Amazon Aurora FAQ

Q: Why is Aurora faster than Mirrored MySQL in Table 1?

A: The main reason is that Mirrored MySQL sends many more bytes of data over the network than Aurora.

Mirrored MySQL involves an ordinary MySQL database server that thinks it is writing to a local disk. Each transaction involves a bunch of large writes to update B-Tree pages; even if a transaction only modifies a few bytes of data, the resulting disk writes are entire file system blocks, perhaps 8192 bytes. The mirroring arrangement sends those 8192-byte blocks over the network to four different EBS storage servers, and waits for them all to write the data to their disks.

Aurora, in contrast, only sends little log records over the network to its storage servers -- the log records aren't much bigger than the actual bytes modified. So Aurora sends dramatically less data over the network, and is correspondingly faster.

The down-side is complexity and reduced generality -- the Aurora storage servers knows a lot about the database -- each storage server is able to apply the log records to the stored data. In effect, a significant chunk of the database functionality is moved into the storage system. The Aurora storage system is probably not useful for anything other than Aurora.

Q: What's the VCL about?

A: During recovery after the database server crashes and is restarted, Aurora has to make sure that it starts with a complete log. The potential problem is that there might be holes in the sequence of log entries near the end of the log, if the old database server wasn't able to get them replicated on the storage servers. The recovery software scans the log (perhaps with quorum reads) to find the highest LSN for which every log entry up to that point is present on a storage server -- that's the VCL. It tells the storage servers to delete all log entries after the VCL. That is guaranteed not to include any log entries from committed transactions, since no transaction commits (replies to the client) until all log entries up through the end of the transaction are in a write-quorum of storage servers.

The log before the VCL may contains log entries for transactions that didn't commit before the crash. The database server sends out new log entries that un-do the modification made by the log entries for those transactions.

This later paper explains some of this more clearly:

Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes, at SIGMOD 2018.

Q: Why also a VDL?

A: The data pages stored in the storage servers are really parts of some complex indexing data structure, perhaps a B-Tree or something similar. Updating the B-Tree can be complex and require multiple writes to different parts (e.g. to balance it, or make room to insert new data). Complex changes to the data structures sometimes span multiple log entries, but if you looked at the data structure between these entries, it wouldn't be consistent and reading it would produce garbage. The CPLs mark points in the log at which it's safe to read the data structures. And the VDL marks the last such point before the

VCL.

One place where this is important is in read-only DB replicas (the "Replica Instances" in Figure 3, explained in 4.2.4). Those read-only replicas serve client read (but not write) requests, and they must be able to read the data structures without getting confused. So they can only look at versions of the data pages that correspond to CPL boundary (e.g. the VDL).

I do not fully understand why the VDL is important for crash recovery. Perhaps the idea is that it's crucial to start with the storage data structures in an internally consistent state, and only then can you roll them forwards and backwards for re-do of committed transactions and un-do of transactions interrupted by the crash.

Q: How does Aurora compare to Google's Spanner?

A: The main pragmatic difference is that Aurora is limited to just one read/write database server. If you have more read/write transactions than can be handled with a single server, Aurora alone won't be good enough. Spanner, in contrast, has a pretty natural scaling story, since it not only shards its data, but also can run transactions that affect different shards entirely independently.

They share three important attitudes. First, they both view having replicas in multiple independent datacenters to be important, to ensure availability even if an entire datacenter goes offline. Second, they both use quorum writes to be able to smoothly tolerate a minority of unreachable or slow replicas (and this is partially to tolerate offline datacenters). Third, they both support high read throughput (Spanner with its time-based consistent reads, and Aurora with its read-only database replicas).

A huge difference is that Aurora is a single-writer system -- only the one database on one machine does any writing. This allows a number of simplifications, for example there's only one source of log sequence numbers. In contrast, Spanner can support multiple writers (sources of transactions); it has a distributed transaction and locking system (involving two-phase commit).

A murky aspect of Aurora is how it deals with failure or partition of the main database server. Somehow the system has to cut over to a different server, while ruling out the possibility of split brain. The paper doesn't say how that works. Spanner's use of Paxos, and avoidance of anything like a single DB server, makes Spanner a little more clearly able to avoid split brain while automatically recovering from failure.

Q: Why do more traditional databases like MySQL generate many I/O disk operations for each transaction?

They need to update their data structures on disk -- the B-Trees or whatever that hold the DB's tables and indices. Plus they have to append log entries to the log on disk. Furthermore, typically there's a file system between the database and the disk, which adds its own writes to update things like i-nodes, block free bitmaps, and the filesystem's own log.

Q: Does the database immediately flush the data to disk, or does it use a buffer cache?

A: The database can't respond to the client until all of the log entries for the transaction are safely on the disk. The database will try to group the log writes of several concurrent transactions together to reduce the number of disk operations (this is called group commit).

The writes to the DB's data on disk (B-Trees &c) can be done later, and usually are, in the hopes of being able to combine the writes of multiple transactions to the same part of the DB into a single disk write.

Q: I don't really understand the content of section 3.2. How is the redo "offloaded"?

A: The storage servers store the "data pages" that make up the database's tables and indices. These are probably B-Trees or something similar.

The database server needs to update its data in the storage servers as it executes client requests. Most other database servers do this by reading the old data pages from storage, modifying them, and writing back entire pages. This requires moving a lot of data, even if the update modifies only a few bytes. Aurora, in contrast, just sends its log records to the storage servers. The log records contain a much smaller description of changes -- just the modified byte values. So it's much quicker to send log entries over the network to storage servers than to send entire data pages. Of course, then the storage servers have to interpret the log entries and apply the modifications to their stored data pages. This is the offloading the paper talks about.

Q: How does Aurora avoid split-brain in the face of a 50/50 network partition?

A: For the 50/50 situation, one half or the other would presumably contain the database server (the Primary Instance in Figure 3). Only that half can operate, since only the database server serves client requests. Since it can't talk to a write quorum (which is 4 of 6, more than 50%), it can't write. It can talk to a read quorum, so it can read.

I think the only way split brain could arise is if there somehow were two database servers, e.g. if the database server were believed to have failed, and a replacement started, but the old one was actually still operating. Presumably this is a real problem, and presumably Amazon has a solution, but the paper does not explain.