Clean Answers over Dirty Databases: A Probabilistic Approach

Periklis Andritsos[*]
University of Trento
periklis@dit.unitn.it

Ariel Fuxman[*]
University of Toronto
afuxman@cs.toronto.edu

Renée J. Miller[*]
University of Toronto
miller@cs.toronto.edu

Abstract

The detection of duplicate tuples, corresponding to the same real-world entity, is an important task in data integration and cleaning. While many techniques exist to identify such tuples, the merging or elimination of duplicates can be a difficult task that relies on ad-hoc and often manual solutions. We propose a complementary approach that permits declarative query answering over duplicated data, where each duplicate is associated with a probability of being in the clean database. We rewrite queries over a database containing duplicates to return each answer with the probability that the answer is in the clean database. Our rewritten queries are sensitive to the semantics of duplication and help a user understand which query answers are most likely to be present in the clean database.

The semantics that we adopt is independent of the way the probabilities are produced, but is able to effectively exploit them during query answering. In the absence of external knowledge that associates each database tuple with a probability, we offer a technique, based on tuple summaries, that automates this task. We experimentally study the performance of our rewritten queries. Our studies show that the rewriting does not introduce a significant overhead in query execution time. This work is done in the context of the ConQuer project at the University of Toronto, which focuses on the efficient management of inconsistent and dirty databases.

1 Introduction

The detection of duplicate records that correspond to the same real-world entity is an important task in data integration and data cleaning. Duplicates may occur due to data entry errors or to the integration of data from disparate sources. Many techniques exist for detecting duplicate records [1, 7, 14, 16, 17, 19, 20, 21]. These techniques may be based on clustering, classification, or other link-analysis or statistical techniques. Duplicate detection is sometimes called *tuple matching* and is supported by commercial data integration tools such as *IBM WebSphere QualityStage*[1] and *FirstLogic Information Quality*[2].

[*]Partially supported by NSERC.
[1]http://www.ascential.com/products/qualitystage.html
[2]http://www.firstlogic.com/dataquality/

Even in carefully integrated databases where structural and semantic heterogeneity has been resolved, duplication may occur due to true disagreement among the sources. That is, two data sources may record different (inconsistent) information about the same entity. This is a common situation, for example, in the domain of Customer Relationship Management (CRM). One of the goals of CRM is to produce an integrated database from a variety of sources containing customer data. It is often the case that the sources contain conflicting information about the *same* customer. Commercial data integration tools typically use *conflict resolution rules* to support the *merging* of tuples corresponding to the same entity. Examples include rules that take the average between multiple conflicting incomes of the same customer. Such rules (sometimes called survivorship rules) may be hard to design, particularly for categorical data where values cannot always be combined. Furthermore, for some applications, there may be no set of rules that correctly resolve all current and future conflicts.

In the absence of complete conflict resolution rules, some data integration tools keep conflicting tuples in the integrated database. In this work, we consider the problem of query answering over dirty databases. Typically, existing data integration tools assume that queries can be executed directly on the dirty database. However, this fails to address the semantics of duplication. For example, suppose that a customer called John has two (inconsistent) incomes of \$80K and \$120K in the dirty database. Assume that we want to answer the query "get customers who make more than \$100K a year". Using standard query answering, John would appear in the answer. However, our intuition is that we do not know *with high certainty* whether John is in the answer because according to one of the sources his income may be \$80K.

The first question that we address here is: what is a good semantics for querying dirty databases? To this end, we note that duplicate tuples are alternative representations for the same real-world entity. In a clean database, only one representation will be present for each entity. In the absence of additional information (such as conflict resolution rules), we will assume that the representation in the clean database includes values from the duplicates. Hence, a dirty database represents several possible databases that are candidates to be in the clean database. If each real-world entity is assigned a unique key, we can think of the dirty database as an *inconsistent database* that violates a set of key constraints [5].

We adopt a probabilistic semantics for querying dirty databases. For this, it is necessary to associate probabilities to each of the tuples of the inconsistent database. There are different ways for assigning such probabilities. For example, we could assign probabilities to the sources: the more reliable the source, the higher its probability. Then, we could use provenance information to assign probabilities to the tuples in the integrated database taking their origin into account. In the absence of provenance information, we could just assume uniform probabilities. Finally, the probabilities may be the output of a tuple matching process performed by a data integration tool.

In this paper, we present a new semantics for querying duplicated data using a notion that we call *clean answers*. The semantics of clean answers is independent of the way the probabilities are produced, but is able to effectively exploit them during query answering. In particular, each answer is given together with its probability of being in the clean database.

loyaltyCard

	cardId	custFk	prob
t_1	111	c1	0.4
t_2	111	c2	0.6

customer

	custId	name	income	prob
s_1	c1	John	\$120K	0.9
s_2	c1	John	\$80K	0.1
s_3	c2	Mary	\$140K	0.4
s_4	c2	Marion	\$40K	0.6

Figure 1. A dirty customer database.

We now illustrate the semantics of clean answers through an example. In Figure 1, we show a dirty database with information regarding a customer loyalty program. There are two tables: `loyaltyCard`, which associates card numbers to customers; and `customer` which stores customers and their income. The attributes `cardId` and `custId` represent identifiers, where tuples sharing the same identifier have been determined to be duplicates. The attribute `custFk` of the `loyaltyCard` relation is a foreign key to the `customer` relation. Finally, each tuple has a probability (indicated in attribute `prob`) representing the likelihood of the tuple being in the clean database.

Suppose that we want to get the card numbers of customers who have an income above \$100K. Intuitively, card 111 should be in the answer because there is some evidence that it belongs to John, who has an income above \$100K with a very high probability. One might ask if it suffices to just clean the database off-line and then answer the query. For example, we may want to remove all the tuples for each duplicate except the one with the highest probability. In this case, we would remove t_1, s_2 and s_3. But then, the result of the query is empty because tuple t_2 (the only tuple for card 111) joins only with tuple s_4, representing Marion whose income is below \$100K.

In the semantics of clean answers, we consider all possible databases that can be obtained from the dirty database by choosing exactly one tuple for each duplicate. There are eight possible databases in this example. Each possible database can be assigned a probability. For example, if we choose tuple t_1 for card 111, and tuples s_1 and s_3 for John and Mary, respectively, we obtain a possible database D_1^{cd}. Since the tuples for each duplicate are chosen independently from the other duplicates, the probability of the possible database is obtained by multiplying the probabilities of its tuples. For D_1^{cd}, we get a probability of $0.4{*}0.9{*}0.4 = 0.144$. The clean answer is obtained by summing the probabilities of all possible databases that satisfy the query. In this case, card 111 is in the result of applying the query to four out of the eight possible databases: $D_1^{cd} = \{t_1, s_1, s_3\}, D_2^{cd} = \{t_1, s_1, s_4\}, D_3^{cd} = \{t_2, s_1, s_3\}, D_4^{cd} = \{t_2, s_2, s_3\}$. Their probabilities are $0.144, 0.216, 0.216$, and 0.024, respectively. Since the sum of these probabilities is 0.6, we will say that card 111 has 60% of probability of being associated with a customer earning over \$100K.pp

The semantics of clean answers is based on proposals for probabilistic databases [6, 10, 13, 18]. In probabilistic query answering, a set of answers is computed together with their probability of being correct. A distinctive feature of our approach is that it uses an SQL query rewriting, that is, given a query, we rewrite it into another SQL query that produces an answer together with the probability of the answer being in the clean database. While the semantics of probabilistic databases has been well explored, to the best of our knowledge, Dalvi and Suciu [13] are the first to consider query rewriting for probabilistic databases. They consider a semantics in which tuple probabilities are *independent*. For independent tuples, in any possible database, the probability of one tuple being in the clean database is independent of any other tuple. We cannot make this assumption for duplicated data. Recall that our possible worlds represent potential clean databases which contain exactly one tuple for each real-world entity. Hence, given a group of potential duplicates containing tuples t_1 and t_2, if t_1 is in a possible database, then the probability of t_2 being in that database is zero. Hence, t_1 and t_2 are not independent. We address this by proposing a new semantics under which tuples representing the same entity are conditionally dependent, and tuples representing different entities are independent.

This work is done in the context of the ConQuer project at the University of Toronto.[3] This project focuses on the efficient management of inconsistent and dirty databases.

In this paper, we make the following contributions.

• We propose a new semantics for querying duplicated data. This semantics complements and extends previous proposals

[3]ConQuer stands for *Consistent Querying*

in the areas of probabilistic databases and consistent query answering.

• We propose a query rewriting technique that, for a large class of select-project-join queries, rewrites the query into an SQL query that computes the answers to the query and their probabilities according to the semantics that we define for dirty databases. We show that although a rewriting cannot be obtained in general, the rewriting that we propose here works for many queries that arise in practice.

• We propose a method that uses the output of a tuple matching technique to assign probabilities to potential duplicates. Our method can be used with tuple matching techniques that only produce groupings of duplicates (with no additional information), or with techniques that output groupings together with a similarity measure or the distance between duplicates [4, 7, 17, 19, 20, 21]. Our method has the advantage that it performs well on categorical data values. It is beyond the scope of this paper to compare the relative advantages of different tuple matching techniques including ours. Deduplication is a well studied area and different approaches are known to work well for different applications and data characteristics. One of the benefits of our approach is that it is modular and can work with different techniques that find matching tuples.

• Using real data, we argue that the computed probabilities have an intuitive semantics and can be computed efficiently. We also show that our rewritten queries can be executed efficiently on large databases. Our studies show that the rewriting does not introduce a significant overhead in the execution time over the original query.

The paper is organized as follows. In Section 2, we define dirty databases and the semantics of clean answers. In Section 3, we present a rewriting that given a query, computes another query that retrieves the clean answers. In Section 4, we present a technique for assigning probabilities to duplicates in a dirty database. Section 5 presents the efficiency evaluation of our approach and, finally, Section 6 concludes the paper and discusses future challenges.

2 Dirty Databases and Clean Answers

We begin this section by discussing how we model dirty databases. Then, we present our semantics for query answering over such databases.

2.1 Dirty Databases

We assume that the dirty databases we use for query answering have been pre-processed by some data integration tool. The first pre-processing step is tuple matching. The output of a matching technique is a grouping or clustering of tuples. We will refer to a group of tuples that have been determined to correspond to the same real-world entity as a *cluster*.

Dfn 1 (clustering) *Let R be a relation. A* clustering *$\mathcal{C}$ partitions R into k disjoint sets of tuples $C_1, \ldots, C_k$, called* clusters, *such that $C_1 \cup C_2 \cup \ldots C_k = R$.*

Most tuple matching techniques apply to a single table; others use information from multiple tables to detect potential duplicates [7]. However, this distinction is immaterial to our discussion. We only require that the final output of the technique be a clustering on each dirty relation.

Matching techniques output cluster identifiers, though the way these identifiers are modeled may differ between tools. For example, some tools, like WebSphere QualityStage, output cross-reference tables that indicate which tuples are associated with which cluster. To use the clustering information in query answering, we will assume that each table has an identifier attribute containing the cluster identifer. This identifier attribute (again depending on the matching tool) may be added to the table as a new column. Alternatively, some matchers will choose one of the key values from the tuples in a cluster to be the identifier for the cluster. Each tuple in the cluster is then assigned this same key value. In this latter approach, the original key attribute of the table will contain the cluster identifier. In either case, the identifier attribute will contain duplicate values (and hence will not be a key of the relation).

Regardless of which approach is used by the tuple matching tool, the foreign keys of all relations must be updated to refer to the identifiers. This is a simple process, which we will call *identifier propagation*, that is supported by many matching tools.

Finally, we assume that each tuple is assigned a probability, in such a way that there is a probability function for each cluster. By definition of probability function, the sum of the probabilities of all tuples within a cluster must be 1. Clearly, a clean tuple (that is, a tuple with no other matching tuples) will have a probability of 1.

Dfn 2 (dirty database) *A* dirty database *is a database where for each relation R_i, we have a clustering $\mathcal{C}_i$ and a function $prob_i$ that maps the tuples of R_i to a probability value. The probabilities within each cluster must sum to 1.*

The following example illustrates how a database with duplicates can be modeled as a dirty database that we can use for query answering.

Example 1 *Consider the database of Figure 2. This database consists of two dirty tables,* `order` *and* `customer` *with original schema* `order[orderId,custFk,quantity]` *and* `customer[custId,name,balance]`*, respectively. We*

order	`id`	**orderId**	**custFk**	cIdFk	**quantity**	prob
t_1	o1	11	m1	c1	3	1
t_2	o2	12	m2	c1	2	0.5
t_3	o2	13	m3	c2	5	0.5

customer	`id`	**custId**	**name**	**balance**	prob
t_4	c1	m1	John	\$20K	0.7
t_5	c1	m2	John	\$30K	0.3
t_6	c2	m3	Mary	\$27K	0.2
t_7	c2	m4	Marion	\$5K	0.8

Figure 2. A dirty database with `order` and `customer` relations

have introduced two new attributes into the `customer` *relation,* `id` *and* `prob`*, for the identifier the tuple matcher produces and for the tuple probabilities. In relation* `order`*, we introduce these same two attributes, but in addition, we introduce a new attribute* `cIdFk` *for the identifier of the customer referenced by* `order.custFk`*. The values of attribute* `cIdFk` *are updated using identifier propagation.*

2.2 Clean Answers

It is well known how to answer queries over clean databases. But, how can we answer queries over a dirty database? And, what is the interpretation of the query results? One alternative is to return an answer together with its probability of being obtained from the clean database. In order to do this, one needs a clear indication of which databases may be clean. In our framework, this indication can be obtained from the semantics of duplication. In particular, since the clean database has no duplicates it is reasonable to assume that it consists of exactly one tuple from each cluster of the dirty database. The following is our definition of a *candidate database*.

Dfn 3 (candidate database) *Let D be a dirty database. We say that D^{cd} is a candidate database for D if (1) D^{cd} is a subset of D; and (2) for every cluster C_i of a relation in D, there is exactly one tuple* `t` *from C_i such that* `t` *is in D^{cd}.*

Candidate databases are related to the notion of *possible worlds*, which has been used to give semantics to probabilistic databases [13]. Notice, however, that the definition of candidate database imposes specific conditions on the tuple probabilities. First, the tuples within a cluster must be exclusive events (a very strong form of conditional dependence), in the sense that exactly one tuple of each cluster appears in the clean database. Second, the probabilities of tuples from different clusters are independent. Cavallo and Pittarelli [10] proposed a model in which all the tuples in a relation are assumed to be exclusive events. In our terms, they assume that each relation consists of exactly one cluster. At the other extreme, most work on probabilistic databases assumes that tuple probabilities are independent [15, 6, 13]. To the best of our knowledge, the only work that considers conditional probabilities in a way that satisfies our conditions is ProbView [18]. However, their work does not focus on producing a rewritten query (i.e., a query that computes the answer according to their semantics), and therefore cannot be used for our purposes.

Example 2 *Consider the dirty* `customer` *and* `order` *relations from Figure 2. There are eight candidate databases:*

$$D_1^{cd} = \{t_1, t_2, t_4, t_6\} \quad D_2^{cd} = \{t_1, t_2, t_4, t_7\}$$
$$D_3^{cd} = \{t_1, t_2, t_5, t_6\} \quad D_4^{cd} = \{t_1, t_2, t_5, t_7\}$$
$$D_5^{cd} = \{t_1, t_3, t_4, t_6\} \quad D_6^{cd} = \{t_1, t_3, t_4, t_7\}$$
$$D_7^{cd} = \{t_1, t_3, t_5, t_6\} \quad D_8^{cd} = \{t_1, t_3, t_5, t_7\}$$

Clearly, not all candidate databases are equally likely to be clean. We model this with a probability distribution, which assigns to each candidate database a probability of being clean. Since the number of candidate databases may be huge (exponential in the worst case), we do not give the distribution by extension. Instead, we use the probabilities of each tuple (function $prob$) to calculate the probability of a candidate database being the clean database. Since tuples are chosen independently from each cluster, the probability of each candidate database can be obtained as the product of the probability of each of its tuples.

Dfn 4 (probability distribution over candidates) *Let D be a dirty database. The probability of a candidate database D^{cd} is defined as: $Pr(D^{cd}) = \prod_{\mathtt{t} \in D^{cd}} prob(\mathtt{t})$*

Notice that, in contrast to the definitions given for cases where tuple independence is assumed [13], we do not need to consider tuples that are *not* in the candidate database. The reason is that, since every candidate has exactly one tuple from each cluster, the probability of such a tuple not being in the candidate is 1.

Example 3 *The following are the probabilities for each of the candidate databases for the dirty database of Figure 2.*

D_1^{cd}	$0.5 * 0.7 * 0.2 = 0.07$	D_2^{cd}	$0.5 * 0.7 * 0.8 = 0.28$
D_3^{cd}	$0.5 * 0.3 * 0.2 = 0.03$	D_4^{cd}	$0.5 * 0.3 * 0.8 = 0.12$
D_5^{cd}	$0.5 * 0.7 * 0.2 = 0.07$	D_6^{cd}	$0.5 * 0.7 * 0.8 = 0.28$
D_7^{cd}	$0.5 * 0.3 * 0.2 = 0.03$	D_8^{cd}	$0.5 * 0.3 * 0.8 = 0.12$

We are now ready to give the semantics of query answering in our framework. Recall that the clean database is unknown. However, we can still reason about the answers obtained by applying the query to the candidate databases. Intuitively, a result is more likely to be in the answer if it is obtained from candidates with higher probability of being clean. Therefore, for each tuple of the query result, we will consider the candidates from which it can be obtained, and sum their probabilities.

Dfn 5 (clean answer) *Let D be a dirty database. We say that a tuple $\mathbf{t}$ is a* clean answer *to a query q if there exists a candidate database D^{cd} such that $\mathbf{t} \in q(D^{cd})$. The probability of $\mathbf{t}$ is: $p = \sum_{D^{cd} |\ \mathbf{t} \in q(D^{cd})} Pr(D^{cd})$*

Example 4 *Consider a query q_1 that retrieves all the customers whose balance is greater than $10K.*

```
select id
from customer c                (q1)
where balance > $10K
```

Customer $c1$ has a balance greater than $10K in every candidate. Thus, it is a clean answer with probability 1. Customer $c2$ satisfies the query only in candidates D_1^{cd} and D_3^{cd}, D_5^{cd} and D_7^{cd}. Therefore, $c2$ is a clean answer with a probability of .20.

The notion of clean answers is a generalization of the *consistent answers* that were defined by Arenas et al. [5] in the context of query answering over inconsistent databases. In particular, if each tuple of the dirty database is assigned a non-zero probability of being in the clean database, then the consistent answers of a query correspond to the clean answers that have a probability of 1 (that is, complete certainty).

3 Efficient Computation of Clean Answers

The clean answers to a query can be obtained directly from the definition if we assume that all the candidate databases of a given dirty database are available. But this is an unrealistic assumption because, in the worst case, the number of candidate databases may be exponential in the size of the dirty database. Since our goal is the efficient computation of clean answers, we present techniques that avoid the materialization of the candidate databases altogether. In particular, we use an approach based on query rewriting. That is, given a query q, we produce another query Q^{rew} that can be applied *directly* on any dirty database in order to obtain the clean answers and their probabilities for q.

There are some queries for which there is no SQL rewriting that computes the clean answers. This follows from existing results in the literature for the problem of obtaining consistent answers which, as we explained in the previous section, is a special case of clean answers. In particular, it has been shown that there are some Select-Project-Join (SPJ) queries for which the problem of obtaining consistent answers is co-NP complete [9, 11]. This implies that, unless P=NP, there are some SPJ queries for which there is no SQL rewriting that computes the clean answers. Thus, the question is: are there large and practical classes of queries for which the clean answers can be efficiently computed? We will show in this section that the answer to this question is positive. More importantly, we will show that, for the queries in this class, there is a simple query rewriting that computes the clean answers directly from the dirty database.

D_1^{cd}	$(o1,c1),(o2,c1),(c1,\$20K),(c2,\$27K)$	0.07
D_2^{cd}	$(o1,c1),(o2,c1),(c1,\$20K),(c2,\$5K)$	0.28
D_3^{cd}	$(o1,c1),(o2,c1),(c1,\$30K),(c2,\$27K)$	0.03
D_4^{cd}	$(o1,c1),(o2,c1),(c1,\$30K),(c2,\$5K)$	0.12
D_5^{cd}	$(o1,c1),(o2,c2),(c1,\$20K),(c2,\$27K)$	0.07
D_6^{cd}	$(o1,c1),(o2,c2),(c1,\$20K),(c2,\$5K)$	0.28
D_7^{cd}	$(o1,c1),(o2,c2),(c1,\$30K),(c2,\$27K)$	0.03
D_8^{cd}	$(o1,c1),(o2,c2),(c1,\$30K),(c2,\$5K)$	0.12

Figure 3. Candidates for database of Figure 2

We start with a number of motivating examples before introducing the rewriting algorithm.

Example 5 *Consider again the query q_1 that retrieves the customers with a balance greater than $10K. We showed in Example 4 of the previous section that the clean answer for this query on the dirty database of Figure 2 is $\{(c1, 1), (c2, 0.2)\}$.*

Consider a naïve rewriting that returns the probability associated with each tuple.

```
select id, prob
from customer c
where balance > $10K
```

The result of applying this rewritten query to the dirty database is $\{(c1, 0.7), (c1, 0.3), (c2, 0.2)\}$, which is not the clean answer. However, we can obtain the clean answer answer by grouping the two tuples for $c1$ and summing their probabilities. Therefore, the following query returns the clean answers for q_1.

```
select id, sum(prob)
from customer c
where balance > $10K
group by id
```

The previous example focuses on a query with just one relation but, as we show in the next example, the rewriting strategy can be extended to queries involving foreign key joins.

Example 6 *Consider the dirty database of Figure 2. As an aid to follow the next examples, in Figure 3 we repeat the candidate databases given in Example 2. Instead of showing the tuple numbers, in the figure we show the attributes that are relevant to the queries (specifically,* `id` *and* `cIdFk` *of the* `order` *relation, and* `id` *and* `balance` *of* `customer`*).*

Consider a query q_2 that selects the orders and their customers, for those customers whose balance is greater than $10K.

```
select o.id, c.id
from order o, customer c              (q2)
where o.cIdFk=c.id
      and c.balance > $10K
```

The order tuple t_1, $(o1, c1)$, appears in every candidate, and the balance of both $c1$ customer tuples is always greater than $10K. Therefore, $(o1, c1)$ has probability 1 of being a *clean answer. The query answer $(o2, c1)$ is an answer in the candidates D_1^{cd}, D_2^{cd}, D_3^{cd} and D_4^{cd} and therefore has a probability of .50 of being in the clean answer. Finally, $(o2, c2)$ is in the answer obtained from D_5^{cd} and D_7^{cd}, since $c2$ has a balance greater than $10K in these databases, and has a probability of .10 of being in the clean answer.*

If we apply the query to the dirty database and compute the probabilities as `c.prob*o.prob` *we get the following.*

o.id	*c*.id	*prob*	*join of*
o1	*c1*	*0.7*	*(o1,c1),(c1,$20K)*
o1	*c1*	*0.3*	*(o1,c1),(c1,$30K)*
o2	*c1*	*0.35*	*(o2,c1),(c1,$20K)*
o2	*c1*	*0.15*	*(o2,c1),(c1,$30K)*
o2	*c2*	*0.1*	*(o2,c2),(c2,$27K)*

It is easy to see that the clean answers can be obtained by grouping on both the order and customer identifiers, and summing the probabilities of each group as follows.

```
select o.id, c.id, sum(o.prob * c.prob)
from order o, customer c
where o.cIdFk=c.id
      and c.balance> $10K
group by o.id, c.id
```

The intuition underlying the rewriting is that we can sum probabilities because, within each group, every pair of tuples comes from disjoint *sets of candidates. For example, the first tuple in the group of $(o1, c1)$ is obtained from the join of $(o1, c1)$ and $(c1, \$20K)$. These tuples occur together in candidates D_1^{cd}, D_2^{cd}, D_5^{cd} and D_6^{cd}. The second tuple of the group is obtained from the join of $(o1, c1)$ and $(c1, \$30K)$, which appear together in candidates D_3^{cd}, D_4^{cd}, D_7^{cd} and D_8^{cd}. The fact that they come from disjoint sets of candidates is crucial. Otherwise, the strategy of summing the probabilities within the group would fail since some of the candidates would be counted more than once.*

The previous examples use a simple strategy of grouping and summing to produce the rewriting. However, as we show in the next example, this strategy may fail for some queries.

Example 7 *Consider a query q_3 that retrieves all the customers whose balance is greater than $25K and who have placed orders for less than 5 items.*

```
select c.id
from order o, customer c
where o.quantity < 5 and o.cIdFk= c.id
      and c.balance > $25K
```

Notice that, unlike the queries of the previous examples, the attribute `id` *of* `order` *is not in the* `select` *clause of the query. We are going to show in this example that for the dirty database of Figure 2, the strategy of grouping and summing fails to produce the clean answers for q_3.*

The candidate databases for the dirty database are given in Figure 3. The customer $c1$ appears in the result of applying q_3 to D_3^{cd}, D_4^{cd}, D_7^{cd} and D_8^{cd}. Therefore, the probability of $c1$ of being in the clean database is 0.3. The customer $c2$ appears in the relation `order` *only in tuple t_3, which corresponds to an order that violates the query (the value for* `quantity` *is 5). Therefore, the probability of $c2$ of being in the clean database is zero.*

Consider now the rewritten query that would be produced by grouping and summing, as we did in the previous examples.

```
select c.id, sum(o.prob*c.prob)
from order o, customer c
where o.quantity < 5 and o.cIdFk= c.id
      and c.balance > $25K
group by c.id
```

In this case, the rewritten query does not compute the clean answers. In fact, the result of the rewritten query is $\{(c1, 0.45), (c2, 0)\}$. The value $(c1, 0.45)$ is obtained as the result of grouping and summing the following:

c.key	*prob*	*join of*
c1	*0.3*	*(o1,c1,3),(c1,$30K)*
c1	*0.15*	*(o2,c1,2),(c1,$30K)*

To understand why the rewritten query fails to compute the clean answers, consider that the join between $(o1, c1, 3)$ and $(c1, \$30K)$ occurs in candidates D_3^{cd}, D_4^{cd}, D_7^{cd} and D_8^{cd}. The join between $(o2, c1, 2)$ and $(c1, \$30K)$ occurs in candidates D_3^{cd} and D_4^{cd}. Therefore, the rewritten query is incorrectly accounting for the probabilities of D_3^{cd} and D_4^{cd} twice.

The previous example shows that there are some queries for which our rewriting may fail. However, we are now going to show that this simple rewriting does work for a large and practical class of queries. We call it the *class of rewritable queries*. Before introducing this class, we given an algorithm **RewriteClean**(q) whose output is shown in Figure 4. Let q be an SPJ query (with only equality joins) where $R_1, \ldots, R_m$ are the relations in the `from` clause, and $A_1, \ldots, A_n$ are the attributes in the `select` clause. The rewritten query produced by **RewriteClean**(q) groups the result of q by the attributes that appear in the `select` clause of q. For each group, it sums the product of the probabilities of the tuples satisfying the conditions of the query.

As we already mentioned, it is known in the literature that there are some queries for which there is no SQL rewriting. We argue, however that the hard queries given in the literature do not commonly arise in practice. For example, they have joins that do not involve an identifier attribute or key (e.g., joins between two foreign keys), they are cyclic, or contain self joins. Since such conditions are rare in practice, we will not allow them in the class of rewritable queries. Our query class relies on the notion of the *join graph* of a query.

Dfn 6 (join graph) *Let q be an SPJ query. The* join graph *G* of *q is a directed graph such that:*

```
RewriteClean(q)
Given an SPJ query q of the form:
select A_1,...,A_n
from  R_1,...,R_m
where  W
Output a query Q^{rew} of the form:
select A_1,...,A_n,sum(R_1 .prob.*...* R_m.prob)
from  R_1,...,R_m
where  W
group by  A_1,...,A_n
```

Figure 4. Query rewriting

• *the vertices of G are the relations used in q; and*

• *there is an arc from R_i to R_j if a non-identifier attribute of R_i is equated with the identifier attribute of R_j.*

We now define the class of rewritable queries.

Dfn 7 (rewritable query) *Let q be an SPJ query, and G be the join graph of q. We say that q is* rewritable *if:*

1. *all the joins involve the identifier of at least one relation,*
2. *G is a tree,*
3. *a relation appears in the* `from` *clause at most once,*
4. *the identifier of the relation at the root of G appears in the* `select` *clause.*

The first condition rules out joins on two non-identifier attributes. However, the definition does allow joins between identifiers (which correspond to the keys of the dirty relations), or between a non-identifier and an identifier (for example, a foreign key join). The second condition states that the join graph of the query must be a tree. The third condition restricts each relation of the schema to appear at most once in the `from` clause. This rules out self-joins, for example, but places no restriction on the number of relations that can appear in the query. The last condition is related to the tree structure of the join graph. In particular, we impose that the identifier of the relation at the root of the tree must appear in the `from` clause. Notice that the query of Example 7 violates this condition because it does not project on the `id` attribute of `order`, the relation at the root the tree. For the data cleaning tasks we wish to support using our rewriting, including the identifier in the `select` clause is not an onerous restriction. Our rewritings are designed not to support general analysis queries, but rather to help a user in understanding the entities in the dirty database.

The next theorem is the main result in this section, and states the correctness of $\mathbf{RewriteClean}(q)$ for the class of rewritable queries. We give the proof of the theorem in the full version of this paper [2].

Theorem 1 *Let q be a rewritable query. Then,* **RewriteClean**(q) *computes the clean answers for q on every dirty database.*

```
Input : A set of tuples $\mathbf{T}$,
  - a clustering $\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$ of $\mathbf{T}$,
    where $c_i$ is the identifier of cluster $i$
  - a distance measure $d$.
Output : For every tuple $\mathbf{t}$ in $\mathbf{T}$, a probability $prob(\mathbf{t})$.
Main Procedure :
 - (Step 1) For $i = 1 \ldots k$:
   * compute cluster representative $rep_i$ for $c_i$
     by merging all the tuples that belong to it.
   * initialize sum of distances for $c_i$, $S(c_i) = 0$.
   - (Step 2) For each tuple $\mathbf{t} \in \mathbf{T}$ that belongs to $c_i$:
     * compute $d_{\mathbf{t}} = d(\mathbf{t}, rep_i)$, the distance of $\mathbf{t}$
         to the representative of its cluster.
     * Add $d_{\mathbf{t}}$ to $S(c_i)$.
   - (Step 3) For each tuple $\mathbf{t} \in \mathbf{T}$ that belongs to $c_i$:
     * compute similarity $s_{\mathbf{t}} = 1 - \frac{d_{\mathbf{t}}}{S(c_i)}$.
     * $prob(\mathbf{t}) = 1.0$ if $|c_i| = 1$, or
       $prob(\mathbf{t}) = \frac{s_{\mathbf{t}}}{|c_i|-1}$ otherwise.
```

Figure 5. Assigning Tuple Probabilities

4 Computing tuple probabilities

We have been assuming that the results from some tuple matching method are given. Even if such methods produce a clustering of the database, they usually do not produce a probability (or some kind of quantitative characterization) indicating how likely a potential duplicate is to be in the clean database. In this section, we present an approach that assigns such probabilities. Given a clustering, we compute a representative of each cluster. Then, we assign each tuple a probability which is based on the distance of this tuple from its cluster representative.

We give a generic procedure to compute the probabilities in Figure 5. In this procedure, we make use of a given clustering $\mathcal{C}$ and a distance measure d. The distance measure is used to assign a probability to each cluster. The actual measure we use in our work will be given in Section 4.1.3. The first step of the algorithm computes the cluster representatives and initializes the sum of distances S in each cluster, which will be used as a normalization factor to compute the probabilities. Notice that this is a general procedure and any clustering method as well as distance measure can be employed. For each tuple $\mathbf{t}$ within a particular cluster with identifier c_i, Step 2 calculates the distance of this tuple to the representative, rep_i, of its cluster and adds this distance to $S(c_i)$. Finally, in order to compute the probability of a tuple $\mathbf{t}$, Step 3 turns the distance computed in the previous step for this tuple into a similarity. Intuitively, if this similarity is small, then the tuple under consideration is almost as good as the representative and will eventually acquire a high probability.

At this point, we would like to emphasize the role of the cluster representatives. In a semi-automatic process of assigning probabilities, a person's intuition guides the ranking of a tuple as more (or less) probable of being in the clean database. In our fully automatic approach, the representatives contain the most common features that appear in the cluster. Thus, the quantification of similarity between the tuples of a cluster and its representative gives us a hint of which tuple is more likely to act as a representative by itself. Overall, the higher the similarity, the higher the probability of a tuple appearing in the clean database. Finally, the similarity of each tuple computed by the algorithm in Figure 5 is normalized to fall in the interval $[0.0, 1.0]$ and the resulting value is the probability of a tuple being in the clean database (denoted as $prob(\mathbf{t})$). Of course, if a cluster consists of a single tuple (i.e., $|c_i| = 1$), the probability assigned to the tuple in this cluster is equal to one, as we are certain about its existence in the clean database.

It is important to find a representative and measure of distance that reflect the semantics of clean answers. To illustrate our approach, consider the `customer` relation of Figure 6, where c_1, c_2 and c_3 are the cluster identifiers for the three clusters that exist in this relation. To do query answering over such a dirty database, we need a way of understanding how likely each tuple is to be in the clean database. Intuitively, in cluster c_1, t_2 seems to be the most likely tuple since it shares all of its values with at least one other tuple in the group. This intuition is based on our query answering semantics. For cluster c_1, 'Mary' is the most likely name value, 'banking' the most likely `mktsegment`, 'USA' is the certain `nation` value, and so on. Our method formalizes this intuition.

	name	mktsegmt	nation	address	
t_1	Mary	building	USA	Jones Ave	c_1
t_2	Mary	banking	USA	Jones Ave	c_1
t_3	Marion	banking	USA	Jones ave	c_1
t_4	John	building	America	Arrow	c_2
t_5	John S.	building	USA	Arrow	c_2
t_6	John	banking	Canada	Baldwin	c_3

Figure 6. A dirty customer relation.

4.1 Dealing with Categorical Data

When we compare tuples, it is often assumed that there exists some well-defined notion of *similarity*, or *distance*, between them. When the objects are defined by a set of numerical attributes, there are natural definitions of distance based on geometric analogies. These definitions rely on the semantics of the data values. For example, values \$10,000 and \$9,500 are more similar than \$10,000 and \$1.

In many domains, data records are composed of a set of descriptive attributes, many of which are neither numerical nor inherently ordered in any way. For example, in the setting of our `customer` relation, it is not immediately obvious what the distance, or similarity, is between the values "banking" and "building", or the tuples of "John S." and "Mary". The values without an inherent distance measure defined among them are called *categorical*. We will deal here with the problem of calculating the distance among tuples defined over categorical values. We will employ a distance measure that has been used in the clustering of large data sets with categorical values [4], based on *information loss*. This distance measure provides a natural way of quantifying the duplication of values within tuples and it has been used to detect redundancy in dirty legacy databases [3]. In the following subsections, we explain the data representation and the actual computation of the distance.

Note that when a distance measure between tuples (e.g., string edit distance) is available, our method can incorporate it. However, since such measures are well studied in the deduplication and approximate matching literature, we do not investigate them further here.

4.1.1 Data representation

We now introduce some conventions and notations that we will use in the rest of the section. We will assume that we have a set $\mathbf{T}$ of n tuples. The tuples are defined over attributes $A_1, A_2, \ldots, A_m$. We assume that the domain $\mathbf{V}_i$ of each attribute A_i is named in such a way that identical values from different attributes are treated as distinct values. For each attribute A_i, a tuple $\mathbf{t} \in \mathbf{T}$ takes exactly one value from the set $\mathbf{V}_i$. Let $\mathbf{V} = \mathbf{V}_1 \cup \cdots \cup \mathbf{V}_m$ denote the set of all possible attribute values. Let $|\mathbf{V}|$ denote the size of $\mathbf{V}$. We represent the data as an $n \times |\mathbf{V}|$ matrix M. In this matrix, the value of $M[\mathbf{t}, v]$ is 1, if tuple $\mathbf{t} \in \mathbf{T}$ contains attribute value $v \in \mathbf{V}$; and zero otherwise. Each tuple contains one value for each attribute, so each tuple vector contains exactly m 1's.

Now, let T and V be random variables that range over $\mathbf{T}$ (the set of tuples) and $\mathbf{V}$ (the set of attribute values), respectively. We normalize matrix M so that the entries of each row sum up to 1. For a tuple $\mathbf{t} \in \mathbf{T}$, the corresponding row of the normalized matrix holds the conditional probability distribution $p(V|\mathbf{t})$. Since each tuple contains exactly m attribute values, for some $v \in \mathbf{V}$, $p(v|\mathbf{t}) = 1/m$ if v appears in tuple $\mathbf{t}$, and zero otherwise.

Example 8 *Table 1 contains the normalized matrix M for the* `customer` *relation. In this matrix, given tuple t_1, we have a probability of 0.25 of choosing one of the four values that appear in it.*

4.1.2 Tuple and Cluster Summaries

Our task now is to build the cluster representatives, against which each tuple of a cluster will be compared. We summarize a cluster of tuples in a *Distributional Cluster Feature (DCF)* [4]. We will use the information in the relevant DCFs to compute the distance between a cluster summary (representative) and a tuple.

Let $\mathbf{T}$ denote a set of tuples over a set $\mathbf{V}$ of attributes, and let T and V be the corresponding random variables, as

	Mary	Marion	John	John S.	banking	building	USA	Canada	America	Jones Ave	Jones ave	Arrow	Baldwin	Cluster Id
t_1	0.25	0	0	0	0	0.25	0.25	0	0	0.25	0	0	0	c1
t_2	0.25	0	0	0	0.25	0	0.25	0	0	0.25	0	0	0	c1
t_3	0	0.25	0	0	0.25	0	0.25	0	0	0	0.25	0	0	c1
t_4	0	0	0.25	0	0	0.25	0	0	0.25	0	0	0.25	0	c2
t_5	0	0	0	0.25	0	0.25	0.25	0	0	0	0	0.25	0	c2
t_6	0	0	0.25	0	0.25	0	0	0.25	0	0	0	0	0.25	c3

Table 1. The normalized `customer` matrix

described earlier. Also let $\mathbf{C}$ denote a clustering of the tuples in $\mathbf{T}$ and let C be the corresponding random variable. The *Distributional Cluster Feature* (DCF) of a cluster with identifier c_i is defined by the pair

$$DCF(c_i) = \Big(|c_i|, p(V|c_i)\Big)$$

where $|c_i|$ is the cardinality of cluster c_i, and $p(V|c_i)$ is the conditional probability distribution of the attribute values given the cluster c_i. Practically, the cluster representative of rep_i of cluster c_i will be its DCF, $DCF(c_i)$.

If c_i consists of a single tuple, then $|c_i| = 1$, and $p(V|c_i)$ is computed as described in the previous subsection. For larger clusters, the DCF is computed recursively as follows. Let c^* denote the cluster we obtain by merging two clusters with identifiers c_1 and c_2. The DCF of the cluster c^* is equal to $DCF(c^*) = \Big(|c^*|), p(V|c^*)\Big)$, where $|c^*|$ and $p(V|c^*)$ are computed using the following equations:

$$\begin{aligned} |c^*| &= |c_1| + |c_2| \\ p(V|c^*) &= \tfrac{|c_1|}{|c_1|+|c_2|}p(V|c_1) + \tfrac{|c_2|}{|c_1|+|c_2|}p(V|c_2) \end{aligned}$$

Intuitively, when computing a new cluster representative, its cardinality becomes the sum of the cardinalities of the merged sub-clusters, while the conditional probability of its values is the weighted average of the conditional probability distribution of the values from the merged sub-clusters. Note that a cluster representative may not necessarily belong to the initial set of tuples. Table 2 depicts the cluster representative for the `customer` relation as dictated by the cluster identifiers given in the last column of Table 1. In this table, we see that rep_1 summarizes three tuples that contain the value USA since the probability of this value in this cluster remains the same as in the initial tuples. Similarly, both tuples t_4 and t_5 contain the values `building` and `Arrow`, which is reflected in their representative, rep_2. Finally, rep_3 contains the last tuple of the relation, t_6.

4.1.3 Distance as Information Loss

Our notion of distance d between tuples and summaries that include categorical values, is based on an intuitive calculation of the loss of information when two distributions are merged. Information here is defined in terms of Information Theory [12]. More precisely, we use the *mutual information* $I(X;Y)$ of two random variables X and Y, to quantify the information that variable X contains about Y and vice versa. To compute the value of $I(X;Y)$, we need the probability distributions of X and Y. In our case, we quantify the information that the clusters in C contain about the values in V and use the distributions that describe the tuples and clusters stored in the corresponding DCFs. The main objective of the quantification of the loss of information is to realize which tuples share as many common values as possible with their cluster representative. Thus, given the DCFs of summaries s_1 and s_2, the information loss (and hence the distance) between s_1 and s_2 is given by the following expression:

$$d(s_1, s_2) = I(C;V) - I(C';V)$$

where C' denotes the clustering after merging the summaries s_1 and s_2.

Example 9 *Table 3 shows the distance between each tuple of the* `customer` *relation (3rd column) and its corresponding representative (2nd column). The same table contains the calculation of the similarity (4th column) and the final probability of each tuple being in the clean database (5th column). Notice that the smaller the distance of a tuple to each representative, the higher the similarity and, consequently, the higher the probability that it belongs to the clean database.*

	rep	$d(\mathbf{t}, rep)$	$s_{\mathbf{t}}$	$p(\mathbf{t})$
t_1	rep_1	0.093	0.665	0.332
t_2	rep_1	0.061	0.781	0.391
t_3	rep_1	0.124	0.554	0.277
t_4	rep_2	0.063	0.500	0.500
t_5	rep_2	0.063	0.500	0.500
t_6	rep_3	0.000	1.000	1.000

Table 3. Probability calculation in `customer`

There are several observations we can make from Table 3. Concerning the tuples of cluster c_1, tuple t_2 is the most probable one to be in the clean database, which agrees with our intuition. This tuple contains the values with the highest frequency in this cluster, and thus it is the one that can replace its cluster representative in the best way. On the other hand, c_2 contains just two tuples, which are equally likely to be in the clean database and, finally, we have no uncertainty in having t_6 in the clean database since it constitutes a cluster summary of its own.

In the next subsection, we present a brief discussion on the effectiveness of using our method for computing tuple probabilities (a more detailed discussion can be found in the extended version of this paper [2]).

	Mary	Marion	John	John S.	banking	building	USA	Canada	America	Jones Ave	Jones ave	Arrow	Baldwin	$\vert c\vert$
rep_1	0.167	0.083	0	0	0.167	0.083	0.25	0	0	0.167	0.083	0	0	3
rep_2	0	0	0.125	0.125	0	0.25	0.125	0	0.125	0	0	0.25	0	2
rep_3	0	0	0.25	0	0.25	0	0	0.25	0	0	0	0	0.25	1

Table 2. The three cluster representatives for `customer` tuples

4.2 Qualitative Evaluation

Our technique for assigning tuple probabilities is performed on each relation of a database separately. Hence, we use a single relation to evaluate the technique and investigate whether the assigned probabilities agree with human intuition. In our experiments [2], we used clusters from the **Cora** data set [19]. This data set contains computer science research papers integrated from several sources. It has been used in other data cleaning projects [19, 8] and we take advantage of previous labelings of the tuples into clusters.

As an example, we considered a cluster that corresponds to a publication by Robert E. Shapire. It contains 56 tuples and due to space considerations, the full cluster is not given here (but can be found in our technical report [2]). In Table 4, we give the most frequent values in all attributes of this cluster, as well as the two most likely and the two least likely tuples as they are ranked by the assignment of probabilities.

Table 4 re-confirms the intuitive ranking of the tuples within a cluster. In particular, the most likely tuple shares all its values with the set of most frequent values, while the next most likely tuple shares all but one of these values (the value of the `volume` attribute). On the other hand, the second least likely tuple (the penultimate tuple in the table) corresponds to a different publication, and thus it should have been placed in a different cluster. Finally, although the least likely tuple (the last tuple in the table) corresponds to the same publication of Shapire, its values are stored in a different way than used in the rest of the tuples for this publication.

5 Experimental Evaluation

In this section, we evaluate the efficiency of our approach. All the experiments were performed on a machine with 2.8 Ghz Intel Pentium 4 CPU and 1GB 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.

5.1 Data Set Generation

In order to assess the efficiency of our techniques on very large data sets, we used a synthetic data generator, the **UIS Database Generator.**[4] This generator was written by Mauricio Hernández and has been used for the evaluation of duplicate detection [17]. Given a parameter that controls the number of tuples and a parameter that controls the number of clusters, the generator creates clusters of potential duplicates. We use this generator to produce data that conforms to the schema of the TPC-H specification (`http://www.tpc.org/tpch`).

[4]A copy of the generator can be found at: `http://www.cs.utexas.edu/users/ml/riddle/data.html`

5.2 Parameter Setting

The parameters for the UIS generator that we set in our experiments are presented below.

Scaling Factor (sf). The scaling factor is used to control the size of the relations created for the TPC-H specification. If $sf = 1$, we create a data set of size 1GB (approximately 8 million tuples), if $sf = 2$ the size is 2GB (approximately 16 million tuples), etc.

Inconsistency Factor (if). The UIS generator creates a number of clusters, where each cluster contains on average the same number of tuples. The cluster cardinalities are drawn from a uniform distribution, whose mean is the value of if. More precisely, if x is the value of if, the generator creates cluster cardinalities between 1 and $2x - 1$. As the value of if increases, the degree of inconsistency increases.

In our experiments, we create data tables for the TPC-H specification that are of size 0.1GB, 0.5GB, 1GB and 2GB, and clusters with 1, 2, 3, 4, 5 and 25 tuples per cluster.

5.3 Efficiency Evaluation

We first evaluate the time required to annotate the database with probabilities. Then, we study the performance of our method for producing clean query answers.

Probability Computation We used the technique of Section 4 to produce the probabilities. For the propagation of identifiers, we used the approach that replaces the values of the original keys of the relations with the identifier selected by the tuple matching tool. We experimented with data sets of size 1GB ($sf = 1$), and values 1, 5 and 25 for the if parameter.[5] The total execution time for the propagation and probability calculation was, for all databases, less than 30 minutes. This is a reasonable time for an off-line technique.

The propagation technique is independent of the number of tuples in each cluster and is only sensitive to the total size of the relations. However, the time to compute probabilities increases as the number of tuples in each cluster increases. This happens since more tuples are used in the computation of the cluster representatives. We experimentally validate this fact with the results in Figure 7. This figure depicts the time taken to propagate and compute the probabilities for the largest relation of our database, the `lineItem` relation,

[5]Notice that the value $if = 1$ corresponds to a completely clean database, and hence the computation of probabilities should include the addition of a probability 1.0 to each tuple. However, since this is an off-line procedure, we decided to keep an un-optimized version of our method for $if = 1$, in order to get a baseline reference.

Most frequent values					
author	**title**	**venue**	**volume**	**year**	**pages**
robert e. schapire	the strength of weak learnability	machine learning	5(2)	1990	197-227
Top-2 Tuples					
author	**title**	**venue**	**volume**	**year**	**pages**
robert e. schapire	the strength of weak learnability	machine learning	5(2)	1990	197-227
robert e. schapire	the strength of weak learnability	machine learning	5	1990	197-227
Bottom-2 Tuples					
author	**title**	**venue**	**volume**	**year**	**pages**
r. schapire	on the strength of weak learnability	proc of the 30th i.e.e.e. symposium...	NULL	1989	pp. 28-33
schapire, r.e.,	'the strength of weak learnability'	machine learning	5 2	(1990)	pp. 197-227

Table 4. Example from the Cora data set

as well as the time required to perform one linear scan over this relation. The running time for the computation of prob-

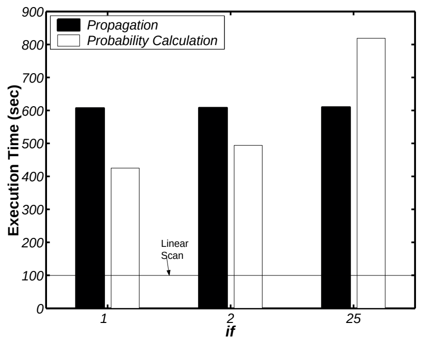

Figure 7. Offline times for `lineItem`

abilities grows as the number of tuples increases within each cluster. When the value of if is small, the distance calculation of each tuple to its representative dominates the time, while with larger if values, it is the time of the creation of representatives that prevails. Hence, when we move from $if = 1$ to $if = 2$, the difference in computation time for the probabilities is mainly due to the merging of more tuples in cluster representatives. This difference is more pronounced as we move from $if = 2$ to $if = 25$.

Clean Query Answering We performed our experiments on thirteen queries from the TPC-H specification, which contain from one to six joins. In particular, we focused on queries 1, 2, 3, 4, 6, 9, 10, 11, 12, 14, 17, 18, and 20 from the specification. The only change that we made to the queries was removing the aggregate expressions. The queries in the TPC-H specification are parameterized. For all queries, we used the parameters suggested by the standard for query validation. The thirteen queries used in the experiments appear in the full version of this paper [2]. For each instance, we created indices on the identifier, and ran the DB2 RUNSTATS command on all attributes.

In Figure 8, we show the running times of the thirteen queries on a 1 GB database with an average cluster size of 3 tuples. Notice that the overhead of running the rewritten queries is not significant. All queries (except Query 9) execute within 1.5 times the time of the original query. Notably, the rewritten versions of eight queries take less than 1.05 times the execution time of the original query. These are Queries 2, 4, 6, 11, 14, 17, 18, and 20.

Note that the running times for Query 9 and its rewriting extend well beyond the scale of this graph (the times are indicated in numbers at the top of the figure). This query has six joins and a high selectivity (a large fraction of tuples satisfying its conditions). With a large number of duplicates satisfying the query, the grouping and aggregation of the rewritten query becomes costly. As a result, this rewriting has the highest overhead (1.8 times the time of the original query).

In Figure 9, we show the effect of cluster size on the running time. These running times correspond to Query 3, which we show next. This query contains a three-way join, three selection conditions, and an `order by` clause.

```
select l_orderkey,
  l_extendedprice*(1-l_discount) as revenue,
  o_orderdate, o_shippriority
from customer,orders,lineitem
where c_mktsegment = 'BUILDING'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < '1995-03-15'
  and l_shipdate > '1995-03-15'
order by revenue desc, o_orderdate
```

We report results on 1GB instances with if =1, 2, 3, 4 and 5. The solid lines correspond to the running times of Query 3 and its rewriting. The running time of both queries increases with the average size of the clusters. The reason for this behavior is that the size of the result set for both queries increases with the number of tuples per cluster because a tuple may join with more tuples of another relation. Therefore, the `order by` of the original query and the `order by` and `group by` of the rewritten query become more costly.

In order to have a better insight into the performance of the rewritten queries, we analyzed the running time of Query 3 and its rewritten version when the `order by` clause is removed. We show the results in Figure 9 using dashed lines. In this case, the running time of the original query without `order by` is not affected by the size of the clusters, as

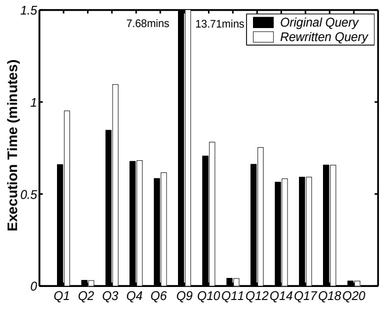

Figure 8. $sf = 1$ **&** $if = 3$

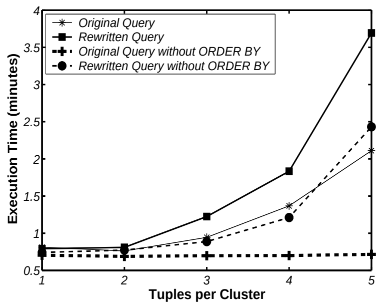

Figure 9. Query 3: $sf = 1$

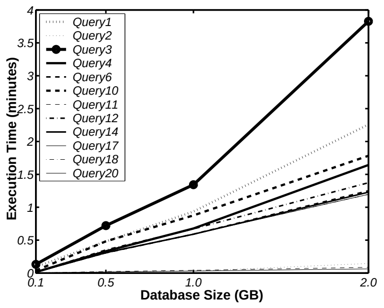

Figure 10. Time over DB size

expected. On the other hand, the rewritten query is affected by cluster size since it has to perform an additional grouping of the tuples.

Finally, we illustrate the scalability of our approach by running the queries on instances of different sizes. In Figure 10, we show the running times of the rewritten queries (with the `order by` clause) on instances with an average cluster size of 3 and database sizes of 100MB, 500MB, 1GB, and 2GB. Except for Query 3, the running times grow in a linear fashion with the size of the database. Query 9, with a higher running time but similar trends, is omitted here but included in the full version of this paper [2].

6 Conclusions

We have introduced a novel approach for querying dirty databases in the presence of probabilistic information about potential duplicates. We presented an algorithm that rewrites queries posed over the dirty database into queries that return answers together with their probability of being in the clean database. Our experiments showed that our methods are both intuitive and scalable. This work, however, opens several avenues for further research. First, we would like to extend the class of queries that can be rewritten to consider, for example, queries with grouping and aggregation. We would also like to relax some of the assumptions of our semantics. For example, we would like to extend our techniques to use dependence information across the clusters.

References

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, pages 586–597, 2002.

[2] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. Technical report, UofT, Dept of CS, CSRG-513, 2005. Available from ftp://www.cs.toronto.edu/csri-technical-reports/513/tr513.pdf.

[3] P. Andritsos, R. J. Miller, and P. Tsaparas. Information-Theoretic Tools for Mining Database Structure from Large Data Sets. In *SIGMOD*, pages 731–742, 2004.

[4] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik. LIMBO: Scalable Clustering of Categorical Data. In *EDBT*, pages 123–146, 2004.

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

[6] D. Barbara, H. G. Molina, and D. Porter. The management of probabilistic data. *IEEE Trans. Knowl. Data Eng.*, 4:487–502, 1992.

[7] I. Bhattacharya and L. Getoor. Deduplication and group detection using links. In *KDD Workshop on Link Analysis and Group Detection*, Seattle, WA, Aug. 2004.

[8] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD*, pages 39–48, 2003.

[9] 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.

[10] R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In *VLDB*, pages 71–81, 1987.

[11] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 1-2(197):90–121, 2005.

[12] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley & Sons, 1991.

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

[14] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 40, 1969.

[15] N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. of Inf. Sys.*, 15(1):32–66, 1997.

[16] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, pages 371–380, 2001.

[17] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[18] L. Lakshmanan, N. Leone, R. Ross, and V. Subrahmanian. Probview: A flexible probabilistic database system. *TODS*, 22(3):419–469, 1997.

[19] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, pages 169–178, 2000.

[20] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD*, pages 269–278, 2002.

[21] W. E. Winkler. The state of record linkage and current research problems. Technical Report RR99/04, US Census Bureau, 1999.