Query Optimization over Cloud Data Market

Yu Li, Eric Lo, Man Lung Yiu, Wenjian Xu
Department of Computing
Hong Kong Polytechnic University
{csyuli, ericlo, csmlyiu, cswxu}@comp.polyu.edu.hk

ABSTRACT

Data market is an emerging type of cloud service that enables a data owner to sell their data sets in a public cloud. Buyers who are interested in a certain dataset can access the data in the market via a RESTful API. Accessing data in the data market may not be free. For example, it costs USD 12 per month to obtain 100 "transactions" from the WorldWide Historical Weather dataset in Windows Azure Data Marketplace, where a transaction is a unit of result size (e.g., a query result of 4400 records would consume 44 transactions as Windows Azure Data Marketplace confines one transaction to 100 records). Therefore, in this paper, we present PayLess, a system that helps data buyers to optimize their queries so that they can obtain the query results by paying less to the data sellers. Experiments over synthetic data and real data sets in Windows Azure Marketplace show that PayLess can cost-effectively handle SQL query processing over data markets.

1. INTRODUCTION

Data market [1, 16, 42] is an emerging type of cloud service that enables a data owner to host and sell their datasets in a public cloud. Buyers who are interested in a certain dataset can access the data in the market via a RESTful API. The REST based API has function-call like interface $\mathcal{X} \rightarrow \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are sets of attributes: given a range or a value for an attribute in $\mathcal{X}$, the data market returns values for the attributes in $\mathcal{Y}$ (if no values are specified for $\mathcal{X}$, the whole table is returned). For example, the Worldwide Historical Weather (WHW) dataset [13] in Windows Azure Marketplace [1] may take a country name and a date, and return a set of tuples, each details the temperature, precipitation, dew point, sea level pressure, windspeed, and wind gust recorded by each weather station in that country on that date.

Accessing data in the data market may not be free. For example, it costs USD 12 to grant access to every 100 "transactions" to the WHW data, where a transaction is a unit of result size (e.g., a query result of 4400 records costs 44 transactions in Windows Azure Marketplace, which confines one transaction to 100 records). There is an increasing trend of selling valuable datasets in data market [31]. Correspondingly, we envision that there is an increasing demand from end users (data buyers) to carry out analytics that involve those datasets. To this end, in this paper, we present PayLess, a system that helps users to optimize their queries so that they can obtain the query results by *paying less* to the data sellers.

Query optimization is never trivial. First, from a data buyer's (the company or the organization) perspective, it is hard to know in advance how many queries will be posed by their end users eventually. Otherwise, downloading the whole dataset would become a viable plan when the foreknowledge tells that the number of transactions incurred by user queries would eventually exceed the number of transactions required to download the complete data set. Second, query optimization would never work well without rich data statistics. Unfortunately, datasets in data market are rarely tagged with rich statistics (e.g., no value distribution), although basic information like the size (cardinality) of each table and the domain size of the attribute is usually available.

Tackling the above two challenges sounds not difficult, especially that we can build a *learning* optimizer like LEO [46] so that it begins with little statistic and introduces a feedback loop to correct the statistics when more queries are issued. The evil, however, lies in the detail of adopting the learning approach to data market query optimization.

First, learning-based optimizers like LEO [46] and POP [38] are originally designed for traditional databases that have full access to the data. In contrast, the access pattern of data market is restricted to only $\mathcal{X} \rightarrow \mathcal{Y}$ style. When a data source has limited access patterns, (a) operations might become complicated and (b) specialized access paths may shine. An example of (a) is that a query that asks `Country = 'Canada' OR Country = 'Germany'` has to decompose into two queries, one asks for `Country = 'Canada'` and another asks for `Country = 'Germany'`. An example of (b) is *bind joins* (other names include theta semi-join, dependent join) [27]. To explain, consider the real access patterns of Worldwide Historical Weather (WHW) dataset in Windows Azure Marketplace listed in Figure 1a.[1] The access patterns are specified using a notation of binding patterns extended from [27]. We write $R^{\alpha}(A_1, A_2, A_3)$ to denote a table R in the data market with three attributes A_1, A_2, and A_3 and *binding pattern* α. We write $\alpha = R(A_1^b, A_2^f)$ to denote a binding pattern that in any query accessing R, the value of attribute A_1 must be *bound* (given/specified). In contrast, the value of attribute A_2 is *free* to be specified or not specified in any query. If an attribute is not included in the binding pattern (e.g., A_3), it is solely served as an output attribute in a query result. In other words, if an access pattern of a table has only free attributes, then we can download the whole table by not specifying any value to any attribute.

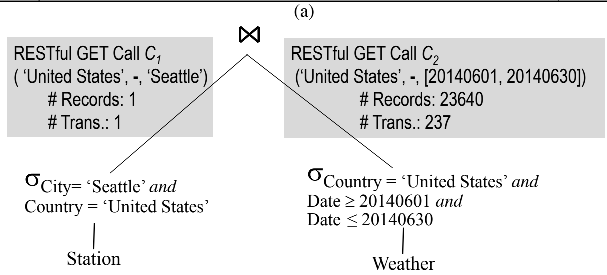
Now, consider the following SQL query that asks the WHW

[1]The attribute names here are renamed for better exposition.

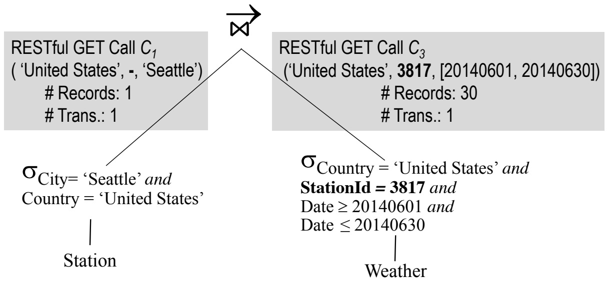
10.5441/002/edbt.2015.21

Data Set	Schema and Access Pattern α	Size
WHW	Station$^{\alpha_1}$(Country, StationID, City, State$\cdots$) α_1=Station(Country^f, StationID^f, City^f)	3962
	Weather$^{\alpha_2}$(Country, StationID, Date, Temperature$\cdots$) α_2=Weather(Country^f, StationID^f, Date^f)	19549140
EHR	Pollution$^{\alpha_3}$(ZipCode, Rank, Latitude, Longitude$\cdots$) α_3=Pollution(ZipCode^f, Rank^f)	44210
local	ZipMap (ZipCode, City)	

(a)



(b) Plan P_1



(c) Plan P_2

Figure 1: Query Processing in Data Market

dataset for the daily temperature of Seattle in June 2014:

```
SELECT Temperature          -------// Query Q1
FROM Station, Weather
WHERE City = 'Seattle' AND
      Country = 'United States' AND
      Date >= 20140601 AND Date <= 20140630 AND
      Station.StationID = Weather.StationID
```

Figure 1b shows an execution plan P_1 for this SQL. It first submits two RESTful GET calls C_1 and C_2, where C_1 gets the StationID of Seattle from Station table, and C_2 gets the weather records for all stations in the United States on June 2014 from Weather table. The final query result is obtained by carrying out a *local join* (i.e., regular join) operation at the end user (data buyer) side because joins cannot be done at the data market [1]. In plan P_1, a total of 238 transactions were incurred – one was spent on RESTful call C_1 and 237 were spent on RESTful call C_2 (there are 788 weather stations in the US and each station contributes 30 days records, resulting in $\lceil 788 \times 30/100 \rceil = 237$ transactions). Figure 1c shows an alternate execution plan P_2. It first gets the list of StationIDs of Seattle (call C_1). Then, it carries out a bind join ($\overrightarrow{\bowtie}$) operation that binds each StationID (e.g., 3817) to an individual RESTful call to Weather. Finally, the weather records for each station in Seattle are collectively retrieved and returned. In this case, plan P_2 incurs only two transactions: call C_1 costs one transaction and call C_3, which returns 30 days of weather records for the only one weather station in Seattle, costs also one transaction.

Second, although there are optimizers designed for queries over remote data sources with limited access patterns (e.g., [17, 24, 27, 33–36, 40, 45]), they focus on minimizing the number of calls to the remote data sources so as to reduce the overall execution time. As an example, assume that there are 15 weather stations in Seattle, those optimizers will pick plan P_1 because it incurs only two RESTful calls (C_1 and C_2). In data market, although P_2 needs to bind each Seattle's weather station id, resulting in $1 + 15 = 16$ RESTful calls and 16 transactions (each transaction returns 30 days of records for each weather station), it is still more economy than P_1, which requires 238 transactions. On the other hand, if we further assume that there are only 20 weather stations in the United States and 15 of them are in Seattle. Then, plan P_1 will cost only $1 + \lceil 20 \times 30/100 \rceil = 7$ transactions. In contrast, plan P_2 still costs 16 transactions. In this case, P_1 is better than P_2.

Summing up the above, we need a (i) learning-based optimizer that (ii) includes bind join as an access path with the goal of (iii) minimizing the amount of (intermediate) retrieved data measured in terms of data market pricing units. Traditional learning-based optimizers satisfy (i) and partially satisfy (iii)[2] but not (ii). Optimizers for queries over remote data sources satisfy only (ii). Therefore, the principal contributions of this paper are centered around the issues of building an optimizer for PayLess that satisfies all (i), (ii), and (iii) above. Those include:

- Defining the cost model and search space for data market query optimization.
- Devising effective techniques to reduce the amount of intermediate retrieved data (e.g., by adapting semantic query rewriting methods) and integrating those techniques into our optimizer.
- Implementing a prototype and evaluating its performance through extensive experiments over synthetic data and real data.

The remainder of this paper is organized as follows. Section 2 gives more background about the data market. Section 3 presents the architecture of PayLess. Section 4 describes the details of PayLess's optimizer. Section 5 reports the results of the evaluation. Section 6 discusses the related work and Section 7 concludes.

2. PRELIMINARIES

According to a recent survey [2], the three most established data marketplaces are Factual [8], Microsoft Windows Azure Data Marketplace [1], and DataMarket [4]. Factual [8] and DataMarket [4] are specialized data markets that sell datasets in a very specific domain (e.g., Factual sells mainly geographical data and DataMarket sells mainly economic indicators). Microsoft Windows Azure Data Marketplace offers data sets in all kinds and many popular data resellers in smaller size like Wolfram Alpha [11], ESRI [7], World Bank [12], data.gov [3], Xignite [14] also provide their data in the Windows Azure Data Marketplace [1]. After Infochimps [9], one of the early data market entrants, gradually leaves the data market business [5, 10], Microsoft Windows Azure Data Marketplace is becoming the de facto data market [2]. Therefore, in this paper, we base our setting on Windows Azure Data Marketplace.

2.1 Data Market

A data market hosts and sells multiple datasets. Each dataset's access/binding pattern is defined by the data owner on per table basis. For numeric attributes, the input can be bound with a single value or a range like $[150, 200)$. Datasets in data market are tagged with very basic statistics, normally the domain of each attribute and

[2]Traditional optimizers also aim to generate plans that minimize intermediate result size of each operation (e.g., push down selection).

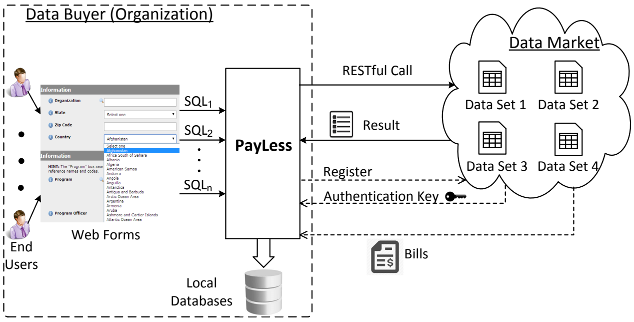


Figure 2: Setting of PayLess

the number of records (cardinality).[3] Datasets in a data market are *append-only* because they are released for analytic purposes. New data could be added periodically (e.g., every month). The price of accessing data is mainly based on the number of tuples retrieved. A *transaction* represents a page of t tuples (e.g., 100 tuples) and it is the smallest pricing unit. Let p be the price per transaction for a particular dataset. Then, the total price of a RESTful call is:

$$p \cdot \lceil \frac{\text{number of resulting records}}{\text{number of tuples per transaction } (t)} \rceil \qquad (1)$$

For easy exposition, in the subsequent discussion, we assume $p =$ \$1 and a transaction page size is $t = 100$ tuples.

2.2 Queries over Data Market

Figure 2 shows the target setting of PayLess. An organization is interested in carrying out certain analytics that involve datasets hosted in a data market. The organization thus registers with the data market to obtain the authentication access keys of the datasets. The access keys are stored in PayLess, which constructs RESTful calls to the data market when necessary. PayLess encapsulates the details of interacting with the data market and exposes a SQL query interface for client query processing. A SQL query to PayLess can query against both tables in a local DBMS and tables in data market. The following is an example PayLess query that aims to retrieve the average temperature for each city in a country whose environmental pollution rank is lower than a threshold within a period:

```
SELECT City, AVG(Temperature)
FROM Pollution, Station, Weather, ZipMap
WHERE Station.Country = Weather.Country = ? AND
      Weather.Date >= ? AND Weather.Date <= ? AND
      Pollution.Rank <= ? AND
      Pollution.ZipCode = ZipMap.ZipCode AND
      ZipMap.City = Station.City AND
      Station.StationID = Weather.StationID
GROUP BY City
```

This query involves joining four tables: the Station and Weather tables from the aforementioned Worldwide Historical Weather (WHW) [13] dataset, another Data Market table, the Pollution table from the Environmental Hazard Ranking (EHR) [6] dataset, and a local table that maps Zip codes to a city name. The access patterns of these tables are shown in Figure 1a. We expect SQL queries to PayLess are parameterized queries embedded in certain application so that users (e.g., data scientists) issue the queries by specifying the parameter values via a web interface. We do not expect the organization restricts her users the number of queries to the data market because that is counter-productive.

[3]If not publicly available, the data sellers would release the basic statistic to data buyers upon email requests [1].

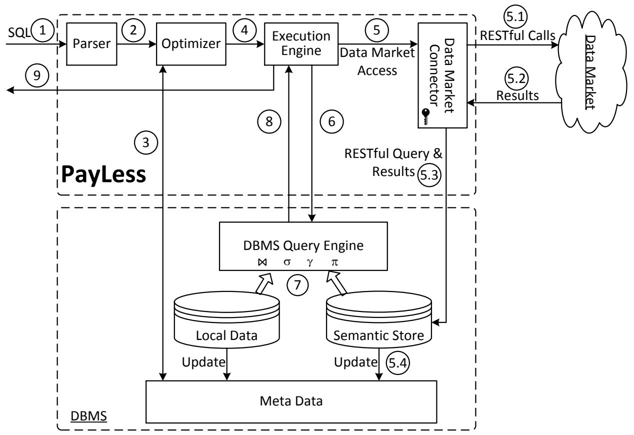


Figure 3: System Architecture

3. SYSTEM OVERVIEW

Figure 3 shows the architecture of PayLess. It is designed to be lightweight and offloads most query processing to a DBMS query engine. It accepts and parses a SQL query (with parameter values instantiated) ①. The parser differentiates local tables and tables from the data market using the information (e.g., the table name) obtained when registering with the data market (see Figure 2). Then, the optimizer of PayLess optimizes the query ② by consulting the statistics of local and data market data ③. The optimized query is then passed to an execution engine ④. A query, after optimization, may be able to skip some or the entire access to the data market. When it is necessary to access the data market, the execution engine will pass the access requests to the data market connector ⑤ and let the connector interact with the data market (5.1) (5.2). PayLess stores all the data market access requests and their returned data in a semantic store (5.3). Whenever new data is retrieved from the data market, PayLess will update its statistics (5.4). In our implementation, we implement our updatable statistics using ISOMER [44]. After this step, all data required by a query should be ready and stored in the DBMS and the execution engine of PayLess instructs the DBMS query engine ⑥ to process the query ⑦. In the end, the execution engine of PayLess retrieves the query result from the DBMS ⑧ and then returns it to the front end ⑨.

PayLess is supposed to be installed by each data buyer and serves all the end users from the same data buyer. As a data buyer would not be interested in all datasets available in the data market, the storage space (for the DBMS) is not a problem here. Cache management is out of PayLess's interest because we deliberately use cheap storage space to store all intermediate results (i.e., no eviction) in order to eschew retrieving redundant data from the data market. Besides, PayLess is indeed amenable for any updatable statistic. As our focus of this paper is to give a proof-of-concept first solution, we will test other updatable statistics (e.g., [25]) in place of ISOMER in the next version of PayLess.

4. QUERY OPTIMIZATION

PayLess's optimizer follows the typical bottom-up, cost-based, and dynamic programming approach [28]. That is, it first consid-

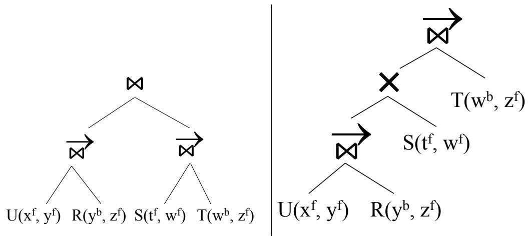


Figure 4: Bushy tree v.s. Left-deep tree

ers the best plan for single relations, then the best plan for joining two relations, and then for three relations, so on. On top of that, PayLess's optimizer considers bind joins $\overrightarrow{\bowtie}$ as an access path in addition to the regular join $\bowtie$. The key feature of PayLess's optimizer is that it carries out *semantic query rewriting* to optimize its queries using the query results stored in the semantic store. Semantic query rewriting [23] is not new, but later we will explain why it is not included in limited access query optimizer (e.g., [17,27,45]) and why it is helpful to us here. We will also explain the limitations of current semantic query rewriting techniques in our setting and our solutions to unlock their potential and integrate them into our optimizer.

This section describes how to derive the optimal execution plan after parsing a SQL query. We first propose several techniques to reduce the plan search space and prove their correctness (see Section 4.1). After that, we illustrate the semantic query rewriting method used in PayLess (see Section 4.2). In the end, we end with some discussions about our query optimization approach (see Section 4.3).

4.1 Plan Space

When optimizing queries for limited access pattern data sources, *bushy trees* are included in the plan space to avoid plans with Cartesian products [27]. For example, consider a query that joins four relations U, R, S and T with access patterns: $U(x^f, y^f)$, $R(y^b, z^f)$, $S(t^f, w^f)$, $T(w^b, z^f)$. Since R has a bind attribute y, it must require values for attribute y to retrieve tuples. In the example, the only choice is thus to carry out a bind join $U \overrightarrow{\bowtie} R$. Similarly, since T has a bind attribute w, it must require values for attribute w to retrieve tuples. In the example, the only choice is thus to carry out a bind join $S \overrightarrow{\bowtie} T$. After that, the only way is to join them together by using a local join, resulting in a bushy tree like Figure 4a. So, if only left-deep plans are allowed, a "logical" cross product must be used to *logically connect* the relations like Figure 4b[4].

Including bushy trees would significantly enlarge the search space. In our problem setting, as our primary goal is to minimize the money-to-pay, we exclude bushy trees in our plan space because:

THEOREM 1. *Given any plan P, we can transform it to a left-deep plan P' such that $\phi(P) \geq \phi(P')$, where $\phi(\cdot)$ denotes the total price of a plan. In other words, the optimal plan must be one of the left-deep plans.*

PROOF. In what follows, we use the terms RESTful call, leaf node, and relation/table interchangeably.

First, we re-iterate a very important fact:

Fact *Only leaf nodes in P contribute to the price $\phi(P)$ because they represent RESTful calls to the data market. Therefore, $\phi(P)$ equals to the sum of prices of leaf nodes in P.*

Without loss of generality, we name the leaf nodes (RESTful calls) in P from left-to-right as: $C_1, C_2, \cdots, C_n$.

We write $P^{(k)}$ to denote that, for all leaf nodes of P, if named from left-to-right, the first k leaf nodes form a left-deep subtree. So, given a plan P with n leaf nodes, if we write $P^{(n)}$, we mean P is a complete left-deep tree. As an example, for the bushy tree P in Figure 4a. $P^{(1)}$ and $P^{(2)}$ hold. As another example, let P be the plan in Figure 4b, then we see that $P^{(1)}$, $P^{(2)}$, $P^{(3)}$ and $P^{(4)}$ all hold.

Now, we proceed to prove $\phi(P) = \phi(P^{(1)}) \geq \phi(P^{(2)}) \geq \cdots \geq \phi(P^{(n)})$. In the following, we first prove $\phi(P^{(1)}) = \phi(P)$ and then prove that for a given $1 \leq k \leq n-1$, we have $\phi(P^{(k+1)}) \leq \phi(P^{(k)})$.

Base case: $k = 1$ $P^{(1)}$ simply means we just look at the left-most leaf nodes of P without moving any nodes, so the cost of the whole plan P is unchanged: $\phi(P^{(1)}) = \phi(P)$.

General case: $\phi(P^{(k+1)}) \leq \phi(P^{(k)})$

When C_{k+1} is C_k's uncle: Figure 5a illustrates this case. In this case, the left-most $k+1$ leaf nodes form a left-deep subtree. So, P^{k+1} holds. Note that we did not move any leaf node yet, so the plan cost would not change: $\phi(P^{(k+1)}) = \phi(P^{(k)})$.

When C_{k+1} is not C_k's uncle: Figure 5b illustrates this case. In this case, the uncle node of C_k, say U, must be a non-leaf node and its subtree contains C_{k+1}. Let T_F be the left-deep subtree rooted at F, the father of C_k. Further, we let G be the grandfather of C_k. Finally, we let T_{UL}, T_{UR} be the left and right subtrees rooted at U, respectively.

We now explain that making $P^{(k+1)}$ holds by joining T_F with C_{k+1} through a new node G' would not increase the overall plan cost. Figure 5c illustrates the resulting plan P' with $P'^{(k+1)}$ holds.

First, we see that the price of subtree T_F is the same among P and P'.

Second, the price of C_{k+1} is the same in both P and P' because C_{k+1} takes the same join result from T_F no matter G or G' is a bind join or a regular (local) join.

Now, we consider the price for each node (other than C_{k+1}) in T_{UL} and T_{UR} in P and P'. Let C_u be such a node. First, if C_u does not require any binding from C_{k+1}, then the price of C_u in P' is unchanged. Second, if C_u requires binding values from C_{k+1}, then the price of C_u depends on the number of distinct binding values from C_{k+1}. Note that in P', C_{k+1} has been joined with the others earlier than P, that causes the number of binding values to C_u possibly decreases. So, the price for C_u would not increase.

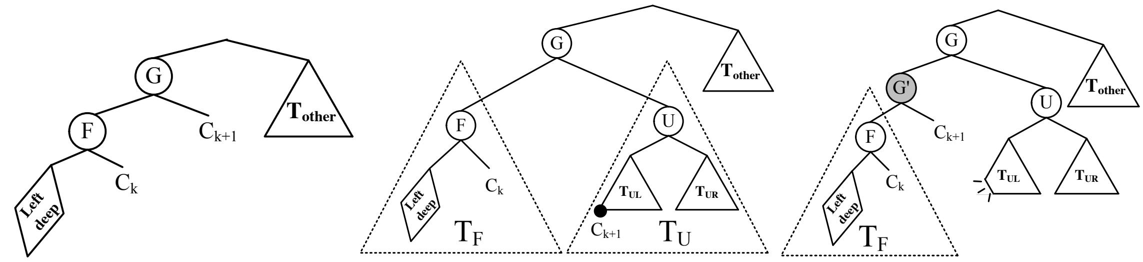
Finally, we look at the subtree T_{other}. As the result of the left operand of T_{other} remains the same, the price of T_{other} is unchanged.

As the price of any C_i in P would not increase, we have $\phi(P^{(k+1)}) \leq \phi(P^{(k)})$. $\square$

Traditional optimizers include only left-deep plans as a heuristic to improve the efficiency of the plan search. In PayLess, with Theorem 1, enumerating only left-deep plans is not a heuristic but with a guarantee that the optimal plan is not lost. Furthermore, in PayLess, including Cartesian product is not a problem because that would not contribute any extra data market transaction.

In addition to enumerating left-deep plans only (Theorem 1), PayLess's optimizer further trims the search space by first joining all relations that incur zero price to the data market. Those relations can either be local relations or relations whose required tuples can be found in the semantic store. In the following, we show that such

[4]The cross product is just logically connecting intermediate results $U \overrightarrow{\bowtie} R$ and $S \overrightarrow{\bowtie} T$. Physically, $(U \overrightarrow{\bowtie} R)$ joins $(S \overrightarrow{\bowtie} T)$ is done by the DBMS, using any equi-join implementation like hash-join.



(a) When C_{k+1} is C_k's uncle (b) Before transformation P (c) After transformation P'

Figure 5: Illustration figures for Theorem 1

zero-price-relations-join-first idea retains the optimal plan in the plan space:

THEOREM 2. *Let $P = \langle C_1, C_2, \ldots, C_n\rangle$ be a left-deep plan with a leaf node (RESTful call) C_i whose price $\phi(C_i) = 0$. Then, the plan $P' = \langle C_i, C_1, \ldots, C_n\rangle$ has $\phi(P') \leq \phi(P)$.*

PROOF. We divide the other calls into two groups: (1) RESTful calls that executed before C_i, i.e. C_1 to C_{i-1}, and (2) RESTful calls that executed after C_i, i.e. C_{i+1} to C_n.

If we move C_i to the left-deepest node of P:

- $\phi(C_i)$ is unchanged and remains 0.
- $\phi(C_j)$ for $j > i$ is unchanged because the join results before executing C_j and the possible binding values for C_j are the same.
- $\phi(C_j)$ for $j < i$ cannot increase. If C_j does not use any binding attributes, then moving C_i before C_j would not increase $\phi(C_j)$. If C_j uses binding values from a bind join, then moving C_i before C_j would not increase (but may decrease) the number of bind join values for C_j, and that would not increase $\phi(C_j)$.

□

PayLess's optimizer applies Theorem 2 repeatedly and moves all zero price calls to the leftmost subtree of P. That way, the search space of PayLess's optimizer is further reduced.

Lastly, PayLess's optimizer would prune some candidate subplans during plan enumeration:

THEOREM 3. *When searching for the best plan for a set $\mathcal{C}$ of relations $C_1, C_2, \ldots, C_n$, if $\mathcal{C}$ can be partitioned into disjoint subsets $\mathcal{C}_1 \ldots \mathcal{C}_j$, where relations in $\mathcal{C}_i$ cannot join with relations in $\mathcal{C}_j$ (unless using Cartesian product $\times$). Then the best plan for $\mathcal{C}$ is $Best(\mathcal{C}_1) \times Best(\mathcal{C}_2) \times \ldots Best(\mathcal{C}_j)$, where $Best(\mathcal{C}_i)$ denotes the best plan for the set of relations in $\mathcal{C}_i$.*

PROOF. The proof is trivial because the relations in $\mathcal{C}_i$ cannot join with relations in $\mathcal{C}_j$, the price of calling $\mathcal{C}_j$ would not be influenced by $\mathcal{C}_i$. So, the best plan for $\mathcal{C}$ becomes simply connecting the best subplans of $\mathcal{C}_1 \ldots \mathcal{C}_j$ using Cartesian product. □

Consider a chain query that joins four relations: $\mathcal{C} = \{U(v, w), R(w, x), S(x, y), T(y, z)\}$. Assuming that the best plans determined for the pairs of relations are:

	$\{U, R\}$	$\{U, T\}$	$\{U, S\}$	$\{R, S\}$	$\{R, T\}$	$\{S, T\}$
Best Plan	$U \overrightarrow{\bowtie} R$	$U \times T$	$U \times S$	$R \overrightarrow{\bowtie} S$	$R \times T$	$S \overrightarrow{\bowtie} T$

So, when determining the best plan for 3-way join, the candidate plans that would be generated are:

	$\{U, R, S\}$	$\{U, R, T\}$	$\{U, S, T\}$	$\{R, S, T\}$
Candidate Plans	$(U \overrightarrow{\bowtie} R) \bowtie S$	$(U \overrightarrow{\bowtie} R) \bowtie T$	...	...
	$(U \overrightarrow{\bowtie} R) \overrightarrow{\bowtie} S$	$(U \times T) \bowtie R$	...	...
	$(U \times S) \overrightarrow{\bowtie} R$	$(U \times T) \overrightarrow{\bowtie} R$	...	...
	...	$(R \times T) \bowtie U$	...	...
	...	$(R \times T) \overrightarrow{\bowtie} U$	...	...

Observe that the set $\{U, R, T\}$ can be partitioned into two disjoint subsets: $\mathcal{C}_1 = \{U, R\}$ and $\mathcal{C}_2 = \{T\}$. So, we can apply Theorem 3 to determine the best plan for the set $\{U, R, T\}$ as $Best(U, R) \times T$, i.e., $(U \overrightarrow{\bowtie} R) \times T$. In other words, Theorem 3 eliminates many candidates (e.g., $(R \times T) \overrightarrow{\bowtie} U$) and eliminates their associated costing steps and semantic rewriting steps.

Let the total number of candidate plans in all levels of the dynamic programming approach be the size of the search space. For a chain query with n relations whose attributes are all free. The use of the above theorems can reduce the search space from $\approx 6^n - 5^n$ down to $\approx 2^{n'} + \frac{2}{3} \cdot n'^3$ with the optimal plan retained, where m is the number of zero price relations and $n' = n - m$. Specifically, the original plan space with dynamic programming is:

$$n + \sum_{k=2}^{n} \left(\binom{n}{k} \cdot \left(\sum_{i=1}^{k-1} \binom{k}{i} \cdot 4^{min\{i,k-i\}}\right) \right) \approx 6^n - 5^n$$

where k represents the level in dynamic programming (e.g., when $k = 2$, we consider joining two relations). At level k, there are $\binom{n}{k}$ size-k subsets to be examined. For each size k subset, we can form a plan by: (i) choosing a size i subset for the left subtree (and the complementary size $k - i$ subset for the right subtree), and (ii) deciding the binding attributes for the join (at root). For (ii), each call on the right subtree can bind with attributes from at most 2 calls from the left subtree; thus, there are 2·2=4 binding choices per call, and at most 4^{k-i} choices per plan. We can tighten this number to $4^{min\{i,k-i\}}$ when i is small and the left subtree can provide at most 4^i binding choices.

The plan space of PayLess's optimizer is:

$$4n' + \sum_{k=2}^{n'} \left(4 \cdot k \cdot (n' - k + 1) + \left(\binom{n'}{k} - (n' - k + 1)\right) \right)$$

$$\approx 2^{n'} + \frac{2}{3} \cdot n'^3$$

where m is the number of zero price relations and $n' = n - m$. Specifically by Theorem 2, we first build a plan with all local m relations. Then, in dynamic programming, we consider growing the plan by using the remaining $n' = n - m$ relations. At level k, there are $\binom{n'}{k}$ size-k subsets. We can divide them into (i) discon-

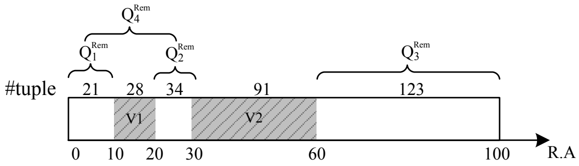


Figure 6: Generation of remainder queries

nected subsets (in which some relations must be joined by Cartesian product), and (ii) connected subsets. For the chain query, there are $n' - k + 1$ connected subsets and $\binom{n'}{k} - (n' - k + 1)$ disconnected subsets. For each disconnected subset, we can compute its best plan directly by Theorem 3. For each connected subset, we can obtain it by Theorem 1, i.e., combining a size-$(k-1)$ subset with a new call. There are k choices for the call and at most 2·2=4 binding choices for that call.

4.2 Semantic Query Rewriting

In PayLess, we store all RESTful queries issued to the data market and their corresponding results in the semantic store. The objective of doing so is to carry out *semantic query rewriting*, i.e., answer the queries using those stored results so as to reduce the amount of data retrieved from the data market. Semantic query rewriting falls into the category of rewriting queries using views [29, 48]. Given a query Q, a set $\mathcal{V}$ of RESTful queries and their corresponding stored results, the key step in semantic query rewriting is to compute the set $\mathcal{R}em(Q, \mathcal{V})$ of *remainder queries* [23]. The set $\mathcal{R}em(Q, \mathcal{V})$ essentially contains the set of RESTful queries that has to be sent to the data market in order to retrieve the tuples required by Q but not covered by $\mathcal{V}$.

Before we delve deeper, we first explain why optimizers for queries over remote data sources like [17, 27, 45] do not use semantic query rewriting. Consider our example query $Q1$ (page 1), which inquires about the daily temperature of Seattle in June 2014, has been issued, and its 30 resulting tuples (one tuple for each day in June) are stored in the semantic store. Assume that there is another query $Q2$ being issued, with $Q2$ shares the same query template like $Q1$ but the date ranges from May 2014 to July 2014 (3 months). Using semantic query rewriting, $Q2$ will generate two remainder queries: one asks for weather records in May (31 records; 1 transaction), another asks weather records in July (31 records; 1 transaction). The final result is then obtained by union the above with the stored results of $Q1$. The plan of using semantic query rewriting incurs a total of two calls to the external data source. In contrast, only one call to the external data source is required if $Q2$ is sent to the external data source without semantic query rewrite. So, in the context of minimizing the number of calls to external data sources, semantic query rewriting obviously is not a fruitful technique because it decomposes a call to several sub-calls.

Now, we show that how could we adapt semantic query rewriting to PayLess's optimizer to yield competitive plans for data market query processing. To illustrate, consider the example in Figure 6. The example assumes that the results of two queries V_1 and V_2 have been stored in the semantic store. Both V_1 and V_2 are range queries on an integer attribute A whose domain is $[0, 100]$. V_1 and V_2 respectively cover the ranges $[10, 20)$ and $[30, 60)$ on attribute A and have retrieved 28 and 91 tuples from table R. In what follows, we write a query Q in the form as

$$Q :- \ R_1(A[s, e], B = \beta, C), R_2(C, ..)$$

which means it joins R_1 and R_2 using C as the join attribute, and tuples in table R_1 have values in numeric attribute A fall between s and e and have values in categorical attribute B equal β.

Now, with V_1 and V_2, we assume the following query Q is posed:

$$Q :- \ R(A[0, 100])$$

Using the vanilla semantic query rewriting techniques, it will generate an invalid remainder query $Q^{Rem}_{invalid}$:

$$Q^{Rem}_{invalid} :- \ R(A[0, 10) \vee [20, 30) \vee [60, 100])$$

In data market, $Q^{Rem}_{invalid}$ is invalid because it involves disjunction, which is not supported by the access pattern of data market. Therefore, our first step to adapt semantic query rewriting techniques is to decompose remainder queries that violate the data source access patterns into a set of valid remainder (sub)queries. For the example above, PayLess will generate a set $\mathcal{R}em_1$ of remainder queries:

$Q^{Rem}_1 :- \ R(A[0, 10))$ //21 tuples; 1 transaction
$Q^{Rem}_2 :- \ R(A[20, 30))$ //34 tuples; 1 transaction
$Q^{Rem}_3 :- \ R(A[60, 100])$ //123 tuples; 2 transactions

So, altogether, $\mathcal{R}em_1$ will cost a total 4 transactions.

Note that such straightforward decomposition may not yield the best plan. For example, the following is another possible set of remainder queries $\mathcal{R}em_2$:

$Q^{Rem}_4 :- \ R(A[0, 30))$ //21+28+34= 83 tuples; 1 transaction
$Q^{Rem}_3 :- \ R(A[60, 100])$//123 tuples; 2 transactions

The remainder query Q^{Rem}_4, although overlaps with stored query V_1, will still cost $\lceil(21 + 28 + 34)/100\rceil = 1$ transaction. So, altogether, $\mathcal{R}em_2$ will cost a total 3 transactions only.

The example above illustrates a new and unique issue specific to the generation of remainder queries in data market. Specifically, we see that there are alternate ways to generate valid remainder queries and it is possible that a lower overall price can be achieved even when *a remainder query overlaps with a stored query*.

PayLess obviously does not want to miss the above opportunity when optimizing the queries. So, we have devised a remainder query generation method that leverages the above opportunity to reduce the overall price to access the data market.

We illustrate our idea using a more general example in Figure 7a. In the example, the query Q is a 2d-query that inquires table R:

$$Q :- \ R(A_1[30, 80], A_2[0, 50])$$

In the example, we assume there are ten RESTful queries $V_1, \ldots, V_{10}$ stored in the semantic store. Figure 7b shows the intersection of Q and the complement of $\mathcal{V}$, i.e., the data supposed to be retrieved from the data market. Denoting that space as $\overline{\mathcal{V}}$, there are alternate sets of remainder queries that can retrieve all the missing data. For example, consider the following set of remainder queries $\mathcal{R}em_3$:

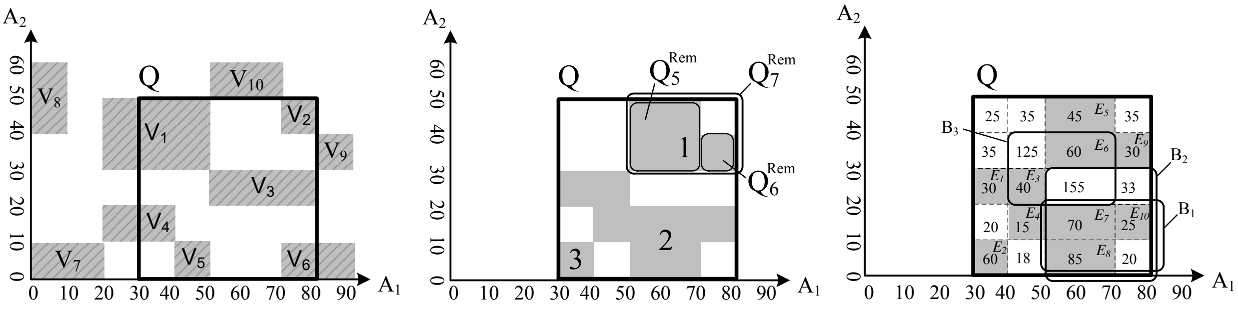
$Q^{Rem}_5 :- \ R(A_1[50, 70), A_2[30, 50])$
$Q^{Rem}_6 :- \ R(A_1[70, 80], A_2[30, 40])$

$\mathcal{R}em_3$ covers the missing data in region 1. Alternately, the following set of remainder queries $\mathcal{R}em_4$ can also cover data in region 1:

$Q^{Rem}_7 :- \ R(A_1[50, 80), A_2[30, 50])$

From the above, we see that our goal boils down to finding a set of bounding boxes that cover all the data regions in $\overline{\mathcal{V}}$ using the least number of data market transactions.

To achieve a good solution, we use a two-step approach. The first step aims to generate a set of promising bounding box $\mathcal{B}$ candidates that cover different data regions in $\overline{\mathcal{V}}$. The bounding box candidates



(a) Query Q and Stored Results (b) $\overline{\mathcal{V}}$ (c) Bounding Boxes in $\overline{\mathcal{V}}$

Figure 7: Generation of remainder queries for data market

may possibly overlap with each other. The second step aims to extract from $\mathcal{B}$ the best set of bounding boxes that cover all the data regions in $\overline{\mathcal{V}}$ in minimum price.

We now elaborate the first step. Specifically, we begin with a decomposition of $\overline{\mathcal{V}}$ into a union $\mathcal{E}$ of disjoint *elementary boxes*. Figure 7c shows an example. On each dimension i, we collect a *separator* set S_i from the corners of each elementary box. For example, elementary box E_8 contributes values 50 and 70 to S_1 and contributes values 0 and 10 to S_2. Accounting for all elementary boxes, then we have $S_1 = \{30, 40, 50, 70, 80\}$ and $S_2 = \{0, 10, 20, 30, 40, 50\}$. Then, we exhaustively construct a set $\mathcal{B}$ of *bounding boxes*, where the extent of a bounding box $B \in \mathcal{B}$ on dimension i is picked from any two values in S_i. For example, the bounding box B_1 in Figure 7c has extent $[50, 80]$ on dimension A_1 and extent $[0, 20]$ on dimension A_2 when it picks values 50 and 80 from S_1 and values 0 and 20 from S_2. Each resulting bounding box represents a remainder query that covers certain data to be retrieved from the data market.

Algorithm 1 presents the pseudo-code of generating the bounding boxes, with powerful pruning rules to prune unpromising bounding boxes. First, it estimates the number of tuples falling into each elementary box in $\mathcal{E}$ from ISOMER (Lines 2–3). Figure 7c shows an illustration with those estimates. (We will discuss the case of insufficient/inaccurate statistics in the Section 4.3). Next, it enumerates a set of bounding boxes from the separator sets $S_1, S_2, ...S_d$, where d is the dimensionality of the query. It applies two pruning rules to discard unpromising bounding boxes.

The first pruning rule (Line 6) prunes a bounding box B if it is not tight. In other words, only *minimum bounding boxes* could stay. Consider the bounding boxes B_1 and B_2 in Figure 7c. They both contain the same set of elementary boxes E_7, E_8, E_{10} but B_2 contains B_1. Therefore, B_2 is not a minimum bounding box and is pruned. This makes sense because B_2 has to download an extra $155 + 33$ redundant tuples comparing with B_1.

The second pruning rule (Line 8) prunes a bounding box if its price is not smaller than the price sum of its individual elementary boxes. Consider bounding box B_3 Figure 7c. It requires $\lceil(125 + 60 + 40 + 155)/100\rceil = 4$ transactions. However, if E_3 and E_6 are individually retrieved, they collectively cost only $\lceil 40/100\rceil + \lceil 60/100\rceil = 2$ transactions. So, in this case, B_3 is not helpful and is pruned as well.

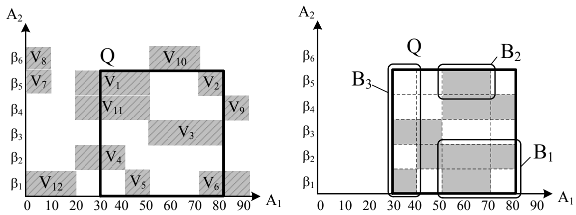
Algorithm 1 would enumerate $\binom{|S_i|}{2}^d$ bounding boxes for a d-dimensional query in the worst case. However, because of the high effectiveness of the pruning rules, the number of (minimum) bounding boxes considered is indeed much fewer than the worst case in practice.

The second step of our idea is to find the best subset of minimum bounding boxes (generated from Algorithm 1) that cover all

Algorithm 1 Generating Candidate Remainder Queries

Input (elementary boxes $\mathcal{E}$, separator sets $\{S_1, S_2, ..., S_n\}$)
Output (A collection of minimum bounding boxes $\mathcal{B}$)

```
1:  initialize B
2:  for each elementary box E_i in E do
3:      E_i.price ← estimate the price of E_i
4:  enumerate every possible bounding box B using the separator
    sets S_1, S_2, ..., S_n.
5:  for each bounding box B do
6:      if B is a minimum bounding box then        ▷ pruning rule 1
7:          estimate the price of B
8:          if B.price < Σ_{E_i∈B} E_i.price then   ▷ pruning rule 2
9:              insert B into B
10: return B
```



(a) Query Q and Stored Results (b) Bounding Boxes in $\overline{\mathcal{V}}$

Figure 8: Generation of remainder queries with a categorical attribute A_2

the elementary boxes (all missing data) in minimum price. This is a weighted set cover problem [22]. Specifically, the weighted set cover problem states that, given (1) a set of elements $\mathcal{E} = \{E_1, E_2, ...\}$ and (2) a family $\mathcal{B}$ of subsets of $\mathcal{E}$, in which each subset in $\mathcal{B}$ is associated with a $cost_i$, find a collection of subsets, namely the cover, $Cover \subseteq \mathcal{B}$, whose union of the elements in $Cover$ is $\mathcal{E}$ and the sum of cost of elements in $Cover$ is the minimum. In our context, we have (1) $\mathcal{E}$ as all elementary boxes and (2) $\mathcal{B}$ as the set of candidate minimum bounding boxes returned by Algorithm 1, $cost_i$ is referred as a bounding box 's estimated transactions. To solve this $\mathcal{NP}$-hard problem, we use the greedy algorithm in [22] that runs in $O(|\mathcal{B}| \cdot |\mathcal{E}|)$ time with $(1 + \ln(|\mathcal{B}|))$ approximation ratio.

The generation of bounding boxes for queries with *categorical attributes* is illustrated as follows. Figure 8a shows an example similar to the previous one but with attribute A_2 now becomes a categorical attribute with the following domain: $\{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6\}$. We remark that there are no stored queries that can span across multiple categorical values because of the limitation of the access interface.

Figure 8b shows the corresponding space $\overline{\mathcal{V}}$. Since A_2 is a categorical attribute, the bounding box B_1, which represents the following remainder query, is invalid:

$$:-\ R(A_1[50,80), A_2 = \beta_1 \vee A_2 = \beta_2)$$

Therefore, we will only generate bounding boxes that span either one value or the whole domain of a categorical attribute. For example, bounding boxes B_2, which represents the following remainder query, is valid and would be generated:

$$:-\ R(A_1[50,70), A_2 = \beta_5)$$

Similarly, bounding boxes B_3, which represents the following remainder query, is also valid and would be generated:

$$:-\ R(A_1[30,40))$$

The generation of bounding boxes for queries with *bind joins* is illustrated as follows. Consider a relation U with binding pattern $U(A_1^f, A_2^f)$ and a relation S with binding pattern $S(A_2^b, A_3^f)$, where all attributes are integer attributes. Further, consider a query V that joins U and S:

$$V :-\ U(A_1[2,3], A_2), S(A_2, A_3[10,15])$$

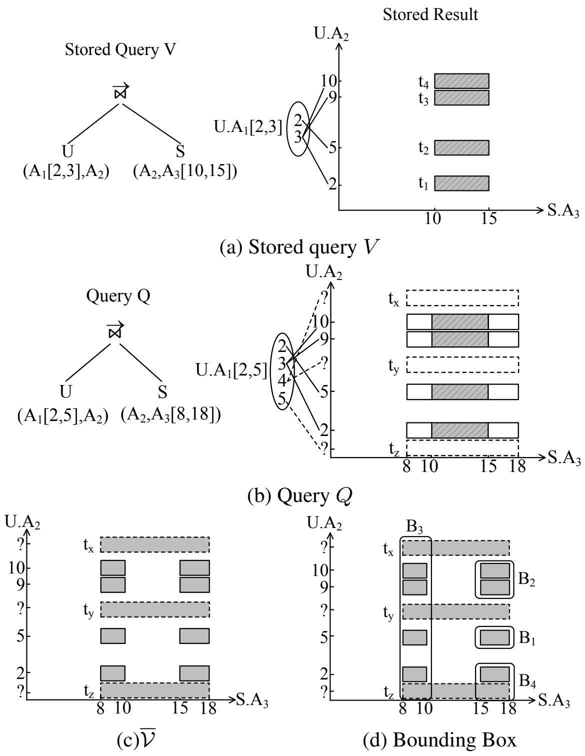
V needs a bind join because A_2 is a bind attribute. So, assume that there are four tuples t_1, t_2, t_3, and t_4 in U having values within the range $[2,3]$ in attribute A_1 and their corresponding values in attribute A_2 are 2, 5, 9, and 10, respectively. Then, the bind join is carried out with S by binding the values 2, 5, 9, and 10 to S's attribute A_2. Note that when retrieving tuples from S whose attribute A_2 has a value, say, 2, those tuples have to satisfy the other condition $A_3[10,15]$ as well. Figure 9a illustrates the above process.

Now, assume the query results of V are stored in the semantic store and let us consider a query Q that shares the same query template as V but with a different query range:

$$Q :-\ U(A_1[2,5], A_2), S(A_2, A_3[8,18])$$

Note that in this case, assuming that we can estimate that two tuples t_x and t_y will be retrieved from U for $A_1 = 4$, one tuple t_z will be retrieved from U for $A_1 = 5$ (we don't need to estimate the cardinality for $A_1 = 2$ and $A_1 = 3$ because we know the exact cardinality from V), exact values of t_x, t_y, t_z's attribute A_2 are still unknown (denoted as ? in Figure 9b). In this case, it will generate $\overline{\mathcal{V}}$ like Figure 9c. Consequently, when enumerating the set of candidate bounding boxes, we can generate a bounding box for each individual elementary box (e.g., B_1), for a range of known values (e.g., B_2), or for the whole domain (e.g., B_3). In contrast, we cannot generate a bounding box like B_4 because the exact value for A_2 of t_z is actually unknown.

Algorithm 2 shows the pseudo code of PayLess optimization. It is self-explanatory and mainly summarizes what we have discussed above, so we do not give it a walkthrough here.



(a) Stored query V

(b) Query Q

(c) $\overline{\mathcal{V}}$

(d) Bounding Box

Figure 9: Example for 2D-Bind

Algorithm 2 PayLess Query Optimization

Input (a query Q, a set $\mathcal{V}$ of RESTful queries and their stored results, the metadata $\mathcal{M}$ for cost estimation)
Output (the optimal plan $P^* : Best(Q)$ for the query Q)

1: $\mathcal{R}_{local} \leftarrow \{C_i \in Q : \phi(C_i) = 0\}$; $\mathcal{R}' \leftarrow \{C_i \in Q\} - \mathcal{R}_{local}$
2: $P_{local} \leftarrow$ the best subplan for $\mathcal{R}_{local}$; found by offloading to a DBMS's optimizer
3: **for** each $C_i \in Q$ **do** ▷ size-1 subplans
4: $\quad Best(C_i) \leftarrow$ **`SemanticRewrite`**$(C_i, \mathcal{V}, \mathcal{M})$
5: execute Line 1 again to update $\mathcal{R}_{local}$ and $\mathcal{R}'$
6: **for** each k from 2 to $|\mathcal{R}'|$ **do** ▷ Theorem 2
7: $\quad$ **for** each size-k subset $\mathcal{R}^k$ of $\mathcal{R}'$ **do**
8: $\quad\quad$ **if** $\mathcal{R}_{local} \cup \mathcal{R}^k$ form ℓ disjoint subsets **then** ▷ Theorem 3
9: $\quad\quad\quad Best(\mathcal{R}^k) \leftarrow Best(\mathcal{R}_1^k) \times Best(\mathcal{R}_2^k) \times \cdots \times Best(\mathcal{R}_\ell^k)$
10: $\quad\quad$ **else for** each call $C_i \in \mathcal{R}^k$ ▷ Theorem 1
11: $\quad\quad\quad$ rewrite C_i as $\overrightarrow{C_i}$ by using binding from $P_{local} \bowtie Best(\mathcal{R}^k - C_i)$
12: $\quad\quad\quad P_{bind} \leftarrow$ **`SemanticRewrite`**$(\overrightarrow{C_i}, \mathcal{V}, \mathcal{M})$
13: $\quad\quad\quad P_{temp} \leftarrow Best(\mathcal{R}^k - C_i) \bowtie Best(C_i)$
14: $\quad\quad\quad$ **if** $\phi(P_{bind}) \leq \phi(Best(C_i))$ **then**
15: $\quad\quad\quad\quad P_{temp} \leftarrow Best(\mathcal{R}^k - C_i) \overrightarrow{\bowtie} P_{bind}$
16: $\quad\quad\quad$ update $Best(\mathcal{R}^k) \leftarrow P_{temp}$ if $\phi(Best(\mathcal{R}^k)) \geq \phi(P_{temp})$

4.3 Discussion

We end this section with a number of discussions about our query optimization approach. First, as in traditional cost-based query optimization, our approach relies on metadata like histograms. In the beginning when no rich statistics such as value distributions are available, PayLess's optimizer would carry out the cardinality estimation using the basic textbook methods (e.g., using the domain size and uniform distribution assumption).

Second, answering a query using the stored query results may include obsolete tuples if datasets permit *in-place* data update. However, so far the datasets we found in Windows Azure Marketplace are append-only. In case in-place data update exists, we will introduce several *consistency levels* into PayLess. That would allow organizations that install PayLess to choose between consistency levels like *(i) weak consistency*, *(ii) X-week consistency*, or *(iii) full consistency*. Weak consistency means all RESTful queries and their results are stored in the semantic store (with obsolete results get updated if new results are retrieved). Under weak consistency, semantic query rewriting is always enabled. Queries may however return partially obsolete results when there are in-place updates in the data market's datasets because it reuses some obsolete stored results. Strong consistency means semantic query writing is simply disabled and PayLess always go to the data market to obtain the latest results. X-week consistency is in the middle, it enables semantic query rewriting using query results retrieved from the past X weeks. The three options are trade-off between price-to-pay and the freshness of the result.

5. EVALUATION

PayLess aims to help organizations to pay less when their end users have to query against the data market. Without PayLess, one option is to employ query optimizers for data sources with limited access pattern because those optimizers at least consider binding patterns and bind joins in their architecture. Another option is to download all required tables from the data market upfront and carry out local processing afterwards. Notice this "Download All" option is not always bad. First, it is optimal if the queries have to scan the whole dataset. In this case, once the whole dataset is downloaded, all queries can work on the downloaded data locally. Second, if the number of transactions incurred by user queries would eventually exceed the number of transactions required to download the complete data set, then downloading the whole dataset upfront would be a more economical option. However, we re-iterate that it is always tough to predict how many user queries would eventually be issued in practice. Consider that the users walk away from the dataset forever after issuing just a few queries (maybe due to no interesting information is found), then downloading the whole dataset would become a very costly option.

In this section, we evaluate the effectiveness of PayLess using both real data and synthetic data. Specifically, we extract query templates from a meteorological application that involves queries to the Worldwide Historical Weather (WHW) [13] and Environmental Hazard Rank (EHR) [6] datasets in Windows Azure Marketplace. Table 1 lists the query templates and Figure 1a lists the sizes of the tables. We generate *valid* query instances from those templates by randomly assigning values to the parameters. A query instance is valid if it returns non-empty results (e.g., we would not instantiate Q_4 with a country equals 'USA' but a zip code in Germany). We also use the TPC-H workload in the experiments. We generate 1G of TPC-H data and 1G of TPC-H skew data [19] with $zipf = 1$. All parametric attributes in TPC-H queries are set as free attributes in the experiments. We set the relations `Nation` and `Region` local. By default, we set 100 tuples as one transaction (i.e., $t = 100$).

Overall effectiveness. We first study the overall effectiveness of PayLess under different workloads and datasets. For comparison, we include the results of using [27] to optimize the queries (denoted as "Minimizing Calls" in the figure). We also include the results of disabling semantic query rewriting (SQR) in PayLess (denoted as "PayLess w/o SQR" in the figure). We respectively generated q query instances per template. The query instances are issued in a random order and the results are reported as an average over 30 repeated experiments. In this experiment, we set $q = 10$ and $q = 200$ for TPC-H workload and real workload, respectively.

Figure 10a illustrates the total (cumulative) number of data market transactions used to answer the real queries. Except the "Download All" option, when more queries are issued, the total (cumulative) number of data market transactions increases. Comparing with those data buyers who recklessly download the whole dataset upfront, PayLess can now help them to answer the queries using about two orders of transactions fewer. The number of transactions used by PayLess grows slowly because many queries are rewritten using the stored results in the semantic cache. PayLess can answer the queries using about an order of transactions fewer than queries optimized using [27]. That is because semantic query rewriting (SQR) is not applicable to their setting but is a powerful helper here in our data market setting. When we disable SQR, PayLess still outperforms [27]. That is because PayLess can find optimal plans in a reduced search space using progressively refined statistics. In contrast, [27] has to find plans in a larger search space (including bushy trees) using heuristics.

Figures 10b and c show the results of using TPC-H workload. TPC-H queries scan a large portion of data. Therefore, without rewriting the queries using the stored data, each query optimized by [27] and PayLess (if SQR is disabled) would retrieve a large portion of the data from the data market, and those data are largely overlapping with each other. That explains why they are worse than "Download All", because the latter only downloads the whole dataset once. When PayLess is in full power with semantic query rewriting, we see that the subsequent queries can largely reuse the stored results, thereby saving a lot more transactions than "Download All" until about 80 queries have been issued. When about 80 queries have been issued, all the data required by TPC-H queries (indeed the whole TPC-H dataset) are stored by PayLess, therefore PayLess would not repeatedly retrieve the data from the data market anymore. From the above experimental results, we regard PayLess to be practically better than "Download All" in all means because nobody could have known the number of queries to be issued and the distribution of the data in practice. A data buyer can freely query against any dataset in the data market and walk away from that dataset anytime — she does not need to worry whether it is worth or not to download the whole dataset in the beginning, or switch to download the whole dataset when she finds out that she has to ask more queries after she has burned a certain amount of money.

Influence of number of tuples per transaction. We next study whether the effectiveness of PayLess would be influenced by the number of tuples per transaction, which could be a different value in different data markets. Since [27] is consistently outperformed by PayLess in all our experiments, so we remove it, together with PayLess with semantic query rewriting disabled, from our discussion.

Figure 11 shows the effectiveness of PayLess when we vary the number of tuples per transaction t. Note that when t is smaller, more transactions are required to retrieve the same number of tuples from the data market. Therefore, the number of transactions used by both PayLess and "Download All" must increase. Nevertheless, we see that the effectiveness of PayLess is not influenced by that data market parameter. PayLess still outperforms "Download All" under real data in all cases. In addition, it still outperforms "Download All" on TPC-H and TPC-H skew data until the whole dataset is retrieved.

Influence of number of query instances per query template. We next study whether the effectiveness of PayLess would be influenced by q, the number of query instances per query template. Figure 12 shows that the effectiveness of PayLess is not influenced by that parameter. We see that PayLess still consistently outperforms "Download All" on real data in all cases. In addition, it still outper-

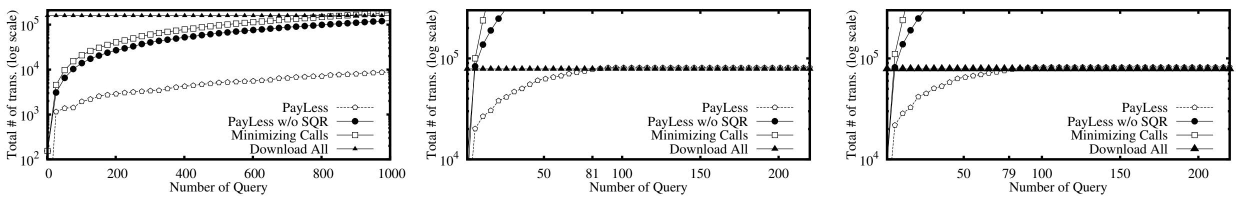
Table 1: Query Templates on Real Data Sets

Q_1	`SELECT * FROM Weather` `WHERE Weather.Country = ? AND Weather.Date >= ? AND Weather.Date <= ?`
Q_2	`SELECT COUNT(ZipCode) FROM Pollution` `WHERE Pollution.Rank >= ? AND Pollution.Rank <= ?`

Q_3	`SELECT AVG(Temperature) FROM Station, Weather`
	`WHERE Station.Country = Weather.Country = ? AND Weather.Date >= ? AND`
	`Weather.Date <= ? AND Station.StationID = Weather.StationID`
	`GROUP BY City`

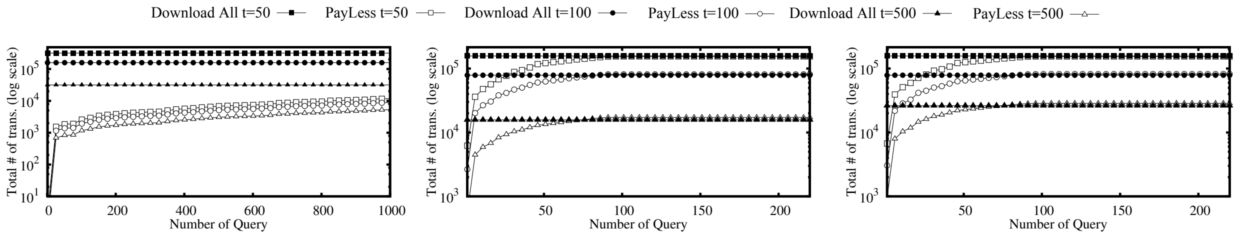
Q_4	`SELECT Temperature FROM Station, Weather, ZipMap`
	`WHERE Station.Country = Weather.Country = ? AND ZipMap.ZipCode = ? AND`
	`Weather.Date >= ? AND Weather.Date <= ? AND`
	`Station.StationID = Weather.StationID AND Station.City = ZipMap.City`

Q_5	`SELECT * FROM Pollution, Station, Weather, ZipMap`
	`WHERE Station.Country = Weather.Country = ? AND Weather.Date >= ? AND`
	`Weather.Date <= ? AND Pollution.Rank >= ? AND Pollution.Rank <= ? AND`
	`Pollution.ZipCode = ZipMap.ZipCode AND ZipMap.City = Station.City AND`
	`Station.StationID = Weather.StationID`



(a) Real data (b) TPC-H (c) TPC-H skew

Figure 10: Overall Effectiveness



(a) Real data (b) TPC-H (c) TPC-H skew

Figure 11: Varying the number of results t per transaction

forms “Download All" on TPC-H and TPC-H skew data until the whole dataset is retrieved.

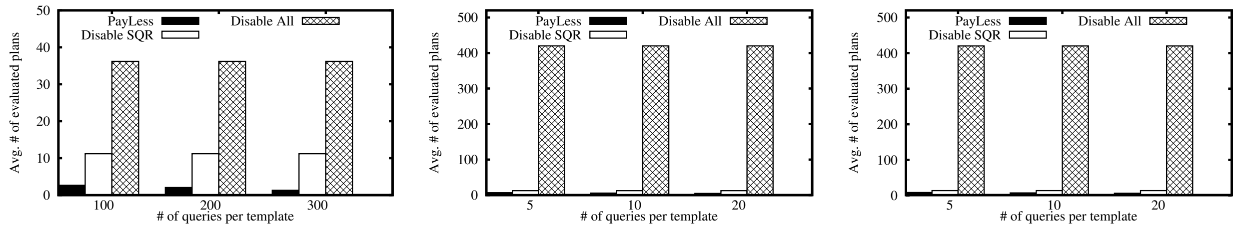
Influence of data size. We also study whether the effectiveness of PayLess would be influenced when the size of the data is varied. As we cannot control the size of the real data, we control only the size of the synthetic data.

Note that when the data size increases, “Download all” needs more transactions to download the whole dataset. But PayLess also needs to retrieve more tuples for each query. Figure 13 shows that PayLess still outperforms “Download All" on TPC-H and TPC-H skew data until the whole dataset is retrieved.

Effectiveness of search space reduction techniques. We have also carried out an experiment to evaluate the effectiveness of our techniques devoted to reducing the search space size. Figure 14 shows the average number of candidate (sub)plans for all query instances under our default setting. We report the case when (i) SQR is disabled (Disable SQR), (ii) both SQR and search space pruning (Theorems 1 to 3) are disabled (Disable All), and (iii) nothing is disabled (PayLess). We can see that our techniques significantly reduce the search space by orders of magnitude. This is actually what enables us to look for optimal plans. We notice that enabling SQR indeed reduces the search plan because SQR would cause some relations become local, which can then trigger Theorem 2. This also explains why the average number of candidate (sub)plans PayLess has to considered decreases when we increase the number of query instances generated for each template. That is because if we increase the number of query instances generated for each template, that would retrieve more data from the data market, which in turn increases the chance of using Theorem 2 to reduce the search space.

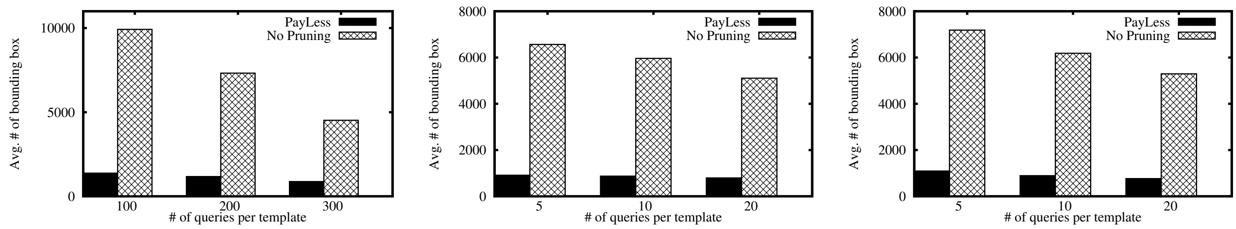
Effectiveness of bounding box pruning. Our last experiment is to evaluate the effectiveness of the bounding box pruning rules in Algorithm 1. Figure 15 shows the average number of bounding boxes generated for all query instances under our default setting. We see that the two pruning rules can reduce about an order bounding boxes generated.

Efficiency. In all experiments, the execution time of a query is, as usual, dominated by the RESTful calls to the data seller. Nevertheless, a query can still finish within seconds. The query optimization and the query execution part done by PayLess on the data buyer side all finish within milliseconds. We omit the detailed numbers here for space reasons.



(a) Real data (b) TPC-H data (c) TPC-H skew data

Figure 14: Effectiveness of search space reduction techniques



(a) Real data (b) TPC-H data (c) TPC-H skew data

Figure 15: Effectiveness of bounding box pruning rules

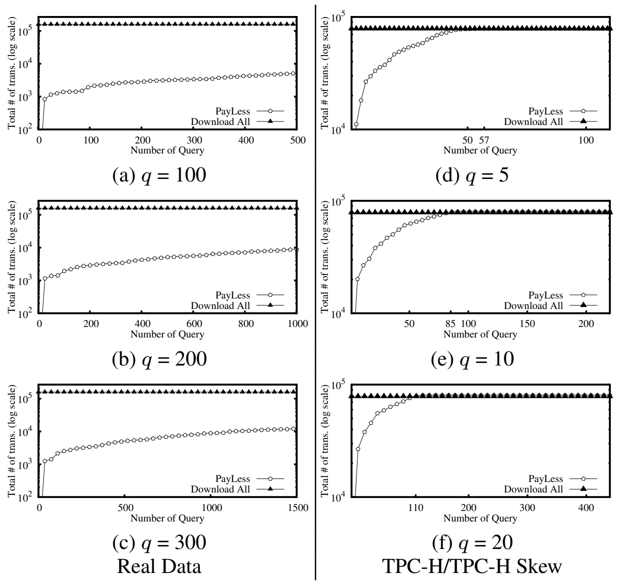
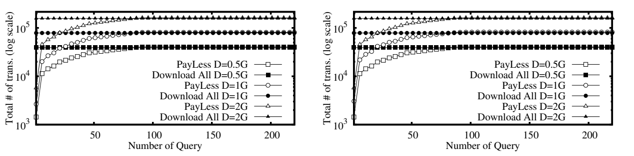


Figure 12: Varying the number of query instances (q) per template



(a) TPC-H data (b) TPC-H skew data

Figure 13: Varying data size

6. RELATED WORK

To the best of our knowledge, this paper is the first to tackle the issue of optimizing queries that access the data market. So far, projects related to the data market are mainly developed for *query market*. In their setting, the query market can support SQL. A data buyer sends a SQL query that accesses a dataset in the query market. The query market computes the results of the query and returns the answer to the buyer. The research focus is how to set the price of arbitrary SQL queries (e.g. [15,16,30,31,37,39,47]). The setting of query market is different from our data market setting. Specifically, existing data market like Windows Azure Marketplace [1] and Xignite [14] are still charging data buyers according to the size of retrieved data.

In terms of problem setting, PayLess is indeed more similar to projects that support queries over remote data sources with limited access patterns (e.g., [17, 20, 24, 27, 33–36, 40, 45]). Nevertheless, as mentioned, all these projects have a very different focus with us — they are designed to minimize the number of calls to external data and/or the execution time. In contrast, PayLess focuses on minimizing the amount of intermediate retrieved data measured in terms of data market transactions. Besides, the optimization of distributed queries with semi-join/magic sets [18, 43] are similar to PayLess; however, they do not consider limited access patterns.

In terms of implementation, PayLess has borrowed the idea of learning optimizer from LEO [46] and has used feedback driven histogram ISOMER [44]. However, PayLess has to develop its own architecture, construct its own plan search space, and devise its own semantic query rewriting technique (e.g., [21,23,32,41]) to fit the data market. In computational geometry, the problem of partitioning an orthogonal polygon into rectangles (PiR) [26] is similar to our remainder query generation problem, but they are not the same. Using Figure 7b as an example, the PiR problem would NOT consider Q_7^{Rem}, which contains some empty regions. In contrast, in our context, Q_7^{Rem} could be a good choice according to our cost function.

7. CONCLUSION

This paper presents PayLess, a system that helps data buyers to freely query against any dataset in the data market and walk away from that dataset anytime. The data buyers do not need to worry whether it is worth or not to download the whole dataset in the beginning. They can simply issue their queries to PayLess and PayLess optimizes their queries with the objective of minimizing their money-to-pay-to-data-sellers. Currently, our use-case does not cover many end users using PayLess simultaneously. When it does, we will incorporate multi-query optimization in PayLess if users are willing to defer theirs to become a batch.

Acknowledgments

The research is partly supported by grant GRF 152201/14E from Hong Kong RGC. The research is partly supported by the Research Grants Council of Hong Kong (GRF PolyU 520413 and 521012) and a research gift from Microsoft Hong Kong.

8. REFERENCES

[1] Azure data marketplace. https://datamarket.azure.com.

[2] Data markets compared. http://radar.oreilly.com/2012/03/data-markets-survey.html.

[3] data.gov. https://www.data.gov/.

[4] Datamarket. https://datamarket.com/.

[5] Discussing data markets in new york city. http://cloudofdata.com/2013/02/discussing-data-markets-in-new-york-city/.

[6] Environmental hazard rank data. http://datamarket.azure.com/dataset/edr/environmentalhazardrank.

[7] Esri. http://www.esri.com/.

[8] Factual. http://www.factual.com/.

[9] Infochimps. http://www.infochimps.com.

[10] Is infochimps running from the data market business? http://cloudofdata.com/2013/02/is-infochimps-running-from-the-data-market-business/.

[11] Wolfram alpha. http://www.wolframalpha.com/.

[12] World bank. http://www.worldbank.org/.

[13] Worldwide historical weather data. https://datamarket.azure.com//dataset//weathertrends//worldwidehistoricalweatherdata.

[14] Xignite data market. http://www.xignite.com.

[15] S. Al-Kiswany, H. Hacigümüs, Z. Liu, and J. Sankaranarayanan. Cost exploration of data sharings in the cloud. In *EDBT*, 2013.

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

[17] A. Calì and D. Martinenghi. Querying data under access limitations. In *ICDE*, 2008.

[18] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, 1998.

[19] S. Chaudhuri and V. Narasayya. Program for tpc-d data generation with skew. 2012.

[20] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *VLDB*, 1993.

[21] B. Chidlovskii and U. M. Borghoff. Semantic caching of web queries. *VLDB J.*, 2000.

[22] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):pp. 233–235, 1979.

[23] S. Dar, M. J. Franklin, B. Þ. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[24] O. M. Duschka and A. Y. Levy. Recursive plans for information gathering. In *IJCAI (1)*, 1997.

[25] A. El-Helw, I. F. Ilyas, W. Lau, V. Markl, and C. Zuzarte. Collecting and maintaining just-in-time statistics. In *ICDE*, 2007.

[26] D. Eppstein. Graph-theoretic solutions to computational geometry problems. *CoRR*, 2009.

[27] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, 1999.

[28] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[29] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 2001.

[30] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-based data pricing. In *PODS*, 2012.

[31] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Toward practical query pricing with querymarket. In *SIGMOD*, 2013.

[32] D. Lee and W. W. Chu. Towards intelligent semantic caching for web sources. *J. Intell. Inf. Syst.*, 2001.

[33] C. Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB J.*, 2003.

[34] C. Li and E. Y. Chang. Query planning with limited source capabilities. In *ICDE*, 2000.

[35] C. Li and E. Y. Chang. Answering queries with useful bindings. *ACM Trans. Database Syst.*, 2001.

[36] C. Li and E. Y. Chang. On answering queries in the presence of limited access patterns. In *ICDT*, 2001.

[37] Z. Liu and H. Hacigümüs. Online optimization and fair costing for dynamic data sharing in a cloud data market. In *SIGMOD*, 2014.

[38] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[39] A. Muschalle, F. Stahl, A. Löser, and G. Vossen. Pricing approaches for data markets. In *BIRTE*, 2012.

[40] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[41] Q. Ren, M. H. Dunham, and V. Kumar. Semantic caching and query processing. *IEEE Trans. Knowl. Data Eng.*, 2003.

[42] F. Schomm, F. Stahl, and G. Vossen. Marketplaces for data: an initial survey. *SIGMOD Record*, 2013.

[43] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*, 1996.

[44] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. Isomer: Consistent histogram construction using query feedback. In *ICDE*, 2006.

[45] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB*, 2006.

[46] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *VLDB*, 2001.

[47] P. Upadhyaya, M. Balazinska, and D. Suciu. How to price shared optimizations in the cloud. *PVLDB*, 2012.

[48] H. Z. Yang and P.-Å. Larson. Query transformation for psj-queries. In *VLDB*, 1987.