Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2010-014 March 14, 2010

Relational Cloud: The Case for a Database Service

Carlo Curino , Evan Jones , Yang Zhang , Eugene
Wu, and Samuel Madden

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Relational Cloud: The Case for a Database Service

Carlo Curino
curino@mit.edu

Evan Jones
evanj@mit.edu

Yang Zhang
yang@csail.mit.edu

Eugene Wu
eugenewu@mit.edu

Sam Madden
madden@csail.mit.edu

ABSTRACT

In this paper, we make the case for "databases as a service" (DaaS), with two target scenarios in mind: (i) consolidation of data management functionality for large organizations and (ii) outsourcing data management to a cloud-based service provider for small/medium organizations. We analyze the many challenges to be faced, and discuss the design of a database service we are building, called Relational Cloud. The system has been designed from scratch and combines many recent advances and novel solutions. The prototype we present exploits multiple dedicated storage engines, provides high-availability via transparent replication, supports automatic workload partitioning and live data migration, and provides serializable distributed transactions. While the system is still under active development, we are able to present promising initial results that showcase the key features of our system. The tests are based on TPC benchmarks and real-world data from *epinions.com*, and show our partitioning, scalability and balancing capabilities.

1. INTRODUCTION

Database systems provide an extremely attractive interface for managing and accessing data, and have proven to be wildly successful in many financial, business, and Internet applications. However, they have several serious limitations:

1. *Database systems are difficult to scale.* Most database systems have hard limits beyond which they do not easily scale. Once users reach these scalability limits, time consuming and expensive manual partitioning, data migration, and load balancing are the only recourse.
2. *Database systems are difficult to configure and maintain.* Administrative costs can easily account for a significant fraction of the total cost of ownership of a database system. Furthermore, it is extremely difficult for untrained professionals to get good performance out of most commercial systems–for example, in [19], we found that it took several months for a senior graduate student to tune and configure a commercial parallel database system for a simple 8-query workload.
3. *Diversification in available systems complicates selection.* The rise of specialized database systems for specific markets (e.g., main memory systems for OLTP or column-stores for OLAP) complicates system selection, especially for customers whose workloads do not neatly fall into one category.
4. *Peak provisioning leads to unneeded costs.* Database workloads are often bursty in nature, and thus, provisioning for the peak often results in excess of resources during off-peak phases, and thus unneeded costs.

In this paper, we make the case that "databases as a service" (DaaS) can address these limitations.

Our vision is that users should have access to database functionality without worrying about provisioning hardware and configuring software, while providers should be able to manage several databases without dedicating hardware and administrators to each database. We try to achieve this vision with our Relational Cloud[1] system, that hosts multiple databases on a pool of commodity servers inside one data center. Relational Cloud is appropriate for a single organization with many individual databases (a private cloud), or as a public service (a public cloud) that allows database provisioning and administration to be outsourced to a third-party provider. By centralizing and automating databases, there are substantial opportunities to reduce initial and operational costs while maintaining quality of service—as proven by the successful experience of *salesforce.com*, an example of ad-hoc multi-tenancy for a specific set of applications.

Table 1: Requirements for Database as a Service

User Requirements	
U1	simple API, with near-zero configuration and administration (i.e., no tuning)
U2	high-performance (e.g., throughput, latency, scalability)
U3	high availability and reliability (e.g., hot standby, backup)
U4	easy access to advanced features (e.g., snapshot, analytics, time travel)
Provider Requirements	
P1	meet user service level agreement (potentially under dynamic workloads)
P2	limit HW and power costs (e.g., intense multiplexing)
P3	limit administration costs (e.g., personnel costs)
Public Cloud Requirements	
C1	pricing scheme: cheap, predictable and proportional to actual usage (elasticity)
C2	security and privacy guarantees
C3	low-latency (relevant for OLTP and Web applications)

Table 1 summarizes a set of key requirements for Database as a Service we have identified after discussions with many researchers and potential users. From the user's perspective, the main need is a DB service with a simple interface that does not need tuning and administration (U1). This is an improvement over traditional solutions that requires provisioning, DBMS selection, installation, configuration, and administration. The user wants good performance, expressed in terms of latency and throughput (U2), that is independent of data sizes and workload changes. This is currently a challenging task that requires extensive human analysis, and costly software and hardware upgrades. High availability is

[1] See `http://relationalcloud.com`.

another key requirements (U3), which is typically offered by traditional databases, but requires careful configuration and maintenance. Finally, advanced database management features, such as database checkpointing, time travel, and analytics, should be readily available and easy to use(U4).

The other important perspective is that of infrastructure providers, responsible for managing the hardware and software providing the service. The provider's goal is to meet the service level agreements (P1), despite data and workload changes. The system must be efficient, using hardware resources effectively (P2). The service model provides the opportunity to do this, by multiplexing workloads and dynamically adjusting resource allocations. Finally, the amount of administration should be minimized (P3). This is achievable with sophisticated workload analysis tools and by centralizing the management of many databases.

There are additional requirements for providers of a public service, such as pricing schemes (C1), security/privacy (C2), and latency (C3). However, these issues are not specific to databases, and can be addressed with the techniques being devised by the cloud-computing community. Thus, we do not discuss them further.

The above list of requirements drove our design choices and helped shape our architecture, as we discuss in Section 2. The system builds on a collection of specialized storage engines, driven by a custom-built distributed transaction coordinator, which also supports high-availability via fail-over to replicas. To achieve elasticity, scalability, and efficient usage of resources, we face major research challenges in the area of adaptability and workload analysis. The main novelty of our Relational Cloud prototype lies in these analysis and adaptation components, specifically:

1. **Partitioning**: To allow workloads to scale across multiple computing nodes, it is important to divide their data into partitions that maximize transaction/query performance. We developed a new graph-based data partitioning algorithm for transaction-oriented workloads that groups data items according to their frequency of co-access within transactions/queries. The idea is to minimize the probability that a given transaction has to access multiple nodes to compute its answer. This component is discussed in Section 2.2. In Section 3, we show how this component automatically derives optimal partitionings for the TPC-C benchmark, and how it tackles the harder problem of a social-network test-case derived from *epinions.com*, with performance gains between 28% and 314% over hash-partitioning.

2. **Live Migration**: One of the key requirements of the cloud is the ability to be *elastic*; in the context of a database service, elasticity means adaptively dedicating resources where they are most needed. This is particularly challenging in a database environment where there are large amounts of data that may need to be moved in order to adapt. Our Live Migration component handles this by attempting to predict when adaptation will be needed before any given node is overloaded, and by partitioning and moving data in small chunks and maintaining the ability to execute transactions while movement occurs. The key ideas behind this approach are described in Section 2.3.

3. **Workload Analysis and Allocation**: To properly co-locate workloads and database instances on machines, it is necessary to analyze and classify their resource requirements. This is the goal of the analysis and allocation component, discussed in Section 2.4. Though we haven't yet built a tool to completely automate this process, we have run experiments that suggest the potential gains from proper storage engine selection and assignment to machines. Experiments show that combining heterogeneous storage engines we can obtain up to almost nine times better performance from the same hardware.

In addition to describing this architecture and components in Section 2, we describe some initial encouraging performance results in Section 3 and discuss related work in Section 4.

2. SYSTEM DESIGN

Our Relational Cloud design is shown in Figure 1. The system runs on commodity servers inside a single data center. This has emerged as the most cost-effective way to provide computing, and can be easily scaled by adding and removing individual servers. Each physical server runs potentially multiple *database instances*, as shown at the bottom of the figure. The database instances may use different storage engines, since specialized engines are often very efficient for specific workloads. Carefully combining workloads and database instances provides an invaluable opportunity to increase efficiency.

Each database is divided into *logical partitions*, by an automatic partitioning engine, as discussed in Section 2.2. These partitions are stored in k-way redundant *replica groups* to guarantee high availability and fault-tolerance. A replica group, shown in the middle of the figure, consists of k database instances each storing a copy of the data of a logical partition. Partitioning of the databases and allocation of replica groups to machines is controlled by the the *workload analyzer*, and is crucial to achieve efficiency, as we discuss in Section 2.4.

Applications communicate with Relational Cloud using a standard interface or a known protocol; the current prototype supports core transactional SQL functionality via JDBC and MySQL interfaces. Incoming SQL statements are sent to the *router*, which analyzes them and consults the metadata database to determine the execution plan. The *distributed transaction system* then distributes the work while ensuring serializability and handling failures.

By constantly monitoring performance and load, the system validates and adjusts partitioning and placement choices on-line. System failures and workload changes require Relational Cloud to evolve partitioning and allocation schemes at run-time. This requires migration of the data across storage engine instances.

2.1 Current Status

We are actively developing our Relational Cloud prototype. Currently, we have implemented the distributed transaction coordinator along with the routing, partitioning and replication components. We support MySQL and HSQLDB storage engines, and have implemented JDBC and MySQL public interfaces. Given a query trace, we can analyze and automatically generate a *good* partitioning for it, and then run distributed transactions against those partitions. Our transaction coordinator supports active fail-over to replicas in the event of a failure. We currently do not have complete implementations of the workload analysis, placement, and live migration components, though we have implemented portions of them so that we can demonstrate the potential performance gains that a fully-implemented Relational Cloud system should achieve.

Much of what we have built so far is infrastructure to allow us to research the best partitioning, placement, and migration schemes. Rather than further describing that infrastructure and its engineering challenges (which we feel is not a particularly novel contribution), in the rest of the paper we focus on our ideas and initial implementations of the partitioning, placement, and migration problems.

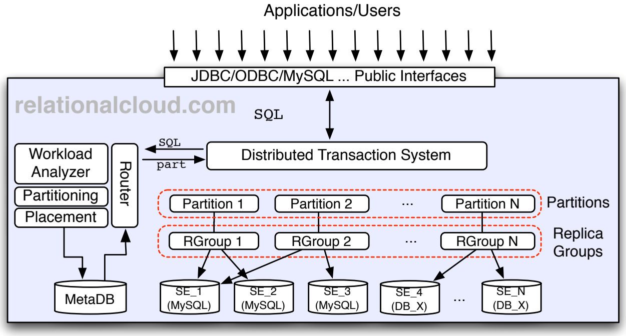

Figure 1: Relational Cloud Architecture.

2.2 Database Partitioning

Partitioning is crucial to enable: (i) scalability beyond a single node and (ii) finer-grained replication, migration and balancing.

Different classes of workloads necessitate different partitioning strategies. For example, OLTP is characterized by short-lived transactions with little internal parallelism, that are best executed in a single location to minimize distributed transaction overhead, while long-running OLAP queries often benefit from the parallelism enabled by distributing the work across many machines (e.g., large sequential scans from disk). We plan to include multiple algorithms targeting different types of workloads in our partitioning engine.

We started by developing a partitioning strategy for OLTP and Web workloads that uses detailed workload execution traces. The traces are processed in two steps: an *agnostic* partitioning phase, and a *justification* phase. The *agnostic* partitioning phase creates a graph with a node for each tuple in the database. For every transaction in the trace, edges are drawn between each tuple that is accessed, forming a clique. An alternative hypergraph formulation is also available, where hyperedges represent individual transactions. We then apply state of the art graph partitioning techniques [15] to find k balanced partitions while minimizing the number of cut edges. Node and edge weights are used to account for skewed workloads or to assign different priorities to transactions. The resulting partitioning minimizes distributed transactions for the workload trace. This phase assigns individual tuples to partitions, this can be represented as look-up tables. However, for naturally partitionable databases a more efficient representation exists.

The goal of the *justification* phase is to find such representation as a set of predicates on the tuple attributes. The system extracts a set of candidate attributes from the predicates used in the trace (e.g., the predicates appearing in the `WHERE` clauses), tests their correlation to the partitioning labels derived by the agnostic partitioning, and selects the highly correlated attributes. The values of this reduced set of attributes are fed into a decision tree algorithm together with the partitioning labels. If the decision tree successfully generalizes the partitioning via few simple rules, we have found a good *justification* for the agnostic partitioning. If no predicate-based explanation is found, the system falls back to look-up tables or bloom filters to represent the partitioning scheme.

The strength of this approach is its independence from schema layout and foreign keys information, which allows us to discover intrinsic correlations hidden in the data. As a consequence, this approach is effective (see experiments in Section 3.1) in partitioning databases containing multiple n-to-n relationships—typical in social-network scenarios. The system is also capable of suggesting replication for read-mostly tables and hash-based partitioning for tables accessed mainly via primary keys.

The primary disadvantage is the difficulty in scaling the graph representation. The naïve approach leads to a graph with N nodes and up to N^2 edges, where N is the cardinality of the database. Unfortunately, most graph partitioning algorithms scale only up to few tens of millions of nodes. For this reason, we devised a series of heuristics that effectively limit the graph size with minimal impact on the quality of the partitioning. Among the most effective heuristics we implemented are: (i) *blanket statement removal*: i.e., the exclusion from the graph of statements that are uncommon and scan large portions of the DB, since they explode the graph size without contributing with much information, (ii) *low relevance filter*: i.e., the removal of tuples that are never or rarely accessed[2], (iii) grouping of tuples with similar access patterns, (iv) sampling (both for tuples and transactions) and (v) summarization. This allows us to represent the workloads of databases with billions of tuples within the scalability limits of graph partitioning algorithms.

The initial validation of this approach on the TPC-C benchmark and a real-world workload shows very encouraging results, as discussed in Section 3.

2.3 Live Migration

The capability to reorganize the resources allocated to a workload, e.g., by moving a workload or a portion of a workload to a new machine, is a key feature that enables elasticity and scalability, two essential features of a cloud-based database service. Although migration is important in other replicated database environments (e.g., when adding a new replica), in a cloud setting it is particularly important for migration to be performed efficiently as it may be used frequently.

While moving data in a network environment is in itself not a difficult problem, supporting *transparent live migration* is hard to achieve. By transparent migration we mean migration that is: (i) transactionally consistent, (ii) fault tolerant, and (iii) does not substantially affect running transactions and perceived performance. Storage engine heterogeneity further complicates the process.

The naïve migration solution involves suspending the service (quiescing running transactions or waiting for them to finish), taking a snapshot of the portion of the database to be moved, moving and loading the data onto the new node, and restarting processing with the new configuration. We believe the performance overhead of this solution is unacceptable, especially when migration is used to adapt to short-lived spikes in load. In fact, significant research has been devoted to this problem, as recently surveyed in [23].

In the Relational Cloud effort we are considering various strategies to improve this naïve approach, including: (i) partitioning the data to be moved into a number of small partitions, and incrementally migrating these smaller partitions, (ii) migrating an existing snapshot/checkpoint and selectively rolling-forward logs, (iii) exploiting existing replicas to serve read-only queries during migration, (iv) prefetching of data to prepare warm stand-by copies.

The above techniques should all reduce the performance overhead of migration. However, we believe it is possible to reduce this overhead to near-zero by using a cache-like approach that works as follows: when a new processing node is added to the pool serving a certain workload, we immediately start routing transactions to it. The new node fetches from the old node the data *as needed* for processing each transaction, caches them in its local storage, and processes reads and writes locally. Over time the new node will accumulate a larger and larger portion of the data and will serve more of the queries/updates locally, effectively reducing the load on the old node. This approach has the advantage that the load

[2] This is possible thanks to the generalization step of the *justification* phase, and similarly by a default partition assignment for tuples not listed in the look-up tables.

on the old node is minimal (i.e., nothing more than the user workload is ever executed on it), and reduces over time. Furthermore, as soon as all the write transactions running on the old node have completed, the new node can exploit multiple read-only replicas of the old node to fetch the data. This strategy requires manipulation of the SQL statements (to fetch data), a careful management of distributed transactions (to guarantee transactional consistency and fault tolerance), and efficient data transfer mechanisms (for speed). We believe all this can be implemented effectively within the architecture we have designed, and is a key goal of our short-term research agenda.

2.4 Adaptive Resource Allocation

Resource allocation is a recurring challenge in the engineering of efficient software systems. In Relational Cloud, the primary resource-related problems include: (i) static and dynamic characterization of workloads, (ii) selection of optimal storage engines, (iii) assignment of workloads to database instances, and (iv) assignment of database instances to physical nodes.

Each of these requires the ability to characterize the demands of workloads and to understand storage engine interactions (e.g., whether two storage engines can co-exist without conflicts on physical node). As we show in Section 3.3, understanding these factors is crucial to efficiently using all available resources. Furthermore, different tenants have service level agreements imposing constraints on performance, replication, and data placement.

The resource allocator must take into consideration all of these concerns—hardware resources, storage engines, user workloads, workload interactions, service level agreements, and replication policies—and produce optimal allocations. We believe this problem can be effectively modeled as a linear programming optimization problem, however deriving faithful cost models for all of the above is an incredibly hard challenge. In practice, we believe that tackling this problem requires both analytical models and on-line dynamic adaptation. The large scale at which we operate exacerbates the challenges, but also enables interesting new opportunities, as we anticipate having many different workloads with different requirements, and a large, heterogeneous collection of hardware and software that will enable fine-grained "packing" of workloads.

We have not begun to seriously address the problems of workload analysis and resource allocation, although they are a key part of our research agenda. Instead, we have focused on measuring the potential benefits of effective resource allocation (in particular, intelligent selection of storage engines and workload placement). As shown in Section 3.3, those benefits are substantial.

3. EXPERIMENTS

In this section, we describe several experiments we have run on our Relational Cloud prototype.

3.1 Partitioning

We validated the partitioning technique introduced in Section 2.2 on two cases studies, an OLTP benchmark derived from TPC-C and a benchmark based on a real dataset from *epinions.com* [17]. These two tests aim at stress-testing two different aspects of our technique. The TPC-C workload is amenable to partitioning as shown in prior literature [25] via manual partitioning. The goal of this test is thus to show that our partitioning scheme is capable of completely automating the partitioning task with results similar to careful manual partitioning, as demonstrated in the context of a scalability test in Section 3.2.

The *epinions.com* experiment aims at challenging our system in a case where no clear "good" partitioning exists, and thus is a way of verifying our effectiveness in discovering intrinsic correlations between data items that are not visible at the schema/query level. The simplified *epinions.com* schema we consider contains four relations: `users`, `items`, `ratings`, `trust`, where the `ratings` relation represents an n-to-n relationship between users and items (capturing user ratings of items), and the `trust` relation represents a n-to-n relationship between pairs of users indicating a binary "trust" value. The data was obtained in [17] via scraping of the *epinions.com* website. We consider three common queries:

Q1. *For logged-in users: given an item provide ratings from the trusted users*

Q2. *Given a user show the list of the users (s)he trusts*

Q3. *For anonymous users: given an item provide the weighted average of all the ratings*

The partitioning is non trivial, since the access queries involve multiple n-to-n relations with opposite requirements on how to group tables and tuples (e.g., Q1 will access a single partition if the data is partitioned by item and ratings and trust are stored with items, while Q2 will access a single-partition if data is partitioned by user and trust is stored with users.)

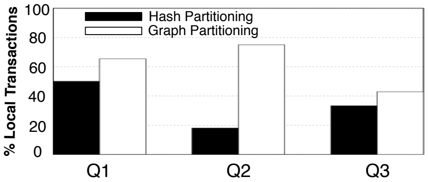

Figure 2: Partitioning performance: the *epinions.com* test

Figure 2 shows the result of executing a uniform mix of 40 million instances of queries Q1–Q3 against two equal-sized partitions generated via hash-partitioning and semi-automatically by our tool.

The hard problem is the relative placement of users and items in order to minimize cross-partition queries/transactions—we fully automate this task. Our system succeeds in capturing the intrinsic correlations between certain groups of users and between certain users and items. In fact, assigning tuples to partitions as our tool suggested (and directing queries to partitions via look-up tables in the *router*) we win significantly against hash-based partitioning with average improvements ranging from 28% to 314%. We also tried standard predicate-based range partitioning, which yielded to results comparable to hash-partitioning.

3.2 Scaling a Database

In order to show how Relational Cloud can scale a workload, we ran an OLTP benchmark derived from TPC-C across a cluster of machines. Each system has two 3.2 GHz Intel Xeon processors, based on Intel's Netburst architecture, and 2 GB of RAM. Each system uses a single 7200 RPM SATA hard drive with 8 MB cache, 4.2 ms average seek time and 120 MB/s maximum data transfer rate, according to the manufacturer's specifications. The systems are interconnected by a single gigabit Ethernet switch. We used MySQL 5.4 as the underlying storage engine.

The workload is based on TPC-C scaled to 10 warehouses. We used the partitioning algorithm described in Section 3.1 to divide the database into 1, 2, and 5 partitions, with results almost indistinguishable from careful manual partitioning/replication. Each partition is hosted on a separate server. These partitions are very close to the same size, and the TPC-C workload is uniform across them. We then generated load at a fixed rate of desired transactions per second, measuring the sustained throughput and the average latency

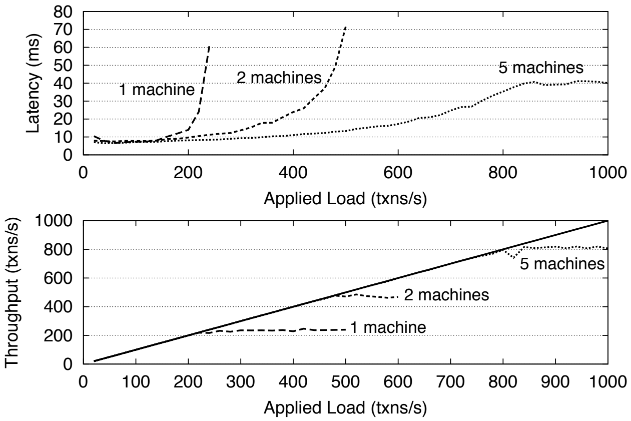

Figure 3: Latency and throughput with increasing TPC-C load

for each request. Latency and throughput are shown in Figure 3.

The throughput shows that a single server is able to achieve approximately 230 transactions per second, and the throughput scales approximately linearly (2× speed-up for 2 servers, 3.5× for 5). This shows that for a workload that can be partitioned cleanly, Relational Cloud can scale with demand by distributing it across more servers. Approximately 6% of the transactions access multiple servers, which shows that the cost of distributed transactions is not limiting our scalability for this experiment.

With very low load, the average latency is minimized with a single database instance. Splitting the workload across two databases increases the average latency, due to the additional latency of distributed transactions. For example, at 5 requests per second, one partition has an average latency of 10 ms, while with two partitions this increases to 14 ms. The latency generally increases linearly with load, until the system approaches saturation. At this point, for 1 and 2 servers, the latency increases sharply. This suggests that average latency may be a useful "leading indicator" to trigger corrective actions, such as redistributing data. Unfortunately, we do not see this pattern for 5 servers, which leads us to believe that there may be other bottlenecks in our experimental evaluation. We are currently investigating latency and other indicators of database overload over a range of workloads.

3.3 Adaptive Resource Allocation

To illustrate the potential gain from effective workload analysis and resource allocation, we compared the performance of heterogeneous storage engine configurations running a fixed workload on a single hardware configuration. The workload simulates a skewed TPC-C workload over 50 warehouses. The target transaction load of our hypothetical customer is 635 TPS with 95% of the load targeting 8 "hot" warehouses. The hardware consists of a single eight-core x64 server with 12 GB memory and a single 7200 RPM SATA hard drive. The results are shown in Figure 4.

In the baseline configuration, the workload is executed by a single MySQL instance given complete system resources, which yields 73 TPS[3]. Even ignoring distributed transactions we would need more at least 9 machines to meet the user requirements. The system is disk-bound, and there is plenty of CPU and RAM that could be used more effectively. In our second configuration, the workload is partitioned using the same strategy as the previous experiment, and the "hot" partitions are stored into an increasing number of HyperSQL[4] main-memory storage engine instances[5]—the limit of 8 instances is due to RAM availability. This allows us to meet the customer requirement of 635 TPS with no additional machines. It is important to notice that the entire database cannot fit in RAM, thus, a pure main-memory approach is not a viable option. We also tested stacking multiple MySQL instances on the same machine. This yielded no improvement, since MySQL is disk limited in this configuration. The performance of each storage instance degrades slightly as more instances are added due to processor and memory bandwidth contention. At the final configuration, we nearly saturate all the machine's resources. Not only is this the highest performance configuration, it will also be very power-efficient. This experiment is designed to show how careful workload analysis and allocation allows to increase performance/efficiency of almost a factor nine, thus, showcasing the potential benefits of a fully developed workload analysis and allocation engine and justifying our research effort in this direction.

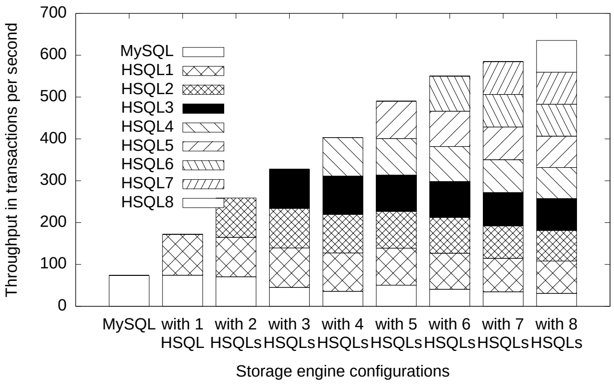

Figure 4: Packing multiple storage engines.

4. RELATED WORK

We group related work into three categories: (i) scalable database services, (ii) partitioning schemes, (iii) workload analysis.

Scalable database services. This is a widely researched area, where substantial research and development efforts have been made by academia as well as commercial and open-source communities. The conflicting goals of providing expressive query languages, consistency, scalability, high-availability, ease of use and low operational costs have been approached from various angles, often significantly favoring one of this requirements over the others. We place ourself in the center of this rich solution space, trying to combine many recent advances in a balanced solution that aims at answering Gassner's call for a cloud-based fully-consistent relational database service for the masses [12]. In contrast, BigTable [5], PNUTS [6], Dynamo [9], S3, and their academic counterparts [4, 16] choose to sacrifice expressive power and/or consistency in favor of extreme scalability. Similarly Helland suggests a way of designing applications to avoid distributed transactions in order to better scale systems [14]. ElasTraS [7] aims at providing a scalable and elastic data store in the cloud by limiting the type of transactions. Recently Hyder promises to scale a database across multiple servers by relying on a very high performance shared disk built from flash [3]. Commercial cloud-based relational services are also starting to appear, as Amazon RDS, Microsoft SQL Azure, and offerings from startups like Vertica. These services are a step in the right direction, however they are effectively offering existing

[3] The performance difference from the previous experiments are well-justified due to different test-machines, different TPC-C implementations (trace-based vs clients), skewed/uniform workloads, and different scale factors.

[4] See `http://hsqldb.org/`

[5] Notice that the underlying replication guarantees persistency.

relational DBMSs virtualized in a public cloud, and thus lack the elasticity and scalability features we are seeking to achieve.

Partitioning Schemes. While we are designing our own workload partitioning algorithms, we plan to make use of several state of the art algorithms for graph partitioning and clustering. Interesting approaches include, but are not limited to: classical work on physical design and partitioning [27], recent works on workload-aware partitioning [21], graph partitioning algorithms [15], the comprehensive study on clustering of Tsangaris and Naughton [26], and recent efforts to scale social networks [20, 2]. Our approach differs from the above by limiting distributed transactions in a multi-node system through the use of novel schema-agnostic partitioning algorithms that simplify attribute selection and handles n-to-n relations.

Workload Analysis. As already mentioned, the workload analysis engine in our system is still in the early stages. We are investigating several research directions inspired by recent advances on workload optimization [18, 13, 10], allocation and prediction [11], automatization of virtual machine configuration [1, 24, 22], and sophisticated caching strategies [8]. The problem we face combines many of the above, plus storage-engine heterogeneity, inter-engine interactions, and multi-tenancy in ways that call for novel solutions.

5. CONCLUSIONS

Cloud-based database services offer the potential to address several serious limitations of existing relational databases related to scalability, ease of use, and provisioning in the face of peak load. In this position paper, we explored the main requirements and technical challenges to make the vision of a database service into a reality and introduced the design and initial evaluation of our system.

Relational Cloud has been designed from scratch to adapt to the peculiarities of the cloud-computing environment. By exploiting multiple dedicated storage engines, providing high-availability via replication, automating workload partitioning and balancing, automating data migration, and supporting distributed transactions we showed that our initial prototype is able to effectively scale OLTP workloads and achieve high-efficiency.

We see these results as an encouraging first step towards providing transactional relational databases as a service, and plan to develop and evaluate our architecture into a full-fledged, open-source cloud-based database platform.

6. REFERENCES

[1] A. Aboulnaga, K. Salem, A. A. Soror, U. F. Minhas, P. Kokosielis, and S. Kamath. Deploying database appliances in the cloud. *IEEE Data Eng. Bull.*, 32(1):13–20, 2009.

[2] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[3] P. Bernstein and C. Reid. Scaling out without partitioning: A novel transactional record manager for shared raw flash. In *HPTS*, October 2009.

[4] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, 2008.

[5] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[6] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2), 2008.

[7] S. Das, D. Agrawal, and A. E. Abbadi. ElasTraS: An elastic transactional data store in the cloud. HotCloud, 2009.

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

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. *SIGOPS*, 2007.

[10] S. Elnaffar and P. Martin. The Psychic–Skeptic Prediction framework for effective monitoring of DBMS workloads. *Data & Knowledge Engineering*, 68(4):393–414, 2009.

[11] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, 2009.

[12] P. Gassner. BigDB.com. In *HPTS*, October 2009.

[13] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper. An integrated approach to resource pool management: Policies, efficiency and quality metrics. In *DSN*, 2008.

[14] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *Conference on Innovative Database Research (CIDR)*, pages 132–141, January 2007.

[15] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), 1998.

[16] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1), 2009.

[17] P. Massa and P. Avesani. Controversial users demand local trust metrics: an experimental study on epinions.com community. In *AAAI'05*, 2005.

[18] N. W. Paton, M. A. T. Aragão, K. Lee, A. A. A. Fernandes, and R. Sakellariou. Optimizing utility in cloud computing through autonomic workload execution. *IEEE Data Eng. Bull.*, 32(1):51–58, 2009.

[19] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[20] J. M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez. Scaling online social networks without pains. NetDB, 2009.

[21] T. Scholl, B. Bauer, J. Müller, B. Gufler, A. Reiser, and A. Kemper. Workload-aware data partitioning in community-driven data grids. In *EDBT*, 2009.

[22] P. Shivam, A. Demberel, P. Gunda, D. E. Irwin, L. E. Grit, A. R. Yumerefendi, S. Babu, and J. S. Chase. Automated and on-demand provisioning of virtual machines for database applications. In *SIGMOD*, 2007.

[23] G. H. Sockut and I. B. R. Online reorganization of databases. *ACM Comput. Surv.*, pages 1–136, 2009.

[24] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, 2008.

[25] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[26] M. M. Tsangaris and J. F. Naughton. On the performance of object clustering techniques. In *SIGMOD*, 1992.

[27] D. C. Zilio. Physical database design decision algorithms and concurrent reorganization for parallel database systems. In *PhD thesis*, 1998.