SMILE: A Data Sharing Platform for Mobile Apps in the Cloud

Jagan Sankaranarayanan
Hakan Hacıgümüş
NEC Labs America, Cupertino, CA
{jagan,hakan}@nec-labs.com

Haopeng Zhang*
University of Massachusetts Amherst
haopeng@cs.umass.edu

Mohamed Sarwat*
University of Minnesota
sarwat@cs.umn.edu

ABSTRACT

We identify an opportunity to share data among mobile apps hosted in the cloud, thus helping users improve their mobile experience, while resulting in cost savings for the cloud provider. In this work, we propose a platform for sharing data among mobile apps hosted in the cloud. A "sharing" is specified by a triple consisting of: (a) a set of data sources to be shared, (b) a set of specified transformations on the shared data, and (c) a staleness (freshness) requirement on the shared data. The platform addresses the following two main challenges: What sharings to admit into the system under a set of specified constraints, how to implement a sharing at a low cost while maintaining the desired level of staleness. We show that reductions in costs are achievable by exploiting the commonalities between the different sharings in the platform. Experimental evaluation is performed with a cloud platform containing 25 sharings among mobile apps with realistic datasets containing user, social, location and checkin data. Our platform is able to maintain the sharings with very few violations, even under a very high update rate. Our results show that our method results in a cost savings of over 35% for the cloud provider, while enabling an improved mobile experience for users.

1. INTRODUCTION

Mobile applications (apps) compete in an increasingly crowded marketplace, with possibly thousand of apps performing similar or identical functions. In this crowded marketplace, developers can differentiate their apps by offering features that make the user's mobile experience more personalized. For instance, apps like SpotiSquare connect with Foursquare venues to determine the current location (venue) of the user, and then choose a music playlist depending on users' current context (e.g., eating dinner, exercising, driving etc.). To create such an experience requires that the app has access to additional information (i.e., datasets) about its user. The following example shows possible interactions among three apps, showcasing the benefits of sharing user information.

EXAMPLE 1. Consider the three apps — Opentable (restaurant reservation), Plango (calendar) and Sonar (friends location monitoring). Appointments of users requiring dinner reservations are shared by Plango with Opentable, which can then suggest restaurant options to users. Sonar can suggest a nearby restaurant as a meeting place by sharing their location information with Opentable. Mobile users get a seamless experience as the three apps now behave as a single entity.

Interestingly, with the increasing use of cloud-based resources, many of these apps may be hosted in the same cloud infrastructure (e.g., Amazon EC2). To enable such rich interactions, mobile apps should make their datasets available for sharing, as a way of encouraging other apps to build complementary features. At the same time, apps can consume several datasets from other apps in the cloud infrastructure. We identify two key considerations in sharing data that is important to mobile app developers.

- App developers want *reliable* access to datasets and do not want to deal with the complexity of creating and maintaining mechanisms (e.g., APIs [1,4] or web services [2]) for sharing data. Furthermore, they desire a service that is flexible enough to meet their needs while providing *guarantees* on the quality of the service.
- App developers want *timely* access to datasets. Mobility of users imposes limits on how much *staleness* app developers can tolerate on the datasets. This is because many types of user-related data get progressively less valuable with time. For instance, the location of a mobile user that is 50 seconds stale may be of limited use to a navigation app; however, 10 seconds stale data may be suitable.

In this paper, we propose a data sharing service in the cloud called SMILE (Sharing MIddLEware). The *service provider*, who manages the cloud infrastructure, offers data sharing as service and like any commercial business makes money by delivering services according to agreed upon quality of service levels. Data sharing is achieved by a mobile app developer (henceforth referred to as a *consumer*) specifying a *sharing*. A sharing must identify datasets of interest, the desired transformations on the data, and a staleness requirement. The consumer and the provider enter into a Service Level Agreement (SLA). The SLA is a contract specifying that the provider will ensure reliable access to the consumer on the shared data at the risk of paying a penalty if it is not maintained at the agreed upon staleness.

To successfully achieve data sharing on a large scale, two practical problems need to be solved by the provider. First, it is important to determine if a new sharing can be *admitted* (i.e., accepted) into the system and maintained at the appropriate staleness. This may not always be possible, especially if the datasets are updated at a *high* rate and the sharing needs to be maintained at a *low* staleness. If a sharing is incorrectly admitted, it will result in significant losses for the provider since the SLA may specify penalties for the provider in case the sharing misses the staleness requirement. Second, implementing the sharings is not free in the sense that the provider has to

*Work done while at NEC Labs

10.5441/002/edbt.2014.75

pay for the resources (i.e., CPU, Disk, Network) consumed in the cloud. Reducing provider cost is another important consideration.

The two practical problems we outlined above pose significant technical challenges that we address in this paper.

1. **Maintaining the shared datasets at required staleness:** The shared datasets are maintained as *materialized views* (MVs) and are always kept under the staleness specified in the SLAs. This is challenging because the sharings involve multiple datasets with varying staleness requirements. Note that there are many other ways (e.g., APIs, web services) of enabling sharing in the cloud; a discussion on the various methods and the pros-and-cons of each is given in [25].

2. **Testing for admissibility:** As noted above, there is a need for an effective method to decide whether to accept or decline a sharing agreement under multiple constraints, such as the given staleness, SLA penalty and platform cost considerations.

3. **Cost reduction for the provider:** To reduce cost, the platform provider needs to identify *commonalities* across multiple sharings with different staleness requirements, in order to save computational effort.

These three problems may not be specific to data sharing for mobile apps only, but also applicable to other areas. However, we observe that mobility makes these problems challenging and the solutions much more relevant in real world settings.

Some of these problems have been previously considered in the context of MV maintainance [5, 16, 23, 24] and multi-query optimization [11, 26]. However, prior work either considers the mechanics of MV maintainance [24] and refresh rates [16], or cost savings by removing commonalities [23, 26], but not staleness and cost requirements simultaneously. While important, the feasibility and economic value of sharing as well as infrastructure and privacy considerations are considered by [9, 13, 25], thus it is not the focus in this work.

In this work, we focus on the practical and technical challenges of enabling data sharing for mobile apps in the cloud. Mobility provides a perfect use-case scenario as it aligns with the three elements of our problem setup: it is cloud-based, requires reliable access to rich information, and has strict staleness requirements on datasets. To our knowledge, this work represents the first systematic, cloud-based platform for enabling data sharing with staleness guarantees. We make the following contributions in this paper.

1. A declarative sharing platform, which is fully implemented as a part of industrial system, with staleness guarantees on the sharings (Section 3).

2. A method of determining the *admissibility* of sharings ensures that the system only admits those sharings that it can maintain (Section 6).

3. A method for reducing the cost of maintaining the sharings by amortizing work across multiple sharings, where each sharing has its own constraints (Section 7).

4. Experimental evaluation is performed on a cloud platform with 25 sharings posed on realistic user, location, social, and checkin datasets. Our results show that the SMILE platform can maintain a large number of sharings with very low SLA violations, even under a high rate of updates. By amortizing work across multiple sharings, SMILE is able to achieve a cost savings of over 35% for the provider (Section 9).

2. RELATED WORK

While there has been some work on sharing in a mobile environment, they consider sharing either in an adhoc setting, such as between two mobile users [19], or among a group of mobile users [22]. A middleware for connecting mobile users and devices has been proposed [27] for providing various mobile services, such as management, security, and context awareness, but not for sharing.

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. First of all, the autonomy of the tenants means that *synchronous* propagation algorithms [10], where all sources are always at a consistent snapshot, are unsuitable for our purposes. Furthermore, to deal with the autonomy of the tenants, one has to resort to a compensation algorithm [28], where the propagation is computed on asynchronous source relations [5, 24, 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. [24] 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. Reducing the cost of maintenance plans of a set of materialized view S is explored in [20], where *common subexpressions* [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 an incremental or recomputation fashion. Staleness of MVs in a data warehouse setup is discussed in [16], where a coarse model to determine periodicity of refresh operation is developed.

As we will see later in the paper, our setup is different from [5, 29] in the sense that multiple MVs are maintained on multiple machines in our multitenant cloud database. Moreover, different update mechanisms with different costs and staleness properties can be generated based on where the updates are shipped as well as where the intermediate relations are placed, making the problem harder than [5, 29]. Next, [24] assumes that all the source relations are locally available on the same machine, which makes the application of their approach to our problem infeasible without an expensive distributed query. We combine propagation queries from [24] with *join* ordering [11, 18], such that propagation queries involving n source relations are computed in a sequence involving two relations at a time, requiring no distributed queries. In particular, we first ensure that the update mechanisms can provide SLA guarantees, after which common expressions among the various sharing arrangements are merged to reduce cost for the provider, which is similar in spirit to [20, 23]. Our work adds several additional dimensions to [20, 23] in terms of placement of relations, capacity of machines, SLA guarantee, and cost.

In contrast to [16], which determines the periodicity of the refresh operation of MVs maintained in a warehouse, our work is distinguished in the following way. Our work develops refresh cycles for multiple MVs from distributed sources with different staleness requirements while simultaneously reducing the total maintenance cost. This is significantly more complicated than the simple setup in [16] where they develop a simple model for determining a single refresh periodicity between a RDBMS and data warehouse without considering cost.

Our work is related to traditional view selection problems [6] in the sense that the set of sharing arrangements could have been obtained via the application of a view selection algorithm taking the

consumer workload as input. Our problem shares common aspects with the cache investment problem [14] 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 [14] piggy banks on queries, whereas we establish a dedicated mechanism to keep the MVs at the desired staleness. Our work shares common elements with [15] in the sense that merging data flow paths with common tuple *lineage* is similar to the way we perform *plumbing* operations on a sharing plan.

A related data sharing effort in the cloud is the FLEXSCHEME [8], where multiple versions of shared schema are maintained in the cloud, with the focus on enabling evolution of shared schema used by multiple tenants. Data markets [9] for pricing data in the cloud looks at the problem from tenant and consumer perspectives, but we look at the problem from the provider's perspective. A similar but not identical problem is reducing the cost for the consumers (i.e., fair pricing of common queries) [9] and sharing work across concurrent running queries [26]. Although we only concern ourselves with the *staleness* of the data as the only quality measure of the data being shared, other considerations such as data completeness, accuracy are also applicable here [7]. In our problem, the challenge is to maintain the sharing arrangements always maintained on agreed upon terms (i.e., SLAs), while keeping down the infrastructure costs. Satisfying these dual goals makes the sharing problem challenging from the provider's perspective.

3. PRELIMINARIES

The provider has to consider a set S of sharings $\{S_1, S_2 \cdots S_m\}$, for inclusion in the sharing platform. Here, we consider the case where there are no existing sharings in the system, yet the solution we develop is equally applicable to the case when the platform already has several prior sharings. Our solution that we later develop will identify which of the sharings in S should be admitted into the sharing framework, while at the same time minimizing provider cost and meeting SLA requirements. Each S_i specifies the applicable datasets, transformations, staleness requirements and penalties as described next.

To specify a sharing S_i, a consumer starts by identifying datasets (i.e., *base relations*) of interest, or subsets of datasets. Next, the consumer must determine a way to combine these datasets by specifying *transformations* on the data. In this work, we restrict the transformations on the base relations to include the following three operators.

1. Choose a subset of tuples using a selection predicate
2. Choose a subset of the attributes
3. Combine base relations using a common key

In other words, the transformation can be specified using a *Select-Project-Join* (SPJ) query that is applied on the base relations. The consumer next specifies a staleness requirement t expressed in time units (e.g., 20 seconds) on the shared data as well as any applicable penalty pen_s. A sharing in SMILE is enabled by the creation of a *materialized view* (MV), which describes the transformations over the base relations. For each sharing S_i, the system creates a MV which is always maintained within a staleness of t time units as specified by S_i. This means even though the base relations are independently updated, the state of the MV is always *consistent* with the state of the base relations within t seconds.

As an illustration of defining a sharing, we revisit Example 1 and provide a more concrete example of a sharing that the Opentable (i.e., restaurant) app may define using the SMILE platform.

EXAMPLE 2. *Plango (i.e., Calendar app) makes the base relation of* `User_Events` *of events extracted from users' calendar available for sharing. Opentable has its own relation* `User_Accts` *of users that use the app. It specifies a sharing, "I want to know about dinner events for the users who use my app within 10 seconds of a new event being recorded so that I could offer recommendations to them." The base relations are* `User_Events` *and* `User_Accts`*, and the transformations are specified as the following SPJ query:* `EventType="dinner"` *from a join of* `User_Events` *and* `User_Accts`*. The staleness t is specified as 10 seconds and the penalty pen_s is \$.001 per late delivery of a tuple.*

After the sharing S_i is defined as per the above example, it is given to the provider for deciding admissibility and implementation in the platform, which is described in Section 6.

4. SMILE ARCHITECTURE

Figure 1 shows the architecture of the system. There is a set of machines available to implement the sharings. Each machine runs a single database instance (Postgresql in our case). The SMILE platform consists of three main components — (a) delta capture, (b) sharing optimizer, and (c) sharing executor — that perform the following functions, respectively: (a) capture changes (i.e., delta) on the base relations as updates are applied on them; (b) generates *plan* for moving these updates from the base relations to the MVs; and (c) schedules the movement of these updates by taking system fluctuations into account. We briefly describe the three main system components below.

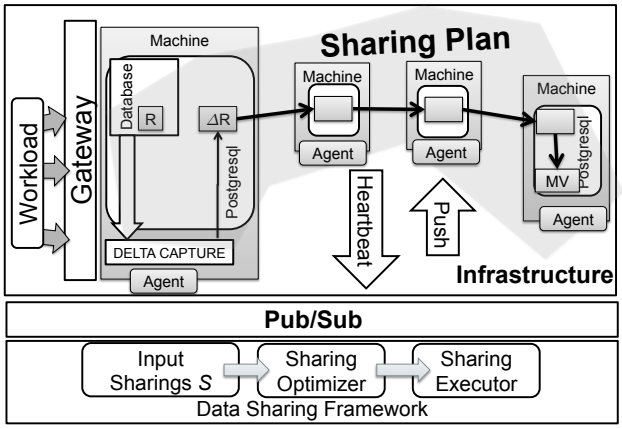

Figure 1: Architecture of the sharing platform

4.0.1 Delta Capture and Timestamps

As the base relations are updated, a *delta* capturing mechanism (i.e., tuple creation, deletion or updates) records the modified tuples. Our mechanism uses the Streaming Replication facility [3] in Postgresql to capture the deltas. This module in Postgres allows the Write Ahead Log (WAL) to be streamed on to another Postgresql instance in recovery mode so that a nearly identical replica of a database can be maintained. Our module fakes itself as a Postgresql instance and obtains a WAL stream. The modified tuples are extracted from the stream, unpacked and written to the disk.

Every base relation R is associated with a *delta* relation, denoted by ΔR that records the modified tuples as update queries are applied on R. The tuples in ΔR are populated by the delta capturing module. The MVs in the system also contain corresponding delta tables. If R is a MV then ΔR contains both prior updates as well as those that have not yet been applied to R. The tuples in ΔR of a MV is populated, moved and applied by the sharing executor.

Every relation, delta of a relation or MV in our platform records its last modification timestamp. The timestamps are generated using a distributed clock [17] that is periodically synchronized. Each tuple in the delta also records an associated timestamp. Maintaining the sharings at their appropriate level of staleness is achieved by keeping track of the last modification timestamps of the base relations and comparing them to the timestamp of the MV. The SMILE system maintains an up-to-date timestamp information on each sharing, hence is aware of the current staleness of all the sharings. Updates are moved from the base relations to the MV in a way that ensures that the sharings do not miss their SLAs.

4.0.2 Sharing Optimizer

Given a set S of new sharings, the sharing optimizer generates an update mechanism for each sharing in S using a three step procedure described below.

a. A sharing S_i in S can be *admitted* if the system can maintain S_i at the desired level of staleness. This determination is necessary to prevent the system from entering into SLAs that it cannot satisfy (Section 6).

b. If S_i is admissible, we generate its *sharing plan* such that it can move updates from the base relations to the MV within the time specified in the staleness SLA. Moreover, the sharing plan is also cost effective in terms of its infrastructure resource consumption (Section 6).

c. Once the individual sharing plans of all the sharings in S are determined, commonalities across sharings are identified and removed to produce a single global sharing plan D that implements all the sharings (Section 7).

4.0.3 Sharing Executor

The sharing executor is the execution engine of the system which maintains the sharings at or below the required staleness level. The sharing executor is an implementation of an asynchronous view maintenance algorithm [24].

The sharing executor computes the current staleness of a sharing by taking 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. It schedules the updates to be applied on the MV so that its staleness is reduced. 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 with the last modification timestamps of the base relations and the MVs.

Our implementation of the executor is *lazy* by design in the sense that it does not refresh unless it is absolutely necessary or the sharing will miss its SLA. This way, the executor bunches as much work as possible thereby reducing redundant effort. The refresh is neither too early nor too late, but finishes just before a sharing is about to miss its staleness SLA. We provide more details on the sharing executor in Section 8.

5. SHARING PLAN

The update mechanism of a sharing is implemented as a *sharing plan*, which is analogous to a query execution plan in databases. We will henceforth refer to it simply as a *plan* in the rest of the paper. The plan is expressed in terms of four operators that form the transformational path for the updates from the base relations to the MV. This is represented using a *Directed Acyclic Graph* (DAG) such that the vertices are relations or deltas of relations tied to a particular machine, and the edges apply transformational operators. The plan is expressed using the following four edge operators, that 1) apply updates (DeltaToRel) , 2) copy updates between machines (CopyDelta) , 3) join updates (Join), and 4) union (i.e., merge) updates (Union).

As the plan operates on base relations that are asynchronously updated, the input vertices to an operator may have different timestamps. An operator takes any mismatch in the timestamps into account by rolling back all the input vertices to the minimum of the timestamps among its inputs. This is referred to as *compensations* [28]. Rolling back the timestamp of a relation or a MV is possible due to the delta relations associated it. The operators for applying, copying and merging updates are based on their standard interpretations, except that they additionally apply compensations to the inputs as the first step. Our join operator performs a compensation which is an implementation of the algorithm from [28].

We will not provide the implementational details of the operators but instead show an example of a plan that performs a relational join on two asynchronous base relations A and B on different machines. The plan is referred to as "in-place" as it does not involve making the copies of the base relations. The vertices and the edge operators in the plan periodically move the updates from the base relations to the MV to keep it maintained incrementally.

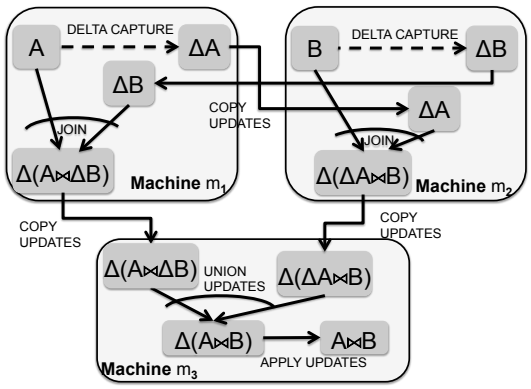

Figure 2: One possible plan involving an in-place join of a base relation A on machine m_1 and B on machine m_2 such that the resulting MV $A \bowtie B$ is placed on machine m_3

EXAMPLE 3. Figure 2 shows the plan of a sharing S_i that performs a transformation $A \bowtie B$ on two base relations, A and B. The plan is a DAG consisting of 12 vertices and 10 edges. The vertices are either base relations (e.g., A, B or its copies), MVs (e.g., $A \bowtie B$) or delta relations (e.g., $\Delta(\Delta A \bowtie B)$). The edges corresponds to operators that either apply, copy, merge, join updates, to complete the transformation path from the base relations to the MV. Note that select and project predicates can be specified in S_i's transformation. All the four edge types can apply select and project predicates to their inputs if one is specified in addition to their usual functionalities. We handle these predicates by using the *pushdown* heuristic [11].

Given a sharing that specifies a set of transformations on the base relations, the plan generation algorithm enumerates all the plans that implement the sharing. However, not all of the plans satisfy the constraints we develop in the reminder of this section. In particular, we concern ourselves with two key properties of a plan, namely its critical time path and dollar costs, which are described below.

5.1 Critical Time Path

The *critical time path* of the plan is the longest path in terms of seconds that represents the most time consuming data transformation path in the plan. Note that the plan is admissible only if the length of its critical time path is less than the required staleness of the sharing, or else the system cannot maintain it.

The sharing optimizer estimates the critical time path of a plan using a time cost model for each operator. The model estimates 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 the 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 plan p shown in Figure 2, CP(p) computes the time taken along the longest transformation path from A or B to the MV $A \bowtie B$. Section 9 provides additional details on how we developed the time cost model for the four operators.

5.2 Cost Model

The cost of the plan, expressed in dollars per second, is computed by the amount of CPU, network, and disk capacity consumed to setup the sharing and maintain it at the required staleness. The provider periodically moves the updates to the MVs and buys CPU, disk and network capacities from the Infrastructure as a Service (IaaS). This cost can be further divided into two categories: resource usage (i.e., CPU, disk capacity, network capacity) and penalty due to possible SLA violations.

Resource Usage. There are existing analytical models that estimate the usage of various resources for maintaining a MV, based on update rate, join selectivity, data location, etc. (e.g., [21]). This analytical model is implemented as a $resCost$ function that computes the cost of the resources consumed by a plan. 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 sharing executor has much flexibility in deciding when to update the MV. Specifically, given a new tuple to the base relations, the service provider can push it to the MV immediately, or wait for as long as 29 second before pushing it. On the other hand, when the staleness becomes close to the critical time path, there is much less flexibility since other sharings in the infrastructure may compete for resources.

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

SLA Penalty. At low staleness values the natural fluctuations in the update rates may cause a plan to miss the SLA. This is because the 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, which also has to be factored into the cost. We estimate this by assuming a Possion arrival of updates, and modeling the plan as a M/M/1 queuing system. Given the arrival rate of each base relations, we can estimate the arrival rate of tuples in the MV 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 plan.

For an M/M/1 queue with arrival rate λ and service rate μ, the percentage of items with sojourn time t larger than the staleness SLA s is $P(t > s) = e^{(\lambda-\mu)\cdot s}$. Thus the dynamic cost of a plan p with staleness s is calculated as:

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

$resCost(p)$ 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. $e^{(\lambda\cdot a-\mu)\cdot s} \cdot pen_s$ 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. SHARING OPTIMIZER

The goal of the sharing optimizer is to produce a low-cost admissible plan. Satisfying the dual constraints of finding an admissible plan that is provably *cheapest* amongst all plans is a hard problem.

A sharing S_i specifies SPJ transformations on a set of base relations. As the base relations are hosted on different machines, there are several ways of combining them as well as where to place the intermediate results. This results in plans with varying dollar cost and critical time paths. For instance, performing many operations in parallel on different machines may produce a plan with a small critical time path. But such a plan may have a high dollar cost due to high infrastructure costs involved in using many machines. On the other hand, operations can also be performed sequentially to reduce the dollar cost but at the expense of a high critical time path.

Among the generated plans those that have a critical time path greater than the SLA of S_i cannot be maintained by the system at the desired staleness level, and hence are not *admissible*. The admissibility of plans forms the *hard constraint* of our problem in the sense that the system should not admit a sharing that cannot be handled by the system. At the same time, it also should not deny admitting sharings that otherwise should have been admissible.

The sharing optimizer is based on the polynomial time heuristic solution developed for System-R [11] and its analogous distributed variant R* [18]. Our approach relies on generating, using a dynamic programming approach, the cheapest possible plan in terms of dollar costs, regardless of its critical time path and another plan with the smallest critical time path, regardless of its dollar costs. We refer to these plans as *Dynamic Programming Dollar* (DPD) and *Dynamic Programming Time* (DPT), respectively. The DPD and DPT plans have the following properties:

1. DPT is a plan with a *low* critical time path that is not optimized on the operating dollar cost. If DPT is not admissible, then the sharing can be safely rejected by the provider as there cannot be a plan with a lower critical time path.

2. DPD is a plan with a *low* operating dollar cost that is not optimized on the critical time path. If DPD is admissible, then it is also of the cheapest cost.

We provide a dynamic programming formulation to produce DPT and DPD in Section 6.1, and provide a plan generation algorithm in Section 6.2.

6.1 Dynamic Programming Formulation

We cast the problem of generating a plan as a bottom-up dynamic programming formulation given by JOINCOST in Algorithm 1. Consider a sharing that specifies a join sequence on the base relations. For example, Figure 2 shows a join sequence of length two using the two base relations, A and B.

Let S_i be a sharing in S such that SRC(S_i) is the set of source vertices of S_i and MV(S_i) is the vertex corresponding to the MV

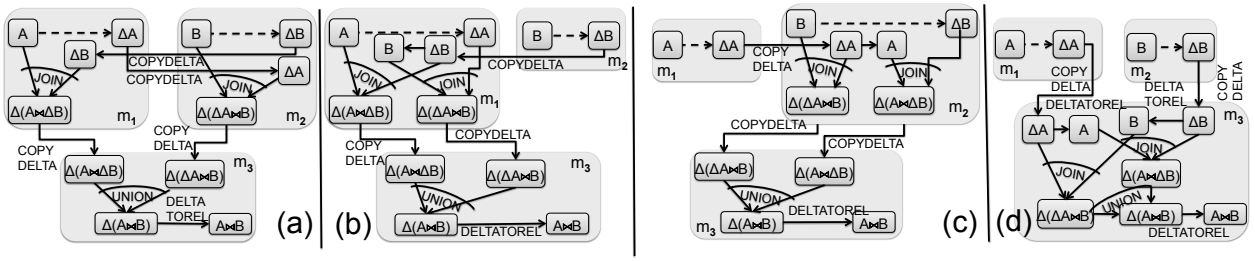

Figure 3: Four ways of joining A and B, (a) in-place (no copies of A or B), (b) copy B (c) copy A, (d) copy A and B.

of S_i. SLA(S_i) is the staleness SLA of S_i. MAC(S_i) denoted the set of dedicated machines available to host the sharing. Below, the short-hand notation <v, m> denotes any vertex v (i.e., relation, MV or a delta of a relation) in the plan that is placed on a machine m.

We capture the *state* of the problem up on creating a join sequence R on a machine m (i.e., <R, m>) as ($D^{<R,m>}$, CAP$^{<R,m>}$) such that:

1. $D^{<R,m>}$ is the *cheapest* plan among all those that produce R on machine m. The cheapest plan is chosen by applying a cost function COSTCALC, which takes a plan as input and produces a cost value. Later, we will specify two implementations of COSTCALC that will produce the DPT and DPD plans.

2. CAP$^{<R,m>}$ is the remaining capacity in the infrastructure (e.g., CPU, disk, network capacities) after discounting the capacity consumed by $D^{<R,m>}$.

We generate the join sequence R at machine m bottom-up by enumerating all the states corresponding to: a) $R-a$ on any machine in MAC(S_i), b) with any remaining capacity. $R - a$ refers to a join sequence R without a base relation a. $D^{<R,m>}$ is generated by adding vertices and edges required to join $R - a$ with a to the plan from the prior state. Among the plans generated this way, we choose the plan with the smallest value produced by COSTCALC.

Such a formulation is used by JOINCOST to obtain the plan of any arbitrary join sequence R on $m_i \in$ MAC(S_i), which is formed by joining $R - a$ on $m_j \in$ MAC(S_i) (i.e., <$R - a, m_j$>) with a base relation a on machine $m_k \in$ MAC(S_i) (i.e., <a, m_k>).

At a high level, given two relations A on machine m_1, and B on machine m_2, we consider four ways of producing $A \bowtie B$ on a machine m_3, which are illustrated in Figure 3. In particular, Figure 3a shows the case where $A \bowtie B$ is produced without copying any of the base relations (i.e., in-place join). Figure 3b–c show the cases when one of the base relations is copied. Figure 3d shows the case when both A and B are copied to m_3, and then an in-place join is performed. Note that the four cases (a)–(d) in the JOINCOST formulation (lines 8–11) correspond to the four ways of joining two relations A and B in Figure 3.

JOINCOST uses an *in-place* join and a *copy* procedure to update the plan $D^{<R-a,m_j>}$ to produce R on machine m_i. We explain below the procedure of creating a copy of a relation and joining two relations using A and B in place of $R - a$ and a, respectively.

To copy of <A, m_1> to form <A, m_2> requires the addition of one vertex and two edges – vertex <A, m_2> and CopyDelta between <$\Delta A, m_1$> and <$\Delta A, m_2$>, and DeltaToRel (apply updates) between <$\Delta A, m_2$>. An in-place join between <A, m_1> and <B, m_2> to produce <$A \bowtie B, m_3$> can add up to 8 vertices and 8 edges, as shown in Figure 3 depending on whether m_1, m_2 and m_3 are all distinct machines.

After adding the necessary vertices and edges, the capacities of machines involved are modified using the $resCost$ function defined previously. If there is no capacity left in m_1 or m_2, COSTCALC function would cost such a plan at ∞ indicating that the plan is infeasible.

Algorithm 1 sub JOINCOST(<R, m_i>)

```
1: for all join sequences R − a do
2:   for m_j ∈ set of available machines do
3:     (CAP^<R−a,m_j>, D^<R−a,m_j>) = JOINCOST(<R − a, m_j>)
4:     for m_k ∈ set of available machines do
5:       CAP' ← CAP^<R−a,m_j>
6:       D' ← D^<R−a,m_j>
7:       Choose cheapest case among (a)–(d), update CAP' and D'
8:         Case (a): In-place join of <R − a, m_j> with <a, m_k>
9:         Case (b): Copy <R − a, m_j> to <R − a, m_k>. In-place join of
           <R − a, m_k> and <a, m_k>
10:        Case (c): Copy <a, m_k> to <a, m_j>. In-place join of
           <R − a, m_j> with <a, m_j>.
11:        Case (d): Copy <R − a, m_j> to <R − a, m_i>. Copy <a, m_k>
           to <a, m_i>. In-place join of <R − a, m_i> and <a, m_i>
12:      Update CAP^<R,m_i>, D^<R,m_i> with D' and CAP' if COSTCALC(D')
         is cheaper.
13:    end for
14:  end for
15: end for
16: return CAP^<R,m_i>, D^<R,m_i>
```

6.2 Plan Generation Algorithm

Finally, we describe an algorithm which takes a sharing S_i and generates two sharing plans depending on the choice of the cost function, which we had left unspecified earlier. Recall that the DPD plan has a *low* dollar cost, whereas the DPT plan has a *low* critical time path.

If a DPD plan is needed, COSTCALC uses COST function (described in Section 5.2), which computes the cost of a plan in dollars per second. If a DPT plan is needed, COSTCALC uses the CP function (described in Section 5.1), which computes the critical time path of the plan.

The plan generation algorithm computes the cheapest way to build all join sequences of length $1 < x \leq |\text{SRC}(S_i)|$ in a bottom-up manner, where SRC(S_i) is set of base relations of S_i. The algorithm obtains the cost to construct longer join sequences of length x, using the output of prior invocations for join sequences of length $x - 1$. The algorithm terminates when it produces the join sequences corresponding to the set of transformations specified in the sharing. The sharing optimizer generates both DPD and DPT plans. If the DPT plan is not admissible, then it means that there *may* not exist a plan of S_i that is admissible and hence, S_i is *rejected* by the provider. If DPD and DPT are both admissible, the sharing optimizer uses DPD as it has a lower cost than DPT.

7. MULTI-SHARING OPTIMIZATION

If two different admitted sharings share similar vertices and edges, there could be an opportunity to further reduce the cost. The provider can take advantage of this *commonality* by amortizing the

operating costs across several sharings. The commonality here is replacing two disjoint sets of vertices and edges belonging to different sharings that perform identical or similar transformation with a common set for multiple sharings.

Although our idea of merging commonalities in plans is similar as merging common subexpressions in concurrent running query execution plans [26], 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, as we show below, unlike [26] we do not restrict to only merging identical subexpressions across plans.

D is a *global* plan obtained by merging the plan of sharings in S and then discarding duplicate edges and vertices. Note that D need not be a single connected component. Each vertex (or edge) $v \in D$ records the identity of all the sharings that it serves, such that $\text{SHR}(v)$ records the sharings to which v belongs. Given a vertex (or a set of vertices) v, let $\text{ANC}(v)$ be the set of vertices and edges that are ancestors of v in D.

Commonalities in D are reduced by applying a *plumbing* operator repeatedly until the D does not perform any redundant work. A plumbing operator takes two sets of vertices v_1 and v_2 belonging to plans as inputs. Then, vertices and edges that supply the vertices in v_2 (or v_1) are retained but those supplying v_1 (or v_2) are discarded. We now discuss the mechanics of a plumbing operator as well as an algorithm to apply them.

7.1 Plumbing Operations

A plumbing operation p is defined between a set of source vertices $\text{SRC}(p)$ and another set of destination vertices $\text{DST}(p)$, such that after the plumbing operation $\text{DST}(p)$ gets tuples via $\text{SRC}(p)$. Applying p on D results in a potential cost reduction is due to the removal of vertices from D, although some expenditure in the way of additional vertices and edges is made to facilitate the plumbing. The *benefit* of p is defined as the dollar cost savings due to the removal of vertices and edges in the plan minus the cost of adding the additional vertices and edges to implement p. Applying p can potentially increase the critical time path of all sharings in $\text{SHR}(\text{DST}(p))$. This is because they now obtain their tuples via $\text{SRC}(p)$, which may be a longer path in terms of time. We note that p is feasible only if it has a positive benefit. Moreover, performing p should not cause the critical time path of any of the sharings in $\text{SHR}(\text{DST}(p))$ to exceed their SLA. We consider the following two kinds of plumbing operations, which are shown in Figures 4a–b.

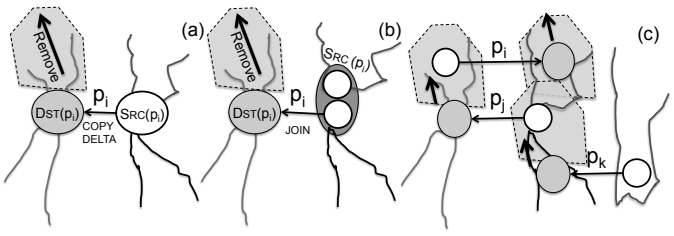

Figure 4: Types of plumbings (a) CopyDelta and (b) Join and (c) how plumbings can affect one another

1. *Copy Plumbing:* Takes two delta vertices on different machines and adds a CopyDelta edge between them.

2. *Join Plumbing:* Takes two vertices – a delta and a relation – and performs a join to obtain the destination delta vertex. For example, $\Delta(A \bowtie \Delta B)$ is plumbed using A and ΔB by a Join edge and up to two CopyDelta edges.

Note that we do not preclude other kinds of plumbing operations here as long as they do not compromise the correctness of the plan and result in a cost reduction.

A plumbing operation p is implemented as follows. First, add the necessary edges and vertices to perform p. For all vertices and edges $v \in \text{ANC}(\text{DST}(p))$, remove $\text{SHR}(\text{DST}(p))$ from the $\text{SHR}(v)$. For all $v \in \text{ANC}(\text{SRC}(p))$, add $\text{SHR}(\text{DST}(p))$ to $\text{SHR}(v)$. Add $\text{SHR}(\text{DST}(p))$ to $\text{SHR}(\text{SRC}(p))$ as well. Remove vertices and edges $v \in \text{ANC}(\text{DST}(p))$ s.t., $\text{SHR}(v) = \emptyset$.

7.2 Optimization Algorithm

Given the global plan D, we want to identity a set of plumbing operations to perform, that would produce the *provably* least cost plan without exceeding the staleness SLA of any of the sharings. The hardness of this problem comes from the observation that a plumbing can affect the benefit of other plumbings. Figure 4c shows an example of three plumbings p_i, p_j and p_k that affect one another. For example, p_i cannot be applied if p_j has already been applied to D as applying p_j would have removed the vertices in $\text{SRC}(p_i)$. For the same reason p_j cannot be applied if p_k has already been applied. Finally, the benefit and the increase to the critical time path due to p_i is affected if p_k has already been already applied.

An optimal algorithm that chooses a set of plumbings to perform resulting in a provably cheapest plan is a hard problem. For our purposes here any plan that is cheaper than D is good enough. Our algorithm is referred to as greedy hill-climbing approach as it applies one plumbing at a time in a greedy fashion, until no more plumbings can be applied to D. This algorithm has at least the desirable property that it is fast to execute and intuitive to understand.

Given the global plan D, let P be the set of all possible plumbing operations that can be performed on D. The greedy hill-climbing algorithm at each iteration first computes P and then applies the plumbing operation $p \in P$ with the maximum benefit. The set of plumbings P is obtained using a two step process. First, we examine all pairs of vertices in D to determine if a copy plumbing can be performed between them. Second, we examine all applicable triples of vertices to determine if a join plumbing can be performed between them. The benefit and the critical time path increase due to each plumbing are computed. If a plumbing has zero or negative benefit, or causes a sharing to miss its SLA, it is discarded from P. The algorithm terminates when no more plumbing operation can be applied to D. The resulting global plan forms the input to the sharing executor. We will show later in Figure 13 that this strategy is quite effective and results in large savings for the provider.

8. SHARING EXECUTOR

The sharing executor takes a global sharing plan D and maintains the individual sharings at the appropriate level of staleness. In its very nature, the sharing executor must be robust to any deviations in the update rate and the behavior of the machines in the infrastructure. The sharing executor applies its own set of run time optimizations, some of which are briefly described. We first describe how staleness is computed and how the push operator reduces the staleness of a sharing arrangement. Next, we design a model to determine the most appropriate time to push the sharings.

8.1 Staleness and Push

Each machine runs an agent which is responsible for sending periodic timestamps updates to the sharing executor. We refer to these messages as *heartbeats*. We implement the following three functions – $\text{TS}(.)$, $\text{MINTS}(.)$, $\text{MAXTS}(.)$ – that obtain the current timestamp of a vertex, and the minimum and maximum timestamps of a *set* of vertices in the sharing plan. For instance, $\text{MINTS}(\text{SRC}(S_i))$ and

$\text{MAXTS}(\text{SRC}(S_i))$ are the minimum and maximum timestamps of the sources of S_i.

The current staleness of a sharing is defined as the difference between the maximum of the timestamps of the base relations of S_i to that of the MV of S_i. Note that the staleness should always be maintained to be less than $\text{SLA}(S_i)$. This is captured by the following inequality.

$$\text{MAXTS}(\text{SRC}(S_i)) - \text{TS}(\text{MV}(S_i)) \leq \text{SLA}(S_i).$$

To reduce the staleness of a sharing S_i, the executor schedules a sequence of PUSH commands in a topological order starting with $\text{SRC}(S_i)$, until the timestamp of the MV has a more up-to-date timestamp. A PUSH command instructs the agent to advance the timestamp of a vertex in the sharing plan by applying an operator denoted by its incoming edge. The incoming edge belongs to one of the four edge types we described in Section 5.

Suppose an agent receives a PUSH command to advance a vertex v to timestamp t. Suppose that e is an incoming edge of v. The agent first obtains a write lock on v. It then compares the current timestamp t' of v with that of t. If $t' \geq t$ then there is no need to perform any work. If $t' < t$, then the agent performs the operation corresponding to e's type so that the timestamp of v can be advanced to t. Once the operation has been performed the agent responds with a PUSHDONE command, and piggybacks useful statistics such as time taken to perform the operation and current timestamps. The executor up on receiving the PUSHDONE proceeds with the outgoing edges of v.

When it comes to maintaining multiple sharings, the design of a sharing executor is simplified due to the observation that any sharing in S can be pushed independently of the others in S even though they may have common edges and vertices. Suppose a vertex is at a timestamp t and there are two concurrent PUSH commands to advance it to t_1 and t_2, $t \leq t_1 \leq t_2$, respectively. Regardless of the order in which the two commands are executed, the amount of work done by e is equal to the work done to advance the timestamp to t_2. This is why the sharing executor does not have to coordinate PUSH commands between the various sharings that it maintains. This makes for a simple design of the sharing executor, especially since the sharings may have different staleness SLAs and may have to be pushed at different rates.

8.2 Design

Our sharing executor uses a simple model to determine two key questions: a) Is it time to push S_i? b) By how much to advance the timestamp of MV of S_i? To determine these two questions, we develop a model to determine the most appropriate time to schedule the push and the timestamp to push the MV to such that the sharing will not miss its SLA.

A simpler design of a sharing executor pushes all the sharings in S every second so that they do not violate their SLA. Given the property that the critical time path of an admissible sharing is less than its staleness SLA, the push will finish before the SLA is violated. However, our sharing executor does not push every second but rather bunches up *sufficient* work so that the push is issued as late as possible. Yet, it is scheduled such that the push would be completed before S_i becomes stale.

To develop the model, we modify the critical time path function $\text{CP}(D_i, x)$ to take an additional parameter x, which corresponds to the amount of timestamp to advance the MV of S_i. In Section 5.1 when we described how we compute the critical time path of a sharing plan, x was defaulted to be one but now can take up any arbitrary value greater than or equal to one. We also added a feedback loop to the CP function so that it constantly recomputes the time model to take into account recent system fluctuations. We record the actual time to perform each of the operators, compare it against estimated time and periodically recompute the time model. This feedback loop allows our system to be reasonably robust to data and machine fluctuations.

An appropriate timestamp t to advance the MV of S_i should be greater than the current timestamp of MV but should be less than or equal to the minimum of the timestamp of the sources of S_i. In particular,

$$\text{TS}(\text{MV}(S_i)) < t \leq \text{MINTS}(\text{SRC}(S_i)).$$

When the push finishes, the MV would be at the timestamp t, while the timestamp of the sources may all be advanced by $\text{CP}(D_i, t - \text{TS}(\text{MV}(S_i)))$. So, the staleness of the sharing at the time the push finishes would be: $\text{MAXTS}(\text{SRC}(S_i)) + \text{CP}(D_i, t - \text{TS}(\text{MV}(S_i)))$. We stipulate that the staleness when the push finishes should be less than the staleness SLA using the following inequality:

$$\text{MAXTS}(\text{SRC}(S_i)) + \text{CP}(D_i, t - \text{TS}(\text{MV}(S_i))) - t \leq \text{SLA}(S_i).$$

On the other hand, the sharing executor does not want to push too *early* as well. In other words, the sharing executor is early if the push command could have waited a bit longer and still could have completed before S_i became *stale*. This can be stipulated by adding the additional constraint that:

$$l * \text{SLA}(S_i) \leq \text{MAXTS}(\text{SRC}(S_i)) + \text{CP}(D_i, t - \text{TS}(\text{MV}(S_i))) - t \leq \text{SLA}(S_i),$$

where $l > 0$ (0.8 in our case) is chosen to account for run-time deviations, such as a queuing delay if the PUSH waits for the capacity on a machine to be available. An appropriate value of t is obtained by performing a binary search between $\text{TS}(\text{MV}(S_i))$ and $\text{MINTS}(\text{SRC}(S_i))$ that satisfies the above constraint.

9. EXPERIMENTS

In this section, we present an experimental study to validate the SMILE sharing platform. We first describe the experimental setup in Section 9.1. We then evaluate the performance of our system for varying rate of updates on the base relation in Section 9.2 and varying SLA in Section 9.3. We examine the effect of varying the number of machines and the sharings in the infrastructure in Section 9.4. Next, the efficacy of the hill-climbing algorithm applied to DPT and DPD is shown in Section 9.5. Finally, we highlight the robustness of the sharing executor in Section 9.6 by varying the update rates on the base relations and varying read workload on the MV.

9.1 Setup

Our experimental setup creates a mobile cloud ecosystem containing 25 apps where sharings are specified using user, social network, location, checkin, and user-content datasets; the datasets and the sharings are representative of those one may find in a mobile cloud. We collected Twitter 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 were unpacked into nine base relations corresponding to the information about the user (i.e., `users` relation), tweets (i.e., `tweets` relation), social network (`socnet` relation), checkins (`foursq`), and user-content (i.e., `urls`, `hashtags`, `curloc`, `photos` relations) associated with the tweets and the location of the user (i.e., `loc` relation). This creates our realistic datasets that capture rich information about users, locations, social contacts, checkins and the various contents the users are interest in.

Table 1: Twitter base relations (left) and the twenty-five sharings (right) used in the evaluations

Base Relations:	
users(uid, ...)	User info
tweets(uid, uid, ...)	Tweets info
socnet(uid, uid2, ...)	Social network
loc(uid, place, ...)	User address
curloc(tid, lat, lng, ...)	User current loc
urls(tid, url, ...)	Tweet links
hashtags(tid, tags, ...)	Tweet entities
photos(tid, urls, ...)	Photo links
foursq(tid, rid, ...)	Rest. checkins

	Sharings (Apps):		
S1	users ⋈ socnet (twitaholic)	S13	foursq ⋈ users ⋈ tweets ⋈loc (locc.us)
S2	users ⋈ tweets ⋈ curloc (twellow)	S14	tweets ⋈ loc (locafollow)
S3	users ⋈ tweets ⋈ urls (tweetmeme)	S15	users ⋈ loc ⋈ tweets ⋈ curloc (twittervision)
S4	users ⋈ tweets ⋈ urls ⋈ curloc (twitdom)	S16	foursq ⋈ users ⋈ tweets ⋈ socnet (yelp)
S5	users ⋈ tweets (tweetstats)	S17	users ⋈ loc (twittermap)
S6	tweets ⋈ curloc (nearbytweets)	S18	users ⋈ tweets ⋈ photos ⋈ curloc (twitter-360)
S7	urls ⋈ curloc (nearbyurls)	S19	users ⋈ tweets ⋈ hashtags ⋈ curloc (hashtags.org)
S8	tweets ⋈ photos (twitpic)	S20	users ⋈ tweets ⋈ hashtags ⋈ photos ⋈ curloc (nearbytweets)
S9	foursq ⋈ tweets (checkoutcheckins)	S21	users ⋈ tweets ⋈ foursq ⋈ photos ⋈ curloc (nearbytweets)
S10	hashtags ⋈ tweets (monitter)	S22	foursq ⋈ curloc (nearbytweets)
S11	foursq ⋈ users ⋈ tweets ⋈ curloc (arrivaltracker)	S23	photos ⋈ curloc (twitxr)
		S24	hashtags ⋈ curloc (nearbytweets)
S12	foursq ⋈ users ⋈ tweets (route)	S25	hashtags ⋈ users ⋈ tweets (twistroi)

We specify 25 sharings by combining these 9 base relations in different ways. A description of the base relations and the 25 sharings used in the evaluation are shown in Table 1. For each of the 25 sharings, we also mention an existing real app that may benefit from such a sharing. For instance, the application *twitter-360*, which displays nearby photos may be interested in S18 which corresponds to users ⋈ tweets ⋈ photos ⋈ curloc. By building an ecosystem around twitter data and choosing sharing that match the functionalities of existing apps, we are ensuring that our evaluation is as realistic as possible. Our setup consisted of 6 machines, such that the 25 sharings were arbitrarily assigned to the available machines. All the machines in our setup ran identical versions of Postgresql 9.1 database. Our system starts with 7 million tweets prepopulated into our databases.

As we vary the rate of arrival of tweets into our system, the rate of update on the base relations (other than tweets) depends on the number of tweets seen so far by the system. For instance, at the beginning any incoming tweet most likely will contain the identity of a user not previously seen by the system, which would result in the insertion of a tuple to the users relation. However, after the system has ingested a sufficient number of tweets, the update rate on the users relation will decrease as some of the users already would be present in the users relation. The dependence between the number of tweets ingested by the system and the *chance* that an incoming tweet will result in an insertion to a base relation is expressed in terms of an *update ratio*. After 7 million tweets have been ingested by our system, the update ratio of encountering a previously unseen user in the next tweet is around 0.3. The update ratio values for some of the remaining relations were 0.25, 0.02, 0.1, 0.2, for socnet, loc, curloc and urls, respectively. Using the update ratios, we can estimate the rate of updates on all the base relations by varying the rate of incoming tweets in the system.

9.1.1 Snapshot Module

To determine the efficacy of our system, an independent auditing module in our sharing executor, called *snapshot*, records the staleness of all the sharings once every 5 seconds. Suppose that a sharing S_j was found to have a staleness less than the SLA staleness at snapshot i. If S_j satisfies the SLA staleness in snapshot $i+1$, then the system is assumed to have maintained S_j at the appropriate level of staleness for all intermediate time periods between i and $i+1$. The converse is true if S_j is found to have violated the SLA at snapshot $i+1$. The snapshot module also keeps track of the cost, and the number of tuples moved between snapshots. Additionally, we record the staleness of all sharings before and after a PUSH operation as well as the cost incurred and time taken for each PUSH operation.

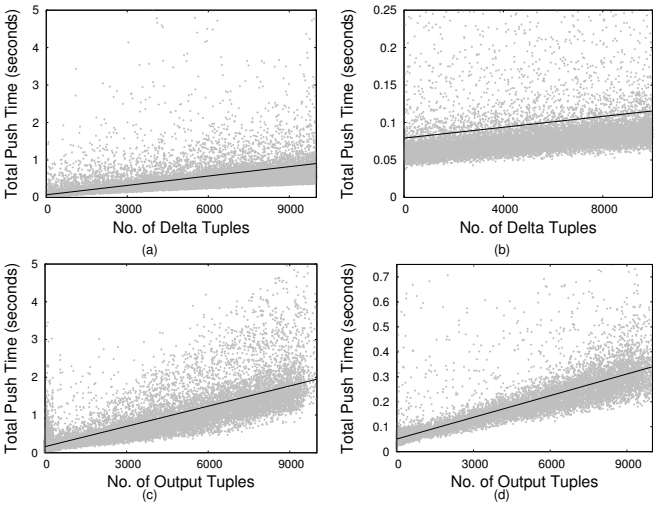

Figure 5: Time cost model of the four edge types, namely a) DeltaToRel, b) CopyDelta, c) Join, and d) Union

9.1.2 Dollar and Time Cost Models

The infrastructure costs for our cost model was obtained from Amazon EC2 pricing available on the web[1]. Our machines are equivalent to large Linux instances, which cost $0.34/hour. For the network cost, we assumed that the instances were in different *availability zone* but in the same *region*, which had a transfer cost of $0.01 per GB. For storage, we used *EBS* storage at $0.11 GB/month.

We developed a time model for each edge type to estimate the time taken to process tuples, as function of the number of input tuples. Our setup to compute a time model consisted of two machines with 15 base relations of varying sizes between 200k and 50 million tuples, number of attributes from 1 to 7 as well as different attribute types forming tens of sharings between the base relations. We pushed a varying number of tuples between 1 and 10k through each edge in the setup and then measured the time taken to perform each PUSH operation, which is recorded in Figure 5. It can be seen that the time taken to push tuples through different edge types is *linear* in the number of tuples for all the edge types, although with different slopes. These plots form the basis of our time cost model.

9.2 Varying Rate

We used 6 machines and 25 sharings as shown in Table 1 with a

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

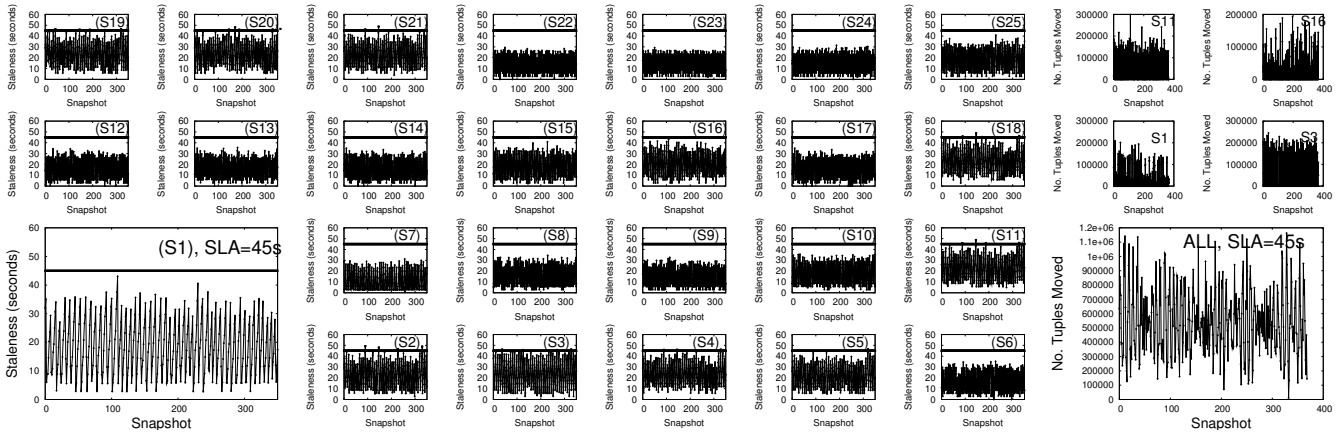

Figure 6: Staleness of twenty-five sharings across snapshots for a rate of 6k tweets/second (left) and the number of tuples moved in each snapshot (right). Due to space constraints, only two zoomed-in graphs are provided for readability, while the remaining figures are rendered small to show trends.

SLA of 45 seconds, while varying the rate of tweets from 50 to 6k tweets/second. At 6k tweets/second (i.e., 3.6 billion tweets/week), the update rate matches the current known volume of tweets in Twitter [12]. We also replayed gardenhose stream, which roughly corresponds to an average of 100 tweets/second. The rate of arrival for tweets for a two hour window is shown in Figure 8c. As the gardenhose is a 1 out of 10 sampling of tweets from the *firehose*, which is a stream containing all the tweets in Twitter, we recreated a stream similar (although by no means equivalent) to firehose by replaying gardenhose at 10X speed.

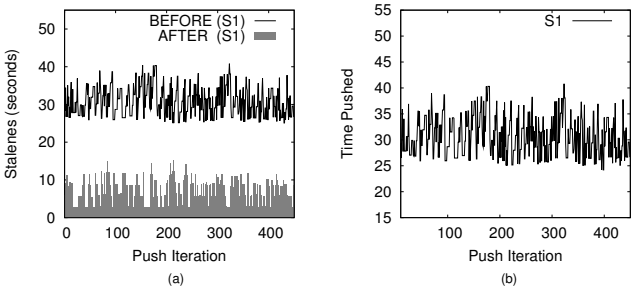

Figure 7: a) Staleness before and after a PUSH on S_1, while b) shows how much is pushed

Figure 6 (left) shows the staleness of a sharing S_1 across different snapshots. It can be seen that the staleness of each of the sharings increases until it comes close to the SLA (i.e., 45 seconds), after which the staleness sharply reduces to a low value due to a PUSH from the sharing executor. The staleness before and after a push operation are shown in Figure 7a, where it can be seen that the PUSH operation reduces the staleness of S1 to less than 10 seconds, just before the staleness of S1 was about to exceed the SLA. Figure 7b shows that every push operation advanced the timestamp of S1 by 25 to 40 seconds, which shows the *lazy* behavior of the sharing executor. One thing to note here is that there were only 31 violations for all the 25 sharings for the entire duration of the experiment lasting about 40 minutes. We summarize some of our observations below.

The number of violations with varying incoming rate of tweets as well as the cost to maintain the sharings in sharing-hour are shown in Figures 8a–b. The results did not show a well defined trend between the number of violations and the rate of incoming tweets, except that the violations were very low, even for 6k tweets/second. First of all, there were zero violations for the firehose (F) and gardenhose (G)

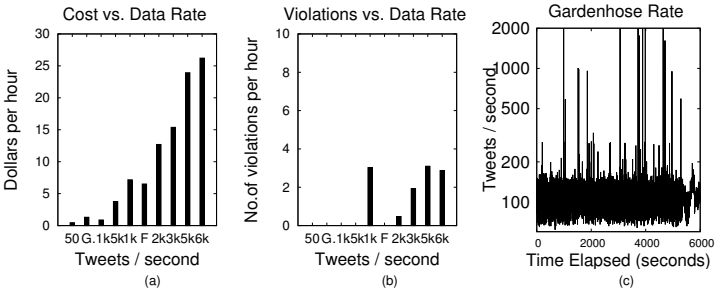

Figure 8: (a) Cost and (b) violations per sharing-hour for gardenhose (G), and firehose (F) streams and rates from 50 to 6k. Variations in Gardenhose (G) rate shown in (c)

streams, and about 3 violations per sharing-hour (i.e., per hour per sharings) for 6k tweets per second. Note that the zero violations for both the gardenhose and firehose streams were in spite of their unpredictable arrival rate, which is shown in Figure 8c. At 6k tweets per second, the cost was about $25 per sharing-hour, although note that some of the sharings were much more expensive than the others. In contrast, the average cost for the firehose (F) stream was about $6 per sharing-hour. Secondly, the number of tuples moved per snapshot (i.e., 5 seconds) across all the sharings was between 600k and 1.1 million tuples as shown in Figure 6 (right).

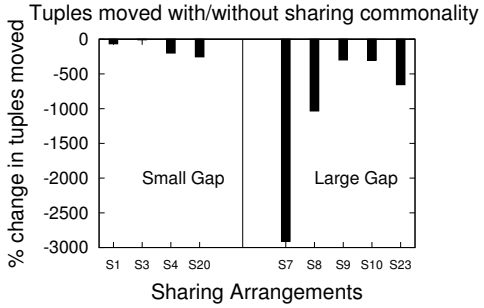

Figure 9: Number of tuples moved in the setup in Figure 6 expressed as percent reduction from the case when individual sharings are run in isolation

Next, notice in Figure 6 (left) that some sharings, such as S7, S8, S9, S10, and S23 have a larger *gap* between the peak staleness value

and the SLA, whereas others such as S1, S3, S4, and S20 have relatively smaller gaps. The reason for this is that those sharings with larger gaps benefit from the commonality with other sharings but not so for those with smaller gaps. To test this hypothesis, we compared the number of tuples moved for each sharing in the above experimental setup with the number of tuples moved when the sharings are run in isolation. The number of tuples moved in the former case is shown as a percentage reduction from the latter case in Figure 9. It can be seen that sharings with small gaps only benefit modestly from the presence of other sharings, whereas those with larger gaps benefit immensely from the presence of other sharings.

9.3 Varying SLA

Table 2: Number of violations per sharing-hour (rounded-up) for varying SLA between 10 and 60 secs

Staleness SLA	10	20	30	40	50	60	Mix
Violations	4	1	2	1	0	0	0

Our setup consisted of 6 machines, 25 sharings and an incoming rate of 1000 tweets/second. We varied the SLA between 10 and 60 seconds. Table 2 shows the effect of varying the SLA in terms of the number of violations. The number of violations is maximum for SLA = 10 seconds at 4 violations per sharing-hour. In general, the number of violations for staleness SLA values greater than 10 seconds is extremely low at either 1 or 2. The higher number of violations for 30 seconds (at 2 per sharing-hour) compared to those for 20 and 40 seconds (at 1 per sharing-hour) was due to temporary fluctuations in system resources.

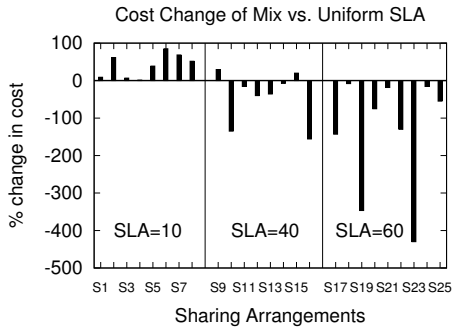

Figure 10: Cost change for *mix* SLA compared to uniform SLA

In the above experimental setup, we also examined a case where we assigned non-uniform SLA (see *mix* in Table 2) to the 25 sharings. In particular, S1–S7 were assigned a SLA of 10 seconds, S8–S15 a SLA of 40 seconds, and S16–S25 a SLA of 60 seconds. From Table 2, we can see that the *mix* case resulted in zero violations, although having comparable dollar costs to the uniform SLA cases. Then in Figure 10 we expressed the cost of an individual sharing in the mix case as a percentage change to the corresponding cost from the uniform case (i.e., compare costs of S1 from mix with uniform when SLA was 10 seconds). It is interesting to note that although the costs of S1–S7 have become marginally more expensive, the cost of the other sharings (i.e., S8–S15, S16–S25) is now significantly cheaper. Hence, we can conclude that few sharings with small SLAs subsidize the operating cost of other (related) sharings.

9.4 Varying Machines and Sharings

In this experimental setup, we varied the machines from 2 to 5, while keeping the number of sharings fixed at 25 and a SLA of 45 seconds. For every setup, the capacity of the machine was determined to be the highest rate of tweets that the set of machines can

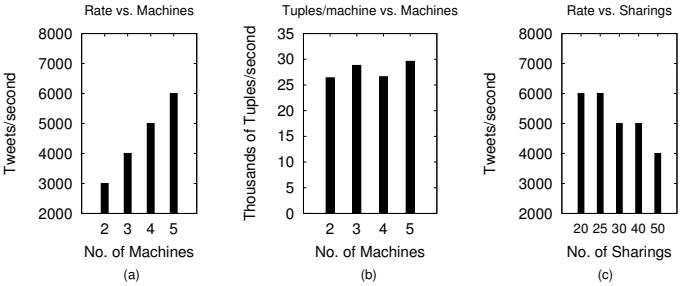

Figure 11: Maximum tweet rate for varying (a) machines, (b) sharings on a setup with SLA = 45 seconds

support without losing the *stability* of the system. We have built an appropriate mechanism to monitor the stability of our system, which is not discussed here due to lack of space. It can be seen from Figure 11a that increasing the number of machines increases the maximum rate that can be handled by our system. Moreover, adding an extra machine increases the processing capacity of our system by at least 25–30k tuples/sec as can be seen from Figure 11b. Next, we varied the number of sharings from 20 to 50 keeping the number of machines fixed at 6 and SLA of 45 seconds, as shown in Figure 11c. We increased the number of sharings beyond 25 by placing the same sharing on more than one machine. With increasing number of sharings, the maximum rate decreases as database and other system bottlenecks start manifesting due to the increased number of vertices and edges that the system has to manage.

9.5 Algorithm Comparisons

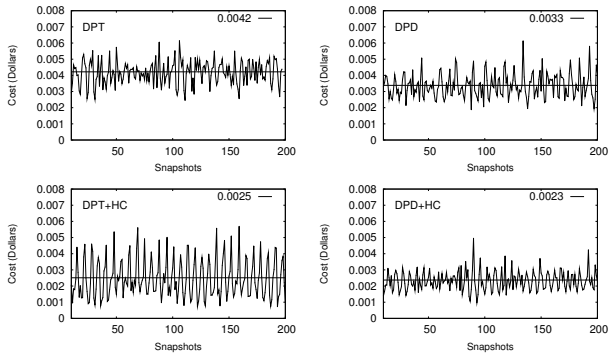

Figure 12: Cost of DPT and DPD reduced by applying hill-climbing algorithm to produce DPT+HC and DPD+HC

We examined the efficacy of the hill-climbing algorithm that we apply to the DPD and DPT algorithms to reduce the cost for the provider. For this experimental setup, we used 6 machines, 25 sharings and a rate of 1000 tweets/second. The cost model we considered was same as before, except that we changed the networking pricing to be within the same availability region in EC2 (i.e., no cost). We generated DPD and DPT sharing plans for this setup, and then applied the hill-climbing algorithm to both these sharing plans to produce DPD+HC, and DPT+HC, respectively. The average cost in dollars per sharing-second for the four sharing plans in sharing-hour were as follows — DPT 0.0042, DPD 0.0033, DPT+HC 0.0025, and DPD+HC 0.0023 as shown in Figure 12. It can be noticed that DPD+HC has the cheapest cost but is comparable to DPT+HC. When we compared DPD with DPD+HC, and DPT with DPT+HC, the difference is quite significant representing a 35% reduction in cost, thus making a case for our hill-climbing approach.

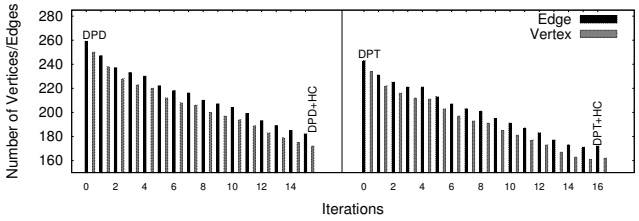

Figure 13: Reduction in vertices and edges as plumbing operations are sequentially applied to DPD and DPT

Figure 13 shows the number of vertices and edges as the hill-climbing algorithm takes DPD or DPT sharing plan as input and performs plumbing operations in a sequential fashion. As can be seen from the figure, the sharing plan is reduced by more than 80 vertices and edges for both DPD and DPT, which represents significant savings in terms of cost.

9.6 Robustness of Sharing Executor

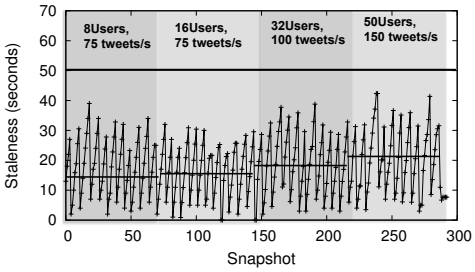

Figure 14: Staleness of S_4 across snapshots as the rate and the workload on the MV abruptly change

This experiment shows the robustness of the sharing executor as the machine capacities change during run-time. The setup consists of 4 machines hosting the sharings $S_1 ... S_4$. Figure 14 shows the staleness of S_4 with an SLA of 50 seconds (marked) as recorded by the snapshot module. The SLA of the remaining 3 sharings was between 20 and 70 seconds. We applied a read workload on each of the four MVs corresponding to the four sharings by the way of a simulated user generating a single query template in a closed loop. Initially, the incoming rate is set at 50 tweets/second and each of the MVs is subjected to two users.

As the system is running we abruptly vary the number of users on each MV as well as the rate of incoming tweets. The number of users per MV is varied in four *phases* from 8 to 50, while simultaneously changing the rate of incoming tweets from 50 to 150 tweets/second. After the first phase when the number of users was increased from 8 to 16 users, all the machines are heavily loaded. The average staleness value for each phase is also marked in Figure 14.

As the number of users and the rate of incoming tweets increase, the machines get progressively more loaded. However, it can be seen from the boundaries of the phase changes in the figure, the sharing executor quickly adapts to the changing data rate and the increased workload on the databases. The model in spite of the infrastructure being loaded never allows the staleness of the sharing to exceed beyond 40 seconds. The sharing executor is able to do this by taking advantage of the slack between the critical time path of the sharing plan, which is a few seconds and that of the staleness SLA, which is 50 seconds. Even if the time taken to push the sharing becomes progressively slower due to the system load, the executor is able to schedule the updates in a way that the SLAs are not violated.

10. CONCLUDING REMARKS

In this paper we presented a platform that can maintain sharings at the appropriate level of staleness. Experimental results showed the effectiveness of our platform in maintaining several sharings with low violations even under a high update rate. We will examine the following possible extensions in a future work. The platform can be extended to support aggregate operators by developing additional operators. Next, easy addition or removal of sharings on the fly as the system is running can be provided. Finally, before markets for sharing data be envisioned, issues related to the pricing of data [9] have to be addressed.

11. REFERENCES

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

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

[3] Postgres streaming replication. `http://wiki.postgresql.org/wiki/Streaming_Replication`, 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*, 1997.

[6] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

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

[8] S. Aulbach, M. Seibold, D. Jacobs, and A. Kemper. Extensibility and data sharing in evolving multi-tenant databases. In *ICDE*, 2011.

[9] M. Balazinska, B. Howe, and D. Suciu. Data markets in the cloud: An opportunity for the database community. *PVLDB*, 4(12):1482–1485, 2011.

[10] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

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

[12] D. Costolo. The power of Twitter as a communication tool. `http://www.fordschool.umich.edu/video/newest/1975704207001/`, 2012.

[13] V. Kantere, D. Dash, G. Gratsias, and A. Ailamaki. Predicting cost amortization for query services. In *SIGMOD*, 2011.

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

[15] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *VLDB*, 2004.

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

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[18] 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*, pages 31–47. Springer, 1985.

[19] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A data-sharing middleware for mobile computing. *Wireless Personal Communications*, 21(1):77–103, 2002.

[20] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, 2001.

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

[22] T. Plagemann, J. Andersson, O. Drugan, V. Goebel, C. Griwodz, P. Halvorsen, E. Munthe-Kaas, M. Puzar, N. Sanderson, and K. Skjelsvik. Middleware services for information sharing in mobile ad-hoc networks. *Broadband Sat. Comm. Sys. and Challenges of Mob.*, 169:225–236, 2005.

[23] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, 1996.

[24] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. In *SIGMOD*, 2000.

[25] J. Sankaranarayanan, H. Hacigumus, and J. Tatemura. COSMOS: A platform for seamless mobile services in the cloud. In *MDM*, 2011.

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

[27] M. M. Wang, J. N. Cao, J. Li, and S. K. Dasi. Middleware for wireless sensor networks: A survey. *JCST*, 23(3):305–326, 2008.

[28] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, 1995.

[29] Y. Zhuge, H. García-Molina, and J. Wiener. The strobe algorithms for multi-source warehouse consistency. In *PDIS*, 1997.