COCCUS: Self-Configured Cost-Based Query Services in the Cloud

Ioannis Konstantinou⋆ Verena Kantere◊ Dimitrios Tsoumakos† Nectarios Koziris⋆

⋆CSLAB, National Technical University of Athens {ikons,nkoziris}@cslab.ece.ntua.gr

◊Institute of Services Science University of Geneva verena.kantere@unige.ch

†Department of Informatics Ionian University dtsouma@ionio.gr

ABSTRACT

Recently, a large number of pay-as-you-go data services are offered over cloud infrastructures. Data service providers need appropriate and flexible query charging mechanisms and query optimization that take into consideration cloud operational expenses, pricing strategies and user preferences. Yet, existing solutions are static and non-configurable. We demonstrate *COCCUS* a modular system for cost-aware query execution, adaptive query charge and optimization of cloud data services. The audience can set their queries along with their execution preferences and budget constraints, while *COCCUS* adaptively determines query charge and manages secondary data structures according to various economic policies. We demonstrate *COCCUS* 's operation over centralized and shared nothing CloudDBMS architectures on top of public and private IaaS clouds. The audience is enabled to set economic policies and execute various workloads through a comprehensive GUI. *COCCUS* 's adaptability is showcased using real-time graphs depicting a number of key performance metrics.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distrib. databases*

Keywords

Cloud computing, Economy, Amortization

1. INTRODUCTION

The IT industry has widely adopted the cloud computing paradigm in order to benefit from its prominent characteristics, such as the 'pay-as-you go', i.e., the ability to rent and utilize only the required resources for as long as they are needed. This feature is also offered for cloud data services: Whether they are low-level generic PaaS services like Microsoft's Azure [1] and Google's AppEngine [2] or high-level query processing frameworks like Google's BigQuery [3] and Infochimps [4], they are offered on the basis of charging a small fee per transaction or hourly operation.

Cloud providers need to tackle two challenges to offer data services: first, they need appropriate and flexible query charging mechanisms to ensure their economic viability, while offering competitive prices; second, they need to manage efficiently the life-cycle of secondary data structures, like indexes, materialized views, etc, to offer low query response time. Both query charging and data-structure management need to consider operational expenses, incurred by query execution and, building and maintaining data structures, respectively. Yet, most data service providers offer a flat charging policy which does not take into account such cloud expenses, at least in a fine-grained and dynamic manner.

Existing data service providers offer limited support for query optimization based on automatic data-structure building and maintenance: usually, the user is enabled to define some data optimizations through a GUI or configuration files. Yet, it is necessary to provide through the cloud full DBMS capabilities self-tuned w.r.t. cost and response-time constraints. Naturally, traditional DBMSs that offer fully fledged query optimization, lack the incorporation of the notion of cost in building and maintaining data structures.

Along the lines for cloud data management [11], we present *COCCUS*, a system for self-**CO**nfigured, **C**ost-based **C**loud q**U**ery **S**ervices. *COCCUS* takes as input user queries and preferences and executes the queries in a DBMS hosted in a IaaS provider. *COCCUS* self-tunes dynamically query performance and query charge to the user preferences and the pricing scheme of the IaaS provider. *COCCUS* features:

- **Cost-aware query execution:** *COCCUS* incorporates the notion of cost in query planning and execution.
- **Adaptive query charge and optimization:** *COCCUS* calculates query charge and schedules data-structure building and maintenance on-the-fly by considering user budget constraints and system-specific economic policies.
- **Cloud economy abstraction:** *COCCUS* is a generic modular system that can be installed on top of different cloud DBMS architectures. Currently, it supports centralized and shared-nothing architectures.

This demonstration enables the audience to examine the behavior of *COCCUS* by varying the: (i) cloud setup: allows a choice of DBMS architectures and query provider's economic policies (ii) workload: allows customization of workload skewness, query arrival rates, data update frequencies and various user preferences.

COCCUS is the prototype system that implements the work in [13]. The latter proposes a prediction model for cost amortization of query services and develops a cloud economy for offering and charging query services. The work in [15]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$10.00.

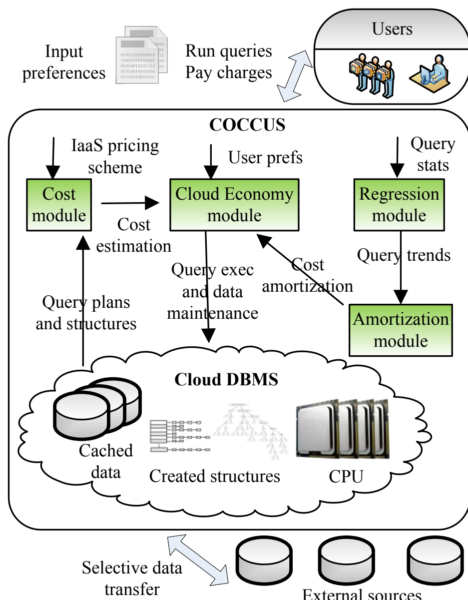

Figure 1: The *COCCUS* Architecture

deals also with the problem of cost amortization of data optimizations. Complementary to [13], this approach considers users to be selfish players, who need to input a possibly false valuation. This solution focuses on cost-recovery and truthfulness at the expense of efficiency. The work in [14] is a theoretic approach that computes the price of a query based on given prices for a set of views. This is orthogonal to [13], in which query execution cost and, building and maintenance cost for data structures is computed directly based on the cost of the cloud infrastructure involved in these operations.

Employing economic notions and methods to manage cloud services has become a trend. For example, Scarce [8] is a framework that manages rental of computing resources, migration and replication, based on their 'economic fitness' in order to perform cost-effective resource allocation. Moreover, the introduction of the notion of cost in cloud data management has given rise to research on cost models. The work in [12] proposes such models for the materialization, maintenance and storage of data views in order to perform static optimization of cost and response time.

2. ARCHITECTURE

COCCUS operates on top of an IaaS provider and runs user queries on data that reside in external data sources or in the cloud. *COCCUS* may either belong to the cloud IaaS provider, or to a separate query service provider that uses services from the underlying IaaS provider. The architecture of *COCCUS* is illustrated in Figure 1. *COCCUS* takes as input user queries and preferences for query execution and cost, and outputs the query result and execution cost charged to the user. The goal of *COCCUS* is to (i) execute queries in the best possible way according to user preferences and cloud economic policies, (ii) accelerate query execution by gradually building and maintaining appropriate data structures, and (iii) keep query cost low by amortizing the building and maintenance cost of data structures to a large number of queries that employ them for execution.

The query is executed in CloudDBMS, which stores cached data along with data structures in a typical data-store (ranging from single-machine to fully distributed key-value store) on top of a IaaS provider, like Amazon's EC2. The query execution cost and the cost of building and maintaining data structures is estimated by the Cost module by utilizing the charging policy of the underlying IaaS provider. The Cloud economy module takes as input the query cost estimation along with the user preferences and selects the query plan to be executed, as well as manages the lifetime of data structures in CloudDBMS. The Regression module analyzes query traffic, and estimates future query trends. The Amortization module receives query trends along with cost estimation for building and maintaining data structures and predicts the number of prospective queries in which this cost should be amortized. The amortized cost of data structures is fed to the Cost module. In detail:

Cloud DBMS: This module contains cached data, indexes, materialized views and a number of reserved CPU processors for parallelizing query traffic. It consists of both the software (i.e., the DBMS) and the hardware (i.e., the reserved cloud storage and computation infrastructure). The demonstration system supports various DBMS architectures ranging from typical single-machine centralized solutions to fully distributed shared-nothing systems running on top of different IaaS providers.

Cloud Economy module: This module determines both the query plan to be executed and the schedule of building new data structures, according to user preferences and cloud economic policies. User preferences include a coarse-grained selection between short execution time or small query cost. Economic policies refer to the prioritization of the general economic goal of the query service provider, which can range from satisfying individually each user query, accelerating overall query execution and increasing cloud profit (see Section 3). Generally, the module selects a plan for execution from a pool of alternative query plans that can be executed right away; moreover, it consults a pool of what-if alternative query plans in order to decide which new data structures to build and when to build them.

Cost module: This module takes as input a query plan (ready to be executed or what-if) or a data structure and outputs the cost estimation, depending on the network traffic, I/O operations, CPU time and storage needed to execute the plan or build (or maintain) the data structure, respectively. The module takes into account the pricing scheme of an IaaS provider like Amazon, RackSpace or GoGrid.

Regression module: This module takes as input the past query workload and implements a novel regression method that fits this workload in probability distributions of the locality of data request and update. The method creates a time series input to a training procedure that is executed periodically and fits the results into a Poisson distribution.

Amortization module: This module amortizes the cost of building and maintaining data structures into prospective future queries that will be executed using these structures. It takes as input the probability distributions output by the Regression module and implements a novel prediction model. The latter balances two opposite tends: short-term and long-term cost amortization. The former aims at fast pay-off, but leads to expensive query plans that may not be selected for execution because of the existence of cheaper alternative plans; the latter aims at small individual payments, but leads to prolonged amortization that runs the risk of remaining incomplete due to change of data request locality or structure invalidation because of data update.

The prediction model estimates the lifetimes of data structures based on survival analysis techniques [10] and breaks the building cost into a number of queries which are expected to occur in the structure's lifetime.

3. CLOUD ECONOMY

In this section we summarize the functionality of the query service provider. For details the reader is referred to [13].

Overall functionality: The user inputs a query to the query service provider as well as preferences for query execution. User queries are charged in order to be served and query performance is measured in terms of execution time. The user preferences with respect to query execution consists of (i) optionally, a preference between fast and cheap execution and, (ii) optionally, a specific budget as a function of time. The provider receives the query and the user preferences and produces alternative query plans. The cost of these plans is estimated and juxtaposed to the user preferences. If there are plans that are in the budget and preferences, the provider chooses the most appropriate one of them, w.r.t. an economic policy. The provider has an account where user payments are deposited and money is invested in new inventory, i.e. structures for faster execution. The latter are indexes, materialized views, cached columns and parallelization of execution on multiple processors.

The investment in structures is based on the notion of *regret*. The provider cannot offer services that employ structures not yet built. A built structure can benefit execution of some queries, either in terms of time or cost. The regret for not offering services because a structure is not built is accumulated and monitored; if the regret for the absence of a structure becomes substantial, the provider decides to invest in the construction of this structure.

The regret is the incentive for the improvement of the query services. The cost of new inventory is paid from the provider'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 cost of the offered services.

The provider maintains a structure pool relevant to the recent past queries. Upon receiving a query, it considers a set of plans that include only existing structures and a plan set that includes also possible new structures. It determines the optimal plan than can be executed right away and the investment in new structures. From the implementation point, the cloud finds structures it can use to execute a query. First, it requires the column information for all the tables included in the query. Second, it determines the index sets that are possibly beneficial by analyzing the query structure. Queries are parallelized using bushy plans.

Query execution cost: The execution cost of a query plan is estimated based on the I/O, CPU, network and disk that it utilizes, according to a pre-defined price list. This estimation is based on statistics maintained in the cloud DBMS for similar queries and depends on the cloud infrastructure architecture. In this demonstration we focus on two architecture types, namely a centralized and a shared-nothing DBMS. For each architecture, we have executed a number of representative queries and we have collected their infrastructure resource utilization.

Cost prediction and amortization: The user pays for each query service the cost of query execution and part of the building cost of data structures employed in execution. The building cost of a new structure is amortized to future query services that employ this structure. A novel prediction model outputs the extent in time and number of prospective queries for cost amortization. In case the structure is evicted because of data update or storage cost exceeding the building cost, the provider bears the non-amortized cost. The predictions are based on observation of past workload. A novel regression method provides input to the prediction model. The regression transforms statistics on query traffic into estimations for the lifetime of potential structures from the building to eviction time.

Economic policies: The cloud economy is self-tuned to the following policies: P_1 individual user satisfaction with the received query services w.r.t. charging; P_2 increase of overall quality of query services, and P_3 cloud profitability at all times. The self-tuning reflects on both the query plan selection for execution and the decision for building new structures. Appropriate decision making for building new structures and appropriate selection of query plans for execution can lead, eventually, to self-tuning to all three policies. According to P_1, the provider selects the query plan closest to the user preferences (e.g. the plan closer to the user budget function) and charges the actual cost of the query plan. Naturally, cloud credit increases gradually and allows for gradual building of new structures. According to P_2 the provider selects for execution the fastest query plan within user preferences and takes decisions for building new data structures based on low regret values. Query services are improved and, in case of good data locality, individual user payments are reduced, achieving both fast and cheap service provision. According to P_3 the provider selects for execution the query plan that allows for the largest cloud profit and charges the query at the top of the user budget. Cloud profit increases fast which allows decisions with low regret and fast improvement of query services.

4. DEMONSTRATION DESCRIPTION

Demonstration scenarios: The demonstration includes motivating scenarios that introduce the prototype system and the research results that it implements, and interactive scenarios that allow the audience to experience the system functionality in various configurations.

(A) Motivating scenarios: Two motivating scenarios exhibit the system operation in two situations. The first scenario shows the operation of a startup IT company that offers query services on a fully fledged cloud DBMS hosted in an IaaS provider and the second shows the operation of an academic query service provider. Naturally, these two environments follow different economic policies (e.g. the first may prioritize cloud profit whereas the second may prioritize query execution quality) and have to serve different query workloads (e.g. the first may serve short one-time queries from millions of users, whereas the second may serve long running computationally intensive queries by few users). It is shown that *COCCUS* (i) adapts its operation to the workload peculiarities and user preferences, (ii) ensures the quality of the primary economic policy, and (iii) gradually, ensures the qualities of the other economic policies, by scheduling structure building and by amortizing their cost.

(B) Interactive scenarios: A range of dynamic interactive scenarios exhibit the system's ability to identify the best query plans, accelerate query execution by building appro-

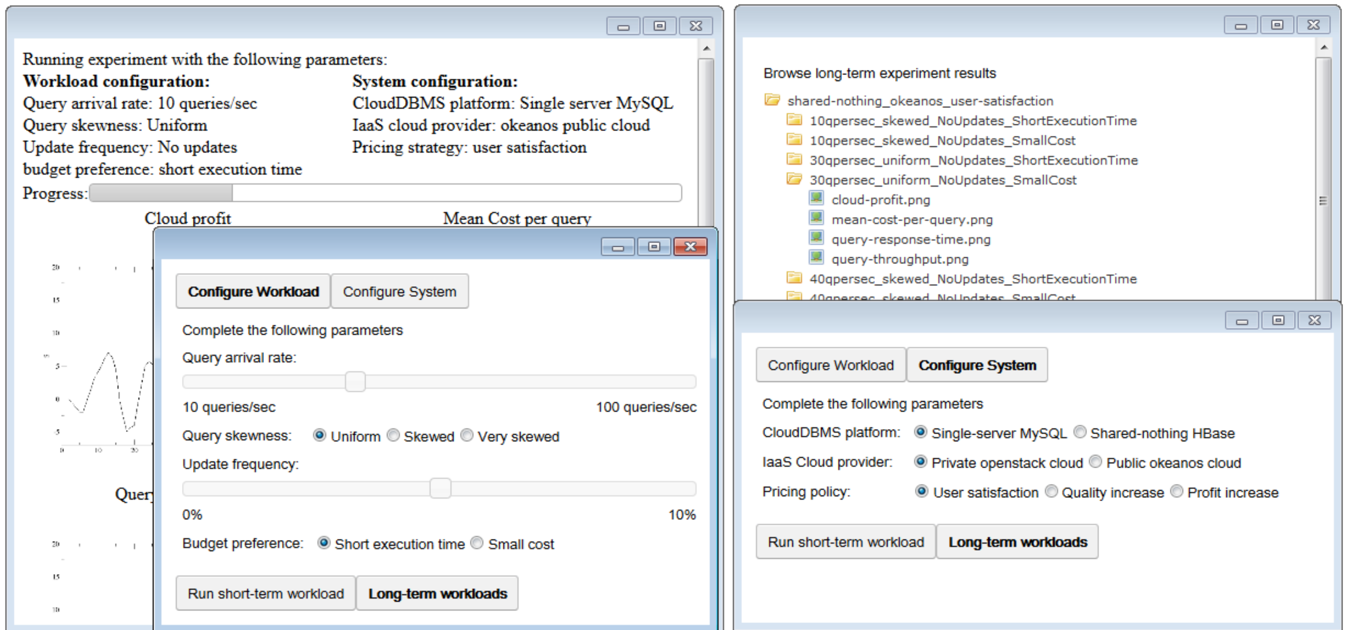

Figure 2: ***COCCUS*** **demo interface.**

priate data structures while achieving low query cost, under different workloads, user preferences, data-stores and cloud infrastructures. The scenarios enable audience interaction for the observation of (i) real-time and (ii) macroscopic results: (i) The attendee can launch a relatively small number (e.g., around 1K) of queries and observe performance metrics as the workload execution evolves (back-left graph, Figure 2), or (ii) she can browse macroscopic pre-compiled aggregated results that show the system's behavior in a big time window, (back-right graph, Figure 2). The metrics include latency and mean response time, the mean query cost and the *COCCUS*'s profit.

The demonstration allows attendees to interact with *COCCUS* on two general axes: Workload and system configuration. Interaction is enabled through a web interface, with available options organized in two tabs:

Workload configuration: The options for workload configuration are depicted in the front-left graph, Figure 2. It consists of a TPC-H based query workload created by the widely used YCSB[9], Yahoo's Cloud Serving Benchmark.

Attendees can configure query arrival rate choosing from a range between 10 up to 100 queries/sec, and query's skewness between uniform and several skewness degrees. Also, they can select the query update frequency, i.e., the workload percentage that performs data updates and, thus, invalidates built data-structures. Finally, attendees can set query execution preferences on time (i.e., fast query plans) or cost (i.e. cheap query plans).

System configuration: The options for system configuration are depicted in the front right graph, Figure 2. Attendees can set the CloudDBMS platform on top of which *COCCUS* operates, by choosing among a single-server centralized MySQL DBMS and HBase [5], a fully distributed NoSQL store that utilizes sharding without replication. Also, they can define the IaaS cloud on top of which they will deploy the CloudDBMS choosing between a private OpenStack [6] cloud and the [7] okeanos public cloud. The applied pricing scheme translates resource usage to actual costs by utilizing an Amazon EC2 price list. Finally, they can set the *COCCUS* 's economic policy, choosing among user-satisfaction (i.e., select query plans that best fits the user preferences), query quality increase (i.e., aggressively invest margin in building new data structures) and profit increase (i.e., execute the query plan that best ensures *COCCUS* 's profit).

5. ACKNOWLEDGMENTS

This work was partially supported by the European Commission in terms of the CELAR 317790 FP7 project (FP7-ICT-2011-8).

6. REFERENCES

[1] http://www.windowsazure.com.
[2] http://appengine.google.com.
[3] http://cloud.google.com/bigquery.
[4] http://www.infochimps.com.
[5] http://hbase.apache.org.
[6] http://www.openstack.org.
[7] https://okeanos.grnet.gr.
[8] N. Bonvin, T. G. Papaioannou, and K. Aberer. Autonomic SLA-Driven Provisioning for Cloud Applications. In *CCGRID*, pages 434–443, 2011.
[9] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SOCC*, pages 143–154, 2010.
[10] D. Cox and D. Oakes. *Analysis of Survival Data*, volume 21. Chapman & Hall/CRC, 1984.
[11] C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *CIDR*, 2011.
[12] L. D'Orazio, S. Bimonte, J. Darmont, et al. Cost Models for View Materialization in the Cloud. In *Proc. of EDBT-ICDT/DanaC 2012)*, 2012.
[13] V. Kantere, D. Dash, G. Gratsias, and A. Ailamaki. Predicting Cost Amortization for Query Services. In *SIGMOD*, pages 325–336, 2011.
[14] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Query-based Data Pricing. In *PODS*, pages 167–178, 2012.
[15] P. Upadhyaya, M. Balazinska, and D. Suciu. How to Price Shared Optimizations in the Cloud. *PVLDB*, 2012.