Price-Optimal Querying with Data APIs

Prasang Upadhyaya
University of Washington
prasang@cs.uw.edu

Magdalena Balazinska
University of Washington
magda@cs.uw.edu

Dan Suciu
University of Washington
suciu@cs.uw.edu

ABSTRACT

Data is increasingly being purchased online in data markets and REST APIs have emerged as a favored method to acquire such data. Typically, sellers charge buyers based on how much data they purchase. In many scenarios, buyers need to make repeated calls to the seller's API. The challenge is then for buyers to keep track of the data they purchase and avoid purchasing the same data twice. In this paper, we propose lightweight modifications to data APIs to achieve optimal history-aware pricing so that buyers are only charged once for data that they have purchased and that has not been updated. The key idea behind our approach is the notion of refunds: buyers buy data as needed but have the ability to ask for refunds of data that they had already purchased before. We show that our techniques can provide significant data cost savings while reducing overheads by two orders of magnitude as compared to the state-of-the-art competing approaches.

1. INTRODUCTION

Data in business and even certain sciences is increasingly being acquired from other companies [5,6,7,8,9,22,26] This has led to the emergence of data markets to facilitate the buying and selling of data. Schomm et al. [20] survey data sellers and list 46 commercial data suppliers as of 2013.

The most common method for selling data online is to make it available through a RESTful API [2, 5, 6, 9, 18, 22, 26, 27]. Existing APIs enable buyers to submit requests for data in the form of parameterized queries. For example, to purchase data from Twitter, one can specify keywords of interest, say a username, in the API call and Twitter returns all activity, up to an API defined limit, that matches the query. Typically, sellers charge buyers based on how much data they purchase. That is, the cost of an API call is the sum of the cost of the tuples returned by that call [26].

In many scenarios, buyers need to make repeated calls to the seller's API. One example is when purchased data drives an application and the use of that application determines the data that needs to be purchased. In those scenarios, buyers may inadvertently purchase the same data twice. In fact, it is hard to build applications that never purchase the same data again. We illustrate with a concrete example:

Proceedings of the VLDB Endowment, Vol. 9, No. 14
Copyright 2016 VLDB Endowment 2150-8097/16/10.

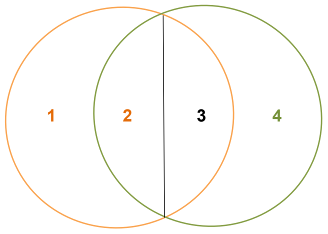


Figure 1: An illustration of the limitations of `REST` data APIs. Alice makes an API call for data in the circle covering regions 1, 2, and 3. Data in region 3 changes. Alice makes another API call for data in the circle covering regions 2, 3, an 4. With today's pricing methods, Alice will pay twice for data in region 2 and the unchanged data from region 3.

EXAMPLE 1.1. *Bob sells data on people who have visited a given business (examples of such services are Yelp [28] and Foursquare [7]). Bob provides an API* `checkins(lat, long, r, t)`*, where* `(lat, long)` *define the latitude and longitude of a circle's center with radius* `r`*.* `checkins` *returns the list of (possibly anonymized) users, along with their attributes, who have visited businesses, after a timestamp* `t`*, that lie in the circle.*

In our example, Alice first makes an API call, `checkins(`x_1`, `y_1`, r, `t_1`)`*, waits for some time, and makes another API call,* `checkins(`x_2`, `y_2`, r, `t_1`)`*, for a different center but the same radius. Figure 1 depicts the two calls. Between the two calls, some businesses in area 3 recieve new visits. Currently, if Alice executes the two queries, she will pay twice for the data in area 2 and for the old data from area 3. Since she does not know what updates were made to the data, she must make the API call to know if the data was updated.*

To only get the updates, Alice can change the time in the second call to $t_2 = t_1 + 1$. She may still end up paying for redundant data if there were any checkins with time $t > t_2$ in her first call. This happens when such customers visit a business in regions 2, 3, or 4 after time t_2, thus, being part of the answer returned to Alice during both API calls. In fact, in this example, it is impossible to avoid overpaying for data purchases with existing APIs.

Even for static datasets, to avoid paying twice for the data *in areas 2 and 3, multiple tiny circles that exactly cover area 4 and nothing outside it, are needed.*[1]

Today, sellers only keep track of the total amount of data purchased by a buyer but not the details of the purchased data. A primary drawback of only storing limited data about user purchases is that it puts the burden on the buyer to never purchase the same data twice or risk paying multiple times for the same data.

Buyers may cache the result of API calls and attempt query rewriting to only ask for new data. Caching will, however, be unusable in the case of time-varying data and caching restrictions. For datasets such as weather and traffic, the underlying data changes over time. In such cases, it may not be possible to predict when changes are made to the subset of data that a buyer is interested in and the only way to know of an update is to redo the call to the data API. Some APIs such as Yelp [28] prohibit caching beyond a single user session, while others, such as Twitter [24], prohibit caching of location and geographic information, except when joined with a tweet, while permitting caching of other parts of the data. Thus, even if the buyer knew that they would require a newly purchased data item in the future, they are prohibited from caching it and reusing it when the need arises. Thus, in both circumstances, the buyer can not avoid making multiple API calls and must incur the cost of repeat purchases of the same data.

Sellers could store users' purchase histories to enable pricing that accounts for prior API calls. It may be beneficial for sellers to provide a service that only charges for data once so as to enable *price discrimination*. Although there are customers who may pay the full price of the data and not worry about paying extra, there are price-conscious customers who may not buy the dataset unless the data is available within their budget. Providing an avenue for such customers to optimize and reduce their data costs can increase revenues.

However, as we evaluate in Section 6, the computational overhead of storing the purchase history at the seller is significant. Space and time overheads can be in the order of the data size and the number of previous API calls. Moreover, this might dissuade those customers who prefer that their querying history not be stored permanently at the seller.

To address the above challenges, in this paper, we propose lightweight modifications to data APIs to achieve the following three goals:

- *Optimal history-aware pricing*: We provide a method to price API calls so that buyers are only charged once for data that they have purchased and that has not been updated. We refer to this as *history-aware* pricing.
- *Constant overheads*: We provide a method to support history-aware pricing that only requires the seller to store a constant amount of state per buyer. Currently, sellers already store such information so as to keep track of a user's aggregate use of their services.
- *Anonymity*: In addition to the above cost and performance properties, we also provide anonymity to the buyers about what data they purchase and when the purchases are made. That is, the seller need not retain any identifying information about the user that can recreate a user's query history. A service to sell data without remembering what the users purchased is a desirable feature for both users and sellers: (a) users are assured of anonymity of their query history; the query history might permit disclosure of competitive intelligence, and, (b) the sellers need not invest additional resources in securing user purchase histories since they are not stored by the seller.

[1]Only a finite number of such API calls are needed since we assume the domains for `lat, long, radius` to be finite.

The key idea behind our approach is the notion of refunds: buyers buy data as needed but have the ability to ask for refunds of data that they had already purchased before. Thus, the payment for data is conducted in two steps: the usual payment when data is received and another round where the buyer asks for refunds. While asking for refunds, the buyer *proves* to the seller that she has been charged multiple times for the same data. The proofs are constructed so as to protect against tampering by the buyer even when the buyer is not truthful or can collude with other buyers.

In this paper, we make the following contributions:

1. In Section 4, we propose the notion of refunds as a way to provide optimal, history-aware pricing for data APIs that can be expressed as SPJU queries, without negation nor duplicate elimination. We describe the construction of refunds for a single-buyer setting with no updates and prove properties about the correctness and optimality of such a system.
2. In Sec. 5, we propose a generic and extensible framework to support refunds. We then show how the framework can be used to accommodate multiple buyers, updates, and optimizations to reduce the computational and communication overheads of using refunds.
3. In Section 6, we evaluate empirically and compare the refund-based approaches to approaches that store user history at the server as well as approaches that do not provide optimal pricing. We show that even for small workloads, cost savings from $10\times$ to $99\times$ can be obtained through the use of refunds, compared to history-agnostic pricing. The associated performance overheads, compared to history-agnostic pricing, are no larger than $2\times$ in the best case (when no refunds need to be asked) and $6\times$ in the worst case (when the entire query is repeated). Refund approaches are comparable in performance to history-based approaches while protecting privacy. Further, the optimizations we develop in Section 5 cut overheads by a factor of $22.2\times$ as compared to the basic refund protocol.

We first define our problem setting (Sec. 2) and describe two algorithms (Sec. 3), that are not based on refunds, to manage the pricing of API calls. In Sec. 4, we describe the desiderata of a basic refund protocol and describe our construction of a refund scheme. In Sec. 5, we generalize the basic refund protocol and optimize the protocols to reduce the overhead of using refunds. In Sec. 6, we compare these algorithms against the refund-based pricing framework.

2. PROBLEM DESCRIPTION

We first define the pricing functions and our setting. In current data markets, it is the buyer who must agree to terms and restrictions set by the seller. Thus, our model assumes that the seller is trusted, while the buyer is not. In our setting, Alice runs an application that acquires data from Bob, the data seller. Bob charges Alice separately for each output tuple in her answer set.

We assume a database D storing relations $D_1, \ldots, D_k$ with schemas of the form: `(tid, ver,` A_1`,...,` A_m`)`. Here the column `tid` is a primary key and `ver` indicates the version number of the tuple. The version numbers are initialized to 0 and are incremented every time an update is made to the tuple; the system only keeps the last version of each tuple. Thus, the version numbers are just an extra attribute that we add to each tuple. The pricing function generates a price per output tuple; that is, for a query Q over D, there is a pricing function p that assigns a price to each output tuple $t \in Q$. This is a common way [26] to price relational data in commercial data markets. In the rest of the paper, we assume that all tuples have the same price (taken to be a unit of some currency), but the techniques generalize to cases with non-uniform prices.

Pricing Full Selection Queries. Given a function that assigns a price to each base tuple, pricing selection queries, `SELECT * FROM Di WHERE [condition]`, is trivial since each query returns a subset of base tuples. The price of each output tuple is the price of the corresponding base tuple. Further, since each base tuple contributes to at most one output tuple, the price of the query is the sum of the prices of the tuples in the query's output.

Pricing Full Join Queries without Self-Joins. For queries with joins, but without projections or self-joins, such as,

```
SELECT * FROM D1,..., Dk WHERE [conditions]
```

where $\forall i, j, i \neq j : D_i \neq D_j$, this paper prices each output tuple, t, as the sum of the prices of the base tuples from relation D_i that belong to the where provenance of t. Since a single tuple may contribute to many output tuples, to prevent charging multiple times for the tuple, Bob must price the overall query as the sum of the prices of the base tuples returned by k queries of the form:

```
SELECT DISTINCT Di.* FROM D1,..., Dk WHERE [conditions]
```

That is, Bob must price the tuples from the individual relations separately, by computing semi-joins for each relation.

Pricing Projection Queries without Duplicate Elimination The price of a query with projections, but without duplicate elimination, is identical to the price of the corresponding full query.

Pricing More Complex Queries. Consider any technique (like the ones presented in this paper) that can price a sequence of SPJ queries without unions, self-joins, or duplicate elimination, such that Alice only pays once for each tuple, irrespective of how many queries in the sequence the tuple belongs to. Any such technique can be augmented to also support sequence of queries with unions and self joins, as we discuss below.

Pricing a Query Sequence with Unions. For queries with unions, say $Q_1 \cup Q_2$, Bob prices Q_1 and Q_2 separately as if they were two independent queries in the sequence.

Pricing a Query Sequence with Self-Joins. Similarly, for queries with self-joins, such as,

```
SELECT * FROM D1, D2,..., Dk WHERE [conditions]
```

Algorithm 1: `HistoryStream`

```
Input: userId INT, tupid INT
begin
    // costs(uid, cost) is table with cost of purchased tuples
    // for user uid.  historyStore(uid, history) stores a bit
    // vector, for user uid, to remember their bought tuples.
    DECLARE myhistory bytea;
    SELECT INTO myhistory history FROM historyStore
        WHERE uid = userId;
    if get_bit(myhistory, tupid) = 0 then
        UPDATE costs SET cost = cost + 1 WHERE uid = userId;
        myhistory := set_bit(myhistory, tupid, 1);
        UPDATE historyStore
            SET history = myhistory WHERE uid = userId;
```

where $D_i = D_j$ for some pair $i, j \in [1, k]$, Bob prices the tuples from the individual relations separately. That is, Bob prices the following *sequence* of queries from $i \in \{1, \ldots, k\}$.

```
SELECT DISTINCT Di.* FROM D1,..., Dk WHERE [conditions]
```

For any framework that provides optimal history-aware pricing, our desiderata are: (1) minimize state at the seller, (2) minimize processing at the seller, (3) keep data transfer overheads low, and (4) minimize query latency overheads.

Given that pricing join and union queries reduces to pricing selection queries, in the rest of the paper, we assume a single relation D and focus on selection query pricing to simplify the presentation.

3. NAÏVE APPROACHES

We now look at two classes of solutions to manage pricing: the first does not provide optimal pricing, in the sense that Alice would pay multiple times for the same tuple she purchases; the other does provide optimal history-aware pricing, but does not satisfy the first two requirements of the desiderata.

The naïve way to compute the prices is through two queries: '`result = Q(D)`' followed by '`SELECT COUNT(*) FROM result`'. Bob charges the amount calculated by the second query to Alice and returns a cursor to `result`. Both queries belong to a single transaction to prevent the data from being updated between the time when the price is computed and the cursor to `Q` is returned. We call this method `CountBlock`, where 'Block' indicates that the query's cost is computed before the cursor to the query's answer is returned to Alice. Another approach, called `CountStream`, counts the cardinality as Alice advances the cursor. Neither `CountBlock` nor `CountStream` store query history at the seller, and hence they charge Alice for each tuple, even if the tuple was purchased by Alice in a previous query.

Another approach is for Bob to track the tuples purchased by Alice in a bit vector with one bit per tuple in table D. Bob keeps one such bit vector for each user. Whenever Alice buys a tuple, the associated bit is set; while, whenever Bob updates a tuple, the corresponding bit is cleared. Algorithm 1, `HistoryStream`, is a function that updates the bit vector in a streaming fashion, as the cursor is advanced; while Algorithm 2, `HistoryBlock`, is a blocking implementation where updates to the bit vector are performed inside a user defined aggregate that aggregates over the set of tuple ids in a query's answer. The pseudo-codes are based on the `PL/PGSQL` syntax. As before, the above steps are encapsulated in a single transaction to prevent updates to

Algorithm 2: `HistoryBlock` user defined aggregate

```
// history has a composite type with (vec bytea, price int).
// vec is the history bit vector.
Input: INOUT history, IN userId, IN tupid
begin
    if history IS NULL then
        history := ROW(0, 0); history.price := 0;
        SELECT ph.history INTO history.vec
            FROM historyStore AS ph WHERE ph.uid = userId;
    else if get_bit(history.vec, tupid) = 0 then
        history.vec := set_bit(history.vec, tupid, 1);
        history.price := history.price + 1;
```

the data between the time when the price is computed and when the cursor to `Q` is returned. One drawback of using the history-based approach is that the seller must provide durable storage for user history. Another drawback is that buyer purchases can no longer be anonymous.

4. REFUNDS

In this section, we propose refunds as a new mechanism for optimal, history-aware pricing of a sequence of queries. With support for refunds, Alice can make multiple API calls without modifying her queries. If she makes repeated purchases, they are identified and the extra amount she paid for the repeated purchases are refunded by Bob to Alice.

To support refunds, Bob computes additional information, called refund coupons, which he returns along with the results of Alice's queries. Bob continues to charge Alice as he normally would (using either `CountStream` or `CountBlock`, whichever leads to higher throughput, depending on the query), without accounting for any previous queries from Alice. The coupons are designed so that if there is a common tuple with identical `tid`, say with value `id`, in the result of two different queries, there is a coupon from the first query and a corresponding coupon from the second query such that Bob can inspect the two coupons to determine that they refer to the same tuple with `tid = id`. Given this, Bob knows that Alice was charged twice for `id` and he can refund the price of the tuple.

Alice is responsible for storing the coupons, detecting repeat purchases, and using the coupons to ask for refunds. Keeping track of coupons is easier than tracking the tuples purchased since (a) coupons naturally work with updates and (b) Alice need not modify her queries.

We now formally define the protocol to support refunds for a single seller and a single buyer over a static database. We then generalize the protocol to multiple buyers and to support updates. In Section 5, we consider specialized optimizations to reduce the overhead of supporting refunds.

We define the protocol by the messages Alice and Bob send to each other. The protocol begins when Alice sends a query `Q` to Bob. Bob sends back two messages: `Q(D)` and `refunds(Q, D)`. Both messages are sets of tuples with the following properties:

1. The schema for `refunds(Q, D)` is `(tid, qid, digest)`, where `tid` is a tuple identifier, `qid` is a query identifier, and `digest` is the output of a hash function. The schema for `refunds` is independent of the schemas for `Q` and `D`. We call each tuple in `refunds(Q, D)` a coupon where coupon `c` is defined as
$$\mathtt{c} = (\mathtt{id}, \tau, \mathcal{H}(\mathtt{id} \oplus \tau \oplus \kappa)) \qquad (1)$$
Here `id` is the tuple identifier; τ is a unique identifier assigned by the server to each query such that τ is monotonically increasing; $\mathcal{H}$ is a cryptographic hash function, `SHA1` in our implementation; $\oplus$ is the XOR operation;[2] and, κ is a secret key only known to Bob. In the single-buyer protocol over static data, τ is an integer that is initialized to 0 and is incremented for each query Alice sends to Bob.
2. There is a one-to-one correspondence between tuples in `refunds(Q, D)` and tuples in `Q(D)`. That is,
$$\forall \mathtt{t} \in \mathtt{Q(D)}, \exists \rho \in \mathtt{refunds(Q,D)} : \mathtt{t}[\mathtt{tid}] = \rho[\mathtt{tid}], \text{ and}$$
$$\forall \rho \in \mathtt{refunds(Q,D)}, \exists \mathtt{t} \in \mathtt{Q(D)} : \rho[\mathtt{tid}] = \mathtt{t}[\mathtt{tid}]$$

In case Alice gets the same tuple, with $\mathtt{tid} = \mathtt{id}$ twice, from queries $\mathtt{Q}_1$ and $\mathtt{Q}_2$, she will also get two coupons $\mathtt{c}_1$ and $\mathtt{c}_2$ such that $\mathtt{c}_1[\mathtt{tid}] = \mathtt{c}_2[\mathtt{tid}] = \mathtt{id}$. Note that we have assumed that all tuples are identically priced.[3] If Alice detects repeat purchases, she can ask Bob for a refund by sending a message consisting of a pair of coupons for the same tuples. Bob verifies that the hash values of the returned coupons are the ones he previously computed and credits the refund to Alice. We call this protocol `BasicRefunds`. Formally, `BasicRefunds` is defined as follows:

1. Alice sends a refund message $\rho = \langle c_1 = (id_1, \tau_1, h_1), c_2 = (id_2, \tau_2, h_2)\rangle$.
2. Bob verifies the following: (a) $id_1 = id_2$, (b) $\tau_1 < \tau_2$, and (c) $\forall i \in \{1, 2\} : h_i = \mathcal{H}(id_i \oplus \tau_i \oplus \kappa)$.

Intuitively, the refund message ρ asks a refund for tuple $id = id_1 = id_2$ purchased for a query with $qid = \tau_2$ using the coupon for the same tuple purchased with a previous query with $qid = \tau_1$.

We now define the criteria for *safety* and *optimality* of any refund-based pricing protocol. Let $W = (M_1, \ldots, M_{n_q+n_r})$ be a sequence of messages from Alice to Bob consisting of n_q queries and n_r refund requests, where each M_i is either a query Q or a refund request ρ. If over the n_q (possibly different) queries, tuple t_i was purchased n_i times, then let
$$T(W) = \{(t_1, n_1), \ldots, (t_m, n_m)\}$$
be the set of all tuples purchased by Alice along with their counts. Given that $p : tid \to \mathbb{R}$ is the function that assigns prices to tuples, we denote by $P(W)$ the amount Alice pays for the queries in W:
$$P(W) = \sum_{Q \in W} \sum_{t \in Q(D)} p(t[tid]) = \sum_{(t,n) \in T(W)} n \cdot p(t[tid]) \quad (2)$$

Similarly, $R(W)$ denotes the amount Bob refunds to Alice after processing W:
$$R(W) = \sum_{\rho \in W} p(\rho[tid]) \qquad (3)$$

W may contain multiple refund requests for the same tuple. For example, say ρ_1 and ρ_2 are refund requests for

[2]For simplicity, all ids are assumed to be integers. If not, they can be cast into a string type along with the use of string concatenation in place of XOR.

[3]For non-uniform prices, use the coupon $\mathtt{c} = (\mathtt{id}, \mathtt{p}, \mathcal{H}(\mathtt{id} \oplus \mathtt{p} \oplus \kappa))$, where `p` is the tuple's price.

the same tuple, $tid = 1$. In such case, $R(W)$ would be $p(\rho_1(tid = 1)) + p(\rho_2(tid = 1)) = 2p(t[tid = 1])$.

Let $\Delta(W) = P(W) - R(W)$ be the net payment by Alice with message sequence W.

Safety. A refund protocol is safe if Alice must pay at least once for each tuple she has purchased. Formally,

$$\forall W : \Delta(W) \geq \sum_{(t,n)\in T(W)} p(t[tid]) \qquad (4)$$

Optimality. A refund protocol is optimal if there is a way to ask for refunds so that Alice never pays more than once for each tuple she has purchased. Formally, if $\bar{Q} = (Q_1, \ldots, Q_{n_q})$ are the queries in W, $\bar{\rho} = (\rho'_1, \ldots, \rho'_{n'_r})$ are refunds, and $W' = \bar{Q} \cdot \bar{\rho}$ is their concatenation, that is, $W' = (Q_1, \ldots, Q_{n_q}, \rho'_1, \ldots, \rho'_{n'_r})$ where W' is the set of all messages that Alice sends to Bob, then,

$$\forall W\, \exists W' : \Delta(W') = \sum_{(t,n)\in T(W)} p(t[tid]) \qquad (5)$$

That is, given the queries in a message sequence W, it is always possible to request refunds to obtain the maximum possible safe refund.

Before analyzing `BasicRefunds`'s safety and optimality, we note that Alice only controls three aspects of the refund protocol: *when* she asks for refunds, the *number* of refund messages, and the *coupons* she uses for her refund messages. She can not forge coupons of her own since $\mathcal{H}$ is a cryptographic hash and only Bob knows the secret key κ.

LEMMA 4.1. *`BasicRefunds` is optimal.*

PROOF. We use induction on the number of queries in W.

Base case. With no queries, $W = \emptyset$, $P(W) = R(W) = \Delta(W) = 0$. Thus, `BasicRefunds` is optimal.

Inductive case. For a sequence of $i - 1$ queries $(Q_1, \ldots, Q_{i-1})$, let W_{i-1} be the optimal sequence of queries and refunds. For a new query Q_i, let the refund messages be $(\rho_{i1}, \ldots, \rho_{ik})$ where each ρ_{ij} is a refund for a tuple t that has been purchased before. Refund ρ_{ij} is constructed by taking the coupon for t, received with query Q_i, and any coupon for the same tuple id $tid = t[tid]$ received with a previous purchase. Then the sequence is $W_i = W_{i-1} \cdot Q_i \cdot \rho_{i1} \cdots \rho_{ik}$ is optimal. Let $T_{new} = \{t \in Q_i(D) \wedge (t, n) \notin T(W_{i-1})\}$ and $T_{old} = \{t \in Q_i(D) \wedge (t, n) \in T(W_{i-1})\}$. Given the notation $p(t) = p(t[tid])$, we get,

$$\begin{aligned}
\Delta(W_i) &= P(W_{i-1}) + P(Q_i) - R(W_{i-1}) - \sum_{j=1}^{k} R(\rho_{ij}) \\
&= P(W_{i-1}) - R(W_{i-1}) + \sum_{t\in T_{new}} p(t) + \sum_{t\in T_{old}} p(t) - \sum_{j=1}^{k} R(\rho_{ij}) \\
&= \Delta(W_{i-1}) + \sum_{t\in T_{new}} p(t) + \sum_{t\in T_{old}} p(t) - \sum_{t\in T_{old}} p(t) \\
&= \Delta(W_{i-1}) + \sum_{t\in T_{new}} p(t) \\
&= \sum_{(t,n)\in T(W_{i-1})} p(t) + \sum_{t\in T_{new}} p(t) \\
&= \sum_{(t,n)\in T(W_i)} p(t)
\end{aligned}$$

Hence, W_i is optimal. □

`BasicRefunds` is not safe, though. Given any non-empty sequence of messages W, W can repeat a non-empty query q, and repeatedly ask for refunds of a single tuple. That is, if $\langle c_1, c_2 \rangle$ is a legitimate refund request, Alice keeps sending the request multiple times and can thus get more as refunds than the cost of the data itself.

To handle this case, we modify `BasicRefunds` to `MonotoneRefunds`. `MonotoneRefunds` is both safe and optimal. To implement the protocol, Bob maintains an expected query id τ_{exp} for refunds by Alice. τ_{exp} is initialized to 0 when Alice registers with Bob. The protocol is as follows:

1. Alice sends a $\langle$ `BEGIN REFUND` τ $\rangle$ message. Here τ is a query id.
2. Alice sends one or more refund messages. Each refund message $\rho = \langle c_1 = (id, \tau_1, h_1), c_2 = (id, \tau, h_2) \rangle$ uses the same query id τ for the second coupon as the τ specified in the $\langle$ `BEGIN REFUND` τ $\rangle$ message.
3. Alice sends a $\langle$ `END REFUND` τ $\rangle$ message.
4. Apart from checking that the digest of the message is equal to the computed hash value as in `BasicRefunds`, Bob also checks that (a) there is only one refund message for each tuple with $tid = id$, (b) the query id of all second coupons, τ are identical and equal to the τ in the $\langle$ `BEGIN REFUND` τ $\rangle$ message, and (c) $\tau \geq \tau_{exp}$.
5. If any of the conditions are not met, all the coupons in the `BEING ...END` block are rejected. Else, Bob credits the total refund to Alice and updates τ_{exp} to $\tau + 1$.

An alternative view of `MonotoneRefunds` refund protocol is that steps 1 through 3 define a "session" where one session is an atomic way for Alice to transmit a set of coupons associated with a single query. Alice checks the coupons after each new query and sends refund coupons once for each query, while Bob only keeps track of the most recent query for which refund coupons have already been processed.

To check the uniqueness of refund messages in Step 4, Bob can use a hash table. To directly check the uniqueness within a DBMS, Bob can also store the refunds in a temporary table, `tempRefunds`, and run: `SELECT 1 FROM tempRefunds GROUP BY tid HAVING COUNT(*) > 1`. A non-empty answer indicates a repeated refund.

LEMMA 4.2. *`MonotoneRefunds` is optimal.*

PROOF. If τ_{latest} is the latest query id whose coupons have not been used for refunds, then $\tau_{exp} \leq \tau_{latest}$. This is because all refunds issued in W must have a query id $\tau \leq \tau_{latest} - 1$ and hence, $\tau_{exp} \leq \tau_{latest}$ by definition. Given this, the construction of the refunds in the proof for Lemma 4.1 is also valid for `MonotoneRefunds` and hence, it is optimal. □

We prove a stronger safety property about individual tuples that implies our original safety definition for queries.

LEMMA 4.3. *For each tuple t, let $k \geq 1$ be the number of queries by Alice that contain t, and let r be the number of valid refund messages that request a refund for t, then, `MonotoneRefunds` ensures that $k - r \geq 1$ at all times.*

PROOF. Let the tuple be t. We prove the safety by induction on the length of W. The base case is trivially true when the first query that includes t is executed. Note that with a single query including t, valid refund messages can not be constructed, since the two coupons in the refunds must have different query ids τ. Thus $k = 1$ and $r = 0$ and the base case is satisfied.

For the inductive case, given a message sequence W_{n-1} of length $n-1$ with $k_{n-1} = k \geq 1$ queries containing t and $r_{n-1} = r-1$ refunds, such that $k_{n-1} - r_{n-1} \geq 1$, we consider M_n, the n^{th} message. There are four cases:

1. M_n is a query Q. If it returns the tuple t, then, $k_n = k_{n-1} + 1$, else $k_n = k_{n-1}$. Since there are no refunds, $r_n = r_{n-1}$. Thus, $k_n - r_n \geq 1$.
2. M_n is a `BEGIN REFUND` message. In this case, $k_n = k_{n-1}$ and $r_n = r_{n-1}$ and thus, $k_n - r_n \geq 1$.
3. M_n is a valid refund message $\rho = \langle c_1 = (t', \tau, h_1), c_2 = (t', \tau, h_2)\rangle$. If M_n is not a valid message for t, that is $t' \neq t$, neither k nor r change. Otherwise, by the induction hypothesis: $k - (r-1) \geq 1$. Thus, $k \geq r$.
(1) If $k \geq r+1$, then the ρ makes $r_n = r_{n-1} + 1 = r$ and $k_n - r_n = k - r \geq 1$ by assumption.
(2) If $k = r$, then consider the $r-1$ previous refunds. They must use coupons from r *distinct* query ids. This is because the first refund uses two distinct query ids (by the construction of coupons) and all $r-1$ refunds use their second coupons from $r-1$ different queries, since only one refund coupon for a tuple is allowed in a `BEGIN REFUND ...END REFUND` block and τ_{exp} is incremented after each valid `END REFUND`. Thus, the $r-1$ previous refunds have used coupons from r queries. Since $k = r$, the expected query id in the refund M_n must be at least one more than the query id of the k^{th} query that contains t. But since no unused coupon for t exists, the refund M_n is not a valid coupon. This is a contradiction.
Thus, the $k_n - r_n = k - r \geq 1$ holds.
4. M_n is a `END REFUND` message. In this case, $k_n = k_{n-1}$ and $r_n = r_{n-1}$ and thus, $k_n - r_n \geq 1$.

Thus, `MonotoneRefunds` is safe for tuple t. $\square$

Lemma 4.3 implies the safety definition in Eq.(4) as shown in our technical report [25, §10.1].

Thus, `MonotoneRefunds` is both optimal and safe.

5. EXTENSIONS AND OPTIMIZATIONS

We now consider extensions and performance optimizations that generalize the protocols to more realistic settings.

5.1 Extensions

In the protocols described in the previous section, the safety and optimality proofs continue to hold as long as the tuple ids are such that different tuples have different ids and identical tuples have the same id, irrespective of the query to which the tuple belongs. This observation allows us to easily extend the protocols to support more than one user and handle updates.

Multiple Buyers. If there is more than one buyer, we change the tuple identifiers to also incorporate the user id. That is, the new tuple id is (id, uid) where id is the tuple's id (as in the single-buyer protocols) and uid is a unique id assigned to each user. The coupons thus look as follows:

$$\mathtt{c} = ((\mathtt{id}, \mathtt{uid}), \tau, \mathcal{H}(\mathtt{id} \oplus \mathtt{uid} \oplus \tau \oplus \kappa))$$

With the updated construction for the coupons, different users will be assigned different tuple ids for the same tuple, while identical tuples for a user will continue to be assigned identical tuple ids. Thus, a buyer can not use refund coupons from another buyer, but can continue to use her own coupons as in the single-buyer setting.

Updates. We can also support updates by modifying the tuple ids. This is applicable when updates to a tuple are priced as if the update is a new tuple. Thus, if Alice purchases tuple t_1 in her first query, then purchases t_1 again in her second query, followed by an update to t_1, denoted now by t_2, followed by another purchase of t_2, then, she should be charged for t_1 in her first query, then refunded in the second, and eventually charged only once more for t_2.

To support updates, Bob maintains a version number, v, for each tuple that is incremented after each update. This version number is now included in the tuple id used for constructing the refund coupons:

$$\mathtt{c} = ((\mathtt{id}, \mathtt{uid}, \mathtt{v}), \tau, \mathcal{H}(\mathtt{id} \oplus \mathtt{uid} \oplus \mathtt{v} \oplus \tau \oplus \kappa))$$

Thus, only identical versions of a tuple have the same tuple id. Version numbers impose a storage overhead but they are useful for other purposes and are maintained by many systems by default. For example, the SDSS [21] adds version numbers to their data releases and SciDB [1] provides a no-overwrite storage system with versioning. So, in many applications, versions already exist.

5.2 Group Coupons

`MonotoneRefunds`, described in Section 4, only computes one coupon per tuple. This leads to a large number of refund messages, each of which is an API call to Bob, when asking for refunds. As Table 2 shows during experimental evaluation, the overhead of processing refunds can be an order of magnitude larger than the query time.

To reduce this overhead, we generalize coupons to allow Bob to create group coupons that can be used to refund a group of one or more purchased tuples with a *single* coupon. With group refunds, Bob sends back the group coupons for tuple groups of his choosing.

EXAMPLE 5.1. *Bob has a dataset with schema* (`key`, `value`)*, where* `key` *is an integer id. Bob provides an API with two parameters, keys* k_1 *and* k_2*, and returns all values in* $[k_1, k_2]$*. Suppose, Alice queries* $[1, 3]$*. With group coupons, Bob can compute coupons for keys* $\{1\}$*,* $\{2\}$*,* $\{3\}$*,* $\{1, 2\}$*,* $\{2, 3\}$*,* $\{1, 3\}$*, and* $\{1, 2, 3\}$*. If the next query by Alice is again for* $[1, 3]$*, Alice can either use the* $\{1, 2, 3\}$ *coupon for a group refund or only singleton coupons* $\{\{1\}, \{2\}, \{3\}\}$*, or a mixture* $\{\{1\}, \{2, 3\}\}$ *or* $\{\{2\}, \{1, 3\}\}$ *or* $\{\{3\}, \{1, 2\}\}$*.*

We emphasize that Bob may compute coupons where the same tuple may belong to multiple group coupons. But, in the refund protocol, Alice can only request refunds using one group coupon for each tuple for safety. In the example above, if Alice could request refunds for $\{1, 2\}$ and $\{2, 3\}$ simultaneously, she could refund 2 twice. Thus, more than one group coupon that includes a common tuple can not be used simultaneously in the same refund round. Further, a refund coupon will reimburse all the tuples covered by the coupon and can not be applied selectively to some tuples.

To construct group coupons, the key idea is to make a unique group id (instead of a tuple id) such that no two groups with different tuples (and with possibly different versions) have the same group id and all groups with identical tuples have the same group id. Bob must provide a way to compute such group ids and also provide a function, *contains* : $id, gid \rightarrow \{\texttt{true}, \texttt{false}\}$, that returns `true` if a tuple with tuple id *id* belongs to the group with id *gid*. Bob sends the `contains` function to Alice, who uses it to ensure that no two group refund requests share any tuple. In the worst case, using `contains` may require that she iterate over both the tuple and coupon sets. However, as Example 5.2 shows below, much more efficient ways exist, when dealing with common parameterized queries, to verify that a collection of group coupons have no common tuples. We show an even more efficient approach in the specific, but common, case of hierarchical coupons in Sections 5.3 and 5.4.

The group coupon is constructed as follows:

$$\texttt{c} = ((\texttt{gid}, \texttt{uid}, \texttt{gv}), \tau, \mathcal{H}(\texttt{gid} \oplus \texttt{uid} \oplus \texttt{gv} \oplus \tau \oplus \kappa))$$

Here *gv* is the group version number and is equal to the sum of the version numbers of the tuples that belong to the group. Another interpretation of *gv* is that it is the total number of updates made to tuples in the group.

For group refunds, the amount Bob refunds to Alice is the total cost of the tuples in the group. Let $I(t, \rho)$ be an indicator variable with value 1 if $contains(t[tid], \rho[gid]) =$ `true` (ρ is a valid group refund for a group containing tuple t), and 0, otherwise. Then, for a workload W, the total refunds, $R(W)$, is:

$$R(W) = \sum_{\rho \in W} \sum_{(t,n) \in T(W)} I(t, \rho) * p(t[tid]) \qquad (6)$$

To use group refunds, we modify `MonotoneRefunds` to `GroupRefunds` by changing the test to validate a refund message $\rho = \langle c_1, c_2 \rangle$ in Step 4 as:

> Apart from checking that the digest of the message is equal to the computed hash value as in BasicRefunds, Bob also checks that (a) *at most one* group coupon contains a tuple with tuple id *id* (Example 5.2 provides an illustration), (b) as with `MonotoneRefunds`, the query id of the second coupon in each refund message pair, that is $c_2[qid]$, is equal to τ in the $\langle$ `BEGIN REFUND` τ $\rangle$ message, and (c) $\tau \geq \tau_{exp}$.

Example 5.2. *Continuing the example, Bob must construct group coupons to check condition (a) in* `GroupRefunds`. *How to accomplish this depends on the kind of group coupon being constructed. For example, for parameterized range queries that select contiguous ranges for each column such as* `low <= col and col <= high` *where* `low` *and* `high` *are values of column* `col` *and* `low <= high`, *an efficient construction is to append the end points of the range,* `low` *and* `high` *for the various columns, in the group id. Given this construction, for any two group coupons, it can be checked without investigating the individual tuples in the group if the ranges for* `col` *overlap. For parameterized queries containing only equality predicates* `col = val` *where* `col` *is a column and* `val` *is a constant, appending* `(col, val)` *to the group id allows constant time checks for whether one coupon is contained in another.*

Lemma 5.1. `GroupRefunds` *is both optimal and safe.*

We defer the proof to the technical report [25, §10.2].

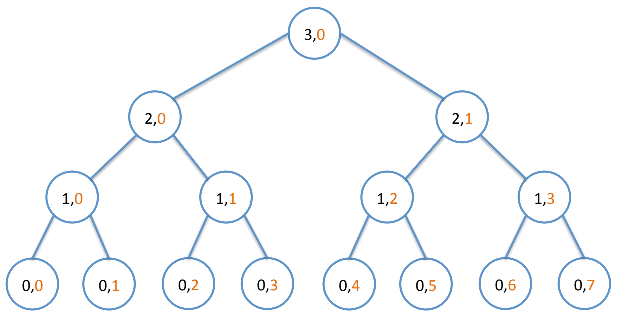


Figure 2: Group identifier assignment for tree-structured group coupons. The tree is built for a database with 8 tuples. It is a balanced binary tree where each node, including the leaves, is assigned an identifier `(height, id)`, where `height` is the height of the node (leaves are at height 0), and `id` is the node's order amongst the nodes at its height, where the leftmost node is assigned the id 0, the next node to the right the id 1, and so on.

5.3 Tree-Structured Group Coupons

There are various ways to group tuples when generating group coupons since any arbitrary subset of tuples in the answer can be a valid candidate. In the remainder of this paper, we discuss hierarchically-structured group coupons, where for any two group coupons, either they represent disjoint groups or they represent groups where one group is a subset of another. We call these tree-structured coupons. We briefly explore hierarchically-structured group coupons where the disjointedness condition does not hold in Section 5.4. We show a tree structured group coupon construction scheme for general conjunctive queries. This structure can be used for point queries and range queries.

In our construction of tree-structured coupons, we require that tuple ids be integers. Figure 2 illustrates how the group coupon identifiers are constructed and how the groups are formed. We focus on binary trees since they minimize the expected number of coupons per refund and are easy to implement, but our technique and analysis generalize for group coupons based on n-ary trees.

For binary trees, we start by treating *all* the tuples of the relation `D`, order by `id`, as leaves. The group identifier of the leaves is $(0, id)$. The next level of the tree is constructed by successively grouping nodes with `ids` $2n$ and $2n + 1$ to give a group identifier $(1, n)$. Here, 1 represents the height of the node. The higher levels of the tree are constructed recursively, by combining the nodes at the lower levels. We stop combining nodes when we only have one node, which forms the root of the tree. Note that we pad the database so that its cardinality is always a power of 2. With this construction, a group node with group id (h, n) is a group that includes all rows with ids in $\{2^h n, \ldots, 2^h(n + 1) - 1\}$.

Formally, the hash digest, `treeHash(uid, height, id, version, qid)`, for tree coupons is computed as follows:

$$\mathcal{H}(\texttt{uid} \oplus \texttt{height} \oplus \texttt{id} \oplus \texttt{version} \oplus \texttt{qid} \oplus \kappa) \qquad (7)$$

While asking for a refund, and to reduce the number of refund requests Alice makes, she asks for the largest valid group refund, that is the group refund with the maximum height such that all the tuples in that group are eligible for a refund.

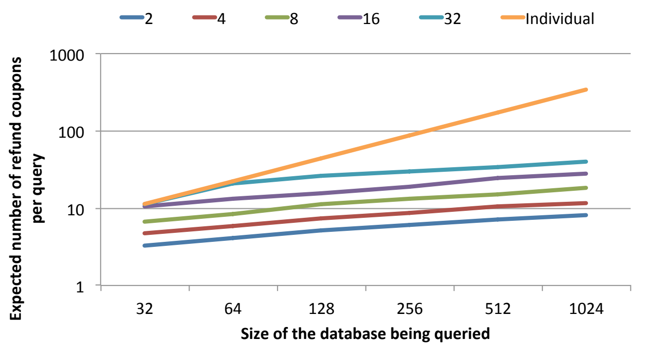


Figure 3: Average number of refund coupons, required by Alice, during the refund phase for range queries where each range is equally likely. We vary the size of the database on the x-axis. "Individual" refers to coupons required without group coupons while other lines refer to fan-outs of 2, 4, 8, 16, and 32.

Analysis. For a binary tree, if we assume that each range is equally probable, the expected number of coupons per refund is approximately $\frac{\log n}{2} + 4$. Further, the additional coupons generated per request with result cardinality $|T|$ is $|T|$. For comparison, using only individual coupons requires $\frac{n}{3} + 1$ coupons for each refund, on average, but with no additional group coupons. We defer the proof to the technical report [25, §10.3].

Figure 3 shows the expected number of coupons per refund that Alice must send to Bob for trees with fan-outs of 2, 4, 8, 16, and 32. As the figure shows, with increasing fanout, the expected number of coupons increases. This is to be expected since there is a binary coupon, at more height, that can be used in lieu of a coupon of a tree with a larger fan-out. Further, binary coupons have more opportunities to group small contiguous ranges.

On the other hand, the benefit of larger fan-outs is that Bob can generate fewer coupons thus reducing the time it takes to answer queries and compute the associated group coupons. In situations where overheads during query runtimes are less desirable than during refund processing, a larger fan-out might be a better choice. We leave the problem of choosing the optimal fan-out to future work.

Algorithms for Tree-Structured Coupons. As before, we only consider full queries that are selections over a single relation Given a query Q that requests a range of tuples from the table, there are different ways to compute tree-structured coupons, and we find that the specific algorithm affects performance. We now describe two algorithms to construct the tree-structured coupons given a query.

The seller can construct the coupon trees in two ways: StreamTree (Algorithm 4) and BlockTree (Algorithm 3). In the BlockTree algorithm, the entire set of certificates is computed before the query's answer is returned to the user; while for the StreamTree algorithm, the certificates are computed as the cursor moves forward through the query's result set.

BlockTree, outlined in Algorithm 3, works by inserting the leaves for the current query Q into the temporary table `tempTable`. It then performs a series of `group-by-having` aggregation SQL queries to construct the layer one level above, and so on. The algorithm is blocking in nature, since

Algorithm 3: BlockTree Coupon Construction

```
  Input  : Query id $\tau$, query Q, user id u.
  Output: Compute the coupons and the query.
  begin
      // tempTable schema:  (height, id, version)
      // refunds schema:  (uid, height, id, version, qid, digest)
      tempTable ← ∅; refunds ← ∅; shiftval ← 1;
      height_c ← 0;
1     INSERT INTO tempTable
2         SELECT 0, id, sum(ver) FROM Q AS t GROUP BY id;
      while true do
          INSERT INTO tempTable
            SELECT height + 1, t.id ≫ shiftval
            FROM tempTable t
            WHERE t.height = height_c
            GROUP BY height + 1, t.id ≫ shiftval
            HAVING COUNT(*) > 1;
          // Below, in PostgreSQL, FOUND returns TRUE
          // if previous SQL query returns a non-empty answer.
          if NOT FOUND then
              break
          height_c ← height_c + 1
      // treeHash computes the hash as described in Equation 7.
      INSERT INTO refunds
        SELECT u, height, id, $\tau$, treeHash(u,height,id,ver,$\tau$)
        FROM tempTable t;
3     return (SELECT * FROM refunds), Q
```

Algorithm 4: StreamTree Coupon Construction

```
  Input  : Query id $\tau$, version ver, user id u, tuple id idIn.
  Output: Updates tempTable to incrementally compute the
          coupons.
  begin
      // tempTable has schema (height, id, version).
      height_c ← 0; id_c ← idIn;
      while true do
          INSERT INTO tempTable (height, id, version) VALUES
          (height_c, id_c, ver);
          if id_c % 2 != 0 && EXISTS (SELECT 1 FROM tempTable
          WHERE height = height_c AND id = id_c - 1) then
              height_c ← height_c + 1
              id_c ← id_c ≫ 1
              ver ← ver + (SELECT version FROM tempTable
              WHERE height = height_c AND id = id_c - 1)
          else
              break
```

the ids of the query's output tuples must be first inserted into `tempTable` before the query's answer can be returned.

We can also define a modification to BlockTree that avoids computing the query twice, once at Line 2 while populating `tempTable` with the leaves and another at the end in Line 3. We call this modification `BlockTreeInt` where the query `Q` is evaluated once and stored in a relation `result` in memory. This is done before Line 1. Subsequently, references to `Q` in Lines 2 and 3 are replaced by references to `result`. This approach can be potentially useful when evaluating the query is expensive.

The StreamTree algorithm works by ordering the results of a query by the primary key `id` and making a single pass over the data while adding an extra UDF that includes the code described in Algorithm 4. As the buyer advances the cursor, the temporary workspace, `tempTable`, is gradually populated with the tree for the query. For example, in Figure 2, if a query selects all the nodes, the nodes that are added to `tempTable` would be the order seen by a post-order traversal of the tree.

5.4 Coupons for Multi-Dimensional Queries

We briefly discuss a heuristic to construct group coupons for queries with multiple range and equality predicates.

Although the optimal choice of group coupons varies with the query workload, the preconditions for the safety of `GroupRefunds` suggest that database indexes are a natural way to structure the construction of the group coupons for queries beyond single ranges or point queries. Database indexes provide meaningful groupings of tuples that are frequently accessed together. We consider the following cases:

- R trees: For spatial queries over spatial datasets with R-tree indices (or variants such as R*-trees), we can construct the group coupons by using the dimensions of the minimum bounding rectilinear-rectangle (MBR) represented by the leaves or non-leaf nodes of the index. Specifically, each MBR of dimension n is represented by $I = ([l_1, u_1], \dots, [l_n, u_n])$, where l_i, u_i are the lower and the upper bound of the rectangle for dimension i, respectively. If a query returns all objects under the non-leaf node with the rectangle I, Bob can construct a group coupon with $gid = I$ and hash digest, as in Equation 7, as

  $$\mathcal{H}(\mathtt{uid} \oplus \mathtt{l_1} \oplus \mathtt{u_1} \oplus \cdots \oplus \mathtt{l_n} \oplus \mathtt{u_n} \oplus \mathtt{version} \oplus \mathtt{qid} \oplus \kappa)$$

  While processing refunds, Bob can safely refund coupons that have MBRs that are not contained within each other. For refunds where one MBR is included in another, Bob must traverse the larger MBR's index in the R-tree to ensure that smaller MBR's index is not present in the former's subtree. This check is necessary since R-trees permit overlapping MBRs and thus, containment in the dimensions does not imply that node of the smaller MBR is a descendant of the larger MBR in the R-tree.
- Range queries with additional point selection predicates: In this case, the group coupon consists of the group id of the corresponding range query (the tree-structured coupons or R tree coupons) along with the value of the selection predicates.

Mirroring coupons in the manner of indexing is not provably optimal. Further, the decision to generate group coupons versus individual coupons depends on the relative cost of generating the coupons and the cost of processing refunds. We experimentally (Section 6) explore this tradeoff but leave a theoretical analysis to future work.

6. EVALUATION

We now experimentally evaluate the performance of the various refund protocols and their implementations.

We answer the following questions:

1. How much can Alice benefit from paying only once for tuples and what performance penalty, if any, should she expect in lieu of this benefit?
2. How costly is it to compute group coupons versus computing only per-tuple coupons? Further, how much time do group coupons save when asking for refunds compared to single-tuple coupons?
3. How do the naïve approaches (Sec. 3) to pricing, *i.e.*, `CountBlock`, `CountStream`, and `History` perform compared to the refund-based approaches (Sec. 5.3), *i.e.*, `MonotoneRefunds`, `BlockTree`, `BlockTreeInt`, and `StreamTree`?

	1	2	4	8	64	512	4096
Agnostic	100	200	400	800	6.4K	51.2K	409.6K
H-A							
Uniform	100	200	400	799.5	6.4K	48.9K	286.7K
Zip 1.7	9.6	12.2	16.4	24.4	111	559	4143
Zip 3	5.2	7.9	12.3	20.4	107	555	4139

Table 1: Amount paid for data with different distributions for the parameters of `pkey.simple` and for different query answer cardinalities (header row). "Agnostic" shows the total amount paid without history-aware pricing, while the others, under heading "H-A", show the total amount with optimal history-aware pricing. "Uniform" denotes the case when the query's parameters are chosen uniformly at random, while "Zipf" denotes the case(s) where the query parameter l is chosen by sampling from the given Zipf distribution.

We run all experiments on a single server running PostgreSQL 9.4 over OS X 10.11.5, equipped with a 2.7 GHz Intel Core i7 processor and 16 GB DDR3 RAM. Coupon generation and refund verification algorithms are PL/pgSQL stored procedures and we use the SHA1 implementation of the module `pgcrypto` for hashing. The client resides on the same machine as the database.

The data setup for the experiments consists of a binary relation with two integer columns, `(tid, val)` in a table, `test`, with 524,288 (2^{19}) rows. Column `tid` is a primary key starting with a value of 0, while `val` is an integer column where the values are a random permutation of $\{0, \dots, N-1\}$ where N is the size of `test`.

The query workload consists of queries that ask for tuples satisfying predicates within a randomly chosen range of sizes in $\{1, 8, 64, 512, 4096\}$. We consider the following classes of queries: `pkey.simple` performs a range selection on the primary key, which is the key on which the data is sorted on disk and has a clustered index; `other.simple`, which performs a range query on the column `val` over which no indices have been constructed; and `join` which performs a join query where `test` is joined with itself.

The queries are:

```
pkey.simple:  SELECT * FROM test WHERE tid >= l AND tid <= u
other.simple: SELECT * FROM test WHERE val >= l AND val <= u
join: SELECT * FROM test a, test b
      WHERE a.val = b.tid AND a.tid >= l AND a.tid <= u
```

For identical values of l and u, the queries return answers with identical cardinalities.

6.1 Overall Results

We now investigate the benefits of optimal history-aware pricing and the associated performance penalty of approaches that can achieve such pricing.

In Table 1, we simulate the amount of money a buyer must pay with different approaches. We construct a workload with 100 instances of `pkey.simple` with varying size of the ranges, that is, $u - l + 1$, while using different distributions for selecting the value of l: *Uniform*, which chooses l uniformly at random, and *Zipf(α)*, which samples l from a heavy-tailed distribution where smaller values are chosen significantly more frequently than others. We experiment with $\alpha \in \{1.7, 3\}$, which enables us to vary the degree of the skew. We run 100 such simulations and take the average amount of money that the buyer must pay. For reference,

Cardinality		Query	Count		History	MonotoneRefunds		BlockTree		BlockTreeInt		StreamTree	
			Stream	Block		Stream	Block	Stream	Block	Stream	Block	Stream	Block
1	N	0.157	0.643	1.74	1.87	**0.571**	2.098	3.636	4.251	4.418	3.789	2.228	3.351
	Y					**1.33**	2.963	4.254	4.743	4.924	4.24	2.656	3.781
8	N	0.153	0.717	1.756	2.162	**0.749**	1.81	2.835	3.915	4.192	3.843	2.613	3.614
	Y					4.047	5.238	3.606	4.711	4.985	4.641	**3.435**	4.445
64	N	0.246	1.747	1.896	4.187	**2.142**	2.228	4.722	4.85	6.161	4.806	6.176	6.285
	Y					25.67	25.486	6.676	6.798	8.181	**6.782**	8.227	8.344
512	N	0.779	13.081	3.277	19.452	15.042	**5.371**	23.349	12.857	24.768	12.609	39.273	29.157
	Y					203.298	192.578	26.974	16.291	28.419	**15.882**	43.002	33.032
4096	N	5.339	288.107	13.188	135.121	307.839	**29.791**	348.191	72.799	354.568	72.867	476.158	194.947
	Y					2011.425	1742.64	353.986	78.552	361.268	**78.466**	482.887	200.678

Table 2: Total time, in ms, taken to evaluate `pkey.simple` with random initialization for l and a range specified in the column "Cardinality". The second column indicates whether the experiment includes the time to ask for a refund ("Y") or not ("N"). "Query" represents the time to execute just the query. The time, additionally, includes the overhead of counting the cardinality of answers for `CountBlock` and `CountStream`; updating the history bit vector for `History`; and, computing the coupons and counting the cardinality of answers for the refund-based approaches, respectively. In the table, "Stream" represents the `CountStream` strategy to compute price, "Batch" represents the `CountBlock` style. Best time for refund techniques is in bold.

"Agnostic" shows the cost of the data if no history-aware pricing is employed such as with `CountBlock`.

With Uniform distribution, it is unlikely that Alice may buy the same data across different queries, especially for short ranges. Cost savings only occur at larger ranges: refund-based pricing is 1.4× cheaper at range size of 4096. But savings are dramatic for skewed distributions. With $\alpha = 1.7$, the history-aware pricing for point queries is 10× cheaper than history-agnostic pricing, while for $\alpha = 3$, it is 19× cheaper. For longer ranges, history-aware pricing is 99× cheaper than a history-agnostic approach.

We now look at the overhead of obtaining these cost savings. In Table 2, we show the time taken to answer queries by the naïve and refund-based techniques. It shows the total time that includes (a) the time to evaluate `pkey.simple` with range lengths specified in column *cardinality*, and (b) the overhead of the associated pricing technique, that is,

- Counting for `CountStream` and `CountBlock`.
- Updating the history bit vector for `History`. In addition to `HistoryStream` and `HistoryBlock`, we also use a blocking variant of `HistoryBlock`, called `HistoryInt`, which stores the query's result in a temporary table, which is subsequently used to update the history bit vector using `HistoryBlock`. `HistoryBlock` runs the query twice, the first run updates the history bit vector while the second returns the answer. `HistoryStream` and `HistoryInt` run the query once. `History` in Table 2 refers to `HistoryBlock` which was the best history algorithm for `pkey.simple`.
- Computing coupons for refund-based techniques along with counting the price, once using `CountStream` and once using `CountBlock`.

In the second column, a 'N' indicates the time without asking for refunds, while 'Y' indicates the time with refunds.

All techniques take more than 2× more time than the query only runtime. With refunds, the best refund-based technique becomes at least 6× more expensive than the best count technique, `CountStream` at smaller cardinalities and `CountBlock` at larger. All refund-based techniques are slower than the best count technique, even without the overhead of asking for refunds since they also count the price.

Compared to `History`, the best refund techniques with the refund round is 1.5× slower for queries with short ranges (8 and 64); but for point queries and longer ranges, the best refund technique are 1.4× to 1.7× faster.

In the best case, refund-based pricing provides reduced data costs compared to history-agnostic pricing techniques and reduced query execution time compared to history-based pricing methods. For large ranges, `BlockTreeInt.Block` is 1.72× faster than `History` and also protects privacy.

6.2 Overhead of Refunds

We now compare the time to compute group coupons in `GroupRefunds` to the time for only per-tuple coupons in `MonotoneRefunds`. Then, we compare the time saved in asking for refunds with `GroupRefunds` versus `MonotoneRefunds`. We measure the overhead of using `MonotoneRefunds` versus `GroupRefunds` on (a) the time to evaluate a query, its price, and return the result and the coupons, and (b) the time to ask for refunds of data previously purchased.

We assign random l and u values to `pkey.simple` to obtain queries that are executed twice. Thus, all the tuples in the second query are eligible for refunds. We compare the time it takes to execute the query, compute its price, and compute the coupons using `MonotoneRefunds` and `BlockTree`. Then, for `MonotoneRefunds`, we ask for refunds one tuple at a time, while for `BlockTree`, we ask for the group refunds for the largest groups (while avoiding overlaps) until all the tuples are covered.

Table 2 shows the results. When comparing only the time to evaluate the query, price, and the coupons, `MonotoneRefunds` is faster than `BlockTree` since `BlockTree` must compute additional group coupons along with the singleton group coupons. For point queries, `BlockTree` is 6.4× slower, while for larger ranges, such as 4096, it is 2.4× slower. This is expected since `MonotoneRefunds` needs to do just one pass over the table and computes the coupons on the fly as the cursor is advanced. `BlockTree` must make two passes of the data, once to select the leaves of the tree-structured coupons and again to evaluate the query itself. It must also suffer the additional cost of computing the tree.

But if we also include the overhead of asking for refunds, this advantage quickly vanishes as range sizes are increased.

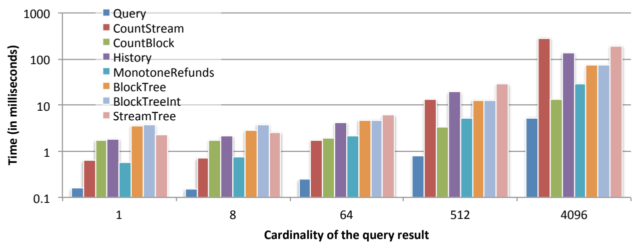
While `MonotoneRefunds` is 3.2× faster for point queries, for longer ranges, `BlockTree` is faster by 1.1×, 3.8×, 11.8×, and 22.2× for ranges of sizes 8, 64, 512, and 4096, respectively.

Thus, if a query is expected to return a small number of tuples or if it is known that group refunds can not be constructed, say when the tuples selected do not have adjacent tuple ids, `MonotoneRefunds` will outperform `GroupRefunds`.

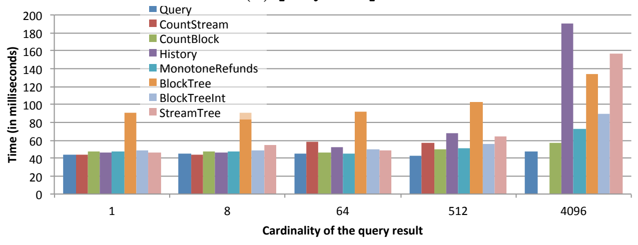
One way to improve `BlockTree` and other tree structured coupons is to explore an increase in the fanout of the internal tree nodes. Then, fewer coupons would be computed during the query. Further, in an actual deployment, refund requests can be asked when the buyer has spare computation cycles as opposed to being asked after each query. This does not reduce the workload on the seller, though.

6.3 Naïve versus Refund-Based Techniques

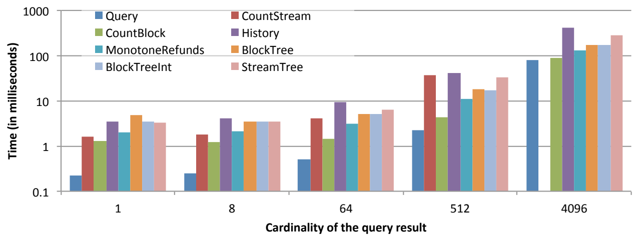
In these experiments, we assign random l and u values to the test queries. These randomly parameterized queries are executed once along with the additional processing of the corresponding naïve or refund-based technique and counting to compute the price. For refund-based techniques, we do not show the overhead of asking for refunds because the overhead is sensitive to the actual overlap in the queries.



(a) `pkey.simple`



(b) `other.simple`



(c) `join`

Figure 4: Total time, in ms, averaged over 100 executions of the workloads, to evaluate the query and run the pricing techniques. Subcaptions mention the parameterized query. For refund-based techniques, only the time to run the query and generate coupons is shown. `CountStream` not shown for '4096' as it has a very large value.

`pkey.simple`. Figure 4a shows the average time per query for `pkey.simple`. As expected, `MonotoneRefunds` outperforms the group-based coupons (note that we do not include the time to ask for refunds). `MonotoneRefunds` also outperforms `History`, where `HistoryBlock` was the most performant implementation since the cost of modifying bits for each output tuple and writing those edits back to disk become increasingly expensive. For large range size, `CountBlock` performs better than `CountStream`. This is because `CountBlock` only uses two SQL statements to execute the query and compute the count whereas `CountStream` must execute an `UPDATE` statement to update the running count of tuples for the query every time the cursor is advanced. This overhead becomes significant as the cardinality of the query increases.

`pkey.other`. Figure 4b shows the average time per query for `other.simple`. Unlike for `pkey.simple`, `MonotoneRefunds` does not significantly outperform the alternatives. This is because the query itself is expensive: without indices, a full scan of the table is needed to compute answers and this cost dominates the total cost. As result sizes increase though, the overheads become noticeable. For `History`, different implementations did best for different ranges. For short ranges, `HistoryStream` was the best implementation, while for longer ranges, `HistoryInt` was the best. `HistoryBlock` was never the best for `other.simple` since it computes the query twice, which is expensive without an index. `HistoryStream` was especially costly at large ranges, since the column `val` is a random permutation and thus the indices in the history bit vector, corresponding to their tuples, are no longer clustered to adjacent bits in the history bit vector and this increases the overhead of commits. Finally, `BlockTree` is approximately 2× more expensive than `CountBlock` and `MonotoneRefunds` since the query is more expensive to compute and `BlockTree` must execute the query twice.

`join`. To identify the distinct `tids` used by the query, its output is stored in a temporary table, `res`, and the following query is executed:

```
SELECT DISTINCT(id) AS tid FROM
(SELECT a.tid AS id FROM res UNION SELECT b.id AS tid FROM res)
```

These distinct `tids` are used to compute the price and the coupons. Figure 4c shows that the trends are similar as for `pkey.simple`. `BlockTree` has similar performance to `BlockTreeInt` since the step of identifying distinct `tids` implicitly stages the query as `BlockTreeInt` does. The implicit staging also removes the advantage that `CountStream` and `StreamTree` had over the blocking versions at smaller cardinalities since all techniques touch the answer tuples thrice: once while computing the answer tuples, once while determining unique `tids`, and lastly to retrieve them from the temporary table as answers. As with `pkey.simple`, `HistoryBlock` was the best `History` implementation.

If we only consider the time to compute coupons and the query, then `MonotoneRefunds` will always be the fastest amongst the refund-based techniques. But in applications with high overlap in the data purchased through different queries, the cost of refunds can become significant. In such cases, `GroupRefunds` might be more efficient than `MonotoneRefunds`. Unfortunately, as can be seen from Figures 4a, 4b, and 4c, different group refund techniques do well in different settings. In Figures 4a and 4c, `BlockTree` does uniformly well across many different ranges while in Figure 4b, `BlockTreeInt` performs uniformly well. Also, for both single-table queries, `StreamTree` outperforms other group refunds for queries with small ranges while the reverse is true for `join`.

7. RELATED WORK

Our solution relies on explicit support from the seller. In the absence of such support, as shown in Example 1.1, it may be impossible to provide optimal history-aware pricing of data. But, buyers can still reduce their costs by caching answers to queries they purchase. They can subsequently use techniques from query answering using views [10] to only acquire such that that is not present in their caches and integrate the new data with the cached data to determine the answer to their queries. Systems that provide semantic caching [3, 4, 19] and transactional caches [16, 17] are examples of such systems.

Apart from pricing APIs by summing up the cost of the tuples that are returned due to an API call, other forms of pricing methods have also been proposed in the literature, though they are not as widespread as tuple-based pricing. The common idea in all the approaches is to directly price queries as opposed to pricing individual tuples. Approaches have been proposed that price data based on minimal why-provenance [23], information and determinacy [11, 12, 15], and statistical noise [13].

Optimal history-aware pricing, where prices are assigned to views [12,14] instead of individual tuples, is not in PTIME in general. For queries which can be priced in PTIME, the history-bit-vector approach is equivalent to our previous "view pricing" framework [12] if (a) each view corresponds to exactly one tuple and (b) whenever a view is purchased, instead of setting the view's price to zero, the corresponding bit is set. The "view pricing" implementation stores each priced view as a row, which is less efficient than representing all views using a single bit-vector.

To adapt coupons to query-pricing techniques such as view pricing [12], coupons must be created per query instead of per tuple. Our protocols, when applied to coupons on queries, will be *safe* but not *optimal*. Intuitively, this is because with tuple pricing, all refund requests can be decomposed into individual requests with the same tuple id; but for query pricing, it is possible that query set, P, is collectively equivalent to query Q, while no strict subset of P equals Q. Our protocol, where coupons must refer to the same entity (tuple or query), can not handle such cases.

Our core idea, using coupons to achieve anonymous and almost stateless query pricing, augments previous pricing approaches. This paper demonstrates the feasibility for tuple based pricing, but coupon and protocol design for other pricing forms is a fruitful direction of future research.

8. CONCLUSION

We provide a novel, lightweight and fast method to support optimal, history-aware pricing of data APIs. With our techniques, even if a buyer makes multiple API calls and ends up purchasing the same data item more than once, she is only charged once for the purchase. To enable this, we propose a framework for pricing that allows buyers to refund repeat purchases of data. We then provide a compact, secure and tamper proof protocol that enables such refunds and guarantees that if there is a repeat purchase, it is always possible to get refunds. Subsequently, we generalize the protocol to handle updates and multiple users; and provide performance improvements through the use of group refunds. We experimentally evaluated our protocol and compare it to current pricing techniques that do not provide history-aware pricing.

Acknowledgements. We thank the anonymous reviewers. This work was supported in part by a gift from NEC Labs.

9. REFERENCES

[1] J. Becla and K.-T. Lim. Report from the SciDB meeting. http://xldb.slac.stanford.edu/download/attachments/4784226/sciDB2008_report.pdf, 2008.
[2] Digital Folio. www.cartbound.com/PriceIntelligence/API.
[3] B. Chidlovskii and U. M. Borghoff. Semantic caching of web queries. *The VLDB JournalThe International Journal on Very Large Data Bases*, 9(1):2–17, 2000.
[4] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan, et al. Semantic data caching and replacement. In *VLDB*, volume 96, pages 330–341. Citeseer, 1996.
[5] DataSift: Pylon Facebook API. http://datasift.com/products/pylon-for-facebook-topic-data/.
[6] Factual. http://developer.factual.com.
[7] Foursquare terms of use. https://foursquare.com/legal/api/platformpolicy.
[8] Google Maps. maps.google.com.
[9] GNIP. https://gnip.com/products/realtime/firehose/.
[10] A. Y. Halevy. Answering queries using views: A survey. 10(4):270–294, 2001.
[11] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-based data pricing. In *PODS*, pages 167–178, 2012.
[12] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Toward practical query pricing with querymarket. In *SIGMOD Conference*, pages 613–624, 2013.
[13] C. Li, D. Y. Li, G. Miklau, and D. Suciu. A theory of pricing private data. In *ICDT*, pages 33–44, 2013.
[14] C. Li and G. Miklau. Pricing aggregate queries in a data marketplace. In *WebDB*, pages 19–24, 2012.
[15] B.-R. Lin and D. Kifer. On arbitrage-free pricing for general data queries. *PVLDB*, 7(9):757–768, 2014.
[16] D. R. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, volume 10, pages 1–15, 2010.
[17] D. R. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, 2012.
[18] quandl. www.quandl.com.
[19] Q. Ren, M. H. Dunham, and V. Kumar. Semantic caching and query processing. *Knowledge and Data Engineering, IEEE Transactions on*, 15(1):192–210, 2003.
[20] F. Schomm, F. Stahl, and G. Vossen. Marketplaces for data: An initial survey. *SIGMOD Rec.*, 42(1):15–26, May 2013.
[21] Sloan Digital Sky Survey. http://cas.sdss.org.
[22] Socrata. http://www.socrata.com/.
[23] R. Tang, H. Wu, Z. Bao, S. Bressan, and P. Valduriez. The price is right - models and algorithms for pricing data. In *DEXA (2)*, pages 380–394, 2013.
[24] Twitter API. dev.twitter.com/overview/terms/agreement-and-policy.
[25] P. Upadhyaya, M. Balazinska, and D. Suciu. Price-optimal querying with data apis. https://www.dropbox.com/s/y8t7c82hndz5d24/paper.pdf?dl=0, University of Washington, February 2016.
[26] Windows Azure Marketplace. http://datamarket.azure.com/.
[27] Xignite. www.xignite.com.
[28] Yelp API. www.yelp.com/developers/display_requirements.