Online Optimization and Fair Costing for Dynamic Data Sharing in a Cloud Data Market

Ziyang Liu
NEC Laboratories America
Cupertino CA, USA
ziyang@nec-labs.com

Hakan Hacıgümüş
NEC Laboratories America
Cupertino CA, USA
hakan@nec-labs.com

ABSTRACT

The data markets are emerging to address many organizations' need to find more useful external data sets for deeper insights. Existing data markets have two main limitations. First, they either sell a whole data set or some fixed views, but do not allow arbitrary ad-hoc queries. Second, they only sell static data, but not data that are frequently updated. While there exist proposed solutions for selling ad-hoc queries, it is an open question what mechanism should be used to sell ad-hoc queries on dynamic data. In this paper, we study a data market framework that enables the sale or sharing of dynamic data, where each sharing is specified by an ad-hoc query. To keep the shared data up-to-date, the service provider should create a view of the shared data and maintain the view for the data buyer. We provide solutions to two important problems in this framework: how to efficiently maintain the views, and how to fairly determine the cost each sharing incurs for its view to be created and maintained by the service provider. Both problems are challenging since in the first problem, different sharings with the same operations in their plans may reuse these operations, and each sharing plan must be generated online. In the second problem, there are several factors that impact the fairness of a costing function and a straightforward mechanism won't work. We propose an intuitive online algorithm for sharing plan selection, as well as a set of fair costing criteria and an algorithm that maximizes the fairness.

1. INTRODUCTION

In the big data era, data has become an integral part of decision making and user experience enhancement. An important observation is that organizations not only use internal data but also find compelling ways of integrating external data (such as publicly available data sets, surveys, curated data from other organizations, etc.) into their decision making and planning processes. As a result, several data markets have emerged, where the data can be sold and bought (e.g., Microsoft Azure Marketplace [3], Infochimps [2], Xignite [5], Gnip [1], etc,), or in some cases data are freely shared with the public in the cloud [4].[1]

[1]Since data can be shared freely or at a price, which could also be called a sale, we use *sharing* and *sale* interchangeably.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2593679.

These recently emerged data markets are limited in functionality in two aspects. First, they either sell a whole data set or some fixed views of a data set, but do not allow arbitrary ad-hoc queries. This limitation leads to buyers needing to browse a large set of pre-defined views and possibly buying more data than what they need [19, 20]. Second, current data markets only sell static data sets[2], e.g., the GDP of a country from 1997 to 2011. This limits the sale of many useful data sets that receive frequent updates. For example, a food retailer may be interested in purchasing users' check-ins at restaurants, tweets, etc., in order to infer a user's food preference and recommend corresponding products; a hotel booking service may be interested in purchasing users' flight booking data and calendar data in order to recommend hotels and design targeted promotions; a deal service may find helpful to purchase users' location data in order to alert the users of good deals near them. The data to be purchased in all these scenarios are dynamic and frequently updated. There are proposed solutions for the first limitation [19, 20], where the authors developed a mechanism that automatically prices ad-hoc queries, which is proved to achieve certain desirable properties such as arbitrage-free.

In this paper, we focus on the second limitation, and propose *a way to sell dynamic data in a data market*. The data market has three roles: *data owner*, *data buyer*, and *data market service provider*. An individual or an organization may be both an owner and a buyer. The data owner is willing to sell/share the data at a price. Although the data owner may choose to sell the data directly to the data buyer, this direct sell would require significant amount of automation, as well as infrastructure efforts. Hence, data sellers typically prefer to go through the data market and leverage the services it offers, which is a common practice in cloud computing. As the data owners benefit from the services provided by the data market, the provider also benefits from serving a multitude of data owners and data buyers by consolidating them to achieve economies of scale.

When a buyer specifies the data sets she's willing to buy, the service provider has two tasks: (1) deliver and maintain the data in a way that minimizes the operational cost (analogous to finding a query plan with minimum cost), and (2) calculate the price of the data, which should be a function of the monetary value of the data specified by the owner, and the operational cost. In this paper we discuss how to deal with these two problems. For problem (2), we only focus on calculating the operational cost. The monetary value of the data is assumed to be determined by the data owner. The function that maps the cost to the price is an economics problem that is out of scope of this work and interested readers can refer to [23, 35] for pricing strategies.

[2]Certain data sets in the markets may be updated, however, the buyers need to buy the updated data set again.

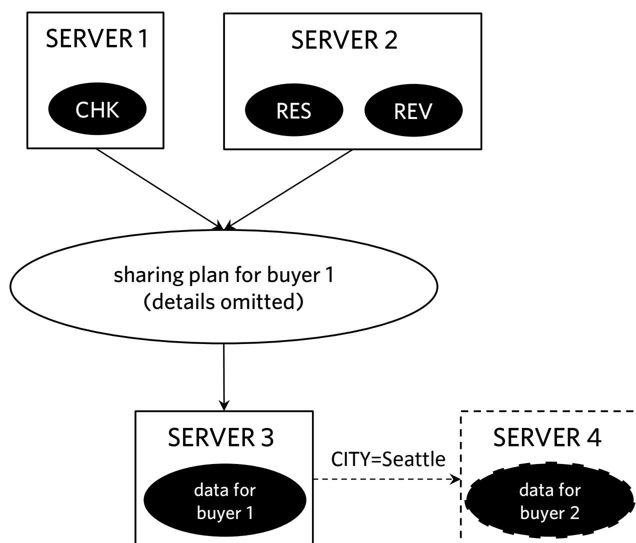

Figure 1: Sharing plans for two buyers. Source data are located on servers 1-2, and the purchased data (view) are located on server 3 (for buyer 1) and server 4 (for buyer 2).

The following example illustrates the problems studied in this paper. Following the terminology in [9], we use the term *dynamic data sharing* to refer to the sale or sharing of such dynamic data sets. A *sharing plan* specifies the set of operations/subexpressions to prepare the data for the buyer (such as the order of joins among the requested tables, time to apply predicates, time to move data between servers, etc.), which is analogous to a query plan for a SQL query.

EXAMPLE 1.1. *[**Selecting Sharing Plans**]: Consider three data sets in the data market in the form of relational tables: check-in at restaurants (CHK), restaurant information (RES), and restaurant reviews (REV). A data buyer (buyer 1) is interested in a dynamic data sharing that joins these three tables. These tables may be owned by different data owners and reside in different physical servers in the cloud infrastructure. It is not trivial to design a plan with minimum cost that delivers and maintains the data requested by buyer 1, which involves the order of join, the way to move data between the servers, etc.; each of these operations may incur a dollar cost for the service provider, especially if the service provider rents infrastructures from an IaaS provider. Furthermore, if there's an existing data sharing that maintains the join of CHK and RES, it should be taken into account when choosing the plan for the new sharing, since the data of the existing sharing (CHK $\bowtie$ RES) may be reused.*

*[**Costing Sharing Plans**]: Suppose we've selected a plan for this data sharing, as shown in solid lines in Figure 1 with details omitted. Later another buyer (buyer 2) is also interested in a dynamic data sharing that joins CHK, RES and REV, but she is only interested in restaurants in Seattle. The service provider decides that the best plan for this buyer is to reuse the previous plan, and add a filter "city = Seattle" in the end, as shown in the dotted part in Figure 1. Now suppose that the operational cost of maintaining these two sharings is $200/month. Then, what is the operational cost of each sharing? If we use a trivial approach that evenly divides the cost of each operation/subexpression among the sharings using the subexpression, the second sharing will be considered more costly than the first, since the second sharing plan has an additional step, "city = Seattle". Consequently, buyer 2 may pay a higher price than buyer 1. However, this is not fair to buyer 2 because if buyer 1 did not exist, the second sharing plan may apply the predicate "city = Seattle" earlier, which may make the RES table much smaller and the sharing plan much cheaper.*

There are similarities between the problem of selecting sharing plans and some classic problems such as distributed query processing, multiple query optimization, view maintenance, and index/view selection. These problems are indeed related but are also fundamentally different. More detailed related work discussion is presented in Section 2.

For selecting sharing plans, we propose an online algorithm. The algorithm should be online since it needs to service a sharing request as soon as it is received without knowing future requests. Our algorithm makes a significant improvement upon an existing work [9], which uses a greedy online algorithm (referred to as algorithm GREEDY in this paper). Algorithm GREEDY enumerates the plans for the new sharing and chooses the plan that incurs the smallest additional dollar cost after integrating into the plans for existing sharings (referred to as *global plan*). We show that algorithm GREEDY can perform arbitrarily badly even for very simple instances of the problem. We also analyze another baseline algorithm named algorithm NORMALIZE, which normalizes the cost of a subexpression using the number of prior occurrences of the subexpression, and show that it can also perform arbitrarily badly. In contrast, our proposed algorithm, named algorithm MANAGEDRISK, judiciously chooses the plan for each sharing such that it neither avoids taking risks nor takes too much risks, which avoids making arbitrarily bad decisions for those instances where the baseline algorithms fail.

For costing sharing plans, we propose a set of *fairness criteria* for costing data sharings that consists of five conditions (details will be given in Section 5) . These five conditions capture the degree of fairness, which is represented as a value between 0 and 1. The five conditions are non-redundant since it is possible to meet any four conditions but not the remaining one. We further present the necessary and sufficient condition of their satisfiability, and present an algorithm, named algorithm FAIRCOST, that maximizes the degree of fairness.

Our contributions are summarized as:

- We propose an online algorithm, algorithm MANAGEDRISK, that selects sharing plans for dynamic data sharings in a cloud data market. Algorithm MANAGEDRISK avoids the pitfalls in the baseline algorithms and avoids making bad decisions observed on the baseline algorithms.
- This is the first study on fair costing of data sharing in a data market. We propose fairness criteria which represent fairness as a value between 0 and 1, and design an algorithm to find the costing function that maximizes the fairness.
- Our experiments verified the effectiveness and efficiency of the proposed approaches.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents backgrounds, including the data market model, dynamic data sharings, sharing plans and cost of a sharing plan. Section 4 discusses several online algorithms for sharing plan selection, including two baseline algorithms GREEDY and NORMALIZE, as well as the proposed algorithm MANAGEDRISK. Section 5 introduces fairness criteria for costing data sharings, and explains an algorithm that maximizes fairness. Results of evaluations are reported in Section 6. Section 7 makes conclusions and discusses future works.

2. RELATED WORK

2.1 Selecting Sharing Plans

The problem of selecting sharing plans is related to but different from several classic problems such as distributed query processing, multiple query optimization, view maintenance and index/view selection.

Distributed Query Processing aims to quickly find a good plan for a query whose data are distributed in multiple machines [18, 25, 27]. In our problem the algorithm for selecting sharing plans doesn't need to have a very short response time, and thus for each sharing, we can often afford to explore a much larger space of possible plans or even enumerate all possible plans. The challenge in our problem is that, since there are multiple sharings and we process them in an online fashion, it is difficult to choose the right plan for each query that will optimize the total cost of the global plan, which is the focus of our proposed algorithm MANAGEDRISK. As to be shown later, if we simply use the cheapest plan for each query (i.e., algorithm GREEDY), the result can be arbitrarily bad even for simple instances of the problem.

The studies on *Multiple Query Optimization* have different focuses. Earlier works such as [32, 33, 34] focus on choosing the right plan for each query such that common subexpressions in multiple queries can be reused. It is related to our problem, however, it is an offline problem where all queries are known in advance and it starts to process all queries at the same time. Algorithms for this problem are based on heuristics such as A* search. Later works mainly focus on how to synchronize a single operation (such as table scan) among multiple query plans that use this operation, which is a hard problem if different queries are submitted at different times [10, 11, 15, 16]. By contrast, in our problem the shared data need to be constantly maintained, and our focus is which plan should be used for each sharing, which is an orthogonal problem.

View maintenance is complementary to the problem of selecting sharing plans. In view maintenance, the views to be maintained are given and the optimization goal is to decide when to propagate updates, reduce the number of queries to bring a view to a consistent state, decide whether to maintain views in incremental or recomputation fashion, etc. [6, 22, 28, 29]. In our problem, the main focus is to determine a plan for each sharing, and after a plan is selected, view maintenance techniques can be applied to maintain the views for the selected plan.

Index/view selection is the problem of selecting indexes to build or views to materialize in relational databases based on the workload [7, 8, 13, 14, 30, 31, 37]. The idea is to select indexes/views which benefit a window of recent queries, with the assumption that queries in the near future will be similar as recent queries. There are mainly three types of algorithms: hill climbing approaches([7, 8, 13, 14]) which enumerate small subsets of indexes/views, select subsets with large benefits and small costs, and greedily increase the size of the subsets until a target is reached; Knapsack approaches([37]), which alter the benefits/costs of indexes/views in a way that accounts for the interactions among indexes/views, and use the knapsack algorithm for index/view selection; and feedback-based approaches([30]) that involve elicitation from DBAs. Many aspects of the index/view selection problem differ from those of our problem, for example: (1) we focus on dynamic data and the views in our problem are constantly maintained; (2) in our problem we cannot choose each view independently, since each sharing plan is composed of multiple views which must be materialized together if this plan is selected; (3) the goal of index/view selection is to maximize the benefit of the selected indexes/views given a storage budget, whereas the goal of our problem is minimizing the cost of the global sharing plan where sharings are received online; (4) in our problem it is much more difficult to determine how selecting a particular plan for one sharing affects (positively or negatively) the cost of another sharing, since the cost may be collectively affected by the plans selected for many other sharings.

A knapsack-based algorithm is not suitable for our problem since there are a large number of interactions among different sharings as well as different plans for the same sharing, and it is infeasible to assign an independent benefit and cost for each individual view or sharing plan. A feedback-based approach is also ineffective since a sharing may have a large number of possible plans that involve an even larger number of views, and they heavily interact with one another. It is difficult for a human being to come up with a feedback for each view or plan. The hill climbing idea is applicable to our problem: given a new sharing, we can generate a subset of plans and select the most desirable one, using the hill climbing philosophy. The two baseline algorithms to be introduced in Section 4, algorithm Greedy and algorithm Normalize, are based on similar ideas. However, their solutions may have an unbounded cost for simple instances of the problem.

2.2 Costing Sharing Plans

Pricing data has been studied in several publications in the database community [17, 19, 20, 21, 36]. [17] aims to help database service providers determine the prices of structures (indexes, materialized views, cached columns) built by the service provider that benefit future queries. [36] has a similar setting as [17] where an optimization can be built that benefits future users. It takes a bidding approach where each user tells the service provider how much she's willing to pay for an optimization. An optimization is built if a set of users can be found such that, by charging each users the same amount, the cost of the optimization can be recovered, and this amount does not exceed each user's budget.

Both [17] and [36] study how to determine the price of a product assuming the cost of the product can be easily obtained. On the other hand, we focus on the problem of determining the *cost* of a product (i.e., sharing), since different sharings interact in the global sharing plan, and it is challenging to calculate the cost of each sharing in a fair way. In our case, given the cost, determining the price is an independent problem, where methodologies in the economics field can be applied. Besides, although the pricing strategies in [17, 36] are also fair, it is much more challenging to achieve fairness in costing sharing plans. In [17, 36], since different users/queries are independent, fairness is simply achieved by charging each user/query the same price to use the same optimization/structure. However, in our costing problem many sharing plans interact in the global plan by reusing subexpressions. It is further complicated by the fact that each sharing has multiple possible plans, and a plan may need to be considered even if it is not used, as shown in Example 1.1. A simple approach that equally distributes the cost of each subexpression to the sharing plans that use this subexpression is *not* fair.

[19] studies the problem of pricing a query that returns a set of tuples, given the price of the whole data set and the prices of some views of the data. The main idea is to find the cheapest data-dependent rewriting of the query using views, and price the query as the sum of the prices of these views (the price of computation is neglected, which is fair for static data since the monetary value of the data may far exceed the cost of computation). If a rewriting cannot be found, the query will have the same price as the whole data set. In this way, the price of the query satisfies two desirable properties: arbitrage-free and discount-free. [20] extends [19] by formulating the rewriting problem for a wide range of queries, most

of which makes the rewriting NP-hard, as ILP. It further discusses how to avoid double-charging when the user buys similar queries, how to price data updates, and how to fairly share the revenue if the query involves data from multiple data sellers.

The problems studied in [19, 20] are orthogonal to ours. They study the price of *data*, rather than the operational cost, i.e., cost of *delivering and maintaining the data*. Our problem, on the other hand, focuses on the latter, as explained in Section 1.

3. PRELIMINARIES

3.1 Data Market and Dynamic Data Sharing

We use the data market model proposed in [9]. A data market is a cloud computing infrastructure where tenants pay to use computing resources to run their applications and have the opportunity to sell data to one another through data sharings. Since tenants' applications keep collecting new data (e.g., the CHK table in Figure 1 keeps collecting new check-in information), the data sold in the data market are dynamic. This is in contrast to the type of data markets like Microsoft Azure Marketplace, Infochimps, etc., where static data sets are sold.

A data owner willing to sell a data set makes the data set accessible to the service provider. In this paper we follow [9, 19, 20] in considering data in the form of relational tables, but other forms can also be used. A buyer willing to purchase data may submit a data sharing request to the service provider in the form of a query, such as the example in Section 1 where a buyer wants to purchase the join of CHK, RES and REV. To service the request, the service provider is responsible for creating and maintaining the view specified in the query, which incurs dollar costs for using resources such as storage, CPU, network, etc., if the service provider rents resources from an IaaS provider such as AWS. As explained before, the price of a data sharing is a function of the data price specified by the data owner, as well as the operational cost incurred to deliver and maintain the data for the buyer.

3.2 Sharing Plan

A sharing plan determines how data should be moved among the servers, in which order the joins and predicates should be applied, etc., in order to maintain the shared data. Each join in the sharing plan can be specified as

$$(A, s_1) \bowtie (B, s_2) \rightarrow s_3$$

where s_1, s_2 are the servers that have a copy of A and B, respectively, which may be frequently updated, and s_3 is where the result should be placed. A possible plan for this join where s_1 = server 1 and s_2, s_3 = server 2 is shown in Figure 2, where an ellipse denotes a base relation and a rounded rectangle denotes a *delta relation*, which receives updates to the corresponding base relation. Note that this plan avoids copying base relations across servers, and only copies delta relations.

For multiple sharings with common subexpressions, such as the two sharings in Example 1.1, the computation of a common subexpression can be reused so that the subexpression is only computed once. A plan involving multiple sharings is called a *global plan*. Next we introduce the costing of sharing plans in the global plan.

3.3 Costing of Sharing Plans

We assume that the data market service provider has a cost model for estimating the dollar cost of each subexpression, e.g., copy, merge, join, etc. To obtain the cost model, there exist analytical models to estimate resource usages for various operations in the cloud [24], and the resource usages can be directly mapped to dollar cost in cloud services such as AWS. Thus the service provider

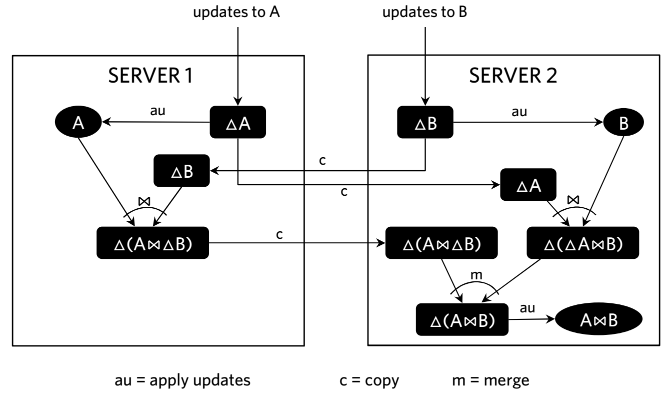

Figure 2: A possible plan for joining relation A on server 1 and relation B on server 2 such that the resulting view $A \bowtie B$ is placed on server 2

can calculate the cost per time unit of an individual sharing plan, which is the sum of the cost of each subexpression in the plan, multiplied by the number of times they are executed per time unit. However, this is not sufficient, as the service provider needs to determine the cost incurred by each sharing plan in the global plan in order to calculate the price of each sharing. This is complicated since different sharing plans in the global plan may reuse common subexpressions, and as said before, simply dividing the cost of each subexpression by the number of sharing plans that use it isn't fair.

Suppose the cost of the global plan is $cost(GP)$ and there are n sharing plans $P_1, \cdots, P_n$ where the cost attributed to P_i (referred to as "attributed cost") is $AC(P_i)$, then the total cost of these sharing plans should equal the cost of the global plan, i.e.,

$$\sum_{i=1}^{n} AC(P_i) = cost(GP)$$

and $cost(GP)$ should be distributed to each $AC(P_i)$ in a fair way. Section 5 will further discuss the criteria of fairness and how to achieve maximum fairness.

4. ONLINE SHARING PLAN SELECTION

4.1 Problem Definition

As discussed before, the service provider needs to select a sharing plan for each new sharing without knowing future sharings. Thus the algorithm needs to be online. We define the following online sharing plan selection problem.

DEFINITION 4.1 (ONLINE SHARING PLAN SELECTION). *The input contains a sequence of dynamic data sharings, a cost model and the initial state of the system. The service provider should select a sharing plan for each sharing without the knowledge of future sharings. The goal of the service provider is to minimize the total cost of servicing the sequence of sharings.*

The cost model is used to calculate the cost of each subexpression in a sharing plan, and the initial state of the system refers to the initial placement of data, i.e., which table is on which servers, and the server capacity constraint, which can be expressed in multiple ways such as how many tuples the server can handle per second.

For ease of illustration and explanation, we first consider a special case of the problem, where servers have unlimited capacity, and each sharing is a join-only query with no predicates or projections.

We will discuss the general case in Section 4.5. Note that servers having unlimited capacity doesn't mean that all sharings are processed on a single server, since different source data may be stored on different servers.

In the following, we denote a sharing as a set of source tables. For example, let a, b, c be three tables. A sharing that joins these three tables is denoted as (a, b, c).[3] A subexpression (i.e., join) is denoted by two sets of tables, e.g., ab is the join of a and b, and $a(bc)$ is the join of a with $b \bowtie c$. A sharing plan is represented by a sequence of joins, e.g., $a(bc)$ is the plan where we first join b with c, and then join the result with a. Note that notation $a(bc)$ may represent either a subexpression or a sharing plan, but it is not a problem when the context is clear.

We use $C[\cdot]$ to represent the cost of a sharing plan and $c[\cdot]$ to represent the cost of a subexpression. For example, $C[a(bc)]$ is the cost of the aforementioned sharing plan, and $c[a(bc)]$ denotes the cost of joining a with $b \bowtie c$. Thus $C[a(bc)] = c[bc] + c[a(bc)]$. Let $\#join(S)$ be the number of joins in a plan of sharing S. For example, the value of $\#join$ for sharing (a, b, c) is 2, and all plans for this sharing have 2 joins.

Next, we discuss two baseline algorithms, namely algorithm GREEDY and algorithm NORMALIZE, before presenting our proposed algorithm MANAGEDRISK. Both baseline algorithms adopt the idea of hill-climbing, which is seen in the algorithms of many classic problems including index/view selection [13]. It refers to the attempt to add a good plan of the new sharing to the global plan. Algorithm GREEDY prefers a plan that adds the smallest cost to the global plan, while Algorithm NORMALIZE considers the subexpressions occurred in the existing sharings and assumes that they will occur again in future, and thus it chooses a plan with this assumption in mind. At a high level, for each sharing, all three algorithms enumerate all possible plans, but use different criteria to decide which plan to select.

Note that in most cases we can afford to enumerate all possible plans, since choosing sharing plans is not an interactive or time-critical task. In case the sharing involves a complex query for which enumerating all plans is infeasible, we can use various heuristics, such as hill climbing and beam search, to generate a manageable subset of all possible plans [12].

4.2 Algorithm Greedy

Algorithm GREEDY is adopted in an existing work [9]. As its name suggests, it enumerates all possible plans for a sharing, and chooses the one with the minimum additional cost after adding it to the global plan. The following example shows how algorithm GREEDY works and why it may perform poorly, even if each sharing has at most two joins.

EXAMPLE 4.1. *Suppose there is a single server, and all sharings are processed within this server. Consider a sequence of sharings* $(a, b, c_1), (a, b, c_2), \cdots$. *Suppose there are two possible plans for each sharing:* $(ab)c_x$ *and* $a(bc_x)$*, such that* $c[ab] = 100$, $c[(ab)c_x] = \epsilon$ *where* ϵ *is a negligibly small positive number, and* $C[a(bc_x)] = 10$. *If there are sufficiently many such sharings (more than 10), an optimal algorithm will use plan* $(ab)c_1$ *for the first sharing, so that all other sharings can reuse the result of ab and will only cost* ϵ *each. Suppose there are n sharings, the total optimal cost is* $100 + n\epsilon$. *Algorithm* GREEDY*, on the other hand, will always use plan* $a(bc_x)$ *for each sharing, and has a total cost of* $10n$*, which is unbounded compared to the optimal cost.*

[3]We focus on natural joins, so no further specification is needed for a sharing. Our approach supports other types of joins but they will make the notations in this paper unnecessarily more complicated.

As we can see, algorithm GREEDY does not take any risk (here "risk" refers to using plan $(ab)c_x$, since we do not know whether there will be future sharings to amortize the cost of ab, $c[ab]$). At the first glance, this seems what an algorithm should do, since it does not know the future and there is no incentive to take the risk and use plan $(ab)c_x$. However, we will show in Section 4.4 that this is not necessarily true.

4.3 Algorithm Normalize

An attempt to solve the weakness of algorithm GREEDY can lead to another baseline algorithm, which we name algorithm NORMALIZE. To explain it, we introduce the following definition.

DEFINITION 4.2. *A sharing S is said to* contain *a subexpression s, denoted as* $s \lessdot S$*, if the subexpression occurs in one of the possible plans for the sharing.*

For example, a sharing (a, b, c, d) may contain subexpressions ab, bc, cd, ac, $(ab)c$, $a(bcd)$, $(ab)(cd)$, etc. (depending on joinability between tables), each of which denotes a join.

Algorithm NORMALIZE normalizes the cost of each subexpression in the current sharing by the number of sharings seen so far that contain this subexpression. Let C_n and c_n denote the normalized cost of a sharing plan and a subexpression, respectively. Algorithm NORMALIZE selects the plan with the smallest normalized cost. For the sharing sequence in Example 4.1, when NORMALIZE processes the xth sharing, if the first $x - 1$ sharings all use plan $a(bc_x)$, then $c_n[ab]$ in the xth sharing is considered to be its original cost (100) divided by x, because ab is contained in all x sharings seen so far.

In this way, NORMALIZE will use $a(bc_x)$ for the first 10 sharings, and for the 11th sharing, $c_n[ab]$ is 100/11, so $C_n[(ab)c_{11}] < C_n[a(bc_{11})]$ and NORMALIZE will select plan $(ab)c_{11}$. In other words, although NORMALIZE makes the wrong choices for the first 10 sharings, it eventually realizes that subexpression ab has occurred many times and decides to use ab even though it adds more cost to the global plan than the other option. Although it doesn't give the optimal solution, its cost is bounded in this particular example compared with the optimal solution.

Although NORMALIZE works better than GREEDY for Example 4.1, it may still have an unbounded cost even if each sharing has at most two joins, as shown in the following example.

EXAMPLE 4.2. *Consider a sequence of n sharings* (a, b, c_1), $(a, b, c_2), \cdots, (a, b, c_n)$. *Again, suppose there are two possible plans for each sharing:* $(ab)c_x$ *and* $a(bc_x)$. $c[ab] = n$. *For* $1 \le x \le n - 1$, $C[a(bc_x)] = \epsilon$ *and* $c[(ab)c_x] = \epsilon$. *For the nth sharing,* $C[a(bc_n)] = 1 + 2\epsilon$ *and* $c[(ab)c_n] = \epsilon$.

For this sharing sequence, NORMALIZE *will choose* $a(bc_x)$ *for the first* $n - 1$ *sharings, incurring a cost of* $(n - 1)\epsilon$. *For the last sharing,* $c_n[ab] = 1$ *(since it is contained in all n sharings), thus* $C_n[(ab)c_n] = 1+\epsilon < C_n[a(bc_n)] = 1+2\epsilon$*, and* NORMALIZE *chooses plan* $(ab)c_n$. *The total cost of* NORMALIZE *is* $n + n\epsilon$. *An optimal algorithm would choose plan* $a(bc_x)$ *for all sharings for a total cost of* $1 + (n + 1)\epsilon$. *Since n can be arbitrarily large and* ϵ *can be arbitrarily small, algorithm* NORMALIZE *has an unbounded cost compared with the optimal cost.*

As we can see, NORMALIZE takes a big risk for the last sharing by using plan $(ab)c_n$, for which it gets no reward since it is the last sharing. To address the problem in both algorithms discussed so far, next we propose algorithm MANAGEDRISK.

4.4 Algorithm ManagedRisk

We can see from the previous two examples that we need to take some risk, since an algorithm that takes no risk, such as GREEDY,

has a poor performance; however, the risk we take needs to be somehow controlled to avoid the situation in Example 4.2. The idea of algorithm MANAGEDRISK, at a high level, is that we should take risks, but we should only take a risk on a sharing if the cost of previous sharings are sufficiently high, so that even if the risk we take turns out to be a bad one, the additional cost incurred can be "absorbed" by previous sharings. We introduce the concept of *regret* to capture this idea.

DEFINITION 4.3. *Let $S_1, S_2, \cdots$ be a sequence of sharings, and let P_i denote the sharing plan for S_i. For each sharing S_i and each subexpression $s \lessdot S_i$, the regret of s wrt S_i, denoted by $rg_i(s)$, is recursively defined as: if the result of s is not produced in any $P_j (1 \le j < i)$,*

$$rg_i(s) = \sum_{S_j | j<i, s \lessdot S_j} \frac{C[P_j] - \sum_{s' \in P_j} rg_j(s')}{m-1} \quad (1)$$

where $m = \#join(S_i)$. Otherwise, $rg_i(s) = 0$.

"The result of s is not produced in any $P_j (1 \le j < i)$" means that the result of s is not available when we process sharing S_i, i.e., if we wish to use s in the plan of S_i, we need to pay a cost of $c[s]$. For example, if $s = (ab)c$, then this means that no sharing prior to S_i uses subexpression $(ab)c$ or $a(bc)$ in its sharing plan.

Algorithm MANAGEDRISK is shown in Algorithm 1. For each sharing S_i in the sequence and each plan P_{ij} for S_i, it uses a scoring function $score(P_{ij})$ defined as

$$score[P_{ij}] = \sum_{s \in P_{ij}} rg_i(s) - C[P_{ij}] \quad (2)$$

A sharing plan with large regret and small cost gets a high score. MANAGEDRISK chooses the plan for sharing S_i with the maximum score among all possible plans for S_i.

The intuition of algorithm MANAGEDRISK is as follows. When we process a sharing S_i, if there exists a subexpression $s \lessdot S_i$ which is contained in some of the previous sharings but is never used before, then we give algorithm MANAGEDRISK an incentive to use s equivalent to $rg_i(s)$. $rg_i(s)$ is large if there are many sharings prior to S_i that contain subexpression s. By giving such an incentive, we can avoid the problem in Example 4.1 where a subexpression is never used, because the incentive keeps increasing if we don't use it, and at some point the incentive will be big enough that the subexpression will be used. Even if this is a bad choice, e.g., future sharings will never utilize this subexpression (like the situation in Example 4.2: after algorithm NORMALIZE uses ab, there is no more sharing to benefit from it), the "damage" it causes will likely be controlled, because the incentive to use this subexpression won't be too large (otherwise it should have been used earlier). These are of course intuitions rather than strict statements, but we will show in Example 4.3 that algorithm MANAGEDRISK does avoid the pitfalls in both previous examples.

Note that the regrets of subexpressions used in each P_j (i.e., $rg_j(s')$ in Eq. (1)) are subtracted from $rg_i(s)$, because $rg_j(s')$ has already made an impact on choosing plan P_j for sharing S_j, and it should not make another impact on choosing the plan for S_i. Otherwise, the selected plans may have an unbounded cost compared with the optimal cost even if each sharing has at most two joins. The factor of $1/(m-1)$ in Eq. (1) is to avoid the total regret of a sharing plan with many subexpressions being too large.

EXAMPLE 4.3. *Consider the sharing sequence in Example 4.1. For the first 10 sharings,* MANAGEDRISK *uses plan $a(bc_x)$, and pays a cost of 10 for each plan. When it processes the 11th sharing, we have $rg_{11}(ab) = 100$, and the regrets of all other subexpressions are 0. Since*

$$rg_{11}(ab) - C[(ab)c_{11}] = -\epsilon > -C[a(bc_{11})] = -10,$$

algorithm MANAGEDRISK *chooses plan $(ab)c_{11}$ for this sharing. Note that even if the 11th sharing is the last sharing, which means using $(ab)c_{11}$ at this point is a bad choice, the cost of* MANAGEDRISK *won't be arbitrarily bad because the incentive given to* MANAGEDRISK *to use ab is no more than the total cost of the first 10 sharing plans. In this example the cost of* MANAGEDRISK *is no more than twice the optimal cost.*

Now consider the sharing sequence in Example 4.2. For $1 \le x \le n-1$, MANAGEDRISK *uses plan $a(bc_x)$, incurring a cost of $(n-1)\epsilon$, and thus $rg_n(ab) = (n-1)\epsilon$. For the nth sharing, since the regrets of all other subexpression are 0, we have*

$$rg_n(ab) - C[(ab)c_n] < -C[a(bc_n)]$$

thus MANAGEDRISK *will choose $a(bc_n)$. In this case, even though subexpression ab is contained in many sharings seen before,* MANAGEDRISK *still doesn't use ab for the nth sharing, since the total cost of all previous sharings that contain ab (i.e., $rg_n(ab)$) is too small and thus the incentive to use ab is not big enough.* MANAGEDRISK *finds the optimal plans for this sharing sequence.*

Algorithm 1: Algorithm MANAGEDRISK for the Special Case

```
Input : a sequence of sharings S_1, ..., S_n. The algorithm
        processes each sharing S_i without the information of
        sharings after S_i.
foreach sharing S_i do
    foreach subexpression p ⋖ S_i do
        compute rg_i(p) using Eq. 1
    end
    enumerate all plans for S_i
    foreach possible plan P_ij of S_i do
        compute C(P_ij) using the cost model
        score(P_ij) = Σ_{s∈P_ij} rg_i(s) − C(P_ij)
    end
    j = arg max score(P_ij)
    use plan P_ij for sharing S_i
end
```

There is a similar notion of regret (also called opportunity loss) in decision theory [26], which is defined as the additional payoff if a different action is chosen. Although the idea is somewhat similar, there are some key differences. First, decision theory aims to make a choice (such as determining the inventory level of a product) that minimizes the future regret if something goes wrong in future; whereas we do not analyze what can possibly happen in the future (because we don't know or make assumptions on how many sharings we will receive in the future, and what they are). Instead, regret is computed from previous sharings. Second, regret in decision theory is simply the difference in payoff, whereas in our problem the "difference in payoff" cannot be easily computed, because using a different plan for one sharing may affect the "difference in payoff" of many other sharings.

After explaining how algorithm MANAGEDRISK works in a special setting, in the next subsection we discuss how to apply algorithm MANAGEDRISK in the general case.

Algorithm 2: Algorithm ManagedRisk for the General Case

Input : a sequence of sharings $S_1, \cdots, S_n$. The algorithm processes each sharing S_i without the information of sharings after S_i.

foreach *sharing* S_i **do**
  foreach *subexpression* $p \triangleleft S_i$ **do**
    compute $rg_i(p)$ using Eq. 1
  end
  enumerate all plans for S_i
  foreach *possible plan* P_{ij} *of* S_i **do**
    compute $C(P_{ij})$ using the cost model
    $score(P_{ij}) = \sum_{s \in P_{ij}} rg_i(s) \cdot perc_s(P_{ij}) - C(P_{ij})$
  end
  foreach *possible plan* P_{ij} *of* S_i *in descending order of score* **do**
    if P_{ij} *does not violate server capacity* **then**
      use plan P_{ij} for sharing S_i
      break
    end
  end
  if *no feasible plan exists* **then**
    reject S_i
  end
end

4.5 Extension of Algorithm ManagedRisk for the General Case

We previously made two simplifications: (1) server capacity is considered unlimited; (2) sharings are join-only with no projections or predicates. To cope with the general case, we propose the following extensions of Algorithm ManagedRisk.

When a server has limited capacity such that the desired plan violates the capacity of some servers, we will use the best plan that does not violate any server capacity. If no such plan exists, the sharing is rejected.

When sharings have predicates and projections, we modify the way we compute the score of a sharing plan (Eq. 2). Intuitively, even if the regret of a subexpression s (e.g., $a \bowtie b$) is high, if a sharing plan P for the current sharing only computes a small subset of the result of s (e.g., $s' = a_{a.x<10} \bowtie b$), then it is not very helpful to use plan P, since it only has a small chance to be helpful for future sharings that contain $a \bowtie b$. Consequently, the incentive to use s' should be smaller than the regret of s. We use $perc_s(P)$ to denote the percentage of tuples computed by subexpression s (possibly with predicates) in plan P, compared with the tuples computed by the same subexpression with no predicate. For a plan P with no predicate, $perc_s(P) = 100\%$ for all $s \in P$. Otherwise, $perc_s(P)$ may be smaller than 100%, which can be estimated using various existing techniques for selectivity estimation. We modify Eq. 2 as follows:

$$score(P_{ij}) = \sum_{s \in P_{ij}} rg_i(s) \cdot perc_s(P_{ij}) - C(P_{ij}) \tag{3}$$

The extension of Algorithm 1 for the general case is shown in Algorithm 2.

5. FAIR COSTING FOR SHARINGS

Calculating the operational cost incurred for the service provider to provide and maintain the view of a sharing is necessary in pricing the sharing. We have shown in Example 1.1 that a fair costing mechanism is not trivial to obtain. To better motivate our proposed fairness criteria, we use a more sophisticated example as follows.

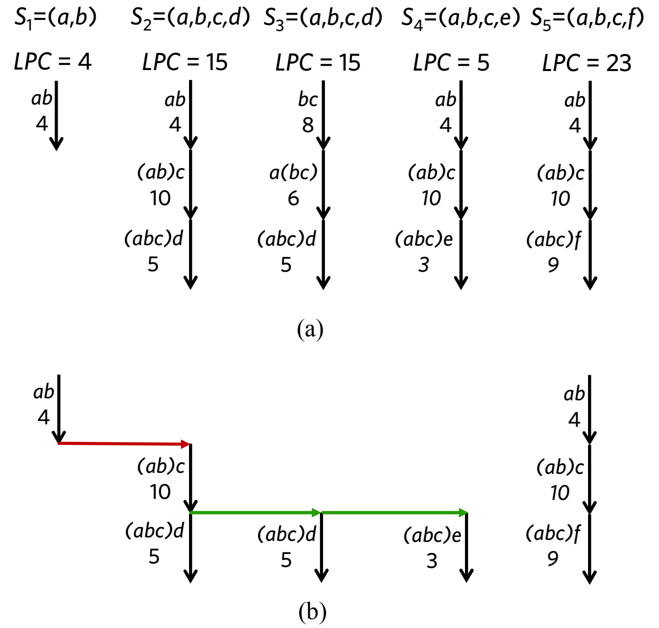

Figure 3: Individual Plans (a) and Global Plan (b) for Five Sharings

Example 5.1. *Consider five sharings shown in Figure 3. Figure 3(a) shows the plans for the sharings generated by an algorithm, where each edge represents a subexpression, and is annotated by the subexpression it represents, along with its cost. It also shows the minimum cost of a plan for this sharing alone, denoted by LPC(·) (lowest possible cost). Figure 3(b) shows the global plan where some common subexpressions in the plans are reused. The red arrow denotes reusing the result of ab and the green arrows denote reusing the result of* $(ab)c$.

Next we introduce and explain the fair costing criteria. We use AC (attributed cost) to denote the cost attributed to each sharing, and our goal is to compute a fair AC for each sharing.

(1) For any two identical sharings $S_1 = S_2$, $AC(S_1)$ *should be identical with* $AC(S_2)$ *regardless of the plans chosen for them.* Buyers only request data sharings. They do not know or care about what plans the service provider decides to use for their sharings. The service provider may use different plans for the same sharings for several reasons, e.g., server capacity limit, reuse of subexpressions, etc. From the buyers' points of view, in order to be fair, neither should get a lower or higher attributed cost than the other. In Example 5.1, sharings S_2 and S_3 are identical. Although they use different plans, i.e., $((ab)c)d$ for S_2 and $(a(bc))d$ for S_3, they should have the same AC.

(2) For any sharing S, $AC(S)$ *should be no more than* $LPC(S)$. Since $LPC(S)$ is the lowest cost of S if no other sharing exists (thus there's no reuse of subexpressions), it represents the actual complexity of S. A sharing with a high LPC is inherently expensive in terms of operational cost, and conversely, a sharing with a low LPC is inherently cheap. For global optimization purpose, the service provider may not use the cheapest plan for a sharing, such as the one with predicate "city = Seattle" in Example 1.1, as well as S_4 in Example 5.1. Both of them use plans that have an additional step after some expensive operations. However, from the fairness

perspective, buyers of such inherently cheap sharings should not be penalized by the optimization, and thus we propose that *AC* cannot be more than *LPC* for a sharing.

(3) For two sharings S_1 and S_2, if S_1's query is contained in S_2's query (i.e., the tuples retrieved by S_1 are a subset of those retrieved by S_2), and $LPC(S_1) \leq LPC(S_2)$, then $AC(S_1)$ should be no more than $AC(S_2)$. Because otherwise, even if a buyer only needs the data of S_1, she can purchase S_2 for a lower price. This is undesirable for the service provider since the service provider pays more but gets a lower revenue.

(4) A sharing plan that has common subexpressions with other sharings, which gives the service provider the opportunity to save cost by reusing subexpressions, should be compensated. In Example 5.1, sharing plans for S_1, S_2, S_4 and S_5 all compute $a \bowtie b$ (denoted by *ab*), and sharing plans for S_2, S_3, S_4 and S_5 all compute $a \bowtie b \bowtie c$. These common intermediate results enable the service provider to reuse them in different sharing plans and reduce the cost. Although an intermediate result may not be reused by all sharing plans that contain this intermediate result (e.g., $a \bowtie b$ in S_1's plan is only reused by S_2), all sharings whose plans contain the intermediate result should be equally rewarded. To capture this idea we introduce the concept of *saving* of an intermediate result in a sharing plan.

DEFINITION 5.1 (SAVING OF AN INTERMEDIATE RESULT). *The saving of an intermediate result r, denoted as saving(r), is the increase of the cost of the global plan if r is no longer reused in the global plan, i.e., all sharings whose plans include r need to compute r and pay the cost of the corresponding subexpressions.*

In Example 5.1, there are two intermediate results that are reused, shown in red (*ab*) and green (*abc*). If we remove the red arrow, sharing S_2 will need to use a separate subexpression *ab*, thus the cost of the global plan increases by 4. If we remove the two green arrows, sharing S_3 will need to use subexpressions *bc* and *a*(*bc*), and sharing S_4 will need to use subexpressions *ab* and (*ab*)*c*, and the cost of the global plan increases by 28.

We require that part of the saving of an intermediate result should be equally awarded to the sharings whose plans include this intermediate result. Let α be a parameter that indicates at least how much percentage of the saving is awarded to the sharings. Let $num(r)$ denote the number of sharings in the global plan whose plans include r as an intermediate result. We require that

$$AC(S) \leq GPC(S) - \alpha \cdot \sum_{r \in S} \frac{saving(r)}{num(r)} \tag{4}$$

where $GPC(S)$ is the cost of S's plan in the global plan. It is calculated by summing up the cost of all edges in S's plan, even if an edge is used by other sharing plans. In Example 5.1, the *GPC* for the five sharings are 4, 19, 19, 17, 23, respectively.

Parameter α reflects the degree of fairness. $\alpha = 0$ means the savings of the intermediate results are not awarded to the relevant sharings, which is the least fair since a sharing with much commonality with other sharings is treated in the same way as a sharing with no commonality with others. $\alpha = 1$ means that the savings are maximally awarded to the sharings. $\alpha = 1$ is not always achievable because of other fairness requirements, and thus we want to find the maximum possible value of α.

(5) Finally, the sum of AC of all sharings in the global plan should equal the cost of the global plan, i.e., the cost of the global plan should be recovered. This is not directly related to fairness per se, but it is a necessary requirement for a costing function.

The five criteria above are collectively referred to as the *fairness criteria*. The following lemmas show that these requirements

Algorithm 3: Algorithm FAIRCOST

```
Input : global plan GP, sharings S_1, ..., S_n
if Σ_{S_i} LPC(S_i) < cost(GP) then
  | return IMPOSSIBLE
end
build a DAG: each node is a sharing (or multiple identical
sharings); each arc (S_i, S_j) indicates that S_i is contained in S_j
and LPC(S_i) ≤ LPC(S_j)
foreach intermediate result r in GP do
  | calculate saving(r) according to definition 5.1
end
lowα = 0, highα = 1, α = 0.5
while true do
  | foreach sharing S in increasing order of LPC do
  |   | let P_S be the predecessors of S in DAG
  |   | costUB(S) =
  |   | min{LPC(S), min_{S'∈P_S} costUB(S'), GPC(S) − α ·
  |   |   Σ_{r∈S} saving(r))/num(r) }
  | end
  | if Σ_{S_i} costUB(S_i) = cost(GP) then
  |   | break
  | end
  | else if Σ_{S_i} costUB(S_i) < cost(GP) then
  |   | highα = α − ε
  | end
  | else
  |   | lowα = α + ε
  | end
  | α = (lowα + highα)/2
end
foreach sharing S_i do
  | AC(S_i) = costUB(S_i)
end
```

are non-redundant, as well as the condition under which they are achievable. We omit the proofs due to space limitation.

LEMMA 5.1. *The five fairness conditions are non-redundant: it is possible to satisfy any four not the fifth.*

LEMMA 5.2. *All five fairness conditions are satisfiable on a global plan GP for a set $\mathcal{S}$ of sharings if and only if $\sum_{S \in \mathcal{S}} LPC(S) \geq cost(GP)$.*

Given a specific value of α, we can use the fairness criteria to compute an upper bound cost for each sharing. Note that conditions (1) and (3) make the set of sharings in the global plan a partially ordered set, which means the cost upper bound of a sharing depends on other sharings. Thus we should calculate the upper bound cost of the sharings according to the partial order, i.e., the cost upper bound of a sharing can be determined only after the cost upper bounds of all its predecessors have been determined. If the sum of all cost upper bounds are higher than the cost of the global plan, it means this value of α is feasible.

The algorithm for computing the maximum value of α, named algorithm FAIRCOST, is shown in Algorithm 3. Note that its input is the global plan and the output is the attributed cost (*AC*) for each sharing, and thus when a new sharing arrives, the costs of existing sharings may change. This is because if the costs of existing sharings cannot be changed, it is impossible to satisfy the above fairness criteria in a non-trivial way (i.e., $\alpha > 0$). However, the price of each sharing S won't change arbitrarily as it will never exceed $LPC(S)$.

Algorithm FAIRCOST first builds a DAG to reflect the partial order between sharings. Multiple identical sharings can be represented by a single node in the DAG. We then do a binary search on α. For a specific value of α, we compute the cost upper bounds for the sharings in the order of *LPC*, which ensures that a sharing is processed after all its predecessors in the DAG have been processed. If the total cost upper bound is more than *cost*(*GP*) we search for a higher α value, and if the total cost upper bound is less than *cost*(*GP*) we search for a lower α value.

If we run Algorithm FAIRCOST on Example 5.1, it first computes the savings of the intermediate results: $savings(ab) = 4$ and $saving(abc) = 28$. There are 4 sharings whose plans include *ab*: S_1, S_2, S_4 and S_5, and there are 4 sharings whose plans include *abc*: S_2, S_3, S_4 and S_5. The maximum possible value of α in this case is 0.8, and the attributed cost of the sharings are: $AC(S_1) = 3.2$, $AC(S_2) = 12.6$, $AC(S_3) = 12.6$, $AC(S_4) = 5$, $AC(S_5) = 16.6$. Their sum is 50, which is exactly the cost of the global plan in Figure 3(b). A higher value of α would mean that the attributed costs of S_1, S_2, S_3 and S_5 all need to be reduced, which is not possible, because the attributed cost of S_4 cannot be increased as it is the same as its *LPC*.

6. EVALUATION

6.1 Setup

6.1.1 Algorithms Evaluated

The evaluation has two parts. The first part (Section 6.2) evaluates online sharing plan selection, and the second part (Section 6.3) evaluates the fair costing of sharings. In the first part we test four algorithms: algorithms GREEDY, NORMALIZE, MANAGEDRISK presented in Section 4, as well as algorithm EXHAUSTIVE, which is an offline algorithm that knows all sharings in the sequence in advance, and searches for the optimal global plan exhaustively. Algorithm EXHAUSTIVE is not feasible in practice, since we do not know all sharings in advance, and even if we do, the search space is much too large. We only test Algorithm EXHAUSTIVE for small sharing sequences.

In the second part we compare Algorithm FAIRCOST with a baseline algorithm that evenly distributes the cost of each subexpression in the global plan to the sharings whose plans use the subexpression. This is a commonly adopted criterion for fairness, e.g., in [17, 36], all users/queries that use the same structure built by the service provider pay the same price for it. As an example, for the global plan in Figure 3(b), the baseline algorithm distributes the cost of *ab* to S_1, S_2, S_3 and S_4, and distributes the cost of (*ab*)*c* to S_2, S_3 and S_4. The two algorithms are compared based on whether they satisfy the fairness criteria discussed in Section 5.

6.1.2 Environment

The experiments were performed on a machine with two Intel Xeon 2.40GHz cores, 12GB memory, running Windows 7 Ultimate. We collected tweets on twitter from a gardenhose stream, which is composed of a 10% sampling of tweets in a 6-month period starting from September 2010. The tweets are stored in 9 base relations: USERS, TWEETS, CURLOC, LOC, SOCNET, URLS, FOURSQ, HASHTAGS, PHOTOS. These base tables are distributed in a round-robin fashion among a number of machines that varies from 5 to 9 machines (6 by default).

We specified 25 base sharings, each of which is a join of two or more tables. These sharings were derived from real mobile applications that join the mentioned tables. The base sharings are shown in Table 1. For each base sharing, we give an existing real application that may benefit from such a sharing (e.g., twitaholic may benefit from USERS $\bowtie$ SOCNET). Predicates are randomly generated and added to the base sharings; each predicate has the form of "Table.Attribute [>, <, =] Constant".

In order to test the scalabilities of the algorithms, we also generated a set of synthetic tables/sharings, which are used in Section 6.2.2. The synthetic data has a star schema with up to 5 fact tables and 30 dimension tables distributed among 1-20 machines, and the sharing sequence contains up to 2500 sharings, each of which is a star-join with no predicates. The cost of each join is a random number between 1 and 10^5.

In a prior work [9] we developed a cost model used to calculate the cost of each subexpression in a sharing plan, which is proved fairly accurate. Thus in this paper we use the cost model in [9] for sharing plan cost calculation instead of setting up and running the sharings in the system.

The algorithm for costing sharings are invoked on the output of Algorithm MANAGEDRISK on the Twitter data.

6.2 Online Sharing Plan Selection

6.2.1 Effectiveness of Plan Selection

We run the three online algorithms, GREEDY, NORMALIZE and MANAGEDRISK, on the 25 base sharings in Table 1. We vary the number of sharings in the sequence (10 to 60), number of predicates in each sharing (0 to 3) and number of machines (5 to 9). On average, the global plans generated by the three algorithms have similar costs. However, there exists certain sharing sequences where each algorithm performs much better (up to 3 times) than the other two. In general, GREEDY tends to use a subexpression *too late* as shown in Example 4.1; NORMALIZE tends to use a subexpression *too early* as shown in Example 4.2. Algorithm MANAGEDRISK generally avoids using a subexpression too late or too early, although it sometimes can still be a bit earlier or later than the optimal solution, which makes it occasionally have a higher cost than either GREEDY or NORMALIZE. However, by taking managed risks, algorithm MANAGEDRISK does not make choices that cause unbounded damages, as shown below.

To further illustrate the worst case scenario, we generate sequences of synthetic sharings, each of which consists of a three-way join without predicates. The synthetic sharings consists of cases similar as Example 4.1 where there exists a subexpression contained in many sharings and used by the optimal solution, and cases similar as Example 4.2 where there exists a subexpression contained in many sharings but not used by the optimal solution, among others. Figure 4 shows the worst case scenario in this test for each algorithm, e.g., "Greedy/MR" is the largest cost ratio between GREEDY and MANAGEDRISK on a sharing sequence in the test. As

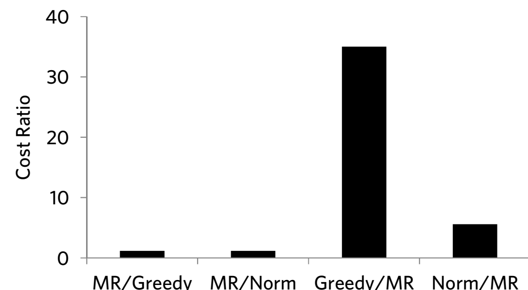

Figure 4: Worst Case Performance for Online Sharing Plan Selection

Table 1: Sharings Used in the Evaluations

S_1	USERS $\bowtie$ SOCNET (twitaholic)	S_9	FOURSQ $\bowtie$ TWEETS (checkoutcheckins)	S_{17}	USERS $\bowtie$ LOC (twittermap)
S_2	USERS $\bowtie$ TWEETS $\bowtie$ CURLOC (twellow)	S_{10}	HASHTAGS $\bowtie$ TWEETS (monitter)	S_{18}	USERS $\bowtie$ TWEETS $\bowtie$ PHOTOS $\bowtie$ CURLOC (twittermap)
S_3	USERS $\bowtie$ TWEETS $\bowtie$ URLS (tweetmeme)	S_{11}	FOURSQ $\bowtie$ USERS $\bowtie$ TWEETS $\bowtie$ CURLOC (arrivaltracker)	S_{19}	USERS $\bowtie$ TWEETS $\bowtie$ HASHTAGS $\bowtie$ CURLOC (hashtags.org)
S_4	USERS $\bowtie$ TWEETS $\bowtie$ URLS $\bowtie$ CURLOC (twitdom)	S_{12}	FOURSQ $\bowtie$ USER $\bowtie$ TWEETS (route)	S_{20}	USERS $\bowtie$ TWEETS $\bowtie$ HASHTAGS $\bowtie$ PHOTOS $\bowtie$ CURLOC (nearbytweets)
S_5	USERS $\bowtie$ TWEETS (tweetstats)	S_{13}	FOURSQ $\bowtie$ USER $\bowtie$ TWEETS $\bowtie$ LOC (locc.us)	S_{21}	USERS $\bowtie$ TWEETS $\bowtie$ FOURSQ $\bowtie$ PHOTOS $\bowtie$ CURLOC (nearbytweets)
S_6	TWEETS $\bowtie$ CURLOC (nearbytweets)	S_{14}	TWEETS $\bowtie$ LOC (locafollow)	S_{22}	FOURSQ $\bowtie$ CURLOC (nearbytweets)
S_7	URLS $\bowtie$ CURLOC (nearbyurls)	S_{15}	USERS $\bowtie$ LOC $\bowtie$ TWEETS $\bowtie$ CURLOC (twittervision)	S_{23}	PHOTOS $\bowtie$ CURLOC (twitxr)
S_8	TWEETS $\bowtie$ PHOTOS (twitpic)	S_{16}	FOURSQ $\bowtie$ USERS $\bowtie$ TWEETS $\bowtie$ SOCNET (yelp)	S_{24}	HASHTAGS $\bowtie$ CURLOC (nearbytweets)
				S_{25}	HASHTAGS $\bowtie$ USERS $\bowtie$ TWEETS (twistori)

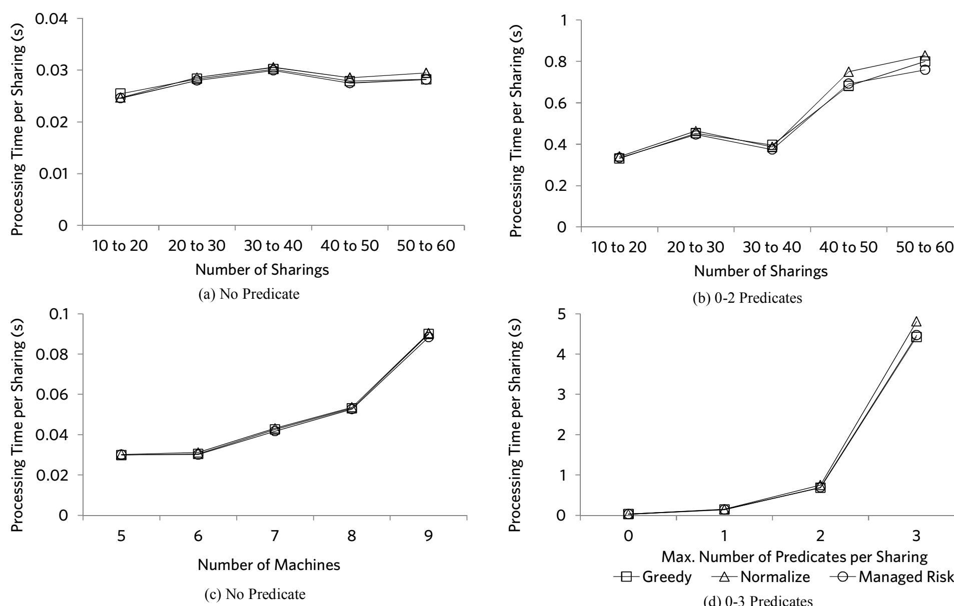

Figure 5: Running Time on Twitter Data

we can see, the sharing plans generated by MANAGEDRISK have much more stable qualities than those generated by the baselines, which validates the effectiveness of our strategy to take managed risks when selecting sharing plans.

6.2.2 *Efficiency and Scalability*

Figure 5 shows the running time of the sharings on Twitter data. In (a), sharings have no predicates and we vary the number of sharings in the sequence from 10 to 60. In (b), sharings in the sequence have 0-2 predicates. Half of the sharings in a sequence have no predicate, and the other half have 1 or 2 randomly generated predicates. In (c), the number of available machines varies from 5 to 9. In (d), the maximum number of predicates per sharing increases from 0 to 3. When the maximum number of predicates is 1, 2 or 3, half of the sharings have no predicates and the other half have random predicates. For example, when the maximum number of predicates is 3, 1/6 of the sharings in the sequence have 1 predicate, 1/6 have 2 predicates, 1/6 have 3 predicates, and the other 1/2 have no predicate.

The running time of the three algorithms are similar. Compared with Algorithm GREEDY, Algorithm NORMALIZE and Algorithm MANAGEDRISK need to maintain the counts and the regrets, respectively, of subexpressions that are contained in previous sharings but not used in existing sharing plans, but they do not add much overhead. The running time grows exponentially with number of predicates (Figure 5(d)) since the number of possible sharing plans grows exponentially. However, the processing time for 3 predicates is under 5 seconds, and as mentioned before, if the sharings are more complicated, heuristics can be applied to filter sharing plans [12].

To test Algorithm EXHAUSTIVE, we use a smaller sharing sequence that consists of 3-5 sharings, each of which has at most 1 predicate. The result is shown in Table 2, which is the average of 50 sharing sequences. The cost of MANAGEDRISK is never 3 times higher than that of EXHAUSTIVE for any sharing sequence. When each sharing sequence contains up to 10 sharings, EXHAUSTIVE cannot finish processing a test case of 50 sharing sequences within 48 hours, and thus we do not further test EXHAUSTIVE for scalability.

Figure 6 is the result of scalability test on synthetic data and randomly generated sharings, in order to test the scalability of the system at larger input scales. By default there's one fact table, 20 dimension tables, and the sequence contains 1000 sharings, each of which consists of a fact table and up to 7 dimension tables. (a)

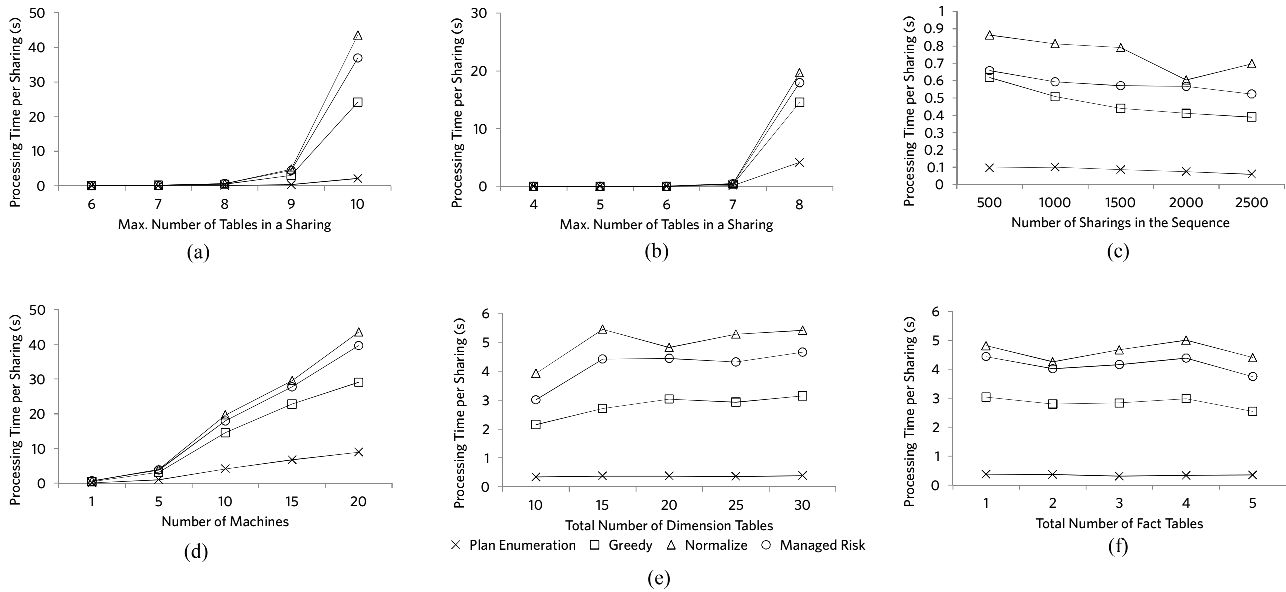

Figure 6: Running Time on Synthetic Data

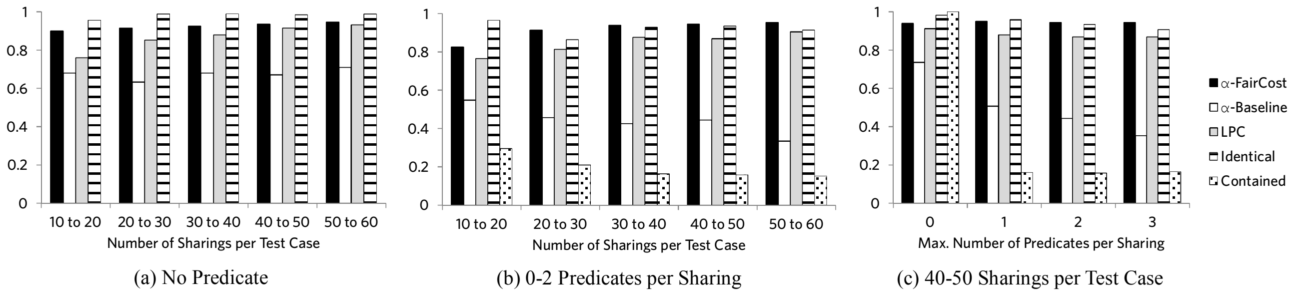

Figure 7: Fair Costing Results

Table 2: Performance of Algorithm EXHAUSTIVE

	MANAGEDRISK	EXHAUSTIVE
cost	1	0.84
time	1	2.18

shows the running time with increasing sharing size on a single machine (where, for example, sharing size of 10 means 1 fact table and 9 dimension tables). (b) shows the running time with increasing sharing size on 10 machines. The running time increases exponentially because the algorithms enumerates all plans, whose numbers increase exponentially wrt the sharing size. But it can be observed that the running time is reasonable for sharings of up to 10 tables on a single machine and sharings of up to 8 tables on multiple machines. And as said before, if the sharings are so big that exhaustive plan enumeration is not feasible, we can use heuristics to explore a manageable subset of plans.

(c) shows the running time wrt the number of sharings in the sequence on a single machine. The running time slightly declines with more sharings since sharings that come later in the sequence have higher probabilities of having occurred before and these sharings don't need to be processed. In (d) we increase the number of machines used from 1 to 20. More machines leads to more possible plans and thus longer processing times. In (e) and (f) we vary the total number of dimension tables and fact tables used in all sharings. These two paramters have little effect on the average processing time per sharing.

6.3 Fair Costing

Figure 7 shows the qualities of the two costing algorithms measured by the fairness criteria proposed in Section 5. *LPC*, *Identical* and *Contained* are for the baseline algorithm (for algorithm FAIRCOST their values are always 1). *LPC* is the percentage of sharings in the sequence whose costs assigned by the baseline algorithm are no more than their LPCs. *Identical* is the percentage of pairs of identical sharings whose costs assigned by the baseline algorithm are the same. *Contained* is the percentage of pairs of sharings such that the first sharing is contained in the second sharing and has a lower LPC than the second sharing, and the cost assigned to the first sharing is no more than that assigned to the second sharing. α is the minimum percentage of the saving of a sharing plan that is awarded to the corresponding sharing, as introduced in Section 5. Higher α, *LPC*, *Identical* and *Contained* values indicate a higher degree of fairness. Note that Figure 7(a) doesn't have "Contained" bars since those sharings have no predicate, and thus the value of "Contained" is always 1.

The results indicate that Algorithm FAIRCOST achieves much better fairness compared with the baseline algorithm.

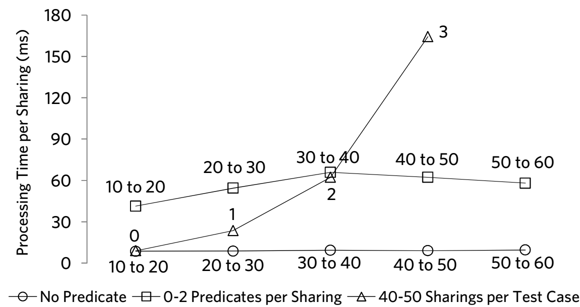

Figure 8: Running Time for Algorithm FairCosting

Figure 8 shows the average processing time of algorithm FAIRCOST for each sharing. ○ and □ test various numbers of sharings per test case, with no predicate (○) or up to 2 predicates per sharing (□); △ tests various numbers of predicates per sharing (from 0 to 3), with 40-50 sharings per test case. The results indicate that the processing time has no noticeable increase for later sharings in the sequence compared to earlier sharings, and that algorithm FAIRCOST is very efficient in calculating the cost of sharings, although the processing time increases fast with the number of predicates, because more predicates leads to a larger number of possible plans, hence more expensive computation of LPC. Heuristics can be applied to prune the search space if necessary.

7. CONCLUSIONS AND FUTURE WORK

This paper studies two problems in building a data market that enables the sharing of dynamic data specified by ad-hoc queries: how to design an online algorithm for selecting sharing plans, and how to fairly calculate the cost of each sharing plan. There are a few interesting future works, for example, whether it is feasible to change the plan of an existing sharing when a new sharing arrives, and how it effects the strategies for selecting sharing plans and costing the sharings; whether it is beneficial to create and maintain views that do not belong to any existing sharing plan (so that future sharings may reuse them), rather than reusing only those views created by existing sharing plans, and how to determine which views to create.

Acknowledgement: We thank Magdalena Balazinska and Dan Suciu for their insightful comments on the earlier versions of the work.

8. REFERENCES

[1] Gnip. http://gnip.com/.

[2] Infochimps. http://infochimps.com/.

[3] Microsoft Azure Marketplace. https://datamarket.azure.com.

[4] Public Data Sets on AWS. http://aws.amazon.com/publicdatasets/.

[5] Xignite. http://xignite.com/.

[6] D. Agrawal, A. El Abbadi, A. K. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *SIGMOD Conference*, pages 417–427, 1997.

[7] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, pages 496–505, 2000.

[8] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Materialized View and Index Selection Tool for Microsoft SQL Server 2000. In *SIGMOD Conference*, page 608, 2001.

[9] S. Al-Kiswany, H. Hacıgümüş, Z. Liu, and J. Sankaranarayanan. Cost Exploration of Data Sharings in the Cloud. In *EDBT*, pages 601–612, 2013.

[10] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. L. Perez. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD Conference*, pages 519–530, 2010.

[11] G. Candea, N. Polyzotis, and R. Vingralek. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *PVLDB*, 2(1):277–288, 2009.

[12] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS*, pages 34–43, 1998.

[13] S. Chaudhuri, M. Datar, and V. R. Narasayya. Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution. *IEEE Trans. Knowl. Data Eng.*, 16(11):1313–1323, 2004.

[14] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, pages 146–155, 1997.

[15] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing One Thousand Queries With One Stone. *PVLDB*, 5(6):526–537, 2012.

[16] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD Conference*, pages 383–394, 2005.

[17] V. Kantere, D. Dash, G. Gratsias, and A. Ailamaki. Predicting Cost Amortization for Query Services. In *SIGMOD Conference*, pages 325–336, 2011.

[18] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[19] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-Based Data Pricing. In *PODS*, pages 167–178, 2012.

[20] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Toward Practical Query Pricing with QueryMarket. In *SIGMOD Conference*, pages 613–624, 2013.

[21] C. Li, D. Y. Li, G. Miklau, and D. Suciu. A Theory of Pricing Private Data. In *ICDT*, pages 33–44, 2013.

[22] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *SIGMOD Conference*, pages 307–318, 2001.

[23] T. Nagle, J. Hogan, and J. Zale. *The Strategy and Tactics of Pricing: A Guide to Growing More Profitably*. Prentice Hall, 2010.

[24] T.-V.-A. Nguyen, S. Bimonte, L. d'Orazio, and J. Darmont. Cost Models for View Materialization in the Cloud. In *EDBT/ICDT Workshops*, pages 47–54, 2012.

[25] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 2011.

[26] M. Peterson. *An Introduction to Decision Theory*. Cambridge University Press, 2009.

[27] S. K. Rahimi and F. S. Haug. *Distributed Database Management Systems: A Practical Approach*. Wiley-IEEE Computer Society Pr, 2010.

[28] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *SIGMOD Conference*, pages 447–458, 1996.

[29] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *SIGMOD Conference*, pages 129–140, 2000.

[30] K. Schnaitter and N. Polyzotis. Semi-Automatic Index Tuning: Keeping DBAs in the Loop. *PVLDB*, 5(5):478–489, 2012.

[31] K. Schnaitter, N. Polyzotis, and L. Getoor. Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications. *PVLDB*, 2(1):1234–1245, 2009.

[32] T. K. Sellis. Multiple-Query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[33] T. K. Sellis and S. Ghosh. On the Multiple-Query Optimization Problem. *IEEE Trans. Knowl. Data Eng.*, 2(2):262–266, 1990.

[34] K. Shim, T. K. Sellis, and D. S. Nau. Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *Data Knowl. Eng.*, 12(2):197–222, 1994.

[35] T. J. Smith. *Pricing Strategy: Setting Price Levels, Managing Price Discounts and Establishing Price Structures*. Cengage Learning, 2011.

[36] P. Upadhyaya, M. Balazinska, and D. Suciu. How to Price Shared Optimizations in the Cloud. *PVLDB*, 5(6):562–573, 2012.

[37] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*, pages 101–110, 2000.