Reverse Query Processing

Carsten Binnig[‡] Donald Kossmann[†] Eric Lo[†]

[†] ETH Zurich, Switzerland

Email: {firstname.lastname}@inf.ethz.ch

[‡] University of Heidelberg, Germany

Email: carsten.binnig@informatik.uni-heidelberg.de

Abstract

Traditionally, query processing gets a query and a database instance as input and returns the result of the query for that particular database instance. Reverse query processing (RQP) gets a query and a result as input and returns a possible database instance that could have produced that result for that query. Rather than making a closed world assumption, RQP makes an open world assumption. There are several applications for RQP; most notably, testing database applications and debugging database applications. This paper describes the formal framework of RQP and the design of a system, called SPQR (System for Processing Queries Reversely) that implements a reverse query processor for SQL.

1 Introduction

In the last thirty years, a great deal of research and industrial effort has been invested in order to make query processing more powerful and efficient. New operators, data structures, and algorithms have been developed in order to find the answer to a query for a given database as quickly as possible. This paper turns the problem around and presents methods in order to efficiently find out whether a table can possibly be the result of a query and, if so, what the corresponding database might look like. 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.

Reverse query processing has several applications. First, it can be used in order to generate test databases. The generation of good test databases has been studied in the literature (e.g., [19, 20, 4]), but it is still an expensive process in practice. Second, RQP can be helpful to debug a database application because it enables programmers to deduce in which states a program with embedded SQL can get. In the same way, RQP is a tool that is needed in order to formally verify database application programs and, thus, a component in order to implement Hoare's Grand Challenge of provably correct software [15]. Other potential applications involve updating views and database sampling.

Reverse query processing is carried out in a similar way as traditional query processing. At compile-time, a SQL query is translated into an expression of the relational algebra, this expression is rewritten for optimization and finally translated into a set of executable iterators

[13]. At run-time, the iterators are applied to input data and produce outputs [11]. What makes RQP special are the following differences:

- Instead of using the relational algebra, RQP is based on a reverse relational algebra. Logically, each operator of the relational algebra has a corresponding operator of the reverse relational algebra that implements its reverse function.

- Correspondingly, RQP iterators implement the operators of the reverse relational algebra which requires the design of special algorithms. Furthermore, RQP iterators have one input and zero or more outputs (think of a query tree turned upside down). As a consequence, the best way to implement RQP iterators is to adopt a push-based run-time model, instead of a pull-based model which is typically used in traditional query processing [11].

- An important aspect of reverse query processing is to respect integrity constraints of the schema of the database. Such integrity constraints can impact whether a legal database instance exists for a given query and query result. In order to implement integrity constraints during RQP, this work proposes to adopt a two-step query processing approach and make use of a model checker at run-time in order to find reverse query results that satisfy the database integrity constraints.

- Obviously, the rules for query optimization and query rewrite are different because the cost tradeoffs of reverse query processing are different. As a result, different rewrite rules and optimizations are applied.

The main contribution of this paper is to address all these aspects and present initial solutions. Furthermore, this paper gives results of performance experiments for the TPC-H benchmark [1] using SPQR, a prototype system for reverse query processing, in order to demonstrate how well the proposed techniques scale. This work focuses on reverse query processing for SQL queries and the relational data model. Studying RQP for, say, XQuery and XML is one of many avenues for future work.

As will be shown, reverse query processing for SQL queries is challenging. For instance, reverse aggregation is a complex operation. Furthermore, model checking is an expensive operation even though there has been significant progress in this research area in the recent past. As a result, optimizations are needed in order to avoid calls to the model checker and/or make such calls as cheap as possible.

The remainder of this paper is organized as follows: Section 2 describes potential applications for RQP in more detail. Section 3 defines the problem and gives an overview of the solution. Section 4 describes the reverse relational algebra for reverse query processing. Sections 5 to 7 present the techniques implemented in SPQR. Section 8 describes the results of experiments carried out using SPQR and the TPC-H benchmark. Section 9 discusses related work. Section 10 contains conclusions and avenues for future work. Overall, we believe that this paper is just a starting point towards more sophisticated and more general solutions for reverse query processing.

2 Applications

2.1 Generating Test Databases

The application that started this work is the generation of test databases for regression tests or to test the specification of an application. The generation of test databases has been studied in previous work (e.g., [19, 20, 4]). Nevertheless, finding good test data for a new or evolving application is a daunting task because all previous work was limited to the processing of integrity constraints. The generation of data that meets the requirements of an application was not described. RQP can be the basis for a generic tool that generates a test database from the database schema (including integrity constraints) and other information such as the application code (white box testing) or a UML model and specification of the application (black box testing).

If the application code is available (e.g., Java with embedded SQL via JDBC or SQL-J), then the application code can be analyzed using data flow analysis in order to find all code paths [2]. Based on this information, RQP can be applied to the SQL statements which are embedded in the application in order to generate a test database that will provide data for all possible code paths. As a very simple example, consider the following pseudocode:

```
foreach x in SELECT a FROM R do
switch (x)
    case 1: do some work;
    case 2: do some work;
    do some work;
end foreach
```

In order to fully test this code fragment, it is necessary that Table R contains tuples whose value of Attribute “a” is 1, 2, and some other value. Reverse query processing can be used in order to generate exactly such a table.

Generating a test database for a decision support application is another example for which RQP is useful. For instance, the programmer gives a SQL query that specifies a data cube. Furthermore, the programmer provides one or several sample reports on that data cube. From the SQL query that defines the data cube and the sample reports, RQP can generate a test database which can be used as a basis for regression tests on future implementations of the reports and tests of other reports (with different levels of aggregation and filters). The performance experiments which are based on the TPC-H benchmark and which are reported in Section 8 are inspired by this application of RQP.

RQP can also be used in order to generate large test databases with different value distributions. Such test databases can be used in order to carry out performance or scalability tests. Efficient algorithms and a framework to implement such test databases have been devised in [12, 3]. That work is orthogonal to our work and these algorithms can be incorporated into the implementation of the operators of the RQP algebra in order to generate a test database that tests the performance and scalability of specific queries.

2.2 Other Applications of RQP

Apart from test database generation, RQP also has a lot of other potiential applications. This section presents some of them. However, additional research is required in order to further explore these applications.

SQL Debugger: Another practical application of RQP is to debug database applications with embedded SQL code. If a query produces the wrong query results, then RQP can be used to step-wise reverse engineer the query based on its query plan and find the operators that are responsible for the wrong query results; e.g., a wrong or missing join predicate.

Program Verification: As mentioned in the introduction, RQP can be an important component for Hoare's Grand Challenge project of program verification [15]. In order to prove the correctness of a program, all possible states of a program must be computed. In order to compute all states of a database program (e.g., Java plus embedded SQL), RQP is needed for finding all necessary conditions of the database in order to reach certain program states.

Updating Views: The SQL standard is conservative and specifies that only views on base tables without aggregates are updateable. Many applications make heavy use of SQL view definitions and, therefore, require a more relaxed specification of updateable views. For example, Microsoft's ADO.NET allows the client-side update of data, regardless of the kind of view that was used to generate that data. The reason why SQL is conservative is that updates to certain views are ambiguous. RQP can be used in order to find all possible ways to apply an update (possible infinitely many). Additional application code can then specify which of these alternatives should be selected.

Database Sampling, Compression: Some databases are large and query processing might be expensive even if materialization and indexing is used. One requirement might be to provide a compressed, read-only variant of a database that very quickly gives approximate answers to a pre-defined set of parametrized queries. Such database variants can be generated using RQP in the following way: First, take a sample of the queries (and their parameters) and execute those queries on the original (large) database. Then, use RQP on the query results and the sample queries in order to find a new (smaller) database instance.

3 RQP Overview

3.1 Problem Statement

As mentioned in the introduction, this paper addresses the following problem for relational databases. Given a 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:

$$R = Q(D)$$

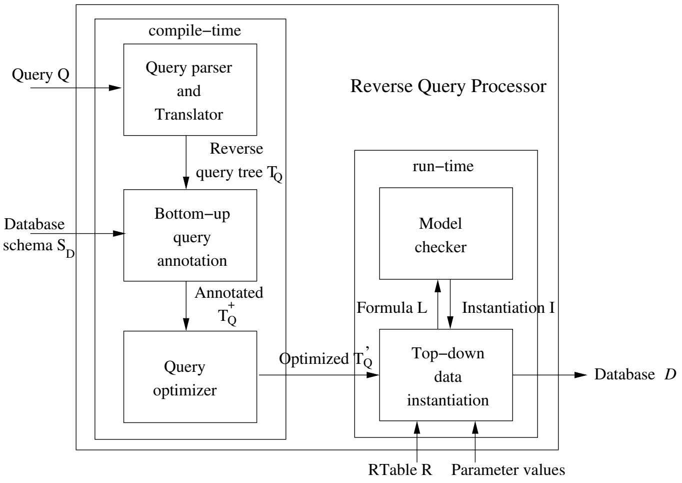


Figure 1: RQP Architecture

and D is compliant with S_D and its integrity constraints.

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. In order to generate test databases, for instance, it might be advantageous to generate a small D so that the running time of tests are reduced. While the design presented in the following sections tries to be minimal, the techniques do not guarantee any minimality. The purpose of this work is to find any viable solution. Studying techniques that make additional guarantees is one avenue for future work.

As shown in [17], the equivalence of two SQL queries is undecidable if the queries include the relational minus operator and if the queries do not follow a distinct operator order. As a result, RQP for SQL is also undecidable; that is, in general it is not possible to decide whether a D exists for a given R and Q if Q contains a minus operator. Furthermore, there are obvious cases where no D exists for a given R and Q (e.g., if tuples in R violate basic integrity constraints). The approach presented in this paper, therefore, cannot be complete. It is a best-effort approach: it will either fail (return an *error* because it could not find a D) or return a valid D.

3.2 RQP Architecture

Figure 1 gives an overview of the proposed architecture to implement reverse query processing. A query is (reverse) processed in four steps by the following components:

Parser: The SQL query is parsed into a query tree which consists of operators of the relational algebra. This parsing is carried out in exactly the same way as in a traditional SQL processor. What makes RQP special is that that 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 4. 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.

Bottom-up Query Annotation: The second step is to propagate schema information (types, attribute names, functional dependencies, and integrity constraints) to the operators of the query tree. Furthermore, properties of the query (e.g., predicates) are propagated to the operators of the reverse query tree. As a result, each operator of the query tree is annotated with constraints that specify all necessary conditions of its result. Section 5 describes this process in more detail. That way, for example, it can be guaranteed that a top-level operator of the reverse query tree does not generate any data that violates one of the database integrity constraints.

Query Optimization: In the last step of compilation, the reverse query tree is transformed into an *equivalent* reverse query tree that is expected to be more efficient at run-time. An example optimization is the unnesting of queries. Unnesting and other optimizations are described in Section 7.

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. In fact, some operators have alternative implementations depending on the application (e.g., test database generation involves different algorithms than program verification, Section 2). The result of this step is a valid database instance D. As part of this step, we propose to use a model checker (more precisely, the decision procedure of a model checker) in order to generate data [5]. How this Top-down data instantiation step is carried out is described in more detail in Section 6.

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. The choice of query parameters again depends on the application; for test database generation, for instance, it is possible to generate several test databases with different parameter settings derived from the program code. In this case, the first three phases of query processing need only be carried out once, and the Top-down data instantiation can use the same annotated reverse query tree for each set of parameter settings.

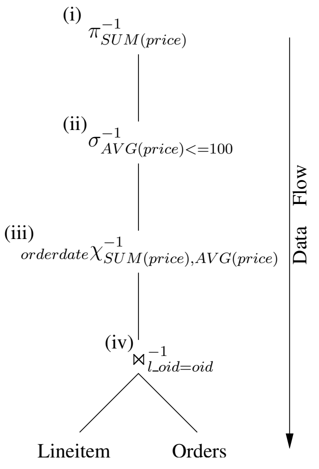
It is also possible to use variables in the RTable. That way, it is possible to specify tolerances. For example, a user who wishes to generate a test database for a decision support application could specify an example report for sales by product. Rather than specifying exact values in the example report, the user could say that the sales for, say, tennis rackets are x with $90K \leq x \leq 110K$. This additional constraint for variable x would be considered during the execution of Top-down data instantiation. Specifying such tolerances has two important advantages. First, depending on the SQL query it might not be possible to find a test database that generates a report with the exact value of 100K for the sales. That is, the RQP instance might simply not be satisfiable. Second, specifying tolerances (if that is acceptable for the application) can significantly speed-up reverse query processing because it gives the model

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

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

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

checker more options to find solutions.

3.3 RQP Example

Figure 2 gives an example for reverse query processing. Figure 2a shows the database schema (definition of the `Lineitem` and `Orders` tables with their integrity constraints) and a 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. The resulting reverse query tree is shown in Figure 2b. This tree is very similar to the query tree used in traditional query processors. The differences are that (a) operators of the reverse relational algebra (Section 4) are used and (b) that the data flow through that tree is from the top to the bottom (rather than from the bottom to the 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 `orderdates`). 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 randomly: depending on the application, alternative heuristics could be used in order to generate values that are more advantageous for the application. 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.

Figure 2 does not demonstrate how the Bottom-up query annotation component annotates the reverse query tree using the integrity constraints of the database schema and properties of the query. The example, however, does show the effects of that step. For example, the result of reverse projection (Table (ii) in Figure 2c) generates values for the `AVG(price)` column which are compliant with the predicate of the `HAVING` clause of the query. This process is described in more detail in Section 5.

4 Reverse Relational Algebra

The Reverse Relational Algebra (RRA) is a reverse variant of the traditional relational algebra [6] 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:

- *Basic Operators:* The reverse variants of the basic operators of the (extended) relational algebra (selection, projection, rename, Cartesian product, 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.
- *Algebraic Laws:* 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 for the RRA (e.g., applying projections before joins); these laws are listed in [17] and must be respected for RQP optimization (Section 7).

The remainder of this section defines the seven basic operators of the reverse relational algebra, which form the basis for a complete implementation of a reverse query processor. A physical implementation (e.g., algorithms) of the RRA operators for generating test databases is described in Section 6.

4.1 Reverse Projection

The reverse projection operator (π^{-1}) generates new columns according to its *output schema*. The output schema of an operator is defined as the set of attributes, constraints (from the database and from predicates of the query), and functional dependencies of the output relation generated by the operator. The output schema of each operator is generated in the Bottom-up annotation phase (Section 5). Again, as for all operators of the reverse relational algebra, $\pi(\pi^{-1}(R)) = R$ must apply for all valid R.

In Figure 2, the reverse projection creates the `orderdate` and `AVG(price)` columns. In order to generate correct values for these columns, the reverse project operator 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 (χ^{-1}) does not fail. In this specific example, there are no integrity constraints from the database schema or functional dependencies 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.

An algorithm to implement the π^{-1} operator is presented in Section 6. This algorithm is based on calls to the decision procedure of a model checker in order to fulfill all constraints or fail (i.e., return *error*), if the constraints cannot be fulfilled.

4.2 Reverse Selection

The simplest operator of the reverse relational algebra is the reverse selection (σ^{-1}): It either returns *error* or a superset (or identity) of its input. *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 salary greater than 10,000 and the RTable contains an employee with salary 1,000, then *error* is returned. Another example of σ^{-1} is given in Figure 2c. Table (ii) in Figure 2c (the output of π^{-1}) is the input of σ^{-1}. Since the input of σ^{-1} is compliant with its output schema, the output of σ^{-1} (Table (iii) in Figure 2c) is the same as its input.

4.3 Reverse Aggregation

Like the π^{-1} operator, the reverse aggregation operator (χ^{-1}) 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*. An algorithm to implement the χ^{-1} operator using the decision procedure of a model checker is presented in Section 6.

Tables (iii) and (iv) of Figure 2c show the input and output of reverse aggregation for 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 (top of Figure 2a). The value of the `price` column is generated using the input (the result of the reverse selection)

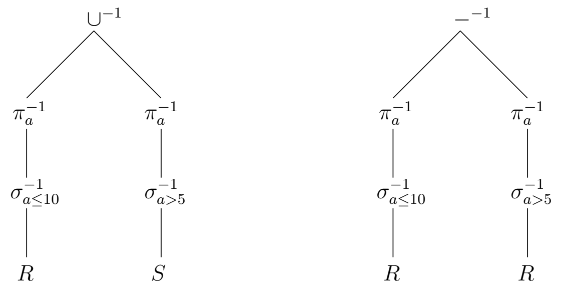


Figure 3: Reverse Union (left); Reverse Minus (right)

and the intrinsic mathematical properties of the aggregate functions. The values of the `l_oid` and `oid` columns are generated obeying the constraints imposed by the join predicate of the query and the `primary key` constraint of the `Orders` table.

4.4 Reverse Join, Cartesian Product

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, then an *error* is raised. Really, the only thing that is special about the $\bowtie^{-1}$ operator is that it has two outputs. Again, an efficient algorithm to implement a reverse join is presented in Section 6. The reverse Cartesian product is a variant of the reverse join with *true* as a join predicate.

4.5 Reverse Union

Like the reverse join, the reverse union operator ($\cup^{-1}$) takes one relation as input and generates two output relations. According to the constraints of the output schemas of the two output relations, the reverse union distributes the tuples of the input relation to the corresponding output relations. An example is given in the left part of Figure 3. Both relations R and S have an attribute a. Let the input for the reverse union be three tuples: $\langle 2\rangle, \langle 12\rangle, \langle 8\rangle$. In this case, the reverse union must output $\langle 2\rangle$ to the left reverse selection operator and output $\langle 12\rangle$ to the right selection operator. $\langle 8\rangle$ can be output to either the left or the right selection operator. If the input of a reverse union involves a tuple that does not fulfill the constraints of any branch (this is not possible in the example of Figure 3), then the reverse union fails and returns *error*.

4.6 Reverse Minus

An example for a reverse minus operator ($-^{-1}$) is shown in the right part of Figure 3. Input tuples are always routed to the left branch or result in an error. Furthermore, it is possible that the $-^{-1}$ generates new tuples for both branches in order to meet all its constraints. In this example, the reverse minus would output an input tuple $\langle 2\rangle$ (or any other input with $a \leq 5$) to its left branch, and it would return *error* if its input contains a tuple with $a > 5$. No new tuples need to be generated in this example.

4.7 Reverse Rename

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.

5 Bottom-up Query Annotation

The bottom-up query annotation phase in Figure 1 annotates each operator op^{-1} of a reverse query tree with an *output schema* S^{OUT} and an *input schema* S^{IN}. This way, each operator can check the correctness of the input and ensure that it generates valid output data. Both schemas (input and output) are defined by (1) the attributes A (names and data types), (2) the integrity constraints C, and (3) the functional dependencies F and join dependencies J (as well as multivalued dependencies as special cases of J). The join dependencies used in that work are a specialization of those known from textbooks like [10]. A join dependency JD in that work is defined as follows:

$$JD = (A_1, A_2, p)$$

A JD defines that the projection of the input R of an operator to the union of all attributes A_1, A_2 must represent a loseless join on the projections of R to A_1, A_2 using p as join predicate: $\pi_{A_1 \cup A_2}(R) = (\pi_{A_1}(R)) \bowtie_{p_1} (\pi_{A_2}(R))$. Join dependencies with more than two sets of attributes A_1, A_2 can be represented as a combination of these join dependencies. More details are given in the corresponding sections of each reverse operator.

A schema S in RQP is formally defined as the following four tuple:

$$S = (A, C, F, J)$$

RQP considers the integrity constraints of SQL (primary key, unique, foreign key, not null, and check) as well as aggregation constraints [21].

In the following $S.A$ denotes the set of attributes A defined in schema S. Similarly, $S.C$ denotes the set of constraints C in S and $S.F$ denotes the set of functional dependencies F in S. In order to denote the different constraint types in a schema S, we use $S.C_{CK}$, $S.C_{UN}$, $S.C_{PK}$, $S.C_{NN}$ and $S.C_{AGG}$ to denote different types of constraints in S. The notations are summarized in Table 1.

Obviously, a unary operator (e.g., σ^{-1}) has only one output schema whereas a binary operator (e.g., $\bowtie^{-1}$) has two output schemas. Analogous to the output schema, the input schema S^{IN} of an operator specifies the attributes, constraints and functional dependencies of the input of the operator. In a reverse query tree, the input and output schemas of operators must match; for example, the input schema of the σ^{-1} is the same as the output schema of the π^{-1} in the example of Figure 2b.

The annotation phase operates in a bottom-up way. It starts with the output schemas of the leaves of the reverse query tree (e.g., the operators that read the `Lineitem` and `Orders` in

Notation	Description
S	Schema S
$S.A$	Attributes in S
$S.C$	Constraints in S
$S.F$	Functional dependencies in S
C_{CK}	Check constraints
C_{UN}	Unique constraints
C_{PK}	Primary key constraints
C_{NN}	Not null constraints in
C_{AGG}	Aggregation constraints

Table 1: Notations used in the bottom-up phase

Figure 2b). The output schemas of these leaves are defined by the database schema (e.g., the SQL DDL code of Figure 2a). Then, for each operator, the input schema is computed from the output schema of the operator. This input schema is then used to initialize the output schema of the operator at the next level up.

The remainder of this section defines a full set of rules for the annotation of each RRA operator and shows how this bottom-up phase works for each operator of the example in Figure 2. Furthermore, we also show how the bottom-up phase works for nested queries. In this regard, our work is an extension of the work presented in [17]; that work describes how functional dependencies and check constraints expressing the equality can be propagated for expressions of the relational algebra. We extend that work for all elements (A, C, and F) contained in a schema S and the aggregation operator. As shown later, the primary key and the unique constraints in C can be derived from F and the not null constraints in S. Furthermore, the rules introduced in the sequel use full qualified attribute names (relation name and attribute name) instead of the position of an attribute in a relation (which is used in [17]) in order to identify the attributes uniquely. Another extension is that we assume *bag* semantics, as in SQL.

5.1 Leaf initialization

As stated above, the output schemas of the leaves of the reverse query tree are initialized using the database schema S_D. We assume that a database schema which is used as input of the bottom-up annotation phase (see Figure 1) defines a schema $S_R = (A, C, F)$ for each relation R. In order to initialize a leaf of a RRA expression representing a relation R, the bottom-up phase must extract the corresponding schema S_R out of the database schema S_D.

`FOREIGN KEY` constraints defined in the output schema are treated specially in the bottom-up phase. They are rewritten as a reverse equi-join with a join predicate representing the key/foreign-key relation.

Example: Assume the table `Lineitem` in the example of Figure 2a defines a foreign key on the attribute `l_oid` which refers the primary key attribute `oid` of the relation `Orders`. The query

```
SELECT name FROM Lineitem WHERE price>100
```

would then be rewritten as

```
SELECT name FROM Lineitem, Orders WHERE price>100 and l_oid=oid.
```

For the rest of the input schema elements of a leaf, they are the same as the elements of the leaf's output schema. For the example of Figure 2b, the input and output schemas of `Lineitem` and `Orders` can be represented in the following way (there are no `UNIQUE`, `FOREIGN KEY`, and `NOT NULL` constraints in this example):

A:	*lid* INTEGER,	A:	*oid* INTEGER
	name VARCHAR(20),		*orderdate* DATE
	...		
	l_oid INTEGER		
C:	PRIMARY KEY(*lid*)	C:	PRIMARY KEY(*oid*)
	CHECK(1 $\geq$ discount $\geq$ 0)		
F:	$\{lid\} \rightarrow \{name, price, ..., l\_oid\}$	F:	$\{oid\} \rightarrow \{orderdate\}$
	$S_{Lineitem}$		S_{Orders}

5.2 Reverse Join

The reverse join has two output schemas called $S^{OUT}_{left}$and S^{OUT}_{right}. Its input schema S^{IN} is computed from these two output schemas by the following rules:

(1) $S^{IN}.A = S^{OUT}_{left}.A \cup S^{OUT}_{right}.A$;

(2) $S^{IN}.F = closure(S^{OUT}_{left}.F \cup S^{OUT}_{right}.F \cup$ `createFD(`p`)`);
– p denotes the join predicate;
– `createFD(`p`)` is a function to create functional dependencies from predicates (see Figure 4).
– the function $closure$ is the function to compute the closure of a given set of functional dependencies in [17].

(3) $S^{IN}.J = S^{OUT}_{left}.J \cup S^{OUT}_{right}.J \cup JD(S^{OUT}_{left}.A, S^{OUT}_{right}.A, p)$

(4) $S^{IN}.C$ is defined for each type as follows:

(4.1) $S^{IN}.C_{CK} = S^{OUT}_{left}.C_{CK} \cup S^{OUT}_{right}.C_{CK} \cup p$;
– p denotes the join predicate;

(4.2) $S^{IN}.C_{NN} = S^{OUT}_{left}.C_{NN} \cup S^{OUT}_{right}.C_{NN}$;

(4.3) $(S^{IN}.C_{PK}, S^{IN}.C_{UN})$ =
`createPKAndUnique(`$S^{IN}.F, S^{IN}.C_{NN}, S^{IN}.A$`);`
– `createPKAndUnique`is a function to create primary key and unique constraints from functional dependencies, not null constraints, and attributes (see Figure 5).

```
createFD(Predicate $p$)
Output:
 -Set $F$ // Set of functional dependencies
(1)  //transform $p$ to conjunctive normal form
(2)  //$cnf\_p = p_{OR1} \wedge ... \wedge p_{ORn}$
(3)  $cnf\_p$ = CNF($p$)
(4)  //Analyze each conjunct $p_{ORi}$
(5)  FOREACH $p_{OR}$ in $cnf\_p$
(6)   //domain equality:  $a_i = a_j$
(7)   //value equality $a_i = c$;
(8)   //$a_i$,$a_j$ are attributes; $c$ is a constant
(9)   IF($p_{OR}$ is domain equality)
(10)    //e.g.  add ($\{a_i\} \rightarrow \{a_j\}$), ($\{a_j\} \rightarrow \{a_i\}$)
(11)    $F$.add($\{p_{OR}.leftAtt()\} \rightarrow \{p_{OR}.rightAtt()\}$)
(12)    $F$.add($\{p_{OR}.rightAtt()\} \rightarrow \{p_{OR}.leftAtt()\}$)
(13)  ELSE IF($p_{OR}$ is value equality)
(14)    //e.g.  add ($\emptyset \rightarrow \{a_i\}$)
(15)    $F$.add($\emptyset \rightarrow \{p_{OR}.leftAtt()\}$)
(16)  //ELSE do nothing for complex predicates
(17)  END IF
(18) END FOR
(19) RETURN F
```

Figure 4: Function `createFD`

(4.4) $S^{IN}.C_{AGG} = S^{OUT}_{left}.C_{AGG} \cup S^{OUT}_{right}.C_{AGG}$;

The set of attributes $S^{IN}.A$ of the input schema is the union of the set of attributes from the reverse join's output schemas (rule 1).

The functional dependencies $S^{IN}.F$ of the input schema are defined as the closure of the union of the functional dependencies in the reverse join's output schemas and the functional dependencies computed from the join predicate by the function `createFD` in Figure 4 (rule 2). The function `createFD` takes a predicate p as input and outputs a set of derivable functional dependencies. This function deals with arbitrary predicates by transforming the given predicate into conjunctive normal form (line 3 in Figure 4). The conjunctive normal form of a predicate consists of one or more conjuncts, each of which is a disjunction (OR) of one or more literals (simple predicates with no boolean operator). Afterwards each conjunct is analyzed separately (line 5 in Figure 4). In case that the conjunct only consists of a simple predicate expressing the equality, it is transformed into a set of functional dependencies (line 9 to 15).

The join dependencies $S^{IN}.J$ of the input schema are defined as set of the join dependencies in the reverse join's output schemas and a new join dependency computed from the attributes of both output schemas and the join predicate p (rule 3). Thus we are able to express joins on joined relations.

The `CHECK` constraints (rule 4.1) are the union of the `CHECK` constraints from the output schemas and the join predicate. The `NOT NULL` constraints (rule 4.2) are the union of the `NOT`

createPKAndUnique(Functional dependencies F, Not null constraints NN, Attributes A)

Output:

-Set PK // Set of primary key constraints

-Set UN // Set of unique constraints

```
 (1) $PK = UN = \emptyset$
 (2) //analyze $F$
 (3) FOREACH $f$ in $F$
 (4)  //if all attributes $A$ are in
 (5)  //attributes of right side of $f$
 (6)  IF($A$ - $f$.rightAtts() = $\emptyset$)
 (7)   IF($NN$ has a constr.  for $f$.leftAtts())
 (8)    $PK$.add(PK($f$.leftAtts()))
 (9)   ELSE
(10)    $UN$.add(UNIQUE($f$.leftAtts()))
(11)   END IF
(12)  END IF
(13) END FOR
(14) RETURN ($PK$,$UN$)
```

Figure 5: Function `createPKAndUnique`

`NULL` constraints of the output schemas. The unique constraints and primary key constraints can be derived from F and the join predicate (rule 4.3). The function `createPKAndUnique` (see Figure 5) used by that rule takes a set of functional dependencies F, a set of not null constraints NN, and a set of attributes A as input and outputs all primary key and unique constraints implied by F and the not null constraints NN. A functional dependency f expresses a unique or primary key constraint on the set of attributes A, if all attributes A appear in the right side of the functional dependency (line 6 in Figure 5). When there are `NOT NULL` constraints on the left side of f, then a `PRIMARY KEY` constraint is added for the attributes; else a `UNIQUE` constraint is added for the attributes (line 7–10 in in Figure 5). The `AGGREGATION` constraint (rule 4.4) is a new type of constraint which is explained in the following section. These constraints are also computed as union of the AGGREGATION constraints of the two output schemas.

Going back to the example of Figure 2, the two output schemas of the $\bowtie^{-1}$ are given by the input schemas of `Lineitem` and `Orders`. Following the complete set of rules for $\bowtie^{-1}$, the resulting input schema of the $\bowtie^{-1}$ can be represented as follows:

A: lid INTEGER, . . ., l_oid INTEGER, /*from Lineitem*/
oid INTEGER, $orderdate$ DATE /*from Orders*/
C: PRIMARY KEY(lid), /*from Lineitem*/
CHECK(1 $\geq$ discount $\geq$ 0), /*from Lineitem*/
CHECK($oid = l\_oid$) /*join predicate*/
F: $\{lid\} \rightarrow \{name, ..., oid, orderdate\}$,
$\{oid\} \rightarrow \{orderdate\}$,
$\{l\_oid\} \rightarrow \{oid\}$,
$\{oid\} \rightarrow \{l\_oid\}$
J: $JD(\{lid, ..., l\_oid\}, \{oid, orderdate\}, lid = l\_oid)$,

5.3 Reverse Aggregation

The input schema of a reverse aggregate operator is defined by the following rules:

(1) $S^{IN}.A = A_{gr} \cup A_{agg}$;
–A_{gr} denotes the GROUP BY attributes,
–A_{agg} denotes the attributes of the new aggregate columns of the SELECT and HAVING clause

(2) $S^{IN}.F = closure($cleanFD$(S^{OUT}.F, A_{gr}) \cup \{A_{gr} \rightarrow A_{agg}\})$;
– cleanFD is a function to filter unrelated FDs (see Figure 6).

(3) $S^{IN}.J =$ cleanJD$(S^{OUT}.J, A_{gr})$
– cleanJD is a function to filter unrelated JDs (see Figure 7).

(4) $S^{IN}.C$ is defined for each type as follows:

(4.1) $S^{IN}.C_{CK} =$
cleanConstraints($S^{OUT}.C, (S^{IN}.A \cup A_{agg}), CK$);
– cleanConstraints is a function to clean constraint (see Figure 8).

(4.2) $S^{IN}.C_{NN} =$
cleanConstraints($S^{OUT}.C, (S^{IN}.A \cup A_{agg}), NN$) $\cup$
createNotNull($A_{agg}, S^{OUT}.C_{NN}$);
– createNotNull is a function to create not null constraints (see Figure 10).

(4.3) $S^{IN}.C_{AGG} =$
cleanConstraints($S^{OUT}.C, (S^{IN}.A \cup A_{agg}), AGGREGATION$) $\cup$
AGGREGATION(GROUP BY A_{gr}, A_{agg})

(4.4) $(S^{IN}.C_{PK}, S^{IN}.C_{UN}) =$
createPKAndUnique($S^{IN}.F, S^{IN}.C_{NN}, S^{IN}.A$);
– createPKAndUnique: a function to create primary key and unique constraints from functional dependencies, not null constraints, and attributes (see Figure 5).

The attributes, A, are given by the attributes in the GROUP BY clause of the query plus the aggregate columns specified in the SELECT and HAVING clause of the query (rule 1).

```
cleanFD(Functional dependencies $F_{OUT}$, Attributes $A_{OUT}$)
 Output:
  -Set $F_{IN}$ // Cleaned functional dependencies
 (1)  $F_{IN} = \emptyset$
 (2)  //analyze FDs in $F^{OUT}$
 (3)  FOREACH $f$ in $F^{OUT}$
 (4)   //if left attributes of $f$ are in $A^{IN}$
 (5)   IF($f$.leftAtts() $\cap$ $A^{IN}$ != $\emptyset$)
 (6)    $f$.rightAtts() = $f$.rightAtts() $\cap$ $A^{IN}$
 (7)    $F^{IN}$.add($f$)
 (8)   END IF
 (9)  END FOR
(10)  RETURN $F^{IN}$
```

Figure 6: Function cleanFD

The computation of F is listed in rule 2. It first uses the function cleanFDs (Figure 6) to keep only functional dependencies f with at least one of the attributes of the left side of f in the input schema (line 5 to 6). Then a new functional dependency which expresses that all aggregate columns are functional dependent from the attributes in the GROUP BY clause is added. If no GROUP BY clause exists, an empty set is used as left side of the new functional dependency.

The computation of J is shown by rule 3. It uses the function cleanJD (Figure 7) to keep only those attributes in a join dependency j with at least one of the attributes in the GROUP BY clause.

The check and not null integrity constraints (rule 4.1 and rule 4.2) are inherited from the output schema only if they are correlated to any attribute of the input schema or a metric attribute of the aggregation function. The function cleanConstraints (Figure 8) takes a set of integrity constraints C^{OUT}, a set of attributes A^{IN}, and the type t of constraints (e.g. NN) which should be returned by that function as input and outputs those integrity constraints C^{IN} of the given type which are correlated to any attribute in A^{IN}. In order to find correlated integrity constraints, it invokes a function createConstraintGraph (Figure 9) to create a constraint graph (line 2 in Figure 8) whose vertices represent the given integrity constraints in C^{OUT} and whose edges show if two constraints refer to at least one common attribute. The function keeps all integrity constraints which are connected to an integrity constraint, which refers to at least one attribute in A^{IN} (line 9 to 16).
The aggregation constraint (represented by $S^{IN}.C_{AGG}$) in Rule 4.3 is a new type of constraint introduced in [21]. An aggregation constraint specifies the requirements of the aggregation functions and the GROUP BY clause. They are also computed by the function cleanConstraints plus a new aggregation constraint for that opertator. The primary key and unique constraints (rule 4.4) can be derived from F, as already described for the reverse join.

In the example of Figure 2, the output schema of the χ^{-1} is given by the input schema of the $\bowtie^{-1}$. The input schema of the χ^{-1} is specified as follows.

```
cleanJD(Join dependencies $J_{OUT}$, Attributes $A_{OUT}$)
Output:
 -Set $J_{IN}$ // Cleaned join dependencies
$J_{IN} = J^{OUT}$.clone()
//analyze JDs in $J^{IN}$
FOREACH $j$ in $J^{IN}$
 //analyze $A_1$ and $A_2$ in $j$ given by $j$.atts()
 FOREACH set $A$ in $j$.atts()
  //remove attributes not in $A^{OUT}$ minus attributes used in $p$
  $A$ = $A$ $\cap$ ($A^{OUT} - j.p.atts()$)
  //remove $j$ from $J_{IN}$ if $A$ is empty
  IF($A$ = $\emptyset$) $J_{IN}$ = $J_{IN}$ - $j$
 END FOR
END FOR
RETURN $J^{IN}$
```

Figure 7: Function cleanJD

```
cleanConstraints(Constraints $C^{OUT}$, Attributes $A^{IN}$, Type $t$)
Output:
 -Set $C^{IN}$ // Cleaned integrity constraints
//create constraint graph of $C^{OUT}$
$G^{OUT}$ = createConstraintGraph($C^{OUT}$)
$G^{IN}$ = ($\emptyset$, $\emptyset$)
//analyze attributes $A^{IN}$
FOREACH $a$ in $A^{IN}$
 //analyze constraints of $G^{OUT}$
 FOREACH $c$ in $G^{OUT}.E$
  //if $a$ is in attributes of $c$
  IF($a \in c.atts()$)
   //subgraph calculates all constraints
   //connected to vertex c in $G^{OUT}$
   $G^{SUB}$ = $G^{OUT}$.subgraph($c$)
   //add constraints to $G^{IN}$
   $G^{IN}$.add($G^{SUB}$)
   $G^{OUT}$.remove($G^{SUB}$)
  END IF
 END FOR
END FOR
//the vertices of $G^{IN}$ are the constraint
$C^{IN}$ := $G^{IN}.V$
IF($t$!=$NULL$) RETURN $C_t^{IN}$ //e.g. $C_{NN}^{IN}$
ELSE RETURN $C^{IN}$
```

Figure 8: Function cleanConstraints

```
createConstraintGraph(Set $C$)
Output:
 -Graph $G = (V, E)$ // Graph of correlated constraints

$V$ = $\emptyset$
$E$ = $\emptyset$
//integrity constraints in set $C$
FOREACH $c$ in $C$
  FOREACH $c'$ in $V$
    //if $c$ and $c'$ have common attributes
    IF($c.atts() \cap c'.atts() != \emptyset$)
      $E$.add($c$, $c'$)
    END IF
  END FOR
$V$.add($c$)
END FOR
RETURN $G = (V, E)$
```

Figure 9: Function `createConstraintGraph`

A: $orderdate$ DATE, SUM($price$) FLOAT, AVG($price$) FLOAT
C: PRIMARY KEY ($orderdate$),
AGGREGATION(GROUP BY $orderdate$,
$\{$SUM($price$), AVG($price$)$\}$)
F: $\{orderdate\} \rightarrow \{$SUM($price$), AVG($price$)$\}$

5.4 Reverse Selection

The input schema of a reverse selection inherits A, F, C, and J from its output schema. The only difference between the output and input schema is that the selection predicate is added to the CHECK constraints of the input schema. The selection predicate is translated into corresponding functional dependencies in the same way as for the predicates of a reverse join (see Figure 4).

In the example of Figure 2, the input schema of the σ^{-1} is almost identical with the input schema of the χ^{-1} (previous paragraph): only, CHECK(AVG(price) $\leq$ 100) is added to the constraints (C).

5.5 Reverse Projection

The π^{-1} operator has similar rules as the χ^{-1} operator. The rules are as follows:

(1) $S^{IN}.A = A_{proj}$;
– A_{proj} denotes the projected attributes.

```
createNotNull(Aggr. constraints $C_{AGG}$, Not Null constraints $C_{NN}$)
Output:
 // Not null constraints for aggregation functions
 -Set $C_{NN'}$

$C_{NN'} = \emptyset$
//analyze aggregation functions in $AGGS$
FOREACH $agg$ in $AGG$
 //if all metrics are NOT_NULL
 IF($agg$.atts() - $C_{NN}.atts()$ = $\emptyset$)
  $C_{NN'}$.add(NOT_NULL($agg$))
 END IF
END FOR
RETURN $C_{NN'}$
```

Figure 10: Function `createNotNull`

(2) $S^{IN}.F$ = `cleanFD`($S^{OUT}.F$, $S^{IN}.A$);
– `cleanFD` is the same function as before, see Figure 6.

(3) $S^{IN}.J$ = `cleanJD`($S^{OUT}.F$, $S^{IN}.A$)
– `cleanJD` is a function to filter unrelated JDs (see Figure 7).

(4) $S^{IN}.C$ = `cleanConstraints`($S^{OUT}.C$, $S^{IN}.A, NULL$);
– `cleanConstraints` is the same function as before, see Figure 8.

The attributes of the input schema (rule 1) are derived from the attributes in the `SELECT` clause. The functional and join dependencies (rule 2 and 3) are calculated by the functions `cleanFD` and `cleanJD` just like in reverse aggregation. Also, the integrity constraints (rule 4) are calculated by the function `cleanConstraints` which keep all constraints correlated to a given set of attributes in this case for all attributes of the input schema.

In the example of Figure 2, the input schema of the π^{-1} is as follows.

```
$A$:  SUM(price) FLOAT
$C$:  CHECK(AVG(price) ≤ 100),
      AGGREGATION(GROUP BY orderdate,
                 {SUM(price), AVG(price)} )
$F$:  $\emptyset$
```

In the example, the `CHECK` constraint is correlated to the `AGGREGATION` constraint, Thus, it is kept in the input schema, although the attribute `AVG(price)` itself is not kept. The reason is that the function `createConstraintGraph` which is used in order to calculate the correlated constraints calls a method $atts()$ of each integrity constraint in the output schema (see Figure 9 line 7). This method call on an `AGGREGATION` constraint returns all aggregated columns (e.g. `AVG(price)`)plus all metrics of the aggregation functions (e.g. `price`). Thus constraints correlated to all aggregation functions and metrics are kept in the input schema.

5.6 Reverse Union

The reverse union has two output schemas like the reverse join. Its input schema is computed from the two output schemas by the following rules.

(1) $S^{IN}.A$ = $S^{OUT}_{left}.A$;

(2) $S^{IN}.F$ = $S^{OUT}_{left}.F \cap S^{OUT}_{right}.F$;

(3) $S^{IN}.J$ = $JD(S^{OUT}_{left}.J \cap S^{OUT}_{right}.J)$;

(4) $S^{IN}.C$ is defined for each type as follows:

(4.1) $S^{IN}.C_{CK} = (S^{OUT}_{left}.C_{CK}) \vee (S^{OUT}_{right}.C_{CK})$;

(4.2) $S^{IN}.C_{NN} = S^{OUT}_{left}.C_{NN} \cup S^{OUT}_{right}.C_{NN}$;

(4.3) $S^{IN}.C_{AGG} = S^{OUT}_{left}.C_{AGG} \cup S^{OUT}_{right}.C_{AGG}$;

(4.4) ($S^{IN}.C_{PK}$, $S^{IN}.C_{UN}$) =
createPKAndUnique($S^{IN}.F$, $S^{IN}.C_{NN}$, $S^{IN}.A$);
see Figure 5 for the algorithm

The set of attributes A of the input schema is equal to the set of attributes of its left output schema (rule 1), if the attribute types of both output schemas match.

The functional dependencies in the input schema (rule 2) are computed by the intersection of the functional dependencies of the two output schemas. Rule 3 states that the input must be a result of the set union of two relations with the given join dependencies.

The derivation of the CHECK constraints is more complex (rule 4.1): the set of CHECK constraints of the input schema is computed by combining the set of CHECK constraints from the left output schema with the set of CHECK constraints from the right output schema disjunctively. However, as the attribute names could be different in the right output schema they have to be renamed to the corresponding attribute (by position) of the left output. The NOT NULL and AGGREGATION constraints are computed by an intersection of these constraints of both output schemas (rule 4.2 and rule 4.3). The primary key and unique constraints (rule 4.4) are again derived from F, as already described for the reverse join.

Example: In the reverse union example in Figure 3, the new CHECK constraints $(R.a \leq 10) \vee (S.a > 5)$ of the input schema of the reverse union is defined by the check constraints of the two output schemas ($(R.a \leq 10)$ and $(S.a > 5)$). We can see that the the attributes of the check constraint of the right output schema are renamed.

5.7 Reverse Minus

According to [17], computing constraints for the minus operator in relational algebra is not complete and thus undecidable.

The reverse minus operator has also two output schemas. To derive the input schema we generally consider its left output schema only. The schema computation for the reverse minus operator is given by the following rules:

(1) $S^{IN}.A = S^{OUT}_{left}.A$

(2) $S^{IN}.F = S^{OUT}_{left}.F$

(3) $S^{IN}.J = JD(S^{OUT}_{left}.J - S^{OUT}_{right}.J)$;

(4) $S^{IN}.C = S^{OUT}_{left}.C \cup \neg S^{OUT}_{right}.C_{CK}$;

The set of attributes of the input schema as well as all functional dependencies and other integrity constraints are equal to the left output schema (rule 1, 2 and 4). In addition, a `CHECK` constraint which is the negation of the conjunction of all `CHECK` constraint predicates of the right output schema which involve a relation of the left branch is added. Rule 3 states that the input must be a result of the set difference of two relations with the given join dependencies. In the reverse minus example in Figure 3, a `CHECK` constraint `!(b>5)` is added to the input schema of reverse minus but only if the check constraints involves a relation in the left branch.

5.8 Reverse Rename

To derive the input schema we only rename the corresponding attribute respectively relation names of the output schema in A, C, and F.

5.9 Annotation of Nested Queries

In order to reverse process a nested query, SPQR uses the concept of nested iterations (sometimes called apply operators) which are known from traditional query processing [8], in a reverse way (see Section 6.10). A nested query has the following general structure:

```
OUTER QUERY bind predicate INNER QUREY correlation predicate.
```

In many cases, the bottom-up phase can be applied to the outer and inner query block separately. However, if the inner query is a query connected by equality (*bind predicate*) to the outer query, then the reverse apply operator adds an additional functional dependency to the outer query. In case that the inner query is correlated to the outer query, the *correlation predicate* must express the equality, otherwise no functional dependency is added to the outer query. The functional dependency added by the reverse apply operator has the following structure: $\{correlation\_attribute\} \rightarrow \{bind\_attribute\}$, where *correlation_attribute* and *bind_attribute* are the attributes of the outer query used in the *correlation predicate* and the *bind predicate*.

Example: Assume the query

```
SELECT s.age, s.salary FROM Student s WHERE s.age =
SELECT MAX(age) FROM Professor p WHERE p.salary=s.salary
```

is given. Then, a functional dependency $\{s.salary\} \rightarrow \{s.age\}$ is added to the output schema of the outer query.

6 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 under the decidability concerns as mentioned in Section 3.1. 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, 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 scalability testing. This section presents the physical algebra of SPQR, a prototype of RQP. The physical algebra of SPQR tries to keep the generated database as small as possible.

Moreover, 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 that 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).

Example: That problem can be shown by the following example query which should be reverse processed disregarding the rule above (Table S has the attributes A, B). We see that table S is referenced multiple times.

```
SELECT S1.A,S1.B,S2.A,S2.B
FROM S as S1, S as S2
WHERE S1.B=S2.B AND
S1.A>5 AND S2.A<=5;
```

Assume that a result R is given which has only one tuple `<6,1,5,1>`. The reverse query tree for that query contains two reverse selections (one on $S1$ and one on $S2$). The reverse selection `A>5` on $S1$ pushes `<6,1>` down to $S1$ and creates an additional tuple which satisfies `!(A>5)`, e.g. `<5,2>` for $S1.A, S1.B$. The reverse selection `A<=5` on $S2$ pushes `<5,1>` down to $S2$ and creates an additional tuple which satisfies `!(A<=5)`, e.g. `<6,2>` for $S2.A, S2.B$. So at the end S would contain four tuples `<6,1>,<5,2>,<5,1>,<6,2>`. If we run the query above on the generated tuples in S the result would contain two tuples `<6,1,5,1>`, `<6,2,5,2>` and not only one tuple as defined.

The remainder of this section is organized as follows. At the beginning we introduce the general architecture model used to implement the physical RRA operators. Afterwards a non blocking implementation is shown for each RRA operator which can be used in most cases. Some special cases which need a blocking implementation, as well as the reverse processing of nested queries are discussed afterwards. Finally, optimizations for some RRA operators are presented.

6.1 Iterator Model

As in traditional query processing, each operator 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 root operator 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 [18]. 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.

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.

6.2 Reverse Projection

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 constraint 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
```

```
$\pi^{-1}$.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 11: Method `pushNext` of π^{-1}

This constraint formula is passed to the model checker which in turn generates values for all variables or *error* if no instantiations that satisfy the formula can be found. 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 11 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 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 12. 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 8) 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

```
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 12: Function `instantiateData` (simplified)

designed for aggregation so that this guessing must be carried out as part of the *instantiateData* function in a trial and error phase (Lines 6 to 11). 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 and able to instantiate data.

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

```
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:[2]

[1] 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.

[2] 20060731 is the integer representation of the date July 7, 2006.

```
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 12) 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 12). Furthermore, in order to avoid the guessing, several optimizations can be applied (Section 6.11). These optimization techniques work very well such that in practice not much guessing is required; in fact, the experimental results in Section 8 show that no guessing is required for the whole TPC-H benchmark.

The pseudocode of Figure 12 is a simplification for the special case that there are no nested aggregations (e.g., `SUM(AVG(price))`) and no joins on aggregated values (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.

6.3 Reverse Aggregation

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 13 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)).

6.4 Reverse Join

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. 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 Section 6.9.1. In any event, the implementation of reverse joins and Cartesian products does not involve calls to a model checker so that these operators are much cheaper than reverse projections and aggregations.

```
$\chi^{-1}$.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 13: Method pushNext of χ^{-1}

```
$\cup^{-1}$.pushNext(Tuple $t$)
 (1)  //Create constraint formulas
 (2)  $L_{left}$:=createConstraint($t$,$S^{OUT}_{left}$)
 (3)  $L_{right}$:=createConstraint($t$,$S^{OUT}_{right}$)
 (4)  //call model checker
 (5)  IF(decisionProcedure($L_{left}$)!=NULL)
 (6)   left_operator.pushNext($t$)
 (7)  //call model checker
 (8)  ELSE IF(decisionProcedure($L_{right}$)!=NULL)
 (9)   right_operator.pushNext($t$)
(10)  ELSE
(11)   return $error$
(12)  END IF
```

Figure 14: Method pushNext of $\cup^{-1}$

6.5 Reverse Selection

The simplest implementation of the σ^{-1} operator would return its input (i.e., implement the identity function). For example in Figure 2c, the σ^{-1} implements the identity function such that its output relation (Table (iii) in Figure 2c) is identical to its input relation (Table (ii) in Figure 2c). If any input tuple is not compliant with the output schema, then *error* is returned.

6.6 Reverse Union

Like the reverse join, the reverse union operator takes one relation as input and generates two output relations. According to the output schemas of the two output relations, the reverse union operator distributes the input tuples to the correct output relation.

Figure 14 shows a implementation of the reverse union. The implementation checks for each input tuple if it is complaint with the output schema of the left output relation by creating a constraint formula representing the input tuple and the constraints imposed by the output schema (Line 1 to 3); and pushes the tuple to the left output relation if they are compatible (line 6). Otherwise, the reverse union checks the compatibility of the input tuple with the right output relation (line 8). If an input tuple is not complaint with any output relations, then *error* is returned (line 11). Obviously, the reverse union implementation is cheap: its complexity is linear to the input size.

6.7 Reverse Minus

The implementation of the reverse minus operator is similar to the reverse union operator. It checks for each input tuple if it is compliant with the left output schema but not the right output schema; and pushes the input tuple to the left output if possible. Otherwise, it returns *error*. Again, the complexity of this implementation is linear to the input size just like the reverse union operator.

6.8 Reverse Rename

Since the reverse rename operator does not have any data manipulation, its implementation is the same as the reverse selection: it returns identity.

6.9 RRA Operators in Special Cases

The implementations of the operators discussed so far are all non-blocking. That is, whenever an operator takes in a tuple, the operator can push the result tuple(s) to the child output operator immediately after processing. However, in some very special cases, RQP needs to use blocking RRA operators in order to guarantee correctness and they are discussed in details in this section. These special cases, however, are very rare in practice. For example, the TPC-H benchmark used in the experiments does not have any of the special cases and all non-blocking operators described above were used in the experiments.

6.9.1 Reverse Join

As discussed before the reverse equi-join that involves a primary key or an attribute with a `UNIQUE` constraint is trivial. However, all other reverse joins need more complex blocking implementations which are shown in the following.

Case 1: If the join predicate expresses the equality of two attributes ($a_i = a_j$) and both a_i and a_j are not the primary key or an attribute with a `UNIQUE` constraint of the output schemas, then a blocking implementation of the reverse join operator is needed.

The blocking implementation is shown in Figure 15. First, the complete input relation is grouped by the attributes of the left output schema (line 1). Afterwards each group is analyzed (line 3 to 24). If the group does not fulfill the join predicate an *error* is returned (line 5 to 7). Afterwards, the left and right output are created for that group (line 10,11). If any of both outputs (in the algorithm we use the left output) of the previous group has the same value for the join attribute as the current group, then the current right output must be the same as the previous right output; else an *error* is returned (line 14 to 19). If all is ok, the current left and

```
$\bowtie^{-1}$.pushNext(Relation $r$)
(1)  $r\_groups$ := groupby($r$, $S^{OUT}_{left}.A$)
(2)  //analyze each group in $r\_groups$
(3)  FOREACH $r\_group$ in $r\_groups$
(4)   //check join predicate
(5)   IF($r\_group$ not fulfills $\bowtie^{-1}$.p)
(6)    RETURN $error$
(7)   END IF
(8)   //projection (with dupl.  elimination)
(9)   //to attributes of output schemas
(10)  $left_{out}$ := $r\_group[S^{OUT}_{left}.A]$
(11)  $right_{out}$ := $r\_group[S^{OUT}_{right}.A]$
(12)  //if join values of previous group
(13)  //are equal to current group
(14)  IF($left_{pre}[\bowtie^{-1}.p.att()] = left_{out}[\bowtie^{-1}.p.att()]$)
(15)  //then right outputs must be the same
(16)   IF($right_{out}! = right_{prev}$)
(17)    RETURN $error$
(18)   END IF
(19)  END IF
(20)  left_operator.pushNext($left_{out}$)
(21)  right_operator.pushNext($right_{out}$)
(22)  $left_{pre}$ := $left_{out}$
(23)  $right_{pre}$ := $right_{out}$
(24) END FOR
```

Figure 15: Case 1: Method pushNext of $\bowtie^{-1}$

right outputs are propagated to the next operators and they are saved as previous outputs for the next loop execution (line 21 to 24).

Moreover, if one of the output schemas allows duplicates, the reverse join operator has to find out the correct cardinality of the outputs out of different possibilities. In that case duplicate elimination is needed in line 10 and line 11. However, this extension is straightforward and not shown in this report.

Example: An example for that case can be shown by the following query:

```
SELECT c.id, c.age, s.id, s.age
FROM Customer c, Supplier s
WHERE c.age=s.age
```

Both relations (Customer and Supplier) have a primary key attribute id. The input is given by the following two tuples: $\langle 1, 27, 1, 27\rangle$, $\langle 2, 27, 2, 27\rangle$. Both tuples are in separate groups because the attributes of the left output $c.id$ and $c.age$ have different values. As the reverse join produces different supplier tuples for the right output of both groups, although they have the same attribute value for the join attribute $c.age$, the input is incorrect. A correct input should have four tuples: $\langle 1, 27, 1, 27\rangle$, $\langle 1, 27, 2, 27\rangle$, $\langle 2, 27, 1, 27\rangle$, $\langle 2, 27, 2, 27\rangle$.

Case 2: If the join is not an equi-join and the join predicate is in the form of $a_i > a_j$ or in the form of $a_i \geq a_j$, then the blocking version of the reverse join operator is needed.

Example: Consider the following query and the given input:

```
SELECT c.cid, c.age, s.id, s.age
FROM Customer c, Supplier s
WHERE c.age>s.age
```

c.id	c.age	s.id	s.age	
1	27	1	25	/*1st group*/
1	27	2	26	
2	28	1	25	/*2nd group*/
2	28	3	27	

With a careful look on the input, it can be seen that the input is not a valid input of the reverse join since a tuple $\langle 2, 28, 2, 26\rangle$ is missing in the second group. As a result, the reverse join operator has to examine all the input before it produces the first result.

The implementation for that case is given in Figure 16. It is similar to the implementation of case 1 - the differences are marked bold. In particular, the reverse join also has to group the input by the attributes of the left output schema (e.g., $c.id, c.age$), and additionally has to sort the input by the join attribute (e.g., $c.age$) in ascending order (descending order is used if the comparison operator is $<$ or $\leq$) (line 1 and 2). This way, the set of output tuples which is produced for the right output (e.g. table $Supplier$) of the the first group must be contained completely in the set of output tuples which is produced for the second group (line 15). If this condition holds among all adjacent groups, then the input is valid; otherwise *error* should be returned (line 16).

Case 3: If the join is not an equi-join and the join predicate is in the form of $!(a_i = a_j)$, then a blocking version of the reverse join operator is needed. The blocking version is implemented similar to the first case: the input tuples are grouped by the left output schema, then the join operator checks if each group produce the same set of output tuples for right output. One difference is, that output tuples that violate the join predicate are excluded from the above checking.

Case 4: In order to process more complex join predicates, the algorithms introduced before must be combined. For example, to check the input of a reverse join operator with a conjunctive predicate like $a_i > a_j \wedge a_k < a_l$, the input must be grouped by a_i and a_k and the groups must be sorted ascending by a_i and descending by a_k. Moreover, to check the input of a reverse join operator with a disjunctive join predicate the tuples of the input must be divided into different input groups each fulfilling one predicate element. E.g. for a join predicate like

```
$\bowtie^{-1}$.pushNext(Relation $r$)
r_groups := groupby($r$, $S_{left}^{OUT}.A$)
r_groups := sortbyatt(r_groups, $S_{left}^{OUT}.A \cap \bowtie^{-1} .p.atts()$, $\bowtie^{-1} .p$)
//analyze each group in r_groups
FOREACH r_group in r_groups
 //check join predicate
 IF(r_group not fulfills $\bowtie^{-1}$.p)
  RETURN error
 END IF
 //projection (with dupl.  elimination)
 //to attributes of output schemas
 $left_{out}$ := r_group[$S_{left}^{OUT}.A$]
 $right_{out}$ := r_group[$S_{right}^{OUT}.A$]
 //right output of successor group
 //must be contained in previous group
  IF($right_{prev} - right_{out} != \emptyset$)
   RETURN error
  END IF
 END IF
 left_operator.pushNext($left_{out}$)
 right_operator.pushNext($right_{out}$)
 $left_{pre}$ := $left_{out}$
 $right_{pre}$ := $right_{out}$
END FOR
```

Figure 16: Case 2: Method pushNext of $\bowtie^{-1}$

$a_i > a_j \vee a_k < a_l$ we divide the input into two groups - one which fulfills the predicate $a_i > a_j$ and another which fulfills the predicate $a_k < a_l$. If a input tuple fulfills more than one predicate the tuple is added to all corresponding input groups. Afterwards, each input groups are checked separately by the algorithms introduced for the previous cases. As each join predicate can be transformed into disjunctive normal form, we are able to process arbitrary reverse join operators.

6.9.2 Reverse Projection and Reverse Aggregation

In two special cases, the top down phase of RQP needs the blocking implementation of the reverse projection operator and the reverse aggregation operator.

Case 1: If the output schema of a reverse projection (or reverse aggregation) operator contains a `CHECK` constraint in the form of $a_j < a_i < a_k$ or in the form of $a_j < a_i < c$ or in the form of $c < a_i < a_j$ (alternatively the predicate could use the $<=$ instead of the $<$ operator), where a_j and a_k are attributes in the input schema and a_i is an attribute in the output schema but not in the input schema and is bound by a unique or primary key constraint, the data instantiation phase should use the blocking implementations of the operators.

Example: An example for this special case is a query like `SELECT b FROM R WHERE b<a and a<10` where the relation R consists of attributes a and b; and a is a primary key. If there are two input tuples $\langle 7 \rangle$ and $\langle 8 \rangle$, then the reverse projection may generate $\langle 9, 7 \rangle$ for the first input tuple $\langle 7 \rangle$. If that is the case, the reverse projection could not find an instantiation for the second tuple $\langle 8 \rangle$ because $\langle 9, 8 \rangle$ is the only possible instantiation (as $b < a < 10$) but this instantiation violates the primary key constraints imposed by the first output tuple $\langle 9, 7 \rangle$ on the attribute a. As a result, a blocking implementation is needed such that the reverse projection and the reverse aggregation operator consider all input tuples and generates the output in one batch. For the example above, the reverse projection has to buffer all the input in order to produce the output $\langle 8, 7 \rangle$ and $\langle 9, 8 \rangle$.

Figure 17 shows a generalized version of the function *instantiateData* which is used by the blocking implementation of both operators. This version takes a complete relation as input and returns an instantiation of the output for the complete input, as well as an array of numbers which represent the number of tuples, which have to be used for each input tuple in order to dissolve aggregations ($n[i]$ is the number of output tuples which have to be created for the i-th input tuple). Therefore the function guesses the right number of output tuples for each input tuple by creating all possible combinations of count values for all input tuples (line 1 to 13). Afterwards the function tries to find an instantiation of the output for each possible combination of count values (line 14 to 20). In case that the function finds an instantiation, it returns this instantiation and the current combination of count values. If none of the combinations is satisfiable $(NULL, NULL)$ is returned.

This function is more expensive than the simple $instantiateData$ function, because of several reasons: One is that the constraint formula is more complex for the complete input and thus the model checker needs more time; another reason is that the trial-and-error has to be

```
instantiateData(Relation $r$, Schema $S^{OUT}$)
Output:
 //data instantiation
 -instantiation $I$
 //number of tuples to ungroup each tuple
 -int[] $n$
//number of tuples to ungroup $r$
int[] $count$, $maxcount$
$i := 1$
//analyze each tuple $t \in r$
FOREACH $t$ in $r$
  IF $t$ includes COUNT of aggregation
    $count[i] = maxcount[i]$:=COUNT value in $t$
  ELSE //USER_THREHOLD=1 if no aggregation
    $count[i]$:=1; $maxcount[i]$:=USER_THREHOLD
  $i = i + 1$
END FOR
//create combinations of count domains
$comb$:=createCombinations($count$,$maxcount$)
FOREACH $n$ in $comb$ //$n$ is a $k-$array; $k$ is the cardinality of $r$
    //Create constraint formula $L$
    $L$:=createConstraint($r$,$S^{OUT}$,$n$)
    $I$:=decisionProcedure($L$)
    IF($I$!=NULL) RETURN ($I$,$n$)
END FOR //Trial-and-error
RETURN (NULL,NULL)
```

Figure 17: Case 1: Function instantiateData

carried out for the complete input and thus the size of combinations grows exponential with the number of input tuples.

Case 2: If a reverse projection (or reverse aggregation) operator generates tuples which are processed by a reverse join operator (implied by a join dependency in the output schema) and its join predicate does not express the equality on a primary key attribute of one of the output schemas, then blocking versions of the operators are needed during the data instantiation phase. Otherwise these operators may generate incorrect values which do not satisfy the join properties. The implementation of the blocking versions for these two operators in this special case needs similar algorithms as the blocking version of the reverse join operators in Section 6.9.1 in order to check the input. The algorithms can be adapted easily from that section and are not shown here.

Additional algorithms are needed in order to produce the output. First, the input must be grouped as described for the different join predicates in Section 6.9.1. Afterwards, the output generation is carried out for each group of the input separatelly in order to generate values which respect the join properties.

In the following we explain the output generation for join predicates which equal to those of case 2 in Section 6.9.1. For illustration purposes we use the following example.

Example: Consider the following query and the given input. The query is similar to the example query of case 2 in Section 6.9.1. However the join attribute $s.age$ is not given by the input:

```
SELECT c.cid, c.age, s.id
FROM Customer c, Supplier s
WHERE c.age>s.age
```

c.id	c.age	s.id
1	27	1 /*1st group*/
1	27	2
2	28	1 /*2nd group*/
2	28	2
2	28	3

First, the operator analyzes which join attributes are given by the input. If at least one join attribute is given by the input (e.g. $c.age$), the input is grouped by the attributes of that output schema of the corresponding reverse join operator which contains that join attribute (e.g. $c.id, c.age$). Afterwards, the input groups are sorted by that join attribute (e.g. $c.age$) ascending or descending depending on the relational operator of the join predicate ($>$, $>=$ or $<$, $<=$). If both join attributes are not given by the input, then the input is grouped by the attributes of the left output schema of the correspoding reverse join and sorted ascending by the cardinality of each group. If the value for the join attribute of the output schema we grouped by is not given by the input (e.g. $c.age$), then the operator has to generate one distinct value per group where the values for all groups are sorted ascending or descending depending on the join predicate (e.g. 27, 28). However, in our example the attribute $c.age$ is given by the input and thus no values have to be generated. Other values which must be generated for that output schema must be distinct for each group, too. If the value for the join attribute of the other output schema is not given by the input (e.g. $s.age$), then the attribute values generated for the first group must be reused by the second group (e.g. we generate 25, 26 for the tuples with $s.id = 1, 2$). Values generated for other attributes of that output schema (not in the join predicate) must be reused, too. New values must be generated for those tuples which are in the second but not in the first group (e.g. we generate 27 for the tuple with $s.id = 3$). The new values for the join attribute have to be greater than the maximum value of the join attribute (e.g. $s.age$) used in the first group in case that the join operator is $>$ or $>=$ or smaller then than miminum value of the join attribute in case that the join operator is $<$ or $<=$. Moreover, all generated join attribute values have to fulfill the join predicate. These steps have to be carried out for all adjacent groups.

The algorithms for other join predicates are straightforward. As the previous algorithm, these algorithms generate values for the join attributes in a similar way such that these values fulfill the properties of the particular join predicate shown in the different cases of Section 6.9.1.

6.9.3 Reverse Union

If both output schemas of a reverse union operator have a primary key or a unique constraint on the same attribute a_i and there is a check constraint on another attribute a_j in the output schema, then a blocking version of the reverse union is needed in the top down data instantiation phase.

Example: An example can be shown by the query in Figure 3 (left side). Assume attribute b is the primary key attribute of both relation R and S and the two input tuples are $\langle 6, 6 \rangle$ and $\langle 2, 6 \rangle$. Using the non-blocking version of the reverse union operator, the first tuple $\langle 6, 6 \rangle$ might be distributed to the relation R. Then, the second tuple $\langle 2, 6 \rangle$ cannot be distributed to relation S because $a = 2$ cannot not fulfill the selection predicate $a > 5$. Alike, this tuple also could not be distributed to relation R because of the primary key constraint. Therefore, a blocking implementation of the reverse union operator is needed which buffers all the input and distributes $\langle 6, 6 \rangle$ to S and $\langle 2, 6 \rangle$ to R.

Figure 18 shows the implementation of the blocking version of the reverse union operator. First, the method analyzes which tuple must be distributed to the left, right, and which tuple can be distributed to both outputs in a similar way as the non-blocking reverse union implementation (line 1 to 20). Afterwards those tuples which can by distributed to both outputs ($both_{out}$) must be divided into two relations, one for each output (by method call $distribute$) (line 22). The method $distribute$ (not shown as algorithm) analyzes possible combinations to distribute tuples in $both_{out}$ to $left_{out}$ and $right_{out}$. In order to check if a combination satisfies the output schemas, two constraint formulas have to be constructed (one for $left_{out}$ and one for $right_{out}$). These formulas have to be checked by the model checker if they are satisfiable. If not, the next combination is tried. If no combination is found, the $distribute$ method returns an error (line 24), else the output is propagated to the left and right branch (line 26, 27) as specified in the combination.

6.10 Processing Nested Queries

As mentioned in Section 5, SPQR uses the concept of nested iterations (sometimes called apply operators) which are known from traditional query processing [8], 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.

Example 1: Assume that the `Lineitem` table (from Figure 3.3a)) has an extra column `shipdate`. Then, the following nested query is processed reversely as follows:

```
SELECT oid FROM orders
WHERE orderdate IN
 (SELECT shipdate FROM lineitem
  WHERE l_oid = oid)
```

First, the reverse query plan of the outer query is executed given an RTable. The values

```
$\cup^{-1}$.pushNext(Relation $r$)
$left_{out}$ := $\emptyset$
$right_{out}$ := $\emptyset$
$both_{out}$ := $\emptyset$
FOREACH $t$ in $r$
  //Create constraint formulas
  $L_{left}$:=createConstraint($t$,$S_{left}^{OUT}$)
  $L_{right}$:=createConstraint($t$,$S_{right}^{OUT}$)
  //call model checker
  IF(decisionProcedure($L_{left} \wedge L_{right}$)!=NULL)
    $both_{out}$.add($t$)
  //call model checker
  ELSE IF(decisionProcedure($L_{left}$)!=NULL)
    $left_{out}$.add($t$)
  //call model checker
  ELSE IF(decisionProcedure($L_{right}$)!=NULL)
    $right_{out}$.add($t$)
  ELSE
    return $error$
  END IF
END FOR
($left_{out}$, $right_{out}$) :=
  distribute($both_{out}$, $left_{out}$, $right_{out}$)
IF($left_{out}$, $right_{out}$=(NULL,NULL))
  return $error$
END IF
left_operator.pushNext($left_{out}$)
right_operator.pushNext($right_{out}$)
```

Figure 18: Method `pushNext` of $\cup^{-1}$ in special case

generated for the bind variable $orderdate$ and the correlation variable oid are used to initialize the input for the reverse query tree of the inner subquery. Processing nested queries is, thus, expensive: it has quadratic complexity with the size of the RTable. Section 7 shows how almost all nested queries can be unnested for reverse query processing in order to improve performance.

In those cases where the bind or the correlation predicate does not express the equality, the reverse apply operator has to be implemented as blocking operator. This is obvious, as each nested RRA expression can be unnested, e.g. by using reverse join operators (as shown in Section 7). In the case that the reverse join operator uses a inequality predicate, it also must use blocking implementation. Thus, the algorithms for the blocking reverse apply operators are similar to the reverse join and not shown in this technical report. The only difference is, that the reverse apply generates new input values for the inner subquery.

Example 2: Assume a similar query is given as before. The only difference is, that the join predicate is $l\_oid > oid$. In that case the reverse apply operators must generate values for the attribute l_oid which satisfy that predicate.

6.11 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 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`. This attribute 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 in the definition of constrictive independent attribute 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 for a constrictive independent attribute a which is only bound by `CHECK` constraints. An example for 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 for constrictive independent attributes a which are only bound by an aggregation constraint. 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 different techniques to instantiate values for a can be used. Some possibilities are shown below:

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 constraint 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 `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 12) 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 [21].

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: As mentioned in Section 3, 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 applicable in the RQP context.

7 Reverse Query optimization

The job of the reverse query optimizer is to transform a reverse query tree into a more *efficient* reverse query tree (Figure 1). As part of such a rewrite, the input and output schemas need to be adjusted (Section 5). Depending on the application, different optimization goals can be of interest. The RQP framework allows the integration of different query optimizers for different goals. In this work, the RQP optimizer tries to minimize the running time of reverse query processing. Designing optimizers with other optimization goals (e.g., minimizing the size of the generated database instances) are beyond the scope of this paper.

Just as in traditional query optimization, the reverse query optimizer rewrites a reverse query tree into an *equivalent* reverse query tree that is expected to have lower cost (or running time). There are several possible definitions of equivalence:

- *General RQP-equivalence:* Reverse Query Trees T_1 and T_2 are generally RQP equivalent for Query Q iff for all RTables R: $Q(T_1(R)) = Q(T_2(R)) = R$.

- *Result-equivalence:* Reverse Query Trees T_1 and T_2 are result-equivalent iff for all RTables R: $T_1(R) = T_2(R)$.

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 and allows more rewrites. It is okay if the rewritten reverse query tree generates a different database instance (in fact, it might even be desired); the only thing that matters is that the overall RQP correctness criterion (Section 3.1) is met. That is why general RQP-equivalence is used in the optimizer of the SPQR prototype.

7.1 Optimizer Design

The most expensive operators of RQP are π^{-1} and χ^{-1} because these operators call the decision procedure of the model checker. The exact cost of these operators is difficult to estimate for a specific query because there are no robust cost models for model checkers; defining such cost models is a research topic in its own right in that community. Nevertheless, it is clear that the simpler and shorter the constraints, the better. One consequence is that it is important to minimize the number of π^{-1} and χ^{-1} operators in a reverse query tree. Therefore, the canonical translation of a SQL query into an expression of the relational algebra [10] is already good because it results in at most one π^{-1} operator at the root of the reverse query tree. Optimizations that add projections and group-bys as devised for traditional query processing need not be applied.

In addition to π^{-1} and χ^{-1}, the execution of nested queries is expensive because it is $\mathcal{O}(n^2)$, with n the size of the input (i.e., RTable or intermediate result). Therefore, it is important to unnest queries. Rules that make it possible to fully unnest almost all queries are given in the next subsection. Furthermore, $\cup^{-1}$ and $-^{-1}$ operators can be expensive because they potentially involve calls to the model checker. As for π^{-1} and χ^{-1} operators, therefore, the goal is to minimize the number of $\cup^{-1}$ and $-^{-1}$ operators in a reverse query plan. Again, the canonical translation of SQL queries is good enough in practice for this purpose.

All other operators are cheap. They are linear in the size of their inputs and do not require any calls to the model checker. In particular, the reverse equi-join that involves a primary key or an attribute with a unique constraint is cheap. As a result, it is not important to carry out cost-based join ordering or worry about different reverse join methods. Again, the canonical relational algebra expression can be used for simple rewrites that eliminate unnecessary operators (e.g., σ^{-1}'s in certain cases) and/or simplifies the expressions in the reverse query tree. Such rewrites are presented in the last subsection of this section.

7.2 Query Unnesting

There are three rewrite rules that can be used to fully unnest most SQL queries. Only some queries that involve the same table in the outer and in the inner query cannot be unnested for RQP. This very aggressive unnesting is possible because of the relaxed equivalence criterion presented at the beginning of this section. However, the optimizer has to keep all functional dependencies for the rewritten queries, which are added by the bottom-up phase for the unnested query.

RR 1: A subquery Q_in1 nested inside a `NOT IN` operator can be removed if (1) the inner and outer queries refer to different tables and (2) no other subquery Q_in2 exists which refers to the same table as Q_in1 and is not nested inside `NOT IN`. As a result, Query Q_1 can be rewritten to Q_2 in the following example:

```
Q1  SELECT name FROM Lineitem WHERE l_oid NOT IN
     (SELECT MAX(oid) FROM Orders
     GROUP BY orderdate);
Q2  SELECT name FROM Lineitem;
```

To check the correctness, consider an RTable R with only one tuple: $\langle$productA$\rangle$. Q_1 and Q_2 are obviously not result-equivalent with respect to R. RQP for Q_1 would generate at least one Lineitem tuple and one Orders tuple; in contrast, RQP for Q_2 would only generate a Lineitem tuple. The queries are general RQP-equivalent, however, because applying Q_1 to both database instance would return the required result; i.e. a single row with value "productA".

RR 2: An inner query in a nested query can be removed if (1) the columns used in the SELECT clause of the inner query are also used in the SELECT clause of the outer query and (2) the two queries are correlated by an equality predicate or by an IN predicate. For example, the following Query Q_3 can be rewritten to Query Q_4:

```
Q3  SELECT name, price FROM Lineitem
     WHERE price=(SELECT MIN(price) FROM lineitem)
Q4  SELECT name, price FROM Lineitem
```

RR 3: If RR 1 and RR 2 are not applicable, all methods proposed in [9] to unnest queries for traditional query processing can be applied to reverse query processing, too. The proof is straightforward because result-equivalence for traditional query processing implies general RQP-equivalence for reverse query processing. However, the optimizer has to take care of the operator order mentioned in [17] in order to preserve the general RQP-equivalence.

7.3 Other Rewrites

At the begin of this section, we would like to mention the following (somewhat surprising) rewrite rule:

RR 4: Remove reverse select operators from the reverse query plan. Section 6 showed that this operator can be implemented using the identity function at run-time. Only a reverse select at the root of the plan must not be removed in order to make sure that its predicate is checked.

There can be found several rewrite rules that help to simplify expressions (e.g., eliminate LIKE and other SQL functions from predicates). One such rewrite rule is:

RR 5: A LIKE predicate can be rewritten as a equality predicate without the wildcards (e.g. %) if (1) the attributes included in the LIKE predicate are not given by the input and (2) these attributes do not have a UNIQUE constraint. It is obvious, that the instantiated values for the rewritten equality predicate also fulfills the LIKE predicate, e.g. all values which fulfill (name='A') also fulfill (name LIKE '%A%')).

8 Performance Experiments and Results

This section presents the results of performance experiments with our prototype system SPQR and the TPC-H benchmark [1]. These experiments show the running times of reverse query processing and the size of the generated databases.

8.1 Experimental Environment

The SPQR system was implemented in Java (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 6 was disabled. As a backend database system, PostgreSQL 7.4.8 was used and installed on the same machine. As a decision procedure, Cogent [7] was used. Cogent is a decision procedure that is publicly available and has been used in several projects world-wide. Cogent was written using the C programming language. 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. Some queries (e.g., Q11) are parametrized and their results and running times depend on random settings of the parameters. 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 (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 (as compared to the size of the original TPC-H database instance) and the running time of reverse query processing.

8.2 Size of Generated Databases

Table 2 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[3] 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 handful of tuples is 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 from 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.

8.3 Running Time (SF=0.1)

Table 3 shows the running times of RQP for the TPC-H benchmark with scaling factor 0.1. In the worst case, the running time is up to one hour (Query 10). However, most queries can be reverse processed in a few seconds. Table 3 also shows the cost break-down of reverse query processing. `QP` is the time spent processing tuples in SPQR (e.g., constructing constraint formulae and calls to the *pushNext* function). For all queries (except Q1), this time is below

[3] Implicit means that the COUNT value can be calculated by the optimization rule *OP 5* in Section 6.11.

	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 2: Size of Generated Databases and RTable (rows)

Query	RQP	QP	DB	MC	M-Invoke
1	26:51	12:01	8:42	6:06	4
2	0:24	< 1ms	0:21	0:02	44
3	19:20	0:14	0:11	18:55	1216
4	0:20	0:05	0:14	< 1ms	5
5	0:12	< 1ms	< 1ms	0:11	10
6	0:02	< 1ms	< 1ms	0:1	2
7	0:10	< 1ms	0:01	0:9	8
8	0:15	< 1ms	0:02	0:13	12
9	4:23	0:02	0:03	4:17	175
10	56:33	0:42	0:37	55:13	3767
11	42:11	0:13	0:14	41:43	2541
12	7:25	0:16	0:11	6:57	3155
13	2:56	1:38	1:16	< 1ms	21
14	0:08	< 1ms	0:01	0:07	6
15	0:03	< 1ms	< 1ms	0:03	3
16	0:29	0:15	0:14	< 1ms	0
17	0:02	< 1ms	< 1ms	0:01	2
18	0:01	< 1ms	< 1ms	< 1ms	15
19	0:02	< 1ms	< 1ms	0:01	2
20	0:21	< 1ms	< 1ms	0:20	42
21	1:43	0:04	0:05	1:34	465
22	0:26	0:01	0:01	0:23	641

Table 3: Running Time (min:sec): SF=0.1

Query	100M	1G	10G
1	26:51	207:11	2054:19
2	0:24	0:47	4:02
3	19:20	183:49	1819:48
4	0:20	2:26	24:15
5	0:12	0:12	0:12
6	0:02	0:01	0:01
7	0:10	0:10	0:09
8	0:15	0:17	0:14
9	4:23	4:33	10:20
10	56:33	566:45	5639:13
11	42:11	18:15	4472:00
12	7:25	83:09	719:56
13	2:56	27:47	276:05
14	0:08	0:08	0:15
15	0:03	0:03	0:04
16	0:29	4:04	36:37
17	0:02	0:02	0:08
18	0:01	0:10	1:54
19	0:02	0:02	0:02
20	0:21	3:24	32:27
21	1:43	14:44	140:47
22	0:26	4:08	42:00

Table 4: Running (min:sec): Vary SF

a minute. Q1 is an exception because it generates many tuples and a great deal of work is necessary in order to carry out the optimizations of Section 6.11 for each tuple. `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 RQP. 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 6 and 7). `M-Invoke` shows the number of times the decision procedure is called. Comparing the `MC` and `M-Invoke` 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). We still have not found a way to predict the cost per call and we are hoping for progress in this matter from the model checking research community.

We also measured the number of attempts each TPC-H query needed for guessing the number of tuples in aggregations (Section 6). These results are not shown in Table 3, but the results are encouraging: in fact, none of the 22 required any trial-and-error. The reason is that the optimizations proposed in Section 6.11 effectively made it possible to pre-compute the right number of tuples for all TPC-H queries.

8.4 Running Time: Varying SF

Table 4 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 (as the size of the generated databases, Table 2) is independent of the scaling factor; example queries are Q5 and Q6. For all those queries, for which the running times were higher 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 RQP potentially scales linearly and that even large test databases can be generated using RQP. 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.

9 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 has a similar goal: find instantiations of logical expressions. Consequently, we use the results of that research community in our design. However, the model checking community has not addressed issues involving SQL or database applications. In addition, that community has not addressed any scalability issues that arise if millions of tuples need to be generated as for the TPC-H benchmark. In order to provide scalability, our design adopted techniques from traditional query processing; e.g., [13, 11].

As mentioned in Section 2, there are several applications for reverse query processing and there is a great deal of related work for all these application areas. For example, there has been significant related work in the area of generating test databases. [19] shows how functional dependencies can be processed for generating test databases. The bottom-up phase of RQP (Section 5) makes use of the findings of the work in [17] and extends it for the complete SQL specification. Likewise, other work on the generation of test databases (e.g., [20, 4]) focuses on one aspect only and falls short on most other aspects of RQP. [16] 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, 3]. That work is orthogonal to our work. In the other potential application areas of RQP (e.g., sampling), to the best of our knowledge, nobody has tried yet to apply techniques such as RQP.

10 Conclusion and Future Work

This work presented a new technique called reverse query processing or RQP, for short. Reverse query processing 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). It could be shown that a full-fledged RQP engine for SQL can be built and that it scales linearly with the size of the database that needs to be generated for the TPC-H benchmark.

We believe that this work is only the first step into a new research direction. The most important avenue for future work is to further explore the applications of RQP. Section 2 lists several ideas for potential applications, but significant additional research is needed in order to exploit the potential of RQP for these applications. Furthermore, additional work is required in order to develop RQP techniques that guarantee certain properties of the generated data (e.g., minimality). In addition, it is going to be important to leverage recent developments of the model checking community. Finally, another avenue of future work is to generate a single consistent database for multiple queries. We are now solving this problem by using a late instantiation approach: each query is reverse processed individually without any data instantiation. Then the constraints imposed by different queries on each relation are merged and data instantiation is done on the database level.

References

[1] TPC benchmark H. http://www.tpc.org/tpch.

[2] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, pages 1097–1107, 2005.

[4] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, verification and reliability*, 2004.

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

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

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

[8] C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 571–581, New York, NY, USA, 2001. ACM Press.

[9] R. A. Ganski and H. K. T. Wong. Optimization of nested sql queries revisited. In *SIGMOD*, pages 23–33, 1987.

[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. Hoare and J. Misra. Verified software: theories, tools, experiments. Vision of a grand challenge project. In *Verified Software: Theories, Tools, Experiments*, 2005.

[16] T. Imielinski and J. Witold Lipski. Inverting relational expressions: a uniform and natural technique for various database problems. In *PODS '83: Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 305–311, New York, NY, USA, 1983. ACM Press.

[17] A. Klug. Calculating constraints on relational expression. *ACM Trans. Database Syst.*, 5(3):260–290, 1980.

[18] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, 2002.

[19] H. Mannila and K.-J. Räihä. Test data for relational queries. In *PODS*, pages 217–223, 1986.

[20] 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.

[21] 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.