Relational Cloud: A Database-as-a-Service for the Cloud

Carlo Curino
curino@mit.edu

Evan P. C. Jones
evanj@mit.edu

Raluca Ada Popa
ralucap@mit.edu

Nirmesh Malviya
nirmesh@csail.mit.edu

Eugene Wu
eugenewu@mit.edu

Sam Madden
madden@csail.mit.edu

Hari Balakrishnan
hari@csail.mit.edu

Nickolai Zeldovich
nickolai@csail.mit.edu

ABSTRACT

This paper introduces a new transactional "database-as-a-service" (DBaaS) called **Relational Cloud**. A DBaaS promises to move much of the operational burden of provisioning, configuration, scaling, performance tuning, backup, privacy, and access control from the database users to the service operator, offering lower overall costs to users. Early DBaaS efforts include Amazon RDS and Microsoft SQL Azure, which are promising in terms of establishing the market need for such a service, but which do not address three important challenges: *efficient multi-tenancy*, *elastic scalability*, and *database privacy*. We argue that these three challenges must be overcome before outsourcing database software and management becomes attractive to many users, and cost-effective for service providers. The key technical features of Relational Cloud include: (1) a workload-aware approach to multi-tenancy that identifies the workloads that can be co-located on a database server, achieving higher consolidation and better performance than existing approaches; (2) the use of a graph-based data partitioning algorithm to achieve near-linear elastic scale-out even for complex transactional workloads; and (3) an adjustable security scheme that enables SQL queries to run over encrypted data, including ordering operations, aggregates, and joins. An underlying theme in the design of the components of Relational Cloud is the notion of *workload awareness*: by monitoring query patterns and data accesses, the system obtains information useful for various optimization and security functions, reducing the configuration effort for users and operators.

1. INTRODUCTION

Relational database management systems (DBMSs) are an integral and indispensable component in most computing environments today, and their importance is unlikely to diminish. With the advent of hosted cloud computing and storage, the opportunity to offer a DBMS as an outsourced service is gaining momentum, as witnessed by Amazon's RDS and Microsoft's SQL Azure (see §7). Such a **database-as-a-service (DBaaS)** is attractive for two reasons. First, due to economies of scale, the hardware and energy costs incurred by users are likely to be much lower when they are paying for a share of a service rather than running everything themselves. Second, the costs incurred in a well-designed DBaaS will be proportional to actual usage ("pay-per-use")—this applies to both software licensing and administrative costs. The latter are often a significant expense because of the specialized expertise required to extract good performance from commodity DBMSs. By centralizing and automating many database management tasks, a DBaaS can substantially reduce operational costs *and* perform well.

From the viewpoint of the operator of a DBaaS, by taking advantage of the lack of correlation between workloads of different applications, the service can be run using far fewer machines than if each workload was independently provisioned for its peak.

This paper describes the challenges and requirements of a large-scale, multi-node DBaaS, and presents the design principles and implementation status of Relational Cloud, a DBaaS we are building at MIT (see `http://relationalcloud.com`). Relational Cloud is appropriate for a single organization with many individual databases deployed in a "private" cloud, or as a service offered via "public" cloud infrastructure to multiple organizations. In both cases, our vision is that users should have access to all the features of a SQL relational DBMS, without worrying about provisioning the hardware resources, configuring software, achieving desired security, providing access control and data privacy, and tuning performance. All these functions are outsourced to the DBaaS.

There are three challenges that drive the design of Relational Cloud: efficient multi-tenancy to minimize the hardware footprint required for a given (or predicted) workload, elastic scale-out to handle growing workloads, and database privacy.

Efficient multi-tenancy. Given a set of databases and workloads, what is the best way to serve them from a given set of machines? The goal is to minimize the number of machines required, while meeting application-level query performance goals. To achieve this, our system must understand the resource requirements of individual workloads, how they combine when co-located on one machine, and how to take advantage of the temporal variations of each workload to maximize hardware utilization while avoiding overcommitment.

One approach to this problem would be to use virtual machines (VMs); a typical design would pack each individual DB instance into a VM and multiple VMs on a single physical machine. However, our experiments show that such a "DB-in-VM" approach requires $2\times$ to $3\times$ more machines to consolidate the same number of workloads and that for a fixed level of consolidation delivers $6\times$ to $12\times$ less performance than the approach we advocate. The reason is that each VM contains a separate copy of the OS and database, and each database has its own buffer pool, forces its own log to disk, etc. Instead, our approach uses a single database server on each machine, which hosts multiple logical databases. Relational Cloud periodically determines which databases should be placed on which machines using a novel non-linear optimization formulation, combined with a cost model that estimates the combined resource utilization of multiple databases running on a machine. The design of Relational Cloud also includes a lightweight mechanism to perform *live migration* of databases between machines.

Elastic scalability. A good DBaaS must support database and workloads of different sizes. The challenge arise when a database workload exceeds the capacity of a single machine. A DBaaS must therefore support *scale-out*, where the responsibility for query processing (and the corresponding data) is *partitioned* amongst multiple nodes to achieve higher throughput. But what is the best way

to partition databases for scale-out? The answer depends on the way in which transactions and data items relate to one another. In Relational Cloud, we use a recently developed *workload-aware partitioner* [5], which uses graph partitioning to automatically analyze complex query workloads and map data items to nodes to minimize the number of multi-node transactions/statements. Statements and transactions spanning multiple nodes incur significant overhead, and are the main limiting factor to linear scalability in practice. Our approach makes few assumptions on the data or queries, and works well even for skewed workloads or when the data exhibits complex many-to-many relationships.

Privacy. A significant barrier to deploying databases in the cloud is the perceived lack of privacy, which in turn reduces the degree of trust users are willing to place in the system. If clients were to encrypt all the data stored in the DBaaS, then the privacy concerns would largely be eliminated. The question then is, how can the DBaaS execute queries over the encrypted data? In Relational Cloud, we have developed *CryptDB*, a set of techniques designed to provide privacy (e.g., to prevent administrators from seeing a user's data) with an acceptable impact on performance (only a 22.5% reduction in throughput on TPC-C in our preliminary experiments). Database administrators can continue to manage and tune the databases, and users are guaranteed data privacy. The key notion is that of *adjustable security*: CryptDB employs different encryption levels for different types of data, based on the types of queries that users run. Queries are evaluated on the encrypted data, and sent back to the client for final decryption; no query processing runs on the client.

A unifying theme in our approach to these three big challenges is *workload-awareness*. Our main design principle is to monitor the actual query patterns and data accesses, and then employ mechanisms that use these observations to perform various optimization and security functions.

2. SYSTEM DESIGN

Relational Cloud uses existing unmodified DBMS engines as the back-end query processing and storage nodes. Each back-end node runs a single *database server*. The set of back-end machines can change dynamically in response to load. Each *tenant* of the system—which we define as a billable entity (a distinct user with a set of applications, a business unit, or a company)—can load one or more databases. A database has one or more tables, and an associated *workload*, defined as the set of queries and transactions issued to it (the set may not be known until run-time). Relational Cloud does *not* mix the data of two different tenants into a common database or table (unlike [1]), but databases belonging to different tenants will usually run within the same database server.

Applications communicate with Relational Cloud using a standard connectivity layer such as JDBC. They communicate with the Relational Cloud front-end using a special driver that ensures their data is kept private (e.g., cannot be read by the database administrator) —this is described in more detail below. When the front-end receives SQL statements from clients, it consults the *router*, which analyzes each SQL statement and uses its metadata to determine the execution nodes and plan. The front-end coordinates multi-node transactions, produces a distributed execution plan, and handles fail-over. It also provides a degree of performance isolation by controlling the rate at which queries from different tenants are dispatched.

The front-end monitors the access patterns induced by the workloads and the load on the database servers. Relational Cloud uses this information to periodically determine the best way to: (1) *partition* each database into one or more pieces, producing multiple partitions when the load on a database exceeds the capacity of a single machine (§3), (2) *place* the database partitions on the back-end machines to both minimize the number of machines and balance load, *migrate* the partitions as needed without causing downtime, and *replicate* the data for availability (§4), and (3) *secure* the data and process the queries so that they can run on untrusted back-ends over encrypted data (§5). The Relational Cloud system architecture is shown in Figure 1, which depicts these functions and demarcates the trusted and untrusted regions.

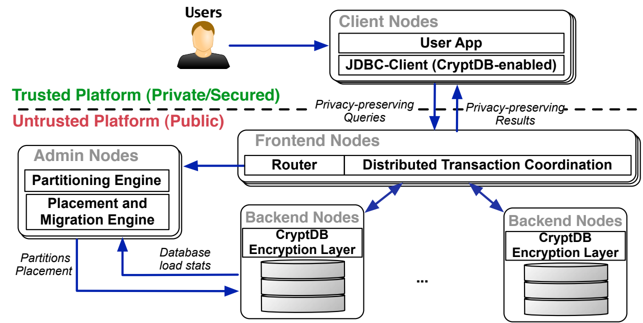

Figure 1: Relational Cloud Architecture.

Applications communicate with the Relational Cloud front-end using a CryptDB-enabled driver on the client, which encrypts and decrypts user data and rewrites queries to guarantee privacy. On the back-end nodes, CryptDB exploits a combination of server-side cryptography and user-defined functions (UDFs) to enable efficient SQL processing—particularly ordering, aggregates, and joins, which are all more challenging than simple selections and projections—over encrypted data.

Current status: We have developed the various components of Relational Cloud and are in the process of integrating them into a single coherent system, prior to offering it as a service on a public cloud. We have implemented the distributed transaction coordinator along with the routing, partitioning, replication, and CryptDB components. Our transaction coordinator supports both MySQL and Postgres back-ends, and have implemented a JDBC public interface. Given a query trace, we can analyze and automatically generate a good partitioning for it, and then run distributed transactions against those partitions. The transaction coordinator supports active fail-over to replicas in the event of a failure. We have developed a placement and migration engine that monitors database server statistics, OS statistics, and hardware loads, and uses historic statistics to predict the combined load placed by multiple workloads. It uses a non-linear, integer programming solver to optimally allocate partitions to servers. We are currently implementing live migration.

Other papers (either published [5] or in preparation) detail the individual components, which are of independent interest. This paper focuses on the Relational Cloud system and on how the components address the challenges of running a large-scale DBaaS, rather than on the more mundane engineering details of the DBaaS implementation or on the detailed design and performance of the components. (That said, in later sections we present the key performance results for the main components of Relational Cloud to show that the integrated system is close to being operational.) At the CIDR conference, we propose to demonstrate Relational Cloud.

3. DATABASE PARTITIONING

Relational Cloud uses database partitioning for two purposes: (1) to scale a single database to multiple nodes, useful when the load exceeds the capacity of a single machine, and (2) to enable more granular placement and load balance on the back-end machines compared to placing entire databases.

The current partitioning strategy is well-suited to OLTP and Web workloads, but the principles generalize to other workloads as well (such as OLAP). OLTP/Web workloads are characterized by short-lived transactions/queries with little internal parallelism. The way to scale these workloads is to partition the data in a way that minimizes the number of multi-node transactions (i.e., most transactions should complete by touching data on only one node), and then place the different partitions on different nodes. The goal is to minimize the number of cross-node distributed transactions, which incur overhead both because of the extra work done on each node and because of the increase in the time spent holding locks at the back-ends.

Relational Cloud uses a workload-aware partitioning strategy. The front-end has a component that periodically analyzes query execution traces to identify sets of tuples that are accessed together within individual transactions. The algorithm represents the execution trace as a graph. Each node represents a tuple (or collection of tuples) and an edge is drawn between any two nodes whose tuples are touched within a single transaction. The weight on an edge reflects how often such pair-wise accesses occur in a workload. Relational Cloud uses graph partitioning [13] to find ℓ balanced logical partitions, while minimizing the total weight of the cut edges. This minimization corresponds to find a partitioning of the database tuples that minimizes the number of distributed transactions.

The output of the partitioner is an assignment of individual tuples to logical partitions. Relational Cloud now has to come up with a succinct representation of these partitions, because the front-end's router needs a compact way to determine where to dispatch a given SQL statement. Relational Cloud solves this problem by finding a set of predicates on the tuple attributes. It is natural to formulate this problem as a classification problem, where we are given a set of tuples (the tuple attributes are features), and a partition label for each tuple (the classification attribute). The system extracts a set of candidate attributes from the predicates used in the trace. The attribute values are fed into a decision tree algorithm together with the partitioning labels. If the decision tree successfully generalizes the partitioning with few simple predicates, a good *explanation* for the graph partitioning is found. If no predicate-based explanation is found (e.g., if thousands of predicates are generated), the system falls back to lookup tables to represent the partitioning scheme.

The strength of this approach is its independence from schema layout and foreign key information, which allows it to discover intrinsic correlations hidden in the data. As a consequence, this approach is effective in partitioning databases containing multiple many-to-many relationships—typical in social-network scenarios—and in handling skewed workloads [5].

The main practical difficulty we encountered was in scaling the graph representation. The naïve approach leads to a graph with N nodes and up to N^2 edges for an N-tuple database, which is untenable because existing graph partitioning implementations scale only to a few tens of millions of nodes. For this reason, we devised a series of heuristics that effectively limit the size of the graph. The two most useful heuristics used in Relational Cloud are: (1) *blanket statement removal*, i.e., the exclusion from the graph occasional statements that scan large portions of the database and (2) *sampling* tuples and transactions.

4. PLACEMENT AND MIGRATION

Resource allocation is a major challenge when designing a scalable, multi-tenant service like Relational Cloud. Problems include: (i) monitoring the resource requirements of each workload, (ii) predicting the load multiple workloads will generate when run together on a server, (iii) assigning workloads to physical servers, and (iv) migrating them between physical nodes.

In Relational Cloud, a new database and workload are placed arbitrarily on some set of nodes for applications, while at the same time set up in a staging area where they run on dedicated hardware. During this time, the system monitors their resource consumption in the staging area (and for the live version). The resulting time-dependent resource profile is used to predict how this workload will interact with the others currently running in the service, and whether the workload needs to be partitioned. If a workload needs to be partitioned, it is split using the algorithms described in the previous section. After that, an allocation algorithm is run to place the each workload or partition onto existing servers, together with other partitions and workloads.

We call the monitoring and consolidation engine we developed for this purpose *Kairos*; a complete paper on Kairos is currently under submission. It takes as input an existing (non-consolidated) collection of workloads, and a set of target physical machines on which to consolidate those workload, and performs the aforementioned analysis and placement tasks. Its key components are:

1. *Resource Monitor*: Through an automated statistics collection process, the resource monitor captures a number of DBMS and OS statistics from a running database. One monitoring challenge is estimating the RAM required by a workload, since a standalone DBMS will tend to fill the entire buffer pool with pages, even if many of those pages aren't actively in use. To precisely measure working set size, Kairos slowly grows and repeatedly accesses a temporary probe table, while monitoring amount of disk activity on the system. Once the probe table begins to evict tuples in the working set, load on the disk will increase as those pages have to be read back into memory to answer queries. We have found that this approach provides an accurate, low-overhead way of measuring the true RAM requirements of a workload.

2. *Combined Load Predictor:* We developed a model of CPU, RAM, and disk that allows Kairos to predict the combined resource requirements when multiple workloads are consolidated onto a single physical server. The many non-linearities of disk and RAM makes this task difficult. In particular, for disk I/O, we built a tool that creates a hardware-specific model of a given DBMS configuration, allowing us to predict how arbitrary OLTP/Web workloads will perform on that configuration. The accuracy of this model at predicting the combined disk requirements of multiple workloads is up to $30\times$ better than simply assuming that disk I/O combines additively. The reason is that two combined workloads perform many fewer I/Os than the sum of their individual I/Os: when combined, workloads share a single log, and can both benefit from group commit. Moreover, database systems perform a substantial amount of non-essential I/O during idle periods (e.g., flushing dirty pages to decrease recovery times)—in a combined workload, this activity can be curtailed without a substantial performance penalty.

3. *Consolidation Engine:* Finally, Kairos uses non-linear optimization techniques to place database partitions on back-end nodes to: (1) minimize the number of machines required to support a given workload mix, and (2) balance load across the back-end machines, while not exceeding machine capacities.

In addition to this placement functionality, another important feature of Relational Cloud is the capability to relocate database partitions across physical nodes. This relocation allows for scheduled maintenance and administration tasks, as well as to respond to load changes that entail the addition (or removal) of back-end machines. Relational Cloud aims to provide *live migration*, where data is moved between back-end nodes without causing downtime or adversely reducing performance. We are currently developing and testing a cache-like approach, where a new node becomes the new *master* for a portion of the data. Data is lazily fetched from the

old master as needed to support queries. In-flight transactions are redirected to the new master without being quiesced or killed.

5. PRIVACY

This section outlines *CryptDB*, the sub-system of Relational Cloud that guarantees the privacy of stored data by encrypting all tuples. The key challenge is executing SQL queries over the resulting encrypted data, and doing so efficiently. For example, a SQL query may ask for records from an employees table with a specific employee name; records whose salary field is greater than a given value; records joined with another table's rows, such as the employee's position; or even more complex queries, such as the average salary of employees whose position requires travel. Simply encrypting the entire database, or encrypting each record separately, will not allow the back-end DBMS to answer these kinds of queries. In addition, we would like a design that will allow DBAs (who operate Relational Cloud) to perform tuning tasks without having any visibility into the actual stored data.

Approach. The key idea in our approach is a notion we call *adjustable security*. We observe that there are many cryptographic techniques that we can build on to execute SQL queries, including randomized encryption (RND) and deterministic encryption (DET), as well as more recently developed order-preserving encryption (OPE) and homomorphic encryption (HOM). RND provides maximum privacy, such as indistinguishability under an adaptive chosen-plaintext attack without access to the key. However, RND does not allow any computation to be efficiently performed on the plaintext. DET provides a weaker privacy guarantee, because it allows a server to check plaintexts for equality by checking for equality of ciphertexts. OPE is even more relaxed in that it enables inequality checks and sorting operations, but has the nice property that the distribution of ciphertext is independent from the encrypted data values and also pseudorandom. Finally, HOM enables operations over encrypted data; in our case, we will mainly use additions and multiplications, which can be done efficiently.

Design. To implement adjustable security, our idea is to encrypt each value of each row independently into an *onion*: each value in the table is dressed in layers of increasingly stronger encryption, as shown in Figure 2. Each integer value is stored three times: twice encrypted as an onion to allow queries and once encrypted with homomorphic encryption for integers; each string type is stored once, encrypted in an onion that allows equalities and word searches and has an associated token allowing inequalities.

CryptDB starts the database out with all data encrypted with the most private scheme, RND. The JDBC client, shown in Figure 1, has access to the keys for all onion layers of every ciphertext stored on the server (by computing them based on a single master key). When the JDBC client driver receives SQL queries from the application, it computes the *onion keys* needed by the server to decrypt certain columns to the maximum privacy level that will allow the query execute on the server (such as DET for equality predicates). The security level dynamically adapts based on the queries that applications make to the server. We expect the database to converge to a certain security level when the application does not issue any more queries with new structures (only with different constants).

To simplify the management of onion keys, CryptDB encrypts all data items in a column using the same set of keys. Each layer of the onion has a different key (different from any other column), except for the lowest layer allowing joins (to allow meaningful comparisons of ciphertexts between two different columns as part of a join). The encryption algorithms are symmetric; in order for the server to remove a layer, the server must receive the symmetric

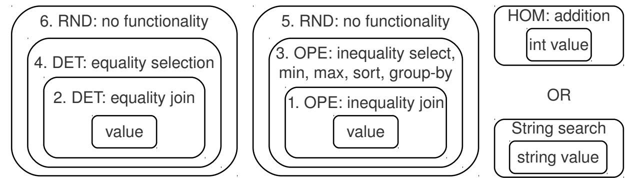

Figure 2: Onion layers of encryption.

onion key for that layer from the JDBC client. Once the server receives the key to decrypt an onion to a lower layer, it starts writing newly-decrypted values to disk, as different rows are accessed or queried. Once the entire column has been decrypted, the original onion ciphertext is discarded, since inner onion layers can support a superset of queries compared to outer layers.

For example, performing joins between tables requires the client to send keys to the server, which decrypts the joined columns to a layer that allows joins. There are two layers of DET and OPE encryption in the onion shown in Figure 2, corresponding to ciphertexts that can be used for comparisons for a single column (e.g., selection on equality to a given value with DET, selection based on comparison with OPE, etc.), and ciphertexts that can be used to join multiple columns together (i.e., using the same key). The database server then performs joins on the resulting ciphertexts as usual.

The key factor in the performance of CryptDB is ciphertext expansion. After the client issues a few queries, the server removes any unneeded onion layers of encryption, and from then on, it does not perform any more cryptographic operations. The server's only overhead is thus working with expanded tuples. If the ciphertext and plaintext are of equal length, most server operations, such as index lookups or scans, will take the same amount of time to compute. For DET, plaintext and ciphertext are equal in length, whereas for OPE, the ciphertext is double in length.

An example. To illustrate CryptDB's design, consider a TPC-C workload. Initially each column in the database is separately encrypted in several layers of encryption, with RND being the outer layer. Suppose the application issues the query `SELECT i_price, ... FROM item WHERE i_id=`N. The JDBC client will decrypt the `i_id` column to DET level 4 (Figure 2) by sending the appropriate decryption key to the server. Once that column is decrypted, the client will issue a SQL query with a `WHERE` clause that matches the DET-level `i_id` field to the DET-encrypted ciphertext of N. The query will return RND-encrypted ciphertexts to the JDBC client, which will decrypt them for the application. If the application's query requires order comparisons (e.g., looking for products with fewer than M items in stock), the JDBC client must similarly decrypt the onion to OPE level 3, and send an OPE ciphertext of the value M.

Suppose the application issues a join query, such as `SELECT c_discount, w_tax, ... FROM customer, warehouse WHERE w_id=c_w_id AND c_id=`N. To perform the join on the server, the JDBC client needs to decrypt the `w_id` and `c_w_id` columns to DET level 2 because the encryption should be deterministic not only within a column, but also across columns. The server can now perform the join on the resulting ciphertexts. Additionally, the JDBC client needs to decrypt `c_id` column to DET level 4, and send the DET-encrypted value N to the server for the other part of the `WHERE` clause.

Finally, CryptDB uses HOM encryption for server-side aggregates. For example, if a client asks `SELECT SUM(ol_amount) FROM order_line WHERE ol_o_id=`N, the server would need the keys to adjust the encryption of the `ol_amount` field to HOM, so that it can homomorphically sum up the encrypted `ol_amount` values, computing the total order amounts.

Table 1: Consolidation ratios for real-world datasets

Dataset	Input # Servers	Consolidated # Servers	Consolidation Ratio
TIG-CSAIL	25	2	12.5:1
Wikia	34	2	17:1
Wikipedia	40	6	6.6:1
Second Life	98	16	6.125:1

6. EXPERIMENTS

In this section, we describe several experiments we have run to evaluate Relational Cloud.

Consolidation/Multi-tenancy. We begin by investigating how much opportunity there is for consolidation in real-world database applications. We obtained the load statistics for about 200 servers from three data centers hosting the production database servers of Wikia.com, Wikipedia, and Second Life, and the load statistics from a cluster of machines providing shared services at MIT CSAIL.

Table 1 reports the consolidation levels we were able to achieve using the workload analysis and placement algorithms presented in Section 4. Here we used traces gathered over a 3-week duration, and found an allocation of databases to servers that Relational Cloud predicts will cause no server to experience more than 90% of peak load. The resulting ratios range from between 6:1 to 17:1 consolidation, demonstrating the significant reductions in capital and administrative expenses by adopting these techniques. One reason we are able to achieve such good consolidation is that our methods exploit the statistical independence and uncorrelated load spikes in the workloads.

In our second experiment, we compared the Relational Cloud approach of running multiple databases inside one DBMS to running each database in its own DBMS instance. Figure 3 (left) compares the performance of multiple databases consolidated inside a single DBMS that uses the entire machine, to the same server running one DBMS per database, in this case all in a single OS. We measure the maximum number of TPC-C instances than can run concurrently while providing a certain level of transaction throughput. Running a single DBMS allows 1.9–3.3× more database instances at a given throughput level. In Figure 3 (right) we compare the same single DBMS to multiple DBMSs, each running in a separate VM with its own OS. We measure the TPC-C throughput when there are 20 database instances on the physical machine—overall, a single DBMS instance achieves approximately 6× greater throughput for a uniform load and 12× when we skew the load (i.e., 50% of the requests are directed to one of the 20 databases).

The key reason for these results is that a single DBMS is much better at coordinating the access to resources than the OS or the VM hypervisor, enabling higher consolidation ratios on the same hardware. In particular, multiple databases in one DBMS share a single log and can more easily adjust their use of the shared buffer pool than in the multiple DBMS case where there are harder resource boundaries.

Scalability. We now measure how well the partitioning algorithm divides a database into independent partitions and how throughput scales as the database is spread across machines.

In this experiment we run TPC-C with a variable number of warehouses (from 16 to 128) and show what happens when database is partitioned and placed by Relational Cloud on 1 to 8 servers. The partitioner automatically splits the database by warehouse, placing 16 warehouses per server, and replicates the item table (which is never updated). We have manually verified that this partitioning is optimal. We measure the maximum sustained transaction throughput, which ranges from 131 transactions per second (TPS) with 16 warehouses on 1 machine up to 1007 TPS with 128 warehouses spread across 8 machines, representing a 7.7× speedup.

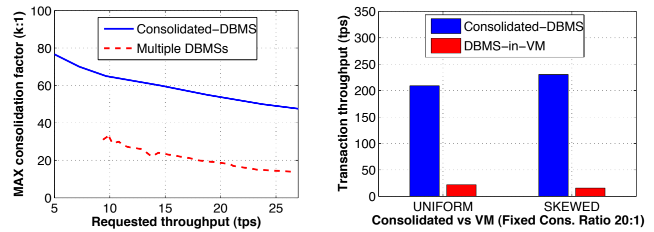

Figure 3: Multiplexing efficiency for TPC-C workloads

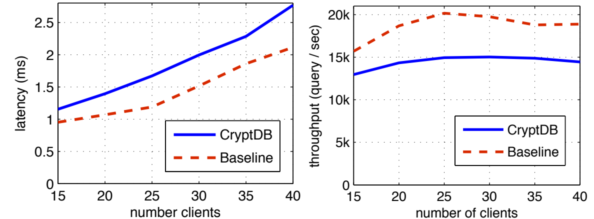

Figure 5: Impact of privacy on latency and throughput

We also measured the latency impact of our transaction coordinator on TPC-C, issuing queries to a single database with and without our system in place. On average, the Relational Cloud front-end adds 0.5 ms of additional latency per SQL statement (which for TPC-C adds up to 15 ms per transaction), resulting in a drop in throughput of about 12% from 149 TPS to 131 TPS.

The cost of privacy. CryptDB introduces additional latency on both the clients (for rewriting queries and encrypting and decrypting payloads), and on the server (due to the enlargement of values as a result of encryption.) We measured the time to process 100,000 statements (selects/updates) from a trace of TPC-C and recorded an average per statement overhead of 25.6 ms on the client side. We measure the effect that this additional latency is likely to cause in the next section. The server-side overhead is shown in Figure 5; the dashed line represents performance (latency–left, throughput–right) without CryptDB, and the solid line shows performance with CryptDB. Overall throughput drops by an average of 22.5%, which we believe will be an acceptable and tolerable performance degradation given the powerful privacy guarantees that are being provided.

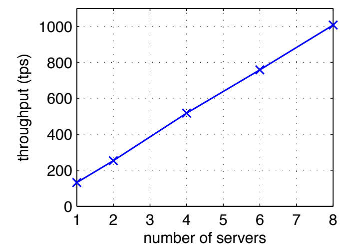

Figure 4: Scaling TPC-C.

The impact of latency.

In our final experiment, we measured the impact that additional latency between database clients and servers introduces on query throughput. This metric is relevant because in a Relational Cloud deployment, it may be valuable to run the database on a service provider like AWS (to provide a pool of machine for elastic scalability) with the application running in an internal data center across a wide-area network. Additionally, our privacy techniques depend on a (possibly remote) trusted node to perform query encoding and result decryption.

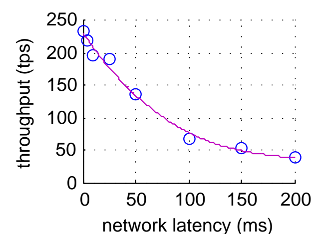

Figure 6: Impact of latency

We again ran TPC-C, this time with 128 warehouses and 512 client terminals (with no wait time). Figure 6 shows how aggregate throughput varies with increasing round-trip latency between the application and the DB server (we artificially introduced latency using a Linux kernel configuration parameter on the client machines.) We note that with latencies up to 20 ms, the drop in throughput is only about 12%, which is comparable to the latency of 10–20 ms we observe between AWS's east coast data center and MIT. The principal cause of this latency degradation is that locks are held for longer in the clients, increasing conflicts and decreasing throughput. Since TPC-C is a relatively high contention workload, it is likely that real-world workloads will experience lower throughput reductions.

Combining the results from the previous experiments, we expect an overall throughput reduction of about 40% when running CryptDB, our transaction coordinator, and a remote application, all at once. However, due to the linear scalability achieved via partitioning, we can compensate for this overhead using additional servers. As a result of the high consolidation ratios we measured on real applications, we still expect significant reduction in the overall hardware footprint, on the order of 3.5:1 to 10:1.

7. RELATED WORK

Scalable database services. Commercial cloud-based relational services like Amazon RDS and Microsoft SQL Azure have begun to appear, validating the market need. However, existing offerings are severely limited, supporting only limited consolidation (often simply based on VMs), lacking support for scalability beyond a single node, and doing nothing to provide any data privacy or ability to process queries over encrypted data. Some recent approaches [17, 18] try to leverage VM technologies, but our experiments show that these are significantly inferior in performance and consolidation to Relational Cloud's DBMS-aware DBaaS design.

Multi-tenancy. There have been several efforts to provide extreme levels of multi-tenancy [1, 12], aiming to consolidate tens of thousands of nearly inactive databases onto a single server, especially when those databases have identical or similar schemas. The key challenge of this prior work has been on overcoming DBMSs' limitations at dealing with extremely large numbers of tables and/or columns. These efforts are complementary to our work; we target heterogeneous workloads that do not share any data or schemas, and which impose significant load to the underlying DBMS in aggregate or even on their own—hence our focus on profiling and placement.

Scalability. Scalable database systems are a popular area for research and commercial activity. Approaches include NoSQL systems [3, 4, 2, 14], which sacrifice a fair amount of expressive power and/or consistency in favor of extreme scalability, and SQL-based systems that limit the type of transactions allowed [11, 7] or exploit novel thread-to-transaction assignment techniques [16]. We differ from these in that we aim to preserve consistency and expressivity, achieving scalability via workload-aware partitioning. Our partitioning approach differs from prior work in that most prior work has focused on OLAP workloads and declustering [15, 20, 8].

Untrusted Storage and Computation. Theoretical work on homomorphic encryption [9] provides a solution to computing on encrypted data, but is too expensive to be used in practice. Systems solutions [10, 6, 19] have been proposed, but, relative to our solution, offer much weaker (at best) and compromised security guarantees, require significant client-side query processing and bandwidth consumption, lack core functionality (e.g., joins), or require significant changes to the DBMS.

8. CONCLUSION

We introduced Relational Cloud, a scalable relational database-as-a-service for cloud computing environments. Relational Cloud overcomes three significant challenges: *efficient multi-tenancy*, *elastic scalability*, and *database privacy*. For multi-tenancy, we developed a novel resource estimation and non-linear optimization-based consolidation technique. For scalability, we use a graph-based partitioning method to spread large databases across many machines. For privacy, we developed the notion of adjustable privacy and showed how using different levels of encryption layered as an "onion" can enable SQL queries to be processed over encrypted data. The key insight here is for the client to provide only the minimum decryption capabilities required by any given query. Based on our performance results, we believe that the Relational Cloud vision can be made a reality, and we look forward to demonstrating an integrated prototype at CIDR 2011.

9. REFERENCES

[1] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: Schema-mapping techniques. In *SIGMOD*, 2008.

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

[3] 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.

[4] 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.

[5] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A Workload-Driven Approach to Database Replication and Partitioning. In *VLDB*, 2010.

[6] E. Damiani, S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing Confidentiality and Efficiency in Untrusted Relational DBMS. *CCS*, 2003.

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

[8] R. Freeman. *Oracle Database 11g New Features*. McGraw-Hill, Inc., New York, NY, USA, 2008.

[9] R. Gennaro, C. Gentry, and B. Parno. Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. *STOC*, 2010.

[10] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. *ACM SIGMOD*, 2002.

[11] P. Helland. Life beyond distributed transactions: An apostate's opinion. In *CIDR*, 2007.

[12] M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting database applications as a service. In *ICDE*, 2009.

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

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

[15] D.-R. Liu and S. Shekhar. Partitioning similarity graphs: A framework for declustering problems. *ISJ*, 21, 1996.

[16] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-Oriented Transaction Execution. *PVLDB*, 2010.

[17] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35(1), 2010.

[18] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic resource allocation for database servers running on virtual storage. In *FAST*, 2009.

[19] B. Thompson, S. Haber, W. G. Horne, T. Sander, and D. Yao. Privacy-Preserving Computation and Verification of Aggregate Queries on Outsourced Databases. *HPL-2009-119*, 2009.

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