Noname manuscript No.
(will be inserted by the editor)

Query Processing Over Incomplete Autonomous Databases: Query Rewriting Using Learned Data Dependencies

Garrett Wolf · **Aravind Kalavagattu** · **Hemal Khatri** · **Raju Balakrishnan** · **Bhaumik Chokshi** · **Jianchun Fan** · **Yi Chen** · **Subbarao Kambhampati**

Received: date / Accepted: date

Abstract Incompleteness due to missing attribute values (aka "null values") is very common in autonomous web databases, on which user accesses are usually supported through mediators. Traditional query processing techniques that focus on the strict soundness of answer tuples often ignore tuples with critical missing attributes, even if they wind up being relevant to a user query. Ideally we would like the mediator to retrieve such possible answers and gauge their relevance by accessing their likelihood of being pertinent answers to the query. The autonomous nature of web databases poses several challenges in realizing this objective. Such challenges include the restricted access privileges imposed on the data, the limited support for query patterns, and the bounded pool of database and network resources in the web environment.

We introduce a novel query rewriting and optimization framework QPIAD that tackles these challenges. Our technique involves reformulating the user query based on mined correlations among the database attributes. The reformulated queries are aimed at retrieving the relevant possible answers in addition to the certain answers. QPIAD is able to gauge the relevance of such queries allowing tradeoffs in reducing the costs of database query processing and answer transmission. To support this framework, we develop methods for mining *attribute correlations* (in terms of Approximate Functional Dependencies), *value distributions* (in the form of Naïve Bayes Classifiers), and *selectivity estimates*. We present empirical studies to demonstrate that our approach is able to effectively retrieve relevant possible answers with high precision, high recall, and manageable cost.

Keywords Incomplete Databases · Query Rewriting · Uncertainty

G. Wolf
Arizona State University, Tempe, AZ, USA
E-mail: garrett.wolf@asu.edu

A. Kalavagattu
E-mail: aravindk@asu.edu

H. Khatri
E-mail: hemal.khatri@asu.edu

R. Balakrishnan
E-mail: rajub@asu.edu

B. Chokshi
E-mail: bmchoksh@asu.edu

J. Fan
E-mail: jianchun.fan@asu.edu

Y. Chen
E-mail: yi@asu.edu

S. Kambhampati
E-mail: rao@asu.edu Corresponding author. This research was supported in part by the NSF grants IIS 308139 and IIS 0624341, the ONR grants N000140610058 and N000140910032, a Google research award, as well as support from ASU (via ECR A601, the ASU Prop 301 grant to ET-I3 initiative.

1 Introduction

Data integration in autonomous web database scenarios has drawn much attention in recent years, as more and more data becomes accessible via web servers which are supported by back-end databases. In these scenarios, a mediator provides a unified query interface as a global schema of the underlying databases. Queries on the global schema are then rewritten as queries over autonomous databases through their web interfaces. Current mediator systems [28, 24] only return to the user *certain answers* that exactly satisfy all the user query

predicates. For example, in a used car trading application, if a user is interested in cars made by *Honda*, all the returned answers will have the value "Honda" for attribute *Make*. Thus, an *Accord* which has a *missing* value for *Make* will not be returned by such systems. Unfortunately, such an approach is both inflexible and inadequate for querying autonomous web databases which are inherently incomplete. As an example, Table 1 shows statistics on the percentage of incomplete tuples from several autonomous web databases. The statistics were computed from randomly probed samples. The table also gives statistics on the percentage of missing values for the *Body Style* and *Engine* attributes. For example, AUTO TRADER source had 13 attributes, 25K tuples of which 33.7% of tuples had null values. 3.6% of the total tuples had null values in the body attribute, while 8.1% had null values in the engine attributes.[1]

Website	# of Attrs.	# of Tuples	% Incomp.	Missing Body	Missing Engine
Auto Trader	13	25127	33.7%	3.6%	8.1%
Cars Direct	14	32564	98.7%	55.7%	55.8%
Google Base	203+	580993	100%	83.4%	91.9%

Table 1 Statistics on missing values in web databases.

Such incompleteness in autonomous databases should not be surprising as it can arise for a variety of reasons, including:

Incomplete Entry: Web databases are often populated by lay individuals without any central curation. For example, web sites such as *Cars.com* and *Yahoo! Autos*, obtain information from individual car owners who may not fully specify complete information about their cars, thus leaving such databases scattered with missing values (aka "null" values). Consider the previous example where a car owner who leaves the *Make* attribute blank, assuming that it is obvious as the *Model* of the car she is selling is *Accord*.

Inaccurate Extraction: Many web databases are being populated using automated information extraction techniques. As a result of the inherent imperfection of these extractions, many web databases may contain missing values. Examples of this include imperfections in web page segmentation (as described in [17]) or imperfections in scanning and converting handwritten forms (as described in [2]).

Heterogeneous Schemas: Global schemas provided by mediator systems may often contain attributes that do not appear in all of the local schemas. For example, a global schema for the used car trading domain has an attribute called *Body Style*, which is supported by *Cars.com*, but not by *Yahoo! Autos*. Given a query on the global schema for cars having *Body Style* equal to *Coupe*, mediators which only return the certain answers are not able to make use of information from the *Yahoo! Autos* database thereby failing to return a possibly large portion of the relevant tuples.[2]

User-defined Schemas: Another type of incompleteness occurs in the context of applications like Google Base [33] which allow users significant freedom to define and list their own attributes. This often leads to redundant attributes (e.g. Make *vs.* Manufacturer), as well as proliferation of null values (e.g. a tuple that gives a value for Make is unlikely to give a value for Manufacturer and vice versa).

Although there has been work on handling incompleteness in databases (see Section 7), much of it has been focused on single databases on which the query processor has complete control. The approaches developed–such as the "imputation methods" that attempt to modify the database directly by replacing null values with likely values–are not applicable for autonomous databases where the mediator often has restricted access to the data sources. Consequently, when faced with incomplete databases, current mediators only provide the certain answers thereby sacrificing recall. This is particularly problematic when the data sources have a significant fraction of incomplete tuples, and/or the user requires high recall (consider, for example, a law-enforcement scenario, where a potentially relevant criminal is not identified due to fortuitous missing information or a scenario where a sum or count aggregation is being performed).

To improve recall in these systems, one naïve approach would be to return, in addition to all the certain answers, all the tuples with missing values on the constrained attribute(s) as *possible* answers to the query. For example, given a selection query for cars made by "Honda", a mediator could return not only those tuples whose *Make* values are "Honda" but also the ones whose *Make* values are missing(null). This approach, referred to as ALLRETURNED, has an obvious drawback, in that many of the tuples with missing values on constrained attributes are *irrelevant* to the query. Intuitively, not every tuple that has a missing value for *Make* corresponds to a car made by *Honda*! Thus, while improving recall, the ALLRETURNED approach can lead to drastically lower precision.

In an attempt to improve precision, a more plausible solution could start by first retrieving all the tuples with *null* values on the constrained attributes, predicting their missing

[1] The significantly larger percentage of incomplete tuples in the case of GOOGLE BASE are a consequence of the fact that in addition to being uncurated, GOOGLE BASE also allows users to dynamically define their own attribute names and fill them.

[2] Moreover, an attribute may not appear in a schema intentionally as the database manager may suppress the values of certain attributes. For example, the travel reservation website *Priceline.com* suppresses the airline/hotel name when booking tickets and hotel.

values, and then deciding the set of relevant query answers to show to the user. This approach, that we will call ALLRANKED, has better precision than ALLRETURNED. However, most of the web-accessible database interfaces we've found, such as *Yahoo Autos*, *Cars.com*, *Realtor.com*, etc, *do not allow the mediator to directly retrieve tuples with null values on specific attributes*. In other words, we cannot issue queries like "list all the cars that have a missing value for *Body Style* attribute". Even if the sources do support binding of *null* values, retrieving and additionally ranking all the tuples with missing values involves high processing and transmission costs.

Our Approach: In this paper, we present QPIAD,[3]a system for mediating over incomplete autonomous databases. To make the retrieval of possible answers feasible, QPIAD bypasses the null value binding restriction by generating *rewritten* queries according to a set of mined attribute correlation rules. These rewritten queries are designed such that there are no query predicates on attributes for which we would like to retrieve missing values. Thus, QPIAD is able to retrieve possible answers without binding null values or modifying underlying autonomous databases. To achieve high precision and recall, QPIAD learns Approximate Functional Dependencies (AFDs) for attribute correlations, Naïve Bayesian Classifiers (NBC) for value distributions, and query selectivity estimates from a database sample obtained off-line. These data source statistics are then used to gauge the relevance of a possible answer to the original user query. Instead of ranking all possible answers directly, QPIAD first ranks the rewritten queries in the order of the number of relevant answers they are expected to bring as determined by the attribute value distributions and selectivity estimations. Then the top *k* rewritten queries are issued in the order of its relevance to the user query. The retrieve tuples are ranked in accordance with the query that retrieved them, which is the order of their relevance. By ordering the rewritten queries rather than ranking the entire set of possible answers, QPIAD is able to optimize both precision and recall while maintaining efficiency. This query rewriting framework can handle selection, aggregation and join queries, as well as support multiple correlated sources.

Contributions: First, to the best of our knowledge, the QPIAD framework is the first that can retrieve relevant possible answers with missing values on constrained attributes without modifying underlying databases. Consequently, it is suitable for querying incomplete autonomous databases, given a mediator 's query-only capabilities and limited query access patterns to these databases. Second, the idea of using learned attribute correlations, value distributions, and query selectivity to rewrite and rank queries, which consider the natural tension between precision and recall, is also a novel contribution of our work. Third, our framework can leverage attribute correlations among data sources in order to retrieve relevant possible answers from data sources not supporting the query attribute (e.g. local schemas which do not support the entire set of global schema attributes). Furthermore, our experimental evaluation over selection, aggregation, and join queries shows that QPIAD retrieves most relevant possible answers while maintaining low query processing costs.

Last, but certainly not the least, we developed and implemented AFDminer, a scalable technique for mining attribute correlations in the form of approximate functional dependencies. We believe that besides QPIAD, AFDminer can also be used to improve the performance of variety of systems that depend on AFDs (e.g. AIMQ [36] and CORDS [21]).

Significance: The need of returning *maybe* answers besides *certain* answers when querying incomplete databases has been well recognized for a long time [23, 30, 2] and is further motivated by a recent work [29], which shows that the problem of query answering in data exchange can be reduced to the problem of query answering over incomplete databases. As discussed in Section 7, our work also has deep connections to probabilistic databases. Specifically, missing values (*null* values) in a deterministic database can be modeled with a *probability distribution* over a set of values. The unique technical challenge addressed by QPIAD is retrieving incomplete relevant tuples from incomplete databases without materializing corresponding probabilistic databases. It does this with the help of novel query rewriting techniques that are driven by learned data dependencies.

Assumptions: In this paper we focus the discussion on ranking tuples with a single null over the set of query constrained attributes, based on the probability that the missing value actually satisfies the query constraint. For tuples with multiple nulls on constrained attributes, we output them after all the tuples with zero or a single null and simply rank them according to the number of null values, since they are much less likely to be interesting to the user, and at the same time entail computation exponential to the number of nulls *to infer their relevance due to attribute correlation). For example, assume the user poses a query $Q{:}\,\sigma_{Model=Accord \wedge Price=10000 \wedge Year=2001}$ on the relation $R(Make, Model, Price, Mileage, Year, BodyStyle)$. In this case, a tuple $t_1(Honda, null, 10000, 30000, null, Coupe)$ would be placed after tuples with a single null on one of the constrained attributes because it has missing values on *two* of its constrained attributes, namely *Model* and *Year*. However, we assume a tuple $t_2(Honda, null, 10000, null, 2001, Coupe)$ would be ranked based on value inference as it only contains a null on one constrained attribute, namely *Model*. The sec-

[3] QPIAD is an acronym for Query Processing over Incomplete Autonomous Databases.

ond missing value is on *Mileage*, which is not a constrained attribute.

Organization: The rest of the paper is organized as follows. In the next section we cover some preliminaries and an overview of our framework. Section 3 proposes online query rewriting and ranking techniques to retrieve relevant possible answers from incomplete autonomous databases in the context of selection, aggregation, and join queries, as well as retrieving possible answers from data sources which do not support the query attribute in their local schemas. Section 4 provides the details of learning attribute correlations, value distributions, and query selectivity used in our query rewriting phase. A comprehensive empirical evaluation of our approach is presented in Section 5. In Section 6 we introduce an improved attribute correlation mining algorithm used to scale QPIAD to large attribute sets. We discuss the relations with existing work in Section 7 and present our conclusions in Section 8.

2 Preliminaries and Architecture of QPIAD

We will start with formal definitions of complete/incomplete tuples and certain/possible answers with respect to selection queries.

Definition 1 (Complete/Incomplete Tuples) Let $R(A_1, A_2, \cdots, A_n)$ be a database relation. A tuple $t \in R$ is said to be complete if it has non-null values for each of the attributes A_i; otherwise it is considered incomplete. A complete tuple t is considered to belong to the set of completions of an incomplete tuple $\widehat{t}$ (denoted $\mathscr{C}(\widehat{t})$), if t and $\widehat{t}$ agree on all the non-null attribute values.

Now consider a selection query $Q: \sigma_{A_m=v_m}$ over relation $R(A_1, \cdots, A_n)$ where $(1 \leq m \leq n)$.

Definition 2 (Certain/Possible Answers) A tuple t_i is said to be a certain answer for the query $Q: \sigma_{A_m=v_m}$ if $t_i.A_m{=}v_m$. t_i is said to be an possible answer for Q if $t_i.A_m{=}null$, where $t_i.A_m$ is the value of attribute A_m in t_i.

Notice an incomplete tuple is a certain answer to a query, if its null values are not on the attributes constrained in the query.

There are several key functionalities that QPIAD needs in order to retrieve and rank possible answers to a user query: (i) *learning attribute correlations* to generate rewritten queries, (ii) *assessing the value probability distributions* of incomplete tuples to provide a ranking scheme for possible answers, (iii) *estimating query selectivity* to estimate the recall and determine how many rewritten queries to issue, and based on the above (iv) *ordering rewritten queries* to retrieve possible tuples that have a high degree of relevance to the query.

The system architecture of the QPIAD system is presented in Figure 1. A user accesses autonomous databases by issuing a query to the mediator. The query reformulator first directs the query to the autonomous databases and retrieves the set of all certain answers (called the *base result set*). In order to retrieve highly relevant possible answers in ranked order, the mediator dynamically generates rewritten queries based on the original query, the base result set, and attribute correlations in terms of Approximate Functional Dependencies(AFDs) learned from a database sample. The goal of these new queries is to return an *extended result set*, which consists of highly relevant possible answers to the original query. Since not all rewritten queries are equally good in terms of retrieving relevant possible answers, they are ordered before being posed to the databases. The ordering of the rewritten queries is based on their expected *F-Measure* which considers the estimated selectivity and the value distributions for the missing attributes.

QPIAD mines attribute correlations, value distributions, and query selectivity using a small portion of data sampled from the autonomous database using random probing queries. The knowledge mining module learns AFDs and AFD-enhanced Naïve Bayesian Classifiers (where the AFDs play a feature selection role for the classification task) from the samples. Then the knowledge mining module estimates the selectivity of rewritten queries. Armed with the AFDs, the corresponding classifiers, and the selectivity estimates, the query reformulator is able to retrieve the relevant possible answers from autonomous databases by rewriting the original user query and then ordering the set of rewritten queries such that the possible answers are retrieved in the order of their ranking in precision.

3 Retrieving Relevant Possible Answers

In this section, we describe the QPIAD query rewriting approach for effectively and efficiently retrieving relevant possible answers from incomplete autonomous databases. We support queries involving selections, aggregations and joins. This query rewriting framework can also retrieve relevant answers from data sources not supporting the entire set of query constrained attributes.

3.1 Handling Selection Queries

To efficiently retrieve possible answers in their order of precision, QPIAD follows a two-step approach. First, the origi-

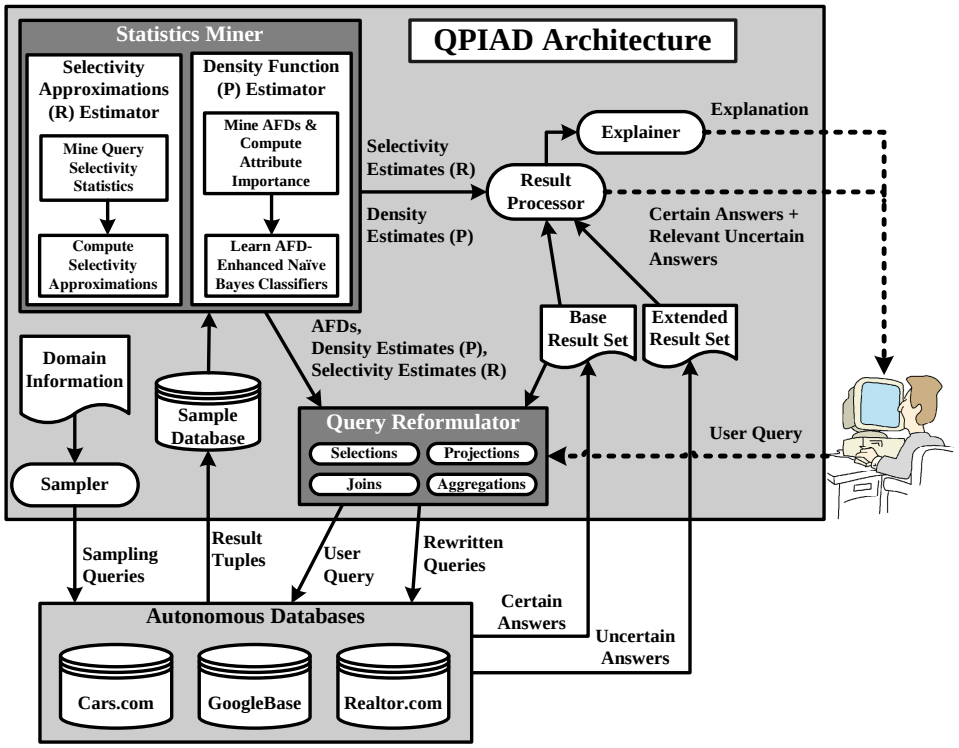


Fig. 1 QPIAD System Architecture.

ID	Make	Model	Year	Body Style
1	Audi	A4	2001	Convt
2	BMW	Z4	2002	Convt
3	Porsche	Boxster	2005	Convt
4	BMW	Z4	2003	null
5	Honda	Civic	2004	null
6	Toyota	Camry	2002	Sedan

Table 2 Fragment of a Car Database

nal query is sent to the database to retrieve the certain answers which are then returned to the user. Next, a group of rewritten queries are intelligently generated, ordered, and sent to the database. This process is done such that the query patterns are likely to be supported by the web databases, and only the most relevant possible answers are retrieved by the mediator in the first place.

3.1.1 Generating Rewritten Queries

The goal of the query rewriting is to generate a set of rewritten queries to retrieve relevant possible answers. Let's consider the same user query Q asking for all convertible cars. We use the fragment of the Car database shown in Table 2 to explain our approach. First, we issue the query Q to the autonomous database to retrieve all the certain answers which correspond to tuples t_1, t_2 and t_3 from Table 2. These certain answers form the *base result set* of Q. Consider the first tuple $t_1 = \langle Audi, A4, 2001, Convt \rangle$ in the base result set. If there is a tuple t_i in the database with the same value for *Model* as t_1 but missing value for *Body Style*, then $t_i.Body\ Style$ is likely to be *Convt*. We *capture this intuition by mining attribute correlations* from the data itself.

One obvious type of attribute correlation is "*functional dependencies*". For example, the functional dependency $Model \rightarrow Make$ often holds in automobile data records. There are two problems in adopting the method directly based on functional dependencies: (i) often there are not enough functional dependencies in the data, and (ii) autonomous databases are unlikely to advertise the functional dependencies. The answer to both these problems involves *learning* approximate functional dependencies from a (probed) sample of the database.

Definition 3 (Approximate Functional Dependency (AFD)) Given a relation R, a subset X of its attributes, and a single attribute A of R, we say that there is an approximate functional dependency (AFD) between X and A, denoted by $X \rightsquigarrow A$, if the corresponding functional dependency $X \rightarrow A$ holds on all but a small fraction of the tuples of R. The set of attributes X is called a *determining set* of A denoted by dtrSet(A).

For example, an AFD $Model \rightsquigarrow Body\ Style$ may be mined, which indicates that the value of a car's *Model* attribute *sometimes* (but not always) determines the value of its *Body Style* attribute. According to this AFD and tuple t_1, we issue a rewritten query $Q'_1: \sigma_{Model=A4}$ with constraints on the *determining set* of the attribute *Body Style*, to retrieve tuples that have the same *Model* as t_1 and therefore are likely to be *Convt* in *Body Style*. Similarly, we issue queries $Q'_2: \sigma_{Model=Z4}$ and $Q'_3: \sigma_{Model=Boxster}$ to retrieve other relevant possible answers.

3.1.2 Ordering Rewritten Queries

In the query rewriting step of QPIAD, we generate new queries according to the distinct value combinations among the base set's determining attributes for each of the constrained attributes. In the example above, we used the three certain answers to the user query Q to generate three new queries: $Q'_1: \sigma_{Model=A4}$, $Q'_2: \sigma_{Model=Z4}$ and $Q'_3: \sigma_{Model=Boxster}$. Although each of these three queries retrieve possible answers that are likely to be more relevant to Q than a random tuple with missing value for *Body Style*, they may not all be equally good in terms of retrieving relevant possible answers.

Thus, an important issue in query rewriting is the order in which to pose the rewritten queries to the database. This ordering depends on two orthogonal measures: the *expected precision* of the query–which is equal to the probability that the tuples returned by it are answers to the original query, and the *selectivity* of the query–which is equal to the number of tuples that the query is likely to bring in. As we shall show in Section 4, both the precision and selectivity can be estimated by mining a probed sample of the database.

For example, based on the value distributions in the sample database, we may find that a *Z*4 model car is more likely to be a *Convertible* than a car whose model is *A*4. As we discuss in Section 4.2, we build AFD-enhanced classifiers which give the probability values $P(Body\ Style{=}Convt|Model{=}A4)$, $P(Body\ Style{=}Convt|Model{=}Z4)$ and $P(Body\ Style{=}Convt|Model{=}Boxster)$. Similarly, the selectivity of these queries can be different. For example, we may find that the number of tuples having $Model{=}A4$ is much larger than that of $Model{=}Z4$.

Given that we can estimate precision and selectivity of the queries, the only remaining issue is how to use them to order the queries. If we are allowed to send as many rewritten queries as we would like, then ranking of the queries can be done just in terms of the expected precision of the query. However, things become more complex if there are limits on the number of queries we can pose to the autonomous source. Such limits may be imposed by the network/processing resources of the autonomous data source or possibly the time that a user is willing to wait for answers.

Given the maximum number of queries that we can issue to a database, we have to find a reasonable tradeoff between the precision and selectivity of the queries issued. Clearly, all else being equal, we will prefer high precision queries to low precision ones and high selectivity queries to low selectivity ones. The tricky issue is how to order a query with high selectivity and low precision in comparison to another with low selectivity and high precision. Since the tension here is similar to the precision vs. recall tension in IR, we decided to use the well known *F-Measure* metric for query ordering. In the IR literature, *F-Measure* is defined as the weighted harmonic mean of the precision (P) and recall (R) measures: $\frac{(1+\alpha)*P*R}{\alpha*P+R}$. We use the query precision for P. We estimate the recall measure R of the query by first computing query throughput, i.e., expected number of relevant answers returned by the query (which is given by the product of the precision and selectivity measures), and then normalizing it with respect to the expected cumulative throughput of all the rewritten queries.

In summary, we use the F-measure ordering to select k top queries, where k is the number of rewritten queries we are allowed to issue to the database. Once the k queries are chosen, they are posed in the order of their expected precision. This way the relevant possible answers retrieved by these rewritten queries need not be ranked again, as their rank – the probability that their null value corresponds to the selected attribute– is the same as the precision of the retrieving query.

Note that the parameter α in the *F-measure*, as well as the parameter k (corresponding to the number of queries to be issued to the sources), can be chosen according to source query restrictions, source response times, network/database resource limitations, and user preferences. The unique feature of QPIAD is its flexibility to generate rewritten queries accordingly to satisfy the diverse requirements. It allows the tradeoff between precision and recall to be tuned by adjusting the α parameter in its *F-Measure* based ordering. When α is set to be 0, the rewritten queries are ordered solely in terms of precision. When α is set to be 1, the precision and recall are equally weighted. The limitations on the database and network resources are taken into account by varying k– the number of rewritten queries posed to the database.

3.2 Query Rewriting Algorithm

In this section, we describe the algorithmic details of the QPIAD approach. Let $R(A_1, A_2, \cdots, A_n)$ be a database relation. Suppose $dtrSet(A_m)$ is the determining set of attribute A_m $(1 \leq m \leq n)$, according to the highest confidence AFD (to be discussed in Section 4.3). QPIAD processes a given

selection query Q: $\sigma_{A_m=v_m}$ according to the following two steps.

1. Send Q to the database and retrieve the base result set $RS(Q)$ as the certain answers of Q. Return $RS(Q)$ to the user.
2. Generate a set of new queries Q', order them, and send the most relevant ones to the database to retrieve the extended result set $\widehat{RS}(Q)$ as relevant possible answers of Q. This step contains the following tasks.
   (a) *Generate rewritten queries.* Let $\pi_{dtrSet(A_m)}(RS(Q))$ be the projection of $RS(Q)$ onto $dtrSet(A_m)$. For each distinct tuple t_i in $\pi_{dtrSet(A_m)}(RS(Q))$, create a selection query Q'_i in the following way. For each attribute A_x in $dtrSet(A_m)$, create a selection predicate A_x=$t_i.v_x$. The selection predicates of Q'_i consist of the conjunction of all these predicates.
   (b) *Select rewritten queries.* For each rewritten query Q'_i, compute the estimated precision and estimated recall using its estimated selectivity derived from the sample. Then order all Q'_is in order of their *F-Measure* scores and choose the top-K to issue to the database.
   (c) *Re-order selected top-K rewritten queries.* Re-order the selected top-K set of rewritten queries according to their estimated precision which is simply the conditional probability of $P_{Q'_i}$=$P(A_m$=$v_m|t_i)$. */* By re-ordering the top-K queries in order of their precision we ensure that the returned tuples are retrieved in the order of their precision, since each tuple will have the same rank as the query that retrieved it. Thus there is no need to (re)rank the retrieved tuples. */*
   (d) *Retrieve extended result set.* Given the top-K queries $\{Q'_1, Q'_2, \cdots, Q'_K\}$ issue them in the according to their estimated precision-base orderings. Their result sets $RS(Q'_1), RS(Q'_2), \cdots, RS(Q'_K)$ compose the extended result set $\widehat{RS}(Q)$. */* All results returned for a single query are ranked equally */*
   (e) *Post-filtering.* If database does not allow binding of null values, (i.e. *access to database is via a web form*) remove from $\widehat{RS}(Q)$ the tuples with $A_m \neq null$. Return the remaining tuples in $\widehat{RS}(Q)$ as the relevant possible answers of Q.

3.2.1 Multi-attribute Selection Queries

Although we described the above algorithm in the context of single attribute selection queries, it can also be used for rewriting multi-attribute selection queries by making a simple modification to Step 2(a). Consider a multi-attribute selection query $Q : \sigma_{A_1=v_1 \wedge A_2=v_2 \wedge \cdots \wedge A_c=v_c}$. To generate the set of rewritten queries Q', the modification requires Step 2(a) to run c times, once for each constrained attribute $A_i, 1 \leq i \leq c$. In each iteration, the tuples from $\pi_{dtrSet(A_i)}(RS(Q))$ are used to generate a set of rewritten queries by replacing the attribute A_i with selection predicates of the form A_x=$t_i.v_x$ for each attribute $A_x \in dtrSet(A_i)$. For each attribute $A_x \in dtrSet(A_m)$ which is not constrained in the original query, we add the constraints on A_x to the rewritten query. As we have discussed in Section 1, we only rank the tuples that contain zero or one *null* in the query constrained attributes. If the user would like to retrieve tuples with more than one null, we output them at the end without ranking.

For example, consider the multi-attribute selection query Q: $\sigma_{Model=Accord \wedge Price\ between\ 15000\ and\ 20000}$ and the mined AFDs $\{Make, Body\ Style\} \rightsquigarrow Model$ and $\{Year, Model\} \rightsquigarrow Price$. The algorithm first generates a set of rewritten queries by replacing the attribute constraint $Model$=$Accord$ with selection predicates for each attribute in the determining set of $Model$ using the attribute values from the tuples in the base set $\pi_{dtrSet(Model)}(RS(Q))$. After the first iteration, the algorithm may have generated the following queries:

Q'_1: $\sigma_{Make=Honda \wedge Body\ Style=Sedan \wedge Price\ between\ 15000\ and\ 20000}$,
Q'_2: $\sigma_{Make=Honda \wedge Body\ Style=Coupe \wedge Price\ between\ 15000\ and\ 20000}$

Similarly, the algorithm generates additional rewritten queries by replacing *Price* with value combination of its determining set from the base set while keeping the original query constraint $Model$=$Accord$. After this second iteration, the following rewritten queries may have been generated:

$$Q'_3: \sigma_{Model=Accord \wedge Year=2002},$$
$$Q'_4: \sigma_{Model=Accord \wedge Year=2001},$$
$$Q'_5: \sigma_{Model=Accord \wedge Year=2003}$$

After generating a set of rewritten queries for each constrained attribute, the sets are combined and the queries are ordered just as they were in Step 2(b). The remainder of the algorithm requires no modification to support multi-attribute selection queries.

3.2.2 Base Set vs. Sample

When generating rewritten queries, one may consider simply rewriting the original query using the sample as opposed to first retrieving the base set and then rewriting. However, since the sample may not contain all answers to the original query, such an approach may not be able to generate all rewritten queries. By utilizing the base set, QPIAD obtains the entire set of determining set values that the source can offer, and therefore achieves a better recall.

3.3 Retrieving Relevant Answers from Data Sources Not Supporting the Query Attributes

In information integration, the global schema exported by a mediator often contains attributes that are not supported

in some of the individual data sources. We adapt the query rewriting techniques discussed above to retrieve relevant possible answers from a data source not supporting the constrained attribute in the query. For example, consider a global schema $GS_{UsedCars}$ supported by the mediator over the sources *Yahoo! Autos* and *Cars.com* as shown in Figure 2, where *Yahoo! Autos* doesn't support queries on *Body Style* attribute. Now consider a query Q: $\sigma_{Body\ Style=Convt}$ on the global schema. The mediator that only returns certain answers won't be able to query the *Yahoo! Autos* database to retrieve cars with *Body Style Convt*. None of the relevant cars from *Yahoo! Autos* can be shown to the user.

Mediator	$GS(Make, Model, Year, Price, Mileage, Body\ Style)$
Cars.com	$LS(Make, Model, Year, Price, Mileage, Body\ Style)$
Yahoo Autos	$LS(Make, Model, Year, Price, Mileage)$

Fig. 2 Global schema and local schema of data sources

In order to retrieve relevant possible answers from *Yahoo! Autos*, we apply the attribute correlation, value distribution, and selectivity estimates learned on the *Cars.com* database to the *Yahoo! Autos* database. For example, suppose that we have mined an AFD $Model \rightsquigarrow Body\ Style$ from the *Cars.com* database. To retrieve relevant possible answers from the *Yahoo! Autos* database, the mediator issues rewritten queries to *Yahoo! Autos* using the base set and AFDs from the *Cars.com* database.

The algorithm that retrieves relevant tuples from a source S_k not supporting the query attribute is similar to the QPIAD Algorithm presented in Section 3.2, except that the base result set is retrieved from the *correlated source* S_c in Step 1.

Definition 4 (Correlated Source) For any autonomous data source S_k not supporting a query attribute A_i, we define a *correlated source* S_c as any data source that satisfies the following: (i) S_c supports attribute A_i in its local schema, (ii) S_c has an AFD where A_i is on the right hand side, (iii) S_k supports the determining set of attributes in the AFD for A_i mined from S_c.

From all the sources correlated with a given source S_k, we use the one for which the AFD for A_i has the highest confidence. Then using the AFDs, value distributions, and selectivity estimates learned from S_c, ordered rewritten queries are generated and issued in Step 2 to retrieve relevant possible answers for the user query from source S_k.

3.4 Handling Aggregate Queries

As the percentage of incomplete tuples increases, aggregates such as *Sum* and *Count* need to take the incomplete tuples into account to get accurate results. To support aggregate queries, we first retrieve the base set by issuing the user's query to the incomplete database. Besides computing the aggregate over the base set (certain answers), we also use the base set to generate rewritten queries according to the QPIAD algorithm in Section 3.2. For example, consider the aggregate query Q: $\sigma_{Body\ Style=Convt \wedge Count(*)}$ over the Car database fragment in Table 2. First, we would retrieve the certain answers t_1, t_2, and t_3 for which we would compute their certain aggregate value $Count(*) = 3$. As mentioned previously, our first choice could be to simply return this certain answer to the user effectively ignoring any incomplete tuples. However, there is a better choice, and that is to generate rewritten queries according to the algorithm in Section 3.2 in an attempt to retrieve relevant tuples whose *BodyStyle* attribute is *null*.

When generating these rewritten queries, tuple t_2 from the base set would be used to form the rewritten query Q'_2: $\sigma_{Model=Z4}$ based on the AFD $Model \rightsquigarrow Body\ Style$. We would find probability $P(Body\ Style{=}Convt | Model{=}Z4)$ and issue the rewritten query. We take this probability—namely the query's precision—and multiply this probability with the aggregate result which is returned. The answer obtained is then added to the certain aggregate answer. For example, the contribution of the tuples having null values for *Body Style* and *Z4* for Model to final count is calculated as

$$Count_{possible}(Model = Z4) = Count(\sigma_{Model=Z4 \wedge Body\ Style=null}) P(Convt | Z4)$$

Similarly the process is repeated for tuples t_1 and t_3 whose corresponding rewritten queries are Q'_1: $\sigma_{Model=A4}$ and Q'_3: $\sigma_{Model=Boxster}$ respectively. Finally all these possible counts are added to the certain count from the complete tuples to get the final count value.

In Section 5, we present the results of our empirical evaluation on aggregate query processing in the context of QPIAD. The results show an improvement in the aggregate value accuracy when incomplete tuples are included in the calculations.

3.5 Handling Join Queries

To support joins over incomplete autonomous data sources, the results are retrieved independently from each source and then joined by the mediator. When retrieving possible answers, the challenge comes in deciding which rewritten queries to issue to each of the sources and in what order.

We must consider both the precision and estimated selectivity when ordering the rewritten queries. Furthermore, we need to ensure that the results of each of these queries agree on their join attribute values. Given that the mediator

provides the global schema, a join query posed to the mediator must be broken down as a pair of queries, one over each autonomous relation. In generating the rewritten queries, we know the precision and selectivity estimates for each of the pieces, thus our goal is to combine each pair of queries and compute a combined estimate of precision and selectivity. It is important to consider these estimates in terms of the query pair as a whole rather than simply considering the estimates of the pair's component queries alone. For example, when performing a join on the results of two rewritten queries, it could be the case that the top ranked rewritten query from each relation does not have join attribute values in common. Therefore despite their high ranks at each of their local relations, the query pair could return little or no answers. As a result, when retrieving both certain and possible answers to a query, the mediator needs to order and issue the rewritten queries intelligently so as to maximize the precision/recall of the joined results.

In processing such join queries over relations $R1$ and $R2$, we must consider the orderings of each pair of queries from the sets $Q1 \cup Q1'$ and $Q2 \cup Q2'$ where $Q1$ and $Q2$ are the complete queries derived from the user's original join query over the global schema and $Q1'$ and $Q2'$ are the sets of rewritten queries generated from the bases sets retrieved from $R1$ and $R2$ respectively. Given that the queries must return tuples whose join attribute values are the same in order for a tuple to be returned to the user, we now consider adjusting the α parameter in our *F-Measure* calculation so as to give more weight to recall without sacrificing too much precision. The details of the approach taken by QPIAD are as follows:[4]

1. Send complete queries $Q1$ and $Q2$ to the databases $R1$ and $R2$ to retrieve the base result sets $RS(Q1)$ and $RS(Q2)$ respectively.
2. For each base set, generate a list of rewritten queries $Q1'$ and $Q2'$ using the QPIAD rewriting algorithm described in Section 3.2.
3. Compute the set of all query pairs QP by taking the Cartesian product of each query from the sets $Q1 \cup Q1'$ and $Q2 \cup Q2'$. For each pair, calculate the new estimated precision, selectivity, and *F-Measure* values.
   (a) For each rewritten query in $Q1'$ and $Q2'$, use the NBC classifiers to determine the join attribute value distributions $JD1$ and $JD2$ given the determining set attribute values from the base sets $RS1$ and $RS2$ respectively as discussed in Section 4.2.
   (b) For each join attribute value v_{j1} and v_{j2} in $JD1$ and $JD2$ respectively, compute its estimated selectivity as the product of the rewritten query's precision, selectivity, and the value probability distribution for either v_{j1} or v_{j2}.
   (c) For each query pair $qp \in QP$ compute the estimated selectivity of the query pair to be

$$EstSel(qp) = \sum_{\substack{v_{j1} \in JD1 \\ v_{j2} \in JD2}} EstSel(qp_1, v_{j1}) * EstSel(qp_2, v_{j2})$$

4. For each query pair, compute its *F-Measure* score using the new precision, estimated selectivity, and recall values. Next, select the top-K query pairs from the ordered set of all query pairs according to the algorithm described in Section 3.2.
5. For each selected query pair qp, if the component queries qp_1 and qp_2 have not previously been issued as part of another query pair, issue them to the relations $R1$ and $R2$ respectively to retrieve the extended result sets $\widehat{RS1}$ and $\widehat{RS2}$.
6. For each tuple $\widehat{t_{i1}}$ in $\widehat{RS1}$ and $\widehat{t_{i2}}$ in $\widehat{RS2}$ where $\widehat{t_{i1}}.v_{j1} = \widehat{t_{i2}}.v_{j2}$ create a possible joined tuple. In the case where either $\widehat{t_{i1}}.v_{j1}$ or $\widehat{t_{i2}}.v_{j2}$ is null, predict the missing value using the NBC classifiers and create the possible join tuple. Finally, return the possible joined tuple to the user.

[4] The selectivity estimation steps are only performed for the rewritten queries because the true selectivity of the complete queries is already known once the base set is retrieved.

4 Learning Statistics to Support Ranking and Rewriting

As we have discussed, to retrieve possible answers in the order of their relevance, QPIAD requires three types of information: (i) attribute correlations in order to generate rewritten queries (ii) value distributions in order to estimate the precision of the rewritten queries, and (iii) selectivity estimates which combine with the value distributions to order the rewritten queries. In this section, we present how each of these are learned. Our solution consists of three stages. First, the system mines the inherent correlations among database attributes represented as AFDs. Then it builds Naïve Bayes Classifiers based on the features selected by AFDs to compute probability distribution over the possible values of the missing attribute for a given tuple. Finally, it uses the data sampled from the original database to produce estimates of each query's selectivity. We exploit AFDs for feature selection in our classifier as it has been shown that appropriate feature selection before classification can improve learning accuracy[5].

4.1 Learning Attribute Correlations by Approximate Functional Dependencies(AFDs)

We mine AFDs from a (probed) sample of database to learn the correlations among attribute. Recall that an AFD ϕ is a functional dependency that holds on all but a small fraction

of tuples. According to [27], we define the *confidence* of an AFD ϕ on a relation R as: $conf(\phi) = 1 - g_3(\phi)$, where g_3 is the ratio of the minimum number of tuples that need to be removed from R to make ϕ a functional dependency on R. We utilize TANE [20] as a blackbox to mine AFDs whose confidence is higher than a specified threshold.

In most cases, AFDs with high confidence are more desirable than AFDs with low confider. However, not all high confidence AFDs are useful for classification and subsequent query generation to retrieve relevant uncertain tuples. The latter include those whose determining set contains high confidence *Approximate Keys (AKeys)*. Similarly as AFD, Akey is defined as a key of all but a small fraction of tuples. For example, consider a relation *car(VIN, Model, Make)*. We mine that *VIN* is an AKey (in fact, a key) which determines all other attributes. Given a tuple t with null value on *Model*, its *VIN* is not helpful in estimating the missing *Model* value, since there are no other tuples sharing t's *VIN* value. Thus, comparing an AFD (VIN$\rightsquigarrow$Make), AFD (Model$\rightsquigarrow$Make), though has a lower confidence, is more useful for classification and query generation in order to retrieve relevant possible tuples from databases for queries with constrained attributes on *Make*. After obtaining all AFDs and AKeys by invoking TANE, we prune AFDs whose determining set is a superset of a high-confidence AKey attributes. Specifically, for each attribute, we find the best AKey whose confidence is above a threshold. Then an AFD is pruned if the difference between its confidence and the confidence of the corresponding AKey is below a threshold δ (currently set at 0.3 based on experimentation).

4.2 Learning Value Distributions using Classifiers

Given a tuple with a null value, we now need to estimate the probability of each possible value of this null. We reduce this problem to a classification problem using mined AFDs as selected features. A classifier is a function f that maps a given attribute vector $\mathbf{x}$ to a confidence that the vector belongs to a class. The input of our classifier is a random sample S of an autonomous database R with attributes $A_1, A_2, \cdots, A_n$ and the mined AFDs. For a given attribute A_m, $(1 \leq m \leq n)$, we compute the probabilities for all possible class values of A_m, given all possible values of its determining set $dtrSet(A_m)$ in the corresponding AFDs.

We construct a Naïve-Bayes Classifier(NBC) A_m. Let a value v_i in the domain of A_m represent a possible class for A_m. Let $\mathbf{x}$ denote the values of $dtrSet(A_m)$ in a tuple with null on A_m. We use Bayes theorem to estimate the probabilities: $P(A_m{=}v_i|\mathbf{x}) = \frac{P(\mathbf{x}|A_m=v_i)P(A_m=v_i)}{P(\mathbf{x})}$ for all values v_i in the domain. To improve computation efficiency, NBC assumes that for a given class, the features $X_1, \cdots, X_n$ are conditionally independent, and therefore we have: $P(\mathbf{x}|A_m{=}v_i) = \prod_i P(x_i|A_m{=}v_i)$. Despite this strong simplification, NBC has been shown to be surprisingly effective[13]. In the actual implementation, we adopt the standard practice of using NBC with a variant of Laplacian smoothing called m-estimates[34] to improve the accuracy.

4.3 Combining AFDs and Classifiers

So far we glossed over the fact that there may be more than one AFD associated with an attribute. In other words, one attribute may have multiple determining set with different confidence levels. For example, we have the AFD $Model \rightsquigarrow Make$ with confidence 0.99. We also see that certain types of cars are made in certain countries, so we might have an AFD $Country \rightsquigarrow Make$ with some confidence value. As we use AFDs as a feature selection step for NBC, we experimented with several alternative approaches for combining AFDs and classifiers to learn the probability distribution of possible values for null. One method is to use the determining set of the AFD with the *highest confidence* which we call the *Best-AFD* method. However, our experiments showed that this approach can degrade the classification accuracy if its confidence is too low. Therefore we ignore AFDs with confidence below a threshold (which is currently set to be 0.5 based on experimentation), and instead use all other attributes to learn the probability distribution using NBC. We call this approach *Hybrid One-AFD*. At the other extreme, we could ignore feature selection based on AFD completely but use all the attributes to learn probability distribution using NBC. Our experiments described in Section 5 show that Hybrid One-AFD approach has the best classification accuracy among these choices.[5]

4.4 Learning Selectivity Estimates

As discussed in Section 3, the F-measure ranking requires an estimate of the selectivity of a rewritten query. This is computed as

$$FMeasure(Q) = SmplSel(Q) * SmplRatio(R) * PerInc(R)$$

Here $SmplSel(Q)$ is the selectivity of the rewritten query Q when it is issued to the sample. *SmplRatio(R)* is the ratio of the original database size over the size of the sample. We

[5] Another alternative to considering single AFDs is to consider mutliple AFDs and use them together to predict the missing attribute. We did experiment with such an "ensemble learning" technique in the initial stages of the work, but found that its predictive accuracy was not better than that provided by the single best AFD. An empirical evaluation of this ensemble learning technique is available in [26].

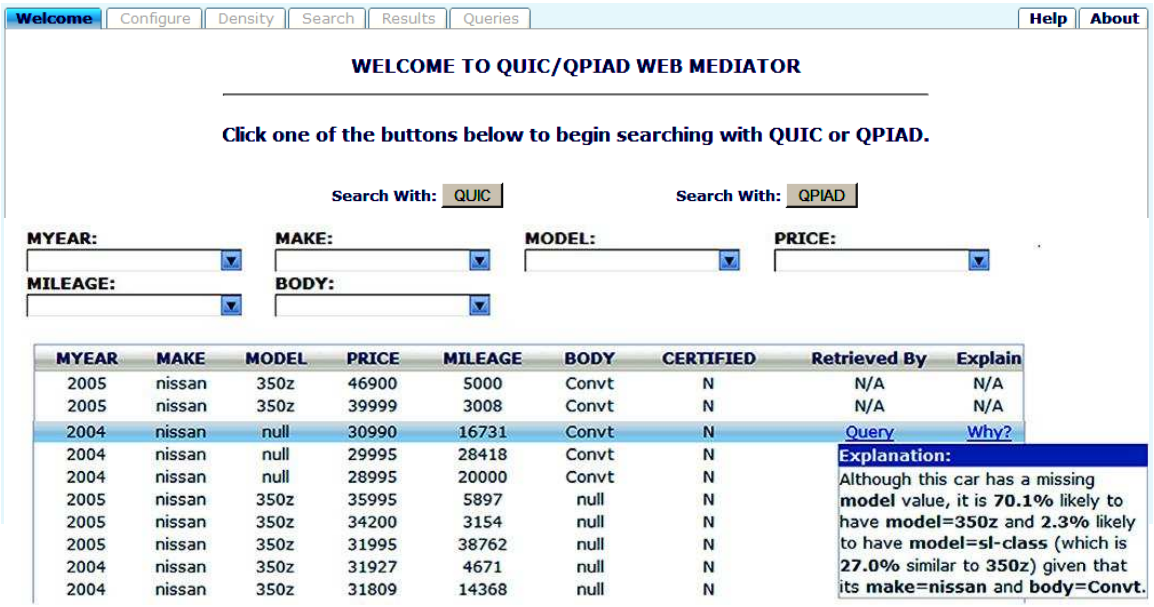


Fig. 3 *A screenshot of the QPIAD prototype (the results shown in the screenshot are in response to the query* $Q : \sigma_{Model=350z \wedge Body=convt}$*)*

send queries to both the original database and its sample offline, and use the cardinalities of the result sets to estimate the ratio. *PerInc(R)* is the percentage of tuples that are incomplete in the database. It can be estimated as the percentage of incomplete tuples that we encountered while creating the sample database.

5 Empirical Evaluation

In this section, we describe the implementation and an empirical evaluation of our system QPIAD for query processing over incomplete autonomous databases.

5.1 Implementation and User Interface

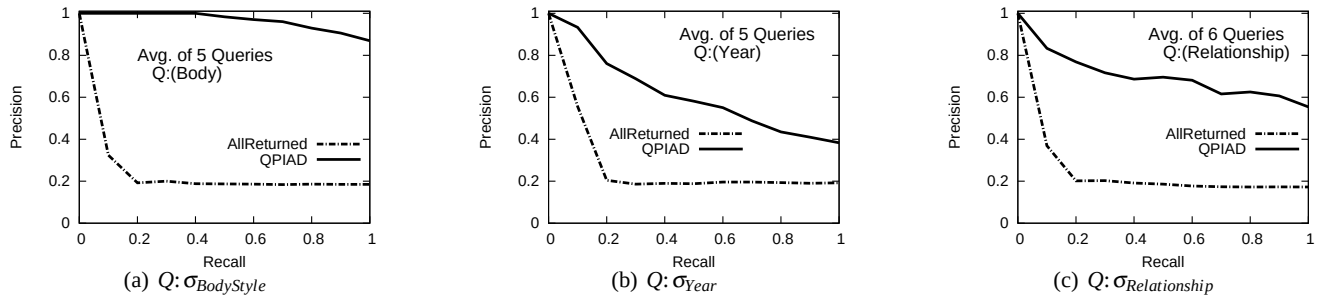
The QPIAD system is implemented in Java and has a web-form based interface through which the users issue their queries. A snapshot of the system in operation is shown in Figure 3. Given a user query, the system returns each relevant possible answer to the user along with a *confidence* measure equal to the probability that the incomplete tuple is an answer to the query. Although the confidence estimate could be biased due to the imperfections of the learning methods, its inclusion can provide useful guidance to the users, over and above the ranking.

In addition, QPIAD can optionally "explain" its relevance assessment by providing snippets of its reasoning as support. In particular, it justifies the confidence associated with an answer by listing the AFD that was used in making the probability/relevance assessment. In the case of our running example, the possible answer t_4 for the query Q' will be justified by showing the learned AFD $Model \rightsquigarrow Body\ Style$.

5.2 Experimental Settings

To evaluate the QPIAD system, we performed evaluations over three data sets. The first dataset, *Cars(year, make, model, price, mileage, body style, certified)*, is built by extracting around 55,000 tuples from *Cars.com*. Databases like this one are inherently incomplete as described in Table 1. The second dataset, *Census(age, workshop, education, marital-status, occupation, relationship, race, sex, capital-gain, capital-loss, hours-per-week, native-country)*, is the *United States Census* database made up of 45,000 tuples which we obtained from the UCI data repository. The third dataset, *Complaints(model, year, crash, fail date, fire, general component, detailed component, country, ownership, car type, market)*, is a *Consumer Complaints* database which contains roughly 200,000 tuples collected from the NHSTA Office of Defect Investigations repository and is used in conjunction with the *Cars* database for evaluating join queries.

To evaluate the effectiveness of our algorithm, we need to have a "ground truth" in terms of the true values corresponding to the missing or null values. To this end, we create our experimental datasets in two steps. First a "ground truth dataset" (GD) is created by extracting a large number of *complete* tuples from the online databases. Next, we create the experimental dataset (ED) by randomly choosing 10% of the tuples from GD and making them incomplete (by



(a) Q: $\sigma_{BodyStyle}$ (b) Q: σ_{Year} (c) Q: $\sigma_{Relationship}$

Fig. 4 Average precision/recall of ALLRETURNED and *QPIAD* for sets of queries on *Cars* (a),(b) and *Census* (c) datasets.

randomly selecting an attribute and making its value null). Given our experience with online databases (see Table 1), 10% incompleteness is fairly conservative.

During the evaluation, the ED is further partitioned into two parts: a training set (i.e. the sample from which AFDs and classifiers are learned) and a test set. To simulate the relatively small percentage of the training data available to the mediators, we experimented with training sets of different sizes, ranging in size from 3% to 15% of the entire database, as will be discussed in Section 5.5.

To compare the effectiveness of retrieving relevant possible answers, we consider two salient dimensions of the QPIAD approach, namely *Ranking* and *Rewriting*, which we evaluate in terms of *Quality* and *Efficiency* respectively. For the experiments, we randomly formulate single attribute and multi attribute selection queries and retrieve possible answers from the test databases.

We compare QPIAD with the ALLRETURNED and ALLRANKED approaches. Recall that ALLRETURNED approach presents all tuples containing missing values on the query constrained attribute without ranking them. The ALLRANKED approach begins by retrieving all the certain and possible answers, as in ALLRETURNED, then it ranks possible answers according to the classification techniques described in Section 4. In fact, neither approach is feasible as web databases are unlikely to support binding of null values in queries. In contrast, the QPIAD approach uses query rewriting techniques to retrieve only relevant possible answers in a ranked order and fits for web applications. Even when bindings of null values are allowed, we show in this section that the QPIAD approach provides better quality and efficiency.

In the initial parts of the evaluation, we focus on comparing the effectiveness of retrieving relevant possible answers. The plots in those parts ignore the "certain" answers as all the approaches are expected to perform equally well over such tuples. The exception are the results presented in Section 5.3.1 which compare the performance of traditional databases and QPIAD as the degree of incompleteness of the underlying database increases.

5.3 Evaluation of Quality

To evaluate the effectiveness of QPIAD ranking, we compare it against the ALLRETURNED approach which simply returns to the user all tuples with missing values on the query attributes. Figure 4 shows the precision and recall curves averaged over sets of queries on the *Cars* and *Census* databases. It shows that the QPIAD approach has significantly higher precision when compared to ALLRETURNED.

To reflect the "density" of the relevant answers along the time line, we also plot the precision of each method at the time when first $K(K{=}1,2,\cdots,100)$ answers are retrieved as shown in Figures 5 and 6. Again QPIAD is much better than ALLRETURNED in retrieving relevant possible answers in the first K results, which is critical in web scenarios.

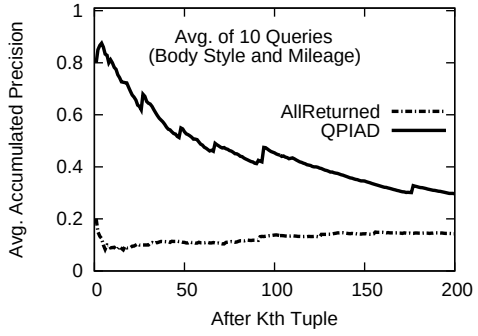


Fig. 5 Avg. Accumulated Precision After Retrieving the Kth Tuple Over 10 Multi-Attribute Queries (Body Style and Mileage).

5.3.1 QPIAD vs. Traditional Databases

In this section, we describe the experiments we have done to evaluate the performance of QPIAD as the degree of in-

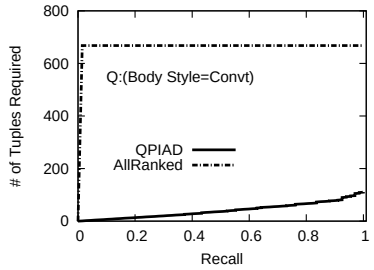


Fig. 7 Number of Tuples Required to Achieve a Given Level of Recall for Query $Q(Cars)\text{:}\,\sigma_{BodyStyle=Convt}$

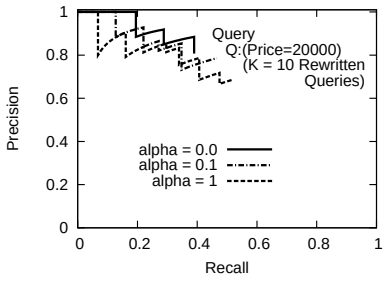


Fig. 8 Effect of α on Precision and Recall in QPIAD for Query $Q(Cars)\text{:}\,\sigma_{Price=20000}$.

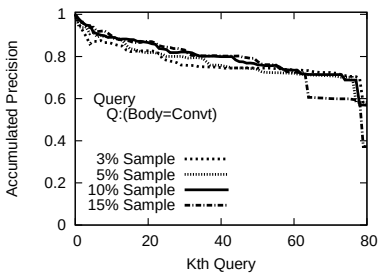


Fig. 9 Accumulated precision curve with different sample sizes on *Cars* database.

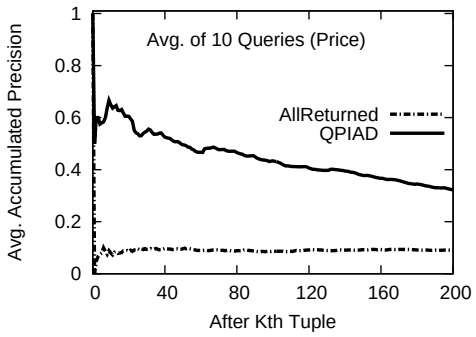
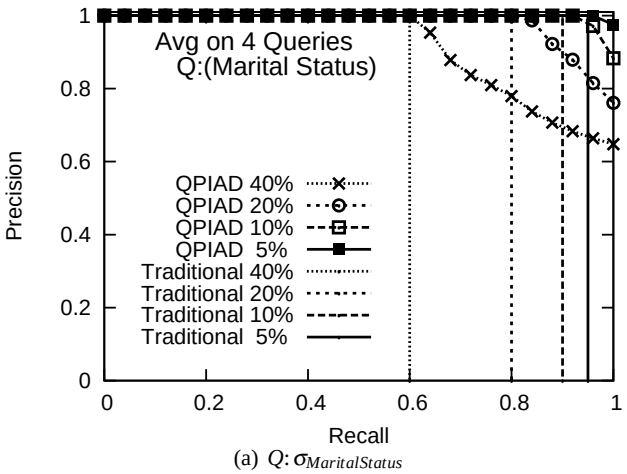


Fig. 6 Avg. Accumulated Precision After Retrieving the Kth Tuple Over 10 Queries (Price).

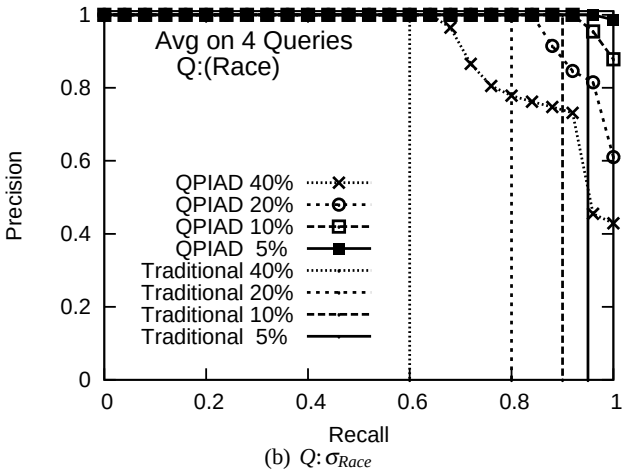
completeness in the underlying database increases. Figure 10 shows the results from our experiments for two example selection queries. We setup the experiments by running each selection query on a database with varying levels of incompleteness on the attribute constrained by the query. For example, the plot named "*QPIAD 20%*" in Figure 10(a) corresponds to running a selection query on marital status on a database that has 20% incompleteness for that attribute (we generated such a database by randomly introducing null values for that attribute into a copy of the master database that is complete).

To put the QPIAD performance in perspective, the plots also show the performance of traditional database techniques (which ignore incomplete tuples) on the same databases. Since traditional databases ignore incompleteness, they have perfect precision on the tuples they return, but fail to return relevant tuples that are incomplete. For example, on a 20% incomplete database, you get full precision until 0.8 recall; no results are returned beyond this. This abrupt loss of recall is shown by the vertical lines in the plots.

Like the traditional databases, *QPIAD* approach is also able to guarantee full precision on the complete tuples. However, unlike the traditional databases, QPIAD is able to continue and provide full recall with slightly reduced precision. As can be expected, the degree of reduction in the precision is correlated with the degree of incompleteness of the underlying database.



(a) $Q\text{:}\,\sigma_{MaritalStatus}$



(b) $Q\text{:}\,\sigma_{Race}$

Fig. 10 Precision/recall of *QPIAD* and traditional databases for increasing levels of incompleteness.

5.3.2 Effect of Alpha Value on F-Measure

To show the effect of α on precision and recall, we've included Figure 8 which shows the precision and recall of the query Q: $\sigma_{Price=20000}$ for different values of α. Here we assume a 10 query limit on the number of rewritten queries we are allowed to issue to the data source. This assumption is reasonable in that we don't want to waste resources by issuing too many unnecessary queries. Moreover, many online sources may themselves limit the number of queries they are willing to answer in a given period of time (e.g. Google Base).

We can see that as the value of α is increased from 0, QPIAD gracefully trades precision for recall. The shape of the plots is a combined affect of the value of α (which sets the tradeoff between precision and recall) and the limit on the number of rewritten queries (which is a resource limitation). For any given query limit, for smaller values of α, queries with higher precision are used, even if they may have lower throughput. This is shown by the fact that the lower α curves are higher up in precision but don't reach high recall. As α increases, we allow queries with lower precision so that we can get a higher throughput, thus their curves are lower down but extend further to the right.

5.4 Evaluation of Efficiency

To evaluate the effectiveness of QPIAD's rewriting, we compare it against the ALLRANKED approach which retrieves all the tuples having missing values on the query constrained attributes and then ranks all such tuples according to their relevance to the query. *As we mentioned earlier, we do not expect the* ALLRANKED *approach to be feasible at all for many real world autonomous sources as they do not allow direct retrieval of tuples with null values on specific attributes*. Nevertheless, these experiments are conducted to show that QPIAD outperforms ALLRANKED even when null value selections are allowed. Figure 7 shows the number of tuples that are retrieved by the ALLRANKED and QPIAD approaches respectively in order to obtain a desired level of recall. As we can see, the number of tuples retrieved by the ALLRANKED approach is simply the total number of tuples with missing values on the query attribute, hence it is independent of the desired level of recall. On the other hand, the QPIAD approach is able to achieve similar levels of recall while only retrieving a small fraction of the tuples retrieved by ALLRANKED. The reason for this is that many of the tuples retrieved by ALLRANKED, while having missing values on the query attributes, are not very likely to be the value the user is interested in. QPIAD avoids retrieving irrelevant tuples and is therefore very efficient. Moreover, the ALLRANKEDapproach must retrieve the entire set of tuples with missing values on constrained attributes in order to achieve even the lowest levels of recall.

5.5 Evaluation of Learning Methods

5.5.1 Accuracy of Classifiers

Since we use AFDs as a basis for feature selection when building our classifiers, we perform a baseline study on their accuracy. For each tuple in the test set, we compute the probability distribution of possible values of a null, choose the one with the maximum probability and compare it against the actual value. The classification accuracy is defined as the proportion of the tuples in the test set that have their null values predicted correctly.

Database	Best AFD	All Attributes	Hybrid One-AFD
Cars	68.82	66.86	68.82
Census	72	70.51	72

Table 3 Comparison of null value prediction accuracy across different AFD-enhanced classifiers

Table 3 shows the average prediction accuracy of various AFD-enhanced classifiers introduced in Section 4.3. In this experiment, we use a training set whose size is 10% of the database. The classification accuracy is measured over 5 runs using different training set and test set for each run. Considering the large domain sizes of attributes in *Cars* database (varying from 2($Certified$) to 416($Model$)), the classification accuracy obtained is quite reasonable, since a random guess would give much lower prediction accuracy. We can also see in Table 3 that the Hybrid One-AFD approach performs the best and therefore is used in our query rewriting implementation.[6]

While classifier accuracy is not the main focus of our work, we did do some comparison studies to ensure that our classifier was competitive, as presented in Figure 11. Specifically, we compared our AFD-enhanced NBC classifier with a NBC classifier, a Bayesian network [18], and decision tree. For Bayes network learning, we experimented with the WEKA Data Mining Software. We found that the AFD-enhanced classifiers were significantly cheaper to learn than Bayes networks; by synergistically exploiting schema-level and value-level correlations, their accuracy was competitive. The details of more evaluations are available [26].

[6] In Table 3 the Best-AFD and Hybrid One-AFD approaches are equal because there were high confidence AFDs for all attributes in the experimental set. When this is not the case, the Hybrid One-AFD approach performs better than the Best-AFD approach.

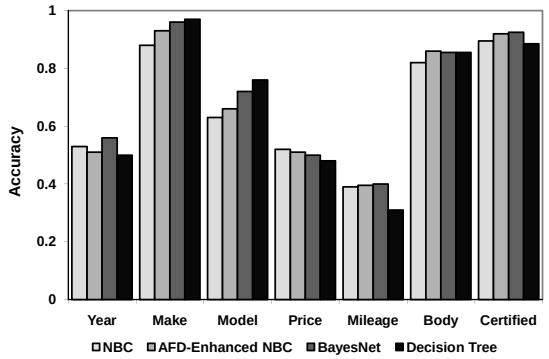


Fig. 11 Comparison of AFD-Enhanced NBC Classifiers and competing classifiers in terms of accuracy.

5.5.2 Robustness w.r.t. Confidence Threshold on Precision

QPIAD presents ranked relevant possible answers to users along with a confidence so that the users can use their own discretion to filter off answers with low confidence. We conducted experiments to evaluate how pruning answers based on a confidence threshold affects the precision of the results returned. Figure 12 shows the average precision obtained over 40 test queries on Cars database by pruning answers based on different confidence thresholds. It shows that the high confidence answers returned by QPIAD are most likely to be relevant answers.

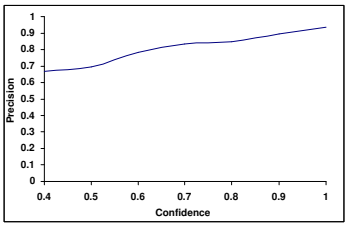


Fig. 12 Average Precision for various confidence thresholds(Cars).

5.5.3 Robustness w.r.t. Sample Size

The performance of QPIAD approach, in terms of precision and recall, relies on the quality of the AFDs, Naïve Bayesian Classifiers and selectivity estimates learned by the knowledge mining module. In data integration scenarios, the availability of the sample training data from the autonomous data sources is restrictive. Here we present the robustness of the QPIAD approach in the face of limited size of sample data. Figure 9 shows the accumulated precision of a selection query on the Car database, using various sizes of sample data as training set. We see that the quality of the rewritten queries all fluctuate in a relatively narrow range and there is no significant drop of precision with the sharp decrease of sample size from 15% to 3%. We obtained a similar result for the Census database [26].

5.6 Evaluation of Extensions

5.6.1 Effectiveness of using Correlation Between Data Sources

We consider a mediator performing data integration over three data sources *Cars.com* (www.cars.com), *Yahoo! Autos* (autos.yahoo.com) and *CarsDirect* (www.carsdirect.com). The global schema supported by the mediator and the individual local schemas are shown in Figure 2. The schema of *CarsDirect* and *Yahoo! Autos* do not support *Body Style* attribute while *Cars.com* does support queries on the *Body Style*. We use the AFDs and NBC classifiers learned from *Cars.com* to retrieve cars from *Yahoo! Autos* and *CarsDirect* as possible relevant possible answers for queries on *Body Style*, as discussed in Section 3.3.

To evaluate the precision, we check the actual *Body Style* of the retrieved car tuples to determine whether the tuple was indeed relevant to the original query. The average precision for the first K tuples retrieved from *Yahoo! Autos* and *CarsDirect* over the 5 test queries is quite high as shown in Figure 13. This shows that using the AFDs and value distributions learned from correlated sources, QPIAD can retrieve relevant answers from data sources not supporting query attribute.

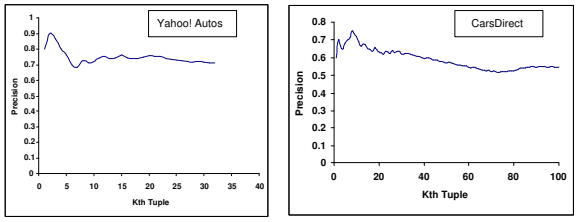


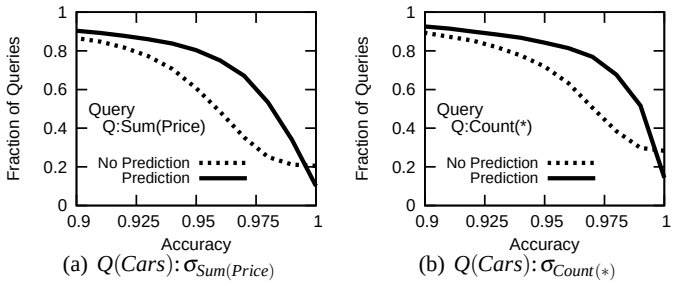
Fig. 13 Precision curves for first K tuples retrieved using correlated source *Cars.com*.

5.6.2 Evaluation of Aggregate Queries

To evaluate our approach in terms of supporting aggregate queries, we measured the accuracy of aggregation queries in QPIAD where missing values in the incomplete tuples are predicted and used to compute the final aggregate result. We compare the accuracy of our query rewriting and missing value prediction with the aggregate results from the complete oracular database and the aggregate results from the incomplete database where incomplete tuples are not considered. Next we will outline the details of our experiments.

We performed the experiments over an *Cars* database consisting of 8 attributes. First, we created all distinct subsets of attributes where the size of the subsets ranged from 1 to 7 (e.g. $\{make\}$, $\{make, model\}$, $\{make, model, year\}$, ..., $\{model\}$, $\{model, year\}$, ..., *etc*.). Next, we issued a query to the sample database and selected the distinct combinations of values for each of these subsets.

Using the distinct value combinations for each of these subsets, we created queries by binding the values to the corresponding attribute in the subsets. We then issued each query to the complete database to find its true aggregate value. We also issued the same query to the incomplete database and computed the aggregate value without considering incomplete tuples. Finally, we issued the query to the incomplete database only this time we predicted the missing values and included the incomplete tuples as part of the aggregate result.



(a) $Q(Cars)$: $\sigma_{Sum(Price)}$ (b) $Q(Cars)$: $\sigma_{Count(*)}$

Fig. 14 Accuracy of aggregate queries with and without missing value prediction.

In Figure 14, we show the percentage of queries achieving different levels of accuracy with and without missing value prediction. The results are significant, as the QPIAD prediction approach almost dominates the no prediction approach for both Figure 14(a) and 14(b). In particular, significantly more fraction of queries achieve 94-99% accuracy for prediction approach than for no prediction approach.[7]

5.6.3 Evaluation of Join Queries

Although *QPIAD* does have the ability to support general joins (as explained in Section 3.5), a tricky issue is that a join can, in general, result in tuples that are no longer guaranteed to be independent and can be mutually exclusive. For example, suppose that a tuple with a null value on attribute "model" is inferred to be "Accord" with probability of 0.6 and be "Civic" with probability of 0.4. This tuple can be joined with two complete tuples, one with the model "Accord" and one with model "Civic." These two tuples generated by the join are clearly mutually exclusive (since a car can be of only one model). The presence of mutually exclusive tuples presents problems in terms of displaying results to the users in an intelligible fashion (since the users can no longer assume that a given subset of the result tuples are actually feasible together). Because of this, in our experiments we focused on a sub-class of join queries that correspond to the join of the results of two selection queries.

Specifically, we performed a set of experiments on the *Cars* and *Complaints* databases. In the experiments, we join the *Cars* and *Complaints* relations for a particular value of the *Model* attribute . The experimental results shown in Figure 15 involve join queries where the attributes from both the relations are constrained. We evaluate the performance of our join algorithm in terms of precision and recall with respect to a complete oracular database.

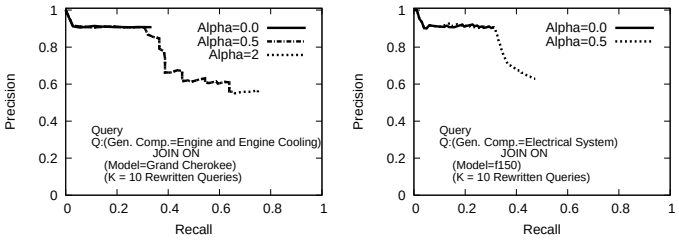


Fig. 15 Precision-Recall Curves for Queries on *Cars* $\bowtie_{Model}$ *Complaints*

We present the results for a join query $Model = Grand\ Cherokee \wedge General\ Component = Engine\ and\ Engine\ Cooling$. We set α to 0, 0.5 and 2 to measure the effect of giving different preferences to precision and recall. In addition, we restricted the number of rewritten queries which could be sent to the database to 10 queries. Figure 15(a) shows the precision-recall curve for this query. We can see that for $\alpha = 0$ high precision is maintained but recall stops at 0.34. For $\alpha = 0.5$ the precision is the same as when $\alpha = 0$ up until recall reaches 0.31. At this point, the precision decreases although, a higher recall, namely 0.66, is achieved. The precision when $\alpha = 2$ is similar to the case where $\alpha = 0.5$ but achieves 0.74 recall with only a small loss in precision near the tail of the curve. When looking at the top 10 rewritten queries for each of these α values we found that when $\alpha = 0$, too much weight is given to precision and thus incomplete tuples are never retrieved from the *Cars* database. This is due to our ability to predict missing values which happens to be better on the *Complaints* database and hence the top 10 rewritten queries tend to include the complete

[7] One seeming anomaly is that the fraction of queries with accuracy one with prediction approach is slightly less compared to no prediction. This however is due to a few queries in the test set containing answers only from the complete tuples, and return fully correct result sets without prediction.

query from the *Cars* database paired with an incomplete query from the *Complaints* database. However, when $\alpha = 0.5$ or $\alpha = 2$ incomplete tuples are retrieved from both the databases because in this approach the ranking mechanism tries to combine both precision and recall. Similar results for the query Q :$_{Model=f150 \wedge General\ Component=Electrical\ System}$ are shown in Figure 15(b).

6 Enhancing the Scalability of QPIAD on Large Attribute Sets

At the start of this work, the QPIAD system utilized TANE [20] as a blackbox to mine approximate functional dependencies (AFD) that meet a confidence threshold on a sample of the databases. However, when we began working with datasets containing larger sets of attributes, we discovered that the TANE algorithm does not scale. It can not handle a database with more than 20 attributes, which limits the applicability of QPIAD. To address this problem and enable QPIAD to handle databases with large attribute sets, we developed an algorithm called AFDMiner. As will be shown in 6.3, the quality of the query results (in terms of precision and recall) using AFDMiner is similar to that by invoking TANE, while AFDMiner provides a significantly better running time performance. AFDMiner's performance advantage allows QPIAD to scale to datasets which were not possible when using the TANE algorithm. In this section, we start with the intuition of how AFDMiner achieves performance speedup, discuss the details of the algorithm in Section 6.1 and Section 6.2, and finally present a performance evaluation in Section 6.3.

It is easy to see that the number of possible AFDs in a database is exponential in the number of attributes in the database, thus AFD mining is in general expensive. As discussed in Section 4.1, not all AFDs with high confidence are useful in terms of generating rewritten queries to retrieve relevant possible answers. In Section 4.1, we prune "useless" AFDs during a post-processing step after all the AFDs and AKeys are discovered by TANE. In this section, we present a technique that pro-actively prunes "useless" AFDs as well as AFDs with extraneous attributes during the mining process itself, thus pruning the search space and improving the efficiency.

6.1 Specificity-based Pruning

In this section, we propose a measure called *"specificity"* which generalizes the intuition discussed in Section 4.1 to quantify how likely an AFD will be useful to QPIAD in terms of retrieving relevant possible answers in query rewriting. Furthermore, the value of specificity can be computed independently with other AFDs, and thus can be exploited to prune search space during AFD mining.

Recall that a high confidence AKey is not useful for retrieving relevant possible answers. For example, consider the case where *VIN* is a high confidence AKey (in fact, a key). Given a query that selects cars whose *Make* is *Honda*, the dependency $VIN \rightarrow Make$ has perfect accuracy, however it is not useful in retrieving tuples that have a missing value for *Make* but are likely to be *Honda*. This is because no two tuples have the same *VIN*.

This example illustrates that the distribution of values for the determining set is an important measure in judging the "usefulness" of an AFD in terms of query rewriting. For an AFD $X \rightsquigarrow A$, the fewer distinct values there are of X, and the more tuples in the database that have the same value, potentially the more relevant possible answers can be retrieved through query rewriting. To quantify this, we first define the *support of a value* α_i of an attribute set X, $support(\alpha_i)$, as the occurrence frequency of value α_i in the training set:

$$support(\alpha_i) = count(\alpha_i)/N,$$

where N is the number of tuples in the training set.

Now we measure how the values of an attribute set X are distributed using specificity, which is defined as the information entropy of the set of all possible values of attribute set X: $\{\alpha_1, \alpha_2, \ldots, \alpha_m\}$, normalized by the maximum possible entropy (which is achieved when X is a key). Thus, specificity is a value that lies between 0 and 1:

$$specificity\ (X) = \frac{-\sum_1^m support(\alpha_i) \times \log_2(support(\alpha_i))}{\log_2(N)}$$

When there is only one possible value of X, then this value has the maximum support and is the least specific, thus we have specificity equals to 0. When all values of X are distinct, each value has the minimum support and is most specific. In fact, X is a key in this case and has specificity equal to 1.

The specificity of an AFD is defined as the specificity of its determining set.

$$specificity\ (X \rightsquigarrow A) = specificity\ (X)$$

Intuitively, using AFDs with lower specificity values in query rewriting allows QPIAD to retrieve more relevant possible answers per issued query.

Monotonicity of specificity: Specificity has a useful monotonicity property that can be exploited in pruning candidate AFDs during the mining phase (see Section 6.2). Given two candidate AFDs $X \rightsquigarrow A$ and $Y \rightsquigarrow A$, where X and Y are determining attribute sets and Y is a superset of X, it is easy to see that the specificity of the second AFD is greater than or equal

to the specificity of the first (since Y has more attributes than X, the number of distinct values of Y is no less than that of X). In other words, given an AFD that already has a specificity beyond a desirable threshold, we do not need to consider adding additional attributes to its determining set.

6.2 A specificity-sensitive algorithm for mining AFDs

We now discuss our algorithm for mining a set of AFDs from a relational table, such that the mined AFDs all have confidence above threshold *minConfidence*, and specificity below *maxSpecificity*.

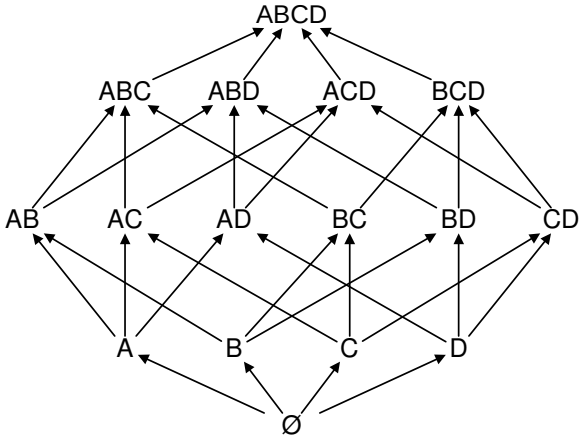


Fig. 16 Set containment lattice of 4 attributes

The operation of the algorithm can be best understood in terms of a search through a set containment lattice, made of attributes in the relation being mined. Specifically, in this lattice, a directed edge connects an attribute set with its superset that contains one more attribute. For example consider a relation with four attributes, *A*, *B*, *C*, *D*. The set containment lattice for this relation is shown in Figure 16.

Our algorithm does a bottom-up breadth-first search in the lattice to mine AFDs that satisfy the thresholds. The search starts from singleton sets of attributes and works its way to larger attribute sets through the set containment lattice. When the algorithm navigates the edge from a set X to its superset $X \cup \{A\}$, it tests whether AFDs of the form $X \rightsquigarrow A$ has a confidence higher than *minConfidence*. Two pruning strategies are interleaved into this navigation.

Specificity-based Pruning. We do not test candidate AFDs whose specificity is guaranteed to violate the specified threshold, *maxSpecificity*. Recall that for two attribute sets X and Y, if $Y \supset X$, then specificity (Y) $\geq$ specificity (X). Thus, if an AFD $X \rightsquigarrow A$ has specificity higher than the threshold, then any AFD $Y \rightsquigarrow A$, $Y \supset X$, must also have its specificity higher than the threshold, and thus does not need to be considered. Based on this property, during the lattice navigation, if specificity (X) is higher than the threshold, then we do not need to further consider any path in the lattice starting from X. This implicitly prunes all AFDs whose determining set is a superset of X.

For example, in Figure 16, if specificity (*AB*) is higher than the threshold, then we do not need to traverse the path from *AB* to *ABC*, from *AB* to *ABD*, and recursively from *ABC* to *ABCD*, from *ABD* to *ABCD*. This indicates that we do not need to consider AFDs $AB \rightsquigarrow C$, $AB \rightsquigarrow D$, and recursively $ABC \rightsquigarrow D$, $ABD \rightsquigarrow C$.

Redundancy-based Pruning. Although our algorithm's focus is on AFDs and rather than FDs (functional dependencies), the FDs it mines along the way do provide powerful pruning on the subsequent AFD mining. Given a functional dependency $X \rightarrow A$, we do not test for candidate AFDs $Y \rightsquigarrow A$, where Y is a superset of X. Note that we only prune AFDs whose determining sets subsume that of an FD. As we have discussed before, given an AFD $X \rightsquigarrow A$, we still have to test $Y \rightsquigarrow A$, $Y \supset X$, as the latter may have a higher confidence and therefore can be a better AFD.

To efficiently prune the search space, for each node (i.e. attribute set) X in the lattice, we use $\mathscr{R}(X)$ to record the set of attributes, such that $\forall A \in \mathscr{R}(X)$, $X \rightsquigarrow A$ can be a potential AFD that does not have extraneous attributes on the determining set, and thus needs to be further tested for confidence. For example, consider the relation with attributes *A*, *B*, *C*, *D*. If we have $A \rightarrow B$, then *B* should not be in any of the following sets: $\mathscr{R}(AC)$, $\mathscr{R}(AD)$, or $\mathscr{R}(ACD)$.

To compute $\mathscr{R}(X)$, we start by initializing it to $U - X$, where U is the set of all attributes in the relation. If we discover a FD $X \rightarrow A$, then we update $\mathscr{R}(X) = \mathscr{R}(X) - \{A\}$. When we navigate an edge in the lattice from X to $X \cup \{B\}$, we update $\mathscr{R}(X \cup \{B\}) = \mathscr{R}(X \cup \{B\}) \cap \mathscr{R}(X)$. In this way, for all nodes Y in the lattice such that $Y \supseteq X$, A is removed from $\mathscr{R}(Y)$ recursively during the bottom-up navigation.

For example, referring to Figure 16, originally we have $\mathscr{R}(A) = \{B,C,D\}$. Suppose that we have discovered a FD $A \rightarrow B$, we update $\mathscr{R}(A) = \{C,D\}$. As we navigate the edge from *A* to *AC*, we have $\mathscr{R}(AC) = \mathscr{R}(AC) \cap \mathscr{R}(A) = \{B,D\} \cap \{C,D\} = \{D\}$, denoting that the only AFD that has *AC* as determining set and needs to be considered is $AC \rightsquigarrow D$. Similarly, $\mathscr{R}(AD) = \mathscr{R}(AD) \cap \mathscr{R}(A) = \{B,C\} \cap \{C,D\} = \{C\}$; $\mathscr{R}(AB) = \mathscr{R}(AB) \cap \mathscr{R}(A) = \{C,D\} \cap \{C,D\} = \{C,D\}$. The process is recursively performed when we navigate the lattice further up. Traversing the edge from *AC* to *ACD*, we have $\mathscr{R}(ACD) = \{B\} \cap \{D\} = \emptyset$.

Furthermore, notice that if $\mathscr{R}(X) = \emptyset$, then $\mathscr{R}(Y) = \emptyset$ for all supersets Y of set X. So we do not need to further consider

all paths in the lattice starting from X. This indicates that all AFDs whose determining set is a superset of X are pruned.

Using the above pruning strategies, the FDs or AFDs mined at the lower levels can reduce the computation at the higher levels thus resulting in efficient AFD mining.

Algorithm 1 presents the outline of our approach. Through lines 8 through 12 it computes all the AFDs and FDs that are not pruned yet at a given level of the lattice, and outputs the ones which meet the *minConfidence* threshold. It also updates $\mathscr{R}(X)$ (line 13) for pruning AFDs with extraneous attributes. Lines 17-18 correspond to the pruning of empty $\mathscr{R}(X)$ sets; and Lines 19-20 correspond to the pruning based on *max*specificity threshold. Line 21 corresponds to the process of generating the next level of lattice based on the nodes at the current level.

Algorithm 1 AFDMiner(*minConfidence*, *maxSpecificity*)

1: $L_0 := \{\emptyset\}$
2: $\mathscr{R}(\emptyset) := R$
3: $L_1 := \{\{A\} \mid A \in R\}$
4: $\ell := 1$
5: **while** $L_\ell \neq \emptyset$ **do**
6: **for all** $X \in L_\ell$ **do**
7: $\mathscr{R}(X) := \bigcap_{A \in X} \mathscr{R}(X \backslash \{A\})$
8: **for all** $X \in L_\ell$ **do**
9: **for all** $A \in X \cap \mathscr{R}(X)$ **do**
10: **if** $Confidence(X \backslash \{A\} \rightarrow A) \geq minConfidence$ **then**
11: **if** $(X \backslash \{A\} \rightarrow A)$ holds exactly **then**
12: output $X \rightarrow A$ as an FD
13: remove A from $\mathscr{R}(X)$
14: **else**
15: output $X \rightsquigarrow A$ as an AFD with its *Confidence*
16: **for all** $X \in L_\ell$ **do**
17: **if** $\mathscr{R}(X) = \emptyset$ **then**
18: delete path starting from X from L_ℓ
19: **if** $Calculatespecificity(X) \geq max$specificity **then**
20: delete path starting from X from L_ℓ
21: $L_{\ell+1} = \{X \mid |X| = \ell + 1$ and $\forall$ Y s.t. Y $\subset$ X and $|Y| = \ell$, we have Y $\in L_\ell$ $\}$.
22: $\ell := \ell + 1$

6.3 Evaluation of AFDMiner in the Context of QPIAD

Now we present an empirical evaluation of AFDMiner and TANE in terms of both speed and the quality of the mined AFDs. (A more comprehensive evaluation is available in [25])

We performed the evaluation over the *Cars* and *Census* datasets described in Section 5.2. In addition, an extended version of the *Census* dataset, referred to as *CensusExt*, was used which contains 25 attributes for the performance evaluation. In the following experiments, whenever the TANE approach is mentioned, we are referring to the approach as described in Section 4.1 where an AFD that has a close confidence with an AKey is pruned in a postprocessing step after all AFDs and AKeys above a confidence threshold are obtained.

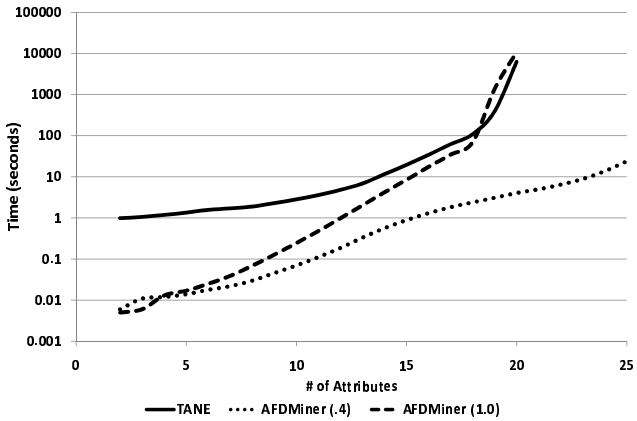


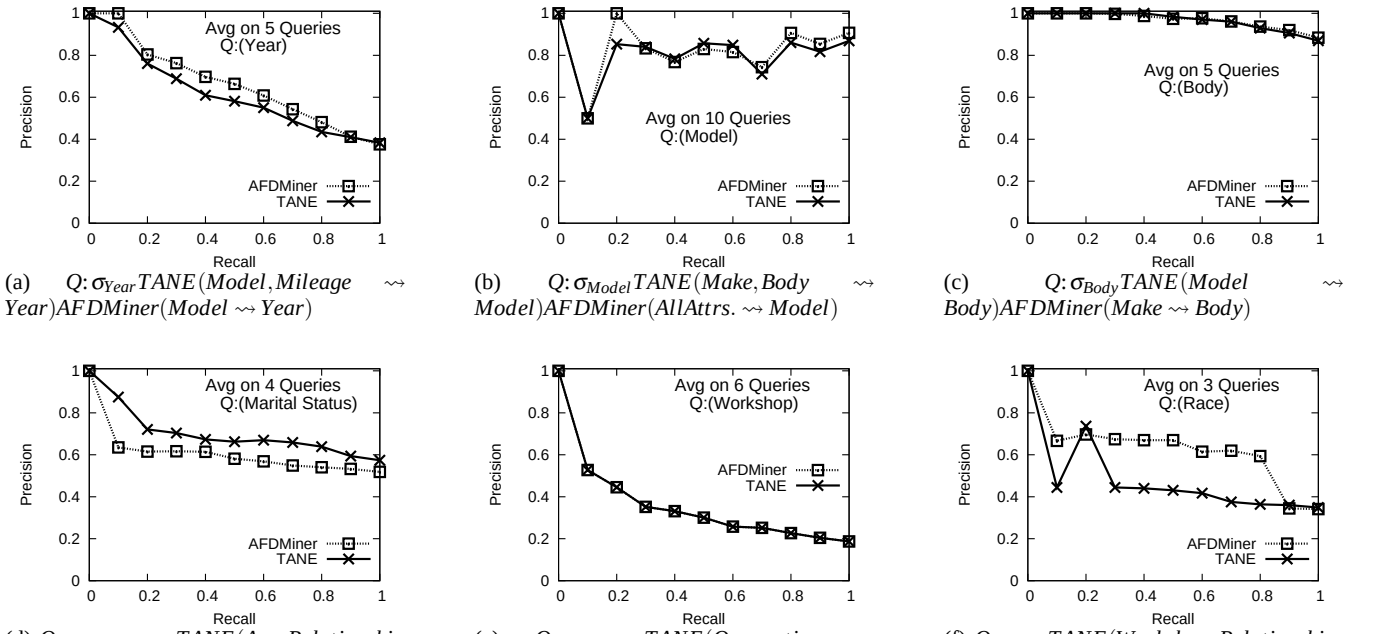
Fig. 17 Varying size of the attribute set for AFDMiner vs. TANE over the *CensusExt* dataset.

6.3.1 Evaluation of Efficiency and Scalability

To evaluate the efficiency of the AFDMiner algorithm, we ran experiments varying the specificity parameter and recorded the time taken to generate the AFDs. For all the efficiency experiments over the extended *Census* dataset, the minimal confidence threshold was set to be 0.8 and the maximal specificity threshold was evaluated at 0.4 and 1.0. The number of tuples and number of attributes, if unspecified, are taken to be 5000 and between 3 and 25 respectively.

As we can see in Figure 17, AFDMiner with a specificity threshold of 0.4 (referred to as AFDMINER0.4 in the figure), significantly outperforms both the TANE approach and the AFDMiner approach with specificity threshold set to 1.0 (referred to as AFDMINER1.0 in the figure), which we'll refer to as AFDMiner1.0. In comparison to the TANE approach, AFDMiner0.4 performs better due to the early pruning of AFDs that don't meet the specificity threshold. The key point to notice is that as the size of the attribute set grows larger, the performance increase due to AFDMiner's early pruning strategy becomes significantly more important. AFDMiner is able to process the entire Census dataset with 25 attributes in under 25 seconds, whereas TANE has trouble processing a 20-attribute dataset in under 1.75 hours, and is unable to process the 25-attribute dataset.

Similarly, when we consider the comparison between AFDMINER0.4 and AFDMINER1.0 where the only difference is the specificity threshold, we again see that AFDMINER0.4 is the clear winner. This shows that setting



(a) $Q{:}\,\sigma_{Year}TANE(Model, Mileage \rightsquigarrow Year)AFDMiner(Model \rightsquigarrow Year)$

(b) $Q{:}\,\sigma_{Model}TANE(Make, Body \rightsquigarrow Model)AFDMiner(AllAttrs. \rightsquigarrow Model)$

(c) $Q{:}\,\sigma_{Body}TANE(Model \rightsquigarrow Body)AFDMiner(Make \rightsquigarrow Body)$

(d) $Q{:}\,\sigma_{MaritalStatus}TANE(Age, Relationship \rightsquigarrow MaritalStatus)AFDMiner(Relationship \rightsquigarrow MaritalStatus)$

(e) $Q{:}\,\sigma_{Workshop}TANE(Occupation \rightsquigarrow Workshop)AFDMiner(Occupation \rightsquigarrow Workshop)$

(f) $Q{:}\,\sigma_{Race}TANE(Workshop, Relationship \rightsquigarrow Race)AFDMiner(NativeCountry \rightsquigarrow Race)$

Fig. 18 Average precision/recall of using AFDs mined from TANE and AFDMiner for sets of queries on *Cars* (a),(b),(c) and *Census* (d),(e),(f) datasets.

a lower specificity threshold effectively prunes the search space and dramatically improves running time performance. It should be noted that as specificity increases, it becomes closer and closer to the TANE approach, wherein specificity based pruning is not taken into account. As Figure 17 shows, AFDMINER1.0 imitates TANE in its inability to scale well to larger attribute sets. In fact, we notice that for larger attribute sets, AFDMINER1.0 seems to be performing worse than the TANE approach. The reason for its poor performance over such datasets is due to the non-minimality of the rules it produces. Given that AFDMINER1.0 is essentially not utilizing any specificity based pruning due to the maximal threshold, the search space that it must traverse is similar to that of TANE except that AFDMINER1.0 will also have to traverse the paths of non-minimal rules. This explains its poor performance compared to TANE.

6.3.2 Evaluation of Quality

To evaluate the quality of the AFDs generated by AFDMiner with those generated by the TANE approach, we measured the precision/recall of queries issued over the *Cars* and *Census* datasets. For each dataset, we selected several attribute binding patterns and for each binding pattern, we issued a variety of queries, each time binding a different value to the bound attributes. The results of these queries were averaged and were used to plot the precision/recall curves in Figure 18.

As the plots show, the precision/recall achieved by AFDMiner's AFDs is on par with those produced by TANE. In some query patterns, the AFDs produced by AFDMiner show higher quality (Figure 18 (a),(f)); in some query patterns, those from TANE show higher quality (Figure 18 (d)) while the remaining plots (Figure 18 (b),(c),(e)) show no clear winner, either equal or tend to alternate as the curves progress.

As shown in the experiments, AFDMiner is more efficient and scalable than TANE for AFD mining; in fact, *AFDMiner is the only solution among the two that scales well enough to maintain its usability in the presence of large attribute sets*. The quality of the AFDs generated by AFDMiner is similar to that generated by TANE. Given the significant performance gains while maintaining comparable quality, AFDMiner is used in the current prototype of QPIAD; it allows QPIAD to handle data with large attribute sets.

7 Related Work and Discussion

Querying Incomplete Databases: The need of returning *maybe* answers besides *certain* answers when querying incomplete databases has been well recognized for a long time

(a)

id	Make	Model	Body Style
1	BMW	Z4	Convt
2	Audi	A4	*null*
.	.	.	.

(b)

id	Make	Model	Body Style	Prob
1	BMW	Z4	Convt	1
2	Audi	A4	Convt	0.7
2	Audi	A4	Sedan	0.3
.	.	.	.	.

(c)

Prob	Make	Model	Model	Prob:
1	BMW	Z4	Convt	1
2	Audi	A4	Convt	0.7
.	.	.	.	.

Table 4 Illustration of processing query $Q : \sigma_{BodyStyle=Convt}$ on a deterministic but incomplete database, shown in Table (a). Table (b) represents a learned probabilistic database consisting of possible completions of incomplete tuples in Table (a). Table (c) represents the result of evaluating Q on Table (b).

[23,30,2] and is further motivated by a recent work [29], which shows that the problem of query answering in data exchange can be reduced to the problem of query answering over incomplete databases.

There are two typical approaches for query answering on incomplete databases. The first ([23,30,2]) handles *null* using one of three different methods, each with an increasing generality: (i) Codd Tables where all the *null* values are treated equally as a special value; (ii) V-tables which allow many different *null* values marked by variables; and (iii) Conditional tables which are V-tables with additional attributes to record conditions.

The second type ([7,3,44,16]) takes a probabilistic approach to quantify the degree of relevance of a possible answer by considering the distribution of possible completions of an incomplete tuple. Our work falls in this second category. The critical novelty of our work is that our approach learns the distribution automatically, and it allows the mediator to query autonomous incomplete databases without modifying the original database in any way. [7] handles incompleteness for aggregate queries in the context of OLAP databases, by relaxing the original queries using the hierarchical OLAP structure. Whereas our work learns attribute correlations, value distributions and query selectivity estimates to generate and rank rewritten queries.

Querying Probabilistic Databases: There are deep connections between incomplete databases and probabilistic databases [42,42,9,41,43]. Specifically, missing values (*null* values) in a deterministic database can be modeled with a *probability distribution* over a set of values. Thus, once this distribution is assessed, it can be used to turn an incomplete deterministic database into a probabilistic database. The probabilistic database that results has "attribute-level uncertainty" (where the uncertainty only arises in terms of which specific value a missing attribute is going to take). In [10], Dalvi & Suciu study the problem of modeling databases with attribute level uncertainty with a special class of probabilistic databases called disjoint-independent probabilistic databases (DI databases). In DI databases, all correlations are confined to within individual tuples, and inter-tuple independence holds.

Through mining attribute and value dependencies, QPIAD models a deterministic incomplete database as a probabilistic complete database. For instance, the incomplete database in Table 4(a) is modeled by the probabilistic database in Table 4(b), where the tuples with the same ids are mapped from the same deterministic tuple in the source database and are considered as mutually exclusive; and tuples with different ids are mapped from different tuples in the source database and are mutually independent. It is easy to see that this probabilistic database is a DI database. The query results generated by QPIAD are the same as the ones generated by processing the query directly on a DI database.

The unique technical challenge addressed by QPIAD is retrieving incomplete relevant tuples from incomplete databases without materializing corresponding probabilistic databases. This is the only viable solution for querying autonomous databases where the mediator does not have capabilities to modify the underlining deterministic databases to probabilistic ones; and a solution that promises efficiency for top k query processing. QPIAD implements this solution by rewriting the original query to a set of probabilistic queries. Answers to these probabilistic queries in the original incomplete database in Table 4(a) will be the same as the answers to the original user query on the probabilistic database in Table 4(b). Further, these rewritten queries are ranked and issued in the order of their probability, thereby generating and presenting the results in the order of probability.

The connection between QPIAD and the DI model of probabilistic databases also provides additional computational justification to some of the practical design choices made in QPIAD. Specifically, as we have discussed, QPIAD supports conjunctive selections, joins across different data sources, projections after selections (but not joins), and aggregation functions SUM and COUNT. The practical motivation in supporting these queries is that they are popularly used in application scenarios involving querying autonomous web databases by lay users. An additional theoretical justification for this choice is that this set of queries corresponds well to the set of queries that are known to be polynomial time for DI databases [37]. While QPIAD architecture can be extended to support more general queries,

should the need arise, such a step may lead to higher computational complexity of query processing.

A recent work [11] studies the problem of inferring query answer probabilities when querying a mediated view of deterministic or probabilistic databases by exploiting the statistical information of the view. Our work addresses the problem of querying a mediated view over deterministic incomplete databases in the absence of such statistical information. We focus on the unique technical challenges of efficient and accurate learning of data dependencies, and effective rewriting of the original deterministic query to probabilistic queries in order to retrieve relevant answers in the order of their expected relevance without materializing the equivalent probabilistic databases.

Querying Inconsistent Databases: Work on handling inconsistent databases also has some connections. While most approaches for handling inconsistent databases are more similar to the "possible worlds approaches" used for handling incompleteness (e.g. [4]), some recent work (e.g. [1]) uses probabilistic approaches for handling inconsistent data.

Query Reformulation & Relaxation: There are work on query reformulation and query relaxation to handle the cases where the original query has an empty result or a small size of query result [36,35]. Our work has a different goal: retrieving and ranking tuples that have missing values on constrained attributes and yet are relevant to the user query, which requires fundamentally different query rewriting techniques.

Learning Missing Values: There has been a large body of work on missing values imputation [12,39,40,44,3]. Common imputation approaches include substituting missing data values by the mean, the most common value, default value of the attribute in question, or using k-Nearest Neighbor [3], association rules [44], etc. Another approach used to estimate missing values is *parameter estimation*. Maximum likelihood procedures that use variants of the Expectation-Maximization algorithm [12,39] can be used to estimate the parameters of a model defined for the complete data. In this paper, we are interested not in the standard imputation problem but a variant that can be used in the context of query rewriting. In this context, it is important to have schema level dependencies between attributes as well as distribution information over missing values.

Another related problem is entity resolution, or deduplication, which recognizes different tuples that correspond to the same real world entities using textual similarity and set of constraints, including aggregation constraints [8], groupwise constraints, etc.. While this paper mines approximate functional dependencies for missing value imputation; and instead of "repairing" the database, we focus on query rewriting techniques for retrieving relevant tuples.

Mining Approximate Functional Dependencies: There is some existing work on mining FDs [46,45,32,31,38] and on mining AFDs with high confidence level [19,22]. Similar to [19] and [38], we take a candidate-generate-and-test approach for AFD mining. Different from all existing work, we propose a metric called *specificity* to quantify whether an AFD is useful for generating rewritten queries to retrieve relevant incomplete tuples in the context of QPIAD framework. Then we develop a novel AFD mining technique which by proactively pruning AFDs that fail specificity threshold, achieves performance speedup.

Conditional Functional Dependencies (CFDs) are deterministic dependencies holding true only for certain values of the determining attributes [6,14]. [15] presents techniques to mine CFDs. It is an open question how to exploit CFDs for querying incomplete databases, which we plan to investigate.

8 Conclusion

Incompleteness is inevitable in autonomous web databases. Retrieving highly relevant possible answers from such databases is challenging due to the restricted access privileges of mediator, limited query patterns supported by autonomous databases, and sensitivity of database and network workload in web environment. We developed a novel query rewriting technique that tackles these challenges. Our approach involves rewriting the user query based on the knowledge of database attribute correlations. The rewritten queries are then ranked by leveraging attribute value distributions according to their likelihood of retrieving relevant possible answers before they are posed to the databases. We discussed rewriting techniques for handling queries containing selection, joins and aggregations. To support such query rewriting techniques, we mine attribute correlations in the form of AFDs and the value distributions in the form of AFD-enhanced classifiers, as well as query selectivity from a small sample of the database itself. To order to handle data sources with large attribute sets, we developed a novel technique for AFD mining that utilizes effective pruning strategies while maintains comparable high quality. We also discuss the semantics of our approach based on its relationship to query evaluation on probabilistic databases. Comprehensive experiments demonstrated the effectiveness of our query processing and knowledge mining techniques.

As we mentioned, part of the motivation for handling incompleteness in autonomous databases is the increasing presence of databases on the web. In this context, a related issue is handling query imprecision–most users of online databases tend to pose imprecise queries which admit answers with varying degrees of relevance (c.f. [36]). In our

ongoing work, we are investigating the issues of simultaneously handling data incompleteness and query imprecision.

References

1. P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, 2006.
2. L. Antova, C. Koch, and D. Olteanu. 10 10 6 worlds and beyond: Efficient representation and processing of incomplete information. *Proc. ICDE*, 2007.
3. G. E. A. P. A. Batista and M. C. Monard. A study of k-nearest neighbour as an imputation method. In *HIS*, 2002.
4. L. Bertossi. Consistent query answering in databases. *ACM SIGMOD Record*, 35(2):68–76, 2006.
5. A. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 1997.
6. P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755. IEEE, 2007.
7. D. Burdick, P. Deshpande, T. Jayram, R. Ramakrishnan, and S. Vaithyanathan. OLAP over uncertain and imprecise data. *The VLDB Journal*, 16(1):123–144, 2007.
8. S. Chaudhuri, A. D. Sarma, V. Ganti, and R. Kaushik. Leveraging aggregate constraints for deduplication. In *SIGMOD Conference*, pages 437–448, 2007.
9. R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD Conference*, 2003.
10. N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12. ACM New York, NY, USA, 2007.
11. N. N. Dalvi and D. Suciu. Answering queries from statistics and probabilistic views. In *VLDB*, pages 805–816, 2005.
12. A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via em algorithm. In *JRSS*, pages 1–38, 1977.
13. A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, 2001.
14. W. Fan. Dependencies revisited for improving data quality. In *PODS*, pages 159–170, 2008.
15. L. Golab, H. J. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1):376–390, 2008.
16. T. J. Green and V. Tannen. Models for incomplete and probabilistic information. *IEEE Data Eng. Bull.*, 29(1):17–24, 2006.
17. R. Gupta and S. Sarawagi. Creating Probabilistic Databases from Information Extraction Models. *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, 32(2):965, 2006.
18. D. Heckerman. A tutorial on learning with bayesian networks, 1995.
19. Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*, 42(2):100–111, 1999.
20. Y. Huhtala, J. Krkkinen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *ICDE Conference*, pages 392–401, 1998.
21. I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658, New York, NY, USA, 2004. ACM.
22. I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *SIGMOD Conference*, pages 647–658, 2004.
23. T. Imieliski and J. Witold Lipski. Incomplete information in relational databases. *Journal of ACM*, 31(4):761–791, 1984.
24. Z. Ives, A. Halevy, and D. Weld. Adapting to source properties in processing data integration queries. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 395–406, 2004.
25. A. Kalavagattu. Mining Approximate Functional Dependencies as Condensed Representations of Association Rules. Master's thesis, Arizona State University, 2008. http://rakaposhi.eas.asu.edu/Aravind-MSThesis.pdf.
26. H. Khatri. Query Processing Over Incomplete Autonomous Web Databases. Master's thesis, Arizona State University, 2006. http://rakaposhi.eas.asu.edu/hemal-thesis.pdf.
27. J. Kivinen and H. Mannila. Approximate dependency inference from relations. In *ICDT Conference*, 1992.
28. D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *KRDB Workshop*, 2002.
29. L. Libkin. Data exchange and incomplete information. In *PODS*, pages 60–69, 2006.
30. W. Lipski. On semantic issues connected with incomplete information databases. *ACM TODS*, 4(3):262–296, 1979.
31. S. Lopes, J. Petit, and L. Lakhal. Functional and approximate dependency mining: database and FCA points of view. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2):93–114, 2002.
32. S. Lopes, J.-M. Petit, and L. Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *EDBT '00: Proceedings of the 7th International Conference on Extending Database Technology*, pages 350–364, London, UK, 2000. Springer-Verlag.
33. J. Madhavan, A. Halevy, S. Cohen, X. Dong, S. Jeffery, D. Ko, and C. Yu. Structured Data Meets the Web: A Few Observations. *IEEE Data Eng. Bull.*
34. T. Mitchell. *Machine Learning.* McGraw Hill, 1997.
35. I. Muslea and T. J. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.
36. U. Nambiar and S. Kambhampati. Answering Imprecise Queries over Autonomous Web Databases. *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)-Volume 00*, 2006.
37. N. Nilesh and D. Suciu. Efficient query evaluation on probabilistic databases. *Proc. of VLDB Conference*, pages 864–875, 2004.
38. N. Novelli and R. Cicchetti. FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies. *LECTURE NOTES IN MATHEMATICS-SPRINGER VERLAG-*, pages 189–203, 2000.
39. M. Ramoni and P. Sebastiani. Robust learning with missing data. *Mach. Learn.*, 45(2):147–170, 2001.
40. D. B. R. Roderick J. A. Little. *Statistical Analysis with Missing Data,Second edition*. Wiley, 2002.
41. A. D. Sarma, O. Benjelloun, A. Y. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.
42. D. Suciu and N. Dalvi. Tutorial: Foundations of probabilistic answers to queries. In *SIGMOD Conference*, 2005.
43. J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.
44. C.-H. Wu, C.-H. Wun, and H.-J. Chou. Using association rules for completing missing data. In *HIS Conference*, 2004.
45. C. M. Wyss, C. Giannella, and E. L. Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In *DaWaK*, pages 101–110, 2001.
46. H. Yao and H. Hamilton. Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2):197–219, 2008.