Predicting Cost Amortization for Query Services

Verena Kantere*
Cyprus University of Technology
A. Kyprianos 31, P.O. Box 50329, 3603 Lemesos, Cyprus
verena.kantere@cut.ac.cy

Debabrata Dash *
ArcSight, an HP Company
Avenue de Florissant 22
Renens, 1020, Switlzerland
debabrata.dash@hp.com

Georgios Gratsias *
ELCA Informatique SA
Av. de la Harpe 22-24 C.P. 519, 1001 Lausanne, Switzerland
georgios.gratsias@elca.ch

Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
anastasia.ailamaki@epfl.ch

ABSTRACT

Emerging providers of online services offer access to data collections. Such data service providers need to build data structures, e.g. materialized views and indexes, in order to offer better performance for user query execution. The cost of such structures is charged to the user as part of the overall query service cost. In order to ensure the economic viability of the provider, the building and maintenance cost of new structures has to be amortized to a set of prospective query services that will use them. This work proposes a novel stochastic model that predicts the extent of cost amortization in time and number of services. The model is completed with a novel method that regresses query traffic statistics and provides input to the prediction model. In order to demonstrate the effectiveness of the prediction model, we study its application on an extension of an existing economy model for the management of a cloud DBMS. A thorough experimental study shows that the prediction model ensures the economic viability of the cloud DBMS while enabling the offer of fast and cheap query services.

Categories and Subject Descriptors

H.2 [**Database Management**]: Database Applications

General Terms

Theory, Performance, Experimentation

1. INTRODUCTION

In the new world of computing it is more and more common to share information through the web. Producers and consumers of data scattered all over the globe meet through online services. Such services provide users with remote access to data sources and may offer possibilities for expedition of query execution. This is achieved through building data structures, such as materialized views and indexes. The provider trades query performance for the building and maintenance cost of such structures. The workload evolution prohibits the uncontrolled building and maintenance of data structures, because the cost outweighs the benefit on query performance.

As a business, the data provider aims at assuring its economic viability, without overcharging the user. The goal of the provider is twofold. First, keep the price of individual services low, in order to remain competitive. Therefore, it is very important to amortize the cost of data structures, such as views and indexes, to all prospective user queries that will benefit from its usage. Second, limit the risk of investment loss due to building and maintaining structures that are not reused. The extent of cost amortization both in time and number of user queries has to be decided beforehand, at the building time of a data structure. Hence, it is necessary to develop a technique that predicts the utilization of a new structure in the future and decides the appropriate extent of cost amortization.

In this work we predict the value of new data structures to the execution of future queries using a novel generic stochastic model. Given the building cost of a structure, the model outputs the extent in time and number of prospective queries for cost amortization. In case the structure has to be evicted from the server for some reason, (data update or storage cost exceeding the building cost), the provider is considered to suffer the loss of the remaining non-amortized cost.

The model predictions are based on the observation of past workload. We present a novel method that regresses the observed workload and provides input for the prediction model. The goal of the regression is to transform existing statistics on query traffic into estimations for the lifetime of potential data structures from the time they are built to the time they are evicted.

We demonstrate the effectiveness of the prediction model on the economy management of a cloud DBMS. An existing economic model for a cloud DBMS [6] is extended so that it takes into account user preferences and budget requirements for query execution in order to decide which data structures to build. The building and maintenance cost of the new structures is amortized to future queries that use them based on the presented prediction model. Without amortization, the cost of a new data structure has to be charged on a single potential query service, which leads to a high cost of this service. High cost makes the potential service non-competitive with existing ones in the cloud, and, therefore, not worth of investing. Hence, the cloud cannot improve the query services, unless it over-charges

*This research was conducted while the first, second and third author were working at the École Polytechnique Fédérale de Lausanne.

SIGMOD'11, June 12–16, 2011, Athens, Greece.

the user. Yet, over-charging a potential service usually exceeds the user's budget. The employment of the prediction model balances offer and demand: hot structures useful to many users are offered at a low cost, whereas specialized structures useful to few users are offered at higher cost, such that the cloud economy is viable.

A thorough experimental study on the cloud DBMS shows the effectiveness of cost amortization using the prediction model for both cheap and expensive data structures. Furthermore, cost amortization benefits simple and, especially, complex queries, the response time of which is decreased due to the earlier building of expensive data structures. Cost amortization leads to offering fast query plans at low prices. Overall, cost amortization achieves the economic viability, and, moreover, the profitability of the cloud DBMS, while gradually improving query services with respect to both response time and cost. The results show that the model achieves effective cost amortization in presence of data updates as well as in absence of data access locality.

In the rest of this paper Section 2 summarizes the related work. Section 3 describes the prediction model for cost amortization and Section 4 the data regression method. Section 5 summarizes the cloud DBMS description, presents the extended cloud economy and discusses the overhead for query execution cost. Section 6 presents the experimental study and Section 7 concludes the paper.

2. RELATED WORK

A big share of research on workload prediction is related to cost assessment of query execution and online index selection. The work in [19] proposes techniques that estimate the correlation of pairs of attributes. The correlation statistics are used in order to improve selectivity estimation for query optimization. The more recent work in [21] proposes algorithms for the discovery of soft multi-attribute functional dependencies that can be employed in query execution and a model that estimates query execution cost. These works focus on the mining of data value correlations and are related to the prediction of the utility of primary and secondary indexes in the presence of query predicates. Similarly, the works in [23] and [10] focus on the estimation of query execution cost using indexes on high-dimensional data and [10] proposes a technique for online index recommendations based on data statistics and query patterns. All the above works contribute in the prediction of query execution cost based on the availability of indexes. Starting from there: our model takes the indexes as input and can output a payment schedule for the amortization of the building and maintenance cost of these indexes.

The recent work in [36] deals with the problem of guaranteeing a predictable performance for any workload and proposes a solution based on a novel implementation of the relational table. This work is orthogonal to ours, since it focuses on implementation decisions that support similar performance irrespective to workload behavior.

There has been notable work on workload prediction in grid and super-computer environments [26, 8, 30, 24]. These works deal with the problem of workload modeling for the production of synthetic workloads employed for the evaluation of real systems. They focus on the site execution locality [8] or the temporal locality [26] of workload characteristics such as runtime, memory and CPU usage etc [24]. These works are orthogonal to ours, which focuses on workload characteristics of the data level. Specific works on modeling job arrivals [29, 25] are complementary to ours, since they can provide the workload distribution for special grid environments.

Structure usage prediction has been studied in other areas, such as the processor cache-line access and survivability prediction [3]. One proposal is a trace-based mechanism that predicts when a cache-line has been last accessed [22]. Another is a time-based mechanism that predicts the death of a block after a specific timeout period [18]. A third one is a counting-based predictor [20] that predicts the line to be dead after a fixed number of accesses. Our approach is closest to the last one; yet, we propose a sophisticated prediction model to utilize the computing and storage power available to full processors compared to the limited power on a cache.

Database as a service has been proposed by many researchers, starting with NetDB2 [11]. The authors identify security, remote access, and user interface as the main challenges. Recently, it has been argued that scalability and privacy still remain the unsolved challenges [1]. While these issues have been addressed in the literature, pricing the service has been generally ignored. In industry, salesforce.com was the first company to successfully commercialize the database as service paradigm [16]. Recently, Amazon's EC2 service [12], Google's AppEngine [13], and Microsoft's Azure service [15] have expanded the commercial application of the paradigm.

3. PREDICTION MODEL

A query service provider offers the possibility to users to query data collections. A query service is the execution of a user query employing a specific query plan. The provider builds data structures, such as indexes and materialized views, at a certain cost, in order to improve query performance. As soon as a structure is built, it is maintained, incurring a maintenance cost. The maintenance cost is the cost of storing the structure in the provider; building cost is expected to be orders of magnitude bigger than maintenance cost[1].

The user pays for each query service. The query service cost is $c_1 + c_2$, where c_1 is the cost of query execution and c_2 is some part of the building cost of data structures employed in this query service. The building cost $Build_S(.)$ of a new structure S has to be amortized to future user query services that employ this structure. In order to choose the value for the smaller number n of query services to which the cost of a structure S should be amortized, we have to estimate the number of prospective queries n' that will benefit from S. The overall gain of the provider is: $n' \cdot (Build_S(S)/n)$. Ideally, we would like $n' > n$. Two opposite factors should be considered to estimate n correctly: (i) As n increases, the initial cost of S is disseminated to more query services, making their overall (individual) cost smaller (ii) As n increases, the risk of a false estimation of n' increases, too; A false estimation $n' < n$ will damage the economy of the provider. Therefore the goal is to predict the value of n so that it is smaller but close to n'.

We present a generic stochastic model for the amortization of the building cost of new data structures. The model estimates n based on provider statistics on query traffic, data update and execution of query plans. This approach focuses on constant amortization CA of the building cost, i.e.:

$$CA(n, Build_S(S)) = Build_S(S)/n \quad (1)$$

Selecting n is a challenging problem, as it depends on the provider's risk aversion as well as the query arrival pattern. First, the model estimates the maximal time extent t_e until which amortization should be completed. Second, the model estimates the number of prospective queries n until the time point t_e that will use the structure.

3.1 Cost Amortization in Time

The model proposes the extent of cost amortization in time, t_e, based on an estimation of the lifetime of the new structure S in the data provider. The lifetime of S ends if one of the following occurs: (i) S is not used in the prospective queries, (ii) the data related to S

[1]Based on the pricing scheme of Amazon EC2 [12] for I/O operations, disk space, CPU time and network bandwidth, the cost of building an index or caching a table column is on average 7 orders of magnitude bigger than the cost of maintaining it thereafter.

is updated. We define respectively the notions of (i) *usefulness* of S and (ii) *stability* of S, in time, which give the probability of $failure$ of S in time. If S is not useful or stable at some time point, it fails, and therefore its lifetime ends. As long as S has not failed yet, it is considered a *reliable* structure, and cost amortization can still be performed. As time moves on from the building point, the reliability of S decreases and the probability of failure increases. The model proposes as the extent of cost amortization in time, t_e, the crossing point after which S is more probable to fail than to remain reliable.

DEFINITION 1. *The usefulness of a structure S, $d_S = d_S(t)$, is the probability that the structure S is useful to the provider cache, i.e. prospective selected query plans will employ S.*

The usefulness of S is the conditional probability of the observed fact that S is useful to a selected query plan, on the grounds that a query has arrived.

DEFINITION 2. *The probability that the data related to a structure S is stable, i.e. not updated, is $st_S = st_S(t)$.*

DEFINITION 3. *The failure $f_S = f_S(t)$ of a structure S is the probability distribution of the occurrences that S is not employed in selected query plans. The cumulative probability of failure of S is denoted as F_S.*

Based on survival analysis [5], $F_S(t)$ represents the fact that the structure S will fail in the time period $[t_0, t]$, where t_0 is the time that S is built in the cache. For simplicity we set $t_0 = 0$.

PROPOSITION 1. *The cumulative failure probability of a structure S, (at a time point t), is the the probability that S is not useful any more to selected query plans or it has become obsolete due to the support of data that has been updated, (at time t).*

JUSTIFICATION It is assumed that a query occurrence is not the result of another query occurrence, and thus a query occurrence does not depend on the results of other queries. Generally, the query workload is not dependent on the data, and therefore, on the update (or, equally, the stability) of data. Thus, the probability of data stability, $st_S(t)$, is independent from the probability of usefulness of S, $d_S(t)$. The joint probability $st_S(t) \cdot d_S(t)$ denotes the fact that S is not obsolete and still useful at time t. In order for this to be true, S must have not failed in the time period $[0, t]$ because of any of these two reasons. Therefore:

$$F_S(t) = 1 - st_S(t) \cdot d_S(t) \quad (2)$$

The failure f_S is easily computed from the cumulative failure F_S as: $f_S(t) = -st_S(t) \cdot \frac{d(d_s)}{dt} - d_S(t) \cdot \frac{d(st_s)}{dt}$. Observe that for the building time $t = 0$, for which the probability that the data is stable and the structure is useful is the highest, (i.e. $st_S(0) = d_S(0) = 1$), the failure depends directly on the rate of change of data stability and structure usefulness: $f_S(t) = -(\frac{d(d_s)}{dt} + \frac{d(st_s)}{dt})$.

PROPOSITION 2. *The extension in time of building cost amortization of a new structure S to n individual payments balances the opposite tends of (a) minimizing the individual payment for the usage of S in a query plan and (b) minimizing the risk of failure of S, i.e. the risk that some of the building cost of S is not amortized.*

JUSTIFICATION The data that the incoming queries request change with time. Data changes cause the necessity to update existing structures (e.g. indexes) that involve the changed data, or even drop, and, possibly, reconstruct them from scratch (e.g.materialized views). We call the incidents of need to change existing structures (update, or dropping and reconstruction) 'failures'. Failures of a structure S incur additional building cost. Frequent failures of S may increase the accumulated building cost of S substantially, such that it is never completely amortized to prospective user queries. Therefore, amortizing $Build_S(S)$ to a small number of prospective queries reduces the risk that the completion of amortization fails. Yet, extending amortization to a big number of prospective queries, the selected plan of which will benefit from the employment of S, reduces the individual amortization payment, and therefore the overall cost of each query service.

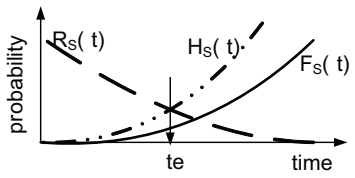

Figure 1: The maximal time of amortization is defined by the reliability and the hazard of failure.

DEFINITION 4. *The reliability $R_S(t)$ of a structure S is the quality of S that indicates that the non-amortized building cost of S at time t will be amortized in the future, (t, ∞).*

From survival analysis, the reliability of a structure is related to the failure of a structure as follows:

$$R_S(t) = 1 - F_S(t) \quad (3)$$

Intuitively, the reliability of a structure depends on the usefulness of it to prospective queries, as well as on the stability of data[2]. Indeed, the combination of equations 2 and 3 shows that: $R_S(t) = st_S(t) \cdot d_S(t)$. The rate of reliability change r_S at time t depends directly on the probability of failure at this time: $r_S(t) = -f_S(t)$.

PROPOSITION 3. *Since a structure S is reliable at building time $t = 0$, the cumulative probability that S will fail in a future time t is:*

$$H_S(t) = \int_0^t \frac{f_S(t)}{R_S(t)} dt \quad (4)$$

JUSTIFICATION The hazard rate, denoted by $h_S(t)$ represents the instantaneous conditional failure of a structure S and by definition it is calculated as: $h(t) = \frac{f(t)}{R(t)}$ [5]. Qualitatively, there is a fine difference between the hazard rate and failure. The hazard rate evaluates the probability that the system is operational on time t, on the condition that it has not failed in $[0, t)$, whereas failure evaluates the probability that the system fails at time t, no matter if it has failed before. The usage of structure S is a 'repairable' process, i.e. structure failures of usage in selected query plans may be followed by success of structure usage in prospective selected query plans. For such a system, the hazard rate provides a safer estimation of future failure, taking into account the degradation of the structure reliability with time. Therefore, since a structure S is reliable at building time $t = 0$, the cumulative probability that S will fail in a future time t is given by the cumulative probability of hazard, $H_S(t)$.

A trade-off between the hazard H_S and the reliability R_S of a structure S can indicate the maximal time limit t_e up to which the cost amortization should extend.

DEFINITION 5. *The balancing rate b_S at time t for a structure S is the difference of the reliability R_S and hazard H_S at this time:*

$$b_S(t) = R_S(t) - H_S(t) \quad (5)$$

[2]The compound reliability of a set of table columns (w.r.t. either caching or indexing) is calculated as $R(t) = R_1(t) \cdots R_k(j)$. Dealing with the problem of dependent reliabilities of j table columns is future work.

b_S indicates the time that amortization should terminate.

PROPOSITION 4. *The amortization of $Build_S$ for a new structure S should extend until the time point t_e, such that the cumulative balance between the respective reliability and hazard is maximized. Therefore, the cumulative balance between benefit and hazard is:*

$$B_S(t) = \int_0^t R_S(t) - H_S(t)dt \quad (6)$$

and the extention time for amortization of $Build_S(S)$ is:

$$B_S(t_i, k) < B_S(t_e, k), \forall t_i, 0 \leq t_i, t_e \quad (7)$$

JUSTIFICATION The balancing rate b_S represents the instant balance between the opposite tends of reliability and hazard. The integral of b_S in a time interval $[t_1, t_2]$ represents the cumulative balance of this pair of tends in this interval. Fixing this interval from the beginning of amortization, i.e. $t = t_0$ until this balance is maximized, $t = t_e$ guarantees the maximal cumulative probability that full amortization of the cost of S will be achieved.

Since both the reliability and hazard are monotonic and continuous [5], the maximization of their cumulative balance occurs at t_e, s.t. $R(t_e) = H(t_e)$, as shown in Figure 1.

3.2 Cost Amortization in Number of Queries

Given the maximal extent of cost amortization in time, t_e, we can estimate the (smaller) number of prospective queries on which the cost will be amortized, n. Taking into account the query arrival pattern and the reliability of S we define the notion of *usage* of S that denotes the number of queries that use S at a time point. Based on the usage of S in time we define the probability of $gain$ which gives an estimation of how the cost is amortized in time. We select n based on a fixed cumulative probability of gain related to the *influence* of S to the economy management of the data provider. As explained in Section 5, the influence of a structure S in the economy of a cloud DBMS represents the financial loss of the cloud before building S incurred by not being able to service queries using S.

For a general purpose prediction, we model the arrival of queries in the provider as a stochastic process $N = \{N_t : t \in T\}$, where T represents time, i.e. $T \equiv [0, \infty)$. The collection N of random variables N_t (or else $N(t)$) takes values from the state space $[1, 2, \ldots, \infty)$ and represents the number of queries that have arrived to the provider in the period $[0, t]$. We assume that the number of users of the provider is big and that queries for service arrive to the provider independently. Therefore, we assume that query arrival follows the Poisson distribution with parameter λ. Thus, the probability that at time t the number of arrived queries is k is given as: $P(N_t = k) = \frac{(\lambda \cdot t)^k}{k!} e^{-\lambda \cdot t}$. Based on the law of large numbers, the parameter can be calculated as $\lambda = N(t_o)/t_o$ for some t_o that can give a general picture about the service traffic of the provider. Thus, λ can be calculated based on traffic statistics (see Section 4).

PROPOSITION 5. *The number of incoming queries to the provider for a time period that statistics of query arrivals do not vary significantly constitute a homogeneous Poisson process.*

JUSTIFICATION Since no assumption is made for the provider and the users, structure usage and query arrivals are independent, and queries arrive on random time points. Since $N(t)$ follows a Poisson distribution the number of query arrivals in any time interval $[t_1, t_2]$, $t_1 < t_2$ is still a Poisson distribution (based on the respective property of the Poisson distribution). Moreover, since individual query arrivals are independent, sets of non-overlapping query arrivals are also independent, i.e. $N(t_2 - t_1)$ is probabilistically independent to $N(t_4 - t_3)$, for any $0 \leq t_1 < t_2 < t_3 < t_4$. Finally, the rate λ of query arrival for a given time period with non-fluctuating statistics $[0, t_o]$, has a constant value, i.e. $\lambda(t) = c, c \in \Re^+$ for $t \in [0, t_o]$. These three facts are by definition [34] necessary and sufficient to infer that the query arrival N in $T = [0, t_o]$ is a homogeneous Poisson process.

The prediction model can accomodate query arrival patterns other than the homogeneous Poisson process, as long as these are known; such input can be offered by recent works [29, 25] that propose query arrival modeling of non-independent query arrival patterns.

DEFINITION 6. *The usage of a structure S, u_S, is the probability of occurrences of the fact that S is employed in a selected query plan.*

PROPOSITION 6. *For a new structure S that is built in the provider, the usage of it in selected query plans, u_S, is a function of time t and query arrivals k, i.e. $u_S = u_S(t, k)$. This function is a correlation of the query arrival probability and the reliability of S.*

JUSTIFICATION It is straightforward that query arrival, $p_N(k, \lambda) = P(N_t = k)$ is independent from the usefulness of S, $d_S(t)$, and the data stability, $st_S(t)$. Therefore, it is safe to assume that R_S is *uncorrelated* to p_N, since there is no evidence that in general the usefulness of S, d_S or the stability of data related to S, st_S are linearly dependent on query arrival [3]. Thus, the usage of S is the joint distribution of the its reliability R_S and the distribution of query arrivals p_N. Therefore, the following form of u_S is adopted:

$$u_S(k, t) = R_S(t) \cdot p_N(k, \lambda) \quad (8)$$

where k is the number of query arrivals at each time point. The above form of $u_S(k, t)$ reflects the following intuitive requirements: (i) if the reliability of S is not degraded with time, i.e. $d_S(t) = st_S(t) = 1$, then S is used in all selected query plans, and therefore $u_S(k, t) = p_N$; (ii) if S is not reliable (for example, S is obsolete: $st_S(t) = 0$ or not useful: $d_S(t) = 0$), then S is not used in any selected query plans, regardless of the query arrival, i.e. $u_S(k, t) = 0$.

Let us assume that G_t is a random variable that represents the amortization of $Build_S(S)$. Since G depends deterministically on the variable that represents the usage of structure S, S_t, we create the following gain probability distribution, $g(t, k)$ w.r.t. time and query arrival.

$$g_S(k, t) = P(G_t = k\frac{Build_S(S)}{n}) = \frac{u_S(t, k)}{\int_0^\infty R_S(t)dt}, k \leq n. \quad (9)$$

The cumulative gain probability is denoted as G_S. The amortization of $Build_S(S)$ to a number of selected query plans that use S, n, is based on G_S. We assume that the influence of the cost of a new structure S, denoted as $Inf(S)$, is a relative measure that indicates the importance of S in the economy of the data service provider w.r.t. the importance of the rest of the structures:

DEFINITION 7. *Assume a vector $w_S[.]$, where each element $w_S[S]$ corresponds $1 - 1$ to the S element of the set of structures offered by the provider and represents a quantification of the importance of S to the economy of the provider. The influence of the cost of a new structure S to the economy is the importance of S, $w_S[S]$ normalized by the importance of all structures in w_S:*

$$Inf(S) = \frac{w_S[S]}{\sum_{i=0}^{|w_S|} w_S[S]} \quad (10)$$

[3]Of course, R_S may be dependent with some other way on p_N, for example R_S may be dependent on p_N^2

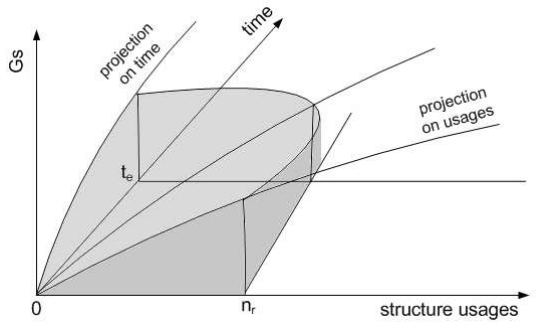

Figure 2: The shaded space is selected to be equal $Inf(S)$.

PROPOSITION 7. *The number n of amortization payments for $Build_S(S)$ is selected such that the gain probability of at most n payments of $Build_S(S)$ in the interval $[0, t_e]$ is equal to the influence of $Build_S(S)$ to the provider economy, $Inf(S)$:*

$$G_S(k \leq n_r, t \leq t_e) = Inf(S) \tag{11}$$

Where n_r is the average rate of usage occurrences of S. The number of payments is calculated as $n = n_r \cdot \int_0^{t_e} R_S(t)dt$.

JUSTIFICATION The probability that at most n_r queries per time unit for an overall time $[0, t_e]$ arrive and $Build_S(S)$ is amortized to them is given by: $G_S(k \leq n_r, t \leq t_e) = \int_0^{n_r} \int_0^{t_e} g_S(k,t)dtdk = \frac{\int_0^{t_e} R_S(t)dt}{\int_0^{\infty} R_S(t)dt} \cdot \int_0^{n_r} p_N dk = P(k \leq n_r) \cdot \frac{\int_0^{t_e} R_S(t)dt}{\int_0^{\infty} R_S(t)dt}$. Therefore $G_S(n_r < k, t \leq t_e) = 1 - G_S(t_e < t) - G_S(k \leq n_r, t \leq t_e) = 1 - G_S(t_e < t) - P(k \leq n_r) \cdot \frac{\int_0^{t_e} R_S(t)dt}{\int_0^{\infty} R_S(t)dt}$. We choose the value of $G_S(k \leq n_r, t \leq t_e)$ to be equal to the influence of S to the provider economy, $Inf(S)$. Intuitively, if S has been influential to the provider economy, i.e. possibly a lot of queries would have benefited from it in the past, it is most likely that this will be the case in the future, too. The opposite holds, too. Thus, the bigger $Inf(S)$ is, the bigger $G_S(k \leq n_r, t \leq t_e)$ is, the bigger n_r and n is, and, of course, the more risk we take in amortization, since, $G_S(n_r < k, t \leq t_e)$ becomes smaller. Figure 2 depicts the the probability $G_S(k \leq n_r, t \leq t_e)$. The constraint on $G_S(k \leq n_r, t \leq t_e)$ is actually a constraint on $P(k \leq n_r)$, which gives the value of n_r. The latter is the selected expected value of query arrival rate. Based on equation 8, we estimate the total number of payments n by weighting n_r with the reliability of the structure and integrating for the whole time interval $[0, t_e]$, i.e. $n = n_r \cdot \int_0^{t_e} R_S(t)dt$. This n is conservative w.r.t. the limit, $Inf(S)$, providing full amortization, (even independently of the importance condition) with success probability $> 1 - Inf(S)$.

4. DATA REGRESSION METHOD

To fit the real workload and selected query plans of the provider to the prediction model, we follow a regression procedure based on an observation period T_{obs} of provider service. The observation data consists of the: (i) query arrivals, (ii) structures employed in selected query plans, (iii) usage occurrences of each structure in the plans, and (iv) the update of data that are related to already built structures.

The time series of query arrivals T_{obs}, $\mathcal{Q} = \{Q_t, t \in T_{obs}\}$ is fit into a Poisson distribution, $p_N(k, \lambda)$, (i.e. the λ parameter is estimated) using the Poisson regression method [7]. Each element Q_t represents the number of queries that arrived at time t. Experimentally, T_{obs} is an ordered set of equal time slots t, and, therefore, Q_t is the number of queries that occurred in time slot t.

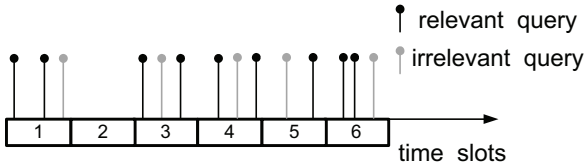

Figure 3: The maximal time of amortization is defined by the reliability and the hazard of failure.

Beyond fitting the query arrival, it is necessary to regress the observation data in order to fit the probability of usefulness, $d_S(t)$, and the probability of data stability, $st_S(t)$, for each newly built structure S. In the following, we describe how the observation data is processed in order to fit the probability of usefulness, $d_S(t)$.

The time series of usefulness w.r.t. a built structure S is $\mathcal{D}_S = \{D_{S_t}, t \in T'_{obs}\}$, where $D_{S_t} \leq Q_t$ and $T'_{obs} \subseteq T_{obs}$, s.t. $t \in T_{obs} \wedge t \in T'_{obs}$ iff $Q_t \neq 0$. Informally, the series $\mathcal{D}_S$ includes only the times (experimentally, time slots) for which there are query arrivals; for each time t, D_{S_t} is the number of arrived queries that employed S in their selected plan.

We process the series $\mathcal{D}_S$ in order to extract a series of sets of consecutive usages of S in selected query plans: Let us assume that the time series of query arrivals that corresponds to the filtered set of times T'_{obs}, is denoted as $\mathcal{Q}'$; it holds that $\mathcal{Q}' \subseteq \mathcal{Q}$. Two or more consecutive events in $\mathcal{D}_S$ are merged according to the *grace period* parameter v, defined as follows:

DEFINITION 8. *The grace period parameter, denoted as v is the sensitivity of the regression process to the observed failures of structure usage. The regression procedure is by v sensitive to $\mathcal{D}_S$ iff two or more consecutive events $D_{S_i}, \cdots, D_{S_{i+m}}$, $i = 1, \cdots, |T'_{obs} - m|$, $m \geq 1$ can be merged in one, s.t. $\forall j, k, i \leq j < k \leq i + m \wedge \forall p \in [j, k]$ it holds that $Q'_p - D_{S_p} \neq 0$ and $\sum_{p=j}^{p=k} Q'_j - D_{S_j} \leq v$.*

The value of v is equal to the maximal number of consecutively occurring selected query plans that did not use the structure S and that are ignored by the regression process. The merging procedure w.r.t. v results into:

$$\mathcal{L}_S = \{L_{S_t} \in P_S\}, |P_S| \leq T'_{obs} \tag{12}$$

$\mathcal{L}_S$ is a new time series and can be employed to perform survival analysis [5] in order to find the overall lifetime probability distribution of the usefulness of a structure S. Specifically, each time $t \in P_S$ is a member of the population P_S, with size $|P_S|$. Each member $t \in P_S$ has a lifetime equal to L_{S_t}. Intuitively, P_S is the set of occurrences of extended usage of the structure S in selected query plans in the observed time period T_{obs}. The lifetime of each such occurrence is the usage of S in terms of query arrivals.

EXAMPLE 1. *Figure 3 shows an example where the observation time is divided into 6 slots and queries relevant or irrelevant to the structure S are shown in black or gray, respectively. The regression ignores slot 2 since there are no queries. If the grace period is $v = 1$, all the gray queries are ignored and S is considered to have survived through all five slots. If $v = 0$, the lifetime of S is considered to end when a gray query occurs and begins again on the arrival of a black one. This results in 5 short lifetime instances of S.*

The regression process fits the lifetimes $\mathcal{L}_S$ of the population P_S to a Weibull distribution [7]. Weibull is employed as the failure [5] w.r.t. the usefulness of the structure S, $d_S(t)$, for $t \in [0, \infty)$, where $t = 0$ is the building time of S.

The regression process calculates the distribution of data stability w.r.t. a structure S, st_S, in a similar way. In this case, the regression is performed on the whole time interval T_{obs} and not on T'_{obs}. At each time $t \in T_{obs}$ the occurrence of unchanged data is modeled as a *success* and the occurrence of data update is modeled as a *failure*. A time series $\mathcal{ST}_S = \{ST_{S_t}, t \in T_{obs}\}$ is created, s.t. ST_{S_t} is the sum of successes decreased by the sum of failures in time t. $\mathcal{ST}$ is regressed to a Weibull distribution using the methodology described for the regression of the time series $\mathcal{D}_S$.

5. A CLOUD DBMS

We demonstrate the effectiveness of cost amortization using the proposed prediction model through its application to the economy management of a cloud that provides data services. Such a cloud DBMS is described in [6].

Briefly, data are permanently stored in a cloud of databases and data structures, i.e. indexes, materialized views and cached columns, as well as multiple processors in the cloud DBMS expedite query execution. Users queries are charged in order to be served. Queries are either executed in the cloud DBMS or on the cloud databases. The goal of the cloud is to hold an economy self-tuned along the lines of the following policies: (i) individual user satisfaction with the query services they receive w.r.t. the money they are charged, (ii) increasing overall quality of query services, and (iii) cloud profitability at all times. Query performance is measured in terms of execution time. The faster the execution, the more expensive the service. Quality increment of services is achieved with money investment on new data structures. The user defines her preferences concerning the service of her query by indicating the budget she is willing to spend on a specific query, according to the respective execution time that the cloud can provide. Also, the user can provide a preference for faster or cheaper query services. The cloud receives the query and the user preferences and produces respective alternative query plans. The price of these query plans is estimated and juxtaposed to the preferences of the user. If there are query plans that the user can afford, according to her budget, then the cloud chooses the most appropriate one of them, w.r.t. the policies.

The cloud's intention is to receive repay for these investments. However, without cost amortization the cloud does not actually arrive at the point of making new investments: If the cost of a new structure is repaid by a single or a few prospective users that use it, then the cost of the respective services can be so high that it most probably exceeds the users' budgets. In this case, the new structure will not be used in prospective query executions and, therefore, the cloud will never decide to build it. Overall, without cost amortization the quality of the services remains static, since the cloud will never decide to invest on new data structures. Employing the prediction model, however, the cost of new structures can be amortized to numerous prospective queries that will be executed using them. Cost amortization reduces the individual user payments to an amount that is covered by the users' budgets. This leads the cloud to improve the services by investing on new structures. We choose to guide cost amortization of a new structure S based on the *regret* of the cloud for not having already invested in S. Therefore the importance of a structure S, $w_S[S]$ (see definition 7) is identified as the respective regret $regret_S[S]$, which is calculated as described in Section 5.1.

Section 5.1 summarizes the operation of a cloud DBMS that provides data services and Section 5.2 extends the economy presented in [6]. The extended economy takes into account user preferences as well as budget requirements for query execution and assigns normalized values to the regret of possible new structures. Section 5.3 discusses the overhead of cost estimation for query execution plans.

5.1 Cloud Operation

The cloud has an account where the user payments for the services they receive are deposited. Also, money from this account are used for new inventory, i.e. new cache structures that can be used to execute queries in the cache, faster and cheaper. The overall credit in this account is denoted as CR. It is not the intension of the cloud to make profit from its investments. Some profit, however, is made in cases that the user is willing to pay more for the provided services than they really cost, without taking advantage of the difference of offered compensation and real cost for services. Moreover, the cloud aims at total amortization of investment cost.

The investment on a new structure is decided based on the notion of *regret*, inspired by [35]. Essentially, the cloud is not able to offer services that employ structures that are not yet built. In case that a structure is available, it is possible that the execution of some queries could benefit by employing it, either in terms of time or cost. The regret for not being able to offer a service because a structure is not built is accumulated and monitored; if the regret for the absence of a structure becomes substantial, then the cloud decides to invest in the construction of this structure.

DEFINITION 9. *The regret for a structure S, which is possible new inventory of the cloud, is a quantification of the accumulated value of the missed chances to provide better quality query services that comprise S.*

The regret is the incentive for the improvement of the query services. The cost of new inventory is paid from the cloud's account. The cost is amortized to prospective users that receive services which include the new inventory. The aim of amortization is to reduce the individual cost of the services. Cost reduction increases the potential that the user's budget covers the price of the offered services. In such a case, the cloud benefits from the difference of actual cost and offered user compensation; therefore, the cloud increases its profit, and thus, CR, which gives the opportunity for more investments directed by regret, leading to even more quality services.

The cloud maintains a pool of structures relevant to the queries in the recent past. Cache structures considered by the cloud are CPU nodes, table columns and indexes. In the current infrastructure the columns of the tables in the back-end databases are cached, in order to facilitate a comparison with [27]. Moreover, indexes constructed in the cache, accelerate the queries. Future work includes the expansion of the infrastructure with caching of materialized views, similar to [37], or partial columns, similar to [33]. The columns are determined by the definition of the queries, and the indexes are subsets of *per-query-optimal* candidate index sets [4]. These indexes typically contain many attributes relevant to a query and indexes which provide useful ordering columns to benefit join and aggregation processing. Since disk is the least expensive, the cloud tends to prefer such large candidate indexes. Additional CPUs are employed to speed up the parallelizable queries. Summarizing, the cache decides on building and maintaining three different types of structures: 1) CPU nodes N, 2) table columns T, and 3) indexes I.

These structures are garbage-collected using LRU policy, so that the structure cache can be searched and processed efficiently for each incoming query plan. Upon receiving an incoming query Q, the cloud considers a set of plans, $\mathcal{P}_Q$. This set consists of two non-overlapping subsets: the set of plans that include only existing cache structures, $\mathcal{P}_{Q_{exist}}$, and the set of plans that include also possible new cache structures, $\mathcal{P}_{Q_{pos}}$. The cloud determines the most optimal execution plan in $\mathcal{P}_{Q_{exist}}$[4] and considers the plans in $\mathcal{P}_{Q_{pos}}$ for possible investment in new cache structures.

[4]The optimal plan can be determined by querying the optimizer or by looking up a set of plans cached from earlier queries.

The cloud produces a set of alternative query plans for the input query Q, $\mathcal{P}_Q$, which holds only the two skylines of query plans (w.r.t. execution time and overall cost) in $\mathcal{P}_{Q_{pos}}$ and $\mathcal{P}_{Q_{exist}}$. From the implementation point of view, the cloud finds the structures it can use to execute an incoming query. First, it requires the column information for all the tables to be included in the query. Second, it determines the set of indexes that are possibly beneficial to the query by analyzing the query structure. Many of the queries can be parallelized using a bushy plan and most of the database vendors support parallelizing such queries. The queries can also be parallalized in a domain-specific way, as shown in [31].

EXAMPLE 2. *Consider the query select* `Q = max(A) from T where B = 0.1`. *The cloud needs columns "A", and "B" to execute the query. The latter can be sped up by using indexes on (A), (B), or (B,A). The query can be parallelized easily using multiple CPUs to scan the index and find the maximum value of A.*

The cost of each query plan $P_Q \in \mathcal{P}_Q$ is estimated according to the cost model described in [6], and produces a discrete budget function $B_{P_Q}(t) : t_{\mathcal{P}_Q} \mapsto \Re^+$, where $t_{\mathcal{P}_Q}$ is the set of execution time values for all plans in $\mathcal{P}_Q$. $B_{P_Q}(t)$ is typically a monotonically decreasing function, i.e. $B_{P_Q}(t_1) \geq B_{P_Q}(t_2), \forall t_1, t_2 \in t_{\mathcal{P}_Q}\ s.t.\ t_1 < t_2$.

The cloud maintains the array $regret_{\mathcal{S}}$, which stores the accumulated regret values for each physical structure or set of cached data $S \in \mathcal{S}$ that is employed by any plan $P_Q \in \mathcal{P}_Q$, for any input user query Q. Specifically, the regret for a non-chosen query plan P_Q is added to the positions in $regret_{\mathcal{S}}$ that correspond to the $S \in \mathcal{S}$ that are employed by P_Q. The accumulated regret value for each S shows the overall regret of the cloud for not employing it in executed query plans. A high regret value indicates that S could have been employed in either numerous or expensive, or both, query plans. When the regret for S, $regret_{\mathcal{S}}[S]$ reaches a high value, then the cloud makes an investment and constructs S. Assuming that the overall credit in the cloud account is CR, then the $regret_{\mathcal{S}}[S]$ must rise to a fraction a of CR, in order for S to be considered for imminent investment:

$$InvestIn(S) = round(\frac{regret_{\mathcal{S}}[S]}{a \cdot CR}) \tag{13}$$

5.2 Cloud Economy

In the following we extend the economy model presented in [6]. The user input to the cloud is a query Q and the elements:

- a preference between short execution time or small cost
- a budget function

It is straightforward that the user always wants her query to be executed in minimum time with minimum cost. By default these two are conflicting directions for query plan selection; therefore, the user is given the choice to express a preference between the two. Moreover, the user is given the choice to input a budget function $B_Q(t)$ that indicates the price she is willing to pay according to the execution time of the query. There are no limitations for the structure of B_Q, while the user is expected to input a function that is descending with time. Specifically, the user inputs a set of possibly non-null elements in the following form:

$$UI = (preference, B_Q(t)), preference = \text{“}time\text{”}, \text{“}cost\text{”} \tag{14}$$

The form of UI enables a flexible user input for query execution directions. For example, the user is allowed to not give any preference or budget function[5], just input a query execution time limit with a preference on faster plans, Figure 4(a), or cheaper plans Figure 4(b), or define a specific budget function without any preferenece on time or cost, Figure 4(c).

A query plan that conforms to the user's budget, but also supports the three policies defined earlier is chosen, depending on B_{P_Q} and UI as follows: The cloud budget, B_{P_Q}, is compared with the user's budget, B_Q. The part of B_{P_Q} that is *covered* by B_Q, is stored in $B^a_{P_Q}$, i.e. $P_Q \in B^a_{P_Q}$ iff $P_Q \in B_{P_Q} \wedge B_{P_Q}(t_{P_Q}) \leq B_Q(t_{P_Q})$. The cloud selects from $B^a_{P_Q}$ an existing query plan to be executed, $P_{Q_{sel}} \in \mathcal{P}_{Q_{exist}}$ and a subset of the possible query plans, $\mathcal{P}_{Q_{regret}} \subseteq \mathcal{P}_{Q_{pos}}$ for which the regret is calculated. $P_{Q_{sel}}$ and $\mathcal{P}_{Q_{regret}}$ are selected depending on combinations of the following:

- the set $B^a_{P_Q}$
- the value of $UI.preference$

The possible combinations are:

1: The user budget does not cover the cloud budget, i.e. $B^a_{P_Q} = \emptyset$[6].

The user is presented with all the query plans in $\mathcal{P}_{Q_{exist}}$ accompanied with their cost and the execution time. The user can pick and pay for one plan $P_{Q_{sel}} \in \mathcal{P}_{Q_{exist}}$ or none. The cloud selects the subset of the possible query plans, $\mathcal{P}_{Q_{regret}} \subseteq \mathcal{P}_{Q_{pos}}$, for which the regret is calculated, depending on the preference value:
1a: If $UI.preference = null$ the regret is not calculated for any possible query plans, i.e. $\mathcal{P}_{Q_{regret}} = \emptyset$[7].
1b: If $UI.preference =$ "*cost*", the regret is calculated for all possible plans that are cheaper, i.e. $P_Q \in \mathcal{P}_{Q_{regret}}$ iff $P_Q \in \mathcal{P}_{Q_{pos}} \wedge B^a_{P_Q}(t_{P_Q}) \leq B^a_{P_Q}(t_{P_{Q_{sel}}})$.
1c: If $UI.preference =$ "*time*", the regret is calculated for all possible plans that are faster, i.e. $P_Q \in \mathcal{P}_{Q_{regret}}$ iff $P_Q \in \mathcal{P}_{Q_{pos}} \wedge t_{P_Q} \leq t_{P_{Q_{sel}}}$.

2: The user budget covers the cloud budget, i.e. $B^a_{P_Q} \neq \emptyset$.

A query plan is selected based on the preference value:
2a: If $UI.preference = null$, the cloud chooses the plan $P_{Q_{sel}}$ that minimizes the gain from the user's payment diminished by the actual cost of the plan, i.e. the plan $P_{Q_{sel}}$ is picked, $s.t.\ B_Q(t_{P_{Q_{sel}}}) - B^a_{P_Q}(t_{P_{Q_{sel}}}) \leq B_Q(t_{P_{Q_j}}) - B^a_{P_Q}(t_{P_{Q_j}}), sel \neq j, P_{Q_j} \in \mathcal{P}_{Q_{exist}}$. The regret is calculated for all possible plans that are less profitable for the cloud, i.e. $P_Q \in \mathcal{P}_{Q_{regret}}$ iff $P_Q \in \mathcal{P}_{Q_{pos}} \wedge B_Q(t_{P_{Q_{sel}}}) - B^a_{P_Q}(t_{P_{Q_{sel}}}) \geq B_Q(t_{P_Q}) - B^a_{P_Q}(t_{P_Q})$
2b: If $UI.preference =$ "*cost*", the cloud chooses the query plan $P_{Q_{sel}}$ that is cheaper, i.e. $\forall P_Q \in \mathcal{P}_{Q_{exist}}, B^a_{P_Q}(t_{P_{Q_{sel}}}) \leq B^a_{P_Q}(t_{P_Q})$. The regret is calculated for all possible plans that are cheaper, i.e. $P_Q \in \mathcal{P}_{Q_{regret}}$ iff $P_Q \in \mathcal{P}_{Q_{pos}} \wedge B^a_{P_Q}(t_{P_Q}) \leq B^a_{P_Q}(t_{P_{Q_{sel}}})$.
2c: If $UI.preference =$ "*time*", the cloud chooses the query plan $P_{Q_{sel}}$ that is faster, i.e. $\forall P_Q \in \mathcal{P}_{Q_{exist}}, t_{P_{Q_{sel}}} \leq t_{P_Q}$. The regret is calculated for all possible plans that are faster, i.e. $P_Q \in \mathcal{P}_{Q_{regret}}$ iff $P_Q \in \mathcal{P}_{Q_{pos}} \wedge t_{P_Q} \leq t_{P_{Q_{sel}}}$.

In case 2 the profit $B_Q(t_{P_{Q_{sel}}}) - B_{P_Q}(t_{P_{Q_{sel}}})$ is credited to the cloud account, and can be invested on new cache structures.

The cloud calculates the regret for not investing in the alternative plans $\mathcal{P}_{Q_{regret}}$, as a weighted difference between the cost of the chosen and the alternative plan:

$$regret(P_Q) = \frac{|B^a_{P_Q}(t_{P_{Q_{sel}}}) - B^a_{P_Q}(t_{P_Q})|}{B^a_{P_Q}(t_{P_Q})}, \forall P_Q \in \mathcal{P}_{Q_{regret}} \tag{15}$$

[5]By convention, a $B_Q(t) = 0$ represents a non-defined user budget.

[6]Note that $B^a_{P_Q} = \emptyset$ also holds in the case that B_Q does not exist.

[7]In practice, if all the users define no preferences, the cloud may choose as default preference = "time".

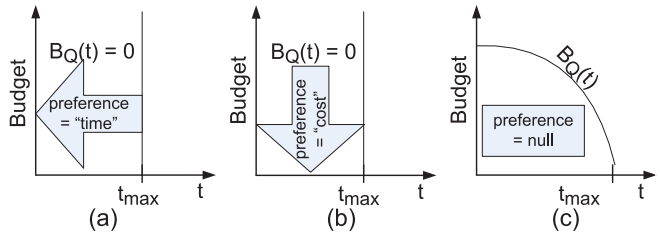

Figure 4: Examples of user preferences and budgets: (a) time-limited budget and preference on time, (b) time-limited budget and preference on cost, (c) budget function without preference.

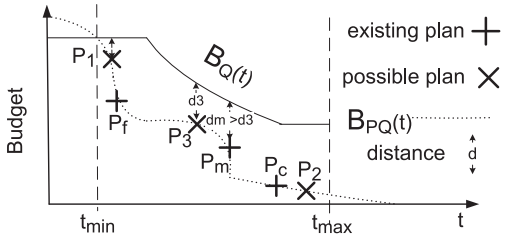

Figure 5: Comparison of the user's and the cloud budget and selection of plans for execution and regret calculation.

The role of the denominator in formula 15 is to decrease the influence of expensive plans in the regret assigned in the involved cache structures. The following example illustrates the service of a user query according to her input directions and the calculation of regret.

EXAMPLE 3. *Assume that the user sends the query Q and the input $UI = (preference, B_Q)$ where B_Q is a piecewise function limited in time, as shown in Figure 5. The skylines of the possible and existing query plans form the cloud budget B_{P_Q}. The cloud budget $B^a_{P_Q} = B_{P_Q} \wedge t \in [t_{min}, t_{max}]$, which is covered by B_Q, is considered for the plan selection. If the user prefers faster over cheaper plans, the cloud selects $P_{Q_{sel}} = P_f$ and calculates the regret for the possible plan covered by B_Q that is faster than P_f, i.e. P_1. If she prefers cheaper over faster queries, the cloud selects $P_{Q_{sel}} = P_c$ and calculates the regret for the possible plan covered by B_Q, that is cheaper, i.e. P_2. If the user does not define any preference, the cloud selects $P_{Q_{sel}} = P_m$, i.e. the plan with the cost closer to the user's budget. The regret is calculated for the possible plans P_1, P_3 with cost even closer to the user's budget.*

The cloud aims at maximizing the probability that the cloud budget is covered by the user's budget, i.e. $B^a_{P_Q} \rightarrow B_Q$. This is achieved by amortizing the cost of new structures to numerous prospective selected query plans.

5.3 Cost of Query Execution

The model that estimates the building and maintenance cost for new data structures can be found in [6]. To determine the structure to be used in the optimizer, the cache needs to find the minimal query cost for the existing structures and the plans for the potential structures. If the regret is assigned only to one plan, then the cache can invoke the optimizer twice, once with existing indexes and once with "what-if" indexes [4]. These simulate statistics for the real indexes, without actually building one. They can be maintained without the overhead of full indexes. Since each optimizer invocation takes in the order of hundreds of milliseconds, for expensive queries the overhead is small.

In case the regret is assigned to all possible query plans, the cache needs to invoke the optimizer many times to estimate costs for all possible index combinations. The overhead can be substantial. To reduce the overhead we use the configuration parametric query optimization techniques, such as INUM [32] and C-PQO [2]. In the the time to estimate costs of many index combinations are reduced by orders of magnitude. Since INUM works outside of the optimizer and C-PQO requires access to the optimizer internals, we prefer to employ INUM for the cloud. INUM caches some "key" plans for each query to determine plans for every possible index combinations. The number of plans cached for each query depends on the join and the ordering query columns. Typically, for TPCH-like complex queries, caching 100 plans on average for each query is required. INUM allows an approximation method [32] that caches only two plans per query to provide a reasonably close upper bound on the query cost. This approximation is used to determine the query cost. This technique provides the upper bound and the cache behaves conservatively, while estimating which structures to materialize.

To further reduce the cost estimation overhead, query similarity can be exploited. Typically, the queries are generated by applications, with limited free form query generation. The cloud extracts the query templates and caches the plans only for the respective queries, using techniques proposed by Ghosh et al. [9]. This drastically reduces the overhead of cost estimation. The costs estimated based on templates are upper bounds on the actual query costs. If higher accuracy is desired, the cache can monitor the actual plan used for each query and adjust the cost based on templates. Modern databases provide simple techniques to retrieve the actually used query plans [14]. Using the above techniques, the cloud reduces the overhead of cost estimation and make the query service realistic.

6. EXPERIMENTS

We present the experimental study for the cost amortization of data structures used in query services of a cloud DBMS that employs the proposed prediction and economy models.

6.1 Experimental Setup and Methodology

Setup. The cloud DBMS is set up with one back-end database. This is the best possible scenario for the back-end database as it eliminates the inter-database communication to answer the queries. The cloud DBMS is operated under a TPC-H-based workload first used in [28]. It consists of 7 TPC-H query templates and simulates the query evolution of a million real SDSS [17] queries against a 2.5TB back-end database. The workload mainly simulates the evolution of column usage seen in SDSS workloads. The authors first plot the column co-occurence matrix, and temporal locality of the columns in the SDSS workload. Then they select 7 queries and change the query composition over time to simulate similar column co-occurence and locality. We select this workload, as it is more portable across different DBMS, and the queries are tunable by using the query generation mechanism of the TPC-H benchmark. The reason that we employ a simulation of the workload rather than run the real 1 million SDSS queries, is that (i) it would take months to run the real workload, and (i) the real queries were posed by the astronomers in a period of one year, and, in order to test the prediction model in a realistic situation of data services, we needed to simulate accurately all the workload features, including time. Furthermore, in order to prove the effectiveness of the prediction model we need a big workload, for which the model could observe and regress a substantial part of it that is able to give enough information for the prediction of the rest. We copy the detailed experimental setup, including the cost model parameters from [6]. The building and maintenance cost of structures is calculated using the prices on I/O operations, disk space, CPU time and network bandwidth of Amazon EC2 [12]. Computations show that the building cost is on average 7

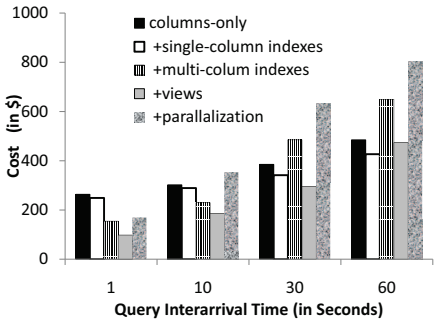

Figure 6: Sensitivity of provider cost w.r.t. availability of database features.

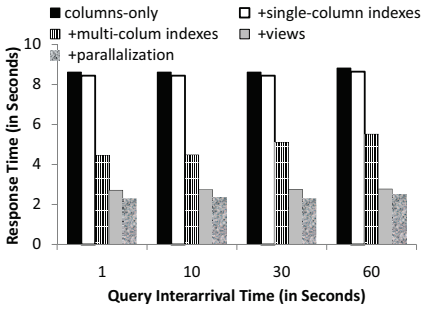

Figure 7: Sensitivity of response times w.r.t. availability of database features.

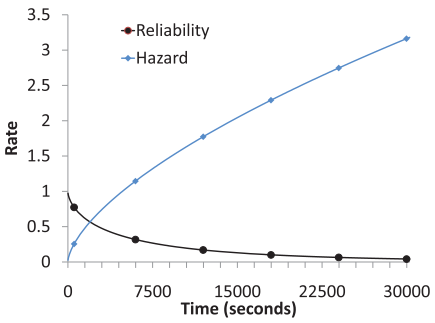

Figure 8: Balance between reliability R and cumulative hazard H.

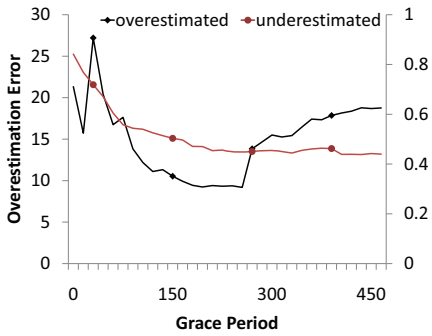

Figure 9: Avg. over- and under-estimation error w.r.t. grace period.

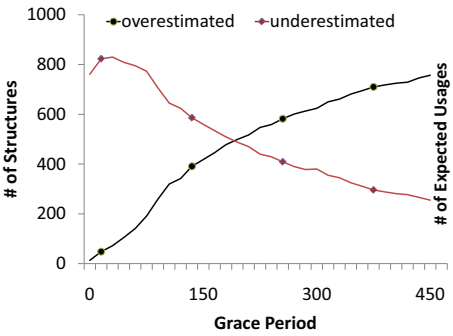

Figure 10: Avg. # of over- and under-estimated structures w.r.t. grace period.

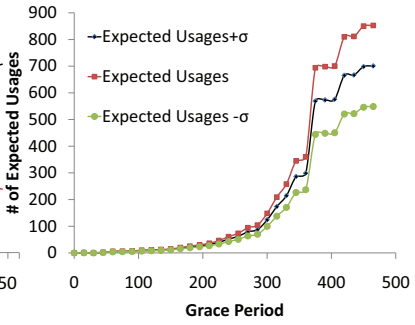

Figure 11: Avg. # of expected structure usages w.r.t. grace period.

orders of magnitude bigger than the maintenance cost. Nevertheless, the fitting of the prediction model is performed with the real SDSS workload.

Methodology. Six sets of experiments cover the major aspects of the proposed cost amortization for data services.

The first set of experiments assumes that the cloud is read-only, thoroughly altruistic, and does not charge anything for building the cache structures. The goal of these experiments is to show the variation of the cloud performance w.r.t. availability of different database features, such as indexes, materialized views, and parallelization. The second set of experiments presents results on fitting the prediction model of Section 3 based on regression of the dataset. The third set of experiments present the results on building cost amortization, as described in Section 3. The fourth set presents results w.r.t. the economy sensitivity to the two most influential workload features, the frequency of data update and the locality of data access. The fifth set of experiments presents results w.r.t. the complexity of queries, which results in long-running query execution. The sixth set of experiments presents results w.r.t. the overhead of constructing the query plans. In the first set of experiments we set the user preference equal to "time" (i.e. the economy selects fast query plans) whereas in all the rest we set preference equal to "cost".

6.2 Experimental Results

This section presents the experimental results.

6.2.1 *Comparison of different database features*

Figures 6 and 7 show the operating cost of a read-only cloud cache and the average query response time for different database features using different inter-query time intervals. We start the cloud database with only database columns, then each of the next columns in the chart adds a database feature to the one on left. We add single-column indexes, then multi-column indexes, then materialized views, and finally multiple CPUs to parallelize the query processing. We observe that the operating cost of the cloud is reasonable in presence of all the features. Hence, the caching service for SDSS-like workload is viable. We also observe that the materialized views provide the most improvement in response time and operating cost. This is especially true, since we do not consider cost of updating the materialized views, and each of the materialized view is optimal for the templates selected in the workload. Hence, the results using materialized views represent the upper bound on the cloud performance. In the absence of materialized views, the cloud should build multi-column indexes even though due to the sizes they become expensive to maintain in large query arrival intervals.

6.2.2 *Fit of the prediction model*

The SDSS-like workload contains data for the period of T = 95 days (specifically 8,214,506 seconds). The regression procedure on the workload showed that it fits the proposed prediction model. Specifically, the query arrivals are regressed w.r.t. the Poisson distribution. The mean value of the standard deviance of the residuals of the Poisson regression is $\approx -0,31$ and the respective standard deviation is $2,82 \cdot 10^{-6}$, which shows a very good fit. Moreover, this conclusion is reinforced by the R-squared value, which is close to 1 [7]. The mean of the Poisson distribution is estimated as $\lambda \approx 0,0273$ (every 1 second). The observed data is regressed using the method in Section 4 in order to calculate the failure and, therefore, the reliability of each structure. The entire workload period, T, is divided in three parts and two of them are used as the observation time T_{obs}, whereas the third one has been used for testing. The cross-validation of the regression and test sets gives the results presented in Figures 9, 10 and 11. Each time slot $t \in T_{obs}$ is set to 15 seconds. Both the observed and predicted data refer to 2054 cached columns. The sensitivity of the regression method, v is varied on the x-axis of the

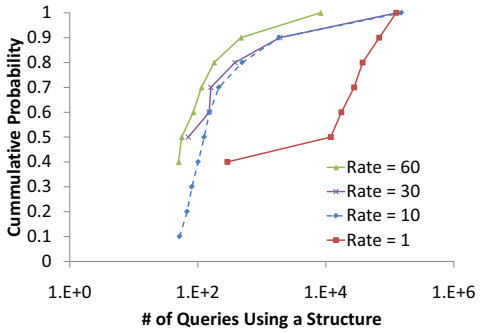

Figure 12: Cumulative probability for the number of queries that use a structure.

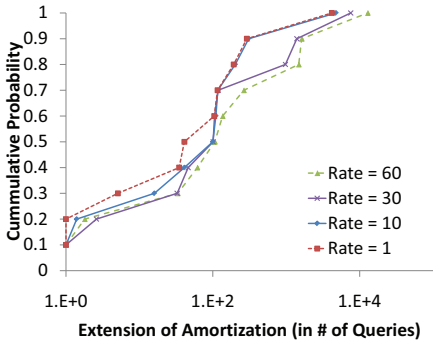

Figure 13: Cumulative probability for total cost amortization.

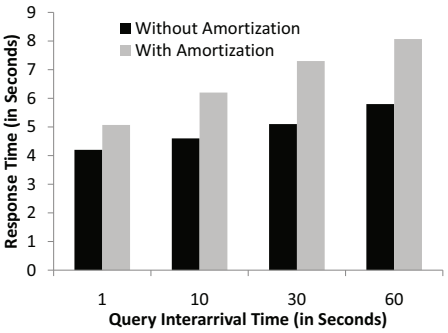

Figure 14: Response times using amortization.

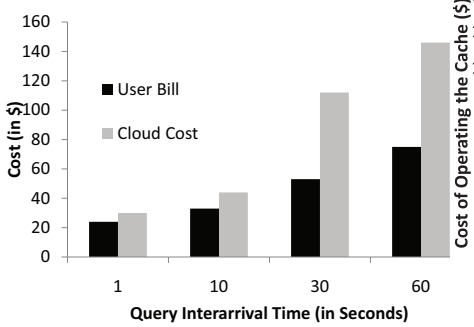

Figure 15: Building and maintenance cost recovery.

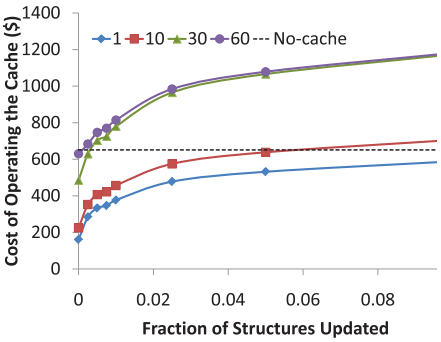

Figure 16: Sensitivity to the frequency of data update.

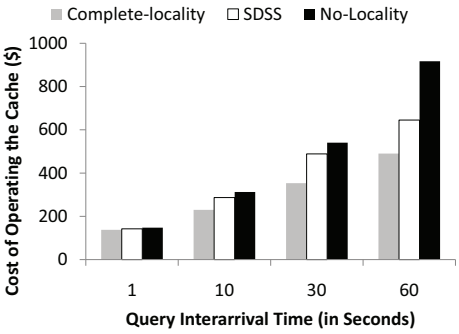

Figure 17: Sensitivity to the locality of data access

respective graphs, so that $0 - 450$ consecutive usage failures are ignored in the cache lifetime of a column. The results show the average error in over- and under-estimation w.r.t. the expected usage of the column in prospective selected query plans. Specifically, the error for each column is computed as:

$$error = \frac{\sum(expected\_\#usages - actual\_\#usages)}{\sum actual\_\#usages} \quad (16)$$

over the entire test period. Naturally, the range of the over- and under-estimation error are not comparable, and therefore they are presented separately. The lowest the errors are, the better the prediction is. Figure 9 shows that under-estimation error is monotonically decreasing, while the over-estimation error has a *bathtub* curve: For small sensitivity values, the error is big because it averages only big errors originating on columns used for long in the train set but not used at all in the test set. Inevitably, as regression becomes less sensitive to the observed usage failures, the predicted column lifetimes are longer and the model is more prone to overestimation. These two tends are balanced for $v = 150, \cdots, 300$ and the over-estimation error is minimized. Moreover, Figure 10 shows that for $v \approx 225$ there is the maximum compensation of over- and under-estimation in terms of number of columns. Naturally, there is a point of sensitivity of the regression to the survival analysis, which balances the influence of sparse and dense occurrences of column usage w.r.t. the predicted lifetime. Figure 11 shows the range (using the standard deviation σ) and the average expected number of occurrences of column usage. Naturally, the average and range of expected usages increases as the regression becomes less sensitive. Employing the regression for $v = 225$, we use survival analysis to predict the reliability of the columns. The mean μ and standard deviation σ values of the predicted failure distributions, i.e. Weibull [5], are $\mu = 291.35$, $\sigma = 1772$ for the shape parameter, and $\mu = 1.21$, $\sigma = 1.004$, for the scale parameter. This means that the failure rate f_S is increasing with time for all columns, but the increment rate df_S/dt varies a lot depending on the column. Figure 8 shows the reliability and the cumulative hazard for the mean predicted failure distribution. The maximum amortization time t_e varies in $[0, 188830]$ sec. and $t_e = 1816$ sec. on average for $v = 225$. 524 columns are predicted with zero lifetime; this is a good prediction since these columns are actually used by 2 queries on average and by 14 queries at most.

6.2.3 *Effect of cost amortization*

This set of experiments examines the effect of cost amortization to the cloud economy. The cloud remains altruistic but, unlike in the case of Section 6.2.1, the building cost of a structure is amortized to the queries for which a plan that uses this structure is selected. Figure 12 shows the computed cumulative distributions for the number of queries that use each cache structure in the workload, while varying the inter-arrival time. Apparently, all structures are used at least by 80 queries even if the queries arrive every 60 seconds. As expected, if the query arrivals are very frequent, i.e. every 1 second, then the number of queries that use a structure increases exponentially and reaches the number of at least 10000 queries with 0.5 probability. Figure 13 shows the computed cumulative distributions for the recovery of the total cost through its amortization on prospective queries. The results show that the success of total cost recovery is sensitive to the arrival rate of queries. The reason is that as the inter-arrival time increases, the maintenance cost increases, too. In order to keep the total cost of the query plans low, the structure cost has to be amortized on more queries to be fully recovered.

The next experiment shows the results for response time and actual cost recovery if the cost is amortized to 100 queries. We run this experiment using column caching and multi-column indexes while

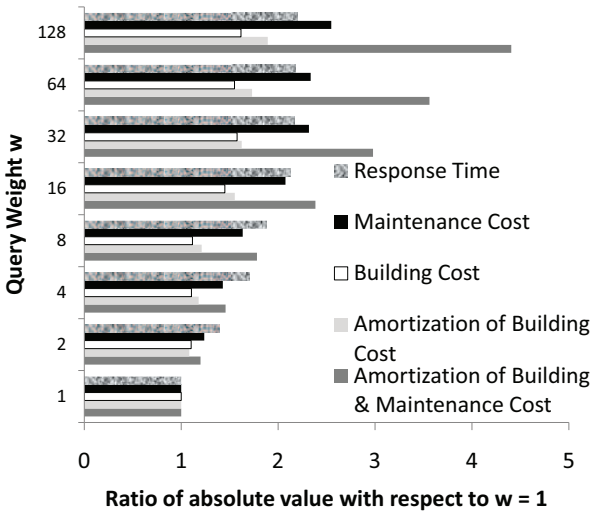

Figure 18: Cost and response time w.r.t. query complexity.

varying query inter-arrival time. Figure 14 shows the response time in the case that the cloud targets to the maximization of cost recovery from the user. Naturally, the cloud is conservative in building new structures and the response time is bigger than in the case of Section 6.2.1. However, as shown in Figure 15, the cloud recovers most of the cost for high to average arrival rates (every 1-10 seconds). In case of sparse arrivals (every 60 seconds), the maintenance cost increases substantially, reducing the competitiveness of the plans, and their chances to be selected to serve query execution. Note that, the cloud does not charge anything after the entire cost of a structure has been recovered.

Overall, the experiments prove the viability of the altruistic cloud, which reconciles the following policies: (i) moderately charging the users for the built structures, giving priority to their satisfaction in terms of cost (i.e. offering them the cheapest plan) (ii) recovering the operational cost with a big guarantee, and (iii) enhancing the offered query services by building new structures. These are the policies that tune an altruistic cloud, as denoted in Section 5.1. Therefore, the proposed cloud achieves its purpose.

6.2.4 Sensitivity to workload changes

Figure 16 shows the sensitivity of the cost for operating the cloud cache to the update frequency in the workload. The results correspond to the scenario that the structures are evicted upon any change to related data. The x-axis shows the fraction of structures that are evicted because of updates and the y-axis shows the the cost of query execution. The cache uses Amazon's cost model, with indexes. For comparison, the graph includes the results for query execution performed thoroughly in the back-end database. For small (= 1 sec.) or moderate (= 10 sec.) query inter-arrival interval, the cloud cache remains beneficial to the user even for high-frequency data update. For big inter-arrival intervals the cached structures are evicted because of data update before they are used to answer the queries. Therefore, the cache is beneficial to the user only for small update frequency.

Figure 17 shows the sensitivity of the cache to the data-access locality. The SDSS workload is compared with two artificial ones. The first is a workload with zero locality, i.e. the query data access is generated randomly. The second is a workload with high locality, i.e. each query is issued about 150,000 times before switching over to a new one. The graph shows that the cost of operating the cache with the SDSS workload is very close to the workload with high locality. Even with the random workload, the cache performs well compared to the SDSS workload, especially for small to moderate query inter-arrival intervals.

6.2.5 Sensitivity to complex queries

Figure 18 shows the effect of complex queries, i.e. long-running queries, on the cloud DBMS. To observe the economy behavior on complex queries, for each possible query plan, we scale the intended running time by a factor w, as shown on the y-axis. The x-axis shows the (i) the average response time, (ii) the maintenance cost, (iii) the building cost of all structures (iv) the amortized building cost for all structures over the entire workload and (v) the amortized of both building and maintenance cost. Concerning the results of trend (v), the maintenance cost was amortized by charging the cheapest (and therefore selected for execution query plan) more but not exceeding the cost of the second cheapest plan. Instead of showing the absolute values, we show the ratio of these values over the respective values corresponding to $w = 1$. In this way we save space by presenting all the trends in the same graph. These results refer to query inter-arrival rate equal to 10 seconds. All values increase as the query weight (w) is increased. The increment, however, is sub-linear. As w increases, the regret increases proportionally to the cost of the possible plans, but the cost to build the structures remains the same. This allows the cloud to build structures much earlier and turn 'possible' plans to 'existing' ones. These new plans are faster but cheaper than old plans, and, thus, they are selected for execution. Therefore, as times goes query services improve, as they are both faster and cheaper. Therefore the response time remains small, even for very complex queries; e.g., for the query plans that are assumed to run $w = 128$ slower than they actually do, the average response time of queries increases only by a factor of 2, because of faster availability of expensive structures. Since the cloud builds more structures, the building and maintenance cost increases. The amortization of the building cost remunerates all of it for $w = 1$ and more than the actual cost as w increases. This increment is much bigger in case the maintenance cost is amortized, too. Therefore, cost amortization not only ensures the economic viability of the cloud, but also its profitability.

6.2.6 Query cost estimation overhead

As discussed in Section 5.3, cost estimation is the biggest overhead in query execution in the cloud. This experiment compares the overhead of different cost estimation methods and their accuracies. The first column of Table 1 shows the total number of optimizer calls that the cloud performs to estimate the query costs. The *Query* method calls the optimizer for each query and each potential combination of indexes; the *INUM* method uses INUM to reduce the combinations needed investigation; the *INUM Approx* method approximates INUM's cost estimation by calling the optimizer only once for each query; *Template* identifies the templates from the queries and uses the templates to estimate the costs for every index combination. The next two techniques, i.e., *Template + INUM* and *Template + INUM Approx* use the INUM and approximation methods. The results show the expected number of optimizer calls for each of these techniques. The *Query* method is the most expensive and it invokes the optimizer about 150 million times. *INUM* improves on that by reducing the number of calls by an order of magnitude, but is still quite expensive. Overheads of *INUM Approx* and *Template* methods are very similar, and using *INUM + Template* reduces the overhead by another factor of 9. Finally, Template + INUM Approx achieves the lowest overhead (about 1803 optimizer calls). The second column of Table 1 shows the cost estimation error for the techniques. The table presents the percentage of average extra cost assigned to the query w.r.t. *Query*. *Template* achieves the smallest estimation error, and *Template + INUM Approx* incurs the highest estimation

Estimation Technique	Estimated Optimizer Calls	Percentage Over Estimation
Query	147396994	0
INUM	18255591	9.051037
INUM Approx	676133	25.22721
Template	387960	4.657116
Template+INUM	48681	13.79859
Template+INUM Approx	1803	32.06346

Table 1: Comparison of query-plan estimation techniques.

error. Comparing the number of optimizer calls with the respective estimation error, we suggest that the cloud uses *Template + INUM* for a balance between the accuracy and overhead of cost estimation.

7. CONCLUSION

In this paper we propose the cost amortization of data structures employed in data services in order to ensure the economic viability of the data provider. We present a novel generic stochastic model that predicts the appropriate amortization in time and number of prospective queries. A novel data regression model processes query statistics and gives input to the prediction model. In order to study the effectiveness of cost amortization we extend an existing economy model for the management of a cloud DBMS. A thorough experimental study demonstrates that cost amortization of data structures can achieve the economic viability of the cloud DBMS, and, moreover, its profitability, while improving the offered query services in terms of both cost and response time.

8. REFERENCES

[1] D. Agrawal, A. E. Abbadi, F. Emekci, and A. Metwally. Database management as a service: Challenges and opportunities. In *ICDE '09*, pages 1709–1716, 2009.

[2] N. Bruno and R. V. Nehme. Configuration-parametric query optimization for physical design tuning. In *SIGMOD*, 2008.

[3] B. Calder and D. Grunwald. Next cache line and set prediction. In *ISCA*, pages 287–296, 1995.

[4] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL server. In *VLDB*, 1997.

[5] D. R. Cox and D. Oakes. *Analysis of Survival Data*. CRC Press, 1984.

[6] D. Dash, V. Kantere, and A. Ailamaki. An economic model for self-tuned cloud caching. In *ICDE*, pages 1687–1693, 2009.

[7] J. J. Faraway. *Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models*. CRC Press, 2006.

[8] D. G. Feitelson. Locality of sampling and diversity in parallel system workloads. In *Intl. Conf. Supercomputing*, 2007.

[9] A. Ghosh, J. Parikh, V. S. Sengar, and J. R. Haritsa. Plan selection based on query clustering. In *VLDB*, 2002.

[10] M. Gibas, G. Canahuate, and H. Ferhatosmanoglu. Online index recommendations for high-dimensional databases using query workloads. *IEEE TKDE*, 20(2):246–260, 2008.

[11] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 29, Washington, DC, USA, 2002. IEEE Computer Society.

[12] http://aws.amazon.com/ec2/.

[13] http://code.google.com/appengine/.

[14] http://msdn.microsoft.com/en us/library/ms189747.aspx.

[15] http://www.microsoft.com/azure/.

[16] http://www.salesforce.com/.

[17] http://www.sdss.org/.

[18] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. *SIGARCH Comput. Archit. News*, 30(2):209–220, 2002.

[19] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *In SIGMOD*, pages 647–658, 2004.

[20] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Comput.*, 57(4):433–447, 2008.

[21] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. Correlation maps: A compressed access method for exploiting soft functional dependencies. In *VLDB*, 2009.

[22] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. *SIGARCH Comput. Archit. News*, 29(2):144–154, 2001.

[23] C. A. Lang and A. K. Singh. Modeling high-dimensional index structures using sampling. *SIGMOD Rec.*, 30(2):389–400, 2001.

[24] H. Li, D. Groep, and L. Wolters. Workload characteristics of a multi-cluster supercomputer. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

[25] H. Li, M. Muskulus, and L. Wolters. Modeling job arrivals in a data-intensive grid. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 2006.

[26] H. Li, M. Muskulus, and L. Wolters. Modeling correlated workloads by combining model based clustering and a localized sampling algorithm. In *ICS*, pages 64–72. ACM, 2007.

[27] T. Malik, R. C. Burns, and A. Chaudhary. Bypass caching: Making scientific databases good network citizens. In *ICDE*, 2005.

[28] T. Malik, X. Wang, R. Burns, D. Dash, and A. Ailamaki. Automated physical design in database caches. In *SMDB*, 2008.

[29] T. N. Minh and L. Wolters. Modeling job arrival process with long range dependence and burstiness characteristics. In *CCGRID*, pages 324–330. IEEE, 2009.

[30] T. N. Minh and L. Wolters. Modeling parallel system workloads with temporal locality. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 2009.

[31] M. A. Nieto-Santisteban, J. Gray, A. S. Szalay, J. Annis, A. R. Thakar, and W. J. OÕmullane. When database systems meet the grid. In *CIDR*, 2005.

[32] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient use of the query optimizer for automated database design. In *VLDB*, pages 1093–1104, 2007.

[33] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[34] H. Taylor and S. Karlin. *An Introduction to Stochastic Modeling*. Academic Press, third edition, 1998.

[35] A. Tewari and P. Bartlett. Optimistic linear programming gives logarithmic regret for irreducible mdps. In *Advances in Neural Information Processing Systems 20*.

[36] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. In *VLDB*, 2009.

[37] X. Wang, T. Malik, R. C. Burns, S. Papadomanolakis, and A. Ailamaki. A workload-driven unit of cache replacement for mid-tier database caching. In *DASFAA*, pages 374–385, 2007.