Cost Exploration of Data Sharings in the Cloud

Samer Al-Kiswany[2,1,#] Hakan Hacıgümüş[1] Ziyang Liu[1] Jagan Sankaranarayanan[1]
[1]NEC Labs America, Cupertino, CA [2]University of British Columbia
samera@ece.ubc.ca, {hakan,ziyang,jagan}@nec-labs.com

ABSTRACT

Enabling data sharing among mobile apps hosted in the same cloud infrastructure can provide a competitive advantage to the mobile apps by giving them access to rich information as well as increasing the revenue for the cloud provider. We introduce a costing tool that allows application owners (i.e., consumers) and the cloud service provider to assess the cost of a desired data sharing. The costing tool enables the consumers to effectively explore the cost space by choosing between alternative configurations of varying data qualities, specified by the staleness and the accuracy of the data sharing. In other words, staleness and accuracy requirements on the data sharing are used as levers for controlling costs. These capabilities are implemented in a What-if analysis tool, which has been integrated with a large data-sharing platform. We conducted extensive experiments on the integrated platform with a sharing ecosystem created around Twitter data and show the effectiveness of the results produced by the What-if tool.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Data Sharing*

General Terms

Algorithms

Keywords

Data Sharing, Mobile Cloud, SLA, Staleness, Accuracy, Pricing, Negotiation, Budget, Materialized Views, Pareto-optimal

1. INTRODUCTION

The cloud is hosting an ever increasing number of web and mobile applications in the same infrastructure. There is an incentive for apps to share information with one another as reliable access to rich information can spur new features. This can result in a much richer experience for their users as well as increased revenue for the cloud operator [22]. Sharing among apps can be enabled through data markets in the cloud. Such data markets already exist and are emerging [1, 2, 4].

As a motivating example, consider the *Tesco* store mobile app [3]. Tesco displays pictures and barcodes of its grocery products at subway stations. While the users are waiting for the metro, they can shop for groceries by simply scanning the barcodes using their mobile phones, and the purchases are delivered to their homes in a few hours. Tesco could benefit from data sharing in the cloud if it obtains access to the user's restaurant checkin information. The checkin information could either be a publicly available data set hosted by the cloud provider or possibly be created by another application such as Foursquare. The Tesco app could then recommend items to purchase based on the users' favorite cuisine types, which can be deduced by analyzing the checkin information.

We focus on the *costing* process for a *consumer* (e.g., Tesco app developer) who is interested in new data sets (e.g., checkin data) available through the sharing service offered by the provider. In our setup there are several *base* relations available for sharing. The consumer is interested in creating a new sharing, which he specifies as a transformation on the base relations. Although there are many ways of enabling sharing in the cloud [22], including API [1, 4], web service [2], and direct SQL access [26], a sharing in this work is enabled by the creation of a *materialized view*, which is defined by a set of transformations over the base relations. A discussion of the pros-and-cons of each of these methods is given in [22]. Note that it is not the goal of this paper to make a case for the importance, feasibility, or the value of data sharing in the cloud as well as its privacy, infrastructure, and economic implications as they are discussed elsewhere [8, 12, 22].

As the base relations are being constantly updated, the cloud provider is responsible for setting up the sharing and maintaining it. The consultation between the consumer and the provider starts as soon as the consumer has identified the base relations and the transformation he is interested in and wants to cost the sharing before committing to it.

The main goal in the paper is to present our "What-if" cost exploration tool that is designed to aid the consumer's cost assessment. The tool is an integral part of a large data-sharing platform, SMILE [22], that aims at providing a seamless, SLA-driven data sharing platform primarily for mobile apps. The What-if tool acts as a stand-in for the provider by answering the hypothetical sharing related questions from the consumers. The What-if estimation tool is fast enough in the sense that it allows for interactive querying by multiple consumers at the same time, and the cost estimates produced by it are close to real costs.

The consumer is concerned about the cost of the sharing and so

[#]Work done while author was at NEC Labs America

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy

we provide two levers for controlling the cost. First, the consumer can tolerate data that is not fresh up to a certain extent. For example, the Tesco app can stipulate that once a user checks into a restaurant, the information can be delayed by say, 60 seconds before it is delivered to it. This is referred to as the acceptable *staleness* of the sharing. Next, the consumer can tolerate some amount of missing data. For example, the Tesco app can specify that only 90% of the new checkin information needs to be delivered, as long as it reduces the cost. We refer to this as the acceptable *accuracy* of the sharing. In this work, we use staleness and accuracy to control the cost for the consumer.

The consumer wants to know from the provider how much it would cost for a sharing with some specified staleness of x and accuracy of y. The difficulty in answering this question comes from estimating the cost of these sharings quickly and ensuring that the cost estimates reasonably agree with the actual costs.

While staleness and accuracy are good levers to control cost, they can be intuitively difficult for the consumers to specify. It is not clear if most applications have rigid staleness and accuracy requirements, nor if there are bounds on both these values beyond which they render the sharing not very useful to the application. For example, it is not clear what is more suitable for the Tesco application – 90 seconds staleness and 90% accuracy, or 80 seconds staleness (better) and 80% accuracy (worse).

The most natural way a consumer would specify the requirements is using a cost budget. For example, the consumer can specify "What can I get for $z?" The difficulty in answering this question is in being able to provide the consumer with the appropriate set of staleness and accuracy configurations without overwhelming the consumer with too many answers. To that effect, the set of answers has to be both interesting to the consumer as well as different from one another in the answer set to provide the consumer with a range of options. The consumer can examine the set of answers for a certain budget and if not satisfied may pose subsequent questions.

In a mature sharing framework, there may be several existing sharings with new ones being added frequently. In this context, another opportunity to reduce the cost for a new consumer is by taking advantage of some of the commonalities of the new sharing with existing sharings in the system, not to mention that it also reduces the infrastructure cost for the provider by reducing duplicated work. For example, if another app wants to implement an alerting feature that informs users when their friends are nearby by creating a new sharing using the checkin information. The new sharing may benefit from its commonality (i.e., use of checkin data) with some of the existing sharings in the system (e.g., Tesco app). These savings can be passed along to new consumers making them more willing to commit to sharing. So, we consider the above cost estimation questions both with and without existing sharings in the system.

To our knowledge our work is the first systematic, generic approach for exploring the cost of a data sharing in cloud applications using staleness and accuracy to control costs. We make the following contributions during the course of the design and implementation of the What-if tool:

1. We design a "What-if" tool that helps consumers explore the space of staleness, accuracy and cost, based on either a specific requirement or a cost budget.

2. We propose a Pareto optimality based method that provides an interesting set of staleness and accuracy configurations to the consumers for data sharing.

3. We propose an algorithm to further reduce the cost for the consumers and the provider in the case of multiple sharings.

4. Experimental results show that the cost estimates produced by our What-if tool is within 10–30% of the actual execution costs when compared to an implementation of the sharing infrastructure with a real workload.

2. TWO NOTIONS OF "COST"

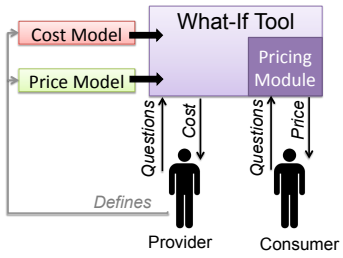

Figure 1: Producer and consumer's interaction with the What-if tool

For the provider, cost is the expense incurred to create and maintain the sharing, while for the consumer the cost is the amount paid for the sharing, which may include a profit for the provider. Typically, the latter is referred to as *price*, while the former is referred to as the provider's *cost*, or simply *cost*. So, all hypothetical questions posed by the consumer during the cost assessment, such as "What is the *price* of ...?", or "What can I buy with a *price* budget of ...?", are all based on *price*.

The problem of how to price a sharing is usually complicated as they are often driven by business priorities [11] or by other concerns, such as making the pricing fair to all consumers [12, 26]. Sharing creates expensive *artifacts* in the clouds, such as copies of base relations, or materialized views, which future sharings entering the system can make use of. This means that the *cost* incurred by the first sharing in the system can be quite high, which can get progressively less if subsequent consumers are interested in similar sharings. A pricing strategy that is proportional to cost is inherently *unfair* to the initial consumers. Recently, [12, 26] show how to amortize the consumer's cost by predicting the potential future use of these expensive artifacts. A consumer creating a new artifact in the system pays only a fraction of the cost to the provider, while the future sharings that use these artifacts pay the remaining. So, it is not even necessary that the price always be greater than the cost, as conventional wisdom would suggest. While, the pricing model in [12, 26] is applicable to our work, it is not our focus. Essentially, we design the What-if tool in a way that a pricing model can be plugged into the tool, which will then be used by the *pricing module* as shown in Figure 1.

The What-if tool we propose in the paper is a *costing* tool that will be used by the provider. The What-if tool along with the pricing module becomes a *pricing* tool which will be used by the consumer, which is captured in Figure 1. The provider defines appropriate cost and price models, which are used by the What-if tool. An implementation of a cost model is given in Section 5.2.

We later show in Section 7 that many *reasonable* pricing models can be incorporated into our What-if tool to make it a pricing tool. However, for the sake of simplicity, we describe the What-if as a costing tool and define a pricing module to be part of the What-if tool through which a consumer interacts. The pricing module converts *price* to *cost*, and vice-versa by discounting or adding the provider's *profit*. We will use *cost* and *price* interchangeably in the following context: When we describe the consumer's hypothetical questions during the cost assessment in terms of *cost budget of* $\$z$, we actually imply that the consumer specified a different price

budget, which the pricing module converted to a cost budget of $\$z$. We also describe scenarios involving direct interactions of the consumer with the What-if tool. In reality, however, all interactions between the consumer and the What-if tool is via the pricing module and the consumer always deals with *price*.

3. SHARINGS

A sharing is specified in terms of a set of transformations (select-project-join in our case) on the base relations. The sharing results in a *materialized view* (MV) for use by the consumer, which is created and maintained by the provider. Since the base relations are constantly updated, the MV lags behind the original data. The staleness requirements need to be specified as some applications need highly fresh data. If new records are inserted into the base relations at a high rate, it becomes expensive for the consumer to maintain the MV. So, some of the updates can be dropped up to a certain rate if the application permits.

The *staleness* captures the freshness of the data obtained by the consumer. A staleness of x seconds means "if there is an update to the shared data, the consumer should be able to see the update within x seconds". For example, in order to make timely recommendations, the Tesco app may get into an additional sharing to obtain the user's current location. The app may need to know the user's location within 30 seconds of entering a subway station as the wait for the metro is not more than a few minutes.

The *accuracy* regulates missing records (tuples) in the shared data. An accuracy of y means that "the number of missing tuples will be no more than a fraction of $1-y$ of the total number of update tuples". This criterion is intended to give the consumer flexibility in selecting a tradeoff between data quality and cost. As an example, the Tesco app can afford to lose say, up to 20% of the users' checkins since the app only computes coarse cuisine interests of the users.

A sharing with a staleness of x seconds and an accuracy of $y\%$ means that at any point in time the MV contains at least $y\%$ of the records of the actual data from x seconds ago. Note that staleness also makes the data *inaccurate* so to speak. While the staleness is a delay and the data will be delivered to the consumer at a later time, accuracy means that the dropped records will never be shown to the consumer.

Once the consumer is satisfied with the staleness, the accuracy and the cost of the sharing, the two parties (i.e., provider and consumer) enter into a Service Level Agreement (SLA), which specifies what is to be shared at what staleness and accuracy.

The consumer explores different configurations of staleness, accuracy and cost before entering into an SLA with the provider. This exploration process should be automated for the service provider, since the cloud may host a large number of applications and the provider cannot afford to answer each of them manually. Hence, the job of costing and answering all of the consumer's hypothetical questions is given to a "What-if" exploration tool, which can answer two common types of What-if questions.

1. Given the sharing I want, what is the cost for the staleness of x seconds and the accuracy of $y\%$?
2. Given the sharing I want, what configurations of staleness and accuracy can I get if I have a budget of z dollars?

Those consumers who know the specific staleness and accuracy requirements for their applications may pose the first question, while the second question will be posed by consumers who have limited budgets and may not know what they want.

The costing of these hypothetical questions can be performed under two different settings that differ from each other in their assumptions about the nature of interactions between the sharings in the system. First, we can cost sharings in isolation in the sense that we need not allow sharings to benefit from one another. This forms the first setting. However, one way of reducing the cost of the sharings is by taking advantage of the *commonality* with the sharings already present in the system, which forms the second setting. The commonality here refers to the *common subexpressions* [20, 23] and to merge them we apply a well-known strategy from traditional materialized view maintenance [18,20] and multiple query optimization [23]. Some of these savings can be passed on the consumer (through some pricing function such as in [12, 26]) which may entice the consumer to enter into an SLA with the provider, not to mention that this results in lower infrastructure cost for the provider by saving on duplicated work. We will examine the above two questions in isolation as well as accounting for the cost savings due to existing sharings.

Since the inter-relationship between staleness, accuracy and cost is not straightforward, the best way of selecting a suitable configuration is by allowing the consumer to choose from a list of configuration options. If the consumer did not find any of the configurations suitable, he may repose the question by specifying a different budget.

Given a fixed budget, there may be practically unlimited number of staleness and accuracy configurations that satisfy the budget. Considering the nature of the problem we adopt a Pareto optimality based approach [16, 17]. To prevent information overload, we generate evenly distributed samples on the Pareto frontier. The configurations are also Pareto-efficient for the given budget, which means that it is not possible to find any other configuration that is better in terms of both the staleness and the accuracy for the same budget. We use a Pareto frontier generation algorithm [16], which will be presented in Section 7.

4. SETUP

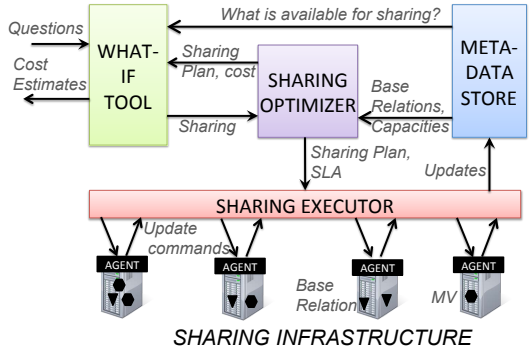

Figure 2: Block diagram of SMILE system with What-If tool

Our What-if tool is implemented as the cost assessment front-end of the SMILE sharing framework [22] (SMILE standing for Sharing MIddLEware). The interaction with the tool is via a user interface that enables the consumer to examine what is available for sharing as well as iteratively arrive at the desired staleness and accuracy. While the provider directly interacts with the tool and obtains cost estimates, the consumer interacts via a pricing module and obtains price estimates.

Figure 2 shows the block diagram of the SMILE system consisting of a front-end What-if tool, a *meta-data store* that maintains useful statistics on the base relations as well as the current state of the infrastructure for use by the *sharing optimizer*. The existing sharings in the system are maintained by the *sharing executor*.

Once the consumer has decided on the sharing, he starts posing a number of hypothetical questions to the What-if tool. The What-if tool queries the *sharing optimizer* module of SMILE, which generates a low cost *sharing plan* (similar to a query execution plan) that implements the sharing. The optimizer works akin to a database optimizer in the sense that it generates all the possible sharing plans that implement a sharing with a specified staleness and accuracy. The sharing optimizer uses the meta-data store to obtain statistics on the base data, including join selectivities, update rates, and the current available capacities on the machines in the infrastructure. Section 5 describes how the sharing optimizer generates *admissible* sharing plans as well as how it *costs* these sharing plans. The cost of a sharing not only includes the cost of resource consumption (i.e., infrastructure cost), but also the possible penalty the consumer is considering in case the staleness or accuracy requirements are violated.

We capture the interactions of the What-if tool and the sharing optimizer for the three hypothetical questions we detailed earlier.

1. In case the consumer specifies both the staleness and the accuracy, the What-if tool queries the sharing optimizer to obtain a low cost sharing plan, providing the cost of this plan as the cost estimate to the consumer. We provide an algorithm for this scenario in Section 6.

2. In case a cost budget of $\$z$ is specified, the What-if tool queries the sharing optimizer several times as it enumerates the two-dimensional configuration space of staleness and accuracy. At each step, it estimates the cost of a configuration and compares it against z. The end result is a set of configurations with an estimated cost of around z that are drawn from the Pareto frontier. We provide an algorithm for this scenario in Section 7.

3. In case the cost estimates have to take into account existing sharings in the system, the What-if tool first obtains all possible plans implementing the sharing. It then merges these plans one by one with the existing *global sharing plan*, which corresponds to the sharing plan of all existing sharings in the system. It chooses the merged global plan with the least estimated cost. We provide an algorithm in Section 8 that revisits the first two scenarios but also takes into account the potential savings due to commonalities with existing sharings.

Once the consumer and provider both agree on the staleness, accuracy and the cost, they enter into a Service Level Agreement (SLA), which may also specify a penalty component in case the system misses the SLA. The SLA along with an admissible sharing plan is given to the sharing executor which performs run time optimizations so that all the sharings in the system are always maintained at or below the specified staleness level.

The sharing executor is an implementation of an asynchronous view maintenance algorithm [21]. Our implementation is *lazy* by design in the sense that it determines, using a learning model, the most appropriate time to refresh a MV. The refresh is neither too early nor too late, but finishes just before a sharing is about to miss its staleness SLA. Each machine in the infrastructure runs an *agent* that communicates with the sharing executor via a pub/sub system (e.g., ActiveMQ). The agents send periodic messages to the sharing executor about the last modification timestamps of the base relations and MV. The sharing executor is aware of the staleness of a sharing, which is calculated as the difference between the *maximum* of the timestamps of all the base relations to that of the MV. The executor keeps track of which of the sharings will soon miss their staleness SLA, and hence schedules updates to be applied to the MVs so that their staleness is reduced. We provide a few more details on the sharing executor in Section 9.

5. SHARING PLAN GENERATION

The update mechanism of a sharing is implemented using a *sharing plan*, which is generated by a plan generation algorithm. A sharing plan is analogous to a query execution plan in that it is expressed in terms operators that transform the updates from the base relations of the sharing to the MV. The sharing plan is expressed using 5 operators implemented in the system, which are a) an operator to apply updates, b) copy updates between machines, c) join updates, d) merge updates and e) selectively drop tuples from updates. We will briefly describe some of the implementation details of these operators in Section 9 and provide an example below of a sharing plan that joins two base relations.

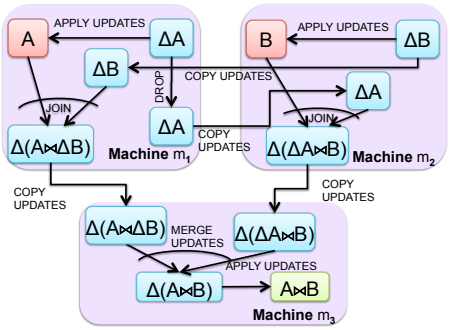

Figure 3: One possible sharing plan involving an in-place join of a base relation A on machine m_1 and B on machine m_2 such that the resulting MV $A \bowtie B$ is placed on machine m_3

EXAMPLE 1. *Figure 3 shows the sharing plan of a sharing S that performs a transformation $A \bowtie B$ on two base relations, A and B. The sharing plan is a DAG consisting of 13 vertices and 11 edges. The vertices are either base relations (e.g., A, B or its copies), MVs (e.g., $A \bowtie B$) or temporary views (e.g., $\Delta(\Delta A \bowtie B)$). ($\Delta A$ stands for updates applied to the base relation A.) The edges corresponds to operators that either apply, copy, merge, join or drop updates, to complete the transformation path from the base relations to the MV.*

Our sharing plan generation algorithm is based on the polynomial time heuristic solution developed for System-R [9] and its analogous distributed variant R* [15]. Given a sharing involving a join sequence R of base relations, our algorithm explores different ways of obtaining R by considering all placements of $R - a$ on any of the machines with available resources, with copies of the base relation a in R available on any machine. Any select, project, or drop operators specified in the sharing are handled using a *pushdown* heuristic [9]. The placement of plan operators on a machine depends on the available capacity of the machine. If there is no spare capacity (i.e., CPU, disk, network) in the set of machines available for a sharing, then the provider has to add additional machines to the infrastructure. This enumeration generates all the sharing plans that implement the sharing, however not all of them satisfy the constraints that we will develop in the remainder of this section. In particular, we concern ourselves with two key properties of a sharing plan, namely its critical time path and cost.

5.1 Critical Time Path

The *critical time path* of a sharing plan is the longest path in terms of seconds that represents the most time consuming data transformation path in the sharing plan. Note that the sharing plan is admissible only if the length of its critical time path is less than the desired staleness of the sharing, or else the system cannot maintain it. The sharing optimizer estimates the critical time path of a sharing plan, using a time cost model for each operator that can estimate the time taken for each operator given the size of the updates. Note that finding the longest path between two vertices on a general graph is an NP-hard problem, but sharing plans are DAGs, on which longest path calculation is tractable. The system implements the procedure CP(p) that takes a sharing plan p and outputs its critical time path in seconds. For example, in the sharing plan p shown in Figure 3, CP(p) computes the time taken along the longest transformation path from A or B to the MV $A \bowtie B$.

5.2 Cost Model

The cost of the sharing plan, expressed in dollars per month, is computed by the amount of CPU, network, and disk capacity consumed to keep the sharing at the desired staleness and accuracy. This can be expressed as the sum of *static* cost, representing an initial investment to setup the sharing, and a *dynamic* cost, which is the expense incurred to periodically move the updates.

Since static cost is sharing-independent, in the following we mainly discuss the dynamic cost associated with a sharing. The dynamic cost can be further divided into two categories: resource usage (e.g., CPU, disk, network) and penalty due to occasional SLA violations.

Resource Usage. There are existing analytical models that estimate the usage of various resources for maintaining a materialized view, based on update rate, join selectivity, data location, etc. (e.g., [19]). Furthermore, the resource usage should also vary with the staleness SLA of the sharing. When the required staleness is much longer than the critical time path, e.g., the critical time path is 1 second and the staleness requirement is 30 seconds, the service provider has much flexibility in deciding when to update the view. Specifically, given a new tuple to the base relations, the service provider can push it to the view immediately, or wait for as long as 29 seconds before pushing it. On the other hand, when the staleness becomes close to the critical time path, the service provider has much less flexibility, and since there are other sharings in the infrastructure, they may compete for resources such as database, network, CPU, etc., which may cause the sharings to miss their SLAs.

In order to reduce the negative interaction at low staleness values, the resources allocated to the sharing plan are over-provisioned by a factor inversely proportional to the required staleness. This simple strategy ensures that the negative interactions are mostly avoided, especially for low staleness values.

SLA Penalty. At low staleness values the natural fluctuations in the update rates may cause a sharing plan to miss the SLA. This is because the sharing plan estimates the critical time path using the average arrival rate, but in practice this is an over simplification as the updates frequently vary. So, we have to estimate how much of penalty may be incurred given the required staleness and accuracy, which also has to be factored into the cost. We estimate this by assuming a Poisson arrival of updates, and modeling the sharing plan as an M/M/1 queuing system. Given the arrival rate of each base relation, we can estimate the arrival rate of tuples in the view based on the selectivity of joins. The average service time of the M/M/1 queue corresponds to the most time consuming operator in the sharing plan.

For an M/M/1 queue with arrival rate λ and service rate μ, the percentage of items with sojourn time larger than s is

$$P(S > s) = e^{(\lambda-\mu)\cdot s}$$

Thus the dynamic cost of a sharing plan p with staleness s and accuracy a is calculated as

$$\text{COST}(p) = resCost(p) \cdot (1 + \frac{CP(p)}{s}) + e^{(\lambda\cdot a-\mu)\cdot s} \cdot pen_s \quad (1)$$

$resCost(p)$ is the cost of resource usage. As discussed before, to avoid SLA violation due to multiple sharings competing for resource, we over-provision the resource by a factor of $CP(p)/s$ where $CP(p)$ is the length of the critical time path of p. $e^{(\lambda\cdot a-\mu)\cdot s}$. pen_s is the estimated penalty of missing the staleness SLA due to higher-than-expected tuple arrival rate, where pen_s is the penalty of missing the staleness SLA for a single tuple.

6. SCENARIO I: BASIC USE CASE

In this section we consider the following simple question, *Given a sharing S with a specific staleness and accuracy, how much does it cost?* To obtain the cost of implementing S, the What-if tool generates all sharing plans for S and then chooses the cheapest plan among them that satisfies both the staleness and accuracy requirements. This is shown in Algorithm 1 given below.

Algorithm 1 sub GENERATESHARINGPLAN(S, t, a)

```
1: /* S is a sharing, t is staleness in sec and a is accuracy */
2: Generate all possible plans P of S with accuracy a
3: Choose p ∈ P such that:
4:     a. CP(p) ≤ s /* Critical time path of p ≤ s */
5:     b. COST(p, s, a) is minimum
6: return p
```

The algorithm takes as input a sharing S, the desired staleness t and accuracy a and produces the cheapest cost plan p that implements S as well as satisfying the staleness and accuracy requirements. It starts by generating all possible plans $\mathcal{P}$ for S with an accuracy of a. The transformation specified in the sharing can involve joining different base relations on different machines. The sharing plans in $\mathcal{P}$ denote the different ways in which joins can be ordered as well as all possible placements of the intermediate results on machines with available capacity. For each of the plans we examine its critical time path and cost.

The algorithm chooses a plan p from $\mathcal{P}$ to be the sharing plan for S if it satisfies the following criteria: First, p is *admissible* in the sense that its critical time path CP(p) should be less than the specified staleness t. Second, p has the lowest cost among all the admissible plans in $\mathcal{P}$.

Note that this scenario estimates the cost of implementing S without considering its *commonalities* with other sharings in the system. We will show later in Section 8, that there can be potential savings in cost for S due to other sharings present in the system.

7. SCENARIO II: WHAT CAN A $Z COST BUDGET BUY?

The previous scenario dealt with the simple case where the consumer requires a specific staleness and accuracy on the sharing. In reality, consumers do not have such a specific preference and hence a What-if tool that only answers this question may not be very useful in practice. In many cases, applications can tolerate a range of staleness and accuracy *configurations*. So choosing an appropriate

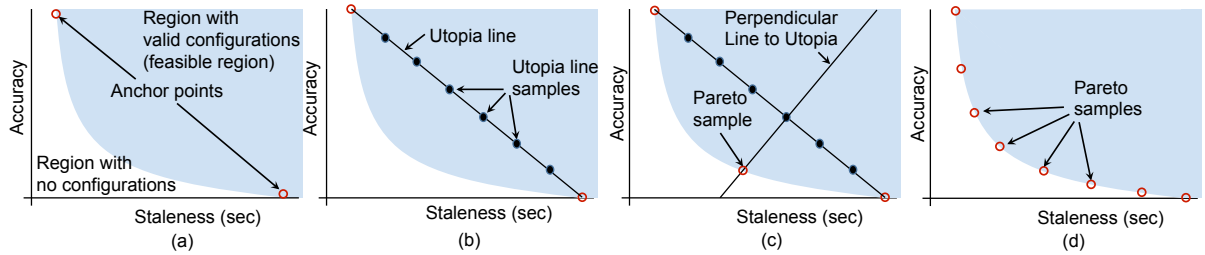

Figure 4: Figure shows the working of the Pareto sample generation algorithm

configuration is driven by a budget constraint. In other words, the consumer suggests a budget that he is willing to spend and the system presents a number of configurations that fit the budget. Hence, this scenario focuses on a consumer asking: For a given sharing, *what staleness and accuracy can a cost budget of* $\$z$ *buy?*

Answering this question is significantly more complex, since presenting all the plans less than a budget of z is not a feasible strategy. First of all, there may be too many possible (staleness, accuracy) configurations that fit the given budget, as both staleness and accuracy can take up *continuous* values, which causes an overload of information. Second, the consumer is usually not interested seeing a (staleness, accuracy) configuration that is dominated by another configuration (i.e., either with strictly better staleness and no-worse accuracy or vise versa). The non-dominated configurations form the *Pareto frontier* of the solution space. Thus we aim to generate a few sample configurations from the Pareto frontier. These samples should be diverse and represent the different scenarios, so that the consumer sees a wide range of options.

We generate these equi-spaced Pareto samples on the frontier by adapting the normalized normal constraint approach [16]. The What-if tool takes as input a sharing S and a budget z, and generates k configurations as answers such that they are not dominated and their cost is no more than $\$z$. Algorithm 2 is a divide and conquer based approach to generate equi-spaced Pareto samples. The algorithm first computes two extreme configurations on the Pareto frontier. The first one has minimum possible staleness (i.e., a configuration that has the smallest staleness over all configurations that satisfy the budget), and the second one has maximum possible accuracy (e.g., 100%). All other configurations on the Pareto frontier has staleness and accuracy values that are contained by these two extreme configurations. Then, it draws a straight line between these two configurations and evenly selects points on the line. Since these points represent configurations that may be dominated (i.e., not necessarily on the Pareto frontier), it performs binary searches based on these points to find Pareto-optimal configurations. The details of the algorithm are shown in Algorithm 2.

Algorithm 2 sub GENERATEPARETOSAMPLE(S, z)

```
1: /* S is a sharing, and z is the budget */
2: PP = ∅ /* set of Pareto points */
3: A = set of anchor points
4: L = CONSTRUCTUTOPIALINE(A)
5: U = GETUTOPIASAMPLES(L)
6: for u ∈ U do
7:    <r_high, r_low> = GETPERPLINEENDPOINTS(u, L)
8:    r_pareto = LINEBINARYSEARCH(S, r_high, r_low, z)
9:    PP = PP ∪ r_pareto
10: end for
11: PP = FILTERPARETOCANDIDATES(PP)
12: return PP
```

1. (line 3) We find the two extreme configurations with minimum staleness and maximum accuracy, respectively, that satisfy the budget. These two points are referred to as the *anchor* points (Figure 4a) and define the search region. The search region is bounded by the x axis, the y axis, a horizontal line corresponding to the accuracy when staleness is minimized, and a vertical line corresponding to the staleness when accuracy is maximized (i.e., 100%).
2. (line 4) We connect the two anchor points with a straight line called Utopia line (Figure 4b).
3. (line 5) We evenly select $k-2$ points on the Utopia line, where k is the number of Pareto-optimal configurations we would like to generate (Figure 4b).
4. (lines 6–10) For each of the sample points on the utopia line, we generate a line perpendicular to the utopia line crossing at the sample point. Each such line contains a Pareto-optimal configuration. For example, the perpendicular line in Figure 4c extends from the region of valid configurations down to a region with no valid configurations, crossing the frontier separating these two regions. The crossing point represents the Pareto-optimal configuration.
5. (line 8) We perform a binary search on this perpendicular line in the range from the point where the perpendicular line intersects with either x- or y-axis to the point where the perpendicular line intersects with upper or right side of the search region, to find the Pareto-optimal configuration on this perpendicular line (Figure 4d).
6. (line 11) We examine all the generated points and eliminate those points that are dominated by another point in PP.

Since the feasible region of our problem is a convex set, the configurations found by the above procedure are guaranteed to be Pareto-optimal configurations [16].

Algorithm 3 sub LINEBINARYSEARCH(S, r_{high}, r_{low}, z)

```
1: /* S is a sharing, r_high and r_low are two end-points of the line,
      and z is the budget */
2: r_mid = r_high
3: r_mid-old = r_low
4: while GEOMETRICDISTANCE(r_mid-old, r_mid) > ε do
5:    r_mid-old = r_mid
6:    r_mid = geometric middle of r_high and r_low
7:    p_r = GENERATESHARINGPLAN(S, r_mid.stl, r_mid.acc)
8:    if p_r = ∅ or COST(p_r, r_mid.stl, r_mid.acc) > z then
9:       r_low = r_mid
10:   else
11:      r_high = r_mid
12:   end if
13: end while
14: return r_mid
```

We describe the binary search in Algorithm 3 that finds a Pareto-

optimal configuration. The configuration's distance from the frontier is bounded by a small constant $\epsilon \geq 0$.

1. The algorithm starts by generating a middle point r_{mid} (line 6) between the two points (r_{high}, r_{low}) bounding the search line. Initially, r_{high} is the point where the perpendicular line intersects the upper or the right side of the search region, and r_{low} is the point where the perpendicular line intersects with either x- or y-axis. Since the feasibility region is convex, r_{high} must be a feasible point where r_{low} must be an infeasible point.
2. (line 7) generates the cheapest plan using GENERATESHARINGPLAN given in Algorithm 1. $r_{mid}.stl$, $r_{mid}.acc$ refers to the staleness and accuracy of the configuration denoted by r_{mid}.
3. (lines 8–12) If no plan exists for the configuration denoted by r_{mid} or if its budget is greater than z, then the search continues in the upper line segment, else the search proceeds to the lower line segment.
4. The algorithm continues until the distance between (r_{high}, r_{low}) is less than ϵ.

Finally, the What-if tool we presented here can be made into a pricing tool by replacing the cost model with a price model given by the PRICE function. As mentioned before, the price function can be complex as it is usually driven by business priorities and considerations. If the feasible region upon replacing the COST function with the PRICE is still a convex set, the algorithm will work with no further modifications. On the other hand, if the feasible region becomes a non-convex set, then the perpendicular line can intersect the boundary of the feasible region at multiple points and our binary search method will have to be modified. We also need to perform a final filtering step to remove non-Pareto solutions from the result set using the method detailed in [16].

8. SCENARIO III. COST SAVINGS DUE TO EXISTING SHARINGS

In this section we revisit the previous two scenarios but take into account sharings already present in the system. Suppose that we want to add a new sharing S in the system. S could benefit from having *commonalities* with existing sharings in the system. The commonalities manifest themselves as common expression between the sharing plans of the existing sharing and that of S. Potential savings in costs can be realized if these expressions are made common between the existing and the new sharing plans. This results in part of the cost being amortized across multiple consumers, leading to savings for the consumer interested in S. Taking advantage of these commonalities also reduces the cost for the provider by improving resource utilization.

Although our idea of merging commonalities in sharing plans is similar as merging common subexpressions in concurrent running query execution plans [23], there are two main differences. First, our infrastructure contains multiple servers and the cost of moving the data across the servers has to be considered. Second, instead of optimizing multiple queries at the same time as in [23], we only optimize a single sharing given the global plan of the existing sharings. In other words, our goal is to find a plan for the new sharing that has the lowest cost when combined with existing sharings, and we do not modify the plans of existing sharings. This ensures that existing sharings are serviced with no interruption. And because of this, our problem has a much smaller search space than the problem in [23], and we can afford to search for the best sharing plan rather than relying on some heuristic methods.

As one can easily imagine, the best plan for a sharing generated by Algorithm 1 may no longer be the best when combining with the existing global plan, since another plan that costs more individually may have more commonality with the existing global plan, thus the combined cost is lower. Next, we illustrate that given a specific sharing plan p, how to plug it into the existing global plan GP and take advantage of the commonalities.

A sharing plan can be represented as a DAG, where the top level nodes represent base relations and a single bottom level node represents the destination (i.e., MV). When we make use of the commonalities and feed the tuples from the global plan GP to an operator o in the sharing plan p, the nodes in p that leads to o may be removed. For example, in Figure 5, e is an operator in the global plan GP, and o is an operator in the plan p of the new sharing. If the output of e is the same as the input of o (i.e., commonality), we may "plumb" o into GP by making operator e feed operator o. In this way, any operator in p above o that is no longer needed can be removed, which saves the cost. On the other hand, it also incurs a new cost of moving the output of e to the machine that contains o (if e and o are on different machines). Thus such "plumbing" may either increase or decrease the total cost.

Note that different plumbing options are not independent. Suppose in plan p, operator o's predecessor is o'. Both o and o' may be plumbed to the global plan; but if we plumb o, o' may be subsequently removed, and thus plumbing o' is no longer an option. Therefore we cannot check the possible plumbings in an arbitrary order. Instead, either a top-down approach or a bottom-up approach can guarantee to identify the optimal set of plumbings. Next we illustrate the bottom-up approach.

The key procedure is PLUMBANDCOSTOPERATOR. It is invoked in Algorithm 4 on the root node of plan p (i.e., MV), and is illustrated in Algorithm 5, where it recursively invokes itself on other operators of p. Procedure PLUMBANDCOSTOPERATOR computes the best way of realizing operator o, by possibly making use of the global plan.

The idea is that, if o can be plumbed to the global plan, then one option to realize o is to make this plumbing. Other options are to not plumb o, then the input of o needs to come from the predecessors of o in plan p. To evaluate which option is the best, we recursively invokes procedure PLUMBANDCOSTOPERATOR on o's predecessors, and compute what is the best way of realizing each of o's predecessors. If an operator o has no predecessor (i.e., it directly operates on the source table), then there are only two options for o: plumb it to the global plan (if possible), or run o on the source table. Next we explain Algorithm 4 and Algorithm 5 in details.

Algorithm 4 sub PLUMBPLAN(p, t, a)

```
1: /* p is a sharing plan of S of accuracy a, staleness t, GP current
   global sharing plan */
2: GP_new = GP
3: PLUMBANDCOSTOPERATOR(GP_new, ROOT(p))
4: if all sharings in GP_new are still admissible then
5:     return GP_new
6: else
7:     return ∅
8: end if
```

Algorithm 4 is the main procedure for computing the lowest cost of integrating a sharing plan p with the global plan.

1. (line 3) Starting with the root of p, the algorithm tries to plumb each operator in p with GP by calling PLUMBANDCOSTOPERATOR.

2. (line 4) Ensure that the new global plan meets the staleness requirement of all sharings already present in the system as well as S.

Algorithm 5 sub PLUMBANDCOSTOPERATOR(GP, o)

```
1: /* GP is existing sharing plan, o is operator to plumb */
2: E = Set of identical operators to o in GP
3: Choose e ∈ E such that plumbing o with e is cheapest
4: plmbCst = cost of plumbing e with o
5: upCst = OPERATORCOST(O)
6: for o' ∈ all upstream operators of o do
7:     upCst += PLUMBANDCOSTOPERATOR(GP, o')
8: end for
9: /* plumb here vs. up */
10: if upCst < plmbCst then
11:     GP = GP ∪ o
12:     return upCst
13: else
14:     GP = PLUMB(GP, o, e)
15:     return plmbCst
16: end if
```

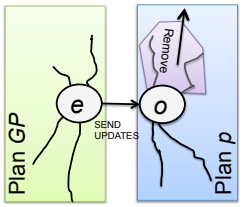

Figure 5: Plumbing of common expressions, e in a new sharing plan p and o in the current global plan GP

Algorithm 5 recursively calls procedure PLUMBANDCOSTOPERATOR on nodes in plan p to find the optimal cost of realizing each operator in p, which are ultimately used to calculate the optimal plumbing that leads to the lowest cost of the root operator of p.

1. (lines 2–4) First, we find a common expression e in GP, such that plumbing o and e results in maximum savings (i.e., least cost) among all possible plumbings involving o. The cost of this plumbing is $plmbCst$.
2. (lines 5–8) Next, we have to decide if it is beneficial to plumb e and o, or perform the plumbing at the predecessors of o. Hence, we compute the cost of retaining o (i.e., OPERATORCOST(o)), while recursively finding plumbings at the predecessors that would produce the maximum savings. We recursively call the PLUMBANDCOSTOPERATOR() method and the maximum savings from not plumbing here at o is given in $upCst$.
3. (lines 10–15) The algorithm compares the saving in plumbing e and o with the plumbing upstream. If it is cheaper to plumb upstream (i.e., $upCst < plmbCst$), o is added to GP else e is plumbed with o, in which case o and all its up stream operators are removed.

We can now implement the two scenarios as before by a small modification to GENERATESHARINGPLAN which both the scenarios use. The COST function invocation in Algorithm 1 is replaced to be the difference between the cost of GP_{new} and GP. With this small modification both scenarios will now take into consideration the savings from existing sharings in the system, which can be quite considerable as our experimental results show in the next section.

9. EXPERIMENTS

Our primary experimental setup creates a sharing ecosystem around Twitter dataset. We collected tweets from a *gardenhose* stream, which is a 10% sampling of all the tweets in Twitter, for a six month period starting from September 2010. The tweets obtained were unpacked into several base relations. Three of them are used in the experiments, including the information about the user (i.e., `users` relation), the tweets (i.e., `tweets` relation), and the current location of the user (i.e., `curloc` relation). The sharing in the evaluation is specified as equijoins of these base relations.

We populate our base relations by processing 7 million tweets. In the experiments, we replay the incoming tweets at different rates and estimate the rate of updates on the other base relations using a probabilistic method that is dependent on the number of tweets already injected into the system. Hence, we can precisely control the rate of updates on the `tweets` relation, and have a coarser control over the rates of updates on the other two base relations.

Applying, copying and merging update operators are based on their standard interpretations. Our join operator performs a *compensation* [28] algorithm as it joins *asynchronous* base relations (i.e., asynchronous as updates on sources can happen independently). A drop operator is implemented using the technique in [10] that pushes down the operator to one of the base relations across a join with *skewed* keys. Due to the unique skewed nature of the join keys in our evaluation setup, a simpler drop operator would suffice here, which is described next.

In all the sharings evaluated in this section, the joins involving base relations either happen over common tweets ids or user ids. Among the unique users in the 7 million tweets we processed, the number of tweets the unique users have sent, and the number of times these users have posted their current location, both follow *long-tailed* distributions. For example, in the `tweets` relation containing 7 million users, 98% of these users have sent less than 10 tweets (with 76% sending one or two only). Given the long tailed distribution of the user ids, our drop operator with a drop rate of f can be implemented using a coin flipping method that uniformly samples based on a bias proportional to f.

We use Amazon EC2 pricing (as of Oct 2012)[1] in order to cost sharings for the provider. Our machines are assumed to be equivalent to large Linux instances, which costs $0.32/hour. For the network cost, we assume that the instances are in different *availability zone* but in the same *region*, which has a transfer cost of $0.12 per GB. For storage, we use *EBS* storage at $0.125 per GB per month.

Finally, note that all the interactions with the What-if tool in this section are from a consumer's perspective and in terms of *cost*.

9.1 Base Case: Varying Single Dimension

In the base case, we evaluate the effect of staleness, accuracy and the shared data's update rate on the sharing cost, as well as the accuracy of our cost model. We use two machines and one base relation. The first machine hosts the base relation (`users`), while the sharing specifies that a copy of `users` be maintained on the second machine. In other words, the sharing involves no transformation of the base relation.

Figure 6a shows the What-if cost estimation for maintaining the sharing with varying staleness between 5 and 1000 seconds, while keeping accuracy fixed at 1 (drop rate defined to be is 1 - accuracy, hence a drop rate of 0) and an update rate at 100 tuples/second on

[1] `http://aws.amazon.com/ec2/pricing/`

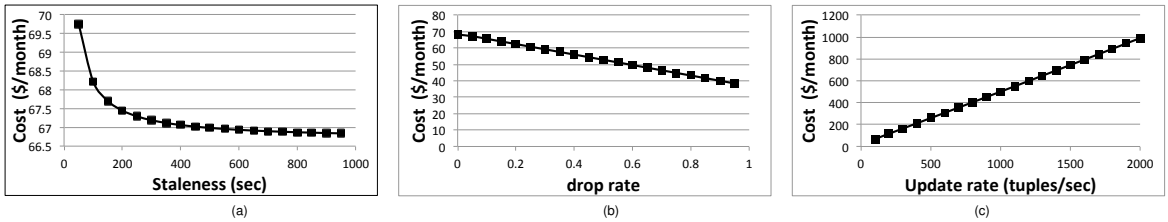

Figure 6: The effect of a) staleness, b) accuracy, and c) update rate on the cost estimated by the What-if tool for a simple sharing

the `users` relation. The curve shows that for small staleness values, the estimated cost of maintaining the sharing increases quite steeply. As the staleness of the sharing becomes close to its critical time path, any negative interaction with other sharings in the infrastructure may cause it to miss the SLA. In other words, there is very little *slack* time for the sharing executor to account for negative interactions and it is remedied by over provisioning resources. Hence, the sharing executor allocates more infrastructure resources (i.e., CPU, network and disk) to the sharing at these low staleness values resulting in increased estimated cost. Further, the lower the staleness the higher the probability of staleness SLA violations due to data fluctuations, which also increases the cost.

Next, Figure 6b shows the What-if cost estimation for maintaining the sharing for increasing drop rate (i.e., decreasing accuracy), while keeping the staleness fixed at 20 seconds and the rate of updates fixed at 100 tuples/second. The cost of sharing reduces linearly as the drop rate increases, as now the amount of tuples pushed through the sharing plan is proportionally fewer as the drop rate increases. Note that there are some additional savings at lower data rates from being able to update the MV in fewer batches, as well as lower probability of missing the SLA, but they do not show up significantly in the figure.

Finally, Figure 6c shows the What-if cost estimation when varying the update rate on the `users` relation between 50 and 2,000 tuples/second, while keeping the staleness fixed at 20 seconds and the drop rate at 0. As the update rate increases the sharing cost increases proportionally, which is because more tuples are processed through the sharing plan. As the staleness of 20 seconds is sufficiently greater than the critical time path even for a sharing plan for high update rates, the estimated cost linearly increases.

Next, we verify if the cost estimates generated by the What-if tool reflect the actual cost of maintaining the sharing in the SMILE system as well as the penalties due to the SLA violations. Our setup consists of two machines with the first machine hosting the `users` relation and the second machine hosting a (possibly staled) copy of the `users` relation. We then measure the running cost incurred by the sharing executor as it keeps the sharing at the desired level of staleness and accuracy. The SMILE system computes the running cost in dollars using a snapshot process, which is an independent auditing module that periodically tallies up the infrastructure cost spent by the sharing executor in the last time period since the previous invocation of the snapshot module. We use this to calculate the monthly cost of maintaining the sharing. We sampled four points from each of the curves in Figure 6 and provided them as the SLA to the sharing executor. We then measured the real cost by running each experiment for about 30 minutes using updates from a Twitter traffic generator at the update rates specified above. The results shown in Figure 7 show both the estimated cost as well as the real running costs.

Figure 7a shows the comparison between the estimated and real cost for varying values of the drop rate. The relative errors between the real and estimated cost for this setup range from 10–22%. Figure 7b shows the comparison results for varying staleness values, and Figure 7c shows the comparison results for varying update rates. In both these figures, we can see that the estimated and real cost curves show a similar trend, but the estimated error for both these cases is around 5–30%. The relative error between the estimated and the real cost can be mainly attributed to the following two reasons. First, What-if cost estimates are based on the estimated time of each operator in the sharing plan, while the real costs in SMILE are based on the actual running time of each operator. Second, it is hard to ensure a fine-grained control over the update rate of the `users` relation, which results in estimation errors.

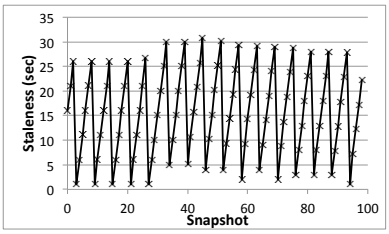

Figure 8: The measured staleness of the sharing as observed by the snapshot module

Finally, Figure 8 illustrates the actual observed staleness of the sharing. The setup for this experiment is same as before except that the staleness SLA is set to 35 seconds with a rate of 100 tuples/second update on the `users` relation. As can be seen from the figure, the observed staleness has large fluctuations. This is because the sharing executor waits until the most opportune moment (almost 30 second staleness) to refresh the MV. Note that the MV is updated just before it exceeds the SLA staleness of 35 seconds, thus illustrating the workings of the sharing executor.

9.2 Cost Function Complexity

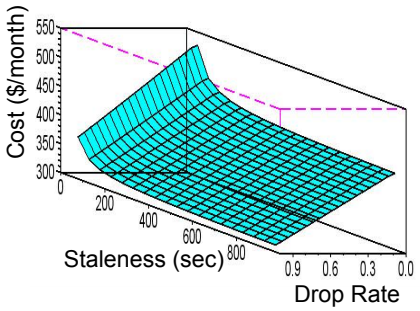

Figure 9: Estimated cost surface for `users` ⋈ `tweets` ⋈ `curloc` for varying staleness and accuracy values

To demonstrate the complexity of the costing problem, we consider the first scenario (detailed in Section 6). In this first scenario, the consumer specifies a sharing along with the desired staleness and accuracy. The What-if tool produces the estimated cost of maintaining such a sharing in the SMILE system. For this evaluation, we use

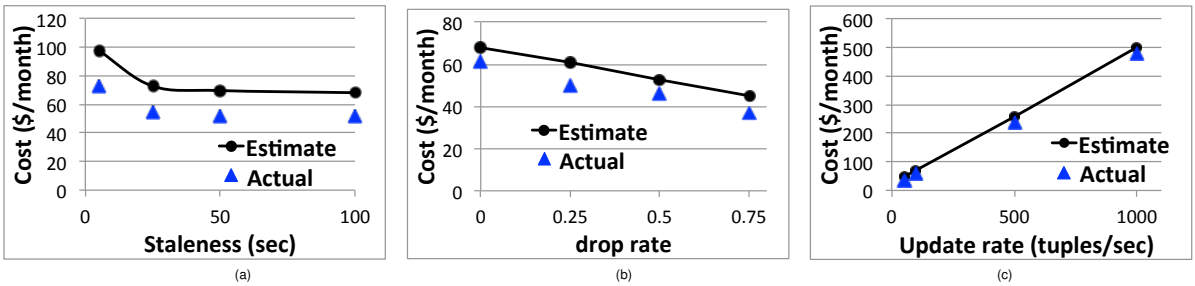

Figure 7: The effect of varying a) staleness, b) accuracy, and c) update rate for a simple sharing on the actual running cost

three base relations namely `tweets`, `users`, and `curloc`, such that the sharing is specified by `tweets` ⋈ `users` ⋈ `curloc`. The update rates on `tweets`, `users`, and `curloc` were 1000, 200, and 110 tuples/second, respectively. We varied the staleness and the drop rate between 5 and 1000 seconds, and between 0 and 1.0, respectively. For each pair of staleness and drop rate we invoke the What-if tool which produces a cost estimate.

Figure 9 shows the resulting estimated cost surface for the sharing. The figure shows the complexity of the staleness-accuracy-cost 3D space and motivates the need for an exploration tool to help the consumer navigate this space. Our What-if tool distills this space into a few interesting sample points that can be used by the consumer.

9.3 Scenario II: What can a Cost Budget Buy?

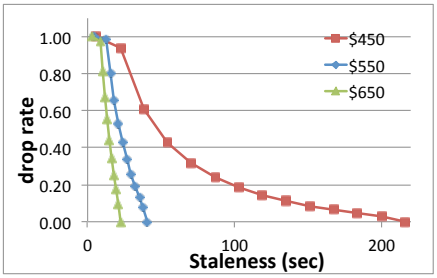

Figure 10: Pareto curves for cost budgets of $450, $550 and $650 for a sharing denoted by `tweets` ⋈ `users` ⋈ `curloc`

In this scenario, the consumer specifies a sharing and a price budget which is converted to a cost budget $\$z$ by the pricing module. The What-if tool produces a set of staleness and accuracy configurations denoting the Pareto curve, such that their estimated cost is less than z. Our setup consists of three base relations: `tweets`, `users`, and `curloc` with update rates of 1000, 200 and 110 tuples/second, respectively. The sharing is denoted by `tweets` ⋈ `users` ⋈ `curloc`. The setup uses four machines. The consumer first specifies a cost budget of $450/month and subsequently increases his budget to $550/month and later to $650/month.

Figure 10 shows the different staleness and accuracy configurations provided by our What-if tool for the three budgets. For the same budget the tool presents a set of Pareto-optimal configurations. For instance, with a cost budget of $450/month the consumer can select between getting an accuracy of 1.0 (i.e. 0 drop rate) with staleness of 216 seconds, or can opt to getting an accuracy of 0.75 (0.25 drop rate) for a better staleness of 87 seconds. Further, the figure shows that when the cost budget is increased, the staleness and accuracy values improve quite significantly. For example, suppose we draw an horizontal line denoting a drop rate of 0.25 and examine the points where it intersects the three curves. The figure shows that when the budget increases from $450/month to $550/month the staleness decreased from 87 seconds to 30 seconds. When the budget was further increased to $650/month, the staleness further dropped to 18 seconds. This experiment shows that the consumer is able to progressively get better staleness and accuracy values by increasing his budget.

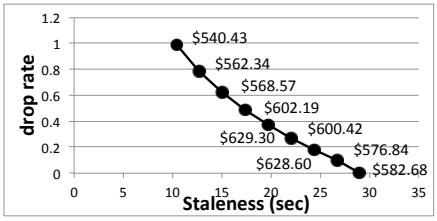

Figure 11: Pareto curves showing actual running cost given a budget of $600/month for a sharing denoted by `tweets` ⋈ `users` ⋈ `curloc`

Next, to verify if the cost estimates align with the actual running costs, we created the above setup on the SMILE system given a budget of $600/month. We generated a set of staleness and accuracy configurations using the What-if tool and obtained the real costs from running these on SMILE. Figure 11 shows that the running cost of each of these setup overlaid near each Pareto point on the curve. As we can see that the running costs for all the points are close to $600 with a relative error of less than 7%. This validates our What-if tool for this case.

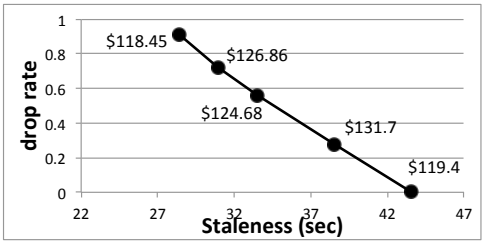

Figure 12: Pareto curve showing actual running cost for a sharing given by the join of `curloc` and `users` given an incoming gardenhose input and a query workload of 25 queries/second on the MV

We test if our cost estimates align with the real cost values for a more realistic setup involving a live update traffic and a read query workload being applied to the MV. Our setup consists of four machines, such that the sharing is denoted by `users` ⋈ `curloc`. The updates to `users` and `curloc` was from a *gardenhose* stream from Twitter which has large fluctuations in the arrival rate. The average update rate on `users` and `curloc` was observed to be 20, and 10 tuples/second, respectively. We applied a workload of 25 queries/second on the MV, thus consuming some of the available

resources on the machine hosting the MV. Figure 12 shows the resulting Pareto curve for $150/month specified by the consumer. The actual costs overlaid around the Pareto curve show an error rate of 15–30%.

Finally, we observed that the sharing executor did not miss the staleness SLA even once during the course of the experiments, each lasting up to 30 minutes. Moreover, we examined if the system missed the accuracy SLA by measuring the drop rate for two data points from Figure 12. We found that the recorded accuracy was sufficiently higher than the SLA accuracy.

9.4 Scenario III: What can a Cost Budget Buy, Given Existing Sharings?

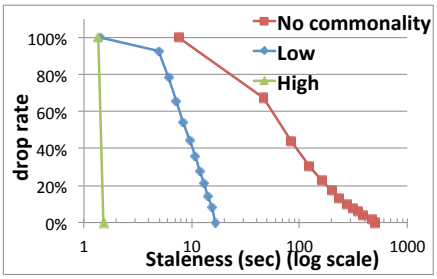

Figure 13: Pareto curves for a budget of $450 for the sharing denoted by `tweets` ⋈ `users` ⋈ `curloc` in three setups: no, low, and high commonality with exiting sharings.

In this scenario, the consumer specifies a cost budget as before but the What-if tool provides answers taking into account the potential savings in cost due to existing sharings present in the system. Our setup consists of a sharing S_1 denoted by `tweets` ⋈ `users` ⋈ `curloc`, with an update rate of 1000, 200 and 110 tuples/second on the `tweets`, `users`, and `curloc` relations, respectively. Another sharing S_2 is already present in the system. We examine the effects of commonality between S_1 and S_2 on the cost of S_1 by devising the following three cases.

In the *no* commonality case, S_2 has nothing in common with S_1, not even base relations. In the *low* commonality case, S_2 has two base relations – `users`, `curloc`, in common with S_1. S_2 produces a join subexpression `users` ⋈ `curloc`, which S_1 does not initially use. However, during the *plumbing* phase, the algorithm recognized the potential benefit in reusing this subexpression from S_2 and took advantage of it. For the *high* commonality case, S_2 is identical to S_1 but is present on different machine, the plumbing phase identifies this similarity and, instead of recomputing the sharing, it copies the updates on the final MV from S_2 to S_1 .

Figure 13 shows the What-if cost estimates given a budget of $450/month. It can be seen from the figure that there is a dramatic reduction in the staleness and accuracy values between the *low* and *no* commonality cases. The improvement is even more remarkable for the *high* commonality case. This experiment indicates that S_1 benefited significantly from the presence of S_2 in the system.

To demonstrate that the What-if estimates match with the actual running cost in the SMILE system, we implement the sharing in SMILE with a staleness of 30 seconds and a drop rate of 0.0. The staleness of S_2 for all the three cases was chosen to be 9 seconds. Figure 14 shows the cost comparisons for the maintaining S_1 in the presence of another sharing S_2. It can be seen that S_1 has a cost of almost $600/month for the *no* commonality case, which reduces to about $112/month for the *low* commonality case. For the *high* commonality case the cost plummets to about $20/month. This shows that dramatic reduction in costs is possible by taking

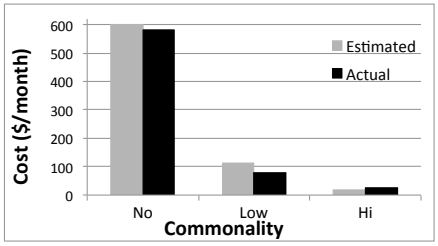

Figure 14: The estimated and actual cost of S_1 in the presence of S_2 for the no, low and high commonality cases

advantage of the commonality with the existing sharings in SMILE.

10. RELATED WORK

Data markets for purchasing data are now being offered by several vendors [1, 2, 4]. A data sharing effort in the cloud is the FLEXSCHEME [6], where multiple versions of shared schema are maintained in the cloud, with the focus on enabling evolution of shared schema used by multiple tenants. Orchestra [25] casts data sharing as a data integration problem, where reconciling the difference in schema, formats, and trust between peers is achieved using provenance information associated with the tuples. A model for costing resources based on availability is developed in Mariposa [24], which is distinguished from our approach in the sense that in our case operators are fixed to a machine while in Mariposa they are executed at the machine with the cheapest cost.

Recently, there has been quite of work in pricing data in the cloud. Balazinska et al. [8] look at the challenges in providing a fine-grained pricing model in the cloud. In a subsequent work [7], they discuss the difficulty in coming up with a consistent pricing scheme when there are several MVs on the data that are available for sharing. This captures some of the complexities in designing a pricing model for our setup, which achieves sharing using MVs. A sharing plan creates expensive artifacts in the cloud that can be used by other sharings in the cloud. Upadhyaya et al. [26] and Kantere et al. [12] examine how to amortize the cost for such consumers that make expensive artifact investments that benefit other consumers in the cloud. Pricing from a consumer's perspective has also been studied [27] in the more general setting of a cloud service, which is applicable to our work.

Sharing using MVs adds interesting dimensions to a well studied problem domain. An MV maintenance process traditionally is broken into a *propagation* step, where updates to the MV are computed and an *apply* step, where updates are applied to the MV. Given that the base relations available for sharing are updated independently, a compensation algorithm [28] is needed, where the propagation step is computed on asynchronous base relations [5,21,29]. In particular, MVs over distributed asynchronous sources have been studied in the context of a single data warehouse [5, 29] to which all updates are sent. The key optimization studied in [5, 29] is in terms of reducing the number of queries needed to bring the MVs to a consistent state in the face of continuous updates on the source relations. [21] shows how n-way asynchronous propagation queries can be computed in small asynchronous steps, which are *rolled* together to bring the MVs to any consistent state between last refresh and present, which forms the basis of our sharing executor.

Reducing the cost of maintenance plans of a set of materialized view S is explored in [18], where *common subexpressions* [20, 23] are created that are most beneficial to S. Their optimization is to decide what set of common subexpressions to create and whether to maintain views in incremental or recomputation fashion. Stale-

ness of MVs in a data warehouse setup is discussed in [14], where a coarse model to determine periodicity of refresh operation is developed. Our problem shares common aspects with the cache investment problem [13] in terms of placement (what and where to be cached) of intermediate results and the *goodness* (another notion of staleness) of cache. Cache refresh in [13] piggybacks on queries, whereas we establish a dedicated mechanism to keep the MVs at the desired staleness.

11. CONCLUSION

In this paper we discussed the challenges and solutions in building a data sharing framework that hosts a large number of web and mobile applications. Similar to the app market ecosystems where the app developers publish apps and the users can purchase them, the data sharing ecosystem enables different applications to share data among one another as needed. We use two levers for controlling the cost a sharing, namely staleness and accuracy, which can become part of the SLA. We then proposed a What-if tool capable of answering the following questions both taking and not taking existing sharings into account. a) How to estimate the cost of a sharing with a specific staleness and accuracy? b) How to enable consumers to explore the configuration space for the most desirable configuration within a given budget? The What-if tool makes our sharing framework easy to use and facilitate data sharing.

In the future, we would like to study the problem of finding a fair and effective pricing model that avoids such "free rides" by some consumers at the expense of others, similar to [26]. We are also interested in investigating the problem of admitting multiple sharings at the same time instead of one by one. In this paper, we only consider staleness and accuracy as the two levers for controlling cost, but one could consider other dimensions or even provide fine-grained controls on staleness and accuracy for controlling costs. For example, the consumer could specify that the address field of a user relation can be updated with a relaxed staleness of a few days, while the location field should be updated within a few seconds.

12. REFERENCES

[1] Infochimps. `http://www.infochimps.com/`, 2012.

[2] Microsoft azure market. `http://datamarket.azure.com/`, 2012.

[3] Tesco mobile app. `http://www.tesco.com/apps/`, 2012.

[4] Xignite. `http://www.xignite.com/`, 2012.

[5] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *SIGMOD*, pp. 417–427, Tucson, AZ, 1997.

[6] S. Aulbach, M. Seibold, D. Jacobs, and A. Kemper. Extensibility and data sharing in evolving multi-tenant databases. In *ICDE*, pp. 99–110, Hannover, Germany, 2011.

[7] M. Balazinska, B. Howe, P. Koutris, D. Suciu, and P. Upadhyaya. A discussion on pricing relational data. Technical Report UW-CSE-11-05-04, University of Washington, 2011.

[8] M. Balazinska, B. Howe, and D. Suciu. Data markets in the cloud: An opportunity for the database community. In *VLDB*, volume 4(12), pp. 1482–1485, Seattle, WA, 2011.

[9] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pp. 34–43, Seattle, WA, 1998.

[10] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD*, pp. 263–274, Philadelphia, PA, 1999.

[11] S. Jain and P. K. Kannan. Pricing of information products on online servers: Issues, models, and analysis. *Mgmt. Sci.*, 48(9):1123–1142, 2002.

[12] V. Kantere, D. Dash, G. Gratsias, and A. Ailamaki. Predicting cost amortization for query services. In *SIGMOD*, pp. 325–336, Athens, Greece, 2011.

[13] D. Kossmann, M. J. Franklin, and G. Drasch. Cache investment: integrating query optimization and distributed data placement. *TODS*, 25:517–558, 2000.

[14] A. Labrinidis and N. Roussopoulos. Reduction of materialized view staleness using online updates. Technical Report CS-TR-3878, UMD CS, 1998.

[15] G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger, and P. F. Wilms. Query processing in R*. In *Query Processing in Database Systems*, pp. 31–47. Springer, 1985.

[16] A. Messac, A. Ismail-Yahaya, and C. A. Mattson. The normalized normal constraint method for generating the Pareto frontier. *Structural and Multidisciplinary Optimization*, 25(2):86–98, 2003.

[17] K. Miettinen. *Nonlinear Multiobjective Optimization*, volume 12 of *International Series in Operations Research and Management Science*. Kluwer Academic Publishers, Dordrecht, 1999.

[18] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, pp. 307–318, Santa Barbara, CA, 2001.

[19] T.-V.-A. Nguyen, S. Bimonte, L. d'Orazio, and J. Darmont. Cost models for view materialization in the cloud. In *EDBT*, pp. 47–54, 2012.

[20] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, pp. 447–458, Montreal, Quebec, Canada, 1996.

[21] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. In *SIGMOD*, pp. 129–140, Dallas, TX, 2000.

[22] J. Sankaranarayanan, H. Hacigumus, and J. Tatemura. COSMOS: A platform for seamless mobile services in the cloud. In *MDM*, volume 1, pp. 303–312, 2011.

[23] T. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

[24] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDBJ*, 5:048–063, 1996.

[25] T.J.Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen. Orchestra: facilitating collaborative data sharing. In *SIGMOD*, pp. 1131–1133, Beijing, China, 2007.

[26] P. Upadhyaya, M. Balazinska, and D. Suciu. How to price shared optimizations in the cloud. *PVLDB*, 5(6), 2012.

[27] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou. Distributed systems meet economics: Pricing in the cloud. In *HotCloud*, Boston, MA, 2010.

[28] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pp. 316–327, San Jose, CA, 1995.

[29] Y. Zhuge, H. García-Molina, and J. Wiener. The strobe algorithms for multi-source warehouse consistency. In *PDIS*, pp. 146–157, Miami Beach, FL, 1997.