How VoltDB does Transactions

Note to readers: Some content about read-only transaction coordination in this document is out of date due to changes made in v6.4 as a result of Jepsen testing. Updates are forthcoming.

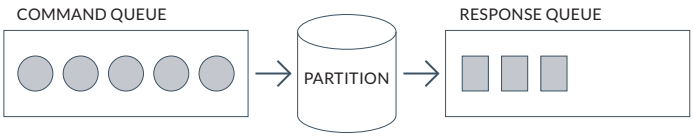
A few years ago (VoltDB version 3.0) we made substantial changes to the transaction management system at the heart of VoltDB. There are several older posts and articles describing the original system but I was surprised to find that we never explained at length how the current transaction system is constructed. Well, no time like the present.

Hammock Time: starting with a thought experiment

Start with some DRAM and a single CPU. How do we structure a program to run commands to create, query and update structured data at the highest possible throughput? How do you run the maximum number of commands in a unit of time against in-memory data?

One solution is to fill a queue with the commands you want to run. Then, run a loop that reads a command off the queue and runs that command to completion. It is easy to see that you can fully saturate a single core running commands in this model. Except for the few nanoseconds it requires to poll() a command from the command queue and offer() a response to a response queue, the CPU is spending 100% of its cycles running the desired work.

In VoltDB's case, the end-user's commands are execution plans for ad hoc SQL statements, execution plans for distributed SQL fragments (more on this in a second), and stored procedure invocations. We refer to the loop that services the command queue as a *partition*.



ACID Semantics

A partition can easily implement atomic, consistent and isolated semantics for each command. While executing the commands, the program maintains in-memory undo logs so that aborted commands can roll back. This provides atomicity. Constraints and data types are enforced during command execution to guarantee (ACID) consistency. Commands run one at a time without overlap, providing serializable isolation.

How do we make this model durable? If all of the commands have deterministic side-effects (running the same queue of commands in the same order against the same starting dataset is promised to produce the same ending dataset), then writing (and fsync-ing) the command queue to disk before executing the commands makes all data changes durable. To recover the ending state, we read the commands from disk and replay them. VoltDB calls this journal the "command log."

Of course the journal cannot expand forever. At some point it needs to be truncated. We write a snapshot (aka checkpoint) of the in-memory state of the partition to disk. When this snapshot is completely written, the command log for commands preceding the snapshot can be forgotten. Starting a snapshot is another command that can be placed in the command queue and processed by the partition. Therefore, the snapshot has a distinct place on the serializable timeline and is a transactionally consistent copy of data. To avoid stalling the partition while the snapshot is written to disk, the partition places itself into a copy-on-write mode. The immutable copy is flushed to disk in the background while new commands query and update live data.

Recovering a partition's durable state is a two-step process. First, the snapshot must be read from disk and restored into memory. Second, journaled commands that follow the snapshot on the serializable timeline must be replayed. The VoltDB database performs these operations whenever a fully stopped cluster is restarted.

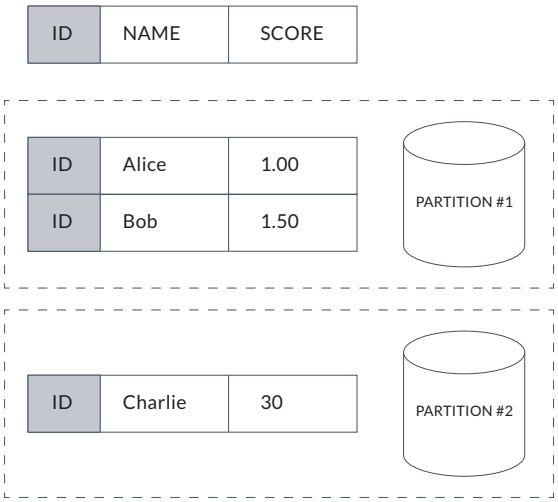
VoltDB makes the snapshot interface available to end-users. A user can request a snapshot for his or her own use independently (for backup or downstream batch processing) from the command log truncation process. VoltDB can produce snapshots in a binary format and also in a CSV format. In all cases the snapshot is a transactionally consistent point-in-time checkpoint of the full contents of the database.

Scaling across multiple CPUs

Saturating a single CPU with useful (fully ACID) end-user work is great. However, a server has several CPUs. How do we scale command execution across many CPUs? The first step is to partition (shard) the data and then queue commands to the relevant partition. This is the default in a distributed NoSQL system: operations specify the key (or document) they execute against.

VoltDB uses a consistent hashing scheme to distribute the rows of a partitioned table across many distinct partitions. Each partitioned table has a user-designated partitioning column. The values in that column are analogous to the keys in a NoSQL store. SQL statements or stored procedures that only operate against a single partitioning column value are routed to the corresponding command queue.

Sometimes an example is helpful to explain row-wise partitioning. Here's a simple table with three columns: Id, Name, and Score. In this example, the Id column is the user-specified partitioning column. The rows with Id=1 and Id=2 are assigned to partition #1. The row with Id=3 is assigned to partition #2.



When the system receives a command for Id=2, it queues that command to the command queue for partition #1. Likewise, when the system receives a command for Id=3, that command is added to the command queue for partition #2.

Adding multiple partitions to a server (and then multiple servers to a cluster) allows the system to consume many CPU cores, each running per-partition commands. Each partition runs independently - each provides ACID semantics for the commands processed from its associated command queue.

We will discuss how VoltDB handles commands that span multiple partitions below. (First we need to describe replication.)

At this point in the discussion it is hopefully clear that:

1. VoltDB executes commands to query and change data serially at a partition.
2. Commands can be ad hoc SQL, stored procedures, or SQL plan fragments.
3. Each command is run with ACID semantics.
4. Table data is distributed across partitions.
5. Execution is scaled across CPU cores (and servers) by adding partitions.

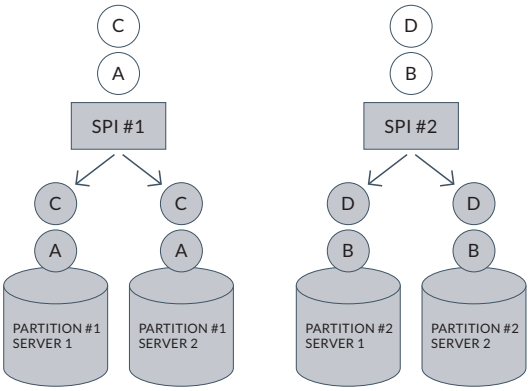
Replicating for High Availability

Modern scale-out systems treat fault tolerance as a first class feature. VoltDB replicates partitions to provide fault tolerance. If a server crashes and its partitions are terminated, the cluster can continue to operate using the other copies available on the surviving servers. So, let's describe how VoltDB implements partition replication.

VoltDB allows the user to specify a desired replication factor. We term this a "k-factor." The cluster keeps k+1 copies of each partition. If the user configures a k-factor of 1, the VoltDB cluster will maintain two identical copies of each partition. K-factor specifies the number of partition replicas that can be lost due to failure or operational commands while still preserving a full copy of data in the running database.

To keep copies identical, VoltDB replicates the commands that are queued at each partition. VoltDB does not replicate tuple changes as deltas across partitions - instead it synchronously replicates all commands that can change data to all copies of a partition. Each partition individually runs the command against the data it manages. We maintain as an invariant that the commands in the command queue at each copy are in the same order. The internal component that communicates the sequence of commands to all copies of a partition is called the single partition initiator (SPI).

We extend the example above to show a representation with a k-factor = 1 (two copies of each partition), an SPI dedicated to partition #1 and an SPI dedicated to partition #2. Each partition in this example has two outstanding commands to process. Partition #1 needs to service commands A and C. Partition #2 needs to service commands B and D. VoltDB will automatically route commands A and C to SPI #1. That SPI will then replicate them with identical sequencing to all copies of Partition #1. SPI #2 will do the same for commands B and D and partition #2.



Partitions route responses back to their controlling SPI. Once the SPI sees a response from each surviving partition, it forwards the response to the request originator.

The use of determinism in VoltDB

The ACID section mentions in passing that command logging relies on commands being deterministic. I'm sure many readers will notice the same is required of the replication method. If running command A at the first copy of partition #1 resulted in different end-states than running the same command at the second copy, the two copies would diverge. Diverged copies are not replicas!

VoltDB relies on deterministic commands. Running a command against a given input will always generate the same output. We rely on determinism for command logging, for replication, and for our export implementation (which is discussed in a separate note). There are two sources of non-determinism that VoltDB controls.

The first source is user code in Java stored procedures. Users often want the current time or a random number, for example. Since each partition is running each command independently, it is important that the commands produce the same results. VoltDB's stored procedure API provides deterministic functions for the current time and also provides deterministic seeds (on a per-command basis) for random number generation. Both of these inputs are provided by the SPI when commands are communicated to partitions. VoltDB forbids as policy (not as a system-enforced constraint) that stored procedures do not access the network, call remote services, or access the disk. All of these IO activities are non-deterministic.

The second source of determinism that must be controlled is non-deterministic SQL. This is relatively rare (it usually does not make sense as business logic to do something random to your data). An example of non-deterministic SQL is selecting two random rows from a table, e.g., SELECT * FROM T LIMIT 2;. Simply querying like this won't cause replicas to diverge; however, if the next action in the stored procedure is to delete the returned rows, the replicas will no longer be exact copies. The VoltDB SQL planner identifies and warns on statements that are non-deterministic.

To detect and prevent divergences, all SQL inputs to a partition that modify data are hashed. The resulting hashcode is communicated with the command result back to the SPI. The SPI compares the hash value from all partitions. If two partitions respond with different hashcodes, it is clear that the partitions did not witness the same inputs as a result of a replicated command. This is viewed as an application error which the SPI logs and terminates the cluster to maintain the consistency of the data. The user must fix the application to remove the non-deterministic behavior and resume operation.

Transactions

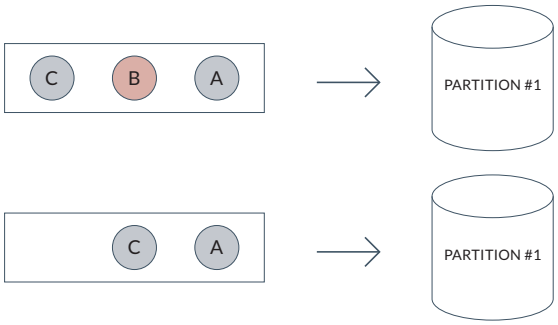
To this point we have only discussed commands that need to execute at (all copies) of a partition. We've also been a little coy with this "command" term. Let's step away from the command abstraction and speak directly to how VoltDB executes transactions.

In VoltDB, a single-partition transaction requires data at exactly one partition to execute to completion. VoltDB runs stored procedures as individual transactions. VoltDB also runs ad hoc SQL as auto-committed transactions. Single-partition transactions can be read-only or read-write. Each single-partition transaction is fully ACID.

Single-partition read-only transactions contain SELECT statements that read data from only one partition. They do not INSERT, UPDATE or DELETE data. As an optimization, read-only transactions skip the SPI sequencing process and are routed directly to a single copy of a partition. There is no useful reason to replicate reads. Effectively, this optimization load-balances reads across replicas. Because VoltDB always performs synchronous replication of read-write transactions within a partition, end-users are guaranteed to read the results of prior writes even when reads bypass the SPI sequencer. This

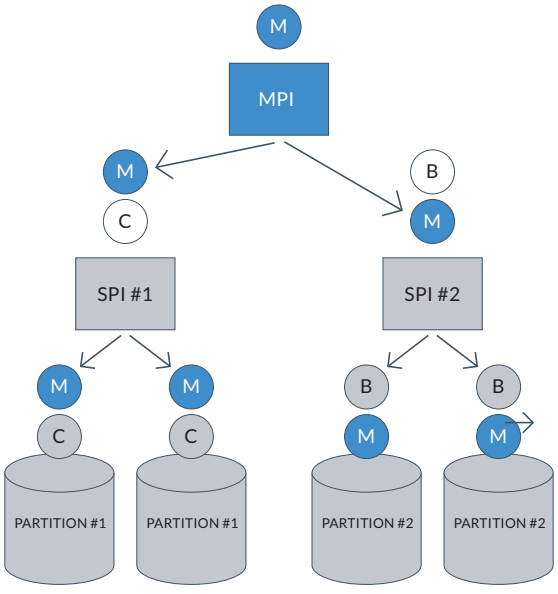
optimization increases total read-only throughput, decreases latencies on read-only transactions, and reduces the work that must pass through the per-partition SPI - three nice wins that result from the synchronous replication model.

Sometimes it takes a moment to understand that slotting reads into the command queue at a partition produces serializable reads. A visual example can help. In this example, transactions A, B, and C are all single-partition transactions. Transaction B is a read-only transaction. B happens-after transaction A and happens-before transaction C. Because B is read-only, it has no side-effects on the state of Partition #1. For both replicas, transaction C begins with the same partition #1 state.



VoltDB also supports transactions that read and write data at multiple partitions. We call these multi-partition transactions. Just as there are SPIs that sequence work for each partition, there is a multi-partition initiator (MPI) for sequencing multi-partition transactions across partitions. The multi-partition initiator communicates with the per-partition SPIs to inject multi-partition commands into the per-partition command queues.

In this example, the MPI is distributing commands for transaction M. SPI #1 sequences the distributed commands for M after single-partition transaction C. SPI #2 sequences the distributed commands for M before single-partition transaction B. The resulting serializable timeline is C -> M -> B.



To execute SQL statements that require data from multiple partitions, VoltDB's SQL planner splits execution plans into fragments. Some fragments are distributed across the cluster and run at multiple partitions. Other fragments run against the collected results of the distributed fragments.

Multi-partition transactions that modify data are two-phase committed across partitions. The MPI distributes commands (SQL fragments) that run remotely at each partition in the prepare phase. If these commands are run successfully without a constraint violation at each partition, the MPI instructs all remote partitions to commit. The remote partitions will stall (they will not process new work from their command queue) until they receive the commit.

Many use cases in VoltDB use multi-partition transactions primarily to perform distributed reads - either to locate a record when the application doesn't know the partitioning column value, or to aggregate analytics and data that's maintained on a per-partition basis. Multi-partition transactions that consist of exactly one read-only batch of work are tagged with metadata indicating that they are one-shot reads. They are routed as above (though the SPI may choose not to replicate them as an optimization). After processing the distributed command for a one-shot read, a partition immediately continues operation and returns to servicing its command queue. It will not stall and does not need to wait for a commit in this case.

Repairing Faults

Each of the described components - partitions, SPIs, and the MPI - can fail as part of a node failure or network partition. Faults are detected and new cluster topologies are established via a distributed agreement protocol. New topology states are updated in an embedded version of ZooKeeper that runs as part of each VoltDB cluster. The transaction management system uses standard ZooKeeper leader election techniques to elect new SPIs and a new MPI to replace any failed components.

Each partition maintains an in-memory log of commands that have been received. The SPI and MPI piggy-back truncation points for this log with new commands. The commit point is only advanced when all partitions have responded to a command. After leader election runs to select replacement SPIs and MPIs, the new leaders run a fault repair algorithm. New SPIs ask each remote partition they control for their log. These collected logs are unioned. The new SPI re-sends messages that were lost as a result of failure, bringing all remote partitions to a consistent state. The MPI undertakes a similar activity. These fault repair algorithms typically take sub-second time to run.

Rejoining Faulted Partitions

After a server fails, a user can issue an operational instruction to rejoin a replacement server. The new server joins and is assigned under-replicated partitions. Each new partition joins the corresponding SPI group in an initializing state. While the partition is initializing, it maintains fault repair metadata, acknowledges commands to the SPI, and logs commands that are arriving. The joining partition initiates an internal snapshot to reconstitute its state - and then replays logged commands to fully rejoin the SPI's partition group. The end result is a fully reconstituted partition replica, restoring k-safety.

Elastically Adding Partitions

If you need more storage or need to increase throughput, new servers can be added to an existing cluster. In this case, a server provisions new partitions with an empty range on the cluster's consistent hash ring. Over time, the hash ring is adjusted to assign small chunks of the ring to the new partition. As new chunks are assigned, an internal transaction runs to delete partitioned rows assigned to that consistent hash range from the source partition, and inserts those rows into the new partition. This process transactionally rebalances data (and the associated work routed to this data) from existing partitions to elastically added new partitions.

Conclusion

This write-up explains the basics of the VoltDB transaction management system. VoltDB provides a scalable, fault-tolerant, serializable, fully durable, high throughput transaction-processing model. VoltDB makes transactions durable at all replicas before responding to a client. (Users can configure relaxed durability if desired.)

VoltDB's design trades off complex multi-dimensional (OLAP) style queries for high throughput OLTP-style transactions while maintaining an ACID multi-statement SQL programming interface. The system is capable of surviving single and multi-node failures. Where failures force a choice between (CAP-wise) consistency and availability, VoltDB chooses consistency. The database supports transactionally rejoining failed nodes back to a surviving cluster and supports transactionally rebalancing existing data and processing to new nodes.

Real world applications achieve 99.9% latencies under 10ms at throughput exceeding 300,000 transactions per second on commodity Xeon-based 3-node clusters.