How to Price Shared Optimizations in the Cloud

Prasang Upadhyaya, Magdalena Balazinska, and Dan Suciu
Department of Computer Science and Engineering,
University of Washington, Seattle, WA, USA
{prasang, magda, suciu}@cs.washington.edu

ABSTRACT

Data-management-as-a-service systems are increasingly used in collaborative settings, where multiple users access common data sets. Cloud providers have the choice to implement various optimizations, such as indexing or materialized views, to accelerate queries over these datasets. Each optimization carries a cost and may benefit multiple users. This creates a major challenge: how to select which optimizations to perform and share their cost among users. The problem is especially challenging when users are selfish and will only report their true values for different optimizations if it maximizes their utility.

In this paper, we present a new approach for selecting and pricing shared optimizations by using Mechanism Design. We first show how to apply the Shapley Value Mechanism to the simple case of selecting and pricing additive optimizations assuming an offline game where all users access the service for the same time-period. Second, we extend the approach to online scenarios where users come and go. Finally, we consider the case of substitutive optimizations.

We show analytically that our mechanisms are truthful and cost-recovering: *i.e.*, selfish users are best-served when revealing their true values and the cloud is guaranteed to recover all optimization costs. Through experiments on the SQL Azure cloud, we further show that our mechanisms yield higher utility than the state-of-the-art approach based on regret accumulation.

1. INTRODUCTION

Over the past several years, "cloud computing" has emerged as an important new paradigm for building and using software systems. Multiple vendors offer cloud computing infrastructures, platforms, and software systems including Amazon [4], Microsoft [10], Google [18], Salesforce [31], and others. As part of their services, cloud providers now offer data-management-in-the-cloud options ranging from highly-scalable systems with simplified query interfaces (*e.g.*, Windows Azure Storage [11], Amazon SimpleDB [9], Google App Engine Datastore [19]), to smaller-scale but fully relational systems (SQL Azure [22], Amazon RDS [6]), to data intensive scalable computing systems (Amazon Elastic MapReduce [5], to highly-scalable unstructured data stores (Amazon S3 [8]), and to systems that focus on small-scale data integration (Google Fusion Tables [17]).

Existing data-management-as-a-service systems offer multiple options for users to trade-off price and performance, which we call generically *optimizations*. They include views [3] and indexes (*e.g.*, users can create indexes in SQL Azure and Amazon RDS, Amazon SimpleDB automatically indexes data), but also the choice of physical location of data –which affects latency and price (*e.g.*, Amazon S3, SimpleDB)– how data is partitioned (*e.g.*, Amazon SimpleDB data "domains" or manual partitioning across SQL Azure instances), and the degree of data replication (*e.g.*, Amazon S3 standard and reduced-redundancy storage, Amazon RDS multi-AZ deployment, Amazon RDS read replicas). Cloud systems have an incentive for enabling all the right optimizations, because this increases their customer's satisfaction and can also optimize the cloud's overall performance.

Today, data owners most commonly pay all costs associated with hosting and querying their data, whether by themselves or by others. Data owners also choose, when possible, the optimizations that should be applied to their data. However, there is a growing trend toward letting users collaborate with each other by sharing data and splitting the costs of accessing that data. For example, in the Amazon S3 storage service, users can currently share their data with select other users, with each user paying his or her own data access charges [7]. Furthermore, cloud database-as-a-service systems often co-locate multiple databases on the same infrastructure and even use a multi-tenant configuration where a single database hosts the data owned by different users (*e.g.*, [31]). In such systems, there are optimizations such as replicating the database that improve the services for many users at once and where optimizing the cloud separately for individual users is much costlier than optimizing for the group.

The combination of data sharing and optimizations creates a major challenge: how to *price optimizations when one optimization can benefit multiple users*. Implementing these optimizations imposes a cost on the cloud that needs to be recovered: resources spent on implementing and maintaining optimizations are resources that cannot be sold for query processing. The question is how to decide what optimization to implement and how to share its cost among users.

Proceedings of the VLDB Endowment, Vol. X, No. Y

A recently-proposed approach by Kantere, Dash, *et al.*, [15, 20] addresses this problem by asking users to indicate their willingness to pay for different query performance values, observing the query workload, and deciding on the optimizations to implement based on optimizations that would have been helpful in the past (*i.e.*, based on *regret*). The cost of implemented optimizations is amortized to future queries that use them. This approach, however, has two key limitations as we show in Section 7. First, it assumes that users in the cloud will truthfully reveal their valuations. In practice, users will try to game the system if doing so improves their utility. Second, this approach does not guarantee that the cost of an optimization is recovered.

Given these two observations, we develop a new approach for selecting and pricing optimizations in the cloud based on Mechanism Design [27, 29]. Mechanism Design is an area of game theory whose goal is to choose a game structure and payment scheme such as to obtain the best possible outcome to an optimization problem in spite of *selfish players having to provide some input to the optimization*. Our goal is to enable the cloud to find the best configuration of optimizations. For this, the cloud needs users (*i.e.*, selfish players) to reveal their valuation for these optimizations.

The most closely related approaches from the Mechanism Design literature are cost-sharing mechanisms [24]. Given a service with some cost, these mechanisms consider the problem of deciding what users to service and how much the users should pay for the service. In this paper, we show how to easily adapt this technique from the theory community to the simplest problem of pricing a single optimization when all users will access the system for a single time-period (*i.e.*, offline game).

The problem of pricing optimizations in the cloud, however, raises two additional challenges. First, in the cloud, users change their workloads, join and leave the system at any time. Such dynamism complicates the problem because users now have new ways of gaming the system: they can lie about the time when they need an optimization and they can emulate multiple users. Dynamism requires an online mechanism. Second, multiple optimizations are available in the cloud. In the simple case, the value that a user derives from a set of optimizations is simply the sum of individual optimization values. We call such optimizations additive. In other cases, the total value from a set of optimizations may be given by a more complex function. In this paper, as a first step, we consider substitutive optimizations, where the user only wants to pay for one optimization in a set. For example, a user may be willing to pay either for an index that accelerates a join or for a materialized view that precomputes the join but not both. Prior work in mechanism design does not handle all the requiremens at a time (See Section 7). In this paper, we develop a suite of mechanisms, that can handle all these challenges at the same time.

We seek the following three standard properties for our mechanisms. First, we want the mechanisms to be *truthful*, also known as *strategy-proof* [27], which means that every player should have an incentive to reveal her true value obtained from each optimization. For example, consider a pricing scheme where the cloud asks users how much they are willing to pay for an optimization and divides the optimization cost accordingly. Users will simply lie, hoping that the optimization happens anyway and they can get a free ride or pay a lower price; thus, such a mechanism is not truthful. Similarly, the approach by Dash, Kantere *et al.* [15, 20] mentioned above is not truthful as we discuss in Section 7. We also want online mechanisms to be resilient to multiple identities, which is another way to lie about value, and to misrepresentation of the time when a user needs an optimization. Second, we want the mechanisms to be *cost-recovering*, which means that the cloud should not lose money from performing the optimizations. For example, in the approach proposed by Dash, Kantere, *et al.* [15, 20], the cloud first decides to implement an optimization and then it amortizes the cost to future queries that use it. Cost-recovery is thus not guaranteed. Finally, we want the mechanisms to be *efficient*, also known as *value-maximizing* [27], which means that we want it to maximize the total surplus of the system *i.e.*, the sum of user values minus the cost of the alternative selected. For example, if several users could benefit from an expensive optimization that none of them can afford to pay for individually, then the cloud should perform the optimization and divide the cost among the users.

In summary, we make the following four contributions:

We first show how the optimization pricing problem maps onto a cost-recovery mechanism design problem (Section 2). We also show how the Shapley Value Mechanism [24], which is known to be both cost-recovering and truthful, solves the problem of pricing a single optimization. We propose a direct extension of the mechanism to the case of additive optimizations in an *offline* scenario, where all users access the system for the same time-period. We call this basic mechanism Add^{Off} *Mechanism* (Section 3).

Second, we present a novel mechanism for the online scenario, where users come and go. We call this new mechanism Add^{On} *Mechanism* . Users of cloud services constantly join and leave the cloud, so in practice, optimizations in the cloud need to be designed for a dynamic setting. However, it turns out to be much more difficult to design mechanisms for the online setting: algorithms that are truthful or cost recovering in the static setting cease to be so in the dynamic setting, see [27, pp 412]. We prove formally that our new mechanism is both cost-recovering and truthful in the dynamic setting (Section 4).

Third, we extend both the Add^{Off} Mechanism and the Add^{On} Mechanism to the case where optimizations are interdependent: In this paper, we consider substitutive optimizations, where the user derives a single value for any optimization in a set. However, implementing more than one optimization from the set does not improve the user value. We call these mechanisms $Subst^{Off}$ *Mechanism* and $Subst^{On}$ *Mechanism* and show that they are truthful and cost-recovering (Section 5).

A well-known result is that achieving both truthfulness in face of selfish agents and cost-recovery comes at the expense of total utility [24]. We experimentally compare our mechanisms against the state-of-the art approach based on regret accumulation [15]. We show that our mechanisms produce higher utility upto 3×, provide the same utility for higher ranges of costs upto 12.5× than the state-of-the-art approach in addition to handling selfish users, and all the while ensuring that the cloud does not make a loss.

2. A MECHANISM DESIGN PROBLEM

In this section, we show how the problem of selecting and pricing optimizations in the cloud can be modeled as a *mech-*

anism design [27] problem. We further show that our problem requires a type of mechanism called *cost-sharing mechanism*. In this paper, we assume that all optimizations are binary. That is, the cloud either implements an optimization or not. We do not consider continuous optimizations (*e.g.*, degree of replication).

We consider a set of users, $I = \{1, \ldots, m\}$, who are using a cloud service provider (*a.k.a.*, *cloud*) to access and query several data sets. Any user can potentially access any data set. Let $J = \{1, \ldots, n\}$ denote the set of all potential optimizations that the cloud could offer for these datasets. For example, j may represent an index; or the fact that a data set is replicated in a second data center; or may represent an expensive fuzzy join between two popular public datasets, which is precomputed and stored as a materialized view. Once the cloud decides to do an optimization j, it may restrict access to j to only certain users; a *grant pair* (i, j) indicates that user i has been granted permission to use the optimization j. While grant permissions artificially prevent a user from accessing an optimization, this restriction is necessary to ensure that users reveal their true value for an optimization and pay accordingly. A *configuration*, also called *alternative* is a set of optimizations j and a set of grant pairs[1] (i, j). We denote an alternative with a and the set of all possible alternatives with A. We also denote $S_j = \{i \mid (i, j) \in a\}$ the set of users who are serviced by the optimization j in alternative a.

The goal of the mechanism will be to select a configuration $a \in A$. The decision will be based on the optimization costs and their values to users, which will determine the users' willingness to pay for various optimizations.

Values to Users. Each user i obtains a certain value $v_{ij} \geq 0$ from each optimization j: *e.g.*, monetary savings obtained from increased performance or the ability to do a more complex data analysis. When multiple optimizations are performed, the total value to a user is given by $V_i(a) \geq 0$, and is obtained by aggregating the values v_{ij} for all grant pairs $(i, j) \in a$. In this and the following two sections, we consider *additive optimizations*, where the value is given by:

$$V_i(a) = \sum_{(i,j) \in a} v_{ij} \quad \geq 0 \tag{1}$$

In Section 5 we will consider *substitutive optimizations*.

An important assumption in mechanism design is that users try to lie about their true values: when asked for their value v_{ij}, user i replies with a bid b_{ij}. In the case of an additive value function, we denote $B_i(a) = \sum_{(i,j) \in a} b_{ij}$. Thus:

$$B_i(a) = \text{ User } i\text{'s } \textit{bid} \text{ about her value } V_i(a)$$

Cost to the Cloud. For each implemented optimization $j \in J$, the cloud incurs an optimization cost $C_j > 0$, which includes the initial cost of implementing the optimization (*e.g.*, building an index) and any possible maintenance costs (*e.g.*, updating the index) for the duration of the service. This cost is an opportunity cost: the resources used to perform the optimization cannot be sold to other users. The cost of an alternative a is then given by:

$$C(a) = \sum_{j \in a} C_j \tag{2}$$

[1]We assume that, if an alternative contains a grant pair (i, j), then it also contains the optimization j.

While each individual cost C_j is small, the combined cost $C(a)$ may be significant because the number of potential optimizations is large.

Payments. Once an outcome a is determined, each user i who is granted access to an optimization j must pay some amount p_{ij}. This payment is called the user's *cost-share*, and is determined based on all users' bids[2], $(b_{ij})_{i=1,m;j=1,n}$. Denoting $P_i = \sum_j p_{ij}$ the total payment for the user i, her *utility* is defined as $U_i(a) = V_i(a) - P_i$. A standard assumption in Mechanism Design is that users are "utility maximizers", *i.e.*, they try to bid in a way that maximizes their utility [27, 29].

Cost-Sharing Mechanism Design Problem. After collecting all bids, a mechanism chooses an outcome $a_0 \in A$ that optimizes some global value function. In the case of cloud based optimizations, we will always aim to optimize the *social surplus*, in other words the mechanism will always choose the following outcome a_0:

$$a_0 = \arg\max \sum_{i \in I} B_i(a) - C(a) \tag{3}$$

Such a mechanism is called *efficient* [24]. Note that the mechanism does not know the true values $V_i(a)$, but uses the bids $B_i(a)$ instead. The goal of mechanism *design* is to define the payment functions p_{ij} in such a way that all users have an incentive to bid their true values, $B_i = V_i$. A mechanism is called *strategy-proof* [27, 29], or *truthful*, if no user can improve her utility $U_i(a)$ by bidding untruthfully $B_i \neq V_i$. Truthful mechanisms are highly desirable, because when users reveal their true values, the mechanism is in a better position to select the optimal alternative.

Another desired property for a cost-sharing mechanism is to be *cost-recovering*, meaning that it always chooses an outcome a_0 such that:

$$C(a_0) \leq \sum_i P_i \tag{4}$$

EXAMPLE 2.1. *Consider the following mechanism. The cloud collects all bids b_{ij}. If $c_j \leq \sum_i b_{ij}$ then it performs the optimization j and splits its cost evenly, by asking each user to pay b_{ij} ($p_{ij} = b_{ij}$). Clearly this mechanism is cost recovering. However, it is not truthful: a user i will simply lie and declare a much lower value $b_{ij} \ll v_{ij}$, hoping that the optimization will be performed anyway and she will end up paying much less. The challenge in designing any mechanism is to ensure that it is truthful.*

Formally, a mechanism is defined as follows:

DEFINITION 2.1. *A mechanism $(f, P_1, \cdots, P_m)$ consists of a function $f : \left(\mathbb{R}^A\right)^m \to A$ (called* social choice function*) and a vector of payment functions $P_1, \cdots, P_m$, where $P_i : \left(\mathbb{R}^A\right)^m \to \mathbb{R}$ is the amount that user i pays.*

The mechanisms works as follows. It collects bids $B_1, \ldots, B_m$ from all users[3]. Then it chooses the alternative $a = f(B_1, \ldots, B_m)$, and each user i must pay $P_i(B_1, \ldots, B_m)$.

[2]This is a very important point: the payment depends not only on the outcome a, but on all bids. For example, in the *second bidders' auction*, the payment of the winner is the second highest bid [29].

[3]Each bid B_i is a function $A \to \mathbb{R}$.

While we would like to design mechanisms that maximize the social surplus Eq.(3), it is a well-known result that one cannot achieve cost-recovery (*a.k.a.* budget balance), truthfulness and efficiency [24] at the same time. In our setting, we choose to ensure only truthfulness and cost-recovery, Eq.(4), at the expense of some efficiency loss. Indeed, if the cloud cannot recover its cost, it will not implement the loss-making optimization.

3. A MECHANISM FOR STATIC COLLABORATIONS

We now show how to use the Shapley Value Mechanism [24], which has many desirable properties, to solve the problem of selecting and pricing *additive* optimizations for *one time-period* (*i.e.*, offline game). We extend the mechanism to online settings, where users come and go across multiple time-periods in Section 4 and to substitutive optimizations in Section 5.

3.1 Background: Shapley Value Mechanism

We start by reviewing the Shapley Value Mechanism [24], shown in Mechanism 3.1. Fix a single optimization j, let C_j be its cost and $b_{1j}, \ldots, b_{mj}$ the users' bids for this optimization. The Shapley Value Mechanism determines whether to perform the optimization or not, and, computes the set of serviced users $S_j \subseteq \{1, \ldots, m\}$, and how much they have to pay, p_{ij}. Recall that a configuration, a, contains all grant pairs (i, j) such that $i \in S_j$. The mechanism starts by setting S_j to the set of all users, and divides the cost C_j evenly among them: $p = C_j/|S_j|$. If p is larger than a user's bid b_{ij}, that user is removed from S_j. The mechanism then recomputes a new price by dividing the cost evenly among the smaller set of users. As a result, the cost per user, $C_j/|S_j|$, may increase and additional users may need to be removed from the set S_j. The process continues until either no users remain or no further users need to be removed from S_j. Each serviced user, $i \in S_j$, pays the same amount, $p_{ij} = C_j/|S_j|$; each non-serviced user, $i \notin S_j$, pays nothing, $p_{ij} = 0$. If $S_j = \emptyset$ then no subset of users are bidding enough to pay for the optimization, and the optimization is not performed at all. It it obvious that this mechanism is cost recovering, since $\sum_i p_{ij} = C_j$. The mechanism has also been proven to be truthful [24]: if the user i bids the true value $b_{ij} = v_{ij}$ then her utility (which is $v_{ij} - p_{ij}$, if $i \in S_j$, and 0 otherwise) is larger than or equal to her utility under any other bid. Indeed, suppose the user i bids low, $b_{ij} < v_{ij}$. Then one of two cases holds. Either user i is removed from the set of serviced users S_j: in this case her utility drops to 0. Or user i remains in S_j: in this case her payment p_{ij} remains unchanged, and so does her utility. Hence, she cannot increase her utility by underbidding; the reader may check that she cannot increase it by overbidding.

Mechanism 3.1 Shapley Value Mechanism for computing the set of users to be serviced by an optimization j, and their cost-share p_{ij} .

Shapley-Mech

Input: Optimization cost C_j; bids $b_{1j}, \ldots, b_{mj}$.
Output: Serviced users S_j; cost shares $p_{1j}, \ldots, p_{mj}$

```
S_j ← {1, ..., m} /* the set of serviced users */
repeat
    p ← C_j / |S_j| /* divide cost evenly */
    S_j ← {i | i ∈ S_j, p ≥ b_ij} /* users still willing to pay */
until S_j remains unchanged, or S_j = ∅
p_ij ← p if i ∈ S_j /* serviced users pay same amount */
p_ij ← 0 if i ∉ S_j. /* non-serviced users don't pay */
return (S_j, (p_ij)_{i=1,m})
```

3.2 Add^{Off} Mechanism

We now propose our first mechanism for cloud optimization, under the simplest setting, when the optimizations are done offline and are additive; we will remove these restrictions in the next sections. Our mechanism, called the Add^{Off} Mechanism iterates over all optimizations. For each one, it simply runs the Shapley Value Mechanism. It adds to a the grant pairs for all serviced users and it implements the optimization when the set is not empty. A user pays the sum of all per-optimization payments. Since the Add^{Off} Mechanism simply runs the Shapley Value Mechanism separately for each optimization, it follows directly that it preserves the latter's truthfulness and cost-recovery properties.

As we mentioned above, a known result is that it is not possible for a mechanism to achieve truthfulness and budget-balance while also being efficient. An important property of the Shapley Value mechanism is that it *minimizes welfare loss*, which is the reduction in total utility due to the cost-recovery constraint [24]. We show in Section 6 how this property enables the Add^{Off} Mechanism to achieve high utility in face of selfish users compared to existing, state-of-the art optimization pricing techniques.

4. A MECHANISM FOR DYNAMIC COLLABORATIONS

The simple offline mechanism in the previous section is insufficient for optimizations in the cloud, because cloud users change over time. In this section, we develop a new *online mechanism* for pricing cloud optimizations, which assumes users join and leave the system at any time. In general, if one applies a truthful offline mechanism to an online setting, the resulting mechanism is no longer truthful [27, pp.412]; similarly, applying an offline cost recovering mechanisms to an online setting may render it non-recovering. Our new mechanism is specifically designed for an online setting, and we prove that it is both truthful and cost recovering. We continue to restrict our discussion to additive optimizations (we drop this assumption in the next section), and therefore, without loss of generality, we discuss the mechanism assuming a single optimization j.

The cost of an optimization has two components: an initial implementation cost (*e.g.*, building an index) and a maintenance cost (*i.e.*, cost of index storage and index maintenance). To avoid oscillations where users can afford the initial cost of implementing an optimization but not its maintenance cost, we propose an approach where the cloud computes a single, fixed cost C_j for each optimization, j. That cost captures both the initial implementation cost and the maintenance cost for some extended period of time T (*e.g.*, a month). Users are allowed to join and leave at anytime during T. However, at the end of the time-period T, the cost of the optimization is re-computed and all interested users must purchase the optimization again.

4.1 Add^{On} Mechanism

We divide T into time-slots numbered $\{1, \ldots, z\}$. These time-slots denote the smallest time interval for which a service is provided to any user. If T is a month, slots could correspond to hours, days, or weeks. The value for user i is a tuple $\theta_{ij} = (s_i, e_i, v_{ij})$, where $s_i, e_i \in \{1, \ldots, z\}$ represent the starting time and the ending time when the user benefits from the optimization, and $v_{ij}(t)$ is a function representing her value at time t. The interpretation is the following. At each time t, if $t \in [s_i, e_i]$ and the user gets access to optimization i at time t, then she obtains a value equal to $v_{ij}(t)$; otherwise she does not obtain any value at time t. Her total value is the sum of these unit values over all time slots t. We assume that whenever $t < s_i$ or $t > e_i$ then $v_{ij}(t) = 0$. Of special interest to us is the case when $v_{ij}(t) = v_{ij}$ is a constant value throughout the interval $t \in [s_i, e_i]$.

Users bid for the optimization j, by declaring their values as $\theta_{ij} = (s_i, e_i, b_{ij})$, where $b_{ij}(t)$ is a function of time over the interval $t \in [s_i, e_i]$. Bids are collected by the cloud at each time slot $t \in [1, z]$: a bid cannot be retroactive ($s_i < t$), but users are allowed to revise their future bids ($b_{ij}(t')$, $t' \geq t$) upwards[4]. For example, at time $t = 1$, user 1 bids $(1, 3, [10, 10, 10])$, meaning $b_{1j}(1) = b_{1j}(2) = b_{1j}(3) = 10$; at time $t = 2$ she may revise her bids as $b_{1j}(2) = 20, b_{1j}(3) = 10$. At each time slot t, the cloud needs to determine the set of serviced users $S_j(t)$, based on the current bids. When a user i leaves the system at time e_i, then she has to pay a certain amount p_{ij}.

EXAMPLE 4.1. *Consider one optimization j, with cost $C_j = 100$, and two users with the following values: $\theta_{1j} = (1, 1, [101])$, $\theta_{2j} = (1, 2, [26, 26])$. Thus, user 1 obtains a value of 101 at time slot $t = 1$ if she can access the optimization; user 2 obtains a value 26 at each of the time slots $t = 1$ and $t = 2$, if she has access to the optimization. Consider the following naïve adaptation of the Shapley Value Mechanism to a dynamic setting. Run the mechanism at each time slot, until the mechanism decides to implement the optimization: at that point the cloud has recovered the cost, and will continue to offer the optimization for free to new users. In our example, the optimization will be performed at time $t = 1$, each user pays 50, and user 2's utility is $52 - 50 = 2$. The problem is that the mechanism is not truthful: user 2 may cheat by bidding $(2, 2, [26])$, in other words she hides her value during the first time slot. Now the entire cost of the optimization is paid by user 1, at time $t = 1$, and user 2 gets a free ride at time $t = 2$, obtaining a utility of $26 - 0 = 26$.*

Our Add$^{\text{On}}$ Mechanism, shown in Mechanism 4.1, computes for each time slot $t \in [1, z]$ the set of serviced users $S_j(t)$, and computes the payment p_{ij} for each user i leaving at time t. It works by running a modified Shapley-Value Mechanism at each time-slot t, which we explain next.

Suppose that no users are serviced yet. Then, the regular Shapley-Value Mechanism is run at time t, on the bids $\sum_{\tau > t} b_{ij}(\tau)$. The sum is computed separately for each user and each optimization. If the outcome is not to perform the optimization, then $S_j(t) = \emptyset$ and the system tries the next time slot $t + 1$. If the mechanism decides to perform the optimization, then the system sets $S_j(t)$ to the set of all users served at that time slot, and also continues with the next time slot $t + 1$. As new bids arrive, or future bids $b_{ij}(\tau)$, $\tau > t$ are revised upwards, the cost-shares for all users are re-computed and may be lowered: as a consequence, users that could not be serviced at time t may be serviced at time $t + 1$. Of course, users who no longer need the optimization (because $e_i < t$) are removed from S_j. Denote the cumulative set of serviced users as $CS_j(t) = \bigcup_{\tau \leq t} S_j(\tau)$. The key modification to the Shapley-Value mechanism is to have it operate on $CS_j(t)$ rather than $S_j(t)$. This is ensured as follows: once a user is serviced at some time τ, $i \in S_j(\tau)$, all its future bid are assumed to be ∞: this ensures that the Shapley-Value Mechanism will always include i in $CS_j(t)$. Finally, whenever a user's bid expires, *i.e.* $t = e_i$, then the user's payment is computed at that time slot, by dividing C_j by the number of all serviced users $CS_j(t)$: this is precisely the payment returned by the Shapley-Value mechanism. The users actually serviced, $S_j(t)$, are the active users in $CS_j(t)$, *i.e.* $i \in CS_j(t)$ and $t \leq e_i$: the set of grant permissions (i, j) at time t is $\{(i, j) \mid i \in S_j(t)\}$. In other words, once a user i is serviced, then the user is guaranteed to pay her cost-share, and this helps servicing more users in the future.

EXAMPLE 4.2. *Let's revisit Example 4.1, and assume the users bid truthfully $(1, 1, [101])$ and $(1, 2, [26, 26])$ respectively. At time $t = 1$ both users are serviced, $S_j(1) = CS_j(1) = \{1, 2\}$. User 1 leaves at this time, so she pays $C_j/2 = 50$. At time $t = 2$ user 2 is serviced, hence the cumulative set of serviced users is $CS_j(2) = \{1, 2\}$. User 2 leaves at this time, so she pays $C_j/2 = 50$: her total utility is $52 - 50 = 2$. Assume that user 2 is lying and bids $(2, 2, [26])$. Then $CS_j(1) = \{1\}$ and user 1 pays 100 when leaving. At time 2, user 2 is in no feasible set since the payment required of her is 50 (with $CS_j(2) = \{1, 2\}$) but it exceeds her reported value. Thus user 2 gets a utility of 0 and has reduced her utility by lying.*

EXAMPLE 4.3. *For a more complex example, consider an optimization with cost $C_j = 100$ and four users bidding $(1, 1, [101]), (1, 3, [16, 16, 16]), (2, 2, [26]), (2, 2, [26])$. Then $CS_j(1) = \{1\}$, $CS_j(2) = \{1, 2, 3, 4\}$, $CS_j(3) = \{1, 2, 3, 4\}$. Note that user 2 is not included in $CS_j(1)$ because his bid 48 is below $C_j/2$. At time $t = 2$ his remaining total value is only 32: however, since now there are four users, each users' share is $C_j/4$ and therefore all users are included in $CS_j(2)$, and in $CS_j(3)$. Users 1,2,3,4 leave at times $t = 1$, $t = 3$, $t = 2$, $t = 2$ respectively, so they pay $100, 25, 25, 25$.*

4.2 Properties

We prove that Add$^{\text{On}}$ Mechanism has three important properties: (1) it is truthful, (2) it is cost recovering, and (3) it is resilient to multiple identities, which is another way of lying about the value.

Truthful. The definition of a truthful mechanism in the dynamic setting is more subtle than in the static setting. In a static scenario, the mechanism is called truthful if for *any set of bids*, user i cannot obtain more utility by bidding $b_{ij} \neq v_{ij}$ than by bidding her true value $b_{ij} = v_{ij}$. In the dynamic case, user utilities depend not only on the other bids happening until now, but also on what will happen in the future. We assume the *model-free* [27] framework to define truthfulness in the dynamic case: it assumes that bidders have no

[4]As a consequence, e_i can only increase.

Mechanism 4.1 Add^{On} Mechanism .

Input: Optimization j; cost C_j; bids $(s_i, e_i, b_{ij})_{i=1,m}$.
Output: Serviced users $(S_j(t))_{t=1,z}$; payments $(p_{ij})_{i=1,m}$

```
CS_j(0) ← ∅    p_ij ← 0, ∀i = 1, m
for each time slot t = 1, z do
  for each user i = 1, m do
    if i ∈ CS_j(t − 1) then
      b'_ij ← ∞ /* force user i to be serviced */
    else if t ≥ s_i then
      b'_ij ← Σ_{τ≥t} b_ij(τ) /* remaining value know at t */
    else
      b'_ij ← 0 /* prune users not yet seen */
    end if
  end for
  /* Update the set of serviced users */
  (CS_j(t), (p'_ij)_{i=1,m}) ← Shapley-Mech(C_j, (b'_ij)_{i=1,m})
  S_j(t) ← {i | i ∈ CS_j(t), t ≤ e_i} /*service active users*/
  for i = 1, m do
    if e_i = t then
      p_ij ← p'_ij /* user i pays when her bid expires */
    end if
  end for
end for
return ((S_j(t))_{t=1,z}, (p_ij)_{i=1,m}).
```

knowledge of the future agents and their preferences. At each time t, every agent assumes their worst utility over all future bids, and they act in order to maximize this worst utility [27].

EXAMPLE 4.4. *Consider Example 4.3. User 2 bids* $(1, 3, [16, 16, 16])$*, thus he could obtain a value 16 at each of the three time slots* $t = 1, 2, 3$*; but he is serviced only at time slots* $t = 2, 3$*, hence his value is* $16 + 16 = 32$*. He pays* 25*, thus his utility is* $32 - 25 = 7$*. Suppose that he cheats, by overbidding* $(1, 3, [17, 17, 17])$*. Now he is serviced at all three time slots, but still pays only* 25 *(because when he leaves there are four users in* CS_j*). Thus, for the particular bids in Example 4.3, user 2 could improve his utility by cheating. In a model-free framework, however, users do not know the future, and they must assume the worst case scenario. In our example, the worst case utility for user* 2 *at time* $t = 1$ *(when he places his bid) corresponds to the case when no new bids arrive in the future: in this case, if he overbids* ≥ 50*, he ends up paying 50, and his utility is* $48 - 50 = -2$*. If he underbids, his worst case utility is still 0. By cheating at time* $t = 1$*, user 3 cannot increase his worst case utility.*

With the *model-free* notion of truthfulness [27], a dynamic mechanism is called truthful if, for each user, revealing his true preferences maximize the minimum utility he can receive, over all possible future users' preferences. This definition of truthfulness reduces to the classic definition of truthfulness for the static case, if we assume a single time slot, $z = 1$.

PROPOSITION 4.1. *Add^{On} Mechanism is truthful.*

PROOF. (Sketch) Consider a user i bidding at time t, *i.e.*, his bid is (s_i, e_i, b_{ij}) and $t \leq s_i$ (bids cannot be placed for the past). We claim that its minimum utility over all future user's preferences (at times $t + 1, t + 2, \ldots$) is when no new bids arrive in the future. Indeed, any new bids in the future can only decrease the payment due by user i (by increasing the set $S_j(e_i)$, hence decreasing his payment $p_{ij} = \frac{C_j}{|S_j(e_i)|}$), and can only increase his value at every future time slot $t' \leq s_i$, by including i in a set $S_j(t')$ where it was previously not included. Thus, the minimum utility for user i is when no new bids arrive after time t. But in that case, Add^{On} Mechanism degenerates to one round of the Shapley-Value Mechanism, run at time t, which we saw was truthful. □

Cost-recovering. We prove:

PROPOSITION 4.2. *Add^{On} Mechanism is cost-recovering.*

PROOF. Consider the last time slot of the algorithm, when $t = z$. Assume w.l.o.g. that $CS_j(z) \neq \emptyset$: otherwise, if $CS_j(z) = \emptyset$, then the optimization is not implemented at all during the time period $T = \{1, \ldots, z\}$, and the cost-recovering property Eq.(4) holds trivially. Let p'_{ij} be the payments determined by Shapley-Value Mechanism for the time slot z (see Mechanism 4.1): by definition, this mechanism ensures $\sum_i p'_{ij} = C_j$. Consider any user i. We claim that its real payment is $p_{ij} \geq p'_{ij}$. Indeed, if $i \notin CS_j(z)$ then $p_{ij} = p'_{ij} = 0$, otherwise $p_{ij} = C_j/|CS_j(e_i)|$ and $p'_{ij} = C_j/|CS_j(z)|$ where e_i is the time when the users' bid expires, and the claim follows from the fact that $CS_j(e_i) \subseteq CS_j(z)$. Hence, $\sum_i p_{ij} \geq \sum_i p'_{ij} = C_j$, proving the proposition. □

Multiple Identities. A user could create multiple identities and place a separate bid for each identity. If at least one identity is given access to the optimization, then the user obtains her full value (by running her queries under that identity). However, the user is responsible for paying on behalf of all identities. It turns out that a user can increase her utility this way: by creating more identities, she could help many more users to be serviced and thus decrease her total payment. For a simple example, consider an optimization whose cost is $C_j = 101$ and a user Alice whose value is $(1, 1, [101])$. Suppose there are 99 other users whose values are $(1, 1, [1])$. Of the 100 users, only Alice is serviced, because even if all the other 99 users were serviced, each payment would be $101/100 = 1.01$ which exceeds their value of 1. However, if Alice creates two identities, each bidding (say) $(1, 1, [101])$, then Add^{On} Mechanism will see 101 users, and now it can service all of them. Each user pays $101/101 = 1$. Alice pays 2, once for each identity. Thus, her utility has increased from $101 - 101 = 0$ to $101 - 2 = 99$. Add^{On} Mechanism does not prevent such ways of gaming the system, because they are indistinguishable from collaborations. For example, instead of cheating, Alice could ask her friend Bob (whose value is at least 1) to participate in the game, then reimburse him for his payment: this is indistinguishable from creating a fake identity. On the other hand, there is nothing wrong with that: through her action, Alice helped more users being serviced, accepting to pay slightly more than the share of the other users. We can prove that this holds in general.

PROPOSITION 4.3. *Suppose a user* i *can increase her utility under Add^{Off} Mechanism or Add^{On} Mechanism by creating multiple identities* $i_1, i_2, \ldots$ *Then no other users' utility decreases.*

Mechanism 5.1 Subst$^{\text{Off}}$ Mechanism : Cost-sharing mechanism for *substitutable* optimizations for a *single* slot.

Input: Opts. J; costs $(C_j)_{j=1,n}$; bids $(b_{ij})_{i=1,m;j=1,n}$
Output: Alternative $a \in A$; cost shares $(p_{ij})_{i=1,m;j=1,n}$

```
a ← ∅    p_ij ← 0, ∀i = 1, m  ∀j = 1, n
loop
  for each optimization j in J do
    /* Compute serviced users, discard payments */
    (U_j, (p'_ij)_{i=1,m}) ← Shapley-Mech(C_j, (b_ij)_{i=1,m})
  end for
  /* Find the smallest cost-share optimization */
  J^f ← {j ∈ J | U_j ≠ ∅} /* Set of feasible opts */
  if J^f ≠ ∅ then
    j_min ← arg min_{j∈J^f} (C_j/|U_j|)
    a ← a ∪ {j_min} /* Perform optimization j_min */
    for each user i ∈ U_{j_min} do
      a ← a ∪ {(i, j_min)}
      p_{ij_min} ← C_{j_min}/|U_{j_min}|
      b_ij ← 0   ∀j ∈ J /* Remove i from future loops */
    end for
    C_{j_min} ← ∞ /* Remove j_min from future loops */
  else
    return (a, (p_ij)_{i=1,m;j=1,n})
  end if
end loop
```

Mechanism 5.2 Subst$^{\text{On}}$ Mechanism Cost-sharing mechanism for *substitutable* optimizations, for *multiple* slots.

Input: Opts J; costs $(C_j)_{j=1,n}$;
bids $\omega_i = (s_i, e_i, (b_{ij})_{j=1,n})_{i=1,m}$.
Output: Serviced users $(S_j(t))_{t=1,z}$; payments $(p_{ij})_{i=1,m}$

```
a ← ∅   p_ij ← 0, ∀i = 1, m
for each time slot t = 1, z do
  for each user i = 1, m do
    if ∃j ∈ J. (i, j) ∈ a then
      b'_ij ← ∞ /* force user i to ber serviced */
      b'_ij' ← 0  ∀j' ∈ J, j' ≠ j /* force i to only use j */
    else if t ≥ s_i then
      b'_ij ← Σ_{τ≥t} b_ij(τ) /* remaining value know at t */
    else
      b'_ij ← 0 /* prune users not yet seen*/
    end if
  end for
  /* Update the set of serviced users */
  (a, (p'_ij)_{i=1,m;j=1,n}) ← Subst^Off(J, (C_j)_{j=1,n}, (b'_ij)_{i=1,m;j=1,n})
  S_j(t) ← {i | ∃j.(i, j) ∈ a, t ≤ e_i}
  for i = 1, m do
    if e_i = t then
      p_ij ← p'_ij /* user i pays when her bid expires */
    end if
  end for
end for
return ((S_j(t))_{j=1,n;t=1,z}, (p_ij)_{i=1:m,j=1:n})
```

PROOF. (Sketch) Consider two games, one with user i with a single account and one with user i creating k identities $i_1, \ldots, i_k$ and associated bids. Her utility can increase by creating dummy identities only if the total payment by the dummies is less than the total payment without the dummies. Let user i's payment with no dummies be p_i and the total payment of her dummies be p'_i. Since creating dummies increases i's utility $p'_i < p_i$, and the payment per dummy (which would be the payment per user as well with the dummy accounts) is $p'_i/k < p'_i < p_i$. Thus, for all users served in the game with no dummies are surely served with dummies too since the payment per user did not increase. Hence the utility of no user decreases. □

5. MECHANISMS FOR SUBSTITUTABLE OPTIMIZATIONS

In this section, we relax the requirement that optimizations be independent. Indeed, when multiple optimizations (*e.g.*, indexes or materialized views) exist, the value to the user from a set of optimizations can be a complex combination of individual optimization values. In this section, we consider the case of substitutable optimizations. Formally, each user defines a set of substitutable optimizations $J_i \subseteq J$ such that $\forall j, k \in J_i : v_{ij} = v_{ik} = v_i > 0$. Additionally, given an outcome a, $V_i(a) = v_i$ if $\exists j \in J_i : (i, j) \in a$ and $V_i(a) = 0$ otherwise. In comparison to the substitutable valuation, the valuation function that we previously used was the *sum*: $V_i(a) = \sum_{(i,j)\in a} v_{ij}$. With substitutable valuations, a user bid takes the form $\theta_i = (J_i, v_i)$, where J_i is the set of substitutable optimizations and v_i is the user value if he is granted access to at least one optimization in J_i.

Substitutable optimizations capture the case where implementing any optimization from a set (*e.g.*, indexes, materialized views, or replication) can speed-up a workload by a similar amount. The user values this total speed-up at v_i and does not have any strong preference as to which of the possible optimizations is implemented to obtain the performance gain. However, the user gets no added value from multiple optimizations being implemented at the same time either because the optimizations cannot be used together (*e.g.*, a materialized view may remove the need for a specific index) or because the user gets no added value from further performance gains.

5.1 Subst$^{\text{Off}}$ Mechanism

We first consider the static game where all users bid and use the system for the same time-period.

EXAMPLE 5.1. *Consider a set of three optimizations with costs* $C_1 = 60$, $C_2 = 180$, *and* $C_3 = 100$. *The bid* $(\{1, 2\}, 100)$ *indicates that a user has value* 100 *if he is granted access to either optimization 1 or 2. Three other example bids include* $(\{3\}, 101)$, $(\{1, 2, 3\}, 60)$, *and* $(\{2\}, 70)$.

The challenge with substitutable optimizations is that users may define overlapping but different sets of optimizations as in Example 5.1. Users also have several new ways of cheating. In addition to lying about their value v_i, they can intentionally drop some optimizations from their set. For example, user 3 could bid $(\{2, 3\}, 60)$ instead of $(\{1, 2, 3\}, 60)$ hoping to improve utility. They can also emulate multiple users with different optimization sets. Our mechanisms are truthful and resilient for the former but not for the latter.

To address this challenge, we develop the mechanism shown in Mechanism 5.1. The Subst$^{\text{Off}}$ Mechanism first runs the Shapley Value mechanism for each optimization independently. It then selects the optimization that yields the lowest cost-share for a non-empty set of serviced users. These users will be serviced by that optimization at the com-

puted cost-share. The mechanism then repeats the analysis for the remaining users and optimizations.

EXAMPLE 5.2. *Consider example 5.1. We have three optimizations with costs* $C_1 = 60$, $C_2 = 180$, *and* $C_3 = 100$ *and four bids* $(\{1,2\},100)$, $(\{3\},101)$, $(\{1,2,3\},60)$, *and* $(\{2\},70)$. *The mechanism first identifies optimization* 1 *as having the lowest cost-share with* $U_1 = \{1,3\}$ *and cost-share* $\frac{60}{2} = 30$. *The mechanism thus implements optimization* 1 *and services users* 1 *and* 3. *Next, the mechanism considers the remaining users (user* 2 *and* 4*) and the remaining optimizations (optimizations* 2 *and* 3*). For these optimizations,* $U_2 = \emptyset$ *while* $U_3 = \{2\}$. *Optimization* 3 *is thus implemented and user* 2 *is given access to it. User* 4 *does not get access to any optimization.*

Due to space constraints we defer the proof that Subst$^{\text{Off}}$ Mechanism is cost-recovering to the technical report [36]. It follows directly from the mechanism's construction.

PROPOSITION 5.1. *The Subst^{Off} Mechanism is truthful.*

PROOF. We prove by induction on $|J|$. For any user i the following holds.

Base case: When $|J| = 1$, the mechanism is identical to Add$^{\text{Off}}$ Mechanism which is truthful for single optimizations (refer to Section 3.2).

Inductive case: Now, assume that the mechanism is truthful for $|J| \leq n$. Consider $|J'| = n+1$. Let j be the optimization found by Mechanism 5.1 with the minimum cost-per-user, p_{ij}, with feasible user set U_j. If $i \in U_j$, increasing her bid $b_{ij} > v_{ij}$ will not reduce p_{ij} (and hence not change her utility). Similarly, reducing $b_{ij} < v_{ij}$ leads to either the same value for p_{ij} (so her utility is unchanged) or increases p_{ij} enough to lead to the denial of optimization j to i and a zero utility. User i might still get serviced a higher-priced optimization but that would also reduce i's utility. If $i \notin U_j$, then

1. the minimum price to access j is more than i's value for j and hence increasing her bid to obtain the optimization would lead to negative utility.
2. $v_{ij} = 0$: in this case, i might want to increase p_{ij} for some j with the hope that j will not get implemented and hence some users from U_j might contribute to another optimization j' that i is interested in. However, bidding any non-negative value for j can only decrease p_{ij} further and increasing the bid for an optimization $j' \neq j$ has no impact on p_{ij}. If i belongs to the feasible set of optimization j' then increasing her bid will not reduce $p_{ij'}$ below p_{ij} since increasing the bid beyond $p_{ij'}$ does not decrease $p_{ij'}$. Reduce $b_{ij'}$ below $p_{ij'}$ will remove i from j' service set and render a utility of zero from j'. If i does not belong to the feasible set of any optimization j' that it is interested in it implies that the minimum price to access j' is more that i's value for j' and increasing her bid to obtain the optimization would lead to negative utility.

Thus, the optimization j with the minimum cost per user is implemented and $I \leftarrow I \setminus U_j$ and $J \leftarrow J' \setminus \{j\}$.

By induction, the mechanism will be utility-maximizing, and hence truthful, for the smaller set of users and the smaller set of optimizations. □

EXAMPLE 5.3. *Consider example 5.2. User* 3 *bids* $(\{1,2,3\},60)$ *and is given access to optimization* 1 *at cost-share* 30 *for a utility of* $60 - 30 = 30$. *If user* 3 *increased her bid, she would not affect the outcome and would get the same utility. Similarly, if user* 3 *lowered her bid to a value in the range* $[30,60]$, *the outcome and her utility would remain unchanged. If the user bid below* 30, *however, she would not be serviced by optimization* 1 *as her bid would be below the cost-share. The user would not get serviced by any other optimization, either, because their cost-shares are higher than that of* 1, *which was the optimization with the lowest cost-share. The user utility would be* $(0 < 30)$. *Finally, if user* 3 *dropped optimization* 1 *and bid* $(\{2,3\},60)$, *then both optimization* 1 *and* 2 *would tie for lowest cost-share at* 60. *Let us assume the mechanism would randomly choose and implement optimization* 2, *user* 3 *would be granted access to this optimization and would pay the cost-share of* 60 *achieving a strictly lower utility of* 0.

5.2 Subst$^{\text{On}}$ Mechanism

We now consider substitutable optimizations, but in a dynamic setting where users can join and leave the system in any time-slot. Given substitutable optimizations J_i, user i bids $\omega_i = (s_i, e_i, b_i, J_i)$, with $[s_i, e_i]$ as the requested interval of service and $b_i(t)$ is the value she gets at time t.

Subst$^{\text{On}}$ Mechanism, shown in Mechanism 5.2, works by running the Subst$^{\text{Off}}$ Mechanism at each time-slot t with the residual value of all the users seen. The first time a user i is granted access to optimization j his bid for j is updated to ∞ so that he is always in the feasible set of j. His bid for the other optimizations are updated to 0 so that he remains serviced by optimization j.

Due to space constraints we defer the proof that Subst$^{\text{On}}$ Mechanism is cost-recovering to the technical report [36].

PROPOSITION 5.2. *The Subst^{On} Mechanism is truthful.*

PROOF. (Sketch) We claim that for all known users at time slot t their minimum utility over all future users' preference (at times $t+1, t+2, \ldots$) is when no bids arrive in the future. Indeed, any new future bids can only reduce the payment due by user i by increasing the set $S_j(e_i)$, hence decreasing his payment $p_{ij} = C_j/|S_j(e_i)|$. It can also only increase his value at every future time slot $t' \leq s_i$, by including i in a set $S_j(t')$ where it was previously not included. Thus, the minimum utility for user i is when no new bid arrive after time t. In that case, however, Subst$^{\text{On}}$ Mechanism reduces to Subst$^{\text{Off}}$ Mechanism, executed at time t, which we saw was truthful in Proposition 5.1. □

Multiple Identities. Unlike for Add$^{\text{Off}}$ and Add$^{\text{On}}$ Mechanisms, for Subst$^{\text{Off}}$ and Subst$^{\text{On}}$ Mechanisms dummy users can increase their own utility at the expense of other users as the following example shows. Consider users $\{1,2,3\}$ with single-slot bids $(\{1\},5)$, $(\{1,2\},2.51)$, and $(\{2\},7)$ for optimizations 1 with cost 6 and 2 with cost 5. With no dummy users, optimization 2 is implemented with a payment of 2.5 and utilities of 0.01 for user 2 and 4.5 for user 3. If user 1 creates two identities $1'$ and $1''$ that make a bid of 2.5 each for optimization 1, then both optimizations are implemented with optimization 1 serving $\{1', 1'', 2\}$ utilities of 1, 0.51, and 2 for users 1, 2, and 3 respectively. Note that user 3's utility has reduced. Thus, extenal checks by the cloud are needed to prevent such forms of cheating.

6. EVALUATION

Our mechanisms guarantee truthfulness and ensure cost-recovery, but they do not optimize total utility (*i.e.*, sum of all user values minus the cost of all implemented optimizations). In this section, we empirically evaluate the utility that our solutions provide. We focus on the online mechanisms (*i.e.*, Add^{On} Mechanism and Subst^{On} Mechanism). For brevity, we use the term "mechanism" instead of the full name of the mechanism when the context is obvious. We compare our mechanisms to a regret-based approach (see Section 6.1) as recently proposed in the literature [15, 20]. The experiments are done through simulations[5].

Evaluation roadmap. The evaluation assesses the total utility of the alternatives selected by our mechanisms and the regret-based approach. The key parameter that affects utility is the relative cost of optimizations compared with the user values. This parameter affects the number of users that are necessary to cover the cost of an optimization. In all graphs, we vary this ratio by varying the per-optimization cost along the x-axis while keeping user values within a fixed range. The second key parameter is how the user values are distributed across available optimizations and over time. We study the impact of this parameter as follows. First, we consider the case where users bid for single time-slots and we vary the size of user groups interested in a single optimization (Section 6.2). Second, we study how sequential and concurrent need for an optimization affect total utility and cloud balance (Section 6.3). In Section 6.4, we look at how usage skewed in time affects total utility and cloud balance: we consider a uniform usage pattern and two non-uniform ones that model early and late interest in an optimization. Finally, in Section 6.5, for substitutable optimizations, we look at how varying the selectivity in choosing the set of substitutes affects the total utility and the cloud balance. All graphs show the average and standard deviation for 1000 independent runs.

6.1 Regret-Based Amortization

Prior work [15, 20] proposed to use a regret-based approach for selecting optimizations. This work developed an entire intricate economy and considered detailed query plans for computing regret. In this paper, we extract and evaluate the performance of the core regret-based approach without the surrounding economy nor query plan details.

Regret for optimization j at time t, termed $R_j(t)$, is defined as the total value, over all users, until time t (and excluding it) that would have been realized had j been implemented and the users serviced. Formally, $R_j(t) = \sum_{\tau<t} \sum_{i \in I} v_{ij}(\tau)$. The policy we adopt as to when to implement the optimization is the greedy approach [27] where the optimization is implemented at time slot t when $c_j \leq R_j(t)$. For the case of substitutable optimizations, once an optimizations j is implemented for a user i, user i stops contributing to the regret of other optimizations in $J \setminus \{j\}$.

To recoup the cost c_j, each future user who gets access to optimization j pays a price p_j until the cost is amortized. Let t_r^j be the time at which the regret-based approach implements j. To fix p_j, we look at the remaining value in the game assuming perfect knowledge of future users and their values. We choose a p_j that minimizes the cloud loss. Let $U_j(p, t_r^j) = \{i \mid \sum_{t>t_r^j} v_{ij}(t) \geq p\}$. Then $p_j = \arg\min_p \max\{c - p \times |U_j(p, t_r^j)|, 0\}$. (In case of multiple choices for p_j we choose the one with the lowest magnitude since that maximizes the user utilities.) Thus, our price point is the optimal choice to minimize the cloud loss. It over-estimates how well regret would work in practice.

Our approach thus computes regret the same way as Kantere, Dash, *et al.* [20, 15] except that, in their approach, users assign values to individual queries. Our approach aggregates this information and assigns values to workloads spanning larger ranges of time.

6.2 Collaboration Size

We compare the regret and mechanism-based approaches for two different collaboration sizes. For both approaches, larger collaborations should enable the implementation of more expensive optimizations and yield higher utilities. We let users pick *one* service slot, uniformly at random, from 12 slots[6]. We experiment with group sizes of 6 and 24 to simulate low and high collaborations with expected number of users per slot being 0.5 and 2, respectively.

6.2.1 Additive Optimizations

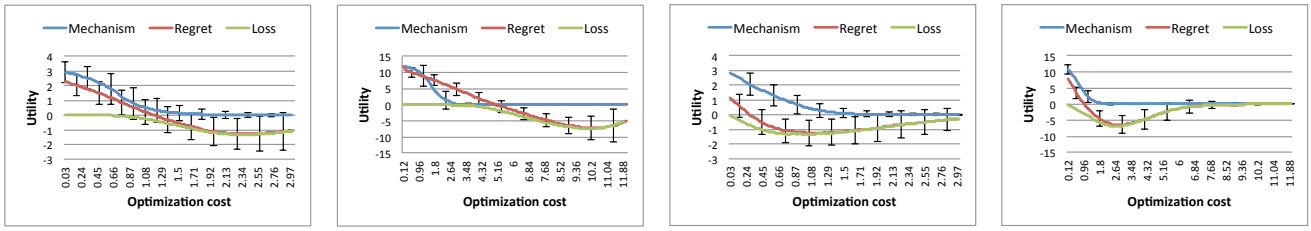
We first consider additive optimizations with a single optimization since optimizations are independent.

Figure 1(a) shows the results for the small collaboration size. As the figure shows, the Regret Algorithm works well for inexpensive optimizations but can quickly lead to cloud loss and even negative total utility. When considering only the optimization costs where the Regret Algorithm yields a zero or positive utility, the mechanism-based approach achieves an average utility 1.43× higher than the Regret Algorithm. Further, the mechanism yields substantial positive utility (taken as 10% of total user value) for optimization costs that are 7× larger than the cost at which the Regret Algorithm starts losing money for the cloud. The Regret Algorithm underperforms compared to the mechanism for two reasons. First, for inexpensive optimizations that should be implemented, the Regret Algorithm loses some user value building up regret. Second, for costly optimizations, the Regret Algorithm implements the optimization even when the values from future users are insufficient to pay for the optimization's cost, hence incurring a negative utility.

For larger collaboration sizes, as shown in Figure 1(b), the mechanism provides worse utility than the Regret Algorithm for a subset of costs. Intuitively, the mechanism looses some opportunities to implement optimizations because it is more cautious than the Regret Algorithm: To avoid losses, the mechanism only implements an optimization when it is certain to recoup the costs given current information. The benefit of regret, however, is limited: only in 5% of the range where the Regret Algorithm achieves a non-zero utility, it also outperforms the mechanism *and* yields no loss. Over all

[5]For context, on SQL Azure, storing and querying a 144 GB scientific dataset [1] costs $1600/month [23]. We find that an index on a commonly queried column requires 14GB for an added cost of $200 in storage per month. The index speeds-up common queries by up to 8X. Our simulation numbers assume that scientists would be willing to pay, on average, $100, for this optimization, with values varying between $0 and $200. Time-slots correspond to months.

[6]The number 12 was chosen since 2, 3, 4, and 6 divide it perfectly and give us a larger space of parameter values to experiment with as compared to some other number like 10 or 15. The other parameter values were chosen to be multiples of 12 for ease of understanding.



(a) Users = 6 (b) Users = 24 (c) Users = 6 (d) Users = 24

Figure 1: All users bid for a uniformly random time slot from 12 slots. User values are uniformly random in $[0, 1)$. Figures 1(a) and 1(b) show average utility of 1000 runs for the additive case and a single optimization, while Figures 1(c) and 1(d) do so for the substitutive case where each user chooses 3 uniformly random substitutes out of 12 optimizations. The figures also show the cloud balance with regret. X- and Y-axes ranges are different in figures (a) and (c) compared with (b) and (d). Error bars show standard deviations.

the costs from 0 to 3.0 the average utility of the mechanism is 0.87 while that of the Regret Algorithm is -0.63.

For large collaborations, the mechanism utilities sharply decrease after a point because of the definition of a feasible set of users: when costs increase, the payment per user increases super-linearly, since the mechanism prunes out users for whom the payments are larger than the value. No users are pruned by the Regret Algorithm and thus it sees a linear reduction in utilities with increasing costs.

Interestingly, the range of costs for which the Regret Algorithm makes a loss depends on the total number of users who bid. It yields a loss at a cost of 0.18 for the small group (Figure 1(a)) and 1.80 for the large one (Figure 1(b)). Since the number of users is not known in advance, the cloud can not know when now to use the Regret Algorithm.

6.2.2 *Substitutive Optimizations*

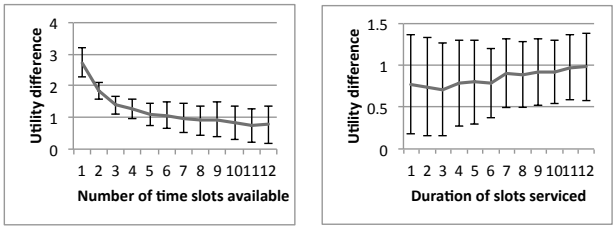
To compare the mechanism-based and regret-based approaches in the case of substitutive optimizations, we consider a scenario with 12 optimizations. Each user selects 3 optimizations, uniformly at random, as the set of substitutes (Section 6.5 experiments with other ratios). Unlike for the experiments in the additive case, the costs of the 12 optimizations are sampled uniformly from $[0, 2c]$ so that c is the average optimization cost: this is to simulate that not all substitutes are equally expensive. Thus the x-axes of Figures 1(c) and 1(d) are the mean value of the optimizations.

Compared to the corresponding additive optimizations in Figures 1(a) and 1(b), we see that both the mechanism and the Regret Algorithm achieve lower overall utility. Indeed, with substitutes, each optimization has fewer users bidding for it and, once an optimization is implemented, the serviced users no longer contribute to the other optimizations. Hence, fewer optimizations are implemented and, in the case of regret, there are fewer users over whom the costs can be amortized. In the scenarios shown, the Regret Algorithm yields a loss earlier than in the additive case.

When averaged over those costs for which the Regret Algorithm yields either zero or positive utility, the mechanism yields 1.63× and 3× the utility achieved by the Regret Algorithm for group sizes of 24 and 6, respectively.

6.3 Overlap in Usage

In this section, we fix the collaboration size at 6, but we vary the degree of user overlap and the manner in which overlap occurs. We consider a single, additive optimization.



(a) The number of available slots for bidding varied on the x-axis.

(b) The service duration is varied on the x-axis. Start times uniformly random in $\{1, \ldots, 12\}$.

Figure 2: User values and cost identical to the setup in Figure 1(a). The y-axis is the average of the difference in utility between the mechanism and the Regret Algorithm over optimization costs in $[0, 3.0]$.

We first repeat the experiment from Figure 1(a) but slowly decrease the total number of slots from 12 to 1. Figure 2(a) shows the results. As the figure shows, with fewer slots to sample from and hence with increased overlap amongst users, the mechanism generates 0.77 to 2.75 more utility, on average, than the Regret Algorithm. (For context, the entire value in the game is 3.0, which sets an upper bound on the utility.) As we decrease the number of time-slots, the probability increases that the mechanism finds enough value in some slot to justify implementing the optimization. In contrast, regret accumulation stays unchanged.

Next, we study what happens when user values are spread across an interval rather than being concentrated in a single time-slot. The setup in Figure 2(b) is identical to the additive case with the group size of 6 in Figure 1(a) except that instead of bidding for only one slot, users bid as $(s_i, s_i + d - 1)$, where d is the duration of the service and is varied on the x-axis. s_i is chosen uniformly at random from 12 slots. Users divide their values, chosen uniformly at random from $[0, 1)$, equally among all d time slots in their bids. The average extra value that the mechanism generates over the Regret Algorithm increases from 0.77 to 0.98. Indeed, as users spread their value across multiple time-slots, the mechanism becomes more likely to find a single time-slot with sufficient value to justify implementing the optimization. The difference, however, is small because the increased probability of overlap due to increased duration is offset by the fact that the values available are also spread over a larger

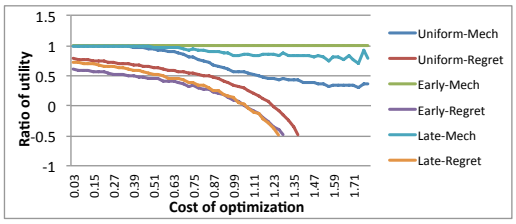


Figure 3: Setup identical to Figure 1(a) but with three arrival patterns: uniform, early, and late. "Mech" refers to the mechanism.

range of slots and hence, even if a user arriving at slot, say 1 and with duration of 2 slots, overlaps with a user arriving at slot 2, only half of her value is available at slot 2.

6.4 Arrival Skew

Figure 3 shows how the two strategies perform as one changes the way users arrive over time. The setup is identical to the additive case with group size of 6 (Figure 1(b)), except that we consider three cases with users arriving: (a) *uniformly* at random in one of 12 slots, (b) *early* following an exponential distribution with mean 1.2[7], (c) *late* following a distribution that is $12-t$ with t sampled exponentially with mean 1.2. Case (b) simulates datasets that become stale and hence less frequently used while (c) simulates datasets that become popular over time. The y-axis is the ratio of the utility of different settings to that of the utility of the mechanism with *early* arrivals.

The mechanism outperforms the Regret Algorithm substantially as user arrival becomes non-uniform (and the latter soon starts generating negative utilities). The mechanism improves with skew because skew increases the chance of a time slot with enough value to pay for all costs. Further, early arrivals can be 6.7× and 1.8× more efficient that uniform and late, respectively. This points to an interesting property of the mechanism-design-based approach: the approach performs much better as non-uniformity increases.

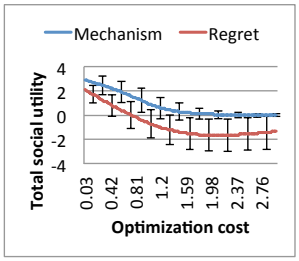
6.5 Selectivity of Substitutes

We now vary the selectivity of the substitutes defined as the ratio of the number of substitutable optimizations to the total number of optimizations. Figures 4(a) and 4(b) show the total utility for selectivities of 0.75 and 0.25, where each user chooses 3 optimizations uniformly at random from 4 and 12 optimizations, respectively.

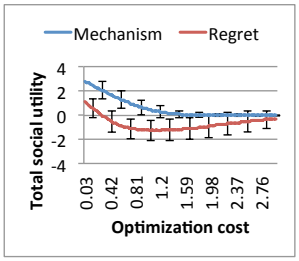
The figures show that, with more selective users, absolute utilities derived by both algorithms decrease. For *e.g.*, the Regret Algorithm goes from a utility of 2.28 to 1.22 for the optimization cost of 0.03 as selectivity increases. Indeed, with more selective users, the number of users per optimization decreases and more optimizations have to be be implemented to satisfy the users. In our simulations, the mechanism yields an average total utility of 1.0 even for optimizations that are 2.5× and 12.5× costlier than the optimizations at which the Regret Algorithm generates utilities of 1.0, for Figures 4(a) and 4(b), respectively.

Summary. In summary, our mechanism-based approaches not only guarantee truthfullness and cost-recovery but also yield utility that frequently exceeds that of the regret-based approach. Our approaches work especially well

[7]With mean 1.2, the maximum starting time slot of 6 users in 1000 runs was 12 as it is in case (a).



(a) Each user chooses 3 uniformly random optimizations out of 4.



(b) Each user chooses 3 uniformly random optimizations out of 12.

Figure 4: User values and optimization costs are identical to the setup in Figure 1(c).

in scenarios where many users derive significant value from an optimization during the same time-slot. They underperform compared to regret in scenarios where users value the same optimization but during non-overlapping periods.

7. RELATED WORK

Today, cloud providers use two strategies for pricing optimizations. In the first, the cost of the optimization is included in the base service price. For *e.g.*, Amazon SimpleDB [9] automatically indexes user data and includes the corresponding overhead in the base-price computation (45 bytes of extra storage are added to each item, attribute, and attribute-value). Similarly, SimpleDB and SQL Azure [22] automatically replicate data and include that cost in the base service cost. The key limitation with this approach is that the cloud must decide up-front what optimizations are worth offering and it forces users to pay for these optimizations. In other cases, users choose desired optimizations and pay their exact cost. For example, in Amazon RDS [6] a user can choose to launch and pay-for a desired number of read-replicas to speed-up her query workload. This approach, however, works well only in the absence of collaborations.

Significant recent work studies existing cloud pricing schemes, economic models, and their implications [21, 34, 39]. In contrast we develop a new pricing mechanism.

Most closely related to our work, Dash *et al.*, developed an approach for pricing data structures (indexes, materialized views, etc.) in a DBMS cloud cache [15]. In their approach, the cloud selects the structures to build based on the notion of regret. The cost is amortized to the first N queries that use the new structure. To compute regret, the cloud relies on budget functions provided by users, which indicate their willingness to pay for various quality of service. In follow-up work Kantere *et al.* [20] tuned their approach and developed a regression-based technique for predicting the extent of cost amortization. In contrast to our work, this previous approach relies on users being truthful and does not guarantee that the cost of an optimization will be recovered. As an example, consider a user who needs to run a single, very expensive query over a private dataset. If the user is truthful, no optimization will be implemented for this one query. Instead, the user thus submits a large number of inexpensive queries over the same dataset. The user expresses willingness to pay zero for processing these extra queries, yet indicates a preference for low execution times over low costs. The regret-based approach will let the user

manually pick slow and cheap service for these queries. It will then compute the maximum possible regret for the missing data structure that would have enabled faster plans for these queries. Once regret accumulates sufficiently, the user can run his single expensive query and pay a small fraction of the total cost of the optimization.

Significant research applies economic principles to resource allocation in distributed systems [2, 12, 13, 14, 16, 30, 32, 38], collaboration promotion in peer-to-peer systems [26, 25, 37], or more recently, VM allocation in the cloud [35]. We study how to choose and price optimizations rather than allocate processing resources. The Mariposa distributed database system [33] introduced a microeconomic paradigm for optimizing distributed query evaluation and data placement. This is a problem orthogonal to ours.

We build on the Shapley Value Mechanism, which is an instance of a Moulin Mechanism [24] that have been designed for various combinatorial cost-sharing problems where the cost of servicing a set of players is determined by solving a *offline* combinatorial optimization problem defined by the set [28]. We design Moulin mechanisms in an online setting.

Online mechanisms [27, Ch. 16] consider games where not all valuations are known simultaneously. While there is work on characterizing truthful mechanisms to maximize social utility in dynamic games [27, Theorem 16.17], to the best of our knowledge, there is no work that applies to cost-sharing in dynamic games.

8. CONCLUSIONS

We studied how a cloud data service provider should activate and price optimizations that benefit many users. We have shown how the problem can be modeled as an instance of cost-recovery mechanism design. We also showed how the Shapley Value mechanism solves the problem of pricing a single optimization in an offline game. We then developed a series of mechanisms that enable the pricing of either additive or substitutive optimizations in either an offline or an online game. We proved analytically that our mechanisms are truthful and cost-recovering. Through simulations, we demonstrated that our mechanisms also yield high utility compared with the state-of-the-art approach based on regret accumulation.

9. REFERENCES

[1] Cosmo benchmark. `http://nuage.cs.washington.edu/benchmark/astro-nbody/`.

[2] D. Abramson, R. Buuya, and J. Giddy. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems*, 18(8), Oct. 2002.

[3] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for vlsd databases. In *Proc. of the SIGMOD Conf.*, pages 179–192, 2009.

[4] Amazon Web Services (AWS). `http://aws.amazon.com`.

[5] Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`.

[6] Amazon Relational Database Service (RDS). `http://www.amazon.com/rds/`.

[7] Amazon S3: Requester Pays Buckets. `http://docs.amazonwebservices.com/AmazonS3/latest/dev/index.html?RequesterPaysBuckets.html`.

[8] Amazon Simple Storage Service (Amazon S3). `http://www.amazon.com/gp/browse.html?node=16427261`.

[9] Amazon SimpleDB. `http://www.amazon.com/simpledb/`.

[10] Windows Azure Platform. `http://www.microsoft.com/windowsazure/`.

[11] Windows Azure Storage Services REST API Ref. `http://msdn.microsoft.com/en-us/library/dd179355.aspx`.

[12] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proc. of the First NSDI Symp.*, Mar. 2004.

[13] Buuya et. al. Economic models for management of resources in peer-to-peer and grid computing. In *Proc of SPIE*, Aug. 2001.

[14] B. N. Chun. *Market-Based Cluster Resource Management*. PhD thesis, University of California at Berkeley, 2001.

[15] D. Dash, V. Kantere, and A. Ailamaki. An economic model for self-tuned cloud caching. In *Proc. of the 25th ICDE Conf.*, pages 1687–1693, 2009.

[16] D. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic models for allocating resources in computer systems. In S. H. Clearwater, editor, *Market based Control of Distributed Systems*. World Scientist, Jan. 1996.

[17] Gonzalez et al. Google fusion tables: data management, integration and collaboration in the cloud. In *Proc. of SOCC*, pages 175–180, 2010.

[18] Google App Engine. `http://code.google.com/appengine/`.

[19] Google App Engine Datastore. `http://code.google.com/appengine/docs/datastore/`.

[20] V. Kantere, D. Dash, G. Gratsias, and A. Ailamaki. Predicting cost amortization for query services. In *Proc. of the SIGMOD Conf.*, 2011.

[21] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Shopping for a cloud made easy. In *Proc. of HotCloud'10*, 2010.

[22] Microsoft SQL Azure. `http://www.microsoft.com/windowsazure/sqlazure/`.

[23] Microsoft SQL Azure Prices. `"http://www.microsoft.com/windowsazure/offers/popup/popup.a\spx?lang=en&locale=en-us&offer=MS-AZR-0003P"`.

[24] H. Moulin and S. Shenker. Strategyproof sharing of submodular costs:budget balance versus efficiency. *Economic Theory*, 18(3):511–533, 2001.

[25] C. Ng, D. C. Parkes, and M. Seltzer. Strategyproof computing: Systems infrastructures for self-interested parties. In *Proc. of P2PECON Workshop*, June 2003.

[26] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc of IPTPS Workshop*, Feb. 2003.

[27] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007.

[28] M. Pal and E. Tardos. Group strategyproof mechanisms via primal-dual algorithms. In *In Proceedings of the 44th Annual Symposium on Foundations of Computer Science (FOCS*, pages 584–593, 2003.

[29] D. C. Parkes. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, May 2001.

[30] J.-A. Quiané-Ruiz, P. Lamarre, S. Cazalens, and P. Valduriez. Managing virtual money for satisfaction and scale up in p2p systems. In *Proc. of DaMaP Workshop*, pages 67–74, 2008.

[31] Salesforce. `http://www.salesforce.com/`.

[32] T. Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *Proc. of the 12th International Workshop on Distributed Artificial Intelligence*, pages 295–308, 1993.

[33] Stonebraker et al. Mariposa: a wide-area distributed database system. *VLDB Journal*, 5(1):048–063, 1996.

[34] P. B. Teregowda, B. Urgaonkar, and C. L. Giles. Implications of moving to the cloud: A digital libraries perspective. In *Proc. of HotCloud'10*, 2010.

[35] K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis. Flexible use of cloud resources through profit maximization and price discrimination. In *Proc. of the 27th ICDE Conf.*, 2011.

[36] Upadhyaya et. al. How to price shared optimizations in the cloud. Technical report, UW, 2011.

[37] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing. In *Proc. of P2PECON Workshop*, June 2003.

[38] Waldspurger et al. Spawn: A distributed computational economy. *IEEE Trans. on Software Engineering*, SE-18(2):103–117, Feb. 1992.

[39] Wang et al. Distributed systems meet economics: Pricing in the cloud. In *Proc. of HotCloud'10*, 2010.