Comdb2
Bloomberg's Highly Available Relational Database System

Alex Scotti, Mark Hannum, Michael Ponomarenko, Dorin Hogea, Akshat Sikarwar,
Mohit Khullar, Adi Zaimi, James Leddy, Rivers Zhang, Fabio Angius, Lingzhi Deng
Bloomberg LP
{ascotti, mhannum, mponomarenko, dhogea, asikarwar1, mkhullar, azaimi, jleddy4,
hzhang320, fangius, ldeng}@bloomberg.net

ABSTRACT

Comdb2 is a distributed database system designed for geographical replication and high availability. In contrast with the latest trends in this field, Comdb2 offers full transactional support, a standard relational model, and the expressivity of SQL. Moreover, the system allows for rich stored procedures using a dialect of Lua. Comdb2 implements a serializable system in which reads from any node always return current values. Comdb2 provides transparent High Availability through built-in service discovery and sophisticated retry logic embedded in the standard API.

In addition to the relational data model, Comdb2 implements queues for publisher-to-subscriber message delivery. Queues can be combined with table triggers for time-consistent log distribution, providing functionality commonly needed in modern OLTP.

In this paper we give an overview of our last twelve years of work. We focus on the design choices that have made Comdb2 the primary database solution within our company, Bloomberg LP (BLP).

1. INTRODUCTION

In recent years there has been a renewed interest in developing large-scale relational databases. This trend was later named *NewSQL* [1] to contrast with the NoSQL [19] designs, which sacrifice full transactional support and consistency for maximum scalability.

In [31] Stonebraker et al. summarized the limits of NoSQL architectures, remarking how ACID properties are still a strong requirement in all mission critical scenarios. These include financial and in-order processing applications.

BLP has served the financial industry world wide since the early 80s and presents one example. The company has dealt with large volumes of data since before RDBMS were being commercialized and has a long history of building its own database systems in-house.

Proceedings of the VLDB Endowment, Vol. 9, No. 13

The latest version, named Comdb2, has been used in production since 2004 and shares many of its features with the latest enterprise solutions. The purpose of this paper is to give a detailed overview of Comdb2 and start a healthy collaboration with the academic and the open-source communities [1].

A crucial point in the design of Comdb2 was the early adoption of Optimistic Concurrency Control (OCC) [18]. OCC is a lock-less protocol that guarantees a higher degree of parallelism in low-contention workloads frequently found with large datasets. The performance gain occurs from locking rows only at the time of commit rather than for the duration of the entire transaction. In a distributed scenario this means that every machine in the cluster can fully serve a transaction without coordinating with others. Only at the time of commit does validation occur in order to guarantee consistency. OCC matches the needs of modern Online Transaction Processing (OLTP) and is a recurrent theme in the latest research work from academics [8, 3] and professionals [29, 23, 6, 30]. OCC works well in distributed systems since the lock managers found in locking based systems often prove difficult to distribute.

While high performance was obviously one of the goals in developing Comdb2, high-availability was and still remains the main concern. At BLP it is essential to have a database that can: (1) synchronously replicate across data-centers, and (2) remain available during any type of outage or maintenance. It is important to remark that while the disruptiveness of schema changes is often used an argument for NoSQL architectures [28], a relational system need not suffer from such disruption. Comdb2 is able to perform every form of schema change live and with a minimum performance drop. In many cases it also supports *instantaneous schema changes* allowing for in-place updates of rows without rebuilds.

This paper is organized as follows: In Section 2 we describe the company and what drove us to build our custom RDBMS. In Section 3 we proceed with a high level description of the system design. In Section 4 we describe the details of our implementation. In Section 5 we discuss common performance tuning options of the system. In section 6 we show a direct comparison of performance with another system. In Section 7 we discuss ongoing challenges faced by the Comdb2 system and work in progress. We conclude with a final discussion in Section 8 and acknowledgments in Section 9.

[1]Comdb2 will be published under an open source license in 2016

2. BACKGROUND

Bloomberg is considered the world's most trusted source of information for financial professionals. At BLP, the central product is *The Terminal*, a proprietary software platform which gives subscribers real-time access to financial data, news, and analytics. Users can obtain price quotes and send trade-compliant emails and instant messages. Institutional investors use The Terminal as a trading platform to move billions of dollars on a daily basis. BLP also provides a real-time market data feed, delivered using a highly-resilient custom architecture.

In 1981, when the company was founded, BLP chose a Perkin-Elmer system running OS/32 as its computing platform. At the time, there was no database software available for this architecture. Over the years BLP has developed several versions of a database system referred to as the Common DataBase (Comdb). We began work on Comdb2 in 2004 when it became apparent that to meet the needs of the industry we needed a more scalable and resilient DBMS. We built Comdb2 with the following concerns in mind: scalability, high availability, and a relational and fully transactional model.

Scalability was a concern when designing Comdb2 as we found scaling vertically had become impossible. Instead of relying on high-capacity mainframes, we chose to engineer large clusters of commodity hardware. For a database, this implied replication over multiple machines, posing the problem of latency vs consistency. Many recent projects adopt *eventual consistency* (EC) [34, 35] in order to minimize write latency. This model states that every machine in the cluster will show updates to a row in the order that they occurred, and that in the absence of new updates at some point in time all machines will converge to the same value. BLP has built its reputation upon accuracy and timeliness. Neither of these qualities is guaranteed by EC. An example of why this does not work for us is the possibility of showing two different prices of a stock to two different customers at the same time. At BLP, databases that work as sources of truth must support *strict consistency*, leaving the decision to lower the consistency requirements to the final user when appropriate.

High availability is also fundamental. Discontinuities directly impact the daily routine of the financial markets [13]. Databases must replicate across several data-centers and their deployment must be *elastic*. Clusters can grow, shrink, and even migrate from one architecture to another without going offline.

While many migrated away from the **relational model** to NoSQL architectures, we have gone the other way. Earlier versions of Comdb stored data in a key-value fashion, where keys and values are any tuple of primitive types. Storing data this way left the programmer with the burden of fetching data efficiently even when this demanded following cross-references or running sub-queries. Modern query planners perform this job more efficiently while preventing users from rewriting the same algorithm multiple times.

Lastly, Comdb2 offers **full transactional** support whereas NoSQL usually offers only micro-transactions (MT). The MT philosophy is to enforce atomicity of write operations only at the row or column level [32, 7, 4, 24]. Some other systems, such as [28], offer transactional support over collections of hierarchically related documents. These models do not fit our workloads which often deal with the need to transactionally update multiple rows in multiple tables.

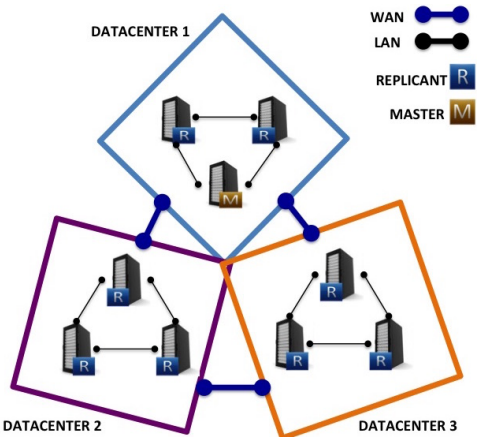

Figure 1: Interconnected nodes of a Comdb2 cluster

3. DESIGN

Comdb2 was designed from the onset to be a highly available relational database system. Many decisions were made with this in mind. Optimistic Concurrency Control was chosen for its low overhead and high performance in a distributed environment. Synchronous replication was chosen for its ability to present the illusion of a single system image. A coherency model was developed along those lines accounting for failures of nodes and guaranteeing consistent reads of latest values.

A Comdb2 cluster consists of a master node and several interconnected replicant nodes. Clusters are built for geographical replication. Nodes in a cluster are partitioned into *rooms* which map to datacenters. Communication between applications and the database occurs within a data center, and communication between data centers occurs only at the time of transaction commit.

3.1 High Availability SQL

Comdb2 provides a highly available experience to application programs similar to [9][2]. Service discovery is provided through the standard APIs which abstract the physical location of the data. The same mechanism provides the ability for an application to reconnect to alternate locations in cases of failure. The HASQL mechanism allows for seamless masking of failures for in-flight SQL transactions. At the beginning of a transaction, the server sends the client a point-in-time token corresponding to the transaction's starting *Log Sequence Number* (LSN), described in detail later in this paper.

The API retains its point-in-time token, and uses it when seamlessly reconnecting to alternate nodes under failure. It instructs the new node to construct a snapshot corresponding to the point in time contained in the token, and then executes the SELECT which was executing at the time of the failure. By keeping track of the number of rows already traversed, the API is able to return the next row in the result set. Writes inside an open transaction are buffered inside the client API and shipped to the new node transparently upon

[2]Unlike Comdb2, Oracle TAF does not support writes

reconnect, allowing for transparent failure masking of INSERT, UPDATE, and DELETE statements. Transactions are supported as well. The OCC system does not perform any actual updates until a transaction commits, so retrying is possible before commit. Global sequencing is used from the client to server allowing for retries and detections of replays. The system does not return errors to applications when a node in a cluster fails. Instead all forms of errors resulting from node failures are masked by transparent retries.

3.2 Isolation Levels

Comdb2 allows applications to chose from a weaker version of ANSI **read committed** to an ANSI-compliant **serializable** implementation depending on their correctness and performance requirements.

Block is the weakest isolation level in Comdb2. This is the only isolation level available to non-SQL clients. In this isolation level, only committed data can be seen. Comdb2 never offers the ability to see uncommitted data. This isolation level makes no effort to mask the underlying OCC nature of Comdb2, and as such, reads within a transaction are unable to see uncommitted writes that have occurred within the same transaction. Many applications are able to function properly with the phenomena present in this level.

Read Committed behaves like *block*, but additionally allows clients to read rows which have been written within the current transaction. Reads merge committed data with the current transaction's uncommitted changes. Changes local to the transaction are stored in data structures described in Section 4.

Snapshot Isolation implements Snapshot Isolation as defined in [2]. Private copies of rows are synthesized as needed when pages are read which have been modified after the start LSN of the transaction.

Serializable implements a fully serializable system. As an OCC system, any transaction which would result in a non serializable history is aborted at commit time and returns a *non-serializable* error. Transactions do not block or deadlock. Serializable isolation adds additional validation to Snapshot Isolation in the form of read-write conflict detection.

3.3 Optimistic Concurrency Control

Concurrency control models fall into two categories: optimistic and pessimistic. An *optimistic* model anticipates a workload where resource contention will not often occur, whereas a *pessimistic* model anticipates a workload filled with contention.

Most commercialized database engines adopt a pessimistic approach whereby rows are manipulated under a *safe* lock, specifically: (1) a read operation will block a write, (2) a write will block a read, and (3) multiple reads will hold a "shared" lock that blocks any write to the same row. In a classical two-phase locking (2PL) scheme every acquired lock is held until the transaction is committed or aborted, hence blocking every transaction that tries to work on the data under a lock. Even MVCC based systems acquire transaction duration write locks on rows being modified while an OCC system never obtains long term write locks.

In an OCC system, transactions are executed concurrently without having to wait for each other to access the rows. Read operations, in particular, will have no restrictions as they cannot compromise the integrity of the data. Write operations will operate on temporary copies of rows. Since persisting the transactions as they are executed would likely violate the ACID properties, the execution of each transaction has to be *validated* against the others.

Comdb2 uses a form of Backwards Optimistic Concurrency Control (BOCC) [14] with concurrent validation. Two distinct validation phases prevent anomalies such as *overwriting uncommitted data*, *unrepeatable reads* and *write skew*. This is a hybrid system using locking for some functions, while adhering to a more traditional OCC approach for others.

In order to detect Write-Write conflicts, Comdb2 uses a form of deferred 2PL to allow for concurrent validation without a critical section [33]. This is based on the notion of a *genid* - **GEN**eration **ID**entifier - associated with each row. **Every modification to a row changes its genid**, and **genids can never be reused**. Genids are *latched* - i.e. remembered - during the execution of a transaction when rows are modified, and later validated at commit time using 2PL on the rows. The structure used to record such modifications is referred to as the Block Processor Log (bplog).

As the genid forms a key into the data internally, the existence of a genid in the system is sufficient to assert the existence and stability of a row before committing. Comdb2 incurs no extra overhead in recording all overlapping write sets for validation, as a standard Write-Ahead Log (WAL) protocol demands that write sets be logged already.

Read-Write conflicts are addressed by non-durably recording the read set of a transaction as degenerate predicates consisting of rows, ranges and tables. During the validation phase the overlapping write sets from the WAL are checked for conflicts against the transaction's read set. Validation runs backwards in several phases.

The ultimate commit operation occurs in a critical section but pre-validation ensures that duration will be brief. Replicants running a transaction are able to begin validation concurrently up to the LSN present on that node. The validation burden then moves to the master in a repeated cycle of validations outside the critical section. The critical section is entered for final validation once pre validation is near enough to the current LSN as determined by a tunable.

3.4 Replication

A transaction in Comdb2 goes through several distinct phases on various nodes of the cluster, as shown in Fig. 2. In the initial phase, the client connects to a geographically close replicant (Fig. 2a), typically in the same data center. The interactive phase of the transaction (SELECT, INSERT, UPDATE, DELETE operations (Fig. 2b)) occurs entirely on that replicant. We will refer to this as the OCC phase of the transaction lifecycle as no locks are acquired. During the execution of this phase write operations are *recorded* for purposes of later execution and validation. This recording occurs on the bplog which is continually shipped to the master and buffered. When the client application finally COMMITs, the master begins the second phase of the transaction lifecycle (Fig. 2c). This is a 2PL phase in which operations are both written and validated to detect OCC read-write or write-write conflicts (Fig. 2d). The master generates phys-

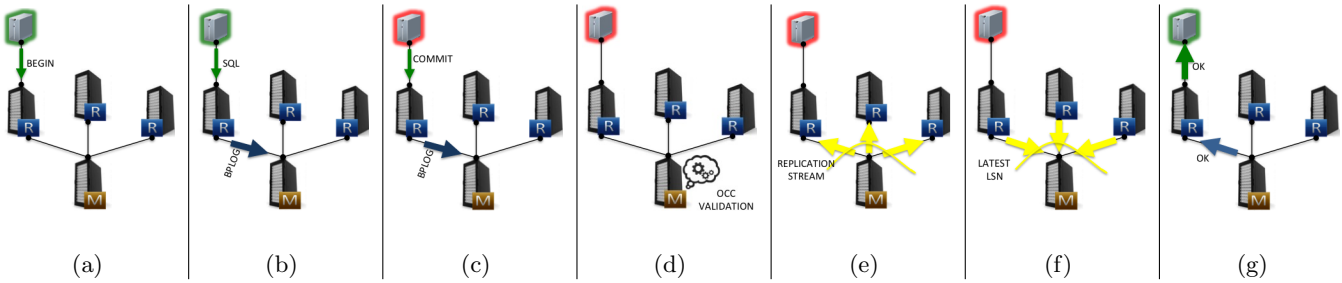

Figure 2: Life cycle of a transaction

iological log records describing changes it makes to B-tree pages. It then duplexes this log to both its local log file (for local durability) and to the network (for high availability) (Fig. 2e). At the end of processing the transaction described in the bplog (after local commit and after releasing locks), the master then synchronously waits for all replicants to acknowledge that they have successfully processed this transaction (Fig. 2f). Each makes the changes described. There is no consensus at this stage, only a set of replicants driven by a master. At this point, the originating application will remain blocked. Only after the master receives acknowledgement of cluster-wide durability will it respond to the blocked session on the original database node. That node will then respond to the originating application with a successful return code (Fig. 2g).

Techniques described later allow safe forward progress without endless blocking on all nodes being correctly operating and available. Additional techniques are used to make this form of synchronous replication lighter weight than it first appears.

3.5 Distributed SQL

Comdb2 allows for remote access of any database instance in the **Comdb2 cloud** to be transparently used as if the remote table were a local table. Extensive changes were made to SQLite [15, 17] to allow for efficient predicate filters to be pushed to remote nodes. The planner pushes generated SQL to remote nodes for execution, and results stream back in parallel for local processing. The system operates without any pre configuration, dynamically learning schemas and index statistics from remote databases as the statements are parsed. The remote information is versioned and locally cached, and re-acquired when remote servers update it. The planner has access to both local and cached remote schema and associated index statistics to select a properly optimized execution plan. In effect all tables in all the Comdb2 instances in a single Comdb2 deployment operate as a single large database.

Tables in Comdb2 exist in a global namespace of the format `<databasename>.<tablename>` though support for preconfigured "aliases" allows the database administrator to permanently map a remote table into the namespace of a local database.

3.6 Stored Procedures, Triggers, and Queues

The stored procedure language in Comdb2 is a customized dialect of the Lua programming language. Extensions include among others, numerous additional base data types matching all database provided types, a strong typing system, pthread-like thread support, and parsers for JSON. Stored procedures run on the server side rather than the client. The Lua engine has full access to the relational engine and can operate on rows in the same way a client can using SQL. Moreover, a stored procedure can return rows to a client the same way a SELECT statement would.

Stored procedures can be used in combination with triggers to handle table events such as: add, update, and delete of a row. Update events can be filtered based on the columns that have been updated. If a table has columns *a,b and c* the trigger can be set to fire only when column *b* has changed. When a trigger fires an internal queue is written to from inside the relevant transaction. A stored procedure is called to consume this event. The execution of the procedure is a transaction which includes consumption from the queue, guaranteeing at most once execution of the procedure for each event.

Triggers can also be *external* in that they feed calling SQL applications rather than an internal stored procedure. This functionality is used for an application needing more involved logic than can be readily placed inside the database. The system provides High Availability by allowing for multiple consuming SQL statements to be issued on the same condition from multiple client machines while providing at most once delivery.

Triggers, stored procedures, and queues can be combined to create a form of asynchronous eventually consistent replication. Logical logging is used to describe all changes to rows, which are enqueued and multicast once the transaction has become durable within the cluster. Any interested party picks up this multicast stream and feeds a local Comdb2 instance, which essentially acts as a local cache of the real database.

4. IMPLEMENTATION

In this section we present the building blocks of Comdb2. Each module has responsibilities within the single database instance and within the cluster.

4.1 Storage Layer

Comdb2 storage is based on an extensively modified version of BerkeleyDB 4.2. In particular: the *memory pool (mpool)*, for paginated file access, and the B-tree implementation.

Throughout the years we integrated the latest solutions for I/O and concurrency into the mpool. Extensions include direct I/O support, parallel and coalesced writes, and an optional double buffering scheme to prevent torn pages [26]. Mpool can trigger cluster-wide page faults under certain circumstances as explained later in this section.

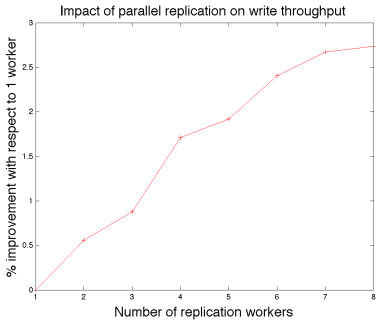

Figure 3

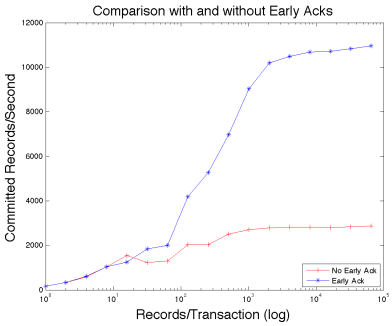

Figure 4

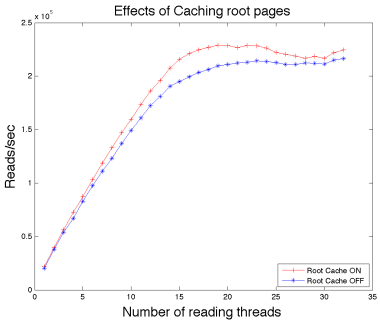

Figure 5

Comdb2 uses B-trees to store every type of data: rows, indexes and blobs. However the B-tree layer from BerkeleyDB knows nothing of these. From the storage perspective a B-tree simply maps a key to a value, both as a raw sequence of bytes. Each B-tree is stored in a separate file and multiple B-trees form a table as explained later in the paper. Some relevant changes that have been added to the original implementation of BerkeleyDB are as follows:

4.1.1 *Row Locks*

The original version of BerkeleyDB used page locks instead of row locks. We added support for row level locking implementing an ARIES style system [22] using logical undo logging and page level micro transactions.

4.1.2 *Prefaulting*

Even though solid-state drives (SSD) have essentially eliminated the seek time penalty, the speed gap between main and secondary memory is still disproportionate. Every *page-fault* sacrifices thousands of CPU cycles. One way of tackling this problem is to anticipate foreseeable access to storage so that data can be available before being requested. This practice is commonly known as *prefaulting* or *readahead*.

B-tree readahead is triggered whenever a B-tree is traversed sequentially: pages are read in advance in order to anticipate future requests. This is not trivial in a B-tree implementation that normally does not have upwards pointers to parents from leaf nodes. Each *cursor* that moves on the tree maintains partial information about its descent to the leaf nodes. This enables preloading of hundreds of pages in parallel without the need to lock inner nodes of the B-tree. Pages are versioned so the cursor is able to identify any inconsistency caused by later changes to the tree structure.

Local prefaulting occurs during transaction execution. High level bplog operation such as "update" or "add" which operate on a row are decomposed into low level operations which "form" and "find" the keys required by the indexes in the background concurrently. These B-tree read requests have the effect of faulting in index pages in parallel, racing with the original transaction.

Remote prefaulting occurs on the master node at time of committing a transaction and propagates through the cluster over a lossy channel (UDP). It transmits the sequence of pages accessed while writing the data. This allows replicants to load pages into memory before they are requested by the replication stream.

4.1.3 *Lock avoidance*

Root pages in a tree are traversed during every descent to the leaf level. Consequently they easily become a concurrency bottleneck. Taking advantage of the fact that root pages of large B-trees change rarely, each working thread maintains it's own private copy that can be accessed without holding locks. We use a simple protocol for invalidation when the original page changes. Fig. 5 shows the performance improvement in a read-only scenario where on the x-axis we see the number of reading threads and on the y-axis the overall throughput. As more readers are added, the contention becomes evident, until the system saturates in the right side of the chart.

4.1.4 *Compression*

Comdb2 B-trees support both page and value compression which allows a trade-off of CPU cycles for less disk I/O. The compression algorithms used are: (1) zLib [10], (2) Lz4 [5], (3) Comdb2 Run-length Encoding (CRLE). Each algorithm offers a different performance to compression ratio. The database is able to sample the data and suggest the ideal compression type. CRLE is novel in that it is hand built to understand the binary representation of Comdb2 datatypes, and captures common patterns which occur when multiple consecutive columns are set to the same value. Page compression occurs on indexes using a form of prefix compression. The compression allows for items to be individually extracted without decompressing an entire page or operating on a "stream", as was added in later versions of BerkeleyDB. The remaining suffix entries are then further compressed individually with CRLE.

4.1.5 *Concurrent I/O*

The buffer pool layer of BerkeleyDB was modified to allow for concurrent writes for the purpose of higher throughout when flushing to disk. Under a sustained heavy write workload, the buffer pool can become completely dirty and the rate limiting step becomes the rate of flushing dirty pages. Multithreaded flushing works in tandem with filesystem support for DirectIO to concurrently write to the same file. Additionally, Comdb2 merges adjacent pages when writing to issue fewer system calls of larger writes.

4.2 Replication

In Comdb2, the replication logic exists solely at the storage layer. This is possible because validation and committing are coordinated by a single master that always attempts

to modify its version of the storage before ordering the replicants to do the same. Changes are forwarded to the entire cluster as a list of logical descriptions of the modified pages. The progress that each machine has made with respect to the master is measured by Log Sequence Number (LSN), a monotonically increasing number that advances as changes are performed to the storage. Upon receiving a new segment of the replication stream the replicant: (1) applies the changes and (2) acknowledges its progress with its latest LSN.

By default, the master waits for every node to acknowledge the latest LSN before considering the changes permanent, although consensus and its quorum can be tuned as explained in Sec. 5. Only after these nodes acknowledge this transaction does the blocked calling application receive a successful response.

Comdb2 provides for a globally consistent reads-follows-writes model. This means that after a write has been reported committed to the originating client application, the write will be immediately visible to that application, as well as any other application reading from any node in the Comdb2 cluster. Unlike a 2PC model, which provides similar distributed semantics, the Comdb2 model proceeds safely in cases of failure of one or more nodes in the cluster during commit without blocking.

As a single master system, the final stage of a commit happens in only one place. Unlike 2PC, no *consensus* is actually required to commit: by the time the other nodes are involved, the commit has already happened. This allows for decoupling of the nodes in a manner looser than 2PC, but causes two potential problems. We first needed to solve the problem of ensuring that only the latest values can be read on all nodes, especially during failures. A more subtle problem encountered during the development of Comdb2 was the possibility of *dirty reads* under certain failure scenarios despite each node locally disallowing this phenomenon.

4.2.1 Replication stream delivery

Using synchronous replication as the basis for reads-follows-writes semantics presents performance concerns. In the most straightforward implementation, this would mean a transaction has been both durably logged and applied to become visible before it can be acknowledged. In practice, both of these techniques result in poor performance. By default Comdb2 considers a transaction durable without writing anything to disk, although this behavior can be changed upon the user's requirements. We consider this a form of **network commit** in that durability for practical purposes is achieved by in-memory replication to multiple machines in geographically distributed locations. In addition, the acknowledgement occurs before any parts of the transaction have actually been applied, even in-memory. We call this optimization **early ack** because it allows a replicant to acknowledge a transaction before processing the data typically needed to ensure reads-follows-writes consistency. The system obtains all locks needed to guard against the possibility of a client reading the previous state of the database before acknowledgment. The replication stream contains all locks needed and presents them in the commit record. Comdb2 asynchronously writes the log files and asynchronously updates in-memory structures while still ensuring that only the most recent value is ever visible to any client after a successful write. Fig. 4 shows the performance improvement obtained using early-ack in a write intensive scenario. As expected, when more rows are modified by the transactions (x-axis) the gains of early-ack become more apparent.

A second performance problem that a synchronous log based replication system encounters comes from the very nature of logging. Logging converts a concurrent set of inputs into a sequential output where every event happens in a strict order. In essence, parallelism is converted into serial execution. Feeding replicants the log stream has the effect of replicants running much slower than the master as the concurrency has been eliminated when processing the log entries in order. Comdb2 solves this by **processing the log concurrently**. Using the same set of locks embedded in the commit record previously described, Comdb2 allows concurrent transactions for execution on replicants when no conflicts are possible. Concurrency within a transaction is also extracted. For example, an INSERT on a table with two indexes can safely modify the one data and the two index structures concurrently without conflicts. Fig. 3 shows the benefit of processing the log concurrently in a workload benefitting from this technique. In this simplistic case rows were being written to many different tables. As more replication worker threads are added (x-axis) the the overall throughput continues to increase.

4.2.2 Coherency: global reads-follows-writes model

Comdb2 ensures that the latest value can be read from any database node after a successful write from any application. We refer to this as our coherency model. All nodes are in one of two states, either coherent or incoherent. A coherent node is allowed to begin new transactions (including serving a simple read) while an incoherent node rejects all requests. The arbiter of this state is the master. The master treats nodes differently based on the coherency state, waiting for acknowledgement from coherent nodes only. A node transitions between coherent and incoherent when a wait for acknowledgement has failed or when the master deems the performance of that node to be sub par. The underlying mechanism is one of **short term leases** which are continuously issued by the master. The lease is an absolute timestamp at a point in the future well beyond any allowed time skew between nodes. Transitioning a node to an incoherent state requires revoking a lease, which is accomplished by no longer issuing a lease to the errant node, and blocking commits for longer than the lease time. This ensures that the lease has expired. This mechanism is fail-safe in that a node that is no longer in contact with a master will not be able to serve stale reads, as it will no longer be receiving leases. A similar reverse protocol exists to dynamically transition nodes that have become healthy back into the coherent state and allow them to be safely used.

4.2.3 Eliminating dirty reads

Allowing for failures during commit exposes a potential for dirty reads even with all nodes in a cluster running with serializability locally. Let us consider the scenario where the commit of a transaction t_1 from client C_1 to master M is acknowledged by machine S_1, but has not yet reached nodes S_2, S_3, S_4, S_5. Concurrently application C_2 issues a read to S_1 which reads the latest update from t_1. A failure of nodes M and S_1 at this point results in $S_2..S_5$ electing a new master amongst themselves. After a new master has been elected the cluster will no longer contain the result of

transaction t_1 even though C_2 has read this data and may have made decisions based upon it. Comdb2 introduces the notion of a **durable LSN** in the cluster to handle this.

When a transaction begins on the replicant it begins at the point in time of the current LSN on that node. It is possible that at that instant the LSN is ahead of the LSN of some other nodes. In essence this node is running slightly *in the future*, though obviously not ahead of the master. The goal is ensuring that this transaction does not begin in a future which will never occur if a failure causes a loss of data which committed to some nodes. The begin operation stalls until the durable LSN received is greater than or equal to the start LSN of the transaction to ensure the read is from durable storage.

In Comdb2 master failures and election are handled in a standard Paxos fashion. When a master is lost the replicants agree on what the latest LSN is and elect a new master among those having the latest version of the storage. Heartbeats are used to detect loss of master.

4.3 Cdb2 Layer

In Comdb2, the SQL engine sees tables and indexes as whole entities and knows nothing of concurrency. The cdb2 layer does all the heavy lifting needed to hide the physical representation of data and the artifacts caused by concurrency.

4.3.1 Data organization

The cdb2 interface exposes the basic functionalities for search, scan, and update on a table. Each table is stored across many files, each of which contains a B-tree. For the sake of simplicity we will first assume that a table has one B-tree to store its rows in addition to one B-tree per index. We will further refer to these respectively as data and index files.

As is common with conventional row-store databases, a data file stores the rows of a table indexed by a primary-key, and index files map the values contained in some selected columns to a primary key to reference a row.

In Comdb2 however, the primary key is always system assigned. Specifically, it is the 64 bit genid that forms the basis of OCC in Comdb2. The most significant part of a genid is always used as the primary key of a row, with high bits masked out before storage. The binary structure of a genid is the following:

counter(48)	update-id(12)	stripe-id(4)

The 48 most significant bits are a monotonically increasing *counter*. This is not a point of contention as allocations are not transactional. They can be lost, and no effort is made to record allocations. Recovery can trivially begin at the next value by seeking to the highest values in the data B-trees.

The next 12 bit section is the *update-id*, which is actually stored in the payload, and not the key, of a data B-tree. The keys of a data B-tree have the update-id masked off to avoid the effect of an update causing a physical relocation of the entry in the B-tree. After 2^{12} updates to a row, a new genid is allocated and relocation occurs.

The last 4 bits identify the *stripe* of the data, explained later in this section.

In a data file, the payload consists of a 56 bit row header followed by the row itself [3]. The structure of the row header is the following:

length(28)	update id(12)	schema version(8)	flags(8)

The length represents the uncompressed length, used to ease memory allocation burdens. The schema version indicates the format of the schema used to store the payload. Comdb2 supports instant schema changes meaning that the table structure can be changed without converting the data stored in it. Whenever a row is read from a data file the database checks if the schema is current. If not, the system internally converts the row in memory before using it, optionally triggering an asynchronous rebuild of the row on disk. The flags record options such as the type of compression used.

The columns themselves are encoded within a row in a machine-agnostic format which allows for memcmp comparisons to be performed on any schema. For performance reasons and design simplicity the comparison routine of the B-tree has no knowledge of schemas or data types.

The payload of an index file is simply a genid. Each index entry points to one data entry, with duplicate index values handled by internal suffixes on keys to differentiate entries. Indexes must contain full genids including the update-id portions to allow for proper OCC validation of plans that operate on covering indexes without reading data.

In practice, data files are split into multiple B-trees (typically eight) which we refer to as *stripes*. This alleviates a concurrency hotspot when writing to the right side of the B-tree. In order to offer a uniform view of the storage, cdb2 implements its own abstraction of a tree, allowing it to transparently merge underlying B-tree structures.

4.3.2 Isolation Levels

The cdb2 trees implement isolation by merging temporary tree structures referred to as shadow trees. These are:

dt-shadow contains the rows that the transaction **should** see, either because the transaction has added the rows to the system or because the rows were present in the system when the transaction started and the isolation level was *snapshot* or higher.

skip-shadow contains the rows that the transaction **should not** see, either because the transaction deleted the rows or because the rows were added to the system after the transaction started using snapshot isolation.

ix-shadow contains the shadow trees of the indexes. There is no need for index skip trees as indexes use the data skip tree for this purpose.

These temporary data structures are kept empty in the **block** isolation level as it does not allow for transactions to read the results of their own uncommitted data. In **read committed** they are filled by the transaction itself, which is allowed to look at the temporary storage as well as the already persisted data. In **snapshot** isolation the shadow trees are filled both by the transaction and replication stream.

Fig. 6 offers a graphical example. On the left side we see the interleaved serializable execution of transactions T1 and

[3]Excluding its blobs that are stored in a dedicated B-tree

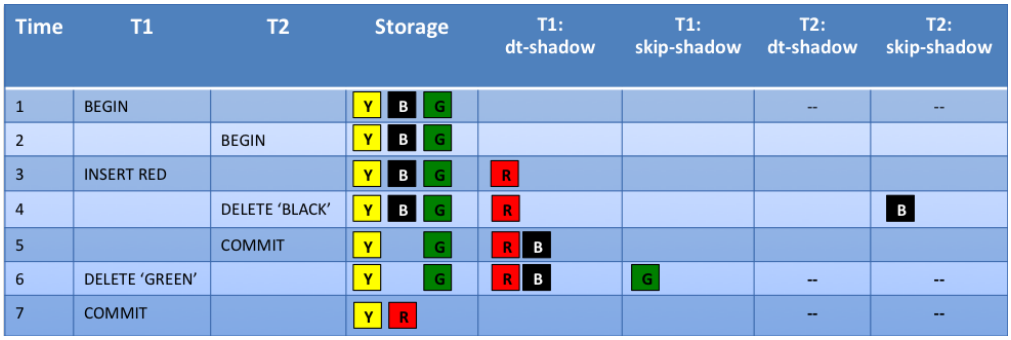

Figure 6: Two interleaved transactions represented internally

T2. On the right side we see how storage and shadow-trees are modified over time. When the transactions begin (time 1 to 2) the storage is untouched and the shadow trees are empty. At time 3, T1 inserts 'RED' causing its dt-shadow to store the value to reflect the change in T1's own view of the storage. Analogously, at time 4, T2 deletes 'BLACK' filling its skip-shadow with it. Note this last operation does not cause any change on T1's shadow trees. Only at time 5 when T2 commits, causing the storage to change, does T1 need to insert 'BLACK' in its skip tree in order to maintain its snapshot. T1 proceeds removing 'GREEN' and finally committing at time 7 making the changes effective on the storage.

This process is costly but optimized by deferring work until a query visits a modified portion of the storage. Comdb2 tracks the LSN at the time the transaction began. When it reads a page of the storage, the LSN on the page is compared with the one associated with the transaction. If the page contains a higher LSN the replication log is used to logically undo all changes on that page by processing the undo records into shadow structures. Auxiliary data structures allow the processing of only the necessary undo records for that page.

The serializable isolation level uses all the shadow trees presented in this section and additional more complex data structures for efficient predicate validation.

4.3.3 BPLog

The bplog is a logical log which is emitted during the OCC phase of transaction execution. For example, an "INSERT" results in an "add record" being logged, and a "DELETE" results in a "delete genid" being logged. The log is a non-durable stream of events (recorded transiently on the initiating node) which is shipped to the master for both execution and validation. The master node executes the steps in the bplog using 2PL, and performs validation of all mentioned genids to assure transactions commit changes to the same rows initially referenced. Execution of the bplog in turn generates the WAL feeding the replication process of Comdb2. The time in which locks are held through 2PL in this model is minimized. While an application may keep a transaction open and continue to write, no locks are being held, as Comdb2 is in its OCC phase. Only once the application commits and relinquishes control does the 2PL phase begin, running as fast as the system will allow.

The bplog is resilient to master swings. If a new master is elected while a transaction is being executed, the bplog is resent to the new master by the replicant that generated it and the execution process will be restarted. Internal usage of globally unique identifiers allows for safe replay detection, resulting in at most once execution when resending bplogs.

4.3.4 Schema changes

In contrast to many other RDBMS, schema changes are implemented in Comdb2 so that the table being modified is always available for reads and writes regardless of the type of change. No machine in the cluster has to go offline. In many cases schema changes are applied instantly with no physical rebuild. The system prefers non schema change transactions when conflicts occur, aiming for little to no impact to the production workload when a schema change is underway. A unique feature of Comdb2 is the notion of *declarative schema change*. As opposed to the low-level model presented through most pure SQL systems, Comdb2 takes the approach of allowing the user to specify a complete new schema (including many changes such as adding, removing, changing data types of columns, and the same for any number of indexes and constraints) and having the "planner" determine the most efficient way to get from the current schema to the new schema. A schema in Comdb2 DDL includes a full definition of the table including all columns, indexes, and constraints.

Tables do not need to be rebuilt when columns are added or when columns are promoted to compatible larger types. Some compatible type conversions are a short integer to a long integer, or any type of number to a string. Incompatible conversions are a long integer to a short integer or a string to any integer. The reason for this restriction is that it would be too late for a constraint check to fail the schema change after the database has already agreed to it. Every time the schema change is compatible with the current schema, the database will just record that a new schema has been applied. Rows will maintain the old format on disk until the first time they are updated.

Whenever a table or index has to be rebuilt, the process is done in background working on new *hidden* versions of the original structures. Only portions of the table (individual indexes, data, etc) which need to be rebuilt will be. The database will keep serving both read and write requests and the hidden structures will be replicated to the replicants as would any other write. Reads occur against the original table while writes are performed on both the regular and the hidden version of the table. The amount of concurrency and priority of the rebuild process is dynamically adapted based on contention with the concurrent workload.

At the end of the schema change, access to the table

is blocked for a fraction of a second as the master simply switches pointers between the old and new B-trees, and logs the event, causing replicants to do the same.

Replicating and logging the schema change allows for a novel feature of Comdb2. A schema change operates as a phoenix transaction [12] and literally comes back from the dead at exactly where it left off during any master swing. This feature is invaluable in practice as it allows for long running changes on very large tables to occur in the face of both hardware failures and maintenance.

4.4 SQL

4.4.1 Transport

Clients and servers communicate with two types of transport: (1) a proprietary IPC protocol used in the Bloomberg cloud and (2) a combination of TCP and Protocol Buffers [11]. The former has the advantage of blending naturally with our infrastructure and taking advantage of features and optimizations built throughout the years including caching and routing. The latter is compatible with more operating systems and languages, and forms the building blocks for our ODBC and JDBC drivers.

4.4.2 Data Model

The database offers a standard relational schema. Tables might have: multiple indexes, unique constrains, null constraints, and foreign key constraints. The datatypes listed in Tab. 1, are similar to the ANSI SQL standard and are also closely coupled with Bloomberg's C++ standard library, BDE [20]. Full support for decimal floating point types are useful for avoiding rounding errors when operating on financial data. The blob and vutf8 datatypes are notable for being variable length but allowing the user to "hint" a size to the database. The database uses this hint to store a certain number of bytes on the page containing the row rather than in an auxiliary B-tree as is used in Comdb2 for variable length columns. Proper hinting can result in large performance gains. On disk every datatype is normalized so that every field can be ordered using a normal memory comparison rather than a dedicated function. A header prefixed to every column allows for NULL to be represented distinctly from any other value.

short, u_short	2 bytes signed/unsigned integer
int, u_int	4 bytes signed/unsigned integer
long	8 byte signed integer
float	4 byte IEEE 754 floating point
double	8 byte IEEE 754 floating point
decimal132	4 byte IEEE 754-2008 decimal floating point
decimal164	8 byte IEEE 754-2008 decimal floating point
decimal128	16 byte IEEE 754-2008 decimal floating point
datetime	ms precision timestamp (-10K BC to 10K AD)
interval	distance between two timestamps
byte[n]	fixed length unsigned byte array
blob	variable length unsigned byte array
cstring[n]	fixed length character array ending with null
vutf8	variable length utf8 string (validating)

Table 1: The list of Comdb2 datatypes

4.4.3 SQLite

SQLite is an embedded database developed by Hwaci [16]. It is publicly available as an open-source product. BLP has maintained a close relationship with the developers of this product for 10 years. A long-standing member of the SQLite consortium, BLP contributed to the development the statistics based query planner and other initiatives. Comdb2 shares its SQL engine with SQLite, which takes care of parsing an SQL statement and creating a query plan. In SQLite a query plan is compiled into a bytecode program that is executed by a virtual machine, named Virtual DataBase Engine (VDBE). The VDBE offers the basic functionalities of a register-based machine and other supplementary functions needed to operate on database tables.

In the original SQLite implementation the VDBE operates on the real B-trees of the database, allowing little to no concurrency. The VDBE acquires locks on an entire database, and releases only at commit time.

In contrast the Comdb2 VDBE operates on the cdb2 abstraction of a table during the OCC phase of transaction execution. Every write is recorded and produces an entry into the bplog as explained previously. The VDBE is unaware of the shadow-trees, auxiliary B-trees for stripes, and variable length columns. These are all handled in the cdb2 layer.

Every Comdb2 instance runs an arbitrary number of VDBE instances depending on the hardware in use. VDBE engines are pre-created and bound to threads in advance. Each VDBE is unaware of the others, since they cannot change the storage. The only time a VDBE may potentially block is when accessing a row under modification by the replication processing.

5. TUNING

5.1 Replication

When building a geographically replicated database the main concern is always latency. Comdb2 allows the user to trade performance with consistency by changing the contract of its replication protocol. Some of the options available are:

Size By default every machine in the cluster has to acknowledge that it has successfully received and applied the changes sent by the master. The size of the consensus can be reduced to half + 1 nodes in the cluster.

Type Replication values are **normal**, which involves every node in the cluster; **room** which requires consensus only within the datacenter and replicates asynchronously outside of it; and **none** which is asynchronous replication.

Incoherent nodes It is possible to fine tune when a node becomes incoherent. The algorithm used is timeout based, causing the master to give up waiting on node when either a) It has waited a given percentage longer for that node than for other node to commit a given transaction or b) the node is consistently a given percentage slower than other nodes.

Asynchronous Logging For durability purposes every time a page of the storage is modified in main memory the change has to be logged on durable storage into the WAL. This practice allows a database to recover from a crash by reapplying any changes not yet applied to the storage. This introduces the cost of an extra I/O operation for every

transaction committed. In a replicated environment this effect is magnified. Comdb2 allows flushing of the WAL to be tuned from forcing flush on every transaction commit to lazily flushing periodically. By default the WAL is flushed every 10 seconds. While in theory this exposes a crashed node to data loss, in practice this does not happen. The crashed node upon recovery is able to retransmit from the rest of the cluster before it serves requests. Failure of an entire cluster simultaneously could indeed result in the loss of committed data, though at BLP we defend against this with a truly shared nothing architecture.

5.2 Handling network failures

Network partitions are a serious problem for a distributed system. Comdb2 favors consistency over availability under a partitioned situation. A node disconnected from a master for a prolonged time (ie longer than lease) is unable to serve any type of request. Further, a network partition leaving no group as a majority results in a cluster with no master and all nodes unable to serve any requests.

We found during the development of Comdb2 that it was possible to reduce the possibility of network partitions by building independent networks which share nothing. Comb2 was designed to transparently support multiple network interfaces for its intra-cluster communication. At BLP we run every database node with multiple independent ethernet cards attached to independent switches running to independent routers, ultimately spanning independent fiber runs between data centers. In reality, network partitions no longer occur as it requires simultaneous loss of multiple independent networks which share nothing. Additionally BLP runs Comdb2 clusters across more than 2 data centers to eliminate lack on consensus from loss of a data center.

6. BENCHMARKS

In this section we present an overview of the overall performance of our system in a 6 node cluster spread across 2 datacenters. Nodes are interconnected by 10Gb/s links, and latency is respectively $0.14ms$ and $1.16ms$ inside and outside the datacenter. Each node contains 16 $3.30GHz$ Intel Xeon E5-2667 cores equipped with 25MB of cache, 256GB of main memory, and uses an array of 6 SSDs as storage. Databases are configured to use 32GB of cache that is emptied before every single test.

Fig. 7 compares Comdb2 and Percona XtraDB Cluster at a benchmark that measures the aggregate number of random rows/second which can be served from a cluster. The results present the number of clients as the x-axis and the rows/second as the y-axis. In this benchmark Comdb2 scales reads faster than Percona however, as can be seen on the right side of the chart, it runs out of hardware capacity faster. Both systems scale essentially linearly at this read benchmark as more nodes are added to a cluster

Fig. 8 compares Comdb2 and Percona XtraDB Cluster with a benchmark measuring the number of rows/second we are able to insert into the system. The results present the number of clients as the x-axis and the rows/second as the y-axis. In this benchmark Comdb2 scales faster at lower concurrency, to be briefly eclipsed by Percona around 160 concurrent clients. As concurrency increases further, both systems appear to plateau in a similar range. There is a clear upper limit to write throughput present in both architectures.

7. CURRENT WORK

A significant limitation in the current Comdb2 system is the lack of ability to write to tables that exist in remote databases. The current production implementation is read only. Work is underway to lift this restriction by building a system that will coordinate across multiple database instances using standard 2PC. The design is such that any node of any database instance can become a coordinator with no pre designation. Within a single database replication group, we have deliberately chosen to use a form of synchronous single copy replication rather than 2PC as the 2PC protocol is well known for reducing availability rather than increasing it. As a cluster becomes larger the probability of an errant node causing 2PC to be unable to commit (and block) becomes more likely. We don't believe this will be a problem in our usage of 2PC across database replication groups. As each database instance is already massively replicated and designed to be highly available, we will not face an availability problem committing with 2PC. Rather than an unavailable node making commit block, it would take an unavailable cluster; which should not happen today with Comdb2.

Comdb2 is also limited today in its ability to scale writes. While reads scale in a nearly linear manner as cluster size increases, writes have a more conservative level of scaling with an absolute limit. The offloading of portions of work (essentially all the WHERE clause predicate evaluations) across the cluster does help scale writes beyond what a single machine can do. Ultimately, this architecture does saturate on a single machine's ability to process the low level bplogs. Solutions to this problem come in the form of multi-master systems or partitioned data sets. We have chosen to pursue the latter in Comdb2 in the form of range and hash partitioned tables. The current system supports a form of partitioned tables using time of insertion, allowing applications to define automatic retention polices. Building on that we will allow for more arbitrary partitioning of tables. Under our design, each portion of the table has its own master (getting many of the benefits of a multi-master system) and is able to write to the portion it owns without coordinating with partitions uninvolved in the transaction. We expect to be able to scale writes (for workloads which do not need to coordinate among many masters) well with such a system.

8. FINAL DISCUSSION

The HA features of Comdb2 rely heavily on discipline used in physical deployment. For one, we require clocks to have some minimum allowable tolerance of skew between data centers. At BLP we maintain GPS clocks in each data center for this purpose. The system needs to be tuned to know that allowance, and to wait significantly longer for lease expiration. Higher allowance does not effect performance of a Comdb2 deployment except for pauses caused during error handling for lease expiration. We also depend on the physical nodes of a cluster being truly independent for our HA claims. At BLP this means each node is physically located away from another (within a data center) sharing as little common physical infrastructure as possible. We also distribute nodes of a cluster across multiple (more than 2) geographically distant data centers. Furthermore, we run multiple independent dedicated networks (sharing no hardware) between all nodes and between each data center. HA

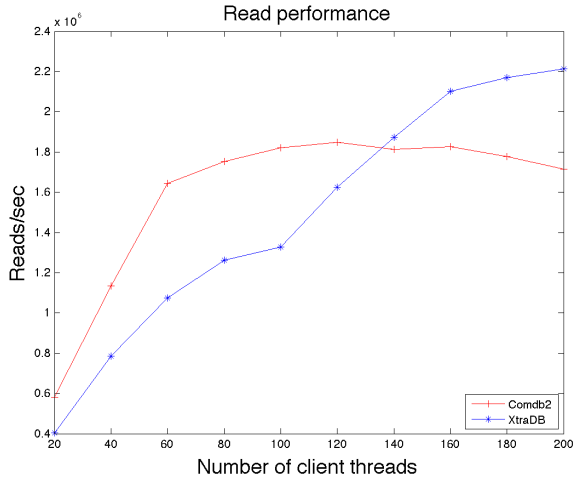

Figure 7

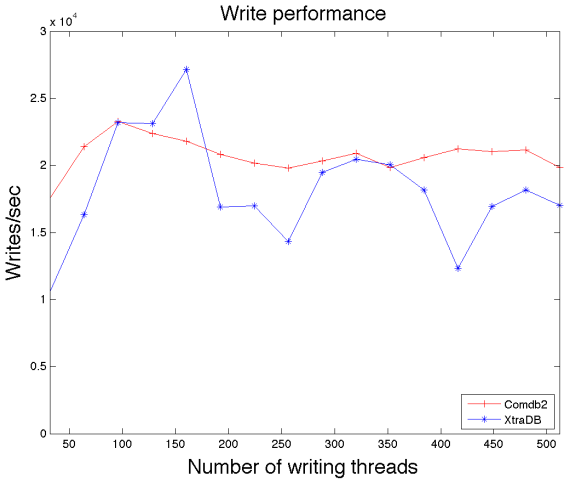

Figure 8

works when failures are isolated to one or some nodes, but clearly no software can mask physical failures of entire clusters. Along the same lines, we run each node with "unreliable" commodity SSD with no RAID, understanding that decisions that lower MTBF of an individual node are well handled with our architecture.

BLP has many years of experience using virtually every major commercial and open source RDBMS in a production setting. Usage is heavily skewed towards Comdb2 within the firm predominantly because of seamless high availability. Application programs in BLP are held to a very high standard of availability. When faced with the option of providing this level of service by integrating an RDBMS which requires the application to handle many forms of failures by taking appropriate actions, most chose the system which masks the issue from the application. Outliers are typically applications requiring an advanced feature not present in Comdb2.

The closest comparable architecture to Comdb2 being used inside BLP is Percona XtraDB Cluster. As such we have spent time studying and benchmarking that system. On the same hardware in a cluster, we see similar scaling of both read and write workloads. Both systems scale read workloads as nodes are added to a cluster but exhibit a cap on write scaling. We hope that our work on fine grained partitioning of tables will improve our ability to scale writes.

Systems that offer a coherency model similar to Comdb2 are of great interest to us. The Percona system currently offers a similar but weaker model, where "causal reads" (as referred to in Percona documentation) are not a global guarantee, but only enforced locally to the application which wrote the data. We are aware of active development in the PostgreSQL community exploring a coherency model inspired by Comdb2[25], divulged at the CHAR(14) PostgreSQL conference [27]. It is also based on the "short term lease" concept. We see this as evidence of real demand for a strong coherency model where all programs are guaranteed to read latest values in accordance with most real applications' expectations.

9. CONCLUSION

In this paper we described Comdb2, the primary database solution used at Bloomberg. Throughout the paper we tried to give insight into the last twelve years of development while highlighting details that made this project successful inside our company. We look forward to publishing this project under an open source license in order give it the visibility that we think it deserves and to begin more open collaboration.

Finally, we would like to thank Richard Hipp for his invaluable assistance on this system.

10. REFERENCES

[1] M. Aslett. How will the database incumbents respond to nosql and newsql. *San Francisco, The*, 451:1–5, 2011.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[3] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1295–1309. ACM, 2015.

[4] Cassandra. cassandra.apache.org.

[5] Y. Collet. Lz4: Extremely fast compression algorithm. `code.google.com`.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, pages 205–220. ACM, 2007.

[8] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: elastic, high performance optimistic

concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 262–275. ACM, 2015.

[9] T. A. Failover. Oracle transparent application failover. `http://www.oracle.com/technetwork/database/features/oci/taf-10-133239.pdf`.

[10] J.-l. Gailly and M. Adler. Zlib compression library. `www.zlib.net`, 2004.

[11] Google. Protocol buffers. `http://code.google.com/apis/protocolbuffers/`.

[12] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[13] Guardian. Bloomberg it meltdown leaves financial world in the dark. `http://www.theguardian.com/business/2015/apr/17/uk-halts-bond-sale-bloomberg-terminals-crash-worldwide`.

[14] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[15] D. R. Hipp and D. Kennedy. Sqlite, 2007.

[16] Hwaci. Hwaci. `https://www.hwaci.com`.

[17] Hwaci. Sqlite. `https://www.sqlite.com`.

[18] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[19] N. Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[20] B. LP. Basic development environment. `https://github.com/bloomberg/bde`.

[21] B. LP. Bloomberg lp. `https://www.bloomberg.com`.

[22] C. Mohan and F. Levine. *ARIES/IM: an efficient and high concurrency index management method using write-ahead logging*. ACM, 1992.

[23] MonetDB. Monetdb. `https://www.monetdb.org/Documentation/Manuals/SQLreference/Transactions`.

[24] MongoDB. www.mongodb.com.

[25] T. Munro. "causal reads" mode for load balancing reads without stale data. `http://www.postgresql.org/message-id/CAEepm=3X4OPP-JDW2cnWyRQHod9VAu-oMBTXy7wZqdRMtyhi0w@mail.gmail.com`.

[26] MySql. Innodb disk i/o. `https://dev.mysql.com/doc/refman/5.5/en/innodb-disk-io.html`.

[27] PostgresSQL. Char(14). `https://www.char14.info`.

[28] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradar, et al. On brewing fresh espresso: Linkedin's distributed data serving platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1135–1146. ACM, 2013.

[29] Redis.io. Redis. `http://redis.io/topics/transactions`.

[30] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[31] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[32] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012.

[33] R. Unland. *Optimistic concurrency control revisited*. PhD thesis, Universität Münster, 1994.

[34] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.

[35] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.