ConQuer: Efficient Management of Inconsistent Databases

Ariel Fuxman, Elham Fazli, Renée J. Miller[*]
{afuxman,elham,miller}@cs.toronto.edu
University of Toronto

ABSTRACT

Although integrity constraints have long been used to maintain data consistency, there are situations in which they may not be enforced or satisfied. In this paper, we present ConQuer, a system for efficient and scalable answering of SQL queries on databases that may violate a set of constraints. ConQuer permits users to postulate a set of key constraints together with their queries. The system rewrites the queries to retrieve all (and only) data that is consistent with respect to the constraints. The rewriting is into SQL, so the rewritten queries can be efficiently optimized and executed by commercial database systems.

We study the overhead of resolving inconsistencies dynamically (at query time). In particular, we present a set of performance experiments that compare the efficiency of the rewriting strategies used by ConQuer. The experiments use queries taken from the TPC-H workload. We show that the overhead is not onerous, and the consistent query answers can often be computed within twice the time required to obtain the answers to the original (non-rewritten) query.

1. INTRODUCTION

Integrity constraints have long been used to maintain data consistency. Data design focuses on developing a set of constraints to ensure that every possible database reflects a valid, consistent state of the world. However, integrity constraints may not be enforced or satisfied for a number of reasons. In some environments, checking the consistency of constraints may be too expensive, particularly for workloads with high update rates. Hence, the database may become inconsistent with respect to the (unenforced) integrity constraints. When data is integrated from multiple sources, each source may satisfy a constraint (for example a key constraint), but the merged data may not (if the same key value exists in multiple sources). More generally, when data is exchanged between independently designed sources with different constraints, the exchanged data may not satisfy the constraints of the destination schema.

[*]Supported in part by NSERC and CITO awards.

One strategy for managing inconsistent databases is data cleaning [9]. Data cleaning techniques seek to identify and correct errors in the data and can be used to restore the database to a consistent state. Data cleaning, when applicable, may be very successful. However, these techniques are semi-automatic at best, and they can infeasible or unaffordable for some applications. Furthermore, committing to a single cleaning strategy may not be appropriate for some environments. A user may wish to experiment with different cleaning strategies, or may desire to retain all data, even inconsistent data, for tasks such as lineage tracing. Finally, data cleaning is only applicable to data that contains errors. However, the violation of a constraint may also indicate that the data contains exceptions, that is, clean data that simply does not satisfy a constraint.

In this work, we take an approach that is applicable to databases with both errors and exceptions. We propose a system, ConQuer, for managing inconsistent data.[1] In ConQuer, a user may postulate a set of integrity constraints, possibly at query time, and the system retrieves all (and only) the query answers that are consistent with respect to the constraints. In order to do this, ConQuer rewrites the query into another SQL query that retrieves the consistent answers. We illustrate the semantics of consistent query answering and ConQuer's rewriting strategy with an example, and provide precise definitions in Section 2.[2]

Example 1. *Consider the database of Figure 1, which contains information about customers and their account balances. Assume that a user specifies that the key of the* `customer` *relation should be* `custkey`*. Note that the database violates this key constraint, perhaps because its data has been integrated from many operational sources. Consider a query that retrieves information about customers whose account balance is over 1000.*

q_1:
```
select custkey
from customer
where acctbal > 1000
```

If we execute this query over the instance of Figure 1, we obtain $\{c1, c2, c3, c3\}$*. This cannot be considered a "consistent" answer for the following reasons. First, it may be the*

[1]ConQuer stands for *Con*sistent *Quer*ying.

[2]In the examples throughout the paper, we use queries motivated by those in the TPC-H workload, but simplified to illustrate specific points. However, in the experiments of Section 6, we use actual TPC-H queries.

	custkey	acctbal
t1	c1	2000
t2	c1	100
t3	c2	2500
t4	c3	2200
t5	c3	2500

Figure 1: Inconsistent instance of the `customer` relation

case that customer $c1$ has an account balance below 1000 (tuple $t2$). However, $c1$ is included in the answer because it appears in another tuple ($t1$) which does satisfy the query. Second, customer $c3$ appears in the answer twice. Since we consider `custkey` *to be the key of the relation, we do not expect to have repeated values in the answer.*

In this case, we would expect $\{c2, c3\}$ to be the "consistent" answer for query q_1. The reason is that $c2$ appears in a single tuple of the database, which satisfies the query and does not violate the key constraint. Even though $c3$ appears in two tuples (which therefore violate the key constraint), both tuples satisfy the query.

ConQuer rewrites queries in order to obtain their consistent answers. For q_1, it would produce the following query rewriting:

```
select distinct custkey
from customer c
where acctbal > 1000
   and not exists (select *
                   from customer c'
                   where c'.custkey=c.custkey
                         and c'.acctbal ≤ 1000)
```

The rewritten query has two differences with respect to q_1. First, it uses the `distinct` *keyword to ensure that each consistent answer is returned the right number of times. In this case, this ensures that $c3$ appears exactly once in the answer. Second, it has a nested subquery related by* `not exists`*. The purpose of this subquery is to filter out those key values that satisfy q_1 in some tuples, but violate it in others. In our example, this subquery filters $c1$ from the answer because it appears in tuple $t2$ with an account balance below 1000.*

We will use a common definition of consistent query answer based on the notion of *repair* [2]. For keys, a repair is a subset of the inconsistent database containing exactly one tuple per key value. A repair is one possible "cleaned" version of the database. A query answer is then consistent if it is an answer to the query in every repair. In contrast, for monotone queries (queries without negation such as the query of Example 1), the original query executed on an inconsistent database returns a set of possible answers. A possible answer is an answer to the query in at least one repair. In the next section, we will give a precise definition of consistent answers for arbitrary queries and constraints (not just keys).

In the absence of user input to decide between tuples that violate a constraint, ConQuer provides the option of retaining all tuples, rather than forcing the user to remove (or ignore) the inconsistent data. Furthemore, our approach is orthogonal to any cleaning that may be done on the database. In particular, ConQuer can be used interactively by users to understand where the data is potentially inconsistent. For example, if we take the difference between the result of the original and rewritten queries of Example 1, we can detect that customer $c1$ satisfies the query but is not a consistent answer. This may indicate that its data should be cleaned. As another example, a user may not be too concerned if there are multiple tuples for a customer that differ on address (something that can be corrected with a specialized data cleaning tool for addresses like Trillium[3]), provided that other important information, such as the market segment, is consistent.

Summary of Results. The main contributions of our work are the following:

- We present algorithms for rewriting SQL queries into SQL queries that return only consistent answers. Our approach is fully declarative, requiring no procedural pre- or post-processing. This enables us to apply our techniques to much larger databases than any of the existing consistent query answering systems.
- We consider not only Select-Project-Join (SPJ) queries, but also SQL's bag semantics and queries with aggregation. Our rewritings for these features are novel contributions, and are needed to enable practical use in analysis queries over an integrated and potentially inconsistent data warehouse.
- We present a rewriting that works over the unchanged (inconsistent) database and a second rewriting that makes use of annotations attached to tuples indicating whether the tuple is known to be consistent. Such annotations are appropriate for high performance environments where the inconsistent data represents valuable data that cannot be removed. We show how such annotations can be exploited using query optimizations specific to the semantics of consistent query answering. These optimizations are highly effective and would not be found by a standard optimizer.
- We present a performance study using the data and queries of the TPC-H decision support benchmark. Our study highlights the overhead of dynamically (at query time) resolving inconsistencies when off-line cleaning is not possible (or not appropriate for the application). We consider different degrees of inconsistency (we experiment with databases in which up to 50% of the database may be inconsistent) and database sizes to understand the applicability of our approach.

In the next section, we precisely define consistent query answering. Our query rewriting strategy for queries without aggregation is presented in Section 3. In Section 4, we consider queries with aggregation. In Section 5, we present an optimization of the rewritings that exploits precomputed information about constraint violations. In Section 6, we present a performance study which uses queries from the TPC-H decision support benchmark. In Section 7, we consider related work in the area of consistent query answering. We close the discussion in Section 8 with some conclusions and directions for future work.

[3]`www.trilliumsoftware.com`

2. PRELIMINARIES

Throughout this paper, we take an unorthodox view of constraints. The constraints that we consider do not restrict the valid database instances. Rather, they constrain the set of valid (in our terminology consistent) answers that can be obtained when posing a query on the database. To avoid confusion, we will refer to such constraints as *query constraints*. In this paper, we will consider sets of query constraints which consist of at most one key constraint per relation of the query.

Let $\mathbf{R}$ be a relational schema, and D be a database over $\mathbf{R}$. We are given a set of query constraints Σ, where the database may be *inconsistent* with respect to Σ. We use a common definition for *consistent query answer* that is based on the concept of a database *repair*, defined by Arenas et al. [2]. A repair $D^{\mathcal{R}}$ of D is an instance of $\mathbf{R}$ such that $D^{\mathcal{R}}$ satisfies the query constraints of Σ, and $D^{\mathcal{R}}$ differs minimally from D. In this case, minimality is defined with respect to the symmetric difference between D and $D^{\mathcal{R}}$, denoted as $\Delta(D, D^{\mathcal{R}})$.

Dfn 1 (Repair [2]). *Let D be a database. A database $D^{\mathcal{R}}$ is a* repair *of D with respect to Σ if $D^{\mathcal{R}}$ satisfies Σ and there is no database D' satisfying Σ where $\Delta(D, D') \subset \Delta(D, D^{\mathcal{R}})$.*

Notice that repairs need not be unique. In fact, even for keys there may be an exponential number of them. Intuitively, each repair corresponds to one possible way of "cleaning" the inconsistent database. For keys, a repair will be a subset of D that contains exactly one tuple for each key value in D.

Example 2. *In our example of Figure 1, we have the following four repairs: $D_1^{\mathcal{R}} = \{t1, t3, t4\}$, $D_2^{\mathcal{R}} = \{t1, t3, t5\}$, $D_3^{\mathcal{R}} = \{t2, t3, t4\}, D_4^{\mathcal{R}} = \{t2, t3, t5\}$. Each repair is a consistent database that is as close as possible to the inconsistent database of Figure 1.*

The notion of repair is used to give a precise meaning to query answering over inconsistent databases [2]. In particular, a *consistent query answer* is required to appear in the result of a query on every repair of the inconsistent database.

Dfn 2 (Consistent Query Answer). *Let D be a database, q be a query, and Σ be a set of query constraints. The consistent query answers for q on D are defined as the set*

$$\bigcap_{D^{\mathcal{R}} \text{ is a repair of } D \text{ wrt } \Sigma} q(D^{\mathcal{R}})$$

In contrast to consistent answers, we could also consider *possible answers*, where a possible answer is an answer on some repair (that is, it is the *union* of the answers to q over all the repairs). Note that when Σ includes only key constraints and q is a monotone query, the original query q on D returns the set of possible answers.

For set semantics, we are only concerned with finding the set of tuples that occur in the query result for every repair. Under bag semantics, the multiplicity of a tuple in the consistent query answer is the minimum multiplicity from any repair.

Our approach to compute consistent answers is based on query rewriting. Specifically, given an SQL query q and a set of key query constraints Σ, we will rewrite q into another SQL query Q^c that retrieves the consistent answers for q. The rewriting is done independently of the data, and works for every inconsistent database. Some rewriting strategies for limited classes of queries have been proposed in the literature [2, 7]. The strategy presented in this paper is motivated by our previous work on rewriting conjunctive queries into first-order logic queries [11]. In Sections 3 and 4, we consider not only conjunctive queries (that is, SPJ queries with set semantics), but also queries with aggregation and bag semantics.

In this paper, we consider sets Σ of query constraints that consist of at most one key constraint per relation of the query. A key query constraint declares a set of attributes X as *the key* of a relation. Each attribute in X is a *key attribute*, all other attributes of the relation are *non-key attributes.*

To define the class of queries that can be handled by ConQuer, we first introduce the notion of a join graph [11].

Dfn 3 (join graph). *Let q be a SQL query. The* join graph *G of q is a directed graph such that:*

- *the vertices of G are the relations used in q; and*
- *there is an arc from R_i to R_j if a non-key attribute of R_i is equated with a key attribute of R_j.*

We present query rewritings for *tree queries.* Such queries can have two kinds of joins. First, they can have joins between key attributes. Second, they can have joins from non-key attributes of a relation (possibly a foreign key) to the primary key of another relation. Arguably, these two types of joins are the most commonly used in practice (and certainly the most common in standards like TPC-H).

Dfn 4 (tree query). *We say that a select-project-join-group-by query q is a* tree query *if (1) every join condition of q involves the key of at least one relation; (2) the join graph of q is a tree. We consider only queries containing equi-joins (no inequality joins). The selection conditions may contain comparisons (e.g., $<$) or functions. Each relation may be used at most once in the query. The query may contain aggregate expressions.*

Note that the previous definition restricts the non-key to key joins of the query to be acyclic, and does not permit non-key to non-key joins (since every join must involve the key of a relation).

3. JOIN QUERIES

In this section, we present ConQuer's rewriting strategy for tree queries without aggregation or grouping. In Section 4, we will consider a larger class of queries that includes aggregation. We illustrate the rewriting approach with the next example.

Example 3. *Consider a query q_2, which retrieves the orders placed by customers with an account balance over 1000.*

q_2:
```
select o.orderkey
from customer c, order o
where c.acctbal>1000 and o.custfk=c.custkey
```

order

	order key	clerk	cust fk
s1	o1	ali	c1
s2	o2	jo	c2
s3	o2	ali	c3
s4	o3	ali	c4
s5	o3	pat	c2
s6	o4	ali	c2
s7	o4	ali	c3
s8	o5	ali	c2

customer

	cust key	acct bal
t1	c1	2000
t2	c1	100
t3	c2	2500
t4	c3	2200
t5	c3	2500

Figure 2: An inconsistent database with `order` and `customer` relations

```
with Candidates as (
   select distinct o.orderkey
   from customer c, order o
   where c.acctbal>1000 and o.custfk=c.custkey),
Filter as (
   select o.orderkey
   from Candidates Cand
        join order o on Cand.orderkey=o.orderkey
        left outer join customer c
                    on o.custfk=c.custkey
   where c.custkey is null or c.acctbal≤1000)
select orderkey
from Candidates Cand
where not exists (select * from Filter F
              where Cand.orderkey = F.orderkey)
```

Figure 3: The rewriting Q_2^c for query q_2

The query constraints are that `orderkey` *should be the key of relation* `order`*, and* `custkey` *should be the key of relation* `customer`*. We will consider the database of Figure 2, where both relations are inconsistent with respect to the query constraints.*

The consistent query answers for q_2 on the database are $\{o2, o4, o5\}$. The reason that order $o1$ is not a consistent answer is that $c1$, the customer that placed order $o1$, is not known (for certain) to have an account balance over 1000. The order $o3$ is not a consistent answer because it might have been placed either by customer $c2$ or $c4$, the latter of which is not a tuple in the `customer` *relation.*

In Figure 3, we show a query rewriting Q_2^c that computes the consistent answers for q_2. Notice that there are two subqueries named `Candidates` *and* `Filter`*.* `Candidates` *corresponds to the original query q_2, except that it uses the* `distinct` *keyword. The reason for this is that* `orderkey` *is a key, and therefore its values appear exactly once in every repair. In this case, the result of applying* `Candidates` *to the database is $\{o1, o2, o3, o4, o5\}$. The query* `Filter` *returns the orders that should be "filtered out" from the result of* `Candidates` *because they are not consistent answers. In this case,* `Filter` *returns $\{o1, o3\}$. The order $o1$ is returned by* `Filter` *because tuple $s1$ joins with tuple $t2$, which corresponds to a customer whose account balance is below 1000 (*`c.acctbal≤1000` *in the* `Filter`*). The order $o3$ is in* `Filter` *because $o3$ appears in tuple $s4$. This tuple does not satisfy the join condition of q_2 because $c4$ does not appear in relation* `customer`*. Notice that* `Filter` *computes a left-outer join between* `order` *and* `customer`*. Since tuple $s4$ does not join with any tuple of* `customer`*, $o3$ appears together with a null value for attribute* `custkey` *in the left-outer join. Therefore, the condition* `c.custkey is null` *is satisfied, and $o3$ is returned by the filter.*

The rewriting Q_2^c finally retrieves the orders in the result of `Candidates` *that are not in* `Filter` *(i.e., they are not filtered out). In this case, it returns $\{o2, o4, o5\}$, which is the consistent answer.*

```
with Candidates as (
   select distinct o.orderkey, o.clerk
   from customer c, order o
   where c.acctbal>1000 and o.custfk=c.custkey),
Filter as (
   select o.orderkey
   from Candidates Cand
        join order o on Cand.orderkey = o.orderkey
        left outer join customer c
                    on o.custfk=c.custkey
   where c.custkey is null or c.acctbal≤1000
 union all
   select orderkey
   from Candidates Cand
   group by orderkey
   having count(*) > 1)
select clerk
from Candidates Cand
where not exists (select * from Filter F
              where Cand.orderkey = F.orderkey)
```

Figure 4: The rewriting Q_3^c for query q_3

To generalize the previous example to an algorithm for tree queries, we must consider how to handle projection. Note that query q_2 returns the key of the relation at the root of the join graph. But if we modify it to return other attributes instead, we must augment the rewriting. We illustrate this in the next example.

EXAMPLE 4. *Consider a query q_3 which retrieves the clerks who have processed orders for customers with a balance over 1000.*

```
q3 : select o.clerk
     from customer c, order o
     where c.acctbal>1000 and o.custfk=c.custkey
```

Note that the only difference with query q_2 of the previous example is that we are projecting on `clerk` *instead of* `orderkey`*. We will again consider the inconsistent database of Figure 2, and the key query constraints* `orderkey` *and* `custkey`*.*

A query rewriting Q_3^c that computes the consistent answers for q_3 is given in Figure 4. Note that in the `Candidates` *subquery, we project not only on* `clerk` *but also on* `orderkey`*. The reason is that the "filtering" must be done based on the key of the relation. In general, we will filter using the key attributes of the relation at the root of the join graph of the query (*`order` *in this case). If we apply* `Candidates` *over the database we get $\{(o1, ali), (o2, jo), (o2, ali), (o3, pat), (o4, ali),$*

$(o5, ali)\}$. As in the previous example, the tuples for $o1$ and $o3$ should be filtered out. Additionally, in this case, we should filter out the tuples for $o2$. The reason is that the clerk of $o2$ may be jo in some repairs, and ali in others. Hence, $o2$ should not contribute its clerks to the consistent query answer of q_3. This is captured with the second subquery of `Filter` *in Figure 4.*

The rewriting Q_3^c finally takes the tuples of the result of `Candidates` *whose order is not retrieved by* `Filter`*, and projects on the* `clerk` *attribute. The final result is $\{ali, ali\}$, which is the consistent answer. Notice that the rewriting computes not only the fact that ali is a consistent answer, but also the correct multiplicity.*

In Figure 5, we give the rewriting algorithm **RewriteJoin** for tree queries without aggregation. The subquery `Candidates` corresponds to the original query, except for its `select` clause. First, this clause uses the `distinct` keyword. Second, it projects on the key attributes of the root relation of the join graph (denoted as `Kroot`). We will call the values computed by `Candidates` the *candidates* for the consistent answers.

We assume that all tuple variables have the name of the relations of the original query. This condition can be relaxed, as long as tuple variables are used consistently in the expressions $\mathcal{NSC}$, $\mathcal{LOJ}$ and $\mathcal{KJ}$. Furthermore, for notational convenience, we take the liberty of treating *vectors* of attributes as if they were a single attribute.

The first subquery of `Filter` returns the set of candidates that are not present in the query answer from at least one repair. To ensure that `Filter` considers all candidates, there is an inner join between `Candidates` and the relation at the root of the join graph (`Rroot`). The relation `Rroot` is joined with the rest of the relations of the original query (expression $\mathcal{LOJ}$). Since we want to be able to detect the cases in which non-key to key joins are not satisfied, we need to perform a left-outer join rather than an inner join. This can be done in our case because we are considering queries whose join graph is a tree. In particular, the left-outer join of the relations is obtained starting at the relation at the root of the join graph (tree), and recursively traversing it in the direction of its arcs (that is, from a relation joined on a non-key attribute to a relation joined on its key). We denote the expression with $\mathcal{LOJ}$, and present the procedure to construct it in Figure 6.

We can now explain the `where` clause of the first subquery of `Filter`. First, it contains an expression $\mathcal{KJ}$ that consists of the key-to-key joins of the query. These joins do not need to be considered in the left-outer join $\mathcal{LOJ}$. To understand why, notice that if a candidate satisfies a key-to-key join, then it must satisfy it in every repair (since every key value appears in every repair). Second, the subquery checks whether there is a tuple which does not satisfy a join of the original query. This is done by checking whether there is a null value in the left-outer join. Finally, it checks whether there is some selection condition of the original query that is not satisfied (expression $\mathcal{NSC}$).

The second subquery of `Filter` takes care of the projected attributes of the original query, as explained in Example 4. Finally, the query rewriting returns the tuples for the candidates that were not filtered out. `RewriteJoin` is a correct query rewriting algorithm for tree queries without aggregation, as stated in the next theorem.

RewriteJoin(q, Σ)

Given a query q of the form

```
            select S
            from Rroot, $R_1, \ldots, R_m$
            where $\mathcal{KJ}$ and $\mathcal{NKJ}$ and $\mathcal{SC}$
            order by $\mathcal{O}$
```

where
- *Rroot* is the relation at the root of the join graph
- $\mathcal{KJ}$ is a conjunction of key-to-key join predicates
- $\mathcal{NKJ}$ is a conjunction of nonkey-to-key join predicates
- $\mathcal{SC}$ is a conjunction of selection predicates

Let
- *Kroot* be the attributes of the key of *Rroot*
- $K_1, \ldots, K_m$ be the attributes of the keys of $R_1, \ldots, R_m$
- $\mathcal{NSC}$ be the *negation* of the selection predicates $\mathcal{SC}$ of q
- $\mathcal{LOJ}$ be the left outer-join of the join graph of q, obtained as shown in Figure 6

The rewriting Q^c of q is the following:

```
with Candidates as (
   select distinct Kroot, S
   from Rroot, $R_1, \ldots, R_m$
   where $\mathcal{KJ}$ and $\mathcal{NKJ}$ and $\mathcal{SC}$)
Filter as (
   select Kroot
   from Candidates C
        join Rroot  on C.Kroot = Rroot.Kroot
        left outer join $\mathcal{LOJ}$
   where $\mathcal{KJ}$ and
      ($R_1.K_1$ is null or ...  or $R_m.K_m$ is null
       or $\mathcal{NSC}$)
 union all
   select Kroot
   from Candidates C
   group by Kroot
   having count(*)>1)
select S
from Candidates C
where not exists (select * from Filter F
                  where C.Kroot = F.Kroot)
order by $\mathcal{O}$
```

Figure 5: Query rewriting algorithm for queries without aggregation

THEOREM 1. *Let q be a tree query without aggregate operators, and Σ be a set of query constraints containing at most one key constraint per relation. Then, the rewriting* **RewriteJoin**(q, Σ) *computes the consistent query answers for q wrt Σ on every database D.*

4. AGGREGATION QUERIES

In this section, we present ConQuer's rewriting strategy for tree queries that may have grouping and aggregation. The obvious extension of the semantics of consistent answers is to return values for the aggregate expressions that hold in every repair. However, as we motivate in the next example, this may be overly restrictive in some cases.

EXAMPLE 5. *Consider a query q_4 that retrieves the sum of all account balances in the* `customer` *relation. Again,* `custkey` *is the key query constraint.*

```
LOJ($T$):
  Let $R$ be the relation at the root of $T$
  If $T$ is a leaf then return
  else
    Let $T_1, ..., T_m$ be the subtrees of the root of $T$
    for $i = 1$ to $m$
        Let $N_i$ be non-key attrs of $R$ that are
          equated with the key $K_i$ of $R_i$
    return "$R_1$ on $R.N_1 = R_1.K_1$
          left outer join ...
          left outer join $R_m$ on $R.N_m = R_m.K_m$
          left outer join LOJ($T_1$)
          left outer join ...
          left outer join LOJ($T_m$)"
```

Figure 6: Left outer-join of join graph T

	cust key	nation key	mkt segment	acct bal
t1	c1	n1	building	1000
t2	c1	n1	building	2000
t3	c2	n1	building	500
t4	c2	n1	banking	600
t5	c3	n2	banking	100

Figure 7: Inconsistent instance of `customer`

```
q4 : select sum(acctbal) as sumBal
     from customer
```

We will consider the inconsistent database of Figure 7. In this case, there are four repairs: $D_1^{\mathcal{R}} = \{t1, t3, t5\}$, $D_2^{\mathcal{R}} = \{t1, t4, t5\}$, $D_3^{\mathcal{R}} = \{t2, t3, t5\}$ and $D_4^{\mathcal{R}} = \{t2, t4, t5\}$. In repair $D_1^{\mathcal{R}}$, the sum of account balances is 1600; in $D_2^{\mathcal{R}}$, it is 1700; in $D_3^{\mathcal{R}}$, 2600; and in $D_4^{\mathcal{R}}$, 2700. Hence, the consistent query answers to q_4 are empty. In fact, it suffices to have one customer with two (inconsistent) account balances in order to have an empty answer for the aggregation on all customers.

Arenas et al. [3] proposed to return bounds for the aggregate expressions, rather than exact values. For instance, in the previous example we can say that $1600 \leq$ `sumBal` ≤ 2700. In order to consider such ranges, we need to slightly modify the semantics of consistent query answers to what we call *range-consistent query answers*.

We will consider SQL queries of the following form:

```
select $G, agg_1(e_1)$ as E1, ..., $agg_n(e_n)$ as En
from $\mathcal{F}$
where $\mathcal{W}$
group by $G$
```

where G is the set of attributes we are grouping on, and $agg_1(e_1), ..., agg_n(e_n)$ are aggregate expressions with functions $agg_1, \ldots, agg_n$, respectively. We will assume that the `select` clause renames the aggregate expressions to `E1, ..., En`. Notice that we are focusing on queries where all the attributes in the `group by` clause appear in the `select` clause. This is a restriction because, in general, SQL queries may have some attributes in the `group by` clause which do not appear in the `select` clause (although not vice versa).

In the definition of range-consistent query answers, we will give a range for each value of G that is a consistent answer. Therefore, we will use a query q_G that consists of q with all the aggregate expressions removed from the `select` clause:

```
$q_G$: select $G$
    from $\mathcal{F}$
    where $\mathcal{W}$
    group by $G$
```

We now introduce the definition of *range-consistent query answers*, and illustrate it with an example.

DFN 5 (RANGE-CONSISTENT QUERY ANS). *Let D be a database, q be a query, and Σ be a set of query constraints. We say that* $(\mathbf{t}, \mathbf{r})$ *is a* range-consistent query answer for q on D *if*

1. $\mathbf{t}$ *is a consistent answer for q_G on D, where q_G is query q with all the aggregate expressions removed; and*
2. $\mathbf{r} = (min_{E1}, max_{E1}, \ldots, min_{En}, max_{En})$, *and for each i such that $1 \leq i \leq n$:*
   - $min_{Ei} \leq \Pi_{Ei}(\sigma_{G=\mathbf{t}}(q(D^{\mathcal{R}}))) \leq max_{Ei}$ *for every repair $D^{\mathcal{R}}$; and*
   - $\Pi_{Ei}(\sigma_{G=\mathbf{t}}(q(D^{\mathcal{R}}))) = min_{Ei}$*, for some repair $D^{\mathcal{R}}$; and*
   - $\Pi_{Ei}(\sigma_{G=\mathbf{t}}(q(D^{\mathcal{R}}))) = max_{Ei}$*, for some repair $D^{\mathcal{R}}$.*

EXAMPLE 6. *Consider the following query, which retrieves the total account balance for customers in the building sector, grouped by nation. Again,* `custkey` *is a key query constraint.*

```
q5: select nationkey, sum(acctbal)
    from customer
    where mktsegment = 'building'
    group by nationkey
```

Let q_G be query q_5 with its aggregate expression removed.

```
qG: select nationkey
    from customer
    where mktsegment = 'building'
    group by nationkey
```

Consider again the database of Figure 7. It is easy to see that nation $n1$ is the only consistent answer to q_G. Therefore, the range-consistent answer consists of a range of values for $n1$.

Recall that there are four repairs for the database: $D_1^{\mathcal{R}} = \{t1, t3, t5\}$, $D_2^{\mathcal{R}} = \{t1, t4, t5\}$, $D_3^{\mathcal{R}} = \{t2, t3, t5\}$ and $D_4^{\mathcal{R}} = \{t2, t4, t5\}$. The result of applying q_5 on the repairs is the following: $q_5(D_1^{\mathcal{R}}) = \{(n1, 1500)\}$; $q_5(D_2^{\mathcal{R}}) = \{(n1, 1000)$; $q_5(D_3^{\mathcal{R}}) = \{(n1, 2500)\}$; and $q_5(D_4^{\mathcal{R}}) = \{(n1, 2000)\}$. Hence, the consistent answers to q_5 is the set $\{(n1, 1000, 2500)\}$, because the sum of the account balances for customers in the building sector and nation $n1$ is:

- *between 1000 and 2500, in every repair;*
- *1000 in repair $D_2^{\mathcal{R}}$;*
- *2500 in repair $D_3^{\mathcal{R}}$.*

Before presenting the rewriting that computes the range-consistent query answers, let us illustrate the approach with some examples.

EXAMPLE 7. *Suppose that we want to obtain a rewriting that computes the range-consistent answers for query q_5 of the previous example. We are going to obtain upper and lower bounds for the account balance of each customer, and then sum the account balances. As in the previous section, we are going to filter some of the customers. We will use the filter for q_G, which can be obtained using the algorithm* `RewriteJoin` *of Figure 5. Intuitively, the filter retrieves the customers which appear in some tuple that does not satisfy q_G.*

Consider again the database of Figure 7. When we apply the filter to the `customer` *table, $c2$ and $c3$ are filtered out. The customer $c1$ is not filtered because its two tuples ($t1$ and $t2$) satisfy query q_G. In repairs $D_1^{\mathcal{R}}$ and $D_2^{\mathcal{R}}$, $c1$ contributes an account balance of 1000. In $D_3^{\mathcal{R}}$ and $D_4^{\mathcal{R}}$, it contributes 2000. Therefore, it contributes a* minimum *of 1000 and a* maximum *of 2000. We can capture this with the following query:*

```
with UnFilteredCandidates as (
   select custkey, nationkey,
          min(acctbal) as minBal,
          max(acctbal) as maxBal
   from customer c
   where mktsegment = 'building'
      and not exists (select * from Filter
                   where c.custkey=Filter.custkey)
   group by custkey, nationkey)
```

The result of applying `UnFilteredCandidates` *to the inconsistent database is $\{(c1, n1, 1000, 2000)\}$. Notice that the filter is necessary because we would otherwise get the tuple $(c2, n1, 500, 500)$ in the result, which states that customer $c2$ contributes an amount of 500 in every repair. This is not correct, since in repairs $D_2^{\mathcal{R}}$ and $D_4^{\mathcal{R}}$ (i.e., the repairs where $t4$ appears), customer $c2$ does not satisfy query q_G and therefore does not contribute to the sum of account balances. Therefore, $c2$ contributes a minimum of 0 and a maximum of 500. This is captured with the following query:*

```
with FilteredCandidates as (
   select custkey, nationkey,
          0 as minBal,
          max(acctbal) as maxBal
   from customer c
   where mktsegment = 'building'
      and exists (select * from Filter
               where Filter.custkey=c.custkey)
      and exists (select * from QGCons
               where QGCons.nationkey=c.nationkey)
   group by custkey, nationkey)
```

The result of `FilteredCandidates` *is $\{(c2, n1, 0, 500)\}$. In addition to checking that the customer is filtered, we check that the nation (i.e., the attribute in the* `group by` *of the original query) appears in the result of the consistent answers to q_G (denoted as* `QGCons` *in the query). This is necessary because we do not want to retrieve ranges for the nations that are not consistent answers.*

Finally, we obtain the range-consistent answers by summing up the lower and upper bounds for each nation in the result of `FilteredCandidates` *and* `UnfilteredCandidates`*, as follows:*

```
select nationkey,
       sum(minBal),sum(maxBal)
from (select * from FilteredCandidates
      union all
      select * from UnfilteredCandidates)
group by nationkey
```

In the previous example, all the numerical values were positive. The following example shows how to produce a rewriting that deals with negative values as well.

EXAMPLE 8. *Consider query q_5, and a database with the following* `customer` *relation. The only difference with the relation of Figure 7 is that the account balance in tuple $t3$ is negative.*

	cust key	*nation key*	*mkt segment*	*acct bal*
t1	*c1*	*n1*	*building*	*1000*
t2	*c1*	*n1*	*building*	*2000*
t3	*c2*	*n1*	*building*	*-500*
t4	*c2*	*n1*	*banking*	*600*
t5	*c3*	*n2*	*banking*	*100*

The repairs are the same as in the previous example. The result of applying q_5 on the repairs is the following: $q_5(D_1^{\mathcal{R}}) = \{(n1, 500)\}$; $q_5(D_2^{\mathcal{R}}) = \{(n1, 1000)\}$; $q_5(D_3^{\mathcal{R}}) = \{(n1, 1500)\}$; and $q_5(D_4^{\mathcal{R}}) = \{(n1, 2000)\}$. Thus, the range-consistent answer to q_5 is $\{(n1, 500, 2000)\}$.

As in the previous example, customer $c1$ is unfiltered, and customers $c2$ and $c3$ are filtered. Also, $c1$ contributes to the upper and lower bound of account balances, and $c3$ does not contribute to any of them. On the other hand, the account balance of customer $c2$ in tuple $t3$ contributes to the lower bound of the sum of account balances, as opposed to the upper bound in the previous example. This is because in the repairs where customer $c2$ appears in tuple $t3$, the total account balance is reduced rather than increased. In the repairs where customer $c2$ does not satisfy query q_G (i.e., $t4$ is in the repair) the contribution to the total account balance is zero. We capture this with the following query:

```
with FilteredCandidates as (
   select custkey, nationkey,
          min(acctbal) as minBal,
          0 as maxBal
   from customer c
   where mktsegment = 'building'
      and exists (select * from Filter
               where Filter.custkey=c.custkey)
      and exists (select * from QGCons
               where QGCons.nationkey=c.nationkey)
   group by custkey, nationkey)
```

Notice that the only difference with `FilteredCandidates` *from the previous example is that there is a value of zero in* `maxBal`*, instead of* `minBal`.

The full rewriting **RewriteAgg** for tree queries (with aggregation and grouping) is given in Figure 8. The query `QGCons` retrieves the consistent answers to q_G. It is not included in the figure because it can be obtained by applying

```
RewriteAgg(q, Σ)

Given a query q of the form
       select G, agg_1(e_1) as E1,...,agg_n(e_n) as En
       from 𝓕
       where 𝓦
       group by G

Let
  Rroot be the relation at the root of the join graph of q,
  Kroot be the attributes of the key of Rroot,
  q_G be query q with all the aggregate expressions removed
  QGCons be the query that obtains the consistent answers
         for q_G, using RewriteJoin
  Filter be the filter subquery of QGcons

The rewriting Q^rc of q is the following:

with UnFilteredCandidates as (
 select Kroot, G, min(e_1) as minE1, max(e_1) as maxE1,
             ..., min(e_n) as minEn, max(e_n) as maxEn
 from 𝓕
 where 𝓦
       and not exists (select * from Filter
                       where 𝓕.Kroot = Filter.Kroot)
 group by Kroot, G),
FilteredCandidates as (
 select Kroot, G,
 case when minE1 > 0 then 0 else min(e_1) as minE1,
 case when maxE1 > 0 then max(e_1) else 0 as maxE1,
       ...,
 case when minEn > 0 then 0 else min(e_n) as minEn,
 case when maxEn > 0 then max(e_n) else 0 as maxEn
 from 𝓕
 where 𝓦
       and exists (select * from Filter
                   where 𝓕.Kroot = Filter.Kroot)
       and exists (select * from QGCons
                   where 𝓕.G = QGCons.G)
 group by Kroot, G ),
select G, agg_1(minE1), ..., agg_n(minEn), ...,
          agg_1(maxE1), ..., agg_n(maxEn)
from (select * from FilteredCandidates
      union all
      select * from UnfilteredCandidates)
group by G
```

Figure 8: Rewriting for queries with aggregation

the rewriting algorithm `RewriteJoin` of the previous section. Recall that in the examples we obtained lower and upper bounds for each customer and nation. The customer was the key attribute of the relation at the root of the join graph (denoted by `Kroot` in the algorithm). The nation was the attribute we were grouping on (denoted with `G`). Thus, in the algorithm we obtain bounds for the attributes `Kroot` and `G`. We take the liberty of denoting by $\mathcal{F}$`.Kroot=Filter.Kroot` and $\mathcal{F}$`.G=QGCons.G` the join between all the attributes of `Kroot` and `G` (associated to the appropriate relations of $\mathcal{F}$) with the corresponding attributes of `Filter` and `QGCons`. The query `UnFilteredCandidates` obtains the bounds for the values of `Kroot` that are not filtered, and therefore contribute to both bounds. The query `FilteredCandidates` obtains the bounds for the values that are filtered, and therefore may contribute zero to some of the bounds. The `case` statements are used to select the appropriate upper and lower bounds, depending on whether they are positive or negative values. This generalizes the strategy presented in Example 8 for negative values. The final result is obtained by aggregating the upper and lower bounds from `UnFilteredCandidates` and `FilteredCandidates` using the aggregation functions of the original query.

THEOREM 2. *Let q be a tree query that may contain aggregate expressions (with functions* `MAX`*,* `MIN`*,* `SUM`*), and Σ be a set of query constraints containing at most one key constraint per relation of the query. Then, the rewriting* **RewriteAgg**(q, Σ) *computes the range-consistent query answers for q with respect to Σ on every database D.*

order

	order key	clerk	cust fk	cons
s1	o1	ali	c1	y
s2	o2	jo	c2	n
s3	o2	ali	c3	n
s4	o3	ali	c4	n
s5	o3	pat	c2	n
s6	o4	ali	c2	n
s7	o4	ali	c3	n
s8	o5	ali	c2	y

customer

	cust key	acct bal	cons
t1	c1	2000	n
t2	c1	100	n
t3	c2	2500	y
t4	c3	2200	n
t5	c3	2500	n

Figure 9: Annotated version of the database of Figure 2

5. ANNOTATED DATABASES

If the query constraints are known in advance, ConQuer can process the database offline in order to store information about constraint violations. In particular, it can annotate every tuple of the database with a flag that states whether the tuple (might) violate a key constraint. This flag is then used to produce optimized query rewritings. Note that a standard query optimizer would not be able to exploit the annotations because it is unaware of their semantics (and the semantics of consistent query answering in general).

In Figure 9, we show an annotated version of the database of Figure 2. The annotations are made assuming that the attribute `orderkey` is the key of the `order` relation, and `custkey` is the key of the `customer` relation. Note that the schema of every relation is augmented with an attribute `cons` that stores the annotation. If the value of `cons` in a tuple is $'y'$, then the tuple satisfies the key constraint; a value of $'n'$ indicates that it might violate the constraint. We illustrate the optimized rewriting with the next example.

EXAMPLE 9. *We will consider again query q_2 from example 3, which retrieves the orders placed by customers with an account balance over 1000. The query constraints are again that* `orderkey` *and* `custkey` *are keys of their relations. We assume that ConQuer knows them in advance, and has produced the annotated database of Figure 9.*

```
q2: select orderkey
    from customer c, order o
    where c.acctbal>1000 and o.custfk=c.custkey
```

Note that tuples s8 and t3 have a value of $'y'$ in their `cons` *attributes, meaning that they do not violate any constraint.*

If we join $s8$ with $t3$, we get a tuple that satisfies query q_2. Furthermore, it is easy to see that this will be the only *tuple in the result for order $o5$. Thus, it must be a consistent answer.*

In general, if a tuple is produced from the join of tuples that satisfy the constraints, it is a consistent answer. Thus, it is not necessary to consider it in the `Filter` *subquery of the rewriting. Note, however, that this is not the case even if some (but not all) of the tuples that are joined satisfy the constraints. For example, consider the join of tuple $s1$ (annotated with 'y') and tuple $t1$ (annotated with 'n'). If we join them, we get a tuple that satisfies q_2. However, it is not a consistent answer because the result of joining $s1$ with another tuple ($t2$) does not satisfy the query.*

To keep track of the join of tuples violating a constraint, we augment the `Candidate` *subquery of the rewriting with an addition attribute* `consCand`*. This is a numerical attribute that calculates the number of tuples involving, in this case, an order that joins with tuples violating a constraint. If this value is greater than zero, the tuple might not be in the consistent answer. The* `Candidates` *subquery for this example is the following:*

```
select o.orderkey
   sum (case when (c.cons='n' or o.cons='n')
             then 1 else 0 end ) as consCand
from customer c, order o
where c.acctbal>1000 and o.custfk=c.custkey),
group by o.orderkey
```

In the rewriting presented in Section 3, the `Filter` *subquery joins all tuples of* `Candidates` *with the relation at the root of the tree (*`order` *in this case). Then, it performs a (possibly costly) left-outer join with all other relations of the original query. We can now avoid joining all tuples of* `Candidates` *because we can select only those whose value of* `consCand` *is greater than zero. This is because a value of zero in this attribute means that the tuple is known to be in the consistent answer and, therefore, cannot be filtered out. The following is the* `Filter` *subquery for this example.*

```
select o.orderkey
from Candidates Cand
     join order o on Cand.orderkey=o.orderkey
     left outer join customer c
                 on o.custfk=c.custkey
where Cand.consCand>0
      and c.custkey is null or c.acctbal≤1000
```

Note that the only difference with the `Filter` *of Figure 3 is that we add the condition* `Cand.consCand>0` *to the* `where` *clause. Although it is up to the query optimizer to perform this selection before the joins, the results of the next section show that it consistently chooses the appropriate strategy. The final subquery of the rewriting is the same as in Figure 3. That is, it returns the orders from* `Candidates` *which do not appear in* `Filter`.

6. EXPERIMENTAL EVALUATION

In this section, we report results for the experiments that we performed in order to quantify the overhead of the rewritings produced by ConQuer. The experiments use realistic queries (taken from the TPC-H specification) and large databases.

6.1 Setup

The experiments were performed on a Dell Optiplex 170L PC with a 2.8 Ghz Intel Pentium 4 CPU and 1 GB of RAM (80 % of which was allocated to the database manager). The queries were run on DB2 UDB version 8.1.8 under Windows XP Professional.

We present experimental results for six queries taken from the TPC-H specification, which are representative of a variety of features supported by our approach. For example, they all have aggregation, and range from one to six joins.[4]

We focus on queries 1, 3, 4, 6, 10, and 12 of the standard. In Figure 10, we summarize the main characteristics of these queries. For each query, we give the number of relations in the `from` clause, the selectivity (high means more tuples satisfy the query), the number of attributes in the `select` clause, and the number of those attributes which are in aggregate expressions. The queries in the TPC-H specification are parameterized, and the standard suggests values for these parameters. In the experiments, we used the suggested values in all the queries. The selectivity we report in Figure 10 takes these parameters into account.

	Relations	Selectivity	ProjAttrs	AggrAttrs
Q1	1	high	10	8
Q3	3	low	4	1
Q4	2	low	2	1
Q6	1	low	1	1
Q10	4	low	8	1
Q12	2	low	3	2

Figure 10: Queries used in the experiments

The TPC-H specification assumes that databases are consistent with respect to the primary keys of the schema. For the experiments, we assumed that primary keys are not part of the schema, but are rather specified as query constraints. The rewritings that we produced take these query constraints into account.

We experimented with a number of (inconsistent) databases, considering the following parameters:

- The size s of the database. We considered databases of 100 and 500 MB, and 1 and 2 GB. A database of 1 GB has 8 million tuples.
- The percentage p of the database that is inconsistent. For example on a 1 GB instance (8 million tuples) where p is 50%, there are 4 million tuples that violate the key constraints of the query. We created the databases in such a way that every relation has the same value of p as the entire database. We experimented with values of p of 0, 1, 5, 10, 20, and 50. We deliberately consider high values of p to test the limits of our approach.
- The number of tuples n that share a common key value (and hence violate the key constraint), for every key

[4]Some of the TPC-H queries cannot be handled by our approach because they contain such features as left-outer-joins or disjunction. There are also queries with nested subqueries. However, many of them can be decorrelated and unnested.

value in the inconsistent portion of the database. For example, if $n = 2$, then every key value in the inconsistent portion of the database appears in exactly two tuples. We experimented with values of n ranging from 2 to 50. Note that p and n together determine the level of inconsistency of the database.

Since the TPC-H data generator (`dbgen`) creates instances that do not violate the primary key constraints, we developed a simple program that generates inconsistent databases. The program works as follows. Suppose that we want to generate an inconsistent database of 1 GB where $p = 10\%$ and $n = 2$. The program first invokes the TPC-H data generator and creates a consistent instance of size 0.95 GB. Then, it selects two sets of tuples of size 0.05 GB from the database. The tuples are selected randomly from a uniform distribution. One of the sets is used to draw the key values of the conflicting tuples that the program will introduce to the database; the other set is used to obtain non-key values.

For each query q, we used ConQuer to generate a rewritten query that takes annotations into account, and another which does not assume an annotated database. In the following, we will say that the queries produced by the former strategy are *annotation-aware*.

Our rewritings contain some common subexpressions specified with the `WITH` clause (such as `Candidates` and `Filter`). In the experiments, we found that running times improve considerably when the results of these subexpressions are temporarily stored rather than computed several times for the *same* query. For some queries, the DB2 optimizer realized that materialization was the best strategy, but for others it did not. So to have a consistent comparison, we materialized all common subexpressions using temporary tables. We include the materialization time in the query running times. Furthermore, we ensured that these materialized results were not cached across runs of the queries.

We created indices for the query constraints and for the annotations (the attribute `cons`). Notice that, since we consider databases that violate the key query constraints, they cannot be defined as unique keys. For each index, we created statistics using the DB2 RUNSTATS command.

6.2 Experimental Results

In Figure 11, we compare the running times of all queries with the rewritings produced by ConQuer.[5] The results correspond to a 1 GB database with 5% of inconsistency and $n = 2$. Although the rewritten queries are more expensive, the overhead is reasonable if we take into account that, in general, consistent query answering is much more onerous than standard query answering [4, 5].

We calculate the overhead of the rewritten queries as $\frac{t_r - t_o}{t_o}$, where t_o and t_r are the running times of the original and rewritten queries, respectively. For the annotation-aware rewritings of all queries except Q1, the overhead is below 0.52 times the running time of the original query. For the ones that do not use annotations, the overhead is less than 0.86 for all queries except Q1. We will justify the high over-

[5]To facilitate the comparison of the running times of the other queries, we leave the running times for the rewritings of Q1 out of scale. The running time of the annotation-aware rewriting for Q1 is 8.3 minutes. The other rewriting runs in 10.1 minutes.

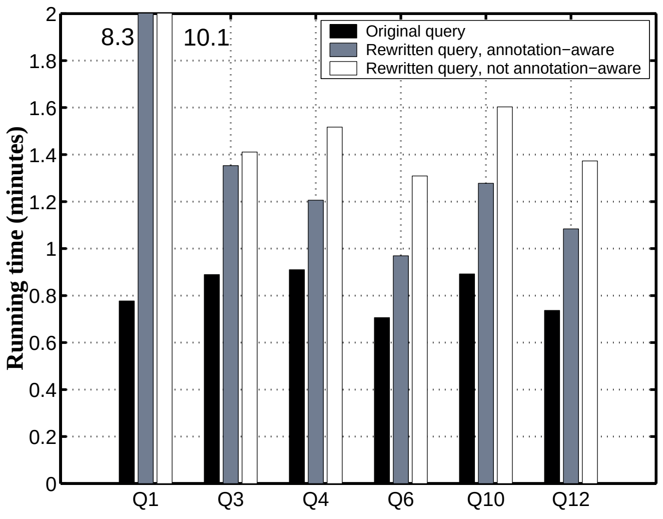

Figure 11: Running time for all queries on a 1 GB database

head of the rewritings of Q1 momentarily. Of the two rewriting strategies, the one that is annotation-aware consistently produces more efficient rewritings. The difference in the running times of the two rewritings ranges from 4% (Q3) to 26% (Q6). We will shortly show a study where the savings of the annotation-aware rewritings are even greater.

We argue that the overhead of the rewritten queries is dominated by the size of the answer to the original query (without considering grouping). This is certainly the case in the database used for the experiments of Figure 11, where the rewritings for Q1 have the largest overhead. If we remove all grouping from Q1, we get a result set of 5.9 million tuples. The size of the result set for all the other queries is below 120,000 tuples. We justify our conjecture in terms of the query rewriting algorithm **RewriteAgg** of Figure 8. In that figure, note that the rewritten queries are obtained as the union of two queries: `UnfilteredCandidates` and `FilteredCandidates`. The former selects tuples that are not in `Filter`. The latter selects tuples that are in `Filter` and in `QGCons`. `QGCons` is the query that obtains the consistent answers for the original query without aggregation and grouping. Thus, it is obtained with the algorithm **RewriteJoin** of Figure 5. In this algorithm, the rewritten query returns the tuples in the result of `Candidates` which are not in the result of `Filter`. Note in Figure 5 that `Filter` focuses solely on tuples that come from the result of `Candidates`. In turn, `Candidates` is obtained from the original query (without grouping and aggregation) except for some changes to its `select` clause.

We also studied the effect of the amount of inconsistency in the database. Here, we report results for one query (Q6) that is representative of the trend that we observed on the other queries as we varied the values of p and n for databases of 1 GB. In Figure 12, we present the running times for databases with a varying percentage p of inconsistency (from 0 to 50%) for a fixed $n = 2$. We can observe that p has little influence on the running time of the original query and the rewriting that is unaware of annotations. The size of the result set of Q6 remains constant as we vary the value of p because all databases are of the same size. This is yet another evidence that the size of the result set of the original query has a considerable impact on the running time of the rewritten queries.

The annotation-aware rewriting is considerably influenced by p. In Figure 12, the overhead of the annotation-aware rewritings ranges from 0.02 (for $p = 0\%$) to 0.56 (for $p = 50\%$). The reason for this is that the annotation-aware rewriting is designed to focus as much as possible on the consistent portion of the database, and thus benefits if the database has a low value of p. Note that on the totally consistent database, the running times of the original query and its annotation-aware rewriting are very close (the overhead is 0.02). Thus, this rewriting is particularly useful for databases that are almost consistent (i.e., the value of p is close to zero).

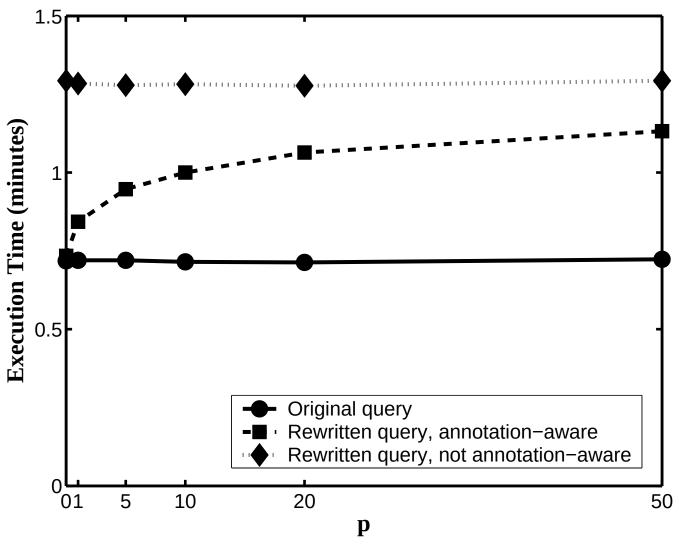

Figure 12: Running time of Q6 over p

In Figure 13, we present the running times for Q6 over databases with varying values of n and a fixed $p = 10\%$. Note that the value of n has little influence on the running time of any of the two alternative rewritings.

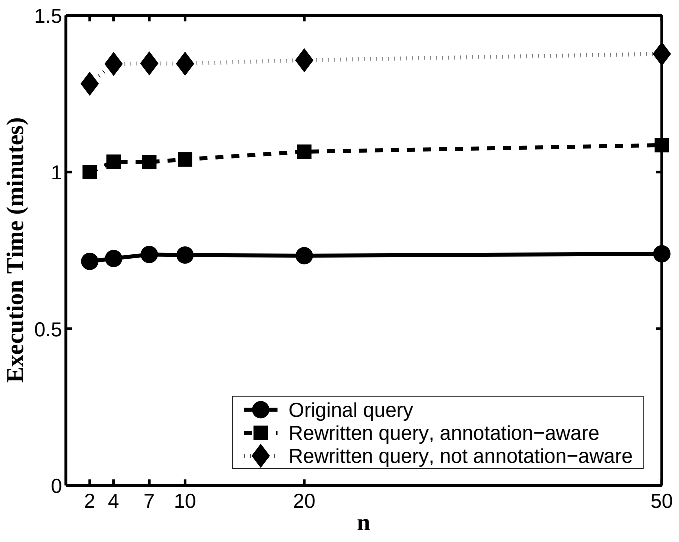

Figure 13: Running time of Q6 over n

Finally, we studied the scalability of our approach. For this, we employed databases of 100 MB, 500 MB, 1 GB, and 2 GB. To ensure a consistent comparison, we present results for databases where the total number of tuples violating the constraints is kept constant. In particular, in Figure 14 we present results for databases with 400,000 inconsistent tuples. Thus, the values of p are 2.5, 5, 10, and 50 for the 2 GB, 1 GB, 500 MB and 100 MB databases, respectively. In all databases, we kept $n = 2$. In the figure, we report the running time of the annotation-aware rewritings for queries 4, 6, and 12. It can observed that the running times grow in a linear fashion with the size of the database.

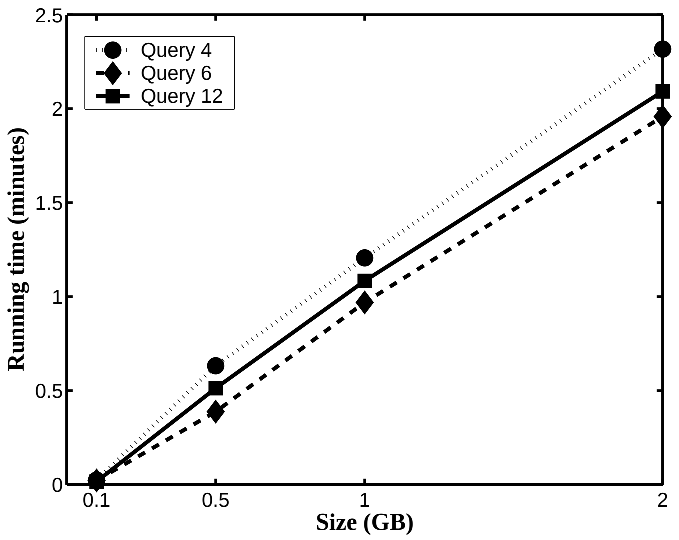

Figure 14: Running time of the rewritten queries over database size

7. RELATED WORK

There has been considerable work on the semantics and complexity of consistent query answering. We do not review all of this work, but instead concentrate on results that are the most related to building practical consistent query answering systems. The computational complexity of the consistent query answering problem has been thoroughly studied in the literature, for different classes of queries and constraints [4, 5]. In the case of SPJ queries with one key query constraint per relation, the problem is known to be co-NP complete in general [4, 5]. However, there remain large practical classes of queries for which the problem is much easier to compute.

First-order query rewritings are relevant in our context because queries in first-order logic can be translated into SQL. Arenas et al. [2] were the first to propose a first-order rewriting algorithm, which is applicable to a class of queries called *quantifier-free*. This class is quite restricted as it requires all attributes to appear in the `select` clause (and hence prohibits most projections), unless an attribute has been equated to another attribute that is in the `select` clause. This work was the foundation for Hippo [6], which, to the best of our knowledge, is the only existing system for consistent query answering on large databases. However, the approach taken by Hippo is quite different from

ConQuer. First, Hippo is not based on query rewriting. Rather, it takes the more procedural approach of producing a Java program which computes the consistent answers. Although the program does interact with a RDBMS back-end, most of the processing is done by processing a main memory (conflict-graph) data structure that contains all the tuples that violate the constraints. Hippo extends the select-join query class of Arenas et al. [2] to consider union (disjunction), an operation we are not considering.

The rewriting algorithm presented in this paper is motivated by our previous work on rewriting conjunctive queries (that is, SPJ queries with set semantics) into first-order logic queries [11]. In the current work, we consider not only conjunctive queries but also SPJ queries with aggregation, grouping, and bag semantics. Furthermore, in our previous work, we were not concerned with the running time of the rewritten queries. In principle, first-order queries can be translated into SQL. However, the queries produced in [11] have a high level of nesting (proportional to the number of relations in the query) and are therefore very inefficient. The rewriting algorithm in our current work produces SQL queries with at most one level of nesting and, as shown in Section 6, reasonable running times.

Our work on aggregation is inspired by [3], which was the first to propose the use of ranges as a semantics for consistent query answering for aggregate expressions, but considers queries with just one aggregated attribute and no grouping. We extend these results to consider general aggregation queries with grouping.

There are a number of systems for consistent query answering that rewrite queries into powerful logics. Infomix [10, 12] is a notable example of such an approach. In Infomix, queries are rewritten into disjunctive logic programs. Such programs are computationally more expensive than SQL, but also more expressive and permit rewritings over a very rich class of query constraints. For example, Infomix considers general functional, inclusion, and exclusion query constraints. These systems focus on expressiveness, more than efficiency and scalability, and therefore address a different design point than the one we are considering. To give an idea of the scale of the difference, one of the few experimental studies available in the literature [10] reports results for databases with at most 100 tuples violating key query constraints (over a database of 50,000 tuples). In contrast, one of our experiments was performed on a database with 4 million inconsistent tuples (over a total of 8 million tuples). Results for Hippo have also only been reported for a database of up to 300,000 tuples. Hippo constructs an in-memory conflict graph which may limit the number of possible conflicts than can be considered [6, 7].

Finally, it is worth noting that the semantics of consistent query answers is similar to that of *certain answers* that is widely accepted as an important semantics for data integration [1]. For certain answers, the set of possible worlds are the legal instances of a global database, rather than the repairs of an inconsistent database.

8. CONCLUSIONS

We have presented the ConQuer system that, given a set of key query constraints, rewrites SQL queries to SQL queries that return only the consistent answers. We use a strong semantics for "consistent". An answer is consistent if every repair supports the answer. Such a semantics is useful for identifying potential inconsistencies in a database. However, our rewritings extend naturally to a semantics based on voting (find answers supported by at least two repairs), or a semantics under which each tuple is given a probability of being correct [8]. We are currently experimenting with rewritings which return the most probable answer over an inconsistent database in which each tuple is assigned a probability of being consistent.

Acknowledgements. We thank Nick Koudas, Mariano Consens, Periklis Andritsos, and Denilson Barbosa for their feedback and help.

9. REFERENCES

[1] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.

[2] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *PODS*, pages 68–79, 1999.

[3] M. Arenas, L. Bertossi, and J. Chomicki. Scalar Aggregation in FD-Inconsistent Databases. In *ICDT*, pages 39–53, 2001.

[4] A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS*, pages 260–271, 2003.

[5] J. Chomicki and J. Marcinkowski. Minimal-Change Integrity Maintenance Using Tuple Deletions. To appear in *Information and Computation.* CoRR cs.DB/0212004, 2004.

[6] J. Chomicki, J. Marcinkowski, and S. Staworko. Computing Consistent Query Answers using Conflict Hypergraphs. In *CIKM*, pages 417–426, 2004.

[7] J. Chomicki, J. Marcinkowski, and S. Staworko. Hippo: A System for Computing Consistent Answers to a Class of SQL Queries. In *EDBT*, pages 841–844, 2004.

[8] N. Dalvi and D. Suciu. Efficient Query Evaluation on Probabilistic Databases. In *VLDB*, pages 864–875, 2004.

[9] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning.* John Wiley, 2003.

[10] T. Eiter, M. Fink, G. Greco, and D. Lembo. Efficient Evaluation of Logic Programs for Querying Data Integration Systems. In *ICLP*, pages 163–177, 2003.

[11] A. Fuxman and R. J. Miller. First-Order Query Rewriting for Inconsistent Databases. In *ICDT*, pages 337–351, 2005.

[12] D. Lembo, M. Lenzerini, and R. Rosati. Source Inconsistency and Incompleteness in Data Integration. In *KRDB*, 2002.