QIRANA Demonstration: Real Time 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

Yash Bidasaria
University of Wisconsin-Madison
Madison, USA
bidasaria@wisc.edu

ABSTRACT

The last decade has seen a deluge in data collection and dissemination across a broad range of areas. This phenomena has led to creation of online data markets where entities engage in sale and purchase of data. In this scenario, the key challenge for the data market platform is to ensure that it allows real time, scalable, arbitrage-free pricing of user queries. At the same time, the platform needs to flexible enough for sellers in order to customize the setup of the data to be sold. In this paper, we describe the demonstration of QIRANA, a light weight framework that implements query-based pricing at scale. The framework acts as a layer between the end users (buyers and sellers) and the database. QIRANA's demonstration features that we highlight are: (i) allows sellers to choose from a variety of pricing functions based on their requirements and incorporates price points as a guide for query pricing; (ii) helps the seller set parameters by mocking workloads; (iii) buyers engage with the platform by directly asking queries and track their budget per dataset; .We demonstrate the tunable parameters of our framework over a real-world dataset, illustrating the promise of our approach.

1. INTRODUCTION

Businesses, entities (both private and public) and even individuals are increasingly becoming data-driven. Cloud based data democratization has allowed data, which was once siloed, to be accessible, tradable and actionable at the click of a button. This unprecedented demand for acquiring data for analysis has created a growing need for data brokers in the digital space. Several online data markets have emerged as a key platform to facilitate exchange of data - Microsoft Azure DataMarket [10], InfoChimps [2], Socrata [8] are primary examples. For the datasets to be vendible, data markets need to define pricing schemes for each dataset. However, most pricing schemes used in practice are not flexible; they typically allow users to either buy the entire dataset or in predefined large chunks. Such schemes are problematic for buyers, since they are interested in purchasing more fine-grained queries over multiple sources [7]. Pricing mechanisms that offer buyers full freedom to chose which query to purchase are called *query-based pricing*. A more serious limitation of query-based pricing schemes used in practice (such as usage based pricing or flat fee per query) is that they can lead to inconsistent pricing. A clever buyer could exploit such a mechanism to obtain some data at no cost. Previous work has studied this problem from both theoretical [3, 5, 6, 1] and practical [3, 4, 9] point of view. The key principle identified in these work is *arbitrage*; i.e it should not be possible for a buyer to acquire the desired query for a cheaper price. Several practical frameworks have been designed using this principle. The QueryMarket [3, 4] prototype used this idea to price relational queries by expressing them as an ILP program with constraints as price points set by the seller for certain queries. More recent work in [9], considers a simple provenance based pricing scheme of tuples that contribute to the output.

The solutions proposed above for pricing all have some limitations. QueryMarket does not allow grouping and aggregations queries. Moreover, even simple join queries do not scale well in the system: for instance, join query over relation with 1000 tuples takes about one minute. Provenance based pricing in [9] also does not allow boolean and aggregate queries. This scheme is also prone to arbitrage attacks. To the best of our knowledge, there is no existing query-based pricing framework that can price a wide spectrum of SQL queries in real time, while providing formal guarantees about arbitrage freeness.

In this demonstration, we highlight the features and capabilities of our pricing framework called QIRANA. QIRANA's design takes into account the following requirements: *(i)* it allows the data seller to choose from several provably arbitrage-free pricing functions, *(ii)* it is applicable to a large class of SQL queries (Selection-Projection-Join queries plus aggregations), and *(iii)* it ensures that buyers are not charged twice for the information they have already purchased, i.e the pricing is history aware. QIRANA offers further advantages: *(i)* it is efficient in using resources (small CPU and memory overhead), *(ii)* it is easy to deploy on top of any DBMS without modifying the database internals, *(iii)* it generates prices of queries in real time.

Contributions. The goal of the demonstration is to provide a comprehensive description of QIRANA for various usage scenarios over real-world datasets. In particular:

Proceedings of the VLDB Endowment, Vol. 10, No. 12

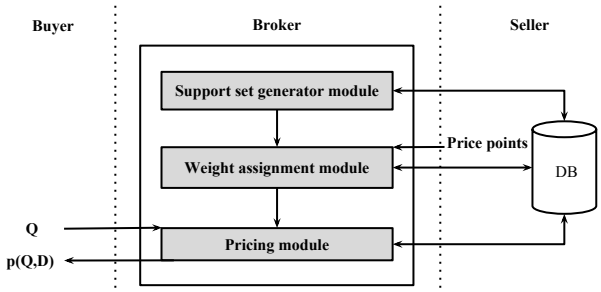


Figure 1: QIRANA architecture.

- We demonstrate how QIRANA helps sellers to set up the dataset they want to sell. In particular, sellers can specify schema information (such instance domain, cardinality constraints and other public information about the database) in order to capture the information content of a query. Additionally, the seller can specify price points for selection and projection queries to make the value of different parts of data non-uniform.
- We show how the seller can tune parameters over a supplied workload in order to improve prices assigned for ad-hoc queries.
- We develop an interface for buyers where they can issue queries over a dataset. Buyers can enter their budget per dataset and track their purchases over time.

Outline. Section 2 describes the system functionality and technical implementation details. We present the demonstration scenarios in Section 3.

2. SYSTEM OVERVIEW

In this section, we describe an overview of QIRANA's functionality and architecture. We consider a general setting where a *data seller* (Bob) wants to sell a database $\mathscr{D}$ which has a fixed schema $\mathbf{R} = (R_1, \ldots, R_k)$. A *data buyer* (Alice) can purchase information from the dataset by issuing SQL queries in the form of a *query bundle* $\mathbf{Q} = (Q_1, \ldots, Q_n)$, which is a vector of queries. We use $\mathcal{I}$ to denote the set of all *possible databases* that conform to public knowledge (such as domain constraints, relation cardinalities, schema, functional dependencies) known to Alice. $\mathscr{D}$ is hosted on a data platform owned by a *data broker* (Dave).

2.1 Pricing Guarantees

In this section, we describe QIRANA's key features with the help of a running example over the database in Figure 2.

Arbitrage Freeness. The main goal of any query-based pricing scheme is to ensure that there are no arbitrage opportunities based on the prices assigned to queries. Suppose Alice asks the following query: Q_1 = `SELECT count(*) FROM User WHERE gender = 'f'` and the price assigned to query is \$7. Next, Alice asks Q_2 = `SELECT gender, count(*) FROM User GROUP BY gender`. Now, if the system prices Q_2 as \$5, then there exists an arbitrage opportunity because Alice can get the information of Q_1 from Q_2 at a cheaper price. We call such an arbitrage opportunity *information arbitrage*. The data seller can avoid information arbitrage by ensuring that $p(Q_1) \leq p(Q_2)$.

The next arbitrage condition we want to avoid is known as *bundle arbitrage*. Consider Q_3 = `SELECT AVG(age) FROM User` and Q_4 = `SELECT SUM(age) FROM User` in our running example. Bundle arbitrage states that it should not be possible to acquire output of a query (Q_3) by combining output of other queries (Q_4 and Q_2). In other words, it should not be the case that combining information from multiple queries is cheaper than asking the query directly. QIRANA uses provably arbitrage-free pricing functions that ensure that there are no inconsistencies in prices assigned to queries.

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

Figure 2: User table for running example

History Aware. In the context of a data market, Alice may want to issue multiple queries $Q_1, \ldots, Q_k$ over time, some which may contain repeated information. A pricing scheme is called history-aware if the whole sequence of the queries will be priced as a bundle rather than individually. For example, consider the query Q_5 = `SELECT COUNT(*) FROM User WHERE gender = 'm'`. Note that this query overlaps with the information from query Q_2. Therefore, if Alice has already purchased Q_2, then system should take into account Alice's query purchase history. In such a history-aware scenario, Q_5 should be free.

2.2 System Architecture

QIRANA is implemented in Python. We implement our framework on top of MYSQL, but it can be deployed over any relational database that supports stored procedures (such as SQL SERVER or POSTGRES). The system architecture is depicted in Figure 1.

Support Set. 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. 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 pricing functions of that form that satisfy specific mathematical properties are arbitrage free [1].

Since $\mathcal{I}$ can be exponentially large (or possibly infinite), keeping track of the all possible instances is infeasible. Instead, QIRANA looks only at a carefully chosen small subset of $\mathcal{I}$ (which we call the *support set* $\mathcal{S}$) that approximates query prices well. The support set is generated by choosing databases in the neighborhood of $\mathscr{D}$: a database D is in the *neighborhood* of $\mathscr{D}$ if it differs in at most two attributes w.r.t $\mathscr{D}$. Our approach generates a support set by sampling uniformly at random from neighboring databases. We consider two ways to create neighboring databases: *(i)* changing one or two attributes of a tuple with a different value in the domain: these are called *row updates*, *(ii)* exchanging the attribute values of two different tuples: these are known as *swap updates*. Note that we do not change the cardinality of the relation as this may be publicly known.

EXAMPLE 2.1. *Consider the* User *relation in our running example. The following row update modifies the* gender *value of the tuple with key* 1 *to* f *to create a neighboring instance of*

Table 1: Pricing function in QIRANA with their properties. w_i is the weight assigned to each $D_i \in \mathcal{S}$ for coverage function. For shannon entropy and q-entropy, $\sum_i w_i = 1$ and $w_B = \sum_{i:D_i \in B} w_i$ for $B \in \mathcal{P}_{\mathbf{Q}}$ where $\mathcal{P}_{\mathbf{Q}}$ is partition induced by equivalence class $D \sim D' : \mathbf{Q}(D) = \mathbf{Q}(D')$

Pricing Function	Formula	Price Points
weighted coverage	$p^{wc}(\mathbf{Q}, \mathscr{D}) = \sum_{i:\mathbf{Q}(D_i) \neq \mathbf{Q}(\mathscr{D})} w_i$	✓
uniform entropy gain	$p^{ueg}(\mathbf{Q}, \mathscr{D}) = \frac{\log\lvert\{D \in \mathcal{S} \mid \mathbf{Q}(D) \neq \mathbf{Q}_1(\mathscr{D})\}\rvert}{\log\lvert\mathcal{S}\rvert}$	✗
shannon entropy	$p^{H}(\mathbf{Q}, \mathscr{D}) = -\sum_{B \in \mathcal{P}_{\mathbf{Q}}} w_B \log w_B$	✗
q-entropy	$p^{T}(\mathbf{Q}, \mathscr{D}) = \sum_{B \in \mathcal{P}_{\mathbf{Q}}} w_B \cdot (1 - w_B)$	✗

$\mathscr{D}$: `UPDATE User SET gender = 'f' WHERE uid =` 1. Note that the new database instance differs in only one attribute. Similarly, a query modifying `age` = 19 for the tuple with key 1 and `age` = 25 for the tuple with key 4 constitutes a swap update. In this case, the neighboring database differs in exactly two attributes as compared to $\mathscr{D}$.

In order to create neighboring database instances, we also need to know the domain of the attributes. QIRANA lets the seller optionally specify the domain for each attribute in the schema. If domain is not specified, we use the active domain. Observe that neighboring database also have the advantage 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 producing D.

Pricing Functions. Table 1 shows the pricing functions available in QIRANA. The right choice of pricing function depends on seller requirements. For example, if the seller wants to incorporate price points, then only the weighted coverage function supports that.

For customizability, weighted coverage pricing function allows seller to incorporate 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 Bob 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). We set weights for each database in $\mathcal{S}$ according to the following entropy maximization (EM) program. A valid solution to this problem guarantees that the price points are satisfied.

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

The first constraint encodes the fact that price of the whole dataset is P. The second constraint makes sure that the price points are satisfied. Intuitively, 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).

Price Computation. Given a pricing function and support set, we compute the price of a query $\mathbf{Q}$ as follows. For all instances D in $\mathcal{S}$, we check if $\mathbf{Q}(D) \neq \mathbf{Q}(\mathscr{D})$. Based on the pricing function, we use this information to calculate the weighted sum(for coverage function and entropy gain) or the entropy(Shannon entropy and q-entropy). Note that we have to run the query as many times as the size of the support set. The key observation is that to overcome this bottleneck, we only need to check whether the update has modified the query output, and not actually run the query. In QIRANA, we use a combination of static analysis and batching techniques to speed up the pricing process.

3. DEMONSTRATION SCENARIOS

The goal of the demonstration is to showcase the attendees QIRANA's ease-of-use and interaction between all parties involved. Attendees play the role of Bob and Alice. The demonstration consists of three parts. First, we present dataset seller's (Bob) setup: this includes the dataset loading process, specifying domain information and price points, and selecting the pricing function. Second, we show how Bob can tune the support size and specify domain information iteratively to capture the information content better. Bob can run mock query workloads during the setup phase to see if the prices generated based on which he can choose to change the setup to tune the query prices. Third, we show Alice's account setup: buyers subscribe to datasets they wish to explore and issue SQL queries. For every query Q, Alice can view the query output and gets charged the corresponding price $p(Q, D)$. Alice can also view her query purchase history visually and also see her savings with respect to a history-oblivious pricing scheme.

Setup Details. To demonstrate how QIRANA behaves on real-world datasets, we use the `US car crash` dataset from Microsoft Azure DataMarket [10]. This dataset contains information about people involved in car accidents in 2011. It has 71,115 tuples in a single relation with 14 attributes. This relation is loaded in MYSQL after some minimal cleaning (missing values for numeric attributes in were replaced with the smallest value found in the corresponding attributes). We consider the following queries that Alice might ask.

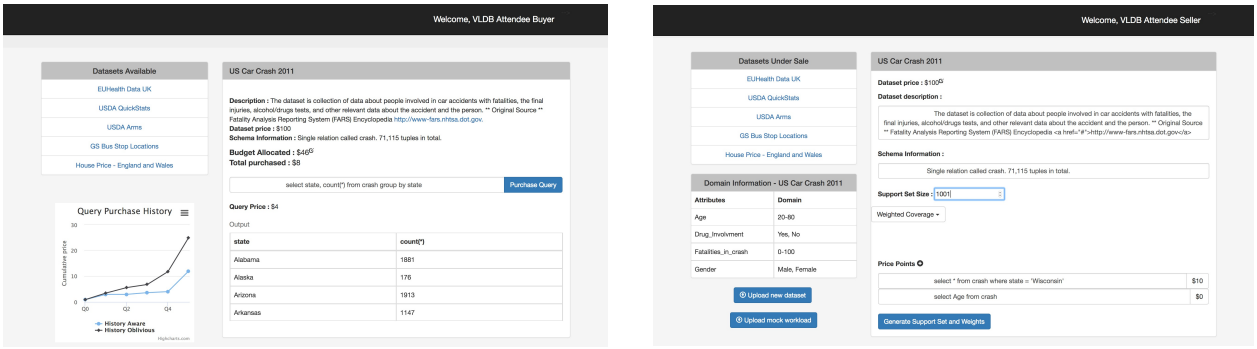
$Q_c^1(u)$: `SELECT state, COUNT(*) FROM crash WHERE Fatalities_in_crash >` u `GROUP BY State;`

$Q_c^2(\vec{u}, v)$: `SELECT` $\vec{u}$ `FROM crash WHERE Atmospheric_condition =` v`;`

$Q_c^3(u, v)$: `SELECT * FROM crash WHERE State =` u `AND Gender =` v`;`

Our demonstration captures 3 different scenarios:

1. Tuning parameters for dataset sale. In the first scenario, attendees will interact with QIRANA as Bob by selecting the corresponding option (see figure 3b). QIRANA loads the `crash` dataset as `.sql` file supplied by Bob. Once the



(a) Buyer view

(b) Seller view

Figure 3: QIRANA demonstration scenarios

relations are loaded in the database, attendees can optionally specify the following information: (i) For each attribute, specify the domain information and the type. For instance, `Age` can only take values between 20 and 80. If the domain is blank, the the active domain of the attribute is used. Next, attendees can pick one of the four pricing functions from Table 1. If the pricing function chosen is weighted coverage, then price points of the form (Q_j, p_j) can be optionally entered. In practice, we restrict Q_j to be selections or projections. For the demonstration, we set price of Q_c^1 as \$10 and Q_c^2 as \$0 following which, the framework automatically generates support set $\mathcal{S}$ and solves the EM program.

2. Fine tuning setup parameters. In order to help Bob choose support set size, QIRANA can run a mock workload supplied by Bob to display the prices. Based on the prices, Bob can adjust the support set size. For this part of the demonstration, attendees will be able to see how we specify a query workload to the system and change the support size. This also demonstrates a trade-off between the prices assigned and the support set size. As the support set size increases, the prices capture information content of a query better. Attendees will be able to change the size and iteratively observe better prices for the mock query workload. Based on this step, Bob may even choose to add more price points for queries that are not assigned prices to his liking. Note that setting price points and fine tuning support size is a one-time setup per dataset being sold. The framework parameters remain fixed once buyers issue queries over the dataset.

3. Issuing queries for purchasing data. Next, the attendees assume the role of Alice and get a chance to purchase data. QIRANA lets buyers track their budget per dataset by monitoring how much has been spent per query. This is especially important for low budget entities such as scientists and lay users who cannot afford to buy the entire dataset. For this usage scenario, we allow parameterized versions on Q_c^1, Q_c^2 and Q_c^3 to experiment with different variants of queries. We issue Q_c^1 with an appropriate argument to QIRANA which executes the query over the database and displays the query price along with the output. Next, we issue another query `SELECT COUNT(State) FROM crash`. Note that this query provides some information that is already known to buyer from Q_c^1. Therefore, full price is not charged for this query showing the history-aware pricing nature of our framework. Similarly, we can issue Q_c^2 and Q_c^3 with appropriately chosen input to see how the framework assigns prices. For this part of the demonstration, the attendees will be able to set different values for the parameterized queries to create a complex workload with different predicates and selectivities that can be priced interactively. By choosing the input arguments appropriately, attendees will also be able to observe history-aware pricing for ad-hoc queries. They will also be able to view the purchase history visually per query (graph in figure 3a). Additionally, QIRANA also shows the savings with respect to a history oblivious pricing scheme.

4. CONCLUSION

We demonstrate QIRANA, a light weight framework that implements arbitrage free query-based pricing. We show how buyers can engage with the framework and ask queries while tracking their budget. QIRANA allows sellers to choose from a list of pricing functions according to their requirements. In order to customize the prices, we show how seller can specify price points by valuing different parts of the database independently. We demonstrate the ease-of-use and effectiveness of our framework on use-cases presented on real-world datasets.

5. REFERENCES

[1] S. Deep and P. Koutris. The design of arbitrage-free data pricing schemes. *In Proceedings of ICDT*, 2017.

[2] Infochimps. infochimps.com.

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

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

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

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

[7] 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.

[8] Socrata. socrata.com.

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

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