HASQL: A Method of Masking System Failures

Mark Hannum, Adi Zaimi, Michael Ponomarenko, Dorin Hogea, Akshat Sikarwar, Mohit Khullar, Rivers Zhang, Lingzhi Deng, Nirbhay Choubey, Joe Mistachkin
Bloomberg L.P.
{mhannum,azaimi,mponomar,dhogea,asikarwar1,mkhullar1,hzhang320,ldeng33,nchoubey,jmistachkin}@bloomberg.net

ABSTRACT

We demonstrate a methodology of masking system failures in a way that doesn't require programmer or operational intervention, and that strives to be imperceptible to the client. High Availability SQL (HASQL) masks system failures in a clustered database-system by *seamlessly* restoring a transaction's state against a different machine in the cluster. We have implemented HASQL in Comdb2, an open source RDBMS developed by Bloomberg L.P.

To demonstrate, we allow participants to kill (via a button) database instances one at a time, and all instances simultaneously, as we execute an ongoing transaction against a Comdb2 cluster. Upon achieving this, viewers will see the command-line session and transaction pause briefly as the Comdb2 client-API connects to a different cluster node and re-establishes the transaction's state, allowing it to resume processing at the exact point of the disconnect. We repeat this demonstration, showing that a transaction's state can be correctly re-established even though it is midway through consuming a result set.

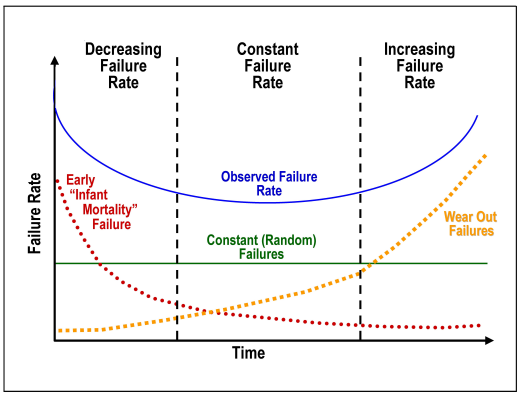


Figure 1: 'Bathtub curve' hazard function, adopted from [1], describes the failure rate of a single-component.

1 INTRODUCTION

Plummeting hardware prices have created increasing pressure on software developers to design redundant systems, as the chance for failure for any component of a system increases over time (see figure 1). For RDBMS systems, *High-Availability* solutions attempt to restore service quickly and minimize the effects of outages[2]; indeed, many commercial RDMBS systems[3, 5, 6] support automatic failover to provide uninterrupted service under the loss of part of a database cluster. In traditional failover strategies, a crashed server will return a `CONNECTION LOST` error to the client. Application writers address this by programming defensively, marrying database API calls to complex and often poorly tested retry logic[7]. Handling a `CONNECTION LOST` error in response to a `COMMIT` directive gives the programmer the additional burden of determining the fate of that transaction.

Our contribution is unique in that we show how to provide *seamless* continuation of in-flight transactions under an *optimistic concurrency control (OCC)* system: HASQL clients do not reissue SQL in the face of machine failure, and need not be aware that a machine failure has occurred, as every in-flight transaction will be automatically re-established and continued against another machine in the cluster, and any partially consumed result set will continue to be returned, as the system we describe guarantees that the client will never experience duplicate or missing data.

Oracle's *Application Continuity*[4] feature is a proprietary implementation which achieves the same goal. As we have implemented this as part of an open source system, we are able to describe our methodology explicitly, and we hope that by doing so, other systems can likewise provide this feature to their users. We note that any correct methodology will ensure that the replayed SQL modifies the same set of records that would have been modified on the original host, but that in an OCC system, write conflicts for replayed transactions are handled in the same manner as write conflicts for any two competing transactions: the winning transaction will be allowed to modify the rows in its write-set, while the losing transaction will return a verify error to the client. So even though HASQL provides the ability to perfectly replay a read-write transaction, as with every transaction in an OCC system, the replayed transaction will only be able to commit successfully if the rows in its write-set have not been modified.

In this paper, we describe HASQL as implemented in Comdb2, an open source RDBMS developed at Bloomberg L.P.[1], noting that a brief overview of HASQL was outlined in [8]. As this method relies on a limited number of architectural features, we describe it generally, trusting that it should be straightforward to implement HASQL in a similar system. Our motivation for this feature grew organically from Bloomberg's business need to have an *always available* database server and to provide an intuitive and reliable software infrastructure layer to its application developers. HASQL achieves this by shifting the responsibility for handling hardware failures from the client application to the database system.

2 SYSTEM OVERVIEW

We now describe the architecture and methodology for implementing HASQL. See [8] for a comprehensive review of Comdb2's architecture.

[1] https://github.com/bloomberg/comdb2

2.1 Architecture

A *database cluster* consists of N *database instances* running on N physically separate machines sharing no components and connected to each other by a network. Each instance contains a complete copy of the data, and is able to start and maintain arbitrary point-in-time snapshot transactions.

The cluster maintains a single *master* which may modify database state, and which synchronously replicates these modifications to every other instance in the cluster. We assume that transactions are immutable and are applied atomically on the master in a serial order defined by the global order of transaction `COMMIT`s.

For any two committed transactions, T1 and T2, if T1 is committed first, a point-in-time snapshot transaction started at T1's commit point will observe all the effects of T1's modifications, and none of the effects of T2's modifications, while new snapshot transactions will observe the effects of both T1 and T2. Modifications are applied to non-master instances (or *replicants*) atomically in the same serial order as they were applied on the master.

A *point-in-time token (or PIT-token)* identifies a specific point in the serial order of committed transactions. Given a PIT token, each instance is able to produce a snapshot view corresponding to that point in the serial order. The *transaction-id (tid)* is a unique identifier for a transaction; it is generated by the client API at the beginning of a transaction.

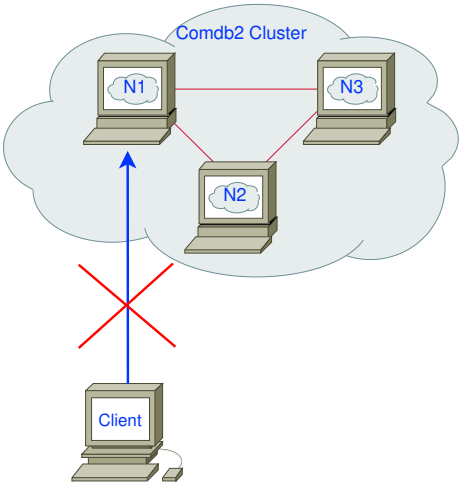
Writes performed against a replicant are not executed locally, but rather validated and executed on the master after the client issues a `COMMIT`. The client API is aware of the cluster's topology and maintains a single connection against an arbitrary replicant.

2.2 Methodology

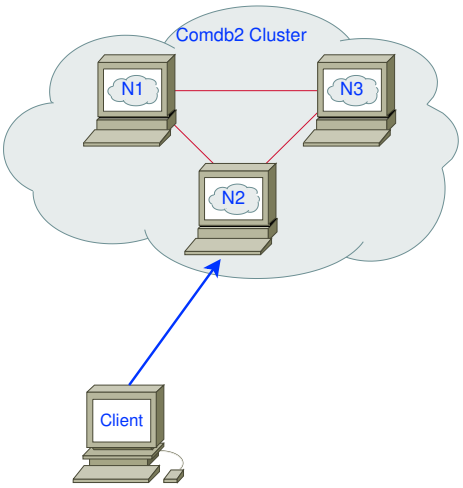
At the beginning of a transaction, the client API generates a tid, establishes a connection against a replicant (if not already connected), and retrieves from that instance a PIT-token corresponding to the current state of that instance. As transactions are applied in the same order on replicants as they are on the master, the snapshot described by this PIT-token is guaranteed to be equivalent to the current or some former state of the data as it existed on the master, and may be used on any instance to recreate the same snapshot so long as that instance has applied transactions at least up to that point. During the course of the transaction, all write statements and the most recent read statement are cached by the client API. It's important to note that the results returned by the most recent read statement need not be retained: rather, the API retains only a count of the rows already retrieved from an in-flight `SELECT` statement.

Upon losing connection with the cluster, the client API reconnects to a different instance and begins a snapshot transaction at the time described by the original PIT-token, as doing so ensures that the client's view of the database on that instance will be identical to its view on the original. The client API then re-issues the transaction's write statements. If the connection was lost while the client was consuming results of a read statement, the client API re-issues the most recent read statement using the cached count to skip (or alternatively, ask the server to skip) records that have been previously returned to the user.

At commit time, the tid is used as a key to store the transaction's result in a replicated *global-transaction-table*. The pre-existence of a tid in the global-transaction-table indicates that the transaction has already executed, and that the current thread should roll back any work that it has done and return the original result to the client. The tid and the global-transaction-table ensure that a transaction will only be executed once should the client API replay a transaction after issuing a `COMMIT`.



(a) Client's Connection to instance N1 gets severed



(b) Client reconnects to instance N2

Figure 2: Connection to Cluster gets severed and Client subsequently reconnects to a different instance.

2.3 Illustrated Example

We describe a concrete example using the event diagram shown in Figure 3. Events in green represent SQL statements submitted by the user. Events in blue represent operations executed on behalf of the user by the client API code. Server responses are displayed in black.

As previously described, upon beginning a transaction, the API generates a unique tid, which is sent to the server along with the `BEGIN` statement. The server responds with the PIT-token. Each write statement of the transaction is cached locally as it is sent to the server. Figure 3 shows a failure occurring after reading the second record of a `SELECT` statement. The client API reconnects to Node 2, begins a transaction at the point-in-time described by the PIT-token, reissues the transaction's write statements, and the most recent incomplete `SELECT`. As the first two rows

have been returned to the application, the client API skips them, returning the third result row to the caller. The client finally issues a `COMMIT` to complete the transaction.

2.4 Considerations

The HASQL scheme as described above works with SQL which behaves deterministically. It requires that each participating instance impose the same implicit order for results not sorted by an `ORDER BY` clause. We note also that the implicit ordering requirement does not exclude server-side parallelism, but concede that any `ORDER BY` clause, either explicit or implicit, may incur a performance penalty.

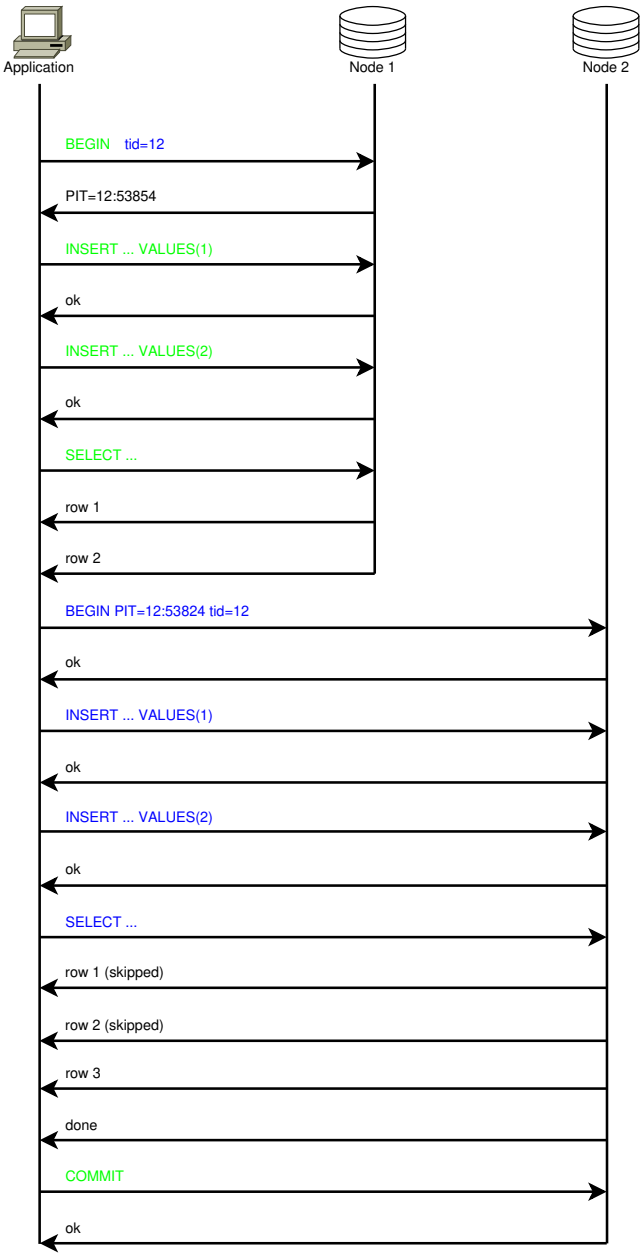


Figure 3: Example HASQL sequence: After connection to Node 1 is severed, Client Application reconnects to Database Cluster Node 2, and continues processing where it left off.

We now consider strategies for handling other non-deterministic SQL. Rather than attempting to address this exhaustively, we consider two situations which can employ a similar strategy; we offer this strategy as a blueprint for how non-determinism might be addressed.

```
BEGIN
DELETE FROM schedule WHERE updatetime <= NOW()
SELECT * FROM schedule
COMMIT
```

Listing 1: SQL using the `NOW()` function

If replayed, the `NOW()` function will execute at a different time on a second machine, and delete a different set of records. A replay which occurs during the `SELECT` could have already returned records which are deleted on the retry. The replay would subsequently skip over *non-deleted* records, as the client API's cached count pertains to the original result set. To address this, we propose that `NOW()` be frozen during the course of a snapshot transaction, always returning the wall clock time of the `BEGIN` issued on the original node. This time can be included as part of an extended PIT-token.

```
BEGIN
UPDATE contestants SET winner = 1 WHERE
    ticketnumber = (SELECT ticketnumber FROM
    contestants ORDER BY RANDOM() LIMIT 1)
SELECT * FROM contestants ORDER BY ticketnumber
COMMIT
```

Listing 2: SQL using the `RANDOM()` function

Because `RANDOM()` can return a different value on the replay machine, the re-issued `UPDATE` statement can update a different record. Should a replay event occur while retrieving results from the `SELECT`, the result set can show that two different contestants have winner set to 1. A system could work around this issue by making note of the `RANDOM()` number generator's current *seed* value at the beginning of a transaction. A replicant which seeds its `RANDOM()` number generator with this value should return the same sequence of random numbers as the original machine. As with `NOW()`, the seed value could be included as part of an extended PIT-token.

3 DEMO

We present a simple interactive demonstration which exhibits the HASQL scheme. We begin by starting a 3-node cluster and presenting a volunteer with three buttons, each programmed to stop and restart the database instance running on one of the cluster nodes. From a separate machine, we open a command-line session and begin a transaction against this cluster. Our volunteer will be instructed to press, at his or her discretion, the kill-and-restart-button corresponding to the machine that is currently executing the transaction. We present this as a simple game, allowing us to demonstrate the HASQL feature in a lighthearted and engaging way.

When the volunteer kills the correct instance, spectators will see the client-session pause briefly as the client API reconnects to a different cluster machine. We encourage our volunteer to kill and restart the active cluster node multiple times as we continue the transaction, being sure to demonstrate HASQL's ability to resume a transaction in the middle of retrieving a result set.

We then perform a second demonstration which is identical to the first, except that instead of killing a single cluster node, we ask

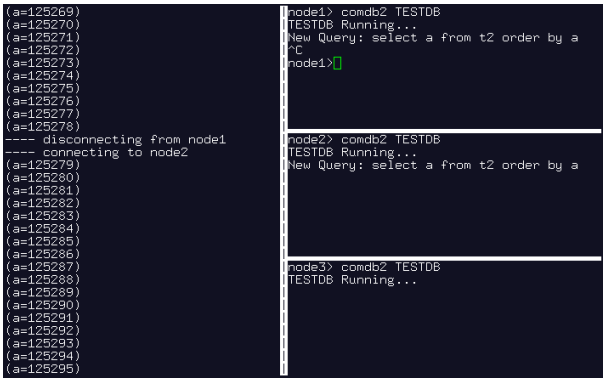


Figure 4: Demonstration of HASQL with additional trace enabled. Left panel shows trace emitted from client application. Right panels show trace emitted by the three Comdb2 cluster nodes. Server instance on Node 1 (top-right) was manually terminated.

three volunteers to kill all three cluster nodes simultaneously. As the cluster restarts, spectators will see again that the transaction is resumed seamlessly.

We proceed to describe our implementation of HASQL, and repeat both demonstrations with additional trace enabled, which allows audience members to witness the sequence of events outlined in Figure 3. In contrast to seemlessly continuing a transaction, this demonstration seeks to exhibit HASQL's underlying mechanism.

We further explore HASQL's behavior by repeating both demonstrations, this time with an increased number of database instances, and with varying levels of background writes. Spectators will observe that HASQL's performance is unaffected by the increased cluster size, but that it is directly impacted by external writes to a table which an HASQL transaction reads. We use this as a starting point for a discussion of Comdb2's implementation of point-in-time snapshot isolation and other architectural features of Comdb2.

4 FUTURE WORK

A transaction which survives a machine crash naturally takes longer to complete. While this does not effect the correctness of a read-only query, the increased transaction time increases the likelihood that a write transaction, T1, will fail, as it allows greater opportunity for a competing transaction to write in the space of T1's write-sets. This is a small concession to make, as prior to HASQL, a machine crash would certainly cause T1 to fail, and there are a great number of non-intersecting write loads that would permit T1 to commit.

Future work includes finding ways to minimize the amount of time it takes to restore a partially completed transaction. We observe that the slowness is most pronounced when a machine crash occurs after a client has retrieved a substantial part of a large result set which must be skipped.

We could gain substantial improvement by maintaining simultaneous connections to multiple cluster machines, using the original PIT-token from the *primary* connection to establish one or more *secondary* connections. Each SQL statement would be issued to the primary and to the secondary handles in lock-step. In the normal case, the redundant sessions are essentially wasted computing power, but as hardware resources continue to become cheaper, this may eventually be a valid concession to make.

5 CONCLUSION

HASQL's contribution is one of resiliency: application developers need not know or care if the underlying system has experienced a critical error. We believe this is superior to other failover schemes, where a machine failure, in addition to failing all outstanding transactions, can stall clients for several minutes before a failover machine is available. An API return code which does not designate the success or failure of an operation places a disproportionate burden on the application programmer in answering a question which would be more appropriately addressed by the database system itself. Though we concede that this is unavoidable at times, HASQL addresses a significant subset of these errors. As hardware is guaranteed to fail, it is the responsibility of system designers to minimize the impact of failure. HASQL demonstrates an intelligent way to utilize increasingly less expensive hardware to create more robust service.

REFERENCES

[1] Bathtub curve. https://en.wikipedia.org/wiki/Bathtub_curve. Accessed: 2017-11-17.

[2] Gray, J. N., and Siewiorek, D. P. High-availability computer systems. *IEEE Computer 24* (1991), 39–48.

[3] MySQL. Mysql high availability. https://www.mysql.com/products/enterprise/high_availability.html. Accessed: 2017-11-17.

[4] Oracle. Application continuity. https://www.oracle.com/database/technologies/high-availability/app-continuity.html. Accessed: 2018-11-28.

[5] Oracle. Oracle database high availability. https://www.oracle.com/database/high-availability/index.html. Accessed: 2017-11-17.

[6] PostgreSQL. Postgresql high availability. https://www.postgresql.org/docs/8.3/static/high-availability.html. Accessed: 2017-11-17.

[7] Scotti, A. Adventures in building your own database. In *In All Your Bases Conference* (November 2015).

[8] Scotti, A., Hannum, M., Ponomarenko, M., Hogea, D., Sikarwar, A., Khullar, M., Zaimi, A., Leddy, J., Angius, F., Zhang, R., and Deng, L. Comdb2: Bloomberg's highly available relational database system. *PVLDB 9*, 13 (2016), 1377–1388.