Toward Practical Query Pricing with QueryMarket

Paraschos Koutris, Prasang Upadhyaya,
Magdalena Balazinska, Bill Howe, and Dan Suciu
University of Washington, Seattle, WA
{pkoutris,prasang,magda,billhowe,suciu}@cs.washington.edu

ABSTRACT

We develop a new pricing system, QueryMarket, for flexible query pricing in a data market based on an earlier theoretical framework (Koutris et al., PODS 2012). To build such a system, we show how to use an Integer Linear Programming formulation of the pricing problem for a large class of queries, even when pricing is computationally hard. Further, we leverage query history to avoid double charging when queries purchased over time have overlapping information, or when the database is updated. We then present a technique that fairly shares revenue when multiple sellers are involved. Finally, we implement our approach in a prototype and evaluate its performance on several query workloads.

Categories and Subject Descriptors

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

General Terms

Algorithms, Economics

Keywords

Data Pricing, Integer Linear Programming

1. INTRODUCTION

Data is increasingly sold and bought online. Apart from websites that sell data that they collect on their own [3, 5, 8], several web-based marketplace services have recently emerged: Azure Data Marketplace [1] and Infochimps [7] are primary examples. These marketplace services offer a public platform where data providers can upload and sell their data and data consumers can purchase it. However, the pricing schemes that are currently used to price data are either simplistic (a seller offers the whole dataset or parts of the dataset for fixed prices) or not flexible (only a limited class of queries is allowed). This limitation is problematic since recent work [17] demonstrated that buyers are interested in purchasing specific and complex information that requires combining subsets of data from multiple sources. In the absence of more flexible pricing schemes, buyers are forced to purchase supersets of the data they need.

In this work, we design and build a new system, called the *QueryMarket*, as a back-end engine to price and manage data in a marketplace service. QueryMarket enables practical and flexible pricing schemes: It enables sellers to upload their data and specify a set of price points in the form of *selection queries* over that data. Buyers can then purchase complex queries over all the data available for sale, thus having the flexibility to buy exactly the information needed. A buyer simply specifies her query and the system automatically computes the best set of selection queries to purchase in order to answer her query, given the current database instance and her past purchases. In prior work [13, 14] we studied the theory of flexible query pricing and showed that all but the simplest queries are hard to price. The main challenges for implementing a pricing system is thus to *(1) guarantee the efficient computation of prices for a large class of queries* and *(2) support the necessary functionality that a practical data marketplace requires*, which includes support for updates, queries over data from multiple vendors and avoiding double-charging for the same data. This paper takes an important first step toward addressing these two core challenges and outlines future work necessary to achieving a full-featured practical query pricing system.

In QueryMarket, the price the price depends only on the *information content* of the query, and not on the computational cost. Once the system computes a price, it fairly remunerates the various data providers involved in the transaction. Moreover, over time a data consumer is buying several queries, so the QueryMarket system must avoid double-charging the buyer for purchasing the same data. To achieve this, the system identifies redundant information between queries and adjusts the price accordingly. Hence, the price of a query also depends on the *query history* of a user. Since it is often the case that users ask closely related queries over a session, incorporating past purchases to avoid overcharging is a fundamental requirement. Finally, as users purchase data, sellers may add to it or may update it. QueryMarket supports various policies for selling updates, including providing updates for free and charging for updates in proportion to the amount of changed information.

The next example presents a practical case for pricing and discusses the challenges and requirements for QueryMarket.

EXAMPLE 1.1. *Our running example concerns selling data on word translation and word frequency. For any pair*

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

of languages, e.g. *English, German, there exists a table* $T_{en,de}(w_1, w_2)$*. An entry* $T_{en,de}$*('thanks','danke') means that 'thanks' in English translates to 'danke' in German. Such data is currently sold through the Azure Marketplace [2], where the price depends on characters per month purchased. For example, 4 million characters per month cost $40.*

*Moreover, the dataset contains tables with frequency data about unigrams and bigrams for various genres (*e.g. *fiction, newspaper, music):* $UF(w, genre, freq, rank)$ *and* $BF(w_1, w_2, genre, freq, rank)$*. Such data is currently sold in [9] in a slightly different format. Both datasets can be purchased for $250 each (or subsets for a lower price).*

In current systems, our data consumer, Alice, can access this dataset in limited ways. The tables UF, BF *can only be bought in large chunks of words, even if Alice is interested in asking the frequency of a single word. The translation dataset enables translations of single words or text: for example, Alice can ask for the German translation of the word 'thanks':* $Q_1(x) = T_{en,de}('thanks', x)$*. However, Alice may be interested in more complex queries that combine multiple tables. Using the concise Datalog notation, the queries are:*

$$Q_2(x, y) = T_{de,en}(x, y), T_{de,fr}(x, y)$$
$$Q_3() = UF(\text{`}rock\text{`}, \text{`}music\text{`}, f_1, y), UF(\text{`}pop\text{`}, \text{`}music\text{`}, f_2, z), f_1 > f_2$$
$$Q_4(x, y) = UF(x, \text{`}math\text{`}, z, r), T_{en,de}(x, y), r \leq 100$$
$$Q_5(x) = T_{en,de}(x, x) \vee T_{en,fr}(x, x) \vee T_{en,sp}(x, x)$$

Query Q_2 *asks for all German words with identical English and French translations: 'Polizei' is such an example, since its English/French translation is 'police'.* Q_3 *asks if the word 'rock' appears in the genre 'music' more frequently than 'pop'.* Q_4 *asks for the German translation of the top 100 English math words. Finally,* Q_5 *asks for any English words such that its Spanish, German or French translation coincides with it. Obviously, in each case, Alice would like to pay only for the information content of that particular query. This is not possible in today's data markets, forcing Alice to buy entire datasets so as to answer complex queries.*

Additionally, multiple vendors may be necessary to answer each one of Alice's queries. For example, one seller S_T *may offer table* T *while another seller* $S_{UF,BF}$ *may sell the* UF *and* BF *tables. Today's data market services force users to purchase data separately from each vendor involved.*

Alice will also issue multiple queries over time. These queries may have overlapping information contents, for example, Alice may later decide that she needs the top 250 math words in English and issue query

$$Q_6(x, y) = UF(x, \text{`}math\text{`}, z, r), T_{en,de}(x, y), r \leq 250$$

similar to Q_4*. More subtle, if Alice issues* Q_5 *after* Q_4*, some of the same* $T_{en,de}(x, y)$ *tuples may need to be purchased for both queries. Ideally, Alice should not pay for information that she previously purchased. Today's data markets, however, commonly ignore any purchase history.*

Finally, the data may change over time. For example, the ranks and frequencies of various words may be updated and new words may be added. Today's marketplace services do not provide support for selling data updates.

QueryMarket is the first pricing system that supports the pricing of the above queries and also handles the other key technical challenges. Our main contributions in this paper can be summarized as follows.

An Implementation of the Static Pricing Framework. (Section 4) With our pricing scheme the seller specifies base prices for selection queries and the system then automatically computes the price of any user query. As argued in prior work [13], the pricing function must be *arbitrage-free*, meaning that no query can be answered from a set of cheaper queries. However, computing an arbitrage-free pricing function is NP-hard for a large class of practical queries, like those shown in Example 1.1. Thus, to design a system that prices an arbitrary query in reasonable time, we propose to formulate the pricing problem as an Integer Linear Program (ILP) and then call a state-of-the-art solver to compute the price. We show how to construct the ILP for a large class of queries, that includes unions of conjunctive queries (the class of SELECT-PROJECT-JOIN queries with disjunctions in SQL), query bundles and interpreted constraints (which add unary predicates for selections, theta joins, and even user-defined functions). We further describe how to incorporate additional knowledge about the data (in the form of key constraints) to the price computation.

This approach gives us a powerful tool to price a large subclass of SQL queries (with the exception of negation and aggregation). We further discuss several optimizations that minimize the size of the ILP to gain performance.

Pricing in a Dynamic Setting. (Section 5) Second, we study the case where a data consumer issues a series of queries over the database. In this case, there may be overlapping information between queries, a fact that reduces the information gain of a buyer. For example, if Alice issues first Q_4 and then Q_5, there may be a word that needs to be translated in both queries: in this case, it is fair to ask Alice to pay once. We propose and evaluate several methods to incorporate the query history of a user when computing the price. In particular, we show that one of our techniques is not only efficient to implement, but also captures well the information gain of the data consumer.

We further use query history to cope with pricing for *dynamic* databases, *i.e.* databases where the data is updated (*e.g.* the word frequency set may count the frequency of word in Twitter during a 24 hour window).

Revenue sharing. (Section 6) Finally, we discuss an extension of the basic single-vendor scenario, and consider the case when the data is contributed by multiple vendors. The buyer issues a query that integrates data from multiple vendors, and the challenge is to allocate the revenue fairly between the sellers. We present a solution to this problem, which only adds a modest amount of complexity over the basic pricing scheme for a single vendor, while ensuring that the revenue is shared fairly between sellers.

Prototype Implementation and Evaluation. (Section 7) We implement the above techniques in a prototype pricing system called *QueryMarket* and evaluate their performance on real data, as described in Example 1.1.

Organization. In Section 2, we review the pricing framework from our prior work [13]. In Section 4, we show how to implement pricing as an ILP and to optimize its construction. Section 5 discusses and compares our solutions for incorporating query history in pricing. Finally, in Section 7 we present and discuss our experimental findings.

2. BACKGROUND: PRICING FRAMEWORK

In prior work [13], we introduced a flexible data pricing framework, which requires that data sellers specify a small

number of distinct price points for the data. It then uses these price points to automatically derive the price of any query submitted by buyers. The key objective is to limit the amount of work done by sellers, yet enable buyers to purchase and pay for personalized data products. A key concept of our framework and related frameworks [15] is that prices should be arbitrage-free. We adopt here the same principled approach to pricing; in this section we review briefly arbitrage-free pricing, following our prior work [13].

In the framework that we adopt, a data seller sells queries over a database D. Instead of specifying a price for every possible query, the seller instead specifies a set of explicit *price points* $\mathcal{S} = \{(V_1, p_1), \ldots, (V_k, p_k)\}$, where a pair (V_i, p_i) denotes that the seller wishes to sell the view V_i over the database D for a price p_i. For example, a seller may decide to sell the set of words with any given rank for \$1: then there is a parameterized view, $V_k(w, g, f, r) = UF(w, g, f, r), r = k$, for each rank k. Thus, $\mathcal{S}$ contains a pair $(V_k, \$1)$ for every rank k. Even better, the prices can depend on the rank, *e.g.* the seller charges \$1 when the rank k is lower than 10, charges \$0.50 when the rank is between 10 and 99, and charges only \$0.10 for ranks 100 and up.

When the system receives a request by a data consumer to answer a query Q, it automatically computes a price $p^{\mathcal{S}}_D(Q)$. In fact, the framework defines the pricing function more generally, for an entire *query bundle* $\mathbf{Q} = \{Q_1, \ldots, Q_k\}$, *i.e.* it allows the buyer to purchase simultaneously a set of queries, which will be priced together. This can be cheaper than pricing the queries separately, since it avoids double charging for overlapping information between queries. The pricing function must agree with the price points set by the seller, and, in addition, must satisfy the following properties:

Arbitrage-free: Whenever a query bundle $\mathbf{Q}$ can be answered entirely from the query bundles $\mathbf{V}_1, \ldots, \mathbf{V}_k$, then it must hold that:

$$p^{\mathcal{S}}_D(\mathbf{Q}) \leq \sum_{i=1,\ldots,k} p^{\mathcal{S}}_D(\mathbf{V}_i) \tag{1}$$

If this condition fails, then there exists an arbitrage situation: the buyer who wants to purchase $\mathbf{Q}$ has no incentive for paying the official price $p^{\mathcal{S}}_D(\mathbf{Q})$; instead she will purchase the view bundles $\mathbf{V}_1, \ldots, \mathbf{V}_k$ and derive the answer to $\mathbf{Q}$ herself. Key in this definition is the concept of *answerability*, also called *determinacy* [18, 13]: a query bundle $\mathbf{Q}$ can be answered from the queries in the bundle $\mathbf{V} = \mathbf{V}_1 \cup \ldots \cup \mathbf{V}_k$, in notation $D \vdash \mathbf{V} \twoheadrightarrow \mathbf{Q}$, if there exists a function that, given the answers to all queries in $\mathbf{V}$, computes the answers to all queries in $\mathbf{Q}$. Thus, there exists some way for the buyer to answer $\mathbf{Q}$ from $\mathbf{V}$, without access to the database D. The function needs to be correct only in the current state of the database D. In the rest of the paper, we will use the following alternative characterization of determinacy [13]: a set of views $\mathbf{V}$ determines a query $\mathbf{Q}$, $D \vdash \mathbf{V} \twoheadrightarrow \mathbf{Q}$, if for any database D' such that $\mathbf{V}(D') = \mathbf{V}(D)$, $\mathbf{Q}(D) = \mathbf{Q}(D')$.

For example, if $\mathbf{V}$ has two views, $R \bowtie S$ and $S \bowtie T$, and $\mathbf{Q}$ has one query, $R \bowtie S \bowtie T$, then for any database, $D \vdash \mathbf{V} \twoheadrightarrow \mathbf{Q}$: indeed, to answer $\mathbf{Q}$, simply perform a natural join of the two views. An arbitrage-free pricing function must be such that $p^{\mathcal{S}}_D(\mathbf{Q}) \leq p^{\mathcal{S}}_D(\mathbf{V})$. On the other hand, if we modify the two views to project out some attributes in S, then, in general, $\mathbf{V}$ no longer determines $\mathbf{Q}$, since the query can not be recovered from the views. In this case, the system may charge more for $\mathbf{Q}$ than for $\mathbf{V}$. However, assume that, in the current state of the database, one of the views is empty: then we have determinacy again, $D \vdash \mathbf{V} \twoheadrightarrow \mathbf{Q}$, because a savvy buyer can infer from $\Pi(R \bowtie S) = \emptyset$ that $R \bowtie S \bowtie T = \emptyset$, and the system must ensure $p^{\mathcal{S}}_D(\mathbf{Q}) \leq p^{\mathcal{S}}_D(\mathbf{V})$. Thus, even if the seller defines the price points independently of the database D (only dependent on the priced views), an arbitrage-free pricing function may depend indirectly on the database.

Discount-free: The prices offer no additional discounts than the ones specified by the data seller; equivalently, the pricing function is *maximum* among arbitrage-free prices.

A pricing function with such properties (if it exists) can be computed using the *arbitrage-price*. Recall that $\mathcal{S}$ are the price points explicitly defined by the seller. The *support* of a query bundle is:

$$supp^{\mathcal{S}}_D(\mathbf{Q}) = \{\mathcal{C} \subseteq \mathcal{S} \mid D \vdash \{V_i \mid (V_i, p_i) \in \mathcal{C}\} \twoheadrightarrow \mathbf{Q}\} \tag{2}$$

The support is the set of all subsets of price-points $\mathcal{C} \subseteq \mathcal{S}$ that are sufficient to answer $\mathbf{Q}$.

Definition 2.1 (Arbitrage-price). *The* arbitrage-price *of a query bundle* $\mathbf{Q}$ *is:*

$$p^{\mathcal{S}}_D(\mathbf{Q}) = \min_{\mathcal{C} \in supp^{\mathcal{S}}_D(\mathbf{Q})} \sum_{(\mathbf{V}_i, p_i) \in \mathcal{C}} p_i \tag{3}$$

The arbitrage-price represents the strategy of a savvy buyer: to purchase $\mathbf{Q}$, buy the cheapest support $\mathcal{C}$ for $\mathbf{Q}$, meaning the cheapest set of views that determine $\mathbf{Q}$. The arbitrage-price is always arbitrage free [13]; moreover, if it agrees with the seller's price points, then it is also the unique arbitrage-free, discount-free pricing function consistent with the seller's constraints. In other words, a pricing system must always compute the arbitrage price, Equation 3. Notice also that the arbitrage-price by definition is independent of the query plan that is used to compute the query.

Example 2.2. *Continuing Example 1.1, we describe how a data seller, Bob, sets the price points for the data set. We will use selections as price points. Consider first $T_{en,de}$. Bob charges \$0.01 for every selection query of the form $V_a(x, y) = \sigma_{x=a}(T_{en,de}(x, y))$, for any word a. He charges \$0.02 for each selection of the form $\sigma_{y=a}(T_{en,de}(x, y))$. Essentially, a data consumer will pay \$0.01 to translate any English word to a German word, and \$0.02 for the inverse.*

As for the word frequency dataset, Bob prices two kinds of selections for the table UF: $\sigma_{w=a}(UF(w, g, f, r))$ for \$0.05, whereas the price of $\sigma_{r=n}(UF(w, g, f, r))$ depends on the rank. For the table BF, Bob chooses to price each selection on 2 attributes: $\sigma_{w_1=a, w_2=b}(BF(w_1, w_2, g, f, r))$ for \$0.1.

Under such price points, one can show (using results in [13]) that Q_2 can be priced efficiently in PTIME. However, Q_4 is NP-hard to price. Further, the pricing complexity of queries such as Q_3 or Q_5 is not formally known.

For the rest of the paper, we consider a database D, and allow the data seller to specify the price points only as *selection queries* over a single or multiple columns, as in Example 2.2. We assume that, for each column X of table R that the seller has specified prices, there exists a finite domain, denoted by $Col_{R.X}$: this *column*[1] is known to both

[1]The notion of a column should not be confused by the notion of the domain or the active domain of the database. To illustrate the distinction, consider a relation attribute representing English words. The domain is the set of strings of a given length, the active domain is the set of words in the database, the column is the set of all English words.

the seller and the buyer, and D is required to include values only from this column. The seller can specify explicit prices only for the values in the appropriate column. For a value $a \in Col(R.X)$, the explicit price of the selection $\sigma_{R.X=a}$ will be denoted by $p(\sigma_{R.X=a})$.

For each relation, we only require that at least one (and not necessarily all) attribute be priced. We also require that, for any priced attribute, prices are set for all values in the column. Otherwise, the price of the relation itself cannot be specified. For a relation R, let $att\_pr(R)$ denote the set of attributes that are priced by the seller; then, $att\_pr(R) \neq \emptyset$.

2.1 Alternative Pricing Models

We compare the framework presented above with two simple pricing schemes: (a) pay-per-tuple, and (b) sell only the pre-defined priced views. The comparison will be performed across three dimensions: system infrastructure, customer satisfaction and revenue for the seller.

In pay-per-tuple the buyer pays a uniform price for each output tuple. This is simple to implement and predictable: the more tuples returned, the higher the price. However, this scheme cannot put a price on negative information, which can be valuable. For example, a financial analyst sells stock recommendations, `StrongBuy(symb)`, and a user retrieves these stocks and their current price, $q(x, y) = \texttt{StrongBuy}(x), \texttt{List}(x, y)$. The query returns few answers, yet the user must pay dearly for them. The list of *all* stocks `List(symb,price)` is a superset of the query, but is freely available or cheap. Clearly, paying the same price per tuple makes no sense here from the seller's perspective. A more refined version of this scheme can differentiate the price paid for each tuple by charging each tuple depending on its provenance. While this fixes our toy example, it fails in more complex scenarios, e.g. when the user intersects the recommendations of two analysts, $q(x) = \texttt{StrongBuy1}(x), \texttt{StrongBuy2}(x)$. The absence of a stock from either recommendation is valuable important information, yet it is not captured by the provenance.

In fixed-views, the buyer can only purchase the views that are listed by the seller; if she needs to answer a query that is not on the list, she needs to determine herself which views to buy in order to answer her query. One advantage is that users obtain the entire views that determine their query and not just the final answer, at no additional cost and they may reuse these views for future queries. However, it is possible to achieve the same advantage in QueryMarket by recording the user's query history, Section 5. On the other hand, the fixed-views scheme, although simple to deploy, has three disadvantages. First, it moves the burden of computing the optimal set of views from the market maker to the buyer, who has to either do this manually (which is impractical), or could use a tool but that would be equivalent to our QueryMarket, simply running on the client rather than the server. Second, determining the optimal set of views to compute a query requires access to the current state of the database. For example, if the query is $q(x) = \texttt{StrongBuy1}(x), \texttt{StrongBuy2}(x)$ and stock A is not recommended by the second analyst, the buyer should buy only the information from the second analyst, and not have to pay the first analyst: QueryMarket can determine that, since it has access to the data, but the buyer does not. Finally, since the buyer cannot compute the optimal set of views to answer a query, she may later discover a cheaper way to buy the same query: this may become obvious by inspecting the views that she purchased, or she may learn this from another user. Such differential pricing is poorly received by customers and should be avoided at all costs [19].

To summarize, simpler pricing schemes either fail to capture the pricing needs of the seller or may defer the non-trivial problem of selecting the views to generate an answer to the buyer and hence making them impractical for buyers.

3. PROBLEM FORMULATION

We aim to develop a *practical* system for query-based data pricing. We build on the above theoretical framework and, to build a system, address the following three challenges.

Problem 1: Practical pricing of a large class of queries.

As discussed above, only a small class of queries (a subset of Conjunctive Queries) can be priced in PTIME. All other queries are NP-hard, even when price points are simple selection queries over single attributes. Our goal with building the QueryMarket system is to overcome this challenge and show that computing the price is a feasible task for a large class of queries common in practice. In particular, we demonstrate in Section 7 that our approach can price a large variety of queries over a real dataset in a few seconds.

We focus on the class of queries that can be expressed as Unions of Conjunctive Queries (UCQs) with interpreted predicates. This class of queries is of particular interest because it captures a large part of SQL (except for negations and aggregations): it captures SELECT-PROJECT-JOIN (SPJ) queries including theta joins, disjunctions, and conjunctions. Our approach also supports the pricing of user-defined functions. In our system, a user submits a query of the form $Q(x_1, \ldots, x_\ell) = R_1(\ldots), \ldots, R_n(\ldots)$, where the head Q of the query contains a (possibly empty) subset of variables from the query body. Each atom R_i in the query body is either a relation for sale available in the database D or the head of another rule in the query.

In order to compute prices, we take advantage of efficient state-of-the-art ILP solvers (*e.g.* GLPK, CPLEX, Mosek; we used GLPK [6] in Section 7) by casting the pricing problem as an integer linear program. Let us recall that to price the query, the system must compute the minimum-cost support for that query (Equation 3). The objective function of the ILP is then to minimize $\sum_{(V_i,p_i)\in\mathcal{S}} p_i x_i$, where (V_i, p_i) are all the price points in $\mathcal{S}$ and x_i is an indicator variable capturing whether the view V_i is being purchased to answer the query Q. The constraints of the ILP are expressed over the variables x_i and must be such that the set of purchased views forms a support for the query. The challenge is in determining how these constraints should be stated in order to achieve that goal.

Problem 2: Handle system dynamism.

The second problem that we address in the QueryMarket is that of system dynamism. First, we consider the problem of pricing a sequence of query bundles, $\mathbf{S} = \langle \mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_k \rangle$, where the buyer purchases one query bundle $\mathbf{Q}_i \in \mathbf{S}$ at a time until all bundles in the sequence have been purchased. The ideal price that the buyer would like to pay is equal to purchasing all the bundles at once, namely $p_D^{\mathcal{S}}(\mathbf{Q}_{1:k})$, where $\mathbf{Q}_{1:k} = \bigcup_{i=1,\ldots,k} \mathbf{Q}_i$. But if she issues the

queries one by one (over time), her total price is $\sum_i p_D^S(\mathbf{Q}_i)$. By arbitrage-freeness, the ideal price never exceeds the total price, and sometimes it can be strictly less, because it avoids double charging the buyer for the same data item; in an extreme case, if the buyer asks the same query repeatedly, $\mathbf{Q}_1 = \ldots = \mathbf{Q}_k = \mathbf{Q}$, then the ideal price is $p_D^S(\mathbf{Q}_{1:k}) = p_D^S(\mathbf{Q})$ (this follows from arbitrage-freeness), while the total price is k times larger. One would like the pricing system to account for past queries, and always charge the buyer the ideal prices, thus avoiding double charging. We demonstrate, however, that, in a dynamic setting, this is computationally too expensive, and propose a set of alternatives that closely approximate this ideal goal.

Second, we consider the problem of updates to the database D. We propose a flexible mechanism that enables sellers to determine their preferred pricing strategy for updated views that have previously been purchased by a buyer. To price updates, sellers define a function $p^u(V) \leq p(V)$ for each view. This function can be equal to zero or can be non-zero and can depend on a variety of parameters including time or the extent of the updates to V.

Problem 3: Ensure fairness of revenue sharing.

Finally, we consider an extension of the setting, where the data is contributed by multiple sellers. Thus, when a buyer issues a query, her answer integrates information from multiple vendors, and the issue is how to share the revenue fairly among the vendors. The arbitrage-free pricing function (Equation 3) is ignorant of multiple sellers: it finds a set of price points $\mathcal{C}$ of minimal total price, whose views are sufficient to answer the query. However, the set $\mathcal{C}$ is often not unique, and different choices correspond to wildly different ways of splitting the revenues between sellers. For a simple example, if the buyer purchases $\sigma_p(R) \bowtie \sigma_q(S)$, and the result is empty because both $\sigma_p(R)$ and $\sigma_q(S)$ are empty, then her payment can either be sent to the vendor of R (to pay for the information that R contains no records satisfying the predicate p), or to the vendor of S. A data market service must ensure *fairness* for the revenue sharing between sellers. In Section 6, we demonstrate that ignoring this problem by arbitrarily picking a solution can easily lead to unfairness, and develop a policy, FairShare, that ensures that each seller will be treated fairly in terms of revenue.

4. A PRACTICAL IMPLEMENTATION OF THE STATIC PRICING FRAMEWORK

In this section, we describe an efficient implementation of the pricing framework in the setting where a user issues a single query bundle for pricing.

4.1 ILP for Full Conjunctive Queries

The first step of the construction is to show how to construct the ILP for a full conjunctive query. In Datalog terminology, a full CQ is a query where every variable appears in the head. For example, Q_2 (Section 1) is a full CQ.

Given a full CQ $Q(x_1, \ldots, x_k)$, any tuple $t \in Col_{x_1} \times \cdots \times Col_{x_k}$ is a possible answer to Q (there are polynomially many such answers). Let t_R be the projection of a tuple $t \in Q(D)$ on the relation R in Q. Also, let $t[R.X]$ be the value of t at position $R.X$. Moreover, we say that a set of selection views $\mathbf{V}$ *covers* a tuple t_R if for some priced attribute $R.Y \in att\_pr(R)$, we have that $\sigma_{R.Y=a} \in \mathbf{V}$ and $t_R[R.Y] = a$. As an example, consider the set $\mathbf{V} = \{\sigma_{A=a_1}(R), \sigma_{A=a_1}(S)\}$ from Table 1. $\mathbf{V}$ covers both tuples $R(a_1), S(a_1, b)$, but not the tuples $S(a_2, b), S(a_3, b)$.

We now distinguish two cases depending on whether $t \in Q(D)$ or not. If $t \in Q(D)$, any set of views $\mathbf{V}$ that determines Q must make sure that any projection t_R of t to an atom R in Q belongs in R^D. To achieve this, $\mathbf{V}$ must cover t_R. In the case where $t \notin Q(D)$, $\mathbf{V}$ must discover a witness that t does not belong in the answer; in other words, $\mathbf{V}$ must discover an atom R such that $t_R \notin R^D$ and $\mathbf{V}$ must cover t_R. This intuition is formally captured by the following theorem.

Theorem 4.1. *Let $Q(x_1, \ldots, x_k)$ be a full CQ, D be a database instance and $\mathbf{V}$ be a set of selection views s.t. $D \vdash \mathbf{V} \twoheadrightarrow Q$. Consider a tuple $t \in Col_{x_1} \times \cdots \times Col_{x_k}$.*

1. *If $t \in Q(D)$, then for each atom R in Q, there exists some priced attribute $R.X \in att\_pr(R)$ such that $\sigma_{R.X=t[R.X]} \in \mathbf{V}$.*
2. *If $t \notin Q(D)$, $\mathbf{V}$ includes at least one of the selections $\sigma_{R.X=t[R.X]}$, where $R.X \in att\_pr(R)$ and $t_R \notin D$.*

Proof. For (1), assume that the claim does not hold. Then, for a tuple $t \in Q(D)$, there exists an atom R such that all selections of the form $\sigma_{R.X=t[R.X]}$, for any $R.X \in att\_pr(R)$, do not appear in $\mathbf{V}$. Notice that $t_R \in D$: consider a database D' where $D = D' - \{R(t_R)\}$. Clearly, $\mathbf{V}(D) = \mathbf{V}(D')$, since t_R does not belong in any of the views in $\mathbf{V}$. However, $t \notin Q(D')$, a contradiction.

For (2), assume again that the claim does not hold. Then, for any projection $t_R \notin D$, $\mathbf{V}$ does not include any of the views $\sigma_{R.X=t[R.X]}$, where $R.X \in att\_pr(R)$. Let D' be $D \cup \{R(t_R) \mid t_R \notin D\}$. Again, $\mathbf{V}(D) = \mathbf{V}(D')$, but now $t \in Q(D')$. □

Exploiting Theorem 4.1, we can construct the linear program as follows. We introduce a variable $x_{R.X,a}$ for each selection of the form $\sigma_{R.X=a}$, where $R.X \in att\_pr(R)$ and $a \in Col_{R.X}$. The variable $x_{R.X,a}$ takes values from $\{0, 1\}$: it is 1 if the corresponding view is purchased, otherwise it is 0. Hence, the objective of the lLP is to minimize the total price captured by the following expression (where $att(Q)$ denotes the priced attributes of atoms in Q):

$$\textit{minimize} \quad \sum_{R.X \in att(Q)} \sum_{a \in Col_{R.X}} p(\sigma_{R.X=a}) x_{R.X,a}$$

We next discuss how to add the constraints for the ILP. There are two cases. (1) For each potential tuple $t \notin Q(D)$, we need to add one constraint as follows:

$$\forall t \notin Q(D): \quad \sum_{t.R \notin D, R.X \in att\_pr(R)} x_{R.X,t[R.X]} \geq 1$$

Notice that this forces, for any projection $t.R \notin D$, any solution to buy at least a selection view that will cover $t.R$. (2) For each tuple $t \in Q(D)$, and for each atom R in Q, we add the following constraint:

$$\forall t \in Q(D), \forall R \in \texttt{atoms}(Q): \quad \sum_{R.X \in att\_pr(R)} x_{R.X,t[R.X]} \geq 1$$

This construction produces an ILP that solves pricing for any full Conjunctive Query. Furthermore, notice that the construction does not prohibit the existence of constants in the CQ or self-joins (*i.e.* when a relation appears more than once in the query).

Schema: $R(A), S(A,B), T(A)$; **Domains**: $Col_A = \{a_1, a_2, a_3\}$, $Col_B = \{b\}$; **Database**: $R(a_1), S(a_1,b), S(a_2,b), T(a_1), T(a_2)$
Price points: $p(\sigma_{R.A=a_1}), p(\sigma_{R.A=a_2}), \ldots, p(\sigma_{S.A=a_1}), \ldots, p(\sigma_{S.B=b}), \ldots, p(\sigma_{T.A=a_1}), \ldots$
Objective Function: $\sum_{i=1}^{3} p(\sigma_{R.A=a_i}) x_{R.A,a_i} + \sum_{i=1}^{3} p(\sigma_{S.A=a_i}) x_{S.A,a_i} + p(\sigma_{S.B=b}) x_{R.B,b} + \sum_{i=1}^{3} p(\sigma_{T.A=a_i}) x_{T.A,a_i}$
Constraints: Query specific, as described below.

Type	Query	Generated ILP Constraints	Intuition
Full Queries	`Q(u,v) = R(u),S(u,v)`	$(a_1,b) \in Q$: $x_{R.A,a_1} \geq 1,\ x_{S.A,a_1} + x_{S.B,b} \geq 1$ $(a_2,b) \notin Q$: $x_{R.A,a_2} \geq 1$ $(a_3,b) \notin Q$: $x_{R.A,a_3} + x_{S.A,a_3} + x_{S.B,b} \geq 1$	(a_1,b) is in the answer, so we must purchase both $R(a_1)$ and $S(a_1,b)$ (by either buying $\sigma_{S.A=a_1}$ or $\sigma_{S.A=b}$); (a_2,b) is not in the answer, and the only way to find this out is to learn that $R(a_2)$ is not in the database; (a_3,b) is not in the answer, and we can learn this by learning that either $R(a_3)$ or $S(a_3,b)$ is not in the database.
Projections	`Q(v) = T(u),S(u,v)`	$(a_1,b) \in Q^f$: $x_{T.A,a_1} \geq z_1,\ x_{S.A,a_1} + x_{S.B,b} \geq z_1$ $(a_2,b) \in Q^f$: $x_{T.A,a_2} \geq z_2,\ x_{S.A,a_2} + x_{S.B,b} \geq z_2$ $b \in Q$: $z_1 + z_2 \geq 1$	To find out that b is in the answer, we need to learn either about (a_1,b) or (a_2,b): in each case, we cover the tuple as in the example on the previous line.

Table 1: An example on the ILP construction for queries on a database with three relations $R(A), S(A,B), T(A)$.

4.2 Extensions

In this subsection, we show how to extend the ILP construction for full conjunctive queries to include projections, UCQs, interpreted constraints, bundles, key constraints, as well as user-defined functions (UDFs).

Projections. Consider a query $Q(x_1, \ldots, x_\ell)$, where the head variables are a strict subset of the body variables (or even an empty set if Q is boolean). Let Q^f be the corresponding full query of Q, *i.e.*, Q^f contains all the body variables as head variables. Take a tuple $t \in Col_{x_1} \times \ldots Col_{x_\ell}$ and consider two cases.

If $t \in Q(D)$, for every tuple $t^f \in Q^f(D)$ that projects to t, introduce a constraint as in Q^f, with the only difference that the right side of the inequality is a new variable u_{tf} (and not 1). Finally, introduce an additional constraint: $\sum_{tf} u_{tf} \geq 1$. Hence, every feasible solution has to satisfy the latter constraint, which means that one of the u_{tf} will be set to 1: thus, the solution has to discover at least one t^f, which will imply that t will also be discovered as an answer. If $t \notin Q(D)$, for every tuple $t^f \notin Q^f(D)$ that projects to t, we construct the inequality constraint for Q^f as before; we need no additional constraints in this case. Indeed, if any of these inequalities fails to be satisfied, we do prove that t^f (and hence t) is indeed a non-answer.

Table 1 gives an illustrative example of the construction.

Interpreted Constraints. Here we address arbitrary selection conditions. The conditions may be unary (*e.g.* selections of the form $x = 1$) or of higher arity (*e.g.* $x = y$), and they may involve any conditional operator (*e.g.* inequalities $x < y$). Common queries that are used in practice are *theta joins* of the form $Q(x,y) = R(x), S(y), \theta(x,y)$, where θ is a predicate: *e.g.* $x > y$, $x \leq y$, $|x - y| < 2$ (band join).

For unary constraints, we assume a predicate $C(x)$. In this case, instead of using Col_x in the construction of the ILP, we use the *filtered* column $Col'_x = \{a \in Col_x \mid C(x)\}$. This transformation can be viewed as pushing the selection operator down the query plan, since it reduces the number of variables in the ILP. More generally, introducing a constraint of arbitrary arity $\theta(x_{i_1}, \ldots, x_{i_m})$, where $m \leq k$, modifies only how we define a potential tuple when constructing the ILP. Indeed, a potential tuple is now defined as $t \in \{\mathbf{a} \in Col_{x_1} \times \cdots \times Col_{x_k} \mid \theta(a_{i_1}, \ldots, a_{i_m})\}$.

Query Bundles. Assume we want to formulate an ILP for a query bundle $\mathbf{Q} = \{Q_1, \ldots, Q_k\}$. For each query Q_i, we construct independently the ILP constraints C_i. The constraints for the bundle $\mathbf{Q}$ will be the set of constraints $\{C_i \mid Q_i \in \mathbf{Q}\}$. As for the objective function of the ILP, this will be to minimize the sum of $p(\sigma_{R.X=a}) \cdot x_{R.X,a}$, where the sum is over any priced attribute for some relation that appears in some Q_i and $a \in Col_{R.X}$.

Selections on Multiple Attributes. We can formulate the ILP even in the case where price points are defined as selections over multiple attributes. For example, consider the bigram dataset in Example 1.1: it is reasonable for the data seller to price the selections on both words, *i.e.* $\sigma_{BF.w_1=a,BF.w_2=b}(BF)$. To incorporate this extension in the ILP, we can generalize Theorem 4.1 to take into account that a tuple t can now be *covered* not only by single-attribute, but also multiple-attribute selections. The generalization is straightforward and is omitted due to space constraints.

Unions of Conjunctive Queries. We will describe the ILP construction for a query $Q = Q_1 \cup Q_2$ that is the union of two CQs, Q_1, Q_2; the process can be easily generalized for the union of more than 2 queries.

For every potential tuple t, we distinguish several cases. If $t \notin Q(D)$, then it is neither in $Q_1(D)$ or $Q_2(D)$, so we construct the ILP constraints separately for Q_1, Q_2 as previously discussed. If $t \in Q(D)$, we distinguish two cases. If t belongs in $Q_1(D)$ and not in $Q_2(D)$, we introduce the constraint for $t \in Q_1(D)$ (symmetrically if $t \in Q_2(D)$ and not in $Q_1(D)$). Finally, if t belongs in both $Q_1(D), Q_2(D)$, we introduce the constraints for $Q_1(D), Q_2(D)$ as before, but now on the right side of the inequalities for Q_1 we replace 1 by a new binary variable $u_{t,1}$ and for Q_2 by a new binary variable $u_{t,2}$. Additionally, we require that $u_{t,1} + u_{t,2} \geq 1$.

Key Constraints. We discuss how to incorporate *key constraints* in our framework.

If the database schema defines a certain attribute as the key of a relation, we can take this information into account to lower the price offered to potential customers. Indeed, the key constraint is an additional piece of information about the database that the buyer knows, hence the buyer potentially needs less information to answer the same question without keys. Since the prices depend solely on information, less information implies a lower price.

EXAMPLE 4.2. *Assume that a data owner wants to sell the table $R(X,Y)$, where the instance is $R^D = \{(a_1, b_1)\}$, $Col_X = \{a_1\}$ is priced to \$10 and $Col_Y = \{b_1, b_2\}$ is priced to \$3 each. Assume we want to price the query $Q(x,y) = R(x,y)$. If no keys exist, then $\sigma_{Y=b_1}, \sigma_{Y=b_2}$ are the cheapest views that determine Q; hence, $p(Q) = 6$. Now, assume that the attribute X is a key for R. Then, purchasing the*

view $\sigma_{Y=b_1}$ suffices to determine Q, since the key constraint restricts a_1 to map to a single value of Y, which is b_1. In this case, $p(Q) = 3$.

Assume that the attribute X is a key for $R(X, Y_1, \ldots, Y_k)$. Let us construct the constraints for the ILP as we have described before, but now we replace each variable $x_{R.X,a}$ with a new variable $y_{R.X,a}$. Furthermore, let $a \in Col_{R.X}$ map to the unique values $b_1, \ldots, b_k$. For each such value a, we introduce the following constraint:

$$y_{R.X,a} \leq x_{R.X,a} + \sum_{R.Y \in pr(R) \setminus X} x_{R.Y,b_i}$$

In other words, the answer to the selection view $\sigma_{R.X=a}$ is the single tuple $(a, b_1, \ldots, b_k)$, and, in order to buy it, it suffices to either buy it as is, or buy any of the other selection views on some b_i.

UDFs. We discuss how to model the pricing of user-defined functions in our pricing system. As a motivating example, suppose that a data seller wishes to sell a function $hashtag(w)$, which determines whether the word w is a Twitter hashtag or not. The idea is to represent the function as a database relation $isHashtag(X)$ that contains all the hashtags and then define a price point for each selection of the form $\sigma_{X=w} isHashtag(X)$, where w is a possible word. Thus, by representing any function $f(x_1, \ldots, x_k)$ for sale as a relation $is_f(X_1, \ldots, X_k)$, we can incorporate UDFs into our pricing system.

4.3 Optimizing the ILP

The resulting ILP can be typically large. Recall that we introduce a variable for every value of every attribute used for pricing. Moreover, the number of constraints varies according to the structure of the query and the attributes that are priced: for example, for a typical join $R(x, y) \bowtie S(y, z)$ where all columns are priced, the number of constraints will be $|Col_x| \cdot |Col_y| \cdot |Col_z|$. To overcome the problem of constructing a huge ILP, we develop several optimization techniques that reduce the size of the resulting ILP.

4.3.1 Independent Subproblems

This technique allows us to split the ILP into one or more smaller ILPs, solve each ILP independently and then efficiently combine their solutions. There are two cases where this technique can be applied.

First, when a CQ contains more than one connected component, it is possible to construct an ILP separately for each connected component and then combine their prices. For example, to compute the answer for $Q() = t_{gr}(x, x), t_{fr}(y, y)$, we independently price the components $Q_1() = t_{gr}(x, x)$ and $Q_2() = t_f r(y, y)$. If both are true, we add up their prices, if one is false we output its price and if both are false we output the minimum of the two prices.

This technique can also be applied in the case of bundles: if the queries in the bundle do not share any predicates, we can price each query separately and then add up their prices.

4.3.2 Problem Size Reduction

The second optimization technique we present reduces the number of variables or constraints of the ILP, which accelerates the price computation. We discuss below two cases where such a reduction of the ILP size is possible.

In the first case, the CQ contains a *hanging variable*, which is a variable that appears once in the body of the query. If the query Q is without self-joins any set of views that determines Q must either choose all the views from the hanging variable or none of them [13]. We can thus construct two separate ILPs: the first sets the prices of the views to zero (and automatically satisfies the relative constraints), whereas the second sets the prices to infinity (in practice we can remove the variables from the ILP).

In the second case, a CQ without self-joins contains a variable that appears multiple times in a single atom. Then, we can reduce the size of the ILP by keeping only one of the columns where this variable appears (see [13] for more details).

4.3.3 Coarse-Grained Pricing

To further reduce the size of the constructed ILP, we can use as price points selections not only on single values, but on groups of values or intervals. As we will discuss below, this also solves the problem of setting explicit price points on continuous or large domains.

EXAMPLE 4.3. *A data seller offers for purchase data on weather conditions. In particular, the table $Temp(x, y, t)$ denotes that a location with coordinates (x, y) has an average temperature t during 2011. Note that the column referring to the temperature is continuous and it makes no sense for the seller to set a price for specific values of the domain. The same problem appears when someone sells data on revenue for companies in the USA; even though the domain is not continuous, it is still impractical to price every value.*

To overcome these two problems, we propose a solution that is based on a more *coarse-grained* pricing. Instead of specifying price points as selections on single values, price points are specified as selections over disjoint groups of values that cover the column. This grouping can refer to a natural partition of a column into intervals, or, in the case of a finite column, an arbitrary partition of the values into groups.

EXAMPLE 4.4. *Continuing Example 4.3, a seller can partition the temperature column of the Temp table into ranges of 10 degrees Fahrenheit, $[-100, -90), \ldots, [90, 100)$, with two additional ranges $(-\infty, -100)$ and $[100, +\infty)$. A coarser pricing would partition into groups of 20 degrees. Moreover, the seller could also group price points on coordinates. For example, a reasonable partition would be to group coordinates based on zip code.*

As we show next, we can still transform the pricing problem into an ILP. However, the construction is not trivial, since joining attributes may be priced at a different granularity or using a different partitioning strategy.

Formally, the data seller can now define price points over some attribute X of relation R as $(\sigma_{R.X \in G_i}(R), p_i)$, where $\{G_i\}_{i=1,k}$ denotes a partition of $Col_{R.X}$ into k disjoint groups that cover $Col_{R.X}$.

Generalized Price Points. We next present an algorithm to compute the price in the case where the price points are groups of values and not single values. The algorithm transforms the problem to a single-value selection instance, which we have previously discussed how to price.

Let D be the database, $P_{R.X}$ be the partition into groups for an attribute $R.X$, and $\mathcal{S}$ be the price points in this case. Moreover, for a value $a \in Col_{R.X}$, let $G(a)$ be the (unique)

group that a belongs to. We construct a database D' and price points $\mathcal{S}'$ as follows. The schema remains the same. For a relation R, we replace the column $Col_{R.X}$ for D with a column $Col'_{R.X}$, where each value in $Col'_{R.X}$ corresponds to a group in $P_{R.X}$. The price points are set such that the price of the selection is the same as the price of the group in $\mathcal{S}$. To construct the database D', for a tuple $t = (a_1, \ldots, a_k) \in R^D$, we introduce $G(t) = (G(a_1), \ldots, G(a_k)) \in R^{D'}$.

Finally, assume a given conjunctive query Q. We construct the corresponding query Q' for $\mathcal{S}'$ and D' as follows. If x is a join variable that appears in atoms $R_1, \ldots, R_m$, we replace its occurrence with $x^1, \ldots, x^m$ in $R_1, \ldots, R_m$ respectively and also introduce the predicate $\cap_{i=1} x^i \neq \emptyset$, *i.e.* that the conjunction of the corresponding groups must be non-empty. We prove next the validity of our construction.

THEOREM 4.5. *We have that* $p^{\mathcal{S}}_D(Q) = p^{\mathcal{S}'}_{D'}(Q')$.

PROOF. Notice that the price points are in one to one correspondence. Indeed, $\sigma_{R.X \in G}(R)$ is a price point in $\mathcal{S}$ iff $\sigma_{R.X=G}(R)$ is a price point in $\mathcal{S}'$. Hence, we need to show that $D \vdash \mathbf{V} \twoheadrightarrow Q$ iff $D' \vdash \mathbf{V} \twoheadrightarrow Q'$.

First, consider some $\mathbf{V}$ such that $D' \vdash \mathbf{V} \twoheadrightarrow Q'$. We will show that $D \vdash \mathbf{V} \twoheadrightarrow Q$. Indeed, consider a potential tuple $t = (a_1, \ldots, a_k) \in Q(D)$ and take its projection t_R for any atom R. By our construction, since $t_R \in R^D$, $G(t_R) \in R^{D'}$. Now, consider a join variable x which corresponds to the position i for t: let R_1, R_m the atoms that have x. Then, notice that the groups $G_1(a_i), \ldots, G_m(a_i)$ have a non-empty intersection, since a_i belongs in all of them. Thus, $G(t) \in Q'(D')$ and for each $G(t_R)$, at least one view covers it, which implies that at least one view will cover t_R as well. A similar proof holds when $t \notin Q(D)$. For the converse direction, assume that $D \vdash \mathbf{V} \twoheadrightarrow Q$. Then, take any tuple $t \in Q'(D')$ and suppose, for the sake of contradiction, that for any atom R, $\mathbf{V}$ does not cover t_R. Thus, we can find a tuple $a \in R^D$ such that $G(a) = t_R$ and this tuple is not covered by $\mathbf{V}$. Now, we also know that the intersection for the join variables is not-empty; hence, we can use any common value for the intersection and create a counterexample for determinacy. If $t \notin Q'(D')$, a similar construction holds. □

Notice that the transformation of the pricing problem to pricing single-valued selections does not increase the number of price points in the problem. Hence, the size of the resulting ILP depends directly on how coarse the partition of the column is.

Prices. Given a column $Col_{R.X}$, consider two partitions P_1, P_2 of the column such that P_1 is finer-grained than P_2, in the sense that, any group $G^2 \in P_2$ can be covered exactly by a disjoint set of groups from P_1. For example, the partition in Example 4.4 into 5-degree ranges is finer than the one into 20-degree ranges. Moreover, assume that the prices of P_1, P_2 are naturally set such that if a collection of groups $\{G^i\}_{i=1}^m$ from one partition covers a group G from the other partition, $\sum_{i=1}^m p(G^i) \geq p(G)$. In this case, it is easy to see that for any query Q, $p^{P_1}_D(Q) \leq p^{P_2}_D(Q)$ for the same database D[2]. Consequently, a more fine-grained partition will in general lead to lower prices for the buyers.

[2]Indeed, if $D \vdash \mathbf{V}^2 \twoheadrightarrow Q$ for P_2, we can cover each $V \in V^2$ by a collection of groups from P_1 of the same price. The resulting set of views may have redundant views, so the price may be even lower

5. PRICING IN A DYNAMIC SETTING

In the previous sections, we have only considered the case where a buyer purchases a single query bundle and the database is *static*, *i.e.* does not change over time. However, in order to build a practical pricing system, we need to consider the case where a data consumer is interested in several queries, which she issues at different times, and also study the case where the database is modified over time.

In both cases, queries may have overlapping content; hence, the information content that a data consumer has purchased in the past must be taken into account when we compute the price of a new query. In the extreme case that the content of a query is determined by previous queries asked, it must be offered for free, since the user has no information gain.

5.1 History

In this subsection, we discuss and explore various strategies for incorporating query history when computing the price. For the rest of the section, we consider the case of a buyer who wants to purchase a sequence of query bundles $\langle \mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_k \rangle$. Notice that the queries are issued by the buyer in an *online* fashion; hence, we cannot assume that the buyer or the system knows the full query sequence from the start. Also, observe that the price for the information content of this sequence is $p(\mathbf{Q}_{1:k})$, *i.e.* the price of the bundle $\mathbf{Q}_{1:k} = \bigcup_{i=1,k} \mathbf{Q}_i$. For ease of exposition, let us denote by $p(\mathbf{Q}_i)$ the price of the bundle $\mathbf{Q}_i$ as a standalone query bundle.

Oblivious Pricing. This pricing strategy, which we denote by p^O, completely ignores the query history. Formally, when asking for the bundle $\mathbf{Q}_i$, we define $p^O(\mathbf{Q}_i) = p(\mathbf{Q}_i)$. The total cost for the sequence in the end will thus be $p^O(\langle \mathbf{Q}_1, \ldots, \mathbf{Q}_k \rangle) = \sum_i p(\mathbf{Q}_i)$. We will use this naive strategy as the baseline for comparison.

Bundle Pricing. The second strategy is to ensure that, after having asked queries $\mathbf{Q}_1, \ldots, \mathbf{Q}_k$, the user will have paid $p(\mathbf{Q}_{1:k})$, *i.e.* the price of the total bundle. In order to achieve this, the system charges for $\mathbf{Q}_i$, where $i > 1$, the price $p^B(\mathbf{Q}_i) = p(\mathbf{Q}_{1:i}) - p^B(\mathbf{Q}_{1:i-1})$. Moreover, $p^B(\mathbf{Q}_1) = p(\mathbf{Q}_1)$. It is easy to see that the definition implies that, for every $i = 1, \ldots, k$: $p^B(\langle \mathbf{Q}_1, \ldots, \mathbf{Q}_i \rangle) = p(\mathbf{Q}_{1:i})$. Hence, the total cost of the sequence is $p^B(\langle \mathbf{Q}_1, \ldots, \mathbf{Q}_k \rangle) = p(\mathbf{Q}_{1:k})$. Unfortunately, bundle pricing as a strategy is computationally inefficient, since now the price computation depends also on the size of the query history.

View Pricing. The third strategy leverages the query history by storing the information that the buyer has purchased. More precisely, when the system prices a query bundle $\mathbf{Q}_i$, it computes a set of views $\mathbf{V}$ of minimum cost that determines $\mathbf{Q}_i$. The pricing system will then consider that the user has purchased these views, and so will offer them for *free* when the same user issues another query. This strategy can be implemented efficiently, since the system will perform the same computation as pricing a standalone query, with the difference that the already purchased views will now be priced to zero. The only overhead of this strategy boils down to the system storing the views that have been purchased. Let us denote the price assigned by view pricing by p^V.

Conditional Pricing. The final pricing strategy we consider is *conditional pricing*, where we are looking for the minimum cost set of views that, together with $\mathbf{Q}_{1:i-1}$ determines $\mathbf{Q}_i$. For this, we use the following notation: $p^C(\mathbf{Q}_i) =$

$p(\mathbf{Q}_i|\mathbf{Q}_{1:i-1})$. Formally, the price will be equal to the minimum cost set of views $\mathbf{V}$ such that $D \vdash \mathbf{V}, \mathbf{Q}_{1:i-1} \twoheadrightarrow \mathbf{Q}_i$.

This notion is equivalent to the one presented in [15]. We do not explore this case in depth, since the problem becomes computationally very hard. In particular, deciding whether $p(Q|Q') = 0$ is equivalent to whether query Q' determines query Q, which is a problem in general harder than NP, even in the case where the price points are only selections.

When comparing the various pricing strategies, we are interested in two parameters. The first parameter is how efficient the computation of the price is (and whether it is independent from the size of the sequence or not). The second is how close the total price of the sequence will be to the price of the query bundle $\mathbf{Q}_{1:k}$. Regarding the second factor, we can formally show the following lemma.

LEMMA 5.1. *For any sequence* $\mathbf{S} = \langle \mathbf{Q}_1, \ldots, \mathbf{Q}_k \rangle$*:*

$$p(\mathbf{Q}_{1:k}) = p^B(\mathbf{S}) \leq p^V(\mathbf{S}) \leq p^C(\mathbf{S}) \leq p^O(\mathbf{S}) \quad (4)$$

$$p^O(\mathbf{S}) \leq k \cdot p(\mathbf{Q}_{1:k}) \quad (5)$$

PROOF. We first prove the first equation. It is easy to see that any strategy cannot do better than p^B. In order to show that $p^V(\mathbf{S}) \leq p^C(\mathbf{S})$, consider any bundle $\mathbf{Q}_i$ and let $\mathbf{V}$ be the set of views purchased so far by the view pricing strategy. By definition, $D \vdash \mathbf{V} \twoheadrightarrow \mathbf{Q}_{1:i-1}$. Hence, for any set of views $\mathbf{V}_i$ such that $D \vdash \mathbf{V}_i, \mathbf{Q}_{1:i-1} \twoheadrightarrow \mathbf{Q}_i$, it also holds that $D \vdash \mathbf{V}_i, \mathbf{V} \twoheadrightarrow \mathbf{Q}_i$, which implies that p^C will be at least as much as p^V. Finally, we show that $p^C(\mathbf{Q}_i) \leq p^O(\mathbf{Q}_i)$. Indeed, if $D \vdash \mathbf{V} \twoheadrightarrow \mathbf{Q}_i$, then trivially $D \vdash \mathbf{V}, \mathbf{Q}_{1:i-1} \twoheadrightarrow \mathbf{Q}_i$.

To show the second equation, notice that we have $p^O(\mathbf{S}) = \sum_i p(\mathbf{Q}_i) \leq k \cdot \max_i\{p(\mathbf{Q}_i)\}$. However, for any query bundle $\mathbf{Q}_i$, $p(\mathbf{Q}_i) \leq p(\mathbf{Q}_{1:k})$. It additionally holds that the bound is tight for p^V, *i.e.* there exists an instance such that $p^V(\mathbf{S}) = k \cdot p(\mathbf{Q}_{1:k})$. □

In Section 7, we experimentally compare the various strategies in terms of price and performance. We propose *view pricing* as the most suitable strategy for incorporating history, since the performance overhead over the oblivious strategy is small and the price is close to the total bundle price. Further, as we show in the next subsection, view pricing can be used to additionally model dynamic databases.

5.2 Updates

In this subsection, we discuss pricing in the presence of *updates*. Datasets that frequently change are often offered for sale: for example, the $hashtag(X)$ table that contains all the words that are hashtags changes when new hashtags are added in the Twitter stream. Also, a dataset with stock prices may be updated daily.

Our framework does not enforce any constraints on how the prices are related when the database changes; however, if some part of the data has not been modified by the update, the pricing system should not charge the user again.

We use query history and the *view pricing* strategy to formalize this problem. The seller, apart from the base prices, can define an *update price* $p^u(V)$ for each view V, which defines how much a user must pay for updates to a view she has already purchased. Clearly, $p^u(V) \leq p(V)$. Moreover, the update price can be set to zero, *i.e.* after a view has been purchased, each modification is offered for free.

As we have discussed above, the data consumer keeps a log of the views that have been purchased: these views are now free for the user. When the database is updated, QueryMarket updates the prices of the purchased views accordingly: if a view has not been modified, its price remains zero. On the other hand, if a view V is modified, the price to purchase this view will be $p^u(V)$. This formulation allows to transform this problem into a regular pricing problem, thus not increasing the complexity of pricing.

6. REVENUE SHARING

Since the datasets may belong to several sellers, the revenue of a priced query must be shared between the sellers who contributed to answering the particular query Q. If a unique minimum-cost set of views $\mathbf{V}$ determines Q, distributing the profit among the sellers is trivial: the revenue of each seller is the sum of the prices of the views that she owns. However, it might be the case that the pricing problem admits *multiple* minimum-cost solutions.

EXAMPLE 6.1. *Consider* $Q(x, y) = R(x, y), S(y, z), y = 1$ *and assume that only the attributes R.y and S.y are priced uniformly at \$1. Moreover, R and S belong to different sellers. If both* $\sigma_{y=1}(R)$ *and* $\sigma_{y=1}(S)$ *are empty, we can determine Q by purchasing either* $\sigma_{y=1}(R)$ *from the one seller or* $\sigma_{y=1}(S)$ *from the other seller. In both cases, one seller will be rewarded with the total revenue (\$1) and the other will have zero profit.*

We next describe FAIRSHARE, a policy to ensure that the revenue will be fairly shared among the data sellers. Let a query bundle $\mathbf{Q}$ and $Sol(\mathbf{Q})$ the set of all the minimum-cost solutions. For a set of views $\mathbf{V}$ that determines $\mathbf{Q}$ and a seller s, let $\mathbf{V}_s$ be the set of views that belong to seller s. Moreover, for a seller s that may contribute to the price of $\mathbf{Q}$, define by $share(s, \mathbf{Q})$ the maximum revenue that seller s can get among all minimum-cost solutions for $\mathbf{Q}$:

$$share(s, \mathbf{Q}) = \max_{\mathbf{V} \in Sol(\mathbf{Q})} p(\mathbf{V}_s)$$

First, observe that one can compute the share for seller s using our standard ILP techniques, without needing to iterate over all possible solutions. For this, before we formulate the ILP, we give to all the views from seller s a tiny discount: a view that costs p will now cost $(1 - \epsilon)p$ for an arbitrary small ϵ. The solution $\mathbf{V}$ with price $p'(\mathbf{V})$ of the ILP will now give us the share for seller s. Indeed, $p'(\mathbf{V}) = p'(\mathbf{V}_s) + p'(\mathbf{V} \setminus \mathbf{V}_s) = (1 - \epsilon)p(\mathbf{V}_s) + p(\mathbf{V} \setminus \mathbf{V}_s) = (1 - \epsilon)p(\mathbf{V}_s) + p(\mathbf{Q}) - p(\mathbf{V}_s) = p(\mathbf{Q}) - \epsilon p(\mathbf{V}_s)$. Thus, any solution with a larger share than $\mathbf{V}$ would give a smaller price than the optimal price of the ILP, a contradiction.

Intuitively, the share of a seller characterizes how much of the "burden" of answering $\mathbf{Q}$ she can afford. In the case of k sellers $s_1, \ldots, s_k$, it is easy to see that: $share(s_i, \mathbf{Q}) \leq p(\mathbf{Q}) \leq \sum_{i=1,k} share(s_i, \mathbf{Q})$.

Given that we know the shares for each of k sellers that may contribute to $\mathbf{Q}$, FAIRSHARE will distribute the revenue in proportion to the share of each seller, *i.e.* the revenue $rev(s_i, \mathbf{Q})$ of seller s_i will be:

$$rev(s_i, \mathbf{Q}) = \frac{share(s_i, \mathbf{Q})}{\sum_{j=1,k} share(s_j, \mathbf{Q})} \cdot p(\mathbf{Q})$$

Continuing Example 6.1, observe that the share of each seller is exactly \$1. Since the price of the query is also \$1, FAIRSHARE will distribute to each seller an equal part of the total price, *i.e.* \$0.5.

7. EVALUATION

In this section, we experimentally evaluate the QueryMarket pricing system. Our system (implemented in C++) runs as a middleware layer and can be used over any relational database management system (DBMS) supporting SQL: we evaluate our system on top of the lightweight SQLite DBMS. We use the open source GLPK ILP solver [6].

In QueryMarket, the explicit price points specified by the seller are stored as tables in the database itself. More precisely, for a priced attribute $R.X$, we create a table $Price[R.X](value, price)$, which stores the value of the selection attribute, along with the corresponding price. A similar construction holds when the price points refer to multiple attributes. In the case where the price is uniform for a specific attribute, we store only the attribute values.

In order to monitor the price tables introduced by the data sellers, QueryMarket keeps an index table, which we refer to as `price_info`, with the following structure:

$$price\_info(table\_name, column1, column2, column3, uprice, price\_table, owner)$$

The column *uprice* refers to the case where the selection has a uniform price. As one can observe, the table also keeps track of the data owner in order to share the revenue.

In order to construct the ILP constraints, the system constructs and issues SQL queries over the price tables and the actual relations. Further, the system needs to store additional information so as to support the *view pricing* strategy discussed in Section 5. Hence, for each buyer u, QueryMarket keeps a table $Purchased\_Views[u]$ that stores the views that have been purchased by the user so far.

We evaluate the various queries and workloads over the database described in Example 1.1. More precisely, the database and price points are constructed as follows:

- $t_{en,de}(x, y)$: price points for both columns
- $t_{en,fr}(x, y)$: price points for column x
- $wf(w, g, f, r)$: price points for columns r and g
- $hashtag(x)$: price points for the unique column x

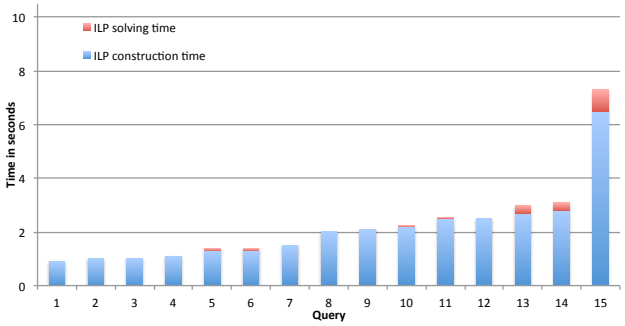
The prices are set uniformly for each attribute to some arbitrary price (the actual price value does not affect the runtime of pricing algorithms).

7.1 Evaluation of the ILP formulation

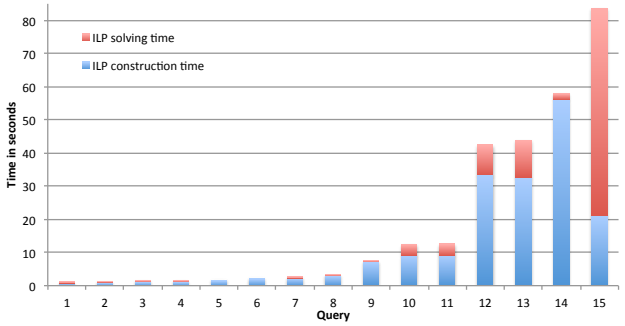
We first study the performance of our price computation algorithm that uses the ILP solver, for various sizes of the database and the columns (the original dataset has size in the order of 10^4, and for smaller sizes we consider subsets of it). In our setting, the size of each column is equal to the size of the projection of the relation on the priced attribute; the sizes of the relations are in general larger than the sizes of the columns, but of the same order of magnitude.

We issue various queries to the pricing system: selections on single attributes with or without projections (Q1 through Q5), selections with interpreted predicates (Q6), 2-way joins without projections (Q7-Q11), 2-way joins with projections (Q12-Q14) and 3-way joins (Q15). We conduct experiments to measure the time to price each query for columns with 10^2 and 10^3 rows[3]. The results are depicted in Figure 1,

[3]Our measurement study of Many Eyes shows that 90% of datasets contain fewer than 1000 rows [16].



(a) Price columns with size 10^2. The construction time and total time lines almost overlap in this case.



(b) Price columns with size 10^3.

Figure 1: Times for ILP construction and ILP solving for price columns of size 10^2(a) and 10^3(b), where the queries are in ascending order of total computation time.

where both the time to construct and the time to solve the ILP are measured.

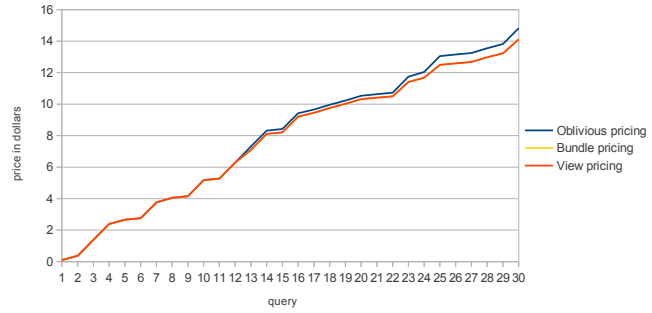
As the figure shows, for the small-scale datasets, pricing is interactive: all but two queries are priced within 3 seconds. The slower two queries still take less than 10 seconds to price. In the larger configuration, 9 out of 15 queries are priced within 10 seconds, and all but one are priced within a minute. While these runtimes are no longer interactive, they are still drastically faster than the current approach, which requires human-to-human negotiations. The slower queries in both cases correspond to 2-way joins (for size 10^3), where the algorithm has to perform heavier computation in order to construct the ILP.

Discussion. The efficiency of our pricing algorithm depends heavily on the attributes that are priced. In general, it is a good practice to price as few attributes per relation as possible: pricing only one attribute (*i.e.* giving only one access point to the buyer) will make the ILP extremely efficient even for very large instances. If a buyer wishes to offer multiple access points to some relation, one strategy is to create copies of the same relation and for each copy offer a different access attribute: any buyer will not be aware that the data belongs in the same relation (that is exactly what the Azure Marketplace does when selling translations).

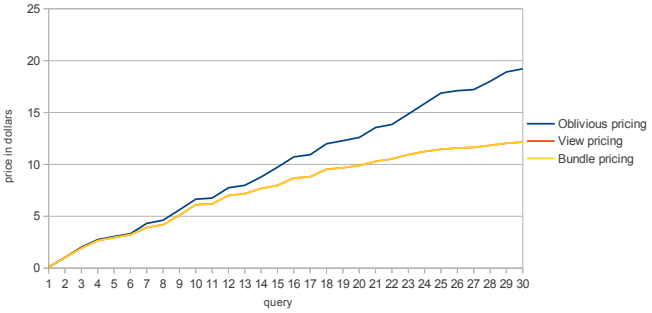
7.2 Query History

In this subsection, we evaluate the various strategies that we proposed to leverage query history discussed in Section 5 based on two parameters: performance and comparison of the prices to the optimal price, which is the price of the total bundle. The size of each column is in the order of 10^3 values.

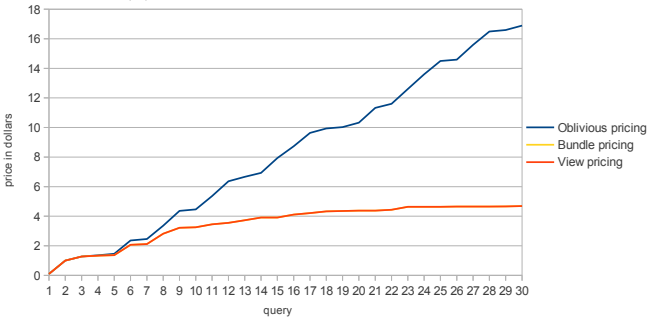
In our experiment, we price three different query sequences of 30 queries $\langle Q_1, \ldots, Q_{30} \rangle$, which consist of a ran-



(a) Weakly overlapping query sequence



(b) Moderately overlapping query sequence



(c) Highly overlapping query sequence

Figure 2: Comparison of the pricing strategies. Oblivious pricing overcharges the users by 5% in (a) to 260% in (c). Bundle pricing computes the ideal price, but is too expensive to compute. View-pricing is easier to compute, and approximates bundle pricing almost perfectly (the graphs are overlapped in (a),(b) and (c)).

dom choice of selections on $t_{en,de}$ and $t_{en,fr}$ and joins between these two datasets and the *hashtag* table. Each query in the sequence is parametrized to access a small random set of English words. The three sequences differ in the amount of overlap between the sets of words: *weakly, moderately and highly overlapping*. We vary the overlap by varying the size of the set from which the set of English words is sampled: for the weakly overlapping set it is the whole column, for the moderately overlapping 20% of the column, and for the highly overlapping 0.05% of the column.

Figure 2a, Figure 2b and Figure 2c show the cumulative price at each point of the sequence for the following three pricing strategies: oblivious pricing (p^O), view pricing (p^V) and bundle pricing (p^B).

A first observation is that, in all three cases, the price for p^V is the same as p^B (although this will not always be the case). Moreover, in Figure 2a, since the information between the queries is essentially not overlapping, the price computed by all strategies is approximately the same, as expected. In contrast, in Figure 2c, where the information content is highly overlapping, the total price remains virtually constant after some point, which means that incoming queries can be answered almost for free, by using the views the user has already purchased. In this case, the user ends up paying 8X less than paying for each query independently.

Table 2: Performance comparison for the history pricing strategies.

Query Overlap	Oblivious	View	Bundle
Weak	142 sec	213 sec	> 20 min
Moderate	74 sec	129 sec	> 20 min
High	138 sec	165 sec	> 20 min

Finally, we evaluate the time that it takes to price each of the three query sequences using each of the three pricing strategies. Table 2 shows the results. As the table shows, the overhead induced by storing and updating the purchased views is small: the time to price the whole sequence is only 20% to 75% longer than the time to price the same sequence using the oblivious pricing method. On the other hand, using bundle pricing makes price computation very costly, since the time to price the bundle $Q_{1:20}$, for example, takes as much as the cumulative time to price the queries $Q_1, \ldots, Q_{20}$ separately. And this cost is incurred repetitively, with each new query in the sequence.

8. DISCUSSION

We now discuss lessons learned and open research problems toward building a truly practical query pricing system.

Performance. As our experiments demonstrate, QueryMarket can price queries over small datasets interactively and over medium-size datasets within a minute or two. These query-pricing times are a first step toward a practical, automated query-pricing system, since they are much faster than the current approach of calling the data seller to negotiate a personalized data product. The ILP construction is the bottleneck step in query pricing, but QueryMarket can use *bucketization*, as described in Subsection 4.3, to reduce the number of price points and hence bring pricing back to interactive speeds, with the tradeoff of increasing the prices offered to the customers. Other optimizations may also be possible including pricing independent queries in parallel and combining ILP construction with ILP solving to reduce total processing times. Overall, however, scaling the pricing framework to datasets in the age of "Big Data" remains a challenging open problem.

We also learned that using history is critical in a dynamic setting. Bundle pricing offers the best functionality, but we discovered that it is too expensive. View pricing achieves a good trade-off between price reduction and efficiency.

Another interesting open problem is whether a system like QueryMarket could leverage approximations to give users an idea of a query's price without computing that price exactly such as to enable subsecond approximate query pricing.

Finally, in a practical system, it may be necessary to price queries based on both the value of the data and the cost of the resources necessary to process these queries.

Towards full SQL support. The current version of QueryMarket supports a large subset of SQL, but does not support aggregation or negation. Since queries with negation can be *non-monotone* and QueryMarket exploits the monotonicity property to compute a query price, using negation in queries would require new algorithms and theory for pricing non-monotone queries. As far as aggregation queries are concerned, the challenge is to fairly price such queries. For example, to determine a query Q that sums up the at-

tribute X for the relation $R(X)$, one must purchase all the views of the form $\sigma_{X=a}(R)$. Thus, Q's cost under our framework is the same as purchasing the relation R, even though the sum operator offers significantly less information than the whole relation. There has been some work on pricing aggregates [15], but finding a way to integrate aggregation in a way that correctly captures the information content remains a challenging open problem.

Approximate Answers. A natural operation for a data marketplace is to allow the user to ask queries with a *budget limit*; if the query is above budget the system may return a good approximate answer (for example, a sample of the answer, or some summarization of the query) that falls within the given budget. This area opens several interesting research questions, such as to find what are relevant approximations of the queries, and further which one is the most informative under a given budget.

Data Integration. QueryMarket currently ignores all issues related to data integration. It can price queries across data providers assuming that the query produces a correct result. Interesting future work includes modeling and accounting for data overlap between multiple vendors, modeling differences in data quality and freshness across these vendors, and perhaps offering credits, free samples, or discounts whenever a buyer attempts to integrate datasets from different vendors but fails to achieve a desired result.

9. RELATED WORK

The notion of pricing in databases has been studied in various contexts, including resource management [20] and physical tuning [12, 21]. The concept of pricing *data*, however, has only recently emerged as a research topic [11, 17]. In this latter context, a series of recent papers [13, 14, 15] developed theoretical frameworks for flexible data pricing. Our paper builds on this theoretical foundation, but is the first towards a practical design and implementation of a pricing system. More specifically, Koutris *et al.* [13] present and study a formal framework for query-based pricing. Central to this framework are the notions of *arbitrage-free* and *discount-free* pricing. The authors explore in depth the case where the price points are selections on single attributes, and present a dichotomy on the complexity of pricing conjunctive queries without self-joins into NP-complete and PTIME. Further work [15] studies the same pricing framework in the case of *linear* aggregate queries. In contrast, our work focuses on developing and evaluating a practical system to price a large class of queries that can be expressed as Unions of Conjunctive Queries with interpreted predicates, while addressing practical concerns related to dynamic environments.

Digital market services for data have recently emerged in the cloud [1, 7]. Similar to our work, these services also implement data pricing systems, but with simplistic pricing schemes. In the case of Infochimps [7], the seller can either specify and price selection queries or entire datasets. On the Azure Marketplace [1], sellers specify parameterized selection queries over the base data or over pre-defined views. The prices are then set based on the number of output records returned (*subscriptions* give users access to a certain number of result pages in a month). Our pricing system is more powerful, since it can automatically price a much greater variety of queries.

There exist many independent vendors selling data online [4, 3, 8, 5, 10]. They either use simple and limited pricing options to sell data en masse or they negotiate custom products with individual buyers. Factual [4] offers their data through APIs and charges based on the number of API calls per day. Similarly, Xignite [10] sells data through annual subscriptions that come with a monthly limit on the number of returned records. Gnip [3] negotiates prices with individual customers. Finally, Aggdata [5] uses fixed prices for entire datasets or provides full access to their data library for a flat subscription cost.

10. CONCLUSIONS

We presented QueryMarket, a system that automatically prices a large class of SQL queries requested by buyers based on price-points specified by sellers. QueryMarket supports updates to the database and accounts for query history. The system also supports multiple sellers by sharing revenues fairly among them. Computing arbitrage-free prices is theoretically hard, and we described a novel approach for computing prices, by translating then into optimized Integer Linear Programs (ILPs).

11. ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation and Microsoft through NSF CiC grant CCF-1047815 and NSF grant IIS-0915054.

12. REFERENCES

[1] datamarket.azure.com/.
[2] datamarket.azure.com/dataset/bing/microsofttranslator.
[3] gnip.com.
[4] http://www.factual.com/.
[5] www.aggdata.com/.
[6] www.gnu.org/software/glpk/.
[7] www.infochimps.com/marketplace.
[8] www.patientslikeme.com.
[9] www.wordfrequency.info.
[10] www.xignite.com/.
[11] M. Balazinska, B. Howe, and D. Suciu. Data markets in the cloud: An opportunity for the database community. *PVLDB*, 4(12), 2011.
[12] D. Dash, V. Kantere, and A. Ailamaki. An economic model for self-tuned cloud caching. In *ICDE*, pages 1687–1693, 2009.
[13] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-based data pricing. In *PODS*, pages 167–178. ACM, 2012.
[14] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Querymarket demonstration: Pricing for online data markets. *PVLDB*, 5(12):1962–1965, 2012.
[15] C. Li and G. Miklau. Pricing aggregate queries in a data marketplace. In *WebDB*, 2012.
[16] K. Morton, R. Kosara, J. Mackinlay, M. Balazinska, and D. Grossman. Masses of visualizations: An analysis of usage patterns on many eyes and tableau public. Technical report, 2013.
[17] A. Muschalle, F. Stahl, A. Löser, and G. Vossen. Pricing approaches for data markets. In *BIRTE Workshop*, 2012.
[18] A. Nash, L. Segoufin, and V. Vianu. Determinacy and rewriting of conjunctive queries using views: A progress report. In *ICDT*, pages 59–73, 2007.
[19] C. Shapiro and H. R. Varian. Versioning: The smart way to sell information. *Harvard Business Review*, 76:106–114, 1998.
[20] Stonebraker et al. Mariposa: a wide-area distributed database system. *VLDB Journal*, 5(1):048–063, 1996.
[21] P. Upadhyaya, M. Balazinska, and D. Suciu. How to price shared optimizations in the cloud. *PVLDB*, 5(6):562–573, 2012.