FlurryDB: A Dynamically Scalable Relational Database with Virtual Machine Cloning

Michael J. Mior and Eyal de Lara
Dept. of Computer Science, University of Toronto
Toronto, Ontario, Canada
{mmior,delara}@cs.toronto.edu

ABSTRACT

Stateless services are easy to scale in the cloud since new replicas of these services can be created at any time and they operate completely independently of other instances. In contrast, scaling stateful services, such as a database system, can take minutes or even hours due to the need to present a consistent view of the system for users of the service. Currently, this problem is addressed by resource overprovisioning in anticipation of demand spikes. FlurryDB uses virtual machine cloning to improve resource utilization by drastically reducing the latency required to add a new replica. We also show that FlurryDB is capable of handling updates to resources in a fashion that preserves consistency across the cloning boundary.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Design, Experimentation, Management, Measurement, Performance

Keywords

Database, Virtualization

1. INTRODUCTION

Providing scalability for relational database management systems (RDBMS) to allow for the handling of increases in offered load has traditionally been a complicated process requiring a full copy of a database to be created. This procedure can potentially take several hours [4]. In some cases, a replica must also be taken offline in order to make this copy. Additionally, since caches are cold on the new server, it can take even longer before the new instance can operate at peak capacity. The current solution to the problem is to overprovision and keep several extra instances available in anticipation of a spike in demand. The need for overprovisioning is worse than the stateless server case since the massive latency to copy the database means that additional slack capacity must be maintained to ensure acceptable system performance while new replicas are brought online.

This paper introduces FlurryDB, an approach that uses virtual machine (VM) cloning to dynamically scale cluster-based database servers by rapidly creating clones of live database workers. The key benefit of FlurryDB is that it does not require that a full copy of the database be present on the new worker in order to add it as a replica. Instead, it uses VM cloning to create a copy of a running replica which fetches both its memory and disk state on demand. This reduces the latency of adding a new replica by orders of magnitude, from minutes or hours, down to seconds.

The key challenge in FlurryDB is the need to handle requests that are ongoing at clone creation time. Specifically, if a replica is processing a request at the time of cloning (a likely scenario on a busy server), then each of the created clones will also be doing so when it comes alive. To handle this case, FlurryDB inserts a cloning-aware proxy into the VM, which is interposed in the communication path between the cluster load balancer and the unmodified database server instance running on the VM. The proxy enables queries which are in progress during a VM clone operation to complete in a manner that preserves database consistency.

Our experimental evaluation using unmodified instances of MySQL shows that FlurryDB can add a new replica to a clustered database in less than 25 seconds, enabling it to swiftly react to changes in load.

The rest of this paper is structured as follows. Section 2 provides an introduction into cluster-based database replication and VM fork, which FlurryDB uses to clone replicas. Section 3 introduces FlurryDB and discusses the main challenges it solves. Section 4 presents our prototype implementation. Sections 5 and 6 present an evaluation of the system and discuss the applicability of FlurryDB to other replication models, as well as avenues for future work. Finally, Sections 7 and 8 compare FlurryDB to related work and conclude the paper.

2. BACKGROUND

In this section we first provide some background on cluster-based database replication. We then describe the VM fork mechanism that FlurryDB uses to swiftly clone database replicas.

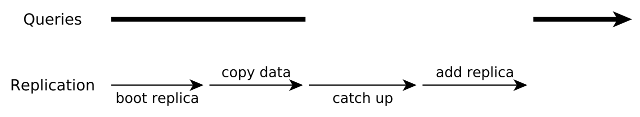

Figure 1: Traditional replica creation process

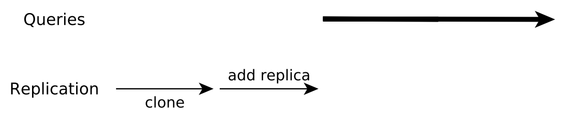

Figure 2: FlurryDB replica creation

2.1 Cluster-Based Database Replication

Cluster-based database servers are implemented as a dynamically changing pool of load-balanced workers. Each worker runs an unmodified single node database server and consistency is maintained by a replication algorithm.

While different replication algorithms have been proposed [10], we consider in this paper read-one write-all replication [2]. In this approach, each replica keeps a full copy of the database. Read queries can be sent to any worker, while write queries are replicated to all workers under the control of a distributed commit protocol, such as two-phase commit [14]. This provides a high-level of scalability for reads, but writes can still be a significant bottleneck since each replica must execute all write transactions. We discuss the implications of our replica addition technique on other database replication and partitioning approaches in Section 6.

Traditionally, scaling a cluster of read-one write-all replicas requires that a copy of the entire database must be made. Depending on the DBMS and the replication algorithm used, one of the replicas may need to be taken offline to perform this copy. If strong consistency is desired, then the new worker must be allowed to catch up before it is added to the pool of accessible servers. This also requires replica addition to be an atomic operation which takes place when no queries that modify the database are in progress. The complete process is outlined in Figure 1. The break in the top line represents the period during which queries must be interrupted while the bottom line shows the steps required to add a new replica. In contrast, FlurryDB uses VM cloning to enable the swift addition of new replicas without requiring database pre-copying.

2.2 VM Fork

FlurryDB uses the SnowFlock [12] implementation of VM fork to swiftly clone database replicas. SnowFlock designates one VM as the master. From the master, VMs can be forked in parallel onto multiple distinct physical hosts. The forked VMs are referred to as clones of the master. The clones start with identical memory and disk state to the master which is fetched on demand over the network on access. To ensure consistency at the point of cloning, all memory on the master VM is write protected and a copy is made on each write.

To instantiate VM clones, an architectural descriptor of the master is created consisting only of the master's page tables and a small amount of additional metadata. This descriptor is then transferred over the network and services are started on each machine to enable state transfer. This allows the core of the cloning operation to complete in less than one second. Pages which are marked non-present in the cloned VM trigger a page fault. SnowFlock uses VM state coloring [3] to optimize the propagation of state to clones by identifying semantically related regions. Specifically, it uses page table introspection to tailor the prefetching of kernel vs. user space regions and code vs. data regions, as well as guest OS introspection to optimize the propagation of the file system page cache.

The semantics of disk cloning are identical to memory: clones see the same disk, although modifications to it remain private, and are discarded upon clone termination. The local disk is provided purely on-demand, but is rarely used by transient clones who find most of their requests satisfied by in-memory caches.

Upon cloning, a clone's IP address is automatically reconfigured. Clones share an internal private network with their parent, other clones, and select entities such as the load balancer. Clones are assigned a new IP address within the private network as a function of their ID. The reconfiguration of IP addresses requires no developer intervention.

On the master VM's side, network connections that are open at the point of cloning remain open and working. On the cloned VM, the connection is inherited but the assignment of a new IP address during cloning forces all inbound and outbound connections to drop – in many cases, this will result in automatically discarding session state not discarded by the parent. However, the new clone has to graciously deal with broken connections to system resources, such as the load balancer. Once those connections are re-established, the new, cloned workers require no further network-plane intervention.

Finally, SnowFlock provides two mechanisms that enable applications running on a clone to react to a cloning operation. First, a reconfiguration hook automatically invokes a reconfiguration script after cloning, if one has been registered. The script can be used, for example, to remount an NFS partition after automatic IP reconfiguration. Second, an asynchronous signal, SIGCLONE, can be sent to processes once a clone comes alive. Processes explicitly subscribe to receive SIGCLONE, which sends a POSIX real-time signal and thus terminate the process if unhandled.

3. FLURRYDB

FlurryDB leverages virtual machine (VM) cloning to rapidly create clones of live database workers. FlurryDB does not require that a full copy of the database be present by leveraging the copy-on-demand mechanism provided by SnowFlock. The complete process is outlined in Figure 2.

FlurryDB is similar to the standard cluster-based database design in that it requires a load balancer that maintains state consistency amongst a pool of worker replicas and has the capability to grow and shrink the replica pool in response to changes in load. Where FlurryDB differs is in its requirement to handle state at clone creation time. Specifically, if a replica is processing a request at the time of cloning (a likely scenario on a busy server), then each of the created clones will also be doing so when it comes alive. An alternative is to sidestep the issue by queuing new write operations at the load balancer and cloning only after all ongoing writes have committed at the master (i.e., forcing a *write barrier*). We decided against this approach as it adds significant latency to the cloning operation, limiting its benefits.

FlurryDB needs to ensure that operations that are underway at the time of cloning are treated as follows: (1) read operations do not require any special treatment. The change in IP address on the clones will force the connections to the load balancer to be dropped, aborting the operation. On the master VM, network connections that are open at the point of cloning remain open and working. Thus the read operation will proceed as normal and its result will be communicated back to the load balancer, which can then forward it to the client; (2) operations that modify database state are a more complex case as they must either finish or abort atomically on all replicas, including the new clones.

The approach that aborts the operation is easier to implement. On the new clones, the change in IP address will force the connection to the load balancer to drop and the operation to abort. What is left to do is for the distributed commit protocol on the load balancer to instruct all pre-existing replicas, including the master VM, to roll back the operation. This approach, however, wastes work and is likely to add even more latency than implementing a write barrier, as the operation has to be resubmitted by the client.

An approach that guarantees that operation will finish atomically on all replicas requires adding cloning awareness to the application running inside the cloned VM. One possibility is to modify the database server so that after receiving the SnowFlock SIGCLONE signal it reconnects to the load balancer, runs the in-flight requests to completion and joins the distributed commit protocol. Instead, we opted for a less intrusive approach that inserts a cloning-aware *frontend proxy* into the VM. The frontend proxy maintains database connections with the database server on behalf of the load balancer. Since it lies inside the VM, its connections to the database server are not affected by the change in IP address and are maintained across the boundaries of the VM clone operation. This allows queries which are in progress during a VM clone operation to complete. After cloning, the frontend proxy reconnects to the load balancer, and joins the distributed commit protocol. As an optimization, the frontend proxy and load balancer communicate using a light weight protocol without heavy authentication. This protocol provides lower connection latency compared to traditional database connection setup. Figure 3 shows the layout of the complete system.

4. IMPLEMENTATION

We implemented our cloning-aware proxy using the MySQL client library as well as some code from the MySQL server. The proxy is capable of serving as both *load balancer* and *frontend proxy* depending on the runtime configuration. A detailed diagram of the architecture of the proxy is given in Figure 4. The arrows in the diagram represent the path of a query through the server. A single loop accepts connections and passes them off to threads which handle each client. When a new query arrives, it is run through the query mapper which defines the mapping between query strings and backend servers. If the query only needs to be passed to a single backend, the client grabs a connection from a pool and issues the query. If the query needs to be sent to multiple

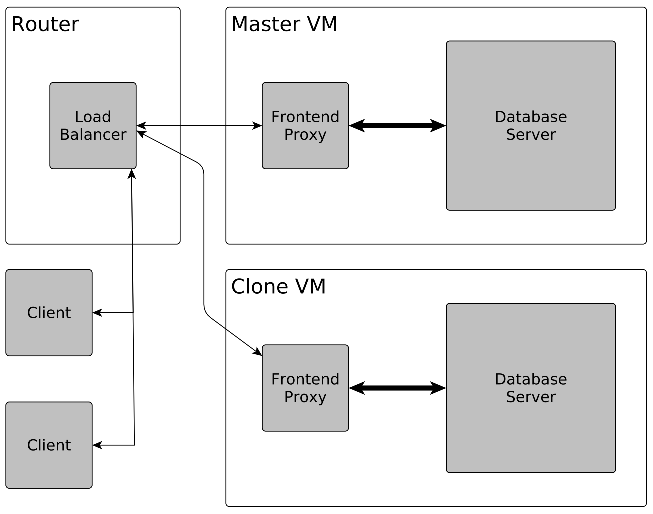

Figure 3: FlurryDB system architecture

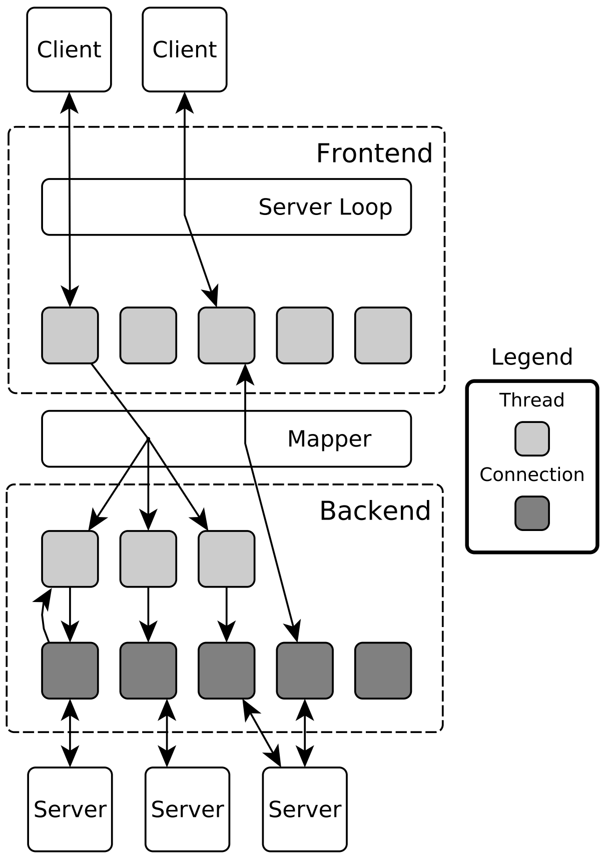

Figure 4: Proxy architecture

```
#define SELECT   "SELECT"
#define SHOW     "SHOW"
#define DESCRIBE "DESCRIBE"
#define EXPLAIN  "EXPLAIN"

proxy_query_map_t proxy_map_query(char *query,
        char *new_query) {
    new_query = NULL;

    /* Anything which starts with the keywords above
    * goes to any backend, otherwise, go everywhere */
    if (strncasecmp(query, SELECT, sizeof(SELECT)-1) == 0
     || strncasecmp(query, SHOW, sizeof(SHOW)-1) == 0
     || strncasecmp(query, DESCRIBE, sizeof(DESCRIBE)-1) == 0
     || strncasecmp(query, EXPLAIN, sizeof(EXPLAIN)-1) == 0)
        return QUERY_MAP_ANY;
    else
        return QUERY_MAP_ALL;
}
```

Figure 5: Read-one write-all query mapper

backend servers (i.e. the query modifies the database), it is passed on to several threads which execute the request in parallel, and then synchronize before one of those backend servers forwards the response to the client.

To allow for flexibility in the way queries are passed to backends, the proxy supports dynamically loading "mapper" libraries. Only a single mapper library may be loaded by the server at any given time. In the absence of a mapper library, a backend server is selected at random. The mapper takes a query string as input and returns a flag which specifies where the query should be sent (currently either `ANY` or `ALL`). The load balancer currently uses read one, write all replication to maintain consistency (see Figure 5). This has the advantage of simplicity and strong consistency (in the absence of network partitions). We plan to examine other techniques to improve performance in the future, as this strategy scales poorly with write-heavy workloads. When serving as the frontend proxy, we have no need for a mapper, since queries which require replication have already been tagged by the load balancer.

The proxy server also introduces several additional SQL statements which are used in the distributed commit protocol in the next section. Using the MySQL protocol to execute these statements eliminates the need to design a new protocol and makes it easy to manage interactive sessions during debugging and testing since the standard MySQL client can be used. The additional commands introduced are given in Table 1.

4.1 Cloning and Distributed Commit

Database cloning is performed using the SnowFlock implementation of VM fork, which is built on top of Xen 3.4.0. The decision to clone is made by the load balancer, which we will refer to as the coordinator. When clones become live, they contact the coordinator so it can add each clone as a new backend server and manage committing across all clones. To ensure consistency, the load balancer queues new writes from the time cloning starts until all new clones are added as replicas. Thus, minimizing the latency of this operation is critical for the performance of the system.

As a proof of concept, we have chosen to use a two-phase commit (2PC) protocol with read-one write-all replication due to its ease of implementation. Other distributed commit protocols such as Paxos commit [8] or eventual consistency [9] may provide better scalability and performance, but this is outside of the scope of this project.

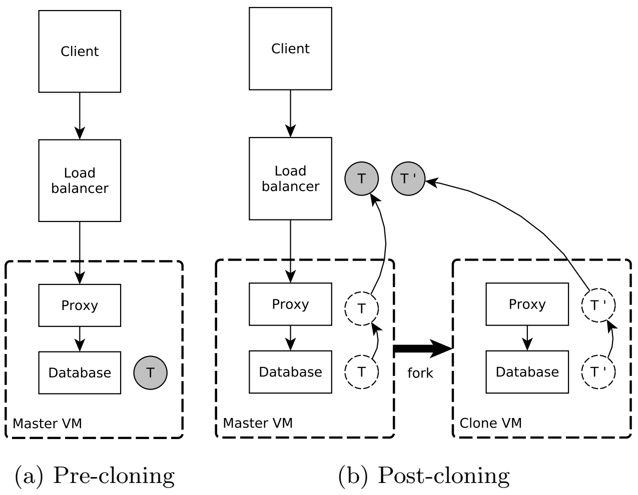

Figure 6: Cloning procedure

We implement 2PC using the standard technique when the set of backend servers is static (i.e. when a cloning operation is not taking place). The load balancer sends the query to each backend and waits until it has received a successful response from each. It will then send a commit message to each backend if all reported success, otherwise the transaction will be rolled back. In either case, a result packet is then synthesized and sent to the client.

The situation becomes more complicated when a query is in progress during cloning. The statement will continue to complete on the backend server and the frontend proxy will receive the response. However, the connection to the coordinator will be severed as a result of the change of IP address on the clone. The clone must therefore reconnect to the coordinator and notify it of the status of the transaction. In order to facilitate identification of the transaction, we add an identifier to each replicated query which can be returned in the status message. It is also used by the coordinator to send a commit or rollback message to the clone once all responses have been received.

Figure 6a illustrates how the system works during normal operation. A write `T` is initially submitted by the client and then passed through the load balancer and proxy to the backend database. When the database has completed processing on this transaction, it will be return via the same path to the client. Figure 6b shows how the process changes when the database is cloned. While cloning the virtual machine, we have also cloned the write operation, resulting in `T'`. The load balancer is aware of this change, and thus waits until it receives the result of both `T` and `T'` before responding to the client. The result of `T` is returned via the same path as the input query. Since the network connection between the load balancer and the cloned VM is severed, when the cloned frontend proxy receives the result of `T'`, it will open a new connection with the load balancer to pass the result.

FlurryDB currently does not support transactional workloads since we make use of the transactional features of MySQL in order to implement 2PC. It is still possible to

Statement	Description	Parameters
GLOBAL STATUS	Status information on the server	N/A
STATUS	As above, but for the current connection	N/A
CLONE	Create a new virtual machine clone	number of clones
ADD	Add a clone to the list of backends	clone ID, clone IP
SUCCESS	Report success of a transaction	clone ID, transaction ID
FAILURE	Report failure of a transaction	clone ID, transaction ID
COMMIT	Signal a commit to a clone	transaction ID
ROLLBACK	Signal a rollback to a clone	transaction ID

Table 1: Proxy commands

support transactional workloads by tracking when clients commit and roll back transactions, but this is left as future work. Therefore, updates to the database must consist of single statements. The semantics are equivalent to running a traditional RDBMS in autocommit mode.

5. EVALUATION

In this section we first use microbenchmarks to quantify the performance benefits of being able to clone a database replica while write operations are ongoing. We then use a realistic benchmark (RUBBoS) to show that FlurryDB can swiftly grow the replica pool of a cluster-based database server enabling it to react to changes in demand.

5.1 Microbenchmarks

The master and clone VMs used in our evaluation are running MySQL 5.1.4.9 with 1GB of RAM, one virtual CPU, and a 3GB disk. Each is hosted on a Sun Fire X2550 with eight 3.0GHz Xeon cores, 8GB RAM, and dual gigabit Ethernet NICs. Traffic produced by SnowFlock and traffic produced by benchmarks are each sent on separate interfaces. Virtualization is performed using SnowFlock, which is built on top of Xen 3.4.0 and Linux 2.6.18. Both the host machine and the VMs are running 64-bit Debian Core 5.

Our load balancer runs on a third machine with identical hardware and operating system, but uses version 2.6.26 of the Linux kernel. We also use an additional machine to run the clients that drive the benchmarks. This system has four 3.6GHz Xeon processors running Fedora Core 8 with version 2.6.16.29 of the Linux kernel and dual gigabit Ethernet NICs.

5.1.1 Benefits of cloning ongoing queries

To examine the benefits of cloning the database while write operations are in progress, we create a simple test table with two integer columns and insert one million rows with consecutive primary keys. We then measure the time it takes to clone the database under four write workloads as we vary the number of concurrent clients issuing queries to the database. We consider two types of queries: *Small*, which updates 1,000 rows; and *Large*, which updates 10,000 rows. For each type, we run two versions: *Different*, where multiple copies of the query target different sets of tuples; and *Same*, where queries modify the same tuples. We disabled table-level locking for these tests so writes to different rows can occur simultaneously.

Table 2 shows the time it takes to add a new replica under two experimental configurations: *FlurryDB*, which uses our frontend proxy to enable cloning while write operations are ongoing, and *Write Barrier*, which waits for all ongoing writes to finish before cloning the VM. Shortening the time it takes to add a new replica is important as it reduces the length of time the load balancer has to wait before issuing new writes.

Workload	Write Barrier	FlurryDB	Clients
Small, Same data	7.5s	6.6s	50
Small, Different data	7.7s	5.5s	50
Small, Same data	20s	7.4s	100
Small, Different data	11.5s	7.4s	100
Large, Same data	28.1s	6.5s	50
Large, Different data	21.5s	6.8s	50

Table 2: Time to add a new replica. *Small* and *Large* update 1,000 and 10,000 rows, respectively. In *Different data* experiments, multiple copies of the query target different sets of tuples; whereas in *Same data* experiments, all copies of the queries modify the same tuples.

The experimental results show that FlurryDB facilitates the swift addition of new replicas. Replica addition times varied between 5.5 sec and 7.4 sec and experienced modest increases as we added more concurrent clients. In contrast, waiting for writes to finish before cloning can significantly increase blocking time (by a factor of 4.3 in our experiments), and results in variable replica addition times that are dependent on both the workload and the number of concurrent queries.

Figure 7 further examines the worse case scenario of large writes to the same block of data as we increase the number of clients. We see that as the number of clients increases, there is a significant increase in waiting for the queries to complete. This is expected as each query is blocked until all other queries which arrived before complete. In contrast, FlurryDB handles ongoing queries effectively, enabling cloning to proceed almost immediately.

5.1.2 Reconnection time

As mentioned in Section 3, we don't use any form of authentication between the frontend proxy server and the load balancer. This approach is acceptable in our environment because the database servers are not publicly accessible and SnowFlock uses MAC-level filtering to provide a private network between the clones and the master.

As a result, we establish connections with significantly less overhead. We measure the overhead by opening and closing 100,000 connections to a standard MySQL server,

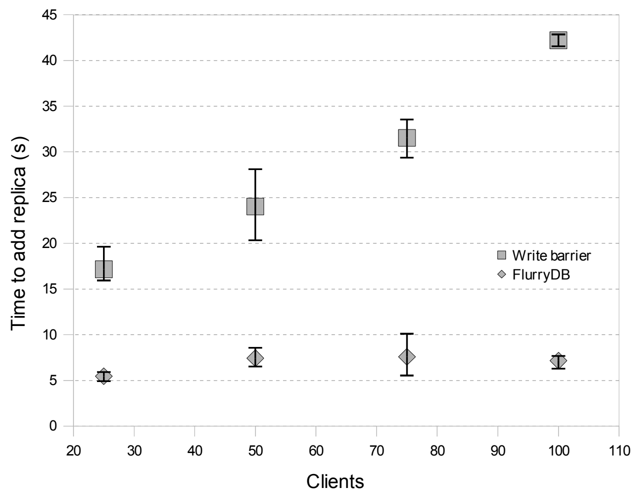

Figure 7: Effect of waiting for query completion on replica addition delay

	Time (s)
MySQL	17.3
FlurryDB	1.55

Table 3: Connection establishment

followed by the same number of connections to the FlurryDB proxy. To eliminate network overhead, we establish all of these connections on the same host. As shown in Table 3, the elimination of authentication allows us to establish connections an order of magnitude faster.

5.1.3 Overhead

To measure the overhead introduced by adding the frontend proxy to the data path, we measure the time it takes to select 367 MB from a test database. We execute the statement (`SELECT * FROM test`) before measurement to ensure caches are warm. We consider three different configurations: (1) a client directly connected to the database server; (2) a client connected to the database server through the load balancer; and (3) a client connected to the database server through the load balancer and the front end proxy. Each hop in the transfer process is on a different physical host. Results are given in Table 4. The final overhead observed is approximately 5% for two levels of proxying as compared to no proxying. The overhead introduced by the frontend proxy is larger than the overhead of the load balancer because the former competes with resources with the database server running on the same VM, whereas the latter runs on a dedicated host.

In practice, the extra overhead of the frontend proxy need not apply to read-only queries, since it would be possible to modify the load balancer so that it sends read queries directly to the database server bypassing the frontend proxy. However, our current prototype does route all reads through the frontend proxy.

Latency is another significant factor affecting performance. Adding an additional network hop can significantly increase this delay. We feel this can be mitigated via kernel-level

Proxying level	Bandwidth (MB/s)
No proxying	28.5
Load balancer	28.4
Frontend proxy and Load balancer	27.1

Table 4: Read bandwidth

layer 7 routing, i.e. performing query mapping in the kernel, as in the Kernel TCP Virtual Server (KTCPVS)[19]. This removes one layer of memory copying along the data path. This also enables the possibility of direct server reply for read-only queries by routing the packet directly to the backend server and allowing it to reply with the IP of the load balancer.

5.2 RUBBoS

We use RUBBoS [1] to illustrate how FlurryDB can swiftly grow the replica pool of a cluster-based database server enabling it to react to changes in load. The RUBBoS benchmark simulates traffic generated by users viewing and commenting on an online news forum. The benchmark is implemented as a PHP application which makes database requests coupled with Java clients which drive the workload.

We use two Sun Fire X2550s to host the master VM and the clone VMs. These machines each have eight 3.0GHz Xeon cores and 8GB of RAM. Each machine also has dual gigabit Ethernet NICs which we use to separate SnowFlock and benchmark traffic. The operating system is 64-bit Debian Core 5 with Linux 2.6.18. The version of SnowFlock we use to perform cloning is built on top of Xen 3.4.0. The load balancer and application server are run on identical hardware. The application server is running Fedora Core 8 with Linux 2.6.23.

Two additional machines are used to execute the RUBBoS Java clients. These systems have four 3.6GHz Xeon processors and dual gigabit Ethernet NICs. Each system runs Fedora Core 8 with Linux 2.6.16.29.

The PHP application uses Apache 2.2.6 and PHP 5.2.4. We use the default state transition tables provided with the RUBBoS distribution for selecting the types of requests made by clients. We did however require slight changes to the PHP application to fix bugs and resolve incompatibilities with the newer version of PHP. Our modifications to RUBBoS may be found on GitHub[15].

We start FlurryDB with a single VM and a copy of the 327MB test database provided with RUBBoS. All tables use the InnoDB database engine. In order to increase the load on the system, we set the think time of all clients to zero. We begin with 25 clients executing a one-minute warmup workload. We then run at full capacity for 5 minutes before executing a clone operation. After the clone has successfully been created and reconnected to the load balancer, we start a second instance of the benchmark with an additional 25 clients.

Figure 8 shows a time series of queries per second passing through the system as observed by the load balancer. As expected, we see a sharp drop in throughput while the cloning operation takes place (around the 300 second mark), followed by a period of degraded performance while SnowFlock fetches the working set. The server then stabilizes and is able to fully utilize the resources of the new replica.

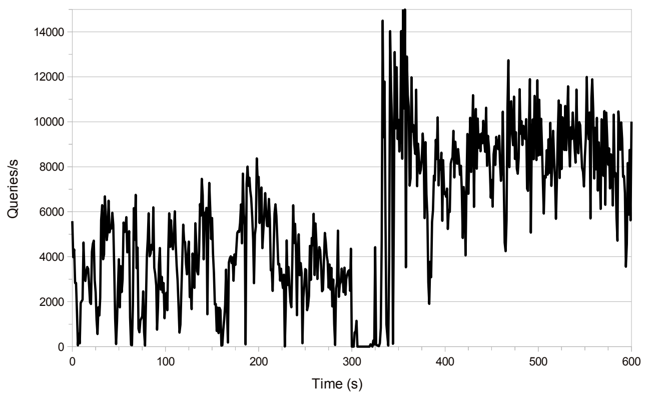

Figure 8: Query throughput for RUBBoS as we clone a new replica during cloning

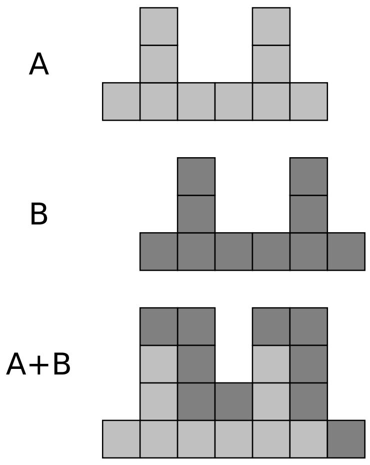

Figure 9: Effect of bursty workload on benchmark throughput

The delay before achieving stability is a result of three factors: (1) writes blocking until completion on the new clone; (2) memory pages being transferred to the clone; (3) clients delaying new requests until previous requests have completed. Interestingly, the average throughput of the server more than doubles after the clone becomes available. This is explained by the bursty nature of the benchmark resulting from the non-uniform distribution of request arrival times and large variations in the run times of different query types. With a single replica, the load offered by the 25 clients creates spikes that surpass the capacity of the server, and periods of low demand when the server is under-utilized. With two replicas, and twice the number of clients, the average load doubles, but spikes become less pronounced as the aggregate load is more uniform. Figure 9 illustrates this process by showing the effect of adding the offered load of two sets of clients.

6. DISCUSSION

FlurryDB could implement a more conservative replication policy. This may require relaxed consistency, but this is acceptable for many classes of applications. The choice of different policies could provide a range of options for trade-offs between consistency and performance. One such scheme is master-slave replication [17]. In the case of FlurryDB, we have the possibility of cloning either a master or a slave. If we make a clone of the master database, the frontend proxy would ensure that queries that are in progress complete in a manner that preserves database consistency. After cloning, the new replica would become a slave. Cloning a slave database should simply require reconnection with the master. At this point, replication could continue after any transactions which were in progress at the time of cloning are committed to the database.

A more complex possibility is to incorporate dynamic partitioning as clones are created. With an appropriate partitioning scheme (e.g., sharding [5]), the need for replication could be significantly reduced. This is likely to be highly challenging in the absence of application-level support. The abandonment of read-one write-all also introduces the problem of reintegration of changes which have not been seen on the master, which complicates scaling down.

To make replication even simpler, we would also like to implement load metrics and a policy based on service level agreements (SLAs) for creating and destroying clones to enable fully automated scalability. This could be coupled with an adaptive Web server, such as that used by the Kaleidoscope [3] project to create a dynamically scalable multi-tier Web application platform.

It is feasible to extend FlurryDB to support transactional workloads by tracking when the MySQL server believes it is in a transaction. One option is to intercept all `COMMIT/ROLLBACK` statements and ensure they are issued only when all statements in the transaction have succeeded on all clones. Another option is to use XA distributed transaction support in MySQL and convert transactions in user workloads into distributed transactions on-the-fly.

The SnowFlock virtual disk was not intended to provide high performance, but rather to provide a base image where applications and configuration files are stored. One major bottleneck in FlurryDB (for databases which do not fit in memory) is the latency of this disk access on clones, as the disk is fetched on-demand over the network. Ideally, the majority of disk state which is used will be otherwise available in either the buffer cache on the clone or application-specific cache in the DBMS. However, some requests inevitably hit the disk, so improvements in SnowFlock virtual disk performance will likely prove to be highly advantageous to FlurryDB. Currently all disk reads to pages not present on a clone are serialized and disk pages are sent one at a time over the network. Possible areas for performance improvements include support of parallel requests and a policy for prefetching to exploit the locality of reads.

7. RELATED WORK

A currently popular solution for database scalability is the use of databases providing eventual consistency or key-value stores such as Cassandra [13] or Amazon's Dynamo [7]. Many systems designed for horizontal scalability abandon the relational model and lose features such as joins. The modified data model often results in data duplication. These alternative data models present an obstacle for database administrators and developers who are familiar with the use of RDBMS. The relational model provides a more intuitive data model and also simplifies application logic. Eventual consistency is a plausible alternative to two-phase commit currently used by FlurryDB with the potential to improve performance.

Kaleidoscope [3] also makes use of SnowFlock, with the intent of providing similar scalability for Web servers. However, the replicated servers are stateless and thus require no effort to maintain consistency. Urgaonkar et al. [18] use rapid changes in resource allocation to provision servers for a multi-tier Internet application. However, they acknowledge that databases are difficult to replicate on-the-fly and instead limit the rate of requests to the database tier. Also, when a server for a tier is deallocated, its resources are reduced such that any cached state is lost.

Soundararajan et al. [16] describe dynamic replication policies for scaling database servers. This work identifies the latency in introducing new replicas as a major bottleneck in its approach. The combination of this replication policy with FlurryDB for provisioning and replication would likely produce a significantly more efficient system. The central issue experienced by Soundararajan et al. was oscillation caused by the delay in adding replicas. Since adding new replicas is a high latency operation, several replicas are added when query response times increase. Finally, when these replicas are available, response time plummets and a replica is destroyed. If the load continues to increase, this cycle could continue. Since FlurryDB is capable of reducing replica addition delay by orders of magnitude, it provides a clean solution to this issue.

Curino et al. introduce a "database-as-a-service" (DBaaS) called Relational Cloud [6] which aims to present efficient multi-tenancy, elastic scalability, and database privacy. For scalability, Relational Cloud relies on a cluster of backend servers running MySQL and Postgres database servers. A transaction coordinator periodically examines the workload to determine an efficient partitioning scheme. Relational Cloud hopes to support live migration of partitions between nodes by fetching data on-demand from the previous node where the backend data was stored. As this has not been implemented, it is not possible to evaluate the performance of this approach. While this approach may be suitable for DBaaS providers who wish to manage their own infrastructure, Relational Cloud does not provide efficient scaling for a single-tenant system. This may be desirable to allow execution in a hybrid cloud computing environment.

Finally, HyPer [11] uses process fork and memory snapshots to enable a hybrid main-memory database which can simultaneously support OLAP and OLTP workloads. Since it is acceptable for OLAP queries to use stale data, HyPer periodically takes a snapshot of the database via fork, employing OS-level copy-on-write mechanisms to maintain the performance of the OLTP workload. While this is a similar approach to the one taken by FlurryDB, it is only designed for use on a single node and provides no solution for handling an OLTP or OLAP workload which exceeds the capacity of the node.

8. CONCLUSION

FlurryDB provides a transparent method of replicating relational databases in the cloud using virtual machine fork. We do this using an off-the-shelf RDBMS and unmodified applications by providing a layered proxying mechanism able to manage the consistency of a virtual cluster across the boundary of VM fork.

The use of VM fork allows FlurryDB to quickly add replicas, reducing delay by orders of magnitude – from minutes or hours to seconds. In addition, allowing writes to continue during the fork allows a further reduction in this delay. For a workload with heavy locking, this can produce a further fourfold decrease in delay. The reduced delay achieved by this approach enables FlurryDB to rapidly adapt to changes in load, reducing the need to maintain idle workers. As future work, we plan to examine the scalability of FlurryDB to larger numbers of nodes as well as to larger database sizes where the overhead of copying is more pronounced.

9. ACKNOWLEDGEMENTS

We would like to thank Alexey Tumanov for his input in the early stages of this project. Thanks are also owed to Roy Bryant for his assistance based on his work on stateless server cloning and to the Kaleidoscope team for improvements to the SnowFlock platform on which FlurryDB is currently built. Finally, to the numerous members of the Systems & Networks Group at the University of Toronto for their feedback prior to submission.

10. REFERENCES

[1] Amza, C., Chanda, A., Cox, A. L., Elnikety, S., Gil, R., Rajamani, K., Zwaenepoel, W., Cecchet, E., and Marguerite, J. Specification and implementation of dynamic web site benchmarks. In *5th Workshop on Workload Characterization* (November 2002).

[2] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency control and recovery in database systems*. Addison Wesley, Massachusetts, 1987.

[3] Bryant, R., Tumanov, A., Irzak, O., Scannell, A., Joshi, K., Lagar-Cavilla, H. A., and de Lara, E. Kaleidoscope: Cloud micro-elasticity via VM state coloring. In *Proceedings of the 6th European Conference on Computer Systems* (2011), ACM Press, pp. 273–286.

[4] Cecchet, E., Candea, G., and Ailamaki, A. Middleware-based database replication: The gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD international conference on management of data* (June 2008), ACM Press.

[5] Ceri, S., Negri, M., and Pelagatti, G. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD international conference on management of data* (June 1982), ACM Press.

[6] Curino, C., Jones, E., Popa, R. A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., and Zeldovich, N. Relational cloud: A database service for the cloud. In *5th Biennial Conference on Innovative Data Systems Research* (2011).

[7] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (Oct. 2007), vol. 41, ACM Press, pp. 205–220.

[8] Gray, J., and Lamport, L. Consensus on transaction commit. *ACM Transactions on Database Systems 31* (March 2006), 133–160.

[9] Gustavsson, S., and Andler, S. F. Self-stabilization and eventual consistency in replicated real-time databases. In *Proceedings of the first workshop on Self-healing systems* (2002), ACM Press, pp. 105–107.

[10] Kemme, B., and Alonso, G. A suite of database replication protocols based on group communication primitives. In *Proceedings of the The 18th International Conference on Distributed Computing Systems* (1998), IEEE Computer Society.

[11] Kemper, A., and Neumann, T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the IEEE 25th International Conference on Data Engineering* (2011), IEEE Computer Society.

[12] Lagar-Cavilla, H. A., Whitney, J. A., Scannell, A. M., Patchin, P., Rumble, S. M., de Lara, E., Brudno, M., and Satyanarayanan, M. SnowFlock: Rapid virtual machine cloning for cloud computing. *European Conference on Computer Systems* (2009), 1–12.

[13] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (Apr. 2010), 35–40.

[14] Lampson, B., and Sturgis, H. Crash recovery in a distributed storage system. Tech. rep., Xerox Palo Alto Research Centre, April 1979.

[15] Mior, M. RUBBoS source repository. `https://github.com/michaelmior/RUBBoS`, 2011.

[16] Soundararajan, G., Amza, C., and Goel, A. Database replication policies for dynamic content applications. *ACM SIGOPS Operating Systems Review 40*, 4 (Oct. 2006), 89–102.

[17] Stonebraker, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions Software Engineering 5*, 3 (1979), 188–194.

[18] Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. Agile dynamic provisioning of multi-tier Internet applications. *ACM Transactions on Autonomous and Adaptive Systems 3*, 1 (2008), 1–39.

[19] Zhang, W., and Zhang, W. Build highly-scalable and highly-available network services at low cost. *Linux Magazine* (2003), 14–14.