A Scalable Data Platform for a Large Number of Small Applications

Fan Yang[*]
Google Inc.
Mountain View, CA, USA
yangf@cs.cornell.edu

Jayavel Shanmugasundaram
Yahoo! Research
Santa Clara, CA, USA
jaishan@yahoo-inc.com

Ramana Yerneni
Yahoo! Inc.
Santa Clara, CA, USA
yerneni@yahoo-inc.com

ABSTRACT

As a growing number of websites open up their APIs to external application developers (e.g., Facebook, Yahoo! Widgets, Google Gadgets), these websites are facing an intriguing scalability problem: while each user-generated application is by itself quite small (in terms of size and throughput requirements), there are many many such applications. Unfortunately, existing data-management solutions are not designed to handle this form of scalability in a cost-effective, manageable and/or flexible manner. For instance, large installations of commercial database systems such as Oracle, DB2 and SQL Server are usually very expensive and difficult to manage. At the other extreme, low-cost hosted data-management solutions such as Amazon's SimpleDB do not support sophisticated data-manipulation primitives such as joins that are necessary for developing most Web applications. To address this issue, we explore a new point in the design space whereby we use commodity hardware and free software (MySQL) to scale to a large number of applications while still supporting full SQL functionality, transactional guarantees, high availability and Service Level Agreements (SLAs). We do so by exploiting the key property that each application is "small" and can fit in a single machine (which can possibly be shared with other applications). Using this property, we design replication strategies, data migration techniques and load balancing operations that automate the tasks that would otherwise contribute to the operational and management complexity of dealing with a large number of applications. Our experiments based on the TPC-W benchmark suggest that the proposed system can scale to a large number of small applications.

1. INTRODUCTION

Recently, many Web sites have opened up their services to allow third-party developers to create applications for social groups and communities, e.g., Facebook [20], Google Gadgets [11], Yahoo Widgets [31]. Such community applications often have relatively small data sizes and throughput requirements as compared to large enterprise applications that typically deal with tera bytes of business data. Mostly, their database sizes range from tens to thousands of megabytes and their workload ranges from tens to hundreds of concurrent user sessions. However, since there can be a large number (say, tens of thousands) of such applications in a large social network, building a data platform for this setting is not an easy task. Specifically, the combined data size and workload of the set of applications is quite large, of the order of peta bytes of data and millions of concurrent user sessions.

Unfortunately, existing data-management solutions are ill suited to handle a large number of small applications for one or more of the following reasons:

Cost: Commercial database systems (e.g., DB2 Enterprise [4], Oracle Real Application Cluster [9], and SQL Server [26]) have leveraged decades of research in the data-management community and have achieved impressive scalability with respect to database sizes and throughput. However, this scalability comes at a large monetary cost. Large installations of such software are expensive and require complex management, which further adds to the monetary cost. Open-source alteratives to commercial database systems are less expensive (free!), but do not scale well because they do not have the sophisticated scalability features that are built into most commercial systems. In fact, part of the reason that commercial systems charge a premium for large installations is that the technology for scaling is complex and difficult to get right.

Lack of sophisticated data management features: There has been a lot of recent interest in peer-to-peer (P2P) technologies such as Distributed Hash Tables (DHTs) [23, 24, 28] and ordered tables [1, 10, 14]. Such systems are designed to scale using commodity hardware and software (low cost) to a large number of nodes, thereby achieving excellent scalability and throughput performance. However, these systems only support very simple data-manipulation operations which essentially translate to equality and range lookups on a single table. Such a simple set of operations is inadequate for most Web applications. Further, such systems lack sophisticated transaction support, which is again crucial for Web applications. While there have been some notable attempts to incorporate more sophisticated data-manipulation

[*]Work done while at Yahoo! Research.

4th *Biennial Conference on Innovative Data Systems Research (CIDR)* January 4-7, 2009, Asilomar, California, USA.

features such as joins in P2P systems [1, 13], such systems still lack sophisticated transaction support such as ACID transactions, which are difficult to achieve at such scale.

There are also other emerging data platforms for Web applications such as BigTable [8], PNUTS [12] and SimpleDB [27]. These platforms scale to a large number of data operations but achieve this scale by trading off consistency and query-processing capabilities. In particular, none of these systems support ACID transactions across multiple records and tables, and they all restrict the kinds of queries applications can issue. In contrast, the set of applications we are targeting, small but full-functional applications on the Web, need consistency and rich query-processing capabilities.

Lack of Multi-Tenancy Support: Most scalable data-management systems are designed to scale one or a few large databases, and they do not provide explicit multi-tenancy support for multiple applications running on shared resources. While it is relatively easy to control the Service Level Agremeements (SLAs) for one or a few databases by explicitly adding resources, it becomes a complex manageability problem when we try to meet the SLAs for many thousands of applications using *shared* resources (using dedicated resources is not a cost-effective option because of the large number of applications).

Our goal in this paper is to design a low-cost, full-featured, multi-tenancy-capable data-management solution that can scale to tens of thousands of applications. While this is a difficult problem in general, we exploit the fact that the applications are "small" and can comfortably fit (with possibly other applications) in a single machine without violating SLAs. This property enables us to design a low-cost solution that meets our design objectives.

The architecture of our system is a set of database clusters. Each database cluster consists of a set of database machines (typically tens of such machines) that are managed by a fault-tolerant controller. Each database machine is based on commodity hardware and runs an instance of an off-the-shelf single-node DBMS. Each DBMS instance hosts one or more applications, and processes queries issued by the controllers without contacting any other DBMS instance. This architecture is low-cost because it only uses a single-node DBMS (we use free MySQL in our prototype) as the building block. It also supports sophisticated data-manipulation functionality, including joins and ACID transactions, because queries and updates are processed using a full-featured DBMS instance.

Given the above architecture, there are two main technical challenges that arise. The first is that of fault-tolerance: when a database machine fails, we need to ensure that all the applications running on that machine can be recovered without violating the ACID semantics of transactions. One of the technical contributions of this paper is a flexible and provably correct way of managing database replicas using commodity single-node DBMS software, while still ensuring ACID semantics. The second challenge is ensuring that the SLAs of applications are met, especially when multiple application databases reside on the same machine. Another contribution of this paper is formalizing the notion of database SLAs, mapping these to measurable parameters on commodity DBMS software, and formulating and solving an optimization problem that minimizes the required number of resources.

We have implemented and evaluated a prototype system based on the above architecture, deployed it on a cluster of machines, and evaluated the system using multiple TPC-W database applications. Our preliminary results show that the proposed techniques are scalable and efficient.

2. SYSTEM ARCHITECTURE

The proposed system provides the illusion of one large centralized fault-tolerant DBMS that supports the following API:

1. Create a database along with an associated SLA
2. Connect to a previously created database using JDBC and perform the set of operations supported by JDBC interface, including complex SQL queries and ACID transactions. Such connections can be established by application servers or other middle-ware.

The main restriction that the system imposes is that the size and SLA requirements for each database should fit in a single physical machine. All other aspects of data management such as fault-tolerance, resource management, and scaling are automatically handled by the system.

Figure 1 shows the system architecture. The system consists of machines in multiple geographically-distributed colos (a colo is a set of machines in one physical location). A client database is (asynchronously) replicated across more than one colo to provide disaster recovery. The colos are coordinated by a fault-tolerant system controller, which routes client database connection requests to an appropriate colo, based on various factors such as the replication configuration for the database, the load and status of the colo, and the geographical proximity of the client and the colo.

Each colo contains one or more machine clusters. Each database is hosted and managed by a single cluster within the colo. The clusters are coordinated by a fault-tolerant colo controller, which routes client database connection requests (which were in turn routed by the system controller) to the appropriate cluster that hosts the database. In addition, the colo controller manages a pool of free machines and adds them to clusters as needed, based on the workload and other resource requirements.

Each cluster contains a small number (tens) of machines, typically on the same server rack, connected through a high-speed network. Each machine runs an off-the-shelf single-node DBMS — MySQL in our implementation — and each client database is mapped to two or more machines within the cluster (for fault-tolerance). Further, each machine in the cluster can host one or more client databases, depending on the resource requirements and SLAs of the individual databases. The machines in the cluster are coordinated by a fault-tolerant cluster controller, which performs three main tasks. First, the cluster controller manages client database connections (routed by the colo controller) and ensures that the multiple replicas of the client database in the cluster are always in synch. Second, the cluster controller deals with machine failures and database recovery within a cluster so that client database connections are not affected by such failures (note that database connections are made to a cluster controller and not to a specific machine in the cluster). Finally, the cluster controller deals with database placement

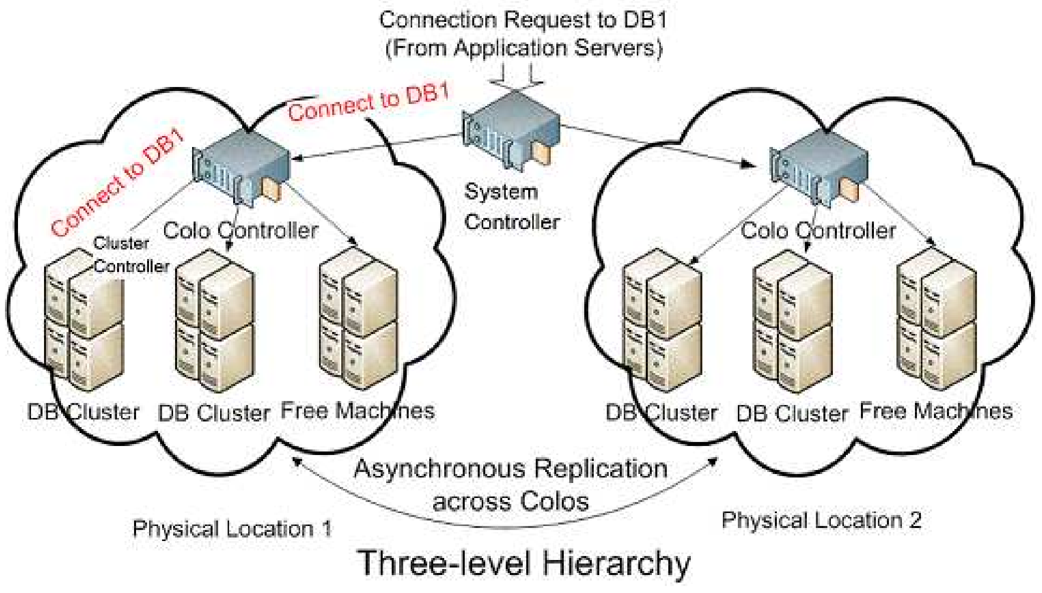

Figure 1: System Architecture

and migration within a cluster so that the SLAs for the individual databases are satisfied.

As mentioned above, the system controller, the colo controller and the cluster controller need to be fault-tolerant. Since the system controller and the colo controller do not have any connection state because they are only involved in the initial database connection set up by the application, a light-weight hot-standby configuration is sufficient to ensure fault-tolerance. Ensuring the fault-tolerance of the cluster controller is more complex because it is effectively the connection manager for the clients, and the failure of the cluster controller machines can lead to the loss of the database connections. The cluster controller is configured to run as a process pair in two machines to provide fault tolerance with respect to a machine failure. When the primary cluster-controller machine fails, the client applications that have a connection managed by the failed cluster controller need to re-establish the database connection with the backup cluster controller that takes over and resumes work. The failure of the cluster controller during a transaction commit is properly handled through the process-pair mechanism, as the backup keeps track of the primary cluster controller's state with respect to committing transactions and cleans up the transactions in transit as part of its take-over processing.

Given the above system architecture, there are two main types of failures that need to be handled: (1) Machine failures within a colo. This type of failure typically is in the form of power failure of a machine and/or disk failure in a machine; (2) Colo failures. This is mostly due to disaster situations and hence, not very common. In situation (1), the proposed system provides strong ACID guarantees by using synchronous replication within a cluster. In situation (2), the system provides weaker guarantees by using asynchronous replication in order to ensure low latency and high scalability.

The main technical focus of this paper is on the design, implementation and evaluation of the cluster controller, focusing specifically on the management of machine failures (Section 3) and the management of database SLAs (Section 4) within a cluster.

3. FAILURE MANAGEMENT IN A CLUSTER

As mentioned above, our goal is to ensure ACID semantics of database transactions even in the presence of machine failures within a cluster. At first glance, it may appear that a simple solution to this problem is to use the synchronous database replication feature in an off-the-shelf database system. However, this solution does not meet our design objectives for one or more of the following reasons. First, commercial database solutions for synchronous replication are expensive, mainly because they focus on the case of replicating one large database, which is complex and difficult to get right. Second, all free database systems that we are aware of have severe limitiations on synchronous replication, again because they focus on the case where a single physical machine is not sufficient to meet the database SLA. For instance, MySQL synchronous replication requires the entire database to fit in the combined main-memory sizes of the replicas. Finally, none of the synchronous replication solutions — commercial or free — have hooks to explicitly manage load balancing in order to meet database SLAs.

To address the above issues, we propose a simple replication architecture that uses only single-node DBMSs, which are coordinated by the cluster controller using the standard 2-phase commit (2PC) and read-one write-all replication protocols [5] (all single-node DBMSs such as MySQL and other commercial databases support the 2PC API).

3.1 Replication Architecture

The cluster controller maintains a map of databases to machines, and each database is hosted in two or more machines in the cluster. The cluster controller also manages all database connections to the client, and further, maintains a database connection to each machine in the cluster. When a client performs a read operation on a database connection, the controller sends the read operation to one of the machines in the cluster that host the database, and returns the result of the read to the client. When a client performs a write operation on a database connection, the controller sends the write operation to all the machines in the cluster that host that database, and returns the result of one of the write operations to the client. When a client performs a commit operation on a database transaction (and there was at least one write operation performed as part of the transaction), the controller invokes the 2PC commit protocol, and commits the transaction if the protocol succeeds, and aborts the transaction otherwise. Abort operations are handled similarly.

The above architecture gives the cluster controller some flexibility as to where to route the read operations for a database transaction. There are three options: (1) All read operations for a database (regardless of the specific transaction they belong to) are routed to the same physical machine, (2) All read operations for a transaction are routed to the same physical machine, but read operations from different transactions can be routed to different machines, and (3) Read operations belonging to the same transaction can be routed to different physical machines.

Which of the above three options should the controller choose? It turns out that the answer is not obvious and depends on various factors. First, in terms of performance, option (1) is the best and option (3) is the worst because of cache and resource usage, as explained in more detail in the experimental section. Second, in terms of flexibility for load-balancing, option (3) is the best and option (1) is the worst because the controller can route operations at a finer granularity in options (2) and (3) to deal with spikes in the workload. Finally, and perhaps most surprisingly (to us), it turns out that options (2) and (3) may violate the ACID semantics of transactions by producing non-serializable results unless we are careful about how write operations are processed in the cluster controller! (this problem does not arise in option (1)).

To elaborate on the last point, note that the cluster controller only needs to obtain the result of a write operation from one of the machines before it can return the result to the client. If the cluster controller is aggressive in this return, i.e., it returns as soon as it obtains the result from one machine (and asynchronously keeps track of whether the writes in the other machines failed, in which case subsequent operations of the transaction are aborted), then it turns out that using options (2) and (3) above can lead to non-serializable schedules! However, if the cluster controller is conservative, and waits for all write operations on the database machines to return successfully before it returns to the client, then serializability is guaranteed even for options (2) and (3). Option (1) guarantees serializability under both aggressive and conservative settings for the cluster controller. Table 3.1 summarizes the above discussion.

We now give some examples and proof sketches in support of the above claims.

Aggressive cluster controller with Options 2 and 3 is not serializable. The main reason for this issue is the following: although we can achieve global one-copy serializability [5] with strict two-phase locking (strict 2PL) and 2PC for distributed transactions, most modern database systems implement many 2PC optimizations. For instance, they usually release read locks after the PREPARE action and before the COMMIT action for distributed transactions. In such situations, serializability may be compromised in the presence of replicated objects (note that ensuring serializability across replicated objects is a slightly stronger requirement than ensuring serializability across independent objects).

	Conservative Controller	Aggressive Controller
Option 1	Serializable	Serializable
Option 2	Serializable	Not Serializable
Option 3	Serializable	Not Serializable

Table 1: Serializability for different read and write options

The following example illustrates how the use of an aggressive cluster controller can result in non-serializable execution. Consider two concurrently executing transactions T_1 and T_2 that are processed by an aggresive cluster controller on two database replicas residing on Machines 1 and 2. T_1 executes the following sequence of operations:

$$r_1(x), w_1(y), p_1, c_1$$

where $r_i(z)$ and $w_i(z)$ denote a read and write operation, respectively, by transaction i on object z, and p_i and c_i denote the prepare and commit operations, respectively, of transaction i. T_2 executes the following sequence of operations:

$$r_2(y), w_2(x), p_2, c_2$$

Under Options 2 and 3 using an agressive cluster controller, the following execution history is possible on Machines 1 and 2, which violates global serializability.

$$r_1(x), w_1(y), p_1, w_2(x), p_2, c_2, c_1.\ \text{Machine 1.}$$

$$r_2(y), w_2(x), p_2, w_1(y), p_1, c_2, c_1.\ \text{Machine 2.}$$

This schedule corresponds to the case when the write operation $w_1(y)$ finishes first on Machine 1, and the transaction T_1 enters the PREPARE phase while the write operation is still executing on Machine 2. Similarly, $w_2(x)$ finishes first on Machine 2, and the transaction T_2 enters the PREPARE phase while the write operation is still executing on Machine 1. The above schedule is only locally serializable but not globally serializable.

Note that the above schedule is not possible under Option 1, because all read operations for the database can only be executed on one machine. Further, even if we modify the schedule and have $r_2(y)$ execute on Machine 1 instead of Machine 2, then since p_1 does not release the write lock on y, transaction T_2 can not be serialized after transaction T_1, while committing before T_1. Instead, either T_2 is scheduled before T_1 or there will be a distributed deadlock. More generally, we can prove that Option 1 always results in a globally serializable execution (below).

Aggressive cluster controller with Option 1 is serializable. Before we sketch a proof, we first review some notation [5]. A serialization graph is a directed graph (V, E) where V is the set of transactions and $(T_i, T_j) \in E$ if and only if transactions T_i and T_j have conflicting operations o_i and o_j, and o_i is scheduled to execute before o_j. Two operations are said to conflict if they operate on the same data item and at least one of them is a write. We use $T_i \rightarrow T_j$ to represent an edge (T_i, T_j) in the serialization graph. We use $o_i < o_j$ to denote that operation o_i was scheduled to execute before operation o_j.

Note that with a read-one-write-all policy, guaranteeing global one-copy serializability is equivalent to showing that the global serialization graph is acyclic [5], which is what we prove below.

Theorem 1: An aggressive cluster controller with Option 1 produces an acyclic global serialization graph.
Proof sketch: Assume to the contrary that there is a cycle in the global serialization graph S. Let the cycle be $T_1 \rightarrow T_2....T_n \rightarrow T_1$. Since each transaction operates on a set of objects that belong to a single database, all the transactions $T_1...T_n$ operate on the same database. At the primary site for the database, the schedule contains all the transactions $T_1..T_n$ and they form an acyclic serialization graph S', because each site ensures local serializability. Therefore, the cycle in the global SG must have at least one edge $T_i \rightarrow T_j$ from a non-primary site that is not already present at the primary site. Since all conflicts at non-primary sites are due to write-write conflicts (under Option 1), we must have $w_i < w_j$. Since the 2PC protocol will not release write locks until the time of commit, we have $c_i < c_j$ on the non-primary site. Since w_i and w_j also conflict on the primary site, if $T_j \rightarrow T_i$, then we must have $c_j < c_i$, which is not possible due to the atomicity of commit of 2PC. So we must have $T_i \rightarrow T_j$ on the primary site — a contradiction.

Conservative cluster controller with Options 1, 2, and 3 is serializable. The proof depends on the fact that commercial databases use Strict 2PL along with 2PC.

Theorem 2: A conservative cluster controller with Options 1, 2 or 3 produces an acyclic global serialization graph.
Proof sketch: Assume to the contrary that there is a cycle in the global serialization graph S. Let the cycle be $T_1 \rightarrow T_2....T_n \rightarrow T_1$. For each edge $T_i \rightarrow T_j$ in the cycle, we must have a corresponding pair of conflicting operations o_i, o_j such that $o_i < o_j$ on some site s. Since the sites implement Strict 2PL and 2PC, we must have $p_i < o_j$ in site s. Since the controller is conservative, it only issues a p_i to a site if *all* the read/write operations by T_i have completed on all relevant sites. Hence, all read/write operations of T_i must chronologically precede o_j on site s. Since there is a cycle $T_1 \rightarrow T_2....T_n \rightarrow T_1$, this implies that all read/write operations of T_1 must chronologically precede a read/write operation o_1 of T_1 in some site — a contradiction.

3.2 Managing Failures

When a machine fails, the cluster controller continues to process client database requests using the available machines. The cluster controller also initiates a background database replication process to create additional replicas of databases that were hosted on the failed machine (so that fault-tolerance is not compromised). The key technical challenge lies in designing the database replication process so that it can create new database replicas that are transactionally consistent with the existing databases, using existing DBMS tools and with minimal downtime to the database.

The database replication process works as follows. We use an off-the-shelf database copy tool that is available with virtually all DBMSs (e.g., mysqldump in MySQL) to copy a database to a new machine. The copy tool typically allows databases to be copied at the granularity of a database or a table in a database. During the copy, the tool obtains a read lock on the database/table, copies over the contents, and releases the lock at the end of the copy.

Note, however, that we cannot rely solely on the locks held by the database copy tool and the client transaction to ensure the transactional consistency of replicas. To see why, consider the case where machine A contains the only copy of a database table T, and the copy tool is copying the table to another machine B. If at this time, there is a client write operation issued on table T, then the controller will route the write operation to machine A (which has the only copy of T at this point), and this operation will try to acquire a write lock on table T, and will block on the read lock acquired by the copy tool. When the copy finally completes, the write operation will succeed on machine A, but the contents of the table T will not be consistent on machines A and B (A will have the update and B will not). Thus, in order to maintain transactional consistency of replicas during the copy, we need some additional coordination to be performed by the cluster controller.

Algorithm 1 shows what the controller does during the database copy. If the client operation is a read operation, it is executed on one of the available machines, excluding the machine being copied to. If the client operation is a write operation on a table that has already been copied, then it is executed on all available machines, including the machine copied to. If the client operation is a write operation on a table that is currently being copied, then it is rejected (this handles the issue in the example above). Finally, if the client operation is a write operation on a table that has not yet been copied, then it is executed on all available machines, excluding the machine being copied to.

Algorithm 1 Controller Algorithm During Database Copy

```
 1: INPUT: $o$, $d$ // Operation $o$ on database $d$
 2: Let $M$ be the set of available machines that host $d$
 3: Let $m'(\notin M)$ be the machine that $d$ is being copied to
 4: Let $T$ be the set of tables in $d$ that have been copied to
    $m'$
 5: Let $t'$ be the table currently being copied to $m'$
 6: if $o$ is a read operation then
 7:     Execute $o$ on some $m \in M$
 8: else if $o$ is a write operation on a table $t \in T$ then
 9:     Execute $o$ on all $m \in M \cup \{m'\}$
10: else if $o$ is a write operation on table $t'$ then
11:     Reject $o$ and abort transaction
12: else
13:     Execute $o$ on all $m \in M$
14: end if
```

The correctness of Algorithm 1 (i.e., providing a guarantee of one-copy serializability) relies on the fact that a SQL update statement can only update a single table. The algorithm does reject some operations over tables being copied over, and this ties into the SLA discussion (next section); we also quantify this effect in the experimental section.

We now sketch a proof of correctness of the algorithm.

Theorem 3: Algorithm 1 ensures one-copy serializability.
Proof sketch: In the absence of replica creation, one-copy serializability is guaranteed by read-one-write-all replication as before. In the presence of replica creation, there are two aspects that we need to prove: (1) we need to prove that during the creation of a new replica, existing transactions are one-copy serializable with respect to the original replicas, and (2) we need to prove that at the time the new replica in added to the original set of replicas, none of the correctness invariants of one-copy serializability are violated.

Proof sketch for (1): From the point of view of the database containing only the original set of replicas, and transactions executing on those replicas, the copying of a table to produce a new replica can be viewed as a read-only transaction that accesses exactly one object — the table being copied (note that the copy tool acquires a single read lock on the entire table, and hence it can be viewed as reading a single object). Since a read-only transaction that only accesses a single object cannot introduce any new cycles in the serialization graph, and cannot modify the state of any of the original replicas, one-copy serializablity is still ensured over the original set of replicas.

Proof sketch for (2): For each table in the database, a consistent snapshot is copied over because no writes are in progress during the time the table is copied over. After the copying over is completed, the table in the new replica is also kept up to date with the writes on the table in the original set of replicas using the read-one-write-all protocol. Therefore, when the new replica is added to the set of replicas for the database (after copying over all the tables) each table in the new replica has the same state as in any of the original replicas. Furthermore, all future writes will be processed by the read-one-write-all scheme using the full set of replicas, which ensures one-copy serializability as before.

4. ENFORCING DATABASE SLAS

The Service Level Agreement (SLA) associated with a database specifies the availability and performance requirements of that database. Given the SLAs for many databases, the problem is to allocate databases to the *minimum* number of machines so that all database SLAs are satisfied. In this section, we briefly discuss the formal model we have developed to represent the SLAs of databases and how to allocate the databases to machines in a way that satisfies the SLAs.

4.1 Problem Definition

We first define the notion of SLAs for databases. Each database has an SLA consisting of the following two requirements:

1. The minimum throughput (measured as transactions per second) over a time period T.
2. The maximum fraction of proactively rejected transactions over a time period T. Proactively rejected transactions are those that are rejected due to machine failures, and do not include transactions that fail due to reasons that are inherent to the application, such as deadlocks.

The first metric is a standard database-throughput requirement. Let $r[j]$ represent the resource requirements for a database replica to meet the throughput SLA of database j. Resources in this context are specified as multi-dimensional vectors representing CPU cycles, main memory size, disk size, and disk bandwidth. Let $R[i]$ be the resources available in machine i. Then, the sum of $r[j]$s for all database replicas hosted on a machine i should not exceed $R[i]$.

The second metric is an availability requirement, measured as the fraction of the total number of transactions that are proactively rejected. Let *machine_failure_rate* be the number of times a machine hosting database j fails over time period T, *recovery_time(j)* be the time needed to copy over database j during recovery, *write_mix(j)* be the fraction of update transactions in the workload of database j, and *reallocation_rate(j)* be the number of times a replica of database j is moved from one machine to another during time period T due to system maintenance and reorganization other than recovery. The availability requirement for the database leads to the following constraint:

(*machine_failure_rate(j)* + *reallocation_rate(j)*) * (*recovery_time(j)* / T) * *write_mix(j)* < fraction of rejected transactions specified for database j

4.2 SLA-Based Placement of Databases

When a new database is created, it is first allocated to a free machine in the cluster to observe the resource requirements needed to maintain its SLA. We currently consider three types of resources: CPU, memory and disk I/O. After the observational period, we need to allocate the database to shared machines in such a way that the SLAs are maintained for the new database as well as the existing databases. More formally, let C denote the set of machines, D denote the set of databases, and M represent the allocation matrix for the databases and the machines. When a new database n arrives, we need to compute a new allocation matrix M' for the set of databases $D' = D \cup \{n\}$ and a set of machines C' that is a superset of C such that the size of C' is minimized while still maintaining the SLAs for all databases in D'.

In our implementation, we restrict M and M' to only differ in the allocation of replicas of the new database, i.e., existing databases are not reallocated. This restricted problem is equivalent to the multi-dimensional bin-packing problem [17], which is NP-hard. Consequently, we use a simple yet effective online algorithm — First-Fit [15] — that solves the problem approximately. A similar greedy algorithm is used when a machine fails and its databases need to be replicated to new machines. Developing a non-greedy algorithm that reallocates existing *and* new databases is much more challenging, and is left for future work.

Algorithm 2 shows the First-Fit algorithm adapted for our purpose. The algorithm allocates machines for the m replicas of the new database. For each replica, it finds the first available machine that can host the replica without violating resource constraints. Each replica is allocated to a different machine. If there are any replicas that cannot be allocated to existing machines, each of them is hosted on a separate new machine obtained from the pool of free machines.

5. EXPERIMENTAL EVALUATION

In the experimental evaluation, we focus on a single cluster in the system for two reasons: (1) the technical focus of this paper is on a single cluster, and (2) the system controller and the colo controller are not scaling bottle-necks

Algorithm 2 Adding a New Database to the System

```
1: INPUT: $M, d, n$
2: M is the set of available machines. d is the new database that needs to be hosted. n is the number of replicas we want to create for d
3: for Machine $m$ in $M$ do
4:   if $\sum_{\text{d' hosted on m}} r[d'] + r[d] < R[m]$ then
5:     Allocate a replica of $d$ to $m$
6:     $n = n - 1$
7:     if n == 0 then
8:       break;
9:     end if
10:  end if
11: end for
12: if $n > 0$ then
13:   Add $n$ new machines to the cluster and allocate one replica of $d$ to each new machine
14: end if
```

because they are mainly responsible for connection routing, and are not in the critical path for query evaluation.

Experimental Setup: Our experimental cluster consists of 10 machines running the FreeBSD 6 operating sytem. Each machine has two 2.80GHz Intel(R) Xeon(TM) CPUs, 4GB RAM and runs MySQL 5 configured with a 2GB buffer pool. All the machines reside on the same server rack. We used three different TPC-W benchmark workloads — browsing mix, shopping mix, and ordering mix — in our evaluation. While we varied the size of each database in our experiments, the total database size was 300GB without replication and 600GB with replication (i.e., 2 replicas per database). We also bypassed the application servers and only focused on the database operations.

Synchronous Replication: Figures 2, 3, and 4 show the results of the three read options described in Section 3.1 in comparison with the no replication case. As shown, Option 1 (sending all read operations for a database, regardless of the transaction, to the same replica) has the best performance, and is within 5 to 25% of the no-replication option. The next best option is Option 2 (sending all read operations for a transaction to the same replica), and Option 3 (sending read operations within a transaction to different replicas) has the worst performance. The reason for the difference in performance is that Option 1 has the best cache usage across database reads, while Option 3 has the worst cache usage. Note that Option 3 still has other advantages compared to the other options, such as better load balancing. We also evaluated the deadlock characteristics of the three options, as shown in Figures 5, 6, and 7, but there was no significant difference in the number of deadlocks for the different options.

Failure Recovery: This set of experiments was performed using an individual database size of 200MB, and using Option 1 for querying. Figure 8 shows the number of rejected transactions per database when we induce a single machine failure and initiate the database recovery process. The number of recovery threads (i.e., concurrent database copy processes) is shown in the x-axis. As shown, the number of rejected operations per database is significantly higher for

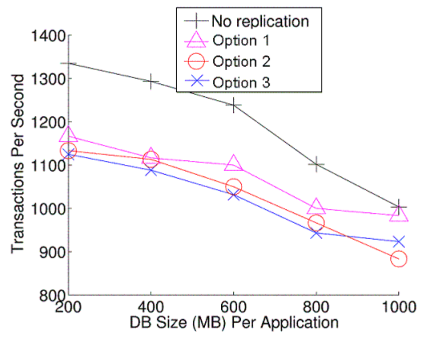

Figure 2: Throughput with Synchronous Replication for Shopping Mix

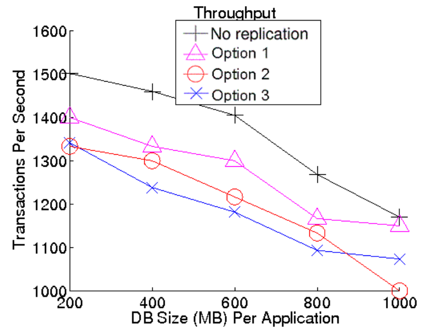

Figure 3: Throughput with Synchronous Replication for Browsing Mix

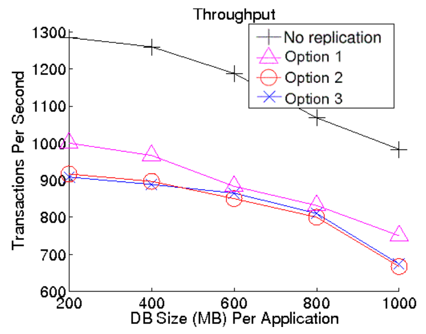

Figure 4: Throughput with Synchronous Replication for Ordering Mix

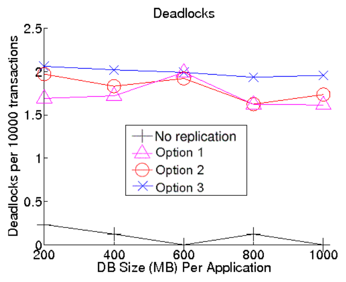

Figure 5: Deadlock Rate for Different Database Sizes with Shopping Mix.

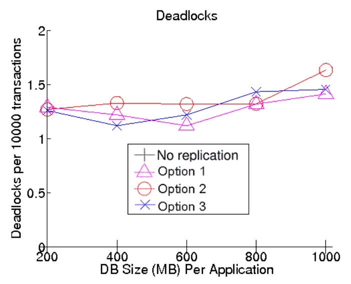

Figure 6: Deadlock Rate for Different Database Sizes with Browsing Mix.

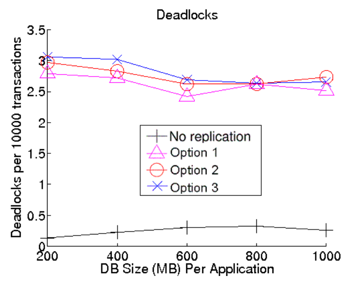

Figure 7: Deadlock Rate for Different Database Sizes with Ordering Mix.

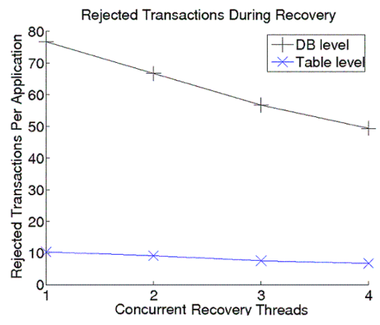

Figure 8: Rejected Transactions during Recovery

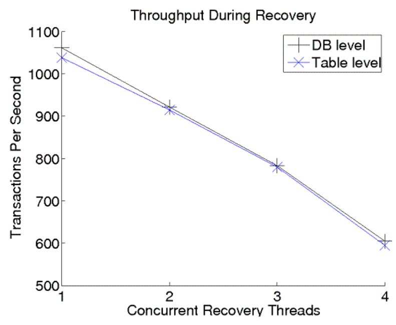

Figure 9: Throughput during Recovery

Skew Factor	0.4	0.8	1.2	1.6	2.0
Average Size (MB)	531	451	398	361	310
Average Throughput (TPS)	3.75	2.29	1.44	0.59	0.29
# of Machine Used	9	6	5	4	4
Optimal Solution	9	6	4	4	4

Table 2: SLA experimental settings and results

database-level copying as opposed to table-level copying because of the lower concurrency in the former approach. But surprisingly, as shown in Figure 9, the throughput of the two approaches is about the same. One possible explanation for this behavior is that the table-level migration approach results in wasted work due to future aborts of transaction operations. In our experiments, it took about 2 minutes to create a new replica of a 200MB database.

SLA Based Database Placement: In this set of experiments, we varied the skew in database sizes and database throughput requirements. Specifically, the database size was drawn from a zipfian distribution with range 200-1000MB, and the throughput was drawn from a zipfian distribution with range 0.1-10 transactions per second. The zipfian skew factors for both parameters were varied from 0.4-2. Table 2 summarizes the results of the experiments and shows the minimum number of machines required to host the databases given our online-allocation method. For comparison, the optimal number of machines is also shown (this number was computed exhaustively offline). As shown, the proposed approach finds a solution that is close to optimal.

6. RELATED WORK

C-JDBC [6], RAIDb-1 [7], and Sequoia [25] develop architectures and systems for managing a large number of single-node commodity database systems on a cluster of machines. However, these approaches do not consider automatic failure recovery strategies and SLA management, which are the main areas of focus of this paper. Ganymed [21], DBFarm [22] and other related systems [19] develop solutions that scale to a large number of databases and database clients, but unlike the approach taken in this paper, they achieve this scalability by relaxing ACID semantics with weaker consistency criteria such as snapshot isolation. Furthermore, these approaches also do not deal with database SLAs.

There has been some recent work on distributing application servers across a cluster so as to effectively manage throughput and load [16, 29]. A key difference between this problem and the problem we consider in this paper is that the application servers do not have persistent state; consequently, the cost of migration and the associated consistency and SLA issues do not arise in that scenario. A related body of work focuses on providing quality of service (QoS) guarantees in shared hosted platforms [2, 30, 18, 3]. Similar to our approach, such systems profile applications on dedicated nodes and use these profiles to guide the placement of applications onto shared nodes. However, these systems rely on operating-system-level changes to monitor and enforce application isolation and performance guarantees, and do not deal with consistency issues for stateful applications. In contrast, our approach is to use commodity hardware and software components (including commodity operating systems and database systems) to monitor and manage databases at a coarse level using observation and appropriate reaction.

7. CONCLUSION

We have described the design and implementation of a data-management platform that can scale to a large number of small applications, i.e., each application can comfortably fit in a single machine and meet its desired SLA, but there are a large number of such applications in the system. A key design principle in our architecture is the use of low-cost single-node commodity hardware and software components. In this context, we have developed and evaluated various techniques for database deployment, replication, migration and SLA management that ensure high throughput and high availability, while still ensuring ACID semantics for database applications.

As part of future work, we are exploring more sophisticated methods for allocating databases to machines while still preserving SLAs, which can further reduce the hardware cost of the system. We are also exploring extensions to the system architecture that can accommodate "some" applications that are larger than the capacity of a single machine, while the majority of the applications can still fit in a single machine.

8. REFERENCES

[1] K. Aberer. P-grid: A self-organizing access structure for P2P information systems. *Lecture Notes in Computer Science*, 2172:179–194, 2001.

[2] M. Aron. Differentiated and predictable quality of service in web server systems. In *PhD thesis, Rice University*, 2000.

[3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Measurement and Modeling of Computer Systems*, pages 90–101, 2000.

[4] C. Baru and et al. Db2 parallel edition. *IBM Systems Journal*, 34(2):292–322, 1995.

[5] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 3 edition, 1987.

[6] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Cjdbc: Flexible database clustering middleware. In *USENIX Conference*, 2004.

[7] Emmanuel Cecchet. Raidb: Redundant array of inexpensive databases. *Parallel and Distributed Processing and Applications*, 2005.

[8] F. Chang and et. al. Bigtable: A distributed storage system for structured data. In *OSDI Conference*, 2006.

[9] Oracle Real Application Cluster. http://www.oracle.com/technology/products/database/clustering/.

[10] A. Crainiceanu and et al. P-ring: An efficient and robust p2p range index structure. In *SIGMOD Conference*, 2007.

[11] Google Gadgets. http://www.google.com/apis/gadgets/.

[12] Community Systems Group. Community systems research at yahoo! *SIGMOD Record*, 36(3):47–54, 2007.

[13] R. Huebsch and et al. The architecture of pier: an internet-scale query processor. In *CIDR Conference*, 2005.

[14] H. V. Jagadish, B.-C. Ooi, and Q.-H. Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB Conference*, 2005.

[15] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, 1997.

[16] A. Karve and et al. Dynamic placement for clustered web applications. In *WWW Conference*, 2006.

[17] L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. *IBM Journal of Research and Development*, 21(2), 1977.

[18] C. Li, G. Peng, K. Gopalan, and T. Chiueh. Performance guarantee for cluster-based internet services. In *ICDCS Conference*, 2003.

[19] Y. Lin and et al. Middleware based data replication providing snapshot isolation. In *SIGMOD Conference*, 2005.

[20] The Facebook Platform. http://developers.facebook.com/.

[21] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Middleware Conference*, 2004.

[22] C. Plattner, G. Alonso, and T. Ozsu. Dbfarm: A scalable cluster for multiple databases. In *Middleware Conference*, 2006.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM Conference*, 2001.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware Conference*, 2001.

[25] Continuent Sequoia Project. http://sequoia.continuent.org/homepage.

[26] Microsoft SQL Server. http://www.microsoft.com/sql/.

[27] Amazon SimpleDB. http://www.amazon.com/b?ie=utf8&node=342335011.

[28] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *SIGCOMM Conference*, 2001.

[29] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *WWW Conference*, 2007.

[30] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared

hosting platforms. In *OSDI Conference*, 2002.
[31] Yahoo! Widgets. http://widgets.yahoo.com/.