QIRANA: a Framework for Scalable Query Pricing

Shaleen Deep
University of Wisconsin-Madison
Madison, USA
shaleen@cs.wisc.edu

Paraschos Koutris
University of Wisconsin-Madison
Madison, USA
paris@cs.wisc.edu

ABSTRACT

Users are increasingly engaging in buying and selling data over the web. Facilitated by the proliferation of online marketplaces that bring such users together, *data brokers* need to serve requests where they provide results for user queries over the underlying datasets, and price them fairly according to the information disclosed by the query. In this work, we present a novel pricing system, called QIRANA, that performs *query-based data pricing* for a large class of SQL queries (including *aggregation*) in real time. QIRANA provides prices with formal guarantees: for example, it avoids prices that create arbitrage opportunities. Our framework also allows flexible pricing, by allowing the data seller to choose from a variety of pricing functions, as well as specify relation and attribute-level parameters that control the price of queries and assign different value to different portions of the data. We test QIRANA on a variety of real-world datasets and query workloads, and we show that it can efficiently compute the prices for queries over large-scale data.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Relational Databases

Keywords

Data Pricing; Arbitrage; Query Determinacy

1. INTRODUCTION

The last decade has seen an explosion of data being collected from a variety of sources and across a broad range of areas. Many companies, including Bloomberg [7], Twitter [29], Lattice Data [19], DataFinder [11], and Banjo [4] collect such data, which then sell as structured (relational) datasets. Today, these datasets are often sold through online data markets, which are web platforms for buying and selling data: examples include Microsoft Azure Marketplace [32], BDEX [5], and QLik DataMarket [24]. But such datasets are often prohibitively expensive, since the sellers put a lot of effort into obtaining, extracting, integrating, cleaning, and transforming the data into a relational format. Existing data markets and data sellers either force users to buy the whole dataset, or support very simplistic pricing mechanisms (e.g., they price according to output size). This means that valuable data is often not accessible to lay users, scientists, or entities with limited budgets. In order to facilitate access to data for more users and expand the market for data-selling companies and marketplaces, we need to tailor the purchase of data to the user's needs, by charging the user according to the query workload rather than the full dataset: this is called *query-based pricing*. In this paper, *we develop and evaluate an end-to-end framework for query-based pricing, and study in detail the tradeoffs and design choices for building a comprehensive query-based pricing system.*

Previous work in this area studied the problem of pricing queries both from a theoretical [16, 20, 21] and practical [18, 30] point of view. This work identified a key principle in designing a pricing function, which is that it must not exhibit *arbitrage*: it should not be possible for a buyer to acquire the desired query for a cheaper price by combining other query results. Pricing functions that exhibit arbitrage will lead to inconsistent pricing for queries, and can cause information leakage. The pricing framework from [16, 18] works by requiring that the seller sets fine-grained price points (prices to simple selection queries with equality predicates) that will be used as a guide to price the other queries. In this setting, pricing join queries is in general NP-hard [16], but for a subclass of join queries it can be done in polynomial time. The QueryMarket prototype [18] showed that by formulating the pricing problem as an ILP and using off-the-shelf ILP solvers, one can price also the hard join queries, albeit for small datasets. More recent work [30] considers simplistic pricing schemes, which assign a price according to the number of tuples that contribute to the answer, and provide no guarantees against arbitrage.

The above solutions proposed for a pricing framework all have limitations. QueryMarket cannot handle queries with aggregation or grouping. Even for simple join queries, pricing is not scalable to even medium-sized datasets: for instance, [18] reports that computing a join query over a relation of about 1,000 tuples takes about one minute. Other pricing schemes that assign a price to a query according to its output size, such as [30], are prone to arbitrage attacks. Thus, to the best of our knowledge, there is no existing query-based pricing framework that prices a wide spectrum of SQL queries in real time, while providing formal guarantees about the properties of the pricing function.

SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3064017

We next provide a motivating example.

EXAMPLE 1.1. *Consider the* TWITTER *database containing two relations* User *and* Tweet *in Figure 1. Assume that the data seller fixes a price of* \$100 *for the whole dataset and wants to price each individual relation at* \$50. *Consider a data analyst, Alice, who wants to perform statistical analysis on demographics of twitter users. Alice cannot afford to purchase the whole dataset, but instead she would like to pay according to the information content of a sequence of analytics queries that she will ask over time. To begin with, she asks about the number of female users on Twitter, by posing the query* Q_1 = `SELECT count(*) FROM User WHERE gender = 'f'`. *Alternatively, she could ask the query* Q_2 = `SELECT gender, count(*) FROM User GROUP BY gender` *to obtain the number of male and female users. If* Q_1 *costs* \$7 *and* Q_2 *costs* \$5, *Alice has an arbitrage opportunity, because she can get the same data (plus additional information) from the cheaper query* Q_2. *The broker can avoid this arbitrage by pricing* $p(Q_2) \geq p(Q_1)$, *for example at* \$8.

Alice decides to buy Q_2. *Next, Alice wants to find out the average age of all Twitter users. She can ask for the query* Q_3 = `SELECT AVG(age) FROM User`, *which costs* \$11. *However, a different way to find the average age is to ask for the sum of age of all users and combine it with number of users known from* Q_2. *Let* Q_4 = `SELECT SUM(age) FROM User` *and say that broker charges* \$2 *for this query. Notice that* Q_3 *is more expensive than asking for both* Q_2 *and* Q_4. *We again have an arbitrage situation: to avoid this case, the broker needs to make sure that* $p(Q_3) \leq p(Q_2) + p(Q_4)$.

Alice decides to purchase Q_3 *as well. Next, she asks* Q_5 = `SELECT COUNT(*) FROM User WHERE gender='m'`, *which has overlapping content with* Q_2. *If the broker does not keep track of the purchase history, Alice will pay for the same information twice and may go over her budget. Therefore, the broker needs to take into consideration Alice's query purchase history when pricing. In such a history-aware scenario,* Q_5 *should be free.*

Our Contribution. In this work, we describe, implement and evaluate a system for query-based pricing, called QIRANA, that formally guarantees certain desirable properties. Our system works as an intermediate layer (*broker*) between the DBMS and the data buyer. Once a buyer issues a query Q on the database D, our pricing framework computes the output $Q(D)$, and charges the user with a price $p(Q, D)$. QIRANA has the following characteristics:

- It allows the data seller to choose from several pricing functions, all of which provably avoid any *arbitrage* opportunities (Section 2.3).
- It supports efficiently *history-aware pricing*, where each buyer is charged not only according to the current query, but also according to past queries. This feature prevents the buyer from overpaying when she has already acquired all or part of the desired information.
- It requires the seller to specify only a single price for the whole dataset, without the need to provide multiple fixed price points. In this case, our framework treats each part of the dataset as being equally valuable. It also provides mechanisms so that the seller can tune the prices by specifying explicitly which relations or attributes should be priced differently (Section 3.3).
- It provides an *efficient and scalable* implementation for a particular pricing function, called *weighted coverage*, over a large class of SQL queries. In Section 5, we show that QIRANA can compute the price of queries over the `TPC-H` and `SSB` datasets of scale factor 1 with a low overhead. QIRANA is, to the best of our knowledge, the first query-based pricing system that allows real-time pricing with formal guarantees.
- It can be deployed on top of any DBMS without any modification of the underlying database system.

User

uid	name	gender	age
1	John	m	25
2	Alice	f	13
3	Bob	m	45
4	Anna	f	19

Tweet

tid	uid	time	location
1	3	23:29	CA
2	3	23:29	WA
3	1	23:30	OR
4	2	23:31	CA

Figure 1: The database for the running example.

Technical Overview. The key idea behind our pricing framework is as follows. From the point of view of the buyer (Alice), there initially exists a set of *possible databases* $\mathcal{I}$, which captures the common knowledge about the data (schema, primary keys, data domain, etc.). Whenever Alice issues a query Q over the database $\mathscr{D}$, she learns more information, and can safely remove from $\mathcal{I}$ any database D such that $Q(\mathscr{D}) \neq Q(D)$, thereby shrinking the number of possible databases. The price assigned to Q can then be formulated as a function of how much $\mathcal{I}$ shrinks. It turns out that there are several choices of such a function that lead to arbitrage-free pricing functions [13].

Unfortunately, it is infeasible to keep track of all possible databases, since their number can be astronomically large. Our main observation is that instead of considering all databases in $\mathcal{I}$, it suffices to look only at a small subset of $\mathcal{I}$ (which we call the *support set*): if we choose it carefully, it can approximate very well the amount of information disclosed by returning $Q(\mathscr{D})$. In particular, QIRANA keeps track of a subset $\mathcal{S}$ of *neighboring databases* of $\mathscr{D}$ in $\mathcal{I}$: these are the databases that differ from $\mathscr{D}$ only in one or two rows. For each neighboring database $D \in \mathcal{S}$, our framework needs to compute $Q(D)$, and check whether it agrees with $Q(\mathscr{D})$.

This approach can compute the price for any query, but it requires that we compute Q over $|\mathcal{S}|$ databases, which we have to keep in the database along with $\mathscr{D}$. Since this is computationally and memory-wise expensive, we model each $D \in \mathcal{S}$ as an *update* applied to $\mathscr{D}$ (we can do this because $\mathscr{D}, D$ are neighbors). To further speed up the execution of the query Q on the updated database, we propose (Section 4) optimizations that allow us to check whether a given update changes the output of the query, without needing to run the query again and again on the full database. Our proposed method can be of independent interest, since it can be applied to any view maintenance setting.

2. THE PRICING FRAMEWORK

In this section, we formally define the pricing framework and introduce the notation used throughout the paper.

2.1 The Basics of Pricing

A *data seller* (Bob) wants to sell a database instance $\mathscr{D}$ through a data market, which functions as the *broker*. For instance, suppose that Bob offers for sale the database instance in Figure 1. A *data buyer* (Alice) can purchase information from the dataset by issuing queries in the form of a *query bundle* $\mathbf{Q} = (Q_1, \ldots, Q_n)$, which is a vector of queries.

The queries can be formed in any query language, but for this work we will focus on SQL queries. We denote the output of the query bundle by $\mathbf{Q}(\mathscr{D}) = (Q_1(\mathscr{D}), \dots Q_n(\mathscr{D}))$.

The database has a fixed schema $\mathbf{R} = (R_1, \dots, R_k)$, which is known to Alice. In addition to the schema, Alice possibly has more (public) knowledge about the database, such as functional or other types of dependencies (e.g. primary keys, foreign keys), domain constraints, or bounds on the size of the relations. For example, Alice knows that User.uid is a primary key for User, Tweet.tid a primary key for Tweet, and that there exists a foreign key dependency from Tweet.uid to User.uid. Let $\mathcal{I}$ denote the set of all *possible databases* that conform to the above constraints.

A *pricing function* $p(\mathbf{Q}, D)$ takes as input a query bundle $\mathbf{Q}$ and a database instance $D \in \mathcal{I}$ and assigns to it a price, which is a number in $\mathbb{R}_+$. Ideally, the price assigned should be representative of the information that Alice learns from obtaining the result $\mathbf{Q}(\mathscr{D})$. Depending on how the price is computed, there exists different types of pricing schemes. We adopt the terminology as introduced in [21]:

- **Instance-independent** (QPS): The price depends only on $\mathbf{Q}$, so $p(\mathbf{Q}, D) = p(\mathbf{Q}, D')$ for any $D, D' \in \mathcal{I}$.
- **Answer-dependent** (APS): The price depends on $\mathbf{Q}$ and the output $E = \mathbf{Q}(D)$. In other words, $p(\mathbf{Q}, D) = p(\mathbf{Q}, D')$ for any $D, D' \in \mathcal{I}$ such that $\mathbf{Q}(D) = \mathbf{Q}(D')$.
- **Data-dependent** (DPS): The price depends on both $\mathbf{Q}$ and the database D.

We also say that $\mathbf{Q}_1$ *determines* $\mathbf{Q}_2$, denoted $\mathbf{Q}_1 \twoheadrightarrow \mathbf{Q}_2$, if whenever $\mathbf{Q}_1(D) = \mathbf{Q}_1(D')$ then $\mathbf{Q}_2(D) = \mathbf{Q}_2(D')$ for all $D, D' \in \mathcal{I}$.[1] If $\mathbf{Q}_1$ determines $\mathbf{Q}_2$, then we can always compute $\mathbf{Q}_2$ from the output $\mathbf{Q}_1(D)$ without access to the underlying database D. For example, coming back to Example 1.1, query Q_2 determines query Q_1. Similarly, we say that $\mathbf{Q}_1$ determines $\mathbf{Q}_2$ under D, denoted $D \vdash \mathbf{Q}_1 \twoheadrightarrow \mathbf{Q}_2$, if whenever $\mathbf{Q}_1(D) = \mathbf{Q}_1(D')$ then $\mathbf{Q}_2(D) = \mathbf{Q}_2(D')$ for all $D' \in \mathcal{I}$. $\mathbf{Q}_1 \twoheadrightarrow \mathbf{Q}_2$ implies that for every database D we have $D \vdash \mathbf{Q}_1 \twoheadrightarrow \mathbf{Q}_2$ but not the other way around.

2.2 Pricing Desiderata

When we design a pricing function, there exist several desirable properties we would like to guarantee, as well as take into account certain practical considerations. We next discuss the various desiderata for a pricing function, both from a buyer's and seller's perspective. These desiderata form a rich design space, with different trade-offs:

Information Arbitrage-Free Pricing We say that a pricing function is *weakly information arbitrage-free* if whenever $\mathbf{Q}_1 \twoheadrightarrow \mathbf{Q}_2$ we have $p(\mathbf{Q}_2, D) \leq p(\mathbf{Q}_1, D)$ for every D. We similarly say that it is *strongly information arbitrage-free* if whenever $D \vdash \mathbf{Q}_1 \twoheadrightarrow \mathbf{Q}_2$ we have $p(\mathbf{Q}_2, D) \leq p(\mathbf{Q}_1, D)$. To see why the seller may require that a pricing function has no information arbitrage, consider the example where the price of a bundle $\mathbf{Q}_1$ is more than the price of bundle $\mathbf{Q}_2$ and $\mathbf{Q}_1$ reveals a subset of information than $\mathbf{Q}_2$. Then, Alice can choose to purchase $\mathbf{Q}_2$ and get the information of $\mathbf{Q}_1$ for a price lower than intended. The strong information arbitrage condition is generally more desirable than the weak, since it is possible that $\mathbf{Q}_1 \not\twoheadrightarrow \mathbf{Q}_2$, but $\mathscr{D} \vdash \mathbf{Q}_1 \twoheadrightarrow \mathbf{Q}_2$ for the particular database that is for sale. In this case, if $p(\mathbf{Q}_1, \mathscr{D}) < p(\mathbf{Q}_2, \mathscr{D})$, the buyer will have an arbitrage opportunity. Of course, since the underlying database $\mathscr{D}$ is unknown to the buyer, this opportunity will occur only by chance and not by following a particular strategy.

The requirement of strong information arbitrage-freeness constraints our choice for a pricing function: any non-constant pricing function that is strongly information arbitrage-free must also be APS ([13], Theorem 3.8), i.e. its price must depend on both $\mathbf{Q}$ and the output $\mathbf{Q}(\mathscr{D})$. Pricing schemes that assign a price depending on the output size, or the provenance of the answer [30] offer no protection against arbitrage. As a simple example, consider the query Q `= SELECT count(*) FROM R`. A provenance-based approach would assign a full price to query Q since all tuples contribute to the output. Even if the provenance is extended to an attribute level, there exist cases where boolean queries that check if database is empty cost substantially (see [21]).

Bundle Arbitrage-Free Pricing Consider the case where Alice wants to obtain the answer for the bundle $\mathbf{Q} = \mathbf{Q}_1 \| \mathbf{Q}_2$, where $\|$ denotes vector concatenation. Instead of asking for $\mathbf{Q}$ all at once, Alice can create two separate accounts, use one to ask for $\mathbf{Q}_1$ and the other to ask for $\mathbf{Q}_2$. To avoid this issue, the seller must ensure that $p(\mathbf{Q}, D) \leq p(\mathbf{Q}_1, D) + p(\mathbf{Q}_2, D)$ for all $D \in \mathcal{I}$. In this case, we say that the pricing function is *bundle arbitrage-free*. Ensuring a bundle arbitrage-free pricing APS function can lead to disproportionately high prices for some queries, as shown in [13]. In particular, an APS pricing function that exhibits no bundle arbitrage will price any query, even one that touches a small part of the data, to at least half the full price of the dataset for many database instances. On the other hand, presence of bundle arbitrage leads to arbitrage opportunities. This gives an important design trade-off that a data seller needs to consider.

History-Aware Pricing In the context of a data market, Alice may want to issue multiple queries $Q_1, \dots, Q_k$ over time, some which may contain repeated information. In such a scenario, the data seller has two choices: (a) price each query individually, or (b) compute the price considering the *purchase history* of queries for that particular buyer. If the seller chooses to price queries individually, then Alice will have to pay the amount $\sum_{i=1}^{k} p(Q_i, \mathscr{D})$, in which she could be charged multiple times for the same information. On the other hand, if the seller opts for history-aware pricing, then after issuing the first k queries, Alice will be charged with the amount $p((Q_1 \| Q_2 \dots \| Q_k), \mathscr{D})$; in other words, the whole sequence of the queries will be priced as a bundle. History-aware pricing can be implemented using different techniques, for example by issuing refunds [30], or by careful bookkeeping of what the buyer has already purchased.

Customizability The data seller should ideally be able to customize the pricing function as much or as less as possible. A customizable pricing function should be able to produce varied prices when the seller offers a single price point (the price of the whole dataset), and also be able to incorporate price suggestions from the seller. For example, Bob can require that asking for all of relation User costs \$100, but relation Tweet only \$10. Or he may want to price a specific attribute to a higher price than another attribute. Previous pricing frameworks allow for fine-grained tuning: the seller can assign a price to each tuple in the dataset [30], or to

[1] We should note that this definition of determinacy is slightly different from the standard definition, where D, D' can be any database instances (not necessarily from $\mathcal{I}$).

Pricing Function	Support Set	Type	Info. Arbitrage Free	Bundle Arbitrage Free
coverage (1)	UNIFORM	APS	strong	✓
	NBRS	DPS	strong	✓
unif. entropy gain (2)	UNIFORM	APS	strong	✗
	NBRS	DPS	strong	✗
Shannon entropy (3)	UNIFORM	QPS	weak	✓
	NBRS	DPS	weak	✓
q-entropy (4)	UNIFORM	QPS	weak	✓
	NBRS	DPS	weak	✓

Table 1: Properties for the pricing functions discussed in Section 2.3 along with different choices of support sets (random uniform vs random neighborhood).

each selection query on a specific attribute [18].

Scalability Efficient computation of the price is a key requirement for any pricing framework that intends to be deployed in practice. Ideally, a pricing system must compute the price with a small overhead relative to the time necessary to compute the query, and scale effectively to large datasets.

Price Leakage The price assigned to a query bundle $\mathbf{Q}$ can potentially be exploited by a user to learn more information about $\mathscr{D}$ than what she can learn from $\mathbf{Q}(\mathscr{D})$; we call this phenomenon *price leakage*. A QPS pricing function never leaks any information, since by definition the price depends only on the query $\mathbf{Q}$. An APS pricing function also leaks no information if we require that the user buys the view if the price is disclosed: this mechanism is called *up-front pricing* (see [21]). If we want to reveal the price without returning the result, then an APS pricing function can disclose information to the data buyer, since the price depends on the query output $\mathbf{Q}(\mathscr{D})$. A DPS pricing function can reveal additional information about the underlying database even if we restrict to up-front pricing. Even though DPS and APS pricing functions can leak information, it could be possible to provide worst-case guarantees that bound the price leakage. We leave this subject for future work; for this paper, we will use the simple guide QPS > APS > DPS to compare pricing schemes in terms of price leakage.

Information-Aware Pricing An important desideratum of a pricing function is that it captures the amount of information disclosed by a query. For instance, a pricing function that assigns a constant price to every query bundle satisfies all the above desiderata in this section, but it does not capture correctly the amount of information disclosed. To capture this property experimentally, we propose a simple benchmark in Section 2.4.

2.3 Constructing Pricing Functions

Let $\mathbf{Q}$ be a query bundle issued by Alice; our task in hand is to assign a price. Constructing a pricing function in our framework consists of two components: (a) choose a subset $\mathcal{S} \subseteq \mathcal{I}$ called the *support set* of size $S = |\mathcal{S}|$, and (b) apply a function on the elements of the support set depending on the database D and the query bundle $\mathbf{Q}$.

Choosing a Function Once Alice obtains the query output $E = \mathbf{Q}(\mathscr{D})$, she knows that any database $D \in \mathcal{I}$ for which $\mathbf{Q}(D) \neq E$ is not possible anymore. The set of such databases is called the *conflict set* of $\mathbf{Q}$:

$$\overline{C}_{\mathbf{Q}}(E) = \{D \in \mathcal{I} \mid \mathbf{Q}(D) \neq E\}$$

We can now compute a price for $\mathbf{Q}$ by applying a set function $f : 2^{\mathcal{I}} \setminus \{\mathscr{D}\} \to \mathbb{R}_+$ on $\overline{C}_{\mathbf{Q}}(E)$, after restricting it to the support set: $p(\mathbf{Q}, D) = f(\overline{C}_{\mathbf{Q}}(E) \cap \mathcal{S})$. A set function f is *monotone* if for sets $A \subseteq B$ we always have $f(A) \leq f(B)$, and *subadditive* if for every set A, B we have $f(A) + f(B) \geq f(A \cup B)$. By picking f to be monotone and subadditive, we can guarantee that pricing function exhibits no information and bundle arbitrage [13]. In this work, we will focus on two functions which will be of practical interest.

The *weighted coverage* function initially assigns a weight w_i to each $D_i \in \mathcal{S}$, and computes the price as the weighted sum of disagreements:

$$p^{wc}(\mathbf{Q}, \mathscr{D}) = \sum_{i:\mathbf{Q}(D_i)\neq E} w_i \tag{1}$$

The *uniform entropy gain* function models the price as the gain in entropy if each database in $\mathcal{S}$ is assigned the same probability:

$$p^{ueg}(\mathbf{Q}, \mathscr{D}) = \frac{\log\left|\overline{C}_{\mathbf{Q}}(E) \cap \mathcal{S}\right|}{\log|\mathcal{S}|} \tag{2}$$

The uniform entropy gain function has no information arbitrage, but it does exhibit bundle arbitrage. The weighted coverage function has no information or bundle arbitrage.

A different class of pricing functions can be constructed by looking at how $\mathbf{Q}$ partitions the support set $\mathcal{S}$. Let $\mathcal{P}_{\mathbf{Q}}$ be the set of blocks (equivalence classes) in the partition induced by the following equivalence relation: $D \sim D'$ iff $\mathbf{Q}(D) = \mathbf{Q}(D')$. We again assign to each database $D_i \in \mathcal{S}$ a weight (probability) w_i such that $\sum_i w_i = 1$. For a block $B \in \mathcal{P}_{\mathbf{Q}}$, define $w_B = \sum_{i:D_i \in B} w_i$. Using the above formulation, we can construct entropy-based pricing functions.

The *Shannon entropy* function computes the price as the entropy of the query output:

$$p^{H}(\mathbf{Q}, \mathscr{D}) = -\sum_{B\in\mathcal{P}_{\mathbf{Q}}} w_B \log w_B \tag{3}$$

The *q-entropy* function (or Tsallis entropy) for $q = 2$ computes the price as a different entropy measure:

$$p^{T}(\mathbf{Q}, \mathscr{D}) = \sum_{B\in\mathcal{P}_{\mathbf{Q}}} w_B \cdot (1 - w_B) \tag{4}$$

For both types of entropy, it has been shown [13] that they are information arbitrage-free and bundle arbitrage-free.

We assume that the data seller has provided a price P for the whole dataset. Since the whole dataset can be retrieved by the query bundle $\mathbf{Q}_{all}$ that returns all the relations, we scale our pricing functions such that $p(\mathbf{Q}_{all}, \mathscr{D}) = P$ (scaling preserves all arbitrage-free guarantees).

Choosing a Support Set Choosing the support set to be $\mathcal{S} = \mathcal{I}$ makes pricing infeasible, since in general $\mathcal{I}$ is a complex and large space. The naive method of running the query $\mathbf{Q}$ on all databases in $\mathcal{I}$ is computationally infeasible, even in the case where $\mathcal{I}$ can be concisely described (e.g. as a a tuple-independent probabilistic database). Indeed, computing the size of the conflict set can be reduced to computing a query over a probabilistic database, which is in general a #P-hard problem, even for the class of join queries [10]. In fact, even checking whether a view agress with $Q(\mathscr{D})$ or not is equivalent to the problem of *view consistency*, which is NP-hard for join queries [1]. Computation becomes even harder when we consider entropy-based pricing functions.

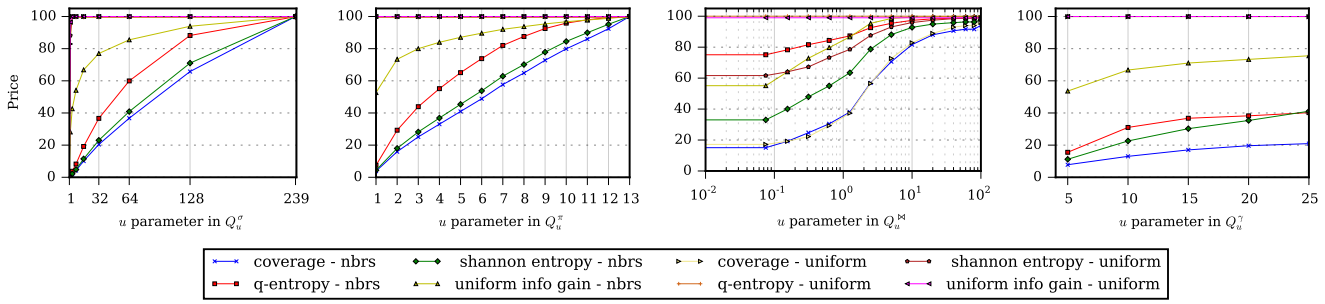


Figure 2: Benchmarking the price behavior for queries $Q_u^{\bowtie}$, Q_u^{γ}, Q_u^{σ} and Q_k^{π} for the `world` dataset. Support set size is $S = 1000$.

To circumvent this issue, our pricing framework picks a support set $\mathcal{S} \subseteq \mathcal{I}$ that is much smaller relative to $\mathcal{I}$. We will examine two methods of choosing a support set:

- **Random Uniform** (UNIFORM): This approach generates a support set by sampling uniformly at random from $\mathcal{I}$ a set of size S. As we will see shortly, this method, although intuitive, is actually *not* a good solution for our proposed pricing functions.
- **Random Neighborhood** (NBRS): Let G be an undirected graph with vertex set $\mathcal{I}$ and edge set as follows: two databases D_1, D_2 are connected by an edge if D_1 and D_2 differ in one or more attributes of a single tuple. In other words, D_1, D_2 are "similar" databases. Let $\mathcal{N}^i(D)$ denote the *neighborhood* of database D in G within distance at most i. The second method for constructing a support set samples databases from the set $\mathcal{N}^2(\mathscr{D})$. The intuition behind this choice is similar to differential privacy: looking at databases that are close to the underlying database makes the pricing function more sensitive to the information disclosed.

2.4 Discussion

We summarize the properties of our pricing functions in Table 1. The choice of the type of support set is orthogonal to the choice of the pricing function. Note that choosing a random neighborhood around $\mathscr{D}$ as a support set implies that the pricing function depends (implicitly) on the database $\mathscr{D}$, and thus the pricing function becomes data-dependent (DPS).

In order to understand how the pricing functions in Table 1 capture information, we construct a simple benchmark to test the resulting prices, using the `Country` relation in the `world` [33] dataset. We add a new candidate key called `ID` to the relation. Consider the parametrized query Q_u^{σ}:

Q_u^{σ}:`SELECT * FROM Country WHERE ID < u ;`

As the parameter `u` ranges from 1 to 240, the cardinality of the output also ranges from 0 to 239 linearly. If the data has uniform value, an information-aware pricing function should capture this semantic in that the price should increase linearly, starting from 0 and ending up to 100. We similarly define the queries Q_u^{π}, $Q_u^{\bowtie}$ and Q_u^{γ}:

Q_u^{π}:`SELECT` $A_1, \ldots, A_u$ `FROM Country ;`
$Q_u^{\bowtie}$:`SELECT * FROM Country C,CountryLanguage CL WHERE C.Code=CL.Code AND CL.Percentage<u ;`
Q_u^{γ}:`SELECT Region, AVG(LifeExpectancy) FROM Country GROUP BY Region LIMIT u;`

For Q_u^{π}, if all attributes are projected out (for `Country`, $k = 13$ excluding primary key), we obtain the full dataset and then the price of Q_{13}^{π} is the price of full dataset. If each attribute is uniformly valued, we expect again that the price decreases linearly with the number of attributes. This benchmark is by no means exhaustive, but rather intends to provide sanity conditions for a reasonable pricing scheme. We leave development of a formal metric for benchmarking as subject for future work. Figure 2 shows the behavior of the 8 different combinations of pricing functions and support sets from Table 1 on the four benchmark queries.

The first observation is that pricing functions that use a random uniform support set are not well behaved. Indeed, the price is almost always equal to the full price of \$100 even if the query touches a small part of the dataset. This phenomenon occurs because a random database is likely very far from $\mathscr{D}$, and thus the probability of disagreement is very high even if the query discloses little information. Observe that a random uniform support set also has a big memory overhead, since we have to store all instances in the database (detailed discussion in Section 3.2). For these reasons, we consider only the NBRS support set in the rest of the paper.

The second observation is that, among the pricing functions that use random neighborhood as support, weighted coverage assigns the most reasonable prices, with Shannon entropy as second. Indeed, recall that we designed our benchmark for the selection and projection query such that an information-aware price would grow linearly as the selectivity (or the number of attributes) grows, since the data has uniform value: p^{wc} and p^{H} most closely match such a linear behavior. As for the query Q_u^{γ}, one would expect that when we output all the groups, the price would be at most the price of 2 out of the 13 attributes (which is \$200/13, since the attributes are priced uniformly); however, only the weighted coverage function is close to this value.

As we will see over the next two sections, weighted coverage has two additional advantages: (a) it can be customized, and (b) it can be optimized for better performance by using view maintenance techniques. Hence, although QIRANA implements all 4 pricing functions, it is strongly recommended to use weighted coverage as the default pricing function.

3. SYSTEM ARCHITECTURE

In this section, we describe the architecture of our pricing system, depicted in Figure 3. QIRANA is built as a layer that sits on top of the DBMS. It consists of three distinct modules: the support set generator and weight assignment module are used in the preprocessing step (when the data is loaded, and the seller tunes the parameters), while the pricing module is used for interactive pricing. The running example throughout this section is the database of Figure 1.

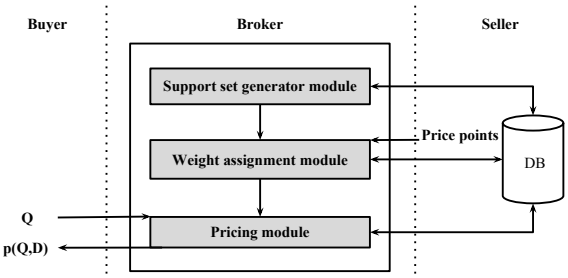


Figure 3: QIRANA architecture.

3.1 Possible DB Instances

Recall that $\mathcal{I}$ is defined as the set of all possible instances of the schema $\mathcal{R}$ that satisfy constraints such as primary key-foreign key dependencies, domain and cardinality constraints. Our pricing module mines all the schema information directly from the database. Once the schema is known, the data seller needs to specify the domain of all attributes for relations in $\mathcal{R}$. Specifying domain information is a low overhead operation for the following reasons. First, datasets available for sale undergo a lot of sanitizing before being presented in a standard format. A frequent first step for such cleaning operations is to find the domain for data standardization. Therefore, the seller already knows this information that can be directly fed into our framework. Second, most of the datasets (such as in [32]) offer metadata about the dataset as part of the preview, so that buyers can decide whether the dataset serves their requirements. The data seller can also choose to specify no domain, in which case our system defaults to using the active domain of the attribute as the domain. Using the active domain or a restricted domain for any attribute does *not* compromise the arbitrage freeness guarantee. The system also restricts $\mathcal{I}$ to databases where the relations maintain the same cardinality (i.e. the size of every relation in $\mathcal{I}$ is the same).

In our running example, apart from the primary key constraints, we assume that any instance of the User and Tweet relations in $\mathcal{I}$ has exactly 4 tuples, and that the seller has specified no domain constraints (so the domain of every attribute is the active domain).

3.2 Support Set Generator

As we discussed, our pricing system implements support sets that choose *random neighbors* of $\mathscr{D}$. The advantage of choosing neighboring databases in the support set is that, in order to represent such a database D, we do not need to store all of D. Instead, we can represent it implicitly through an *update* query that, when applied on the underlying database $\mathscr{D}$, will result in D. We define two types of updates: *row* updates and *swap* updates.

A database D is generated by a *row update* if $D \in \mathcal{N}^1(\mathscr{D})$. In other words, a row update changes one or more attributes of a single tuple and replaces the attribute value with a different value from the specified domain of the particular attribute (or active domain if no domain is specified). For example, the following row update modifies the User.gender value of the tuple with key 1 to f to create a neighboring instance of $\mathscr{D}$:

```
UPDATE User SET gender = 'f' WHERE uid = 1;
```

A database D is generated by a *swap update* if we exchange one of more attributes of 2 tuples in a single relation. Intuitively, swap updates are 2 row updates where the corresponding attribute values are "swapped", and hence $D \in \mathcal{N}^2(\mathscr{D})$. For example, the following swap update sets User.age = 19 for the tuple with key 1 and User.age = 25 for the tuple with key 4:

```
UPDATE User SET age = 19 WHERE uid = 1;
UPDATE User SET age = 25 WHERE uid = 4;
```

Both row and swap updates always generate a new instance that is different from the original database $\mathscr{D}$. For every update, we also construct an *undo update* that restores the affected row(s) to the original state. Our module decides which updates to apply (and hence what elements of the support set to generate) using a random generator:

1. We choose the relation to be updated uniformly at random. In our running example, we will update User with probability 0.5 and Tweet with probability 0.5.
2. We choose each attribute (that is not the primary key) in the relation to be updated independently with probability 0.5. Thus, the number of attributes that will be modified are drawn from the geometric distribution with $p = 0.5$. We do this to be more biased to databases that will be "closer" to $\mathscr{D}$.
3. We choose whether we have a row or swap update using a predefined ratio ρ of row to swap updates. For a row (swap) update, we pick one (two) uniformly random tuple(s) from the chosen relation. To understand the effect of each type of update, consider following two queries: Q_1 = `SELECT age FROM User` and Q_2 = `SELECT AVG(age) FROM User`. Notice that if a database is generated by any sequence of swap updates, then the output for both queries remains unchanged. On the other hand, a row update on age will always modify both results. Since we do not want the price of Q_1 to be too low or the price of Q_2 to be too high, we need to determine the right ρ. We will test different values for ρ in Section 5.

After we choose an appropriate size S for the support set (we discuss in detail in Section 5 how the choice of the size effects the prices and runtime), we call the random update generator S times to obtain a sequence of updates, which is stored into a table UPDATEQUERIES that resides in the database. The corresponding undo updates are stored in another table called UNDOUPDATEQUERIES. We denote the i-th update by $\mathsf{up}^{\uparrow}[i]$ and the i-th undo update by $\mathsf{up}^{\downarrow}[i]$, where $i = 1, \ldots, S$. The i-the element of the support set is obtained as $D_i = \mathsf{up}^{\uparrow}[i](\mathscr{D})$, and also $\mathscr{D} = \mathsf{up}^{\downarrow}[i](D_i)$.

Remark. We only use row and swap updates instead of a wider class of updates (such as updates over multiple rows, insertions and deletions) for two reasons. First, as we will see in Section 4.2, choosing updates that are from $\mathcal{N}^2(\mathscr{D})$ allows us to make special optimizations where we can check efficiently if the query output has changed. Second, the set of possible instance $\mathcal{I}$ contains only databases with the same number of tuples as $\mathscr{D}$; this cardinality constraint would be violated if we applied insertion or deletion updates that would add or remove tuples from the database.

3.3 Customizing the Price

As we argued in the previous section, a desirable feature of a pricing function is that it the prices can be customized according to the seller's specifications. We show here how

the *weighted coverage* function can be customized to reflect such choices. We emphasize that our technique does not apply to the other pricing functions, which is an argument supporting using weighted coverage as the default pricing function. Recall that the weighted coverage function assigns a weight w_i to all instances $D_i \in \mathcal{S}$. The default way to assign these weights when the data seller has provided at the minimum the price P of the full database is to assign equal weight $w_i = P/|\mathcal{S}|$ to each instance in the support set.

Our framework allows the data seller to specify additional price points as a set of pairs $(\mathbf{Q}_j, p_j)$: this specifies that for any pricing function p that we compute, it must be that $p(\mathbf{Q}_j, D) = p_j$. For instance, the data seller in our running example can specify that the price of the relation `User` must be 70 using the price point $(Q_1, 70)$, where Q_1 = `SELECT * FROM User`. The seller can also provide more fine-grained specifications about the pricing function, for example by pricing the attribute `Car.age` higher (with the price point (`SELECT uid, age FROM User`, 50)), or by specifying (`SELECT * FROM User WHERE ID = 4`, 30).

QIRANA can incorporate these price points into the price by assigning different weights to the instances of the support set. To choose the weights appropriately, we solve the following (convex) entropy maximization(EM) problem:

$$\begin{aligned} \textbf{maximize} \quad & -\sum_{i=1}^{|\mathcal{S}|} w_i \cdot \log(w_i) \\ \textbf{subject to} \quad & \sum_{D_i \in \mathcal{S}} w_i = P \\ & \sum_{i:\mathbf{Q}_j(D_i) \neq \mathbf{Q}_j(\mathscr{D})} w_i = p_j, \quad j = 1, \ldots, k \\ & w_i \geq 0, \quad i = 1, ..., |\mathcal{S}| \end{aligned}$$

The first constraint encodes the fact that the price of the whole dataset is P while the second constraint encodes the price points. The objective maximizes the entropy of the weights, since under the presence of no additional information, we want to make the weights as uniform as possible (i.e. every part of the data equally valuable).

In our implementation, we solve this convex program using the SCS conic solver [23] in CVXPY [14]. We should note here that it is possible that the solver finds out that there exists no feasible solution for the given support set and the price points. In this case, we call the solver again after we (a) resample the support set, or (b) increase the size of the support set. In such cases, SCS returns a certificate of infeasibility or unboundedness of the problem. However, the algorithm returns optimal points to a modest objective accuracy when a solution exists. Hence, even though QIRANA supports arbitrary price points, in practice we restrict the price points to be queries of two forms: selections over multiple attributes, and projections. Based on our experience, we observed that with such restrictions, the solver could always find a solution within one call when the number of price points was at most 11 per relation.

3.4 Computing the Pricing Function

Here we will present how QIRANA computes the price for the pricing functions in Section 2.2. We will focus on the pricing algorithms for (a) weighted coverage, and (b) Shannon entropy, since the computation for the uniform entropy gain is similar to the weighted coverage, and for q-entropy similar to Shannon entropy. Let $\mathbf{Q} = (Q_1, Q_2, \ldots, Q_k)$ be the query bundle we want to price.

Weighted Coverage. Recall that the weighted coverage function computes the price as the weighted sum of the instances D in the support set for which $\mathbf{Q}(D) \neq \mathbf{Q}(\mathscr{D})$. QIRANA performs this computation using Algorithm 1. The algorithm initially computes the hash value of the output $\mathbf{Q}(\mathscr{D})$ (line 3). Then, it iterates over the sequence of updates: for each update $\mathsf{up}^{\uparrow}[i]$, it applies the update to create a new database D (line 5), computes the hash value of $\mathbf{Q}(D)$, and if it is different from the hash value of the original output, it adds w_i to the price (line 7). Finally, it resets the database to its original state by executing the corresponding undo update $\mathsf{up}^{\downarrow}[i]$ (line 8).

Algorithm 1: WEIGHTEDCOVERAGE($\mathbf{Q}$)

```
1 price ← 0
2 D ← 𝒟
3 OUT ← h(Q(D))
4 for i = 1, ..., |S| do
5 |   D ← up↑[i](D)
6 |   if h(Q(D)) ≠ OUT then
7 |   |   price ← price + w_i
8 |   D ← up↓[i](D)
9 return price
```

Shannon Entropy. In order to compute the price as the Shannon entropy over the support set, we use a similar algorithm, depicted in Algorithm 2. The algorithm uses a dictionary d to keep a weighted sum of the instances that have the same hash value (line 5). When it has iterated over the update sequence, it computes the entropy function based on the dictionary values. The same algorithm can be used to compute other entropy measures, with the only difference in the last line (where the output function is different).

Algorithm 2: SHANNONENTROPY($\mathbf{Q}$)

```
1 D ← 𝒟
2 dict d ← {}
3 for i = 1, ..., |S| do
4 |   D ← up↑[i](D)
5 |   d[h(Q(D))] ← d[h(Q(D))] + w_i
6 |   D ← up↓[i](D)
7 return −Σ_{b∈d.keys} h[b] · log(h[b])
```

Both pricing algorithms have to run the query on the (modified) database as many times as the size of the support set. This means that when the query runtime is slow, or the support set is too large, the time to compute the price will be slow as well. To overcome this bottleneck, we observe that for Algorithm 1 we only need to check whether the update has modified the query output, and not actually run the query. In Section 4 we exploit this observation to speed up the price computation by orders of magnitude.

3.5 Incorporating History

In the previous section, we computed the price of a query bundle while oblivious of the query history of the data buyer. We will show now how QIRANA efficiently supports history-aware pricing while keeping a small memory footprint. For simplicity of exposition, we will discuss only the case for the weighted coverage pricing function, but our approach generalizes to any other pricing function. Suppose that the data

buyer has already issued queries $Q_1, \ldots, Q_k$, which together form a bundle $\mathbf{Q}$. In history-aware pricing, the buyer has paid so far a total of $p(\mathbf{Q}, \mathscr{D})$ (instead of $\sum_j p(Q_j, \mathscr{D})$ in the history-oblivious case). When a new query Q_{k+1} is issued, our pricing system must compute the new total price as $p(\mathbf{Q}', \mathscr{D})$, where $\mathbf{Q}'$ is the bundle $(Q_1, \ldots, Q_k, Q_{k+1})$. Define:

$$\mathcal{S}_{k+1} = \{D \in \mathcal{S} \mid \mathbf{Q}(D) = \mathbf{Q}(\mathscr{D}), Q_{k+1}(D) \neq Q_{k+1}(\mathscr{D})\}.$$

Our key observation is that we express the new price as $p(\mathbf{Q}', \mathscr{D}) = p(\mathbf{Q}, \mathscr{D}) + \sum_{i:D_i \in \mathcal{S}_{k+1}} w_j$. In other words, our history-aware pricing algorithm suffices to keep track of which of the instances in $\mathcal{S}$ agree with $\mathscr{D}$ on all previous queries. Indeed, if at some point in the query history D_i disagreed with $\mathscr{D}$, then D_i has already contributed w_j to the price, so it can be safely removed from the support set. If at some point every instance has disagreed with $\mathscr{D}$, all future queries are free for the buyer because the entire dataset has already been paid for.

The detailed history-aware pricing algorithm is presented in Algorithm 3. The only bookkeeping that the algorithm needs is a bitmap b that remembers which elements of the support set have already contributed to the price (bit set to 1), so they do not need to be considered anymore. Notice that as the query history grows, computing the price becomes actually faster, since we need to consider less instances of the support set.

Algorithm 3: WEIGHTEDCOVERAGE($\mathbf{Q}$) with history

```
1  b: bitmap that keeps history information
2  price ← 0
3  D ← 𝒟
4  OUT ← h(Q(D))
5  for i = 1, ..., |S| do
6      if b[i] = 0 then
7          D ← up↑[i](D)
8          if h(Q(D)) ≠ OUT then
9              price ← price + w_i
10             b[i] = 1
11         D ← up↓[i](D)
12 return price
```

4. OPTIMIZING PRICE COMPUTATIONS

As we discussed in the previous section, the baseline pricing algorithm that runs a query Q on the (modified) database as many times as the size of the support set can be slow. To overcome this issue, we exploit the fact that it suffices to check whether a given (row or swap) update modifies the query output, instead of actually running the query. In this section we present an algorithm that can tackle this problem efficiently for a large class of SQL queries.

Formally, given a database D, an update $\mathsf{up}^\uparrow$, and a query Q, the problem is to decide whether $Q(D) \neq Q(\mathsf{up}^\uparrow(D))$, i.e. whether there exists a disagreement. This is a *view maintenance* problem, where the update is of a specific form, and we only check whether the output has changed instead of maintaining the view. We first give the intuition about the algorithm using our running example.

EXAMPLE 4.1. *Consider the query* `Q = SELECT * FROM User WHERE age > 40 AND gender = 'm'`. *Consider the row update* $\mathsf{up}_1^\uparrow$ *that sets the* `age` *of the tuple with key 3 to* 30. *Since the updated tuple does not satisfy the predicate of* Q *anymore, the database* $D' = \mathsf{up}_1^\uparrow(D)$ *disagrees with* D *for the query* Q. *Next, consider the row update* $\mathsf{up}_2^\uparrow$ *that modifies the* `gender` *of the tuple with key* 4 *to* $'m'$. *It is not straightforward to check if* $Q(\mathsf{up}_2^\uparrow(D))$ *agrees with* $Q(D)$ *because we do not know the* `age` *value for the tuple with* `uid` $= 4$. *In this case, we need additional information about all tuples* $t \notin Q(D)$ *to determine if the view output changes or not, which we can obtain by issuing an ad hoc query.*

We first consider the class of *SPJ* queries (queries with Selection, Projection, Join) without self-joins under bag (SQL) semantics. We then show how to extend our algorithm for aggregations.

4.1 Basic Algorithm

Recall that the database has schema $\mathbf{R} = (R_1, \ldots, R_k)$. Let P_i denote the *primary key* of R_i. We can express an SPJ query Q in the following form in Relational Algebra:

$$Q = \pi_{\mathbf{A}}(\sigma_C(R_{i_1} \times \cdots \times R_{i_\ell})) \tag{5}$$

Here, $\mathbf{A}$ denotes the projected attributes in Q, and C is an arbitrary boolean expression over the attributes (which includes both selection and join conditions). Given a tuple t in R, we denote by $C[t]$ the new boolean expression that results when we replace any attribute A of R in C with the value $t.A$. We also need to transform Q to an *augmented query* $\hat{Q}$ by including all primary keys P_i in the projection of the output for all relations R_i that participate in Q. Specifically,

$$\hat{Q} = \pi_{P_{i_1}, \ldots, P_{i_\ell}, \mathbf{A}}(\sigma_C(R_{i_1} \times \cdots \times R_{i_\ell})).$$

Our algorithm for checking disagreements for row updates returns $True$ if we find a disagreement, otherwise $False$. This is a key difference between our setting and standard view maintenance, where the actual change on the output result needs to be computed. It will be convenient to describe a row update $\mathsf{up}^\uparrow$ as a pair of tuples (u^-, u^+) from the same relation R: we remove u^- and subsequently add u^+. By construction of the row updates, we will have that $u^-.P = u^+.P$, where P is the primary key of R.

Algorithm 4: ROWDISAGREE($Q, D, \mathsf{up}^\uparrow$)

```
1  up↑ = (u⁻, u⁺)
2  R ← updated relation
3  P ← primary key of R
4  B ← updated attributes of R
5  if R ∉ {R_{i_1}, ..., R_{i_ℓ}} then
6      return False
7  if u⁻.P ∈ π_P(Q̂(D)) then
8      if B ∩ A ≠ ∅ or C[u⁺] is unsatisfiable then
9          return True
10     else
11         if Q((D \ R) ∪ {u⁻}) ≠ Q((D \ R) ∪ {u⁺}) then
12             return True
13 else
14     if Q((D \ R) ∪ {u⁺}) ≠ ∅ then
15         return True
16 return False
```

Algorithm 4 first checks (line 5) whether the relation that is modified by the update is involved in the query; if not, then we can safely say that the query result remains unmodified. Next, it checks (line 7) whether the tuple u^- has

contributed to the output. If so, we check whether any of the updated attributes B are in the projected attributes $\mathbf{A}$, or whether the new tuple u^+ cannot participate in an answer because it makes C unsatisfiable. We know that in both cases the output will certainly change. We should note here that the unsatisfiability check for $C[u^+]$ in our implementation is conservative for efficiency reasons (specifically, we check if the conditions that use only attributes of R become false). Also, observe that so far we perform the checks statically given the output of $\hat{Q}$ (which we compute once), without running Q on every updated database.

If such a static check is not possible, we need to request additional information from the database (lines 11 and 14). In this case, we execute the query Q, but instead of running the query using the full relation R, we replace it with the singleton $\{u^+\}$. In practice, we implement this as a left outer join over the updated and original databases. The proof for the correctness of Algorithm 4 can be found in Appendix A:

THEOREM 4.1. *Given a row update $\mathsf{up}^\uparrow = (u^-, u^+)$ and an SPJ query Q, Algorithm 4 outputs False if $Q(D) = Q(\mathsf{up}^\uparrow(D))$, otherwise it returns True.*

The case for swap updates is essentially the same,but with some additional optimizations. The pseudocode for the algorithm SWAPDISAGREE is presented in detail in Appendix A.

4.2 Batching Updates

We describe here a further optimization on the basic algorithm presented in the previous section. ROWDISAGREE speeds up the computation in two ways. First, it can filter out many possible updates from consideration without asking a query on the updated database. In practice, we noticed that a large fraction of updates was evaluated this way. Second, even if this is not the case, we can speed up the computation by running the query on the reduced databases $Q((D \setminus R) \cup \{u^-\})$ and $Q((D \setminus R) \cup \{u^+\})$, for the update $\mathsf{up}^\uparrow = (u^-, u^+)$. However, we still need to run these queries for as many updates as necessary; even though computation will be faster than $Q(D)$, if we start with a large support set, pricing may have a large overhead.

To overcome this problem, we *batch* many such query computations together in a single query. Suppose that we have n row updates $\mathsf{up}_1^\uparrow, \ldots, \mathsf{up}_n^\uparrow$ over the same relation R, and the algorithm needs to check whether $Q((D \setminus R) \cup \{u_i^+\}) = \emptyset$, for $i = 1, \ldots, n$. Let upid be an additional attribute that we use to keep the unique id of an update (which corresponds to an element of the support set). Let i be the upid of $\mathsf{up}_i^\uparrow$ for our case. We now create an instance $R^+ = \{(1, u_1^+), \ldots, (n, u_n^+)\}$ (we abuse the notation (i, u_i^+) to mean that we append an additional attribute upid to the tuple u_i^+). R^+ essentially aggregates all the updates together in a single relation. Let D^+ be the database that replaces R in D with R^+, and $\hat{Q}$ the extended query that runs as Q, but projects the attribute upid as well. We next compute $\hat{Q}(D^+)$ and observe the following: for any $i = 1, \ldots, n$, $Q((D \setminus R) \cup \{u_i^+\}) \neq \emptyset$ if and only if the output contains the upid with value i.

In general, for each relation R in the database we create three batch queries. The one we just described corresponds to the check we do in line 14 of Algorithm 4. We need two more batch queries for the check in line 11. The benefit of batching is that we only ask a constant number of queries on the database, independent of the size of the support set. Further, these queries will be much faster than $Q(D)$, since the instances we run them on will be of smaller size.

4.3 Aggregation

We present here an extension of our disagreement algorithm for *SPJ* queries that supports SQL queries with aggregation and grouping. We consider queries of the form

$$Q_\gamma = \gamma_{\mathbf{G}, \mathsf{agg}_1(A_1), \ldots, \mathsf{agg}_k(A_k)}(Q)$$

where Q is an SPJ query, agg_i are aggregation functions (SUM, COUNT, MIN, MAX, AVG), and $\mathbf{G}$ are the grouping attributes. Given a (row or swap) update $\mathsf{up}^\uparrow$, recall that our task is to check whether $Q_\gamma(D) = Q_\gamma(\mathsf{up}^\uparrow(D))$.

We first describe our algorithm for the case of row updates and aggregation that involves only COUNT; we will discuss how our technique can be generalized in the end of this section. For now, consider an aggregation query of the form $Q_\gamma = \gamma_{\mathbf{G}, \mathsf{COUNT}(*)}(Q)$. Let Q_γ° be the *unrolled* version of the aggregation query, which removes the aggregation and projects the relevant attributes: $Q_\gamma^\circ = \pi_{\mathbf{G}}(Q)$. For example, the query $\gamma_{age, \mathsf{COUNT}(*)}(User)$ on our running example is *unrolled* into the query $\pi_{age}(User)$.

The detailed approach is described in Algorithm 5. Notice that to check whether the tuple u^- of the update has contributed to the output, we need to compute the augmented unrolled query $\hat{Q}_\gamma^\circ(D)$ (line 9). Now if the update modifies any of the grouping attributes $\mathbf{G}$, we know for sure that the output of the aggregate query will change, since the groups will change. Similarly, if the new tuple u^+ makes the selection or join conditions unsatisfiable, we know that the groups where u^- contributed will have their size reduced, and thus the output will again change. If none of these happens, we have to go to the database and actually check if Q_γ changes by running the query on the update database (line 13). Finally, if u^- has not contributed to the output, we need to check whether u^+ will do so (line 16). If $Q_\gamma^\circ((D \setminus R) \cup \{u^+\}) \neq \emptyset$, then we know that the update will either add a new group, or increase the count of existing group. Observe that for algorithm ROWDISAGREEAGG it is not possible to batch the check of line 13 as with SPJ queries, but we can still batch the execution of line 16.

Algorithm 5: ROWDISAGREEAGG$(Q_\gamma, D, \mathsf{up}^\uparrow)$

```
1  Q_γ = γ_{G,COUNT(*)}(Q)
2  Q°_γ = π_G(Q)
3  up↑ = (u⁻, u⁺)
4  R ← updated relation
5  P ← primary key of R
6  B ← updated attributes of R
7  if R ∉ {R_{i_1}, ..., R_{i_ℓ}} then
8  |  return False
9  if u⁻.P ∈ π_P(Q̂°_γ(D)) then
10 |  if B ∩ G ≠ ∅ or C[u⁺] is unsatisfiable then
11 |  |  return True
12 |  else
13 |  |  if Q_γ(D) ≠ Q_γ((D \ {u⁻}) ∪ {u⁺}) then
14 |  |  |  return True
15 else
16 |  if Q°_γ((D \ R) ∪ {u⁺}) ≠ ∅ then
17 |  |  return True
18 return False
```

THEOREM 4.2. *Given a row update $\mathsf{up}^\uparrow = (u^-, u^+)$ and an aggregate query $Q_\gamma = \gamma_{\mathbf{G}, \mathsf{COUNT}(*)}(Q)$, where Q is an SPJ*

query, Algorithm 5 outputs False if $Q_\gamma(D) = Q_\gamma(up^\uparrow(D))$*, otherwise it returns True.*

The above algorithm can be slightly modified to also handle swap updates, in similar fashion to what we did with swap updates for SPJ queries. For aggregate queries involving other types of aggregation functions, disagreements are computed in the same way as COUNT queries, except that we also need to keep track of the aggregate values of each group in the output $Q_\gamma(D)$. For example, consider the case of a query that has a MAX aggregate: $\gamma_{\mathbf{G},\text{MAX}(A)}(Q)$, where the unrolled query is $\pi_{\mathbf{G},A}(Q)$. Let $up^\uparrow$ be a row update for which, according to Algorithm 5, we need to check the condition in line 16. If $Q^\circ_\gamma((D \setminus R) \cup \{u^+\}) \neq \emptyset$, we know that the update has contributed some new tuples in the unrolled query Q°_γ. However, we can't be sure that these new tuples will change the output, since it could be that they belong in a group that has a larger maximum value. To overcome this issue, we need to remember the maximum values from $Q_\gamma(D)$ and check the newly added tuples against the group maxima. The SUM aggregation works the same way. However, if we know that all values in the aggregate column are strictly positive, we can apply exactly Algorithm 5, without the need to do any additional bookkeeping (since we know that the addition or removal of tuples will certainly change the sum aggregate).

5. EXPERIMENTAL RESULTS

We have implemented our pricing framework in a tool called QIRANA in Python. We used MySQL (version 5.6.26) as the underlying DBMS that stores the data, but our approach can work for any DBMS that supports SQL. We evaluate QIRANA on two aspects: (a) the behavior of the prices assigned to queries, and (b) the runtime performance of the system for pricing queries on large datasets.

Benchmark Datasets. We perform our experimental evaluation running queries over 5 real-world datasets in various domains (see Table 2 for a summary of their characteristics):

1. `world`: a popular database provided for developers. It has 3 relations: `Country`, `City` and `CountryLanguage`.
2. `US Car crash 2011` [31]: A dataset about people involved in car accidents with fatalities available on Microsoft Azure DataMarket.
3. `DBLP` [12]: describes a co-authorship network: 2 authors are connected if they have a publication together.
4. `TPC-H` [28]: a benchmark for performance metrics for systems operating at a scale.
5. `SSB` [26]: a benchmark designed for measuring performance of classical data warehouse style workloads.

We use the `world` dataset to evaluate how prices are affected by support set size, as well as the fraction of row to swap updates. The `DBLP` and `US Car Crash` datasets are used to show the prices of a range of queries in a real-world setting. The `TPC-H` and `SSB` datasets are used to evaluate the scalability of our framework.

Experimental Setup. For all 5 datasets, we construct the support set assuming that the buyer knows the active domain and cardinality of the relations. In order to assign the weights, we assume that the seller has provided no other price points apart from the total price of the dataset.

QIRANA implements all four of the pricing functions in Section 2.3 with support set as random neighborhood around $\mathscr{D}$. In the following experiments we focus on the weighted coverage function p^{wc} unless explicitly stated. We made this choice because the weighted coverage function is the only pricing function that satisfies all three of the following properties: (a) it behaves well according to the benchmarking experiments in Section 2.4, (b) it is amenable to customization by using price points, and (c) it can be optimized for performance using the techniques of Section 4. We further fix a 1 : 1 ratio of row to swap updates for all our experiments, unless explicitly mentioned. We run all our experiments on a single machine running OS X 10.10.5, equipped with 2.2GHz processor and 16GB RAM. For all performance related experiments, we report the time averaged over 3 runs.

dataset	# relations	# tuples	# attributes
`world`	3	5,302	21
`US car crash`	1	71,115	14
`DBLP`	1	1,049,866	2
`TPC-H`	8	SF=1	61
`SSB`	8	SF=1	56

Table 2: Dataset characteristics.

5.1 Effect of Framework Parameters

We first evaluate the behavior of prices according to different choices of parameters. In particular, we study the effect of the support set size and the ratio of swap-row updates.

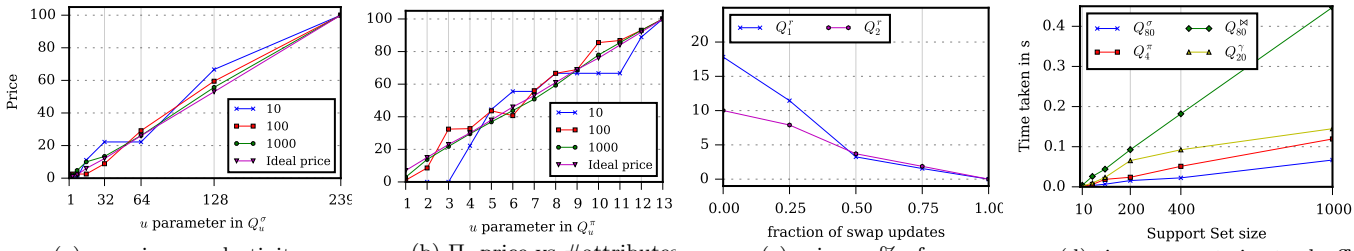
Varying Support Set Size. Figures 4a and 4b show the result of executing the benchmark queries Q^σ_u and Q^π_k (as listed in Section 2.4) for varying support set size: 10, 100 and 1000. The ideal price line plotted in both figures depicts the price in the case where the support set is equal to the whole set of neighboring databases. We can observe that for small values the resulting prices show high variability, since the support set does not cover the whole dataset well enough. Indeed, in the extreme where there exists one database in the support set, every query will have price either 0 or the price of the full database. As the support set size increases, the prices for both selection and projection gradually converge to the ideal price line. On the other hand, Figure 4d shows that as the support set size increases, the computation required to compute the price increases almost linearly. This experiment shows the seller has a tradeoff space between performance and how fine-grained the prices are, which she can tune to her desires.

Varying Row to Swap Update Ratio. We next study the effect of varying the ratio of row to swap updates. Consider the following two queries:

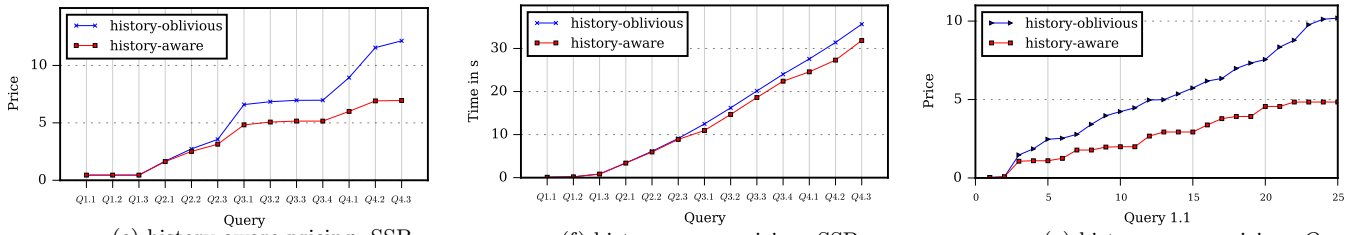
```
Q1r: SELECT AVG(Population) FROM Country ;
Q2r: SELECT Name FROM Country WHERE
     Population > 2,000,000,000 ;
```

For this experiment, we will assume that the buyer does not know the domain for `Population`, in which case a row update can introduce a new value. In Figure 4c, we plot the price of both queries when the fraction of swap updates (over the size of the support set) ranges from 0 to 1.

We can observe in Figure 4c that the price of both Q^r_1 and Q^r_2 is 0 when the fraction of swap updates is 1. This happens because a swap update will never lead to any disagreement for any of the two queries. This is because every swap will only exchange two values and the maximum value for `Population` in `Country` is 2,000,000,000. This is not de-

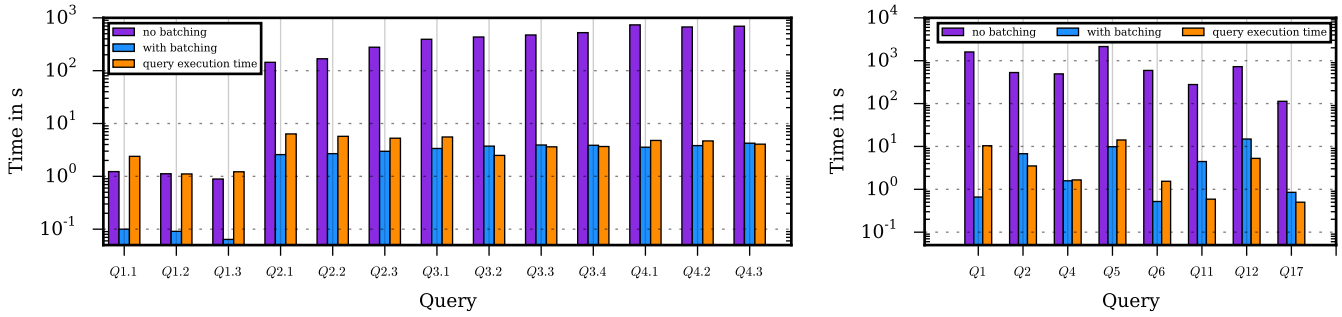


(a) $\sigma-$price vs selectivity (b) Π -price vs #attributes (c) price vs % of swaps (d) time-support size tradeoff



(e) history aware pricing: SSB (f) history aware pricing: SSB (g) history aware pricing: $Q_{1.1}$

Figure 4: Experiments on the `world` and `SSB` datasets. The support set size in figures 4e, 4f, 4g is fixed to 100000, and in figure 4c to 1000.



(a) SSB scalability (b) TPC-H scalability

Figure 5: Time in seconds to price SSB and TPCH queries on a support set of size 100000 without/with batching.

sirable, since when the user learns that the output of $Q^{\tau}_{2}(\mathscr{D})$ is empty, she has learned some information about the maximum possible value of `Population`. Similarly, the output of Q^{τ}_{1} discloses the average value.

As the fraction of swap updates decreases, the prices of both queries gradually increase. In the extreme that the support set contains only row updates, all updates to `Population` will lead to a disagreement. This is also not desirable, because the price of learning `AVG(Population)`($17 in our experiment) is close to price of learning the projecting of the entire column (in expectation, this would be $\$\frac{1}{13}100$ if all columns are uniformly valued). In our experience, an equal number of row and swap updates captures prices for aggregate queries best.

5.2 Scalability

In order to check how our algorithm scales, we evaluate our system on the `SSB` and `TPC-H` datasets using a scale factor of 1. As we mentioned in the beginning of the section, we price using only the weighted coverage function. The support sets size for both datasets is set to $S = 100,000$.

Our experimental results can be viewed in Figure 5a and 5b for the `SSB` and `TPC-H` benchmarks respectively. We evaluate the runtime performance for two different algorithms: the basic algorithm in Section 4.1, and the algorithm optimized to perform batch updates in Section 4.2. For reference, we also report the time to execute the query Q on the database $\mathscr{D}$. We should note here that the time to compute the price excludes the time to compute the query. Hence, we depict only the *overhead* of query pricing relative to the the query computation time.

Our experiments show that with all optimizations activated, we can price a query as fast (and sometimes faster) as the database can compute the query. Since the price computation involves asking queries over the database, a faster DBMS would result in more efficient pricing. We can also observe that the update batching optimization is critical to the efficiency of our pricing algorithm, since the basic algorithm involves executing a query multiple times over the data. Using batching makes pricing one to two orders of magnitude faster, since it avoids going to the database very frequently. For queries $Q1.1$-$Q1.3$ the price computation is even faster, since the pricing algorithm does not need to run any queries on the database to check for disagreements.

5.3 History-Aware pricing

For queries in the `SSB` benchmark, we also evaluate history-aware pricing to (a) observe the resulting prices assigned and (b) report the runtime when history-aware pricing is activated. Figure 4f depicts the runtime for history-aware pricing when we run all 13 `SSB` queries in a sequence. We can observe that history-aware pricing is more efficient compared to the history-oblivious version. This is because of the decrease in the number of support set elements that we need to consider at a later time. Figure 4e shows the pric-

	Q_1^d	Q_2^d	Q_3^d	Q_4^d	Q_5^d	Q_6^d
p^{wc}+nbrs	2.07	0	4.29	0.29	0.045	58.82
p^{H}+nbrs	3.05	0	6.91	0.29	0.048	62.29
	Q_7^d	Q_1^c	Q_2^c	Q_3^c	Q_4^c	
p^{wc}+nbrs	0.035	8.00	0.60	0.70	0	
p^{H}+nbrs	0.038	9.03	0.58	0.76	0	

Table 3: Prices for `DBLP` (Q_i^d) and `US Car crash` (Q_i^c)

ing savings for the same workload. Without history-aware pricing, a buyer would need to pay $12.14 instead of $6.94.

We observe similar behavior with parametrized queries. In the following, we generated 25 instances of query $Q1.1$ from `SSB` with varying parameter values for `dwdate`, `lo_discount` and `lo_quantity` sampled uniformly from their domain. Figure 4g shows the history-aware pricing for these queries. The history-oblivious version forces the buyer to pay more than 2× as compared to history-aware pricing.

5.4 Prices for real-world datasets

Our final experiment computes the prices for queries over two real world datasets: the `US Car crash` dataset and the `DBLP` dataset. The `DBLP` dataset describes a graph with 317,080 nodes and 1,049,866 edges (number of tuples). This information is public to all users who use this dataset. Figure 3 shows the history-oblivious query prices for both the datasets (the full list of the queries in Appendix B).

We next explain in more detail the prices of some of the queries. Consider the query Q_2^d whose price is 0: this is expected because Q_2^d asks for the average degree of an undirected graph, and both the number of nodes and the number of edges is public knowledge and hence known to the user. Q_6^d asks for all authors who have exactly one collaborator. Since the majority of the nodes in the `DBLP` graph have only one adjacent edge, a price of $58.82 is reasonable for this particular query. Q_4^c is assigned a price 0 even though the query output reveals some information about $\mathscr{D}$: this occurs because the query is very selective and no neighboring database in the support set caused a disagreement. In this case, choosing a bigger support set would result in a small non-zero price.

6. RELATED WORK

Data Pricing. There exist various simple mechanisms for data pricing (see [22] for a survey), including a flat fee tariff, usage based and output based. These pricing schemes do not provide any guarantees against arbitrage. The vision for arbitrage free query-based pricing was first introduced by Balazinska et al. [3], and was further developed in a series of papers [16, 18, 17]. The proposed framework requires that the seller sets fine-grained price points, which are prices assigned to selection queries with equality predicates over all possible values; these price points are used as a guide to price the incoming queries. Even though the pricing problem in this setting is in general NP-hard for SPJ queries, the QueryMarket prototype [18] showed that it is feasible to compute the prices for small datasets, albeit not in real-time. Further, it does not support aggregation queries, which are the most ubiquitous type of queries in a data analytics setting.

In QIRANA, price points are not a first-class citizen; we can compute fine-grained query prices even when the seller has specified a single price for the whole dataset, by assigning uniform value to every piece of the data. The seller provides price points only when necessary. In contrast, QueryMarket requires the seller to specify very fine-grained price points; even if the seller decides to assign the same equal price to every available price point, this can lead to undesired effects where a particular attribute value that touches most of the data is valued the same as one that touches a small part of the data. Another drawback of QueryMarket w.r.t. the price points is that it is not possible to control the price on an attribute-level, something that QIRANA supports.

Further work on data pricing has proposed new criteria for interactive pricing in data markets [20], and has described new necessary arbitrage conditions along with several negative results related to the trade-off between flexible pricing and arbitrage avoidance [21]. Upadhyaya et al. [30] investigates history-aware pricing using refunds. The pricing framework of QIRANA is based on recent work on how to characterize the possible space of pricing functions with respect to different arbitrage conditions [13].

View Maintenance. Our proposed techniques in Section 4 fall under the broad umbrella of incremental view maintenance (IVM). IVM has been studied heavily under set semantics [8, 6] as well as bag semantics [9]. The most related technique to ours is by Blakeley et al. [6], which describes a set of necessary and sufficient conditions to detect an *irrelevant* update without access to the database. We use such a condition as a part of our algorithm, but for our setting it is also necessary to check whether the output has changed for relevant updates as well. Several other approaches have been proposed for IVM: [15, 27, 34] develops techniques for queries with aggregation, negation and recursion while [25, 2] explore the idea of maintaining additional views to speed up the computation of the updated views.

One major difference between the above IVM settings and ours is that it is sufficient to check if the output changes rather than actually computing the change. In other words, the updates we consider are not materialized in any view and they do not change the underlying database. This allows for further optimizations: for instance, if we detect that an output tuple is not generated anymore after the update, we can immediately stop and not consider the other tuples. A second difference is that our pricing algorithms use only specific types of updates that touch only one or two rows in a table: this also allows for optimizations that may not be applicable in a general setting.

7. CONCLUSION

In this work, we present and evaluate QIRANA, a scalable framework for pricing relational queries with formal guarantees. We show that our pricing system supports a variety of pricing functions that guarantee arbitrage-free pricing, can compute the prices for a variety of SQL queries, and further scales effectively to large datasets. As a byproduct of our system design, we also propose and evaluate an algorithm that efficiently performs a form of view maintenance, which we believe could be of independent interest.

8. ACKNOWLEDGMENTS

Support for this research was provided by the University of Wisconsin-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation.

9. REFERENCES

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

[2] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, 5(10):968–979, 2012.

[3] M. Balazinska, B. Howe, and D. Suciu. Data markets in the cloud: An opportunity for the database community. *PVLDB*, 4(12), 2011.

[4] Banjo. ban.jo.

[5] Big Data Exchange. www.bigdataexchange.com.

[6] J. A. Blakeley, N. Coburn, P. Larson, et al. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems (TODS)*, 14(3):369–400, 1989.

[7] Bloomberg Market Data. www.bloomberg.com/enterprise/content-data/market-data.

[8] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *ACM Transactions on Database Systems (TODS)*, 4(3):368–382, 1979.

[9] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 190–200. IEEE, 1995.

[10] N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

[11] DataFinder. datafinder.com.

[12] DBLP dataset. https://snap.stanford.edu/data/com-DBLP.html.

[13] S. Deep and P. Koutris. The design of arbitrage-free data pricing schemes. *arXiv preprint arXiv:1606.09376*, 2016.

[14] S. Diamond and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993.

[16] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-based data pricing. In M. Benedikt, M. Krötzsch, and M. Lenzerini, editors, *PODS*, pages 167–178. ACM, 2012.

[17] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Querymarket demonstration: Pricing for online data markets. *PVLDB*, 5(12):1962–1965, 2012.

[18] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Toward practical query pricing with querymarket. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *ACMSIGMOD 2013*, pages 613–624. ACM, 2013.

[19] Lattice Data Inc. lattice.io.

[20] C. Li and G. Miklau. Pricing aggregate queries in a data marketplace. In *WebDB*, 2012.

[21] B. Lin and D. Kifer. On arbitrage-free pricing for general data queries. *PVLDB*, 7(9):757–768, 2014.

[22] A. Muschalle, F. Stahl, A. Löser, and G. Vossen. Pricing approaches for data markets. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, pages 129–144. Springer, 2012.

[23] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. SCS: Splitting conic solver, version 1.2.6. https://github.com/cvxgrp/scs, Apr. 2016.

[24] QLik Data Market. www.qlik.com/us/products/qlik-data-market.

[25] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *ACM SIGMOD Record*, volume 25, pages 447–458. ACM, 1996.

[26] SSB Benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[27] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *VLDB*, volume 96, pages 3–6, 1996.

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

[29] Twitter GNIP Audience API. gnip.com/insights/audience.

[30] P. Upadhyaya, M. Balazinska, and D. Suciu. Price-optimal querying with data apis. In *PVLDB*, 2016.

[31] USA Car crash 2011 dataset. https://datamarket.azure.com/dataset/bigml/carcrashusa2011.

[32] Windows Azure Marketplace. www.datamarket.azure.com.

[33] world dataset. https://dev.mysql.com/doc/world-setup/en/.

[34] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. *ACM SIGMOD Record*, 24(2):316–327, 1995.

APPENDIX

A. DETAILS FOR SECTION 4

Correctness of Algorithm 4. We first need a basic result over bag semantics.

LEMMA A.1. *Let E, F, G be bags. Then, $E \cup F = E \cup G$ if and only if $F = G$.*

PROOF. For a bag S, let f_S be a function that given a value t returns its multiplicity in S. Since $E \cup F = E \cup G$, we can write that for every t, $f_E(t) + f_F(t) = f_E(t) + f_G(t)$, which implies that $f_F(t) = f_G(t)$. Hence, $F = G$. □

LEMMA A.2. *Consider a database D, and a row update $up^{\uparrow} = (u^-, u^+)$, where $u^- \in R$. Then, $Q(D) \neq Q(up^{\uparrow}(D))$ if and only if $Q((D \setminus R) \cup \{u^-\}) \neq Q((D \setminus R) \cup \{u^+\})$.*

PROOF. Let us denote $D'' = D \setminus \{u^-\}$. We can now write:

$$Q(D') = Q((D \setminus \{u^-\}) \cup \{u^+\}) = Q(D'') \cup Q((D \setminus R) \cup \{u^+\})$$

$$Q(D) = Q((D \setminus \{u^-\}) \cup \{u^-\}) = Q(D'') \cup Q((D \setminus R) \cup \{u^-\})$$

where the last equality holds because union distributes over cartesian product, selection and projection. Notice that if the query includes only selection and projection over the relation R, then we simply have $(D \setminus R) \cup \{u^+\} = \{u^+\}$. We can apply now Lemma A.1 to obtain the desired result. □

PROOF OF THEOREM 4.1. Let P be the primary key for R. We distinguish the analysis to the following two cases:

1. $u^-.P \notin \pi_P(\hat{Q}(D))$: in this case, we know that u^- did not contribute to the output $Q(D)$, and thus it must be that $Q((D \setminus R) \cup \{u^-\}) = \emptyset$. By Lemma A.2, it now suffices to check whether $Q((D \setminus R) \cup \{u^+\}) = \emptyset$. Intuitively, if u^+ now contributes to some output tuple, we are certain that the output will change.
2. $u^-.P \in \pi_P(\hat{Q}(D))$: in this case, u^- contributes to at least one output tuple. If the new tuple u^+ makes C unsatisfiable, or any of the attributes modified are projected, we are certain that the output will change. If not and the query has no joins, we do not need to do anything further, since we can claim that the output stays the same. In the case of joins, we need to fall back to running the query. By applying Lemma A.2, it suffices to run the query not on the whole database, but the one that replaces R with the singletons $\{u^-\}$ and $\{u^+\}$ respectively.

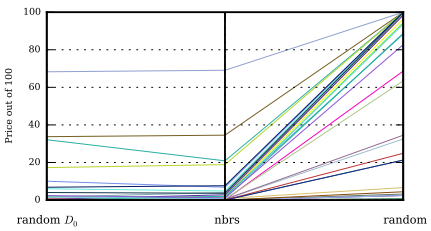
This concludes our correctness proof. □

Computing Disagreements for Swap Updates. We show in detail the algorithm that checks for disagreements in the case of a swap update (Algorithm 6). We denote a swap update as $\mathsf{up}^\uparrow = (u_1^-, u_1^+, u_2^+, u_2^+)$, where all tuples come from the same relation R. This update sequence removes u_1^-, u_2^- and adds u_1^+ and u_2^+.

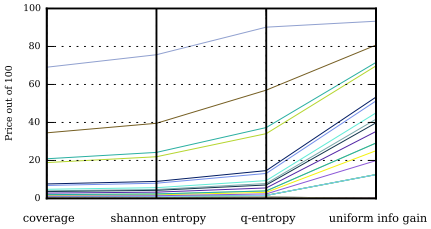
Algorithm 6: SWAPDISAGREE$(Q, D, \mathsf{up}^\uparrow)$

```
up↑ = (u1⁻, u1⁺, u2⁺, u2⁺)  R ← updated relation
P ← primary key of R
B ← updated attributes of R
if R ∉ {R_{i1}, ..., R_{iℓ}} then
    return False
if u1⁻.P and u2⁻.P ∉ π_P(Q̂(D)) then
    if Q((D \ R) ∪ {u1⁺, u2⁺}) ≠ ∅ then
        return True
else
    if B ∩ A ≠ ∅ then
        return True
    if C[u2⁺] and C[u1⁺] are unsatisfiable then
        return True
    else
        if
          Q((D \ R) ∪ {u1⁻, u2⁻}) ≠ Q((D \ R) ∪ {u1⁺, u2⁺})
          then
            return True
return False
```

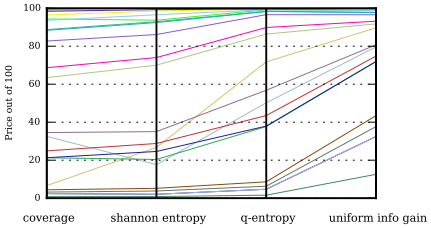
Instance Reduction. We present here another optimization for price computation. This optimization replaces the initial instance D with a smaller instance based on the choice of the support set $\mathcal{S}$. In particular, we replace the instance R_i with a new relation R_i^0 by choosing from R_i only the tuples that are updated in $\mathcal{S}$. Since a typical support size is smaller than the relation, R_i^0 will also be smaller in size. We denote the new database by D_i^0 (all other relations remain the same). We will next show that we can run the basic algorithm on any D_i^0 (instead of D), and obtain exactly the same disagreements.



(a) weighted coverage



(b) nbrs $\mathscr{D}$ support set



(c) uniform random support set

Figure 6: Additional benchmarking on `world` database

LEMMA A.3. *Let $\mathsf{up}^\uparrow$ be a row or swap update that modifies R_i. Then, $Q(D) = Q(\mathsf{up}^\uparrow(D))$ iff $Q(D_i^0) = Q(\mathsf{up}^\uparrow(D_i^0))$.*

PROOF. Let us first consider a row update $\mathsf{up}^\uparrow = (u^-, u^+)$ that updates R_i. From Lemma A.2, we know that $Q(D) = Q((D \setminus \{u^-\}) \cup \{u^+\})$ if and only if $Q((D \setminus R_i) \cup \{u^-\}) = Q((D \setminus R_i) \cup \{u^+\})$. But now for any tuple t, we have that $Q((D \setminus R_i) \cup \{t\}) = Q((D_i^0 \setminus R_i) \cup \{t\})$. Thus, if we apply again Lemma A.2, we will obtain that the desired result holds if and only if $Q(D_i^0) = Q(D_i^0 \setminus \{u^-\} \cup \{u^+\})$. The proof is similar for swap updates. □

B. BENCHMARK QUERIES

In this section, we list all the queries used for pricing for the real datasets. Queries for `world`, US car crash databset and DBLP are listed in Table 7, 9 and 8 respectively.

We also performed additional benchmarking experiment to test the behavior of the pricing functions on a variety of different queries on the `world` database (the full list of queries is in Appendix B). The results are plotted in Figures 6a, 6b, 6c. We can see here the same behavior observed in our initial benchmark.

	Query
Q_1^w	select count(Name) from Country where Continent = ‘Asia’
Q_2^w	select count(distinct Continent) from Country
Q_3^w	select avg(Population) from Country
Q_4^w	select max(Population) from Country
Q_5^w	select min(LifeExpectancy) from Country
Q_6^w	select count(Name) from Country where Name like ‘A
Q_7^w	select Region, max(SurfaceArea) from Country group by Region
Q_8^w	select Continent, max(Population) from Country group by Continent
Q_9^w	select Continent, count(Code) from Country group by Continent
Q_{10}^w	select * from Country
Q_{11}^w	select Name from Country where Name like ’A%’
Q_{12}^w	select * from Country where Continent=’Europe’ and Population > 5000000
Q_{13}^w	select * from Country where Region=’Caribbean’
Q_{14}^w	select Name from Country where Region=’Caribbean’
Q_{15}^w	select Name from Country where Population between 10000000 and 20000000
Q_{16}^w	select * from Country where Continent=’Europe’ limit 2
Q_{17}^w	select Population from Country where Code = USA’
Q_{18}^w	select GovernmentForm from Country
Q_{19}^w	select distinct GovernmentForm from Country
Q_{20}^w	select * from City where Population >= 1000000 and CountryCode = ’USA’
Q_{21}^w	select distinct Language from CountryLanguage where CountryCode=’USA’
Q_{22}^w	select * from CountryLanguage where IsOfficial = ’T’ ’
Q_{23}^w	select Language, count(CountryCode) from CountryLanguage group by Language
Q_{24}^w	select count(Language) from CountryLanguage where CountryCode = ’USA’
Q_{25}^w	select CountryCode, sum(Population) from City group by CountryCode
Q_{26}^w	select CountryCode, count(ID) from City group by CountryCode
Q_{27}^w	select * from City where CountryCode = ’GRC’
Q_{28}^w	select distinct 1 from City where CountryCode = ’USA’ and Population > 10000000
Q_{29}^w	select Name from Country , CountryLanguage where Code = CountryCode and Language = ’Greek’
Q_{30}^w	select C.Name from Country C, CountryLanguage L where C.Code = L.CountryCode and L.Language = ’English’ and L.Percentage >= 50
Q_{31}^w	select T.district from Country C, City T where C.code = ’USA’ and C.capital = T.id
Q_{32}^w	select * from Country C, CountryLanguage L where C.Code = L.CountryCode and L.Language = ’Spanish’
Q_{33}^w	select Name, Language from Country , CountryLanguage where Code = CountryCode
Q_{34}^w	select * from Country , CountryLanguage where Code = CountryCode

Figure 7: Queries used for `world` database

	Query
Q_1^d	select FromNodeId, count(ToNodeId) from dblp group by FromNodeId having count(ToNodeId) > 100
Q_2^d	select avg(cnt) from (select FromNodeId, count(ToNodeId) as cnt from dblp group by FromNodeId) as rc
Q_3^d	select count(*) from dblp A where FromNodeId > 10000
Q_4^d	select FromNodeId, count(*) from dblp A where A.FromNodeId in (select FromNodeId from dblp B where B.ToNodeId = 38868) group by FromNodeId
Q_5^d	select ToNodeId from dblp where (FromNodeId = 148255 or FromNodeId = 45479)
Q_6^d	select FromNodeId, count(*) as collab from dblp group by ToNodeId having collab = 1
Q_7^d	select * from dblp A where A.FromNodeId = 38868 or A.ToNodeId = 38868

Figure 8: Queries used for DBLP database

	Query
Q_1^c	select state, count(*) from crash group by State
Q_2^c	select count(*) from crash where State = ’Texas’ and Gender = ’Male’ and Alcohol_Results > 0.0
Q_3^c	select sum(Fatalities_in_crash) from crash where State = ’California’ and Crash_Date >= date ’2011-01-01’ and Crash_Date < date ’2011-01-01’ + interval ’6’ month
Q_4^c	select count(Fatalities_in_crash) from crash where State = ’Wisconsin’ and Injury_Severity = ’Fatal Injury (K)’ and (Atmospheric_Condition = ’Snow’)

Figure 9: Queries used for US car crash dataset