The VoltDB Main Memory DBMS

Michael Stonebraker
stonebraker@csail.mit.edu
VoltDB, Inc.

Ariel Weisberg
aweisberg@voltdb.com
VoltDB, Inc.

Abstract

This paper describes the VoltDB DBMS as it exists in mid 2013. The focus is on design decisions, especially in concurrency control and high availability, as well as on the application areas where the system has found acceptance. Run-time performance numbers are also included.

1 Introduction

The purpose of VoltDB is to go radically faster than traditional relational DBMSs, such as MySQL, DB2 and SQL Server on a certain class of applications. These applications include traditional transaction processing (TP), as has existed for years. In addition, more recent TP applications include a variety of non-traditional use cases, such as maintaining the state of internet games and real-time ad placement on web pages. Also included are high velocity use cases whereby an incoming firehose of messages (either from humans or ”the internet of things”) must be processed and some sort of state maintained. High velocity applications include maintaining the overall risk in an electronic trading application and a variety of telco applications. Lastly, this collection includes use cases where some sort of state must be reliably maintained over repeated update, such as maintaining the directory of a distributed file system. In all cases, a database must be reliably maintained when updated by a firehose of transactions, interspersed with queries. For ease of exposition, we call the above class of applications, **ModernTP**, to distinguish it from the various subclasses which it encompasses.

The motivation for the design of VoltDB (and its research prototype predecessor, H-Store [4]) comes from the plummeting cost of main memory. Today, one can buy 1 Tbyte of main memory for perhaps $30,000, and deploy it readily as 128 Gbytes on each of 8 nodes in a computer network. Over time, 10 - 100 Tbyte databases will be candidates for main memory deployment. Hence, most (but not all) ModernTP applications are a candidate for main memory deployment, either now or in the future.

Toward the end of this decade it is possible that a new memory technology, for example Phase Change Memory (PCM) or Memristors, will become viable. Such a technology may well be more attractive than DRAM in main memory database applications, thereby further enlarging the reach of byte-addressible DBMSs.

If your data fits in main memory, then [3] presents a sobering assessment of the performance of traditional disk-based DBMSs on your application. Specifically, on TPC-C around 10% of the time is spent on useful work (i.e. actually executing transactions). The remaining 90% is consumed by four sources of overhead, namely buffer pool overhead, multi-threading overhead, record-level locking and an Aries-style [6] write-ahead log. The details of these measurements appear in [3] and their relative sizes vary with the exact transaction mix being

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

observed. For the purposes of this paper, we can just say each source of overhead is larger than the useful work in the transaction.

The conclusion to be drawn is that all four sources of overhead must be eliminated if one wishes to perform an order of magnitude better than traditional systems. A quick calculation shows that eliminating three of the four sources will result in only marginally better performance. This motivation drives the architecture of VoltDB.

In Sections 3-6 we discuss our solution for eliminating each source of overhead. Then in Section 7, we indicate the current engineering focus, followed in Section 8 by a discussion of early application areas where VoltDB has found acceptance, along with some performance numbers on VoltDB speed. We begin in Section 2 with some assumptions about ModernTP applications that have guided our design.

2 Assumptions

2.1 Workload

First, high availability (HA) is a requirement in all but the lowest end applications. If a node fails, then the system should seamlessly fail-over to a backup replica(s) of the data and keep going. As such single-node computer systems have no possibility of supporting HA, and VoltDB assumes a multi-node cluster architecture. In this environment, it makes sense to horizontally partition every table into ”shards” and deploy the shards to various nodes in the cluster. VoltDB supports hash (and range in the future) partitioning for shards and assumes every shard is replicated K +1 times, supporting the concept of K-safety.

Second, VoltDB strives to provide ACID properties both for concurrent transactions and for replicas. This is in contrast to many DBMSs, which provide some lesser guarantee. Moreover, so-called ”eventual consistency” provides no guarantee at all, unless the transaction workload is commutative. Even Google, long the champion of squishy guarantees, is coming around to an ACID point of view [8]. Hence, VoltDB is a pure ACID SQL system, deployed across a replicated, sharded database.

We assume that applications are a mix of transactions, containing both small/medium reads and small writes. Large reads are best performed against a downstream data warehouse, which is optimized for this purpose. In addition, the writes we see have the following composition:

(Vast majority) **Single-node transactions.** In this case, there is a database design that allocates the shards of each table to nodes in such a way that most transactions are local to a single node. Looking up a banking account balance or a purchase order is an example of a single-node transaction.

(Lesser number) **One-shot transactions.** In this case, the transaction consists of a collection of SQL statements, each of which can be executed in parallel at various nodes in a computer network. For example, moving $ X from account Y to account Z (without checking for an overdrawn condition) is a one-shot that will be executed at two nodes, one holding the shard of Y and the other the shard containing Z.

(Even less) **General transactions.** These are transactions that consist of multiple phases where each phase must be completed before the next phase begins. Moreover, one or more of the phases touches multiple nodes.

Lastly, VoltDB transactions are assumed to be deterministic. Hence, they cannot have code that, for example, branches on the current reading of a system clock. In addition, they cannot block, for example by trying to communicate with a human user or a remote site, external to the VoltDB system.

In our world, a transaction is a stored procedure that is registered with the system. Transactions are analyzed at compile time, categorized as above, and stored inside the VoltDB runtime system. A user executes a transaction by identifying the stored procedure to be called and supplying run-time parameters. Ad-hoc queries and updates are accepted and compiled on-the-fly into temporary stored procedures that are called in the same way.

2.2 LAN and WAN Behavior

Most VoltDB applications are deployed on single clusters connected by LAN technology. In such worlds, LAN network partitions are exceeding rare, and are dominated by node failures, application errors (errant logic corrupts data in the database) and DBA errors (for example, accidentally deleting data) as noted in [10]. The latter two will force a VoltDB cluster to stop. Hence, network partitions are not ”the high pole in the tent” and providing HA in this case will not significantly improve overall availability. Therefore, VoltDB chooses consistency over availability during network partitions, consistent with our position of providing ACID wherever possible.

We do have users who are running replicas across a WAN between data centers. None of these users are willing to pay the cost (in transaction latency) of ensuring replica consistency across a WAN by performing synchronous replica update. Hence, VoltDB performs synchronous replication across LAN replicas and asynchronous replication across a WAN. As a result, it is possible for a WAN replica to lag its primary replica and to lose transactions on a WAN failure. Our users are happy with this design choice, and sometimes put resources into replicated networks to lower the probability of a WAN failure. Lastly, we see many cases of transient WAN failures. Hence, our users do not want to see automatic WAN failover as a result of transient errors. Instead, they want DBA-initiated WAN failover. A contrary opinion concerning synchronous WAN replication and automatic cutover is presented in [8].

3 Multi-Threaded Operation

VoltDB statically divides shared memory into ”chunks” and statically allocates chunks to individual cores. These CPUs are not multi-threaded. Hence, there are no latches or critical sections to contend with in the VoltDB code. In this paper, a node refers to this pairing of non-shared main memory and a CPU. To improve load balance, we are in the process of supporting the movement of chunk boundaries. Notice that VoltDB is not sensitive to the race conditions prevalent in multi-threaded systems. In our opinion these are the cause of many concurrency control and recovery errors. As such, VoltDB has been relatively easy to debug as it consists of non-parallel code.

4 Main Memory Operation

VoltDB stores all data in main memory format. Of course, performance degrades if real memory is exhausted, and virtual memory starts to swap. It is crucial to provision VoltDB clusters with an appropriate amount of physical memory. All data is stored in main memory format and any pointers exist in virtual memory. There is no buffer pool and no buffer pool code.

5 Concurrency Control

VoltDB executes all operations in a deterministic order. Previous work in this area has typically used time as the ordering mechanism (timestamp order); however, VoltDB does not use a clock-based scheme. A single-node transaction is examined in the user-space VoltDB client library, where parameters are substituted to form a runnable transaction. The client library is aware of VoltDB sharding, so the transaction can be sent to a **node controller** at the correct node. This node controller serializes all single-node transactions and they are executed from beginning to end in this order without any blocking. Any application that consists entirely of single-node transactions will obtain maximum parallelism and, in theory, run all CPUs at 100% utilization, as long as the sharding achieves good load balance.

Other kinds of transactions are sent to a special **global controller**, which is responsible for deciding a serial order. The various SQL commands in such a transaction must be sent to the correct sites, where they are inserted into the single-node transaction stream at any convenient place. Hence, they are arbitrarily interleaved into the single-node transaction mix. A moment's reflection should convince the reader that the result is serializable. Various optimizations are implemented and others are planned for the classes of more general transactions.

It should be noted that our deterministic ordering scheme is one example of a class of deterministic concurrency control schemes. Other candidates are presented in [8, 9, 11].

6 Crash Recovery and High Availability (HA)

There are two aspects we need to discuss in this section. First, we discuss replication, fail-over and fail-back. Following that we consider global cluster failures, such as power failures.

6.1 Replication

High availability is achieved through replication. VoltDB implements a simple and elegant scheme. When a transaction is executed, it is first sent reliably to all replicas, which process the transaction in parallel. In other words, it is an active-active system. Since concurrency control is deterministic, each transaction will either succeed or fail at all nodes, and no additional synchronization is needed. In contrast, other popular concurrency control schemes, such as MVCC [1], do not have this deterministic property and must have some protocol to ensure that one gets the same outcome at each replica. Alternately, they would have to use an active-passive scheme that moves the log over the network, which we expect will offer much worse performance.

Nodes exchange ”I am alive” messages. When a node is determined by its peers to be dead, it is disconnected from the VoltDB cluster and the remainder of the cluster continues operation. Up to K such node failures can be accommodated in this fashion. On the K + 1st error, the cluster stops.

When a node returns to operation, it executes the protocol in Section 6.2 to bring its state nearly current with the rest of the cluster. Then, the system catalogs are updated to reflect the newly added node and the tail of the transaction log (discussed in the next subsection) is executed at the newly operational node. Then, the cluster resumes normal operation. Users report that this failover-failback occurs without anybody being aware that it is happening.

6.2 Global Errors

We have a number of VoltDB customers running on off-premises shared servers (the cloud). Such servers usually have an uninterruptable power supplies (UPS). On-premises servers often do not have a UPS system. In such circumstances, a power failure or other cluster-wide error can cause all nodes in the cluster to stop, and the contents of main memory will be lost. Obviously, this is unacceptable behavior.

To deal with this error condition, VoltDB takes asynchronous, transaction-consistent checkpoints of the state of main memory. Specifically, when checkpoint-begin occurs, a local executor goes into a copy-on-write mode. Main memory is passed by the checkpoint code, and the results of all transactions before a specific **anchor** one are written to disk. Indexes are not checkpointed, but instead are rebuilt during recovery. A user can control the frequency of checkpoints and only the last one need be retained for subsequent recovery. The net result of a checkpoint is a disk image of main memory together with the anchor transaction.

In addition, VoltDB keeps a transaction log. For each transaction, we record the transaction identifier and its parameters. A group commit is used to reliably write this log to disk, before results are returned to the user.

When a global error occurs, the latest checkpoint is restored at each local node. A protocol is used to trade the tail of the log among nodes, so the set of committing transactions can be determined. Then, the transactions since the anchor transaction are re-executed as conventional user transactions. The net result is a transaction consistent database, which is then free to accept new user transactions.

Our implementation of checkpointing and the transaction log degrades the run time performance of a VoltDB cluster by around 5%. In contrast, some of us implemented a main memory data logging scheme which pushed the effects of each transaction to disk. This technique is described in [5] and is a simplified and streamlined variation of Aries. In testing the run-time performance of the two recovery schemes it was found that the overall throughput of transaction logging was 3 times the throughput of the data logging scheme. Of course, recovery time was substantially longer with transaction logging. Since global errors are rare, it seems appropriate to optimize for run time performance. Also, a cluster is up and running much more often than it is recovering from a node failure. Also, single node failures can be recovered in the background, so again recovery time is less critical than performance at run time.

7 Current Development Focus

7.1 SQL

VoltDB engineering is focused on enlarging the subset of SQL that is supported. After all, the SQL specification is some 1500 pages long, and a startup obviously has to proceed in stages. Hence, every VoltDB release adds SQL features, based on customer requests.

7.2 On-Line Reprovisioning

Our customers, especially the ones offering some sort of web service, often experience dramatic load swings as well as substantial increases in load over time (the so-called hockey stick of success). Obviously, they wish to begin operation on K server nodes, and then add or subtract processing nodes, re-sharding the data as needed, all without incurring down time. On-line reprovisioning should be completely operational by the time this paper appears.

7.3 Integration with Downstream Repositories

As noted in Section 2, VoltDB is not architected to solve resource-intensive long-running data warehouse-style queries. Moreover, the storage requirements for long-term historical data repositories can be extreme. Hence, VoltDB is often ”upstream” from some sort of large storage system. Our customers request high performance parallel export to the major warehouse vendors, and we are well-along at satisfying this need.

8 Application Areas and Benchmarks

8.1 Early Application Areas

VoltDB found early acceptance in the web community, where it is used to maintain the state of internet games, leaderboards, and other applications which need high speed update of a shared state. In addition, it is widely used to execute real-time ad placement on web pages.

It has also found early adopters in aspects of real-time electronic trading, for example to keep track of the global position of the enterprise across multiple electronic trading desks. In addition, it is also used in real-time compliance applications.

Lastly, VoltDB has become widely used in embedded products in a variety of application areas, whereby an application needs to maintain a high velocity state of some sort. In all VoltDB has about 200 production users, and is now finding application in a variety of markets, that are not regarded as early adopters. Hence, VoltDB should be considered as ”having crossed the chasm” [7].

8.2 VoltDB Performance

We have been benchmarked on a variety of applications and include numbers in this section for a representative Voter benchmark, which obeys the application assumptions of Section 2. Voter uses the schema in Table 1, and has three transactions. The first one simulates a telephone vote tallying system, whereby each customer gets a collection of X votes, which he can allocate to contestants one-by-one until they are exhausted. The second transaction produces a heat map for a contestant for all states once a second, while the third produces a leaderboard once a second.

```
Tables:
    Contestants(c_number, c_name) -- replicated at all nodes
    Area_code(area_code, state)   -- replicated at all nodes
    Votes(phone_number, state, c_number) -- partitioned on phone_number
Materialized views:
    Votes_by_phone_number(phone_number, count)
    Votes_by_contestant_number_state(c_number, state, count)
```

Table 1: The Voter Schema

Referring to Table 1, the Vote transaction does a SQL lookup in the Contestants table to verify that the consumer is voting for a legitimate contestant. Then, it does a SQL lookup to the Votes_by phone_number materialized view to ensure that the customer has sufficient votes left to do his proposed operation. Third, the procedure looks up the state for the given area code from the Area_code table. Then, it does an insert into the Votes table. Finally, the transaction causes updates to the materialized views whereby appropriate counts are incremented. In all there are three selects, one insert and two updates. Since Votes is partitioned on phone number, each materialized view is partitioned the same way and the two read-only tables are replicated at all nodes, the transaction has 6 SQL commands and is single-noded. Finally, once per second the heatmap and leaderboard must be constructed. The tally of votes by contestant is a one-shot which aggregates the Votes_by_contestant_number_state materialized view. Similarly the heatmap is a one-shot constructed from the same materialized view. Performance on the Voter benchmark is measured by the number of votes per second that can be processed for the indicated schema, performing the one-shots every second. Details about Voter can be obtained from files in the directory: https://github.com/VoltDB/voltdb/blob/master/examples/voter/.

A five node dual quad-core SGI system (total of 40 cores) performs 640,000 Voter transactions/second. This is about one SQL command per core every 10 microseconds. Adding nodes to the configuration resulted in linear scalability. In all a 30 node system executed 3.4 million transactions/sec, and linear scalability was observed with a slope of 0.88. Details concerning these results can be found at www.sgi.com/pdfs/4238.pdf.

9 Summary

VoltDB is a main memory DBMS designed for ModernTP applications running on a cluster of nodes. It uses deterministic scheduling and an active-active replication strategy. As an example of a ”NewSQL” DBMS, we expect it (and other products) will gradually take over the ModernTP market from legacy disk-oriented vendors because of dramatically superior performance.

References

[1] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI'12: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[3] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD Conference*, pages 981–992, 2008.

[4] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[5] N. Malviya. et. al. Recovery algorithms for in-memory OLTP databases. *(submitted for publication)*.

[6] C. Mohan, D. J. Haderle, B. G. L. 0001, H. Pirahesh, P. M. Schwarz, and P. M. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, pages 94–162, 1992.

[7] G. Moore. *Crossing the Chasm: Marketing and Selling Disruptive Products to Mainstream Customers*. Harper-Collins, 2002.

[8] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[9] A. Pavlo, E. P. C. Jones, and S. B. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB*, 5(2):85–96, 2011.

[10] M. Stonebraker. In search of database consistency. *Commun. ACM*, 53(10):8–9, Oct. 2010.

[11] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Conference*, pages 1–12, 2012.