0:03	$<$ 1ms	$<$ 1ms	*0:03*	*0:03*	*0:04*
16	0	$<$ 1ms	0:15	0:14	*0:29*	*4:04*	*36:37*
17	2	0:01	$<$ 1ms	$<$ 1ms	*0:02*	*0:02*	*0:08*
18	15	$<$ 1ms	$<$ 1ms	$<$ 1ms	*0:01*	*0:10*	*1:54*
19	2	0:01	$<$ 1ms	$<$ 1ms	*0:02*	*0:02*	*0:02*
20	42	0:20	$<$ 1ms	$<$ 1ms	*0:21*	*3:24*	*32:27*
21	465	1:34	0:04	0:05	*1:43*	*14:44*	*140:47*
22	641	0:23	0:01	0:01	*0:26*	*4:08*	*42:00*
			a) SF=0.1			b) SF=1	SF=10

Table 2. Running Time (mm:ss): Varying SF

(1 GB; 8.6 million rows), and 10 (10 GB; 86 million rows). Then, the 22 queries were run, again as specified in the original TPC-H benchmark. The query results were then used as inputs (RTables) for reverse query processing of each of the 22 queries. We measured the size of the resulting database instance for each single query and the running time of reverse query processing.

5.1. Size of Generated Databases

Table 1 shows the size of the databases generated by SPQR for all queries on the three scaling factors. For queries which include an explicit or implicit[4] COUNT value in R, the size of the generated database for different scaling factors depends on that COUNT value. For example, Q1 generates many tuples (600,572 tuples for SF=0.1) from a small RTable R because Q1 is an aggregate query where R explicitly defines big COUNT values for each input tuple. For those queries which do not define a COUNT value, only a small number of tuples are generated because the trial-and-error phase starts from creating one output tuple per input tuple (e.g., Q6). In that case, the size of the generated database is independent of the scaling factor. As a summary, we see that the generated databases are already as small as possible. Huge databases are only generated by SPQR, if the query result explicitly states the size.

5.2. Running Time (SF=0.1)

Table 2a shows the running times of SPQR for the TPC-H benchmark with the scaling factor 0.1. In the worst case, the *Total* running time is up to one hour (Query 10). However, most queries can be reverse processed in a few seconds. Table 2a also shows the cost break-down of reverse query processing. `QP` is the time spent processing tuples in SPQR (e.g., constructing the constraint formulae and calling the *pushNext* function). For all queries (except Q1), this time is below a minute. Q1 is an exception, because it generates many tuples. `DB` shows the time that is spent by PostgreSQL in order to generate new tuples (processing SQL `INSERT` statements through JDBC). Obviously, this time is proportional to the size of the database instance generated as part of SPQR. The `MC` column shows the time spent by the decision procedure of the model checker. It can be seen that this time dominates the overall cost of RQP in most cases; in particular, it dominates the cost for the expensive queries (Q10 and Q11). This observation justifies the decision to focus all optimization efforts on calls to the decision procedure (Sections 4). `#M-Inv` shows the number of times the decision procedure is invoked. Comparing the `MC` and `#M-Inv` columns, it can be seen that the cost per call varies significantly. Obviously, the decision procedure needs more time for long constraints (e.g., Q10) than for simple constraints (e.g., Q22). As a future work, we hope to find a way to predict the cost per call in order to carry out even better optimization.

We also measured the number of attempts each TPC-H query needed for guessing the number of tuples in aggregations (Section 4). These results are not shown in Table 2, but the results are encouraging: in fact, none of the 22 required any trial-and-error (because OP 5 in Section 4.5 made it possible to pre-compute the right number of tuples for all queries) or trial-and-error was successful for $n = 1$.

5.3. Running Time: Varying SF

Table 2 (refer to the *Total* columns) shows the running times of reverse processing the 22 TPC-H queries for the three different scaling factors. In some cases, due to the nature of the queries, the running times (and the size of the generated databases) is independent of the scaling factor; example queries are Q5 and Q6. For all those queries which

[4]Implicit means that the COUNT value can be calculated by the optimization rule *OP 5* in Section 4.5.

have higher running times for a larger scaling factor, the running time increased linearly. Examples are queries Q10 and Q21. Again, these results are encouraging because they show that SPQR potentially scales linearly and that even large test databases can be generated using SPQR. Note that Q11 has a parameter that is set randomly; this observation explains the anomaly that the running time for SF=0.1 is higher than for SF=1 for that query.

6. Related Work

To the best of our knowledge, there has not been any previous work on reverse query processing. The closest related work is the work on model checking which finds instantiations of logical expressions. However, the model checking community has not addressed issues involving SQL and has not addressed any scalability issues that arise if millions of tuples need to be generated. Consequently, our solution combines techniques of traditional query processing (e.g., [13, 11]) with model checking [6].

In the area of generating test databases, [18] shows how functional dependencies can be processed for generating test databases. The bottom-up phase of RQP extends the work in [16] for the complete SQL specification. Other work on test databases generation (e.g., [21, 5]) focuses on one aspect only and falls short on most other aspects of RQP. [15] discusses a similar problem statement as RQP but only applicable to a very restricted set of relational expressions. There has also been work on efficient algorithms and frameworks to produce large amounts of test data for a given statistical distribution [12, 4]. That work is orthogonal to our work.

7. Conclusion and Future Work

This work presented a new technique called reverse query processing or RQP, for short. RQP combines techniques from traditional query processing (e.g., query rewrite and iterator model) and model checking (e.g., data instantiation based on constraint formulae of propositional logic). RQP has several applications. This paper shows that a full-fledged RQP system (SPQR) for SQL can be built in order to generate test databases for functional testing database applications. SPQR scales linearly with the size of the database that is generated for the TPC-H benchmark.

We believe that this work is only the first step into a new research direction. One future work is to explore the possible applications of RQP as mentioned in the introduction. Now, we are extending SPQR to support multiple SQL statements for complex applications. In addition, we are also building another RQP system for RDBMS performance testing.

References

[1] IBM DB2 Test Database Generator. http://www-306.ibm.com/software/data/db2imstools/db2tools/db2tdbg/.

[2] TPC-H Benchmark. http://www.tpc.org/tpch.

[3] C. Binnig, D. Kossmann, and E. Lo. Reverse Query Processing. Technical report, ETH Zurich, http://www.dbis.ethz.ch/research/publications/rqp.pdf, 2006.

[4] N. Bruno and S. Chaudhuri. Flexible Database Generators. In *VLDB*, pages 1097–1107, 2005.

[5] D. Chays, Y. Deng, and P. G. Frankl. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability*, 2004.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[7] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM*, 13(6):377–387, 1970.

[8] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification . In *CAV*, volume 3576, pages 296–300, 2005.

[9] C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, pages 571–581, 2001.

[10] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[11] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[12] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD*, pages 243–252, 1994.

[13] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *SIGMOD*, pages 377–388, 1989.

[14] J. M. Hellerstein and J. F. Naughton. Query Execution Techniques for Caching Expensive Methods. In *SIGMOD*, pages 423–434, 1996.

[15] T. Imielinski and J. Witold Lipski. Inverting relational expressions: a uniform and natural technique for various database problems. In *PODS*, pages 305–311, 1983.

[16] A. Klug. Calculating constraints on relational expression. *TODS*, 5(3):260–290, 1980.

[17] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *ICDE*, 2002.

[18] H. Mannila and K.-J. Räihä. Test Data for Relational Queries. In *PODS*, pages 217–223, 1986.

[19] G. Miklau and D. Suciu. A Formal Analysis of Information Disclosure in Data Exchange. In *SIGMOD*, pages 575–586, 2004.

[20] R. Muller and H. Kuchen. Generating Glass-Box Test cases using a symbolic virtual machine. In *ICSE*, 2004.

[21] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating Consistent Test Data for a Variable Set of General Consistency Constraints. *VLDB J.*, 2(2):173–213, 1993.

[22] K. A. Ross, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Foundations of Aggregation Constraints. In *Principles and Practice of Constraint Programming*, pages 193–204, 1994.