**Transactional Integrity without Transactions**

As outlined in the architectural overview document, the central model in version 2.0 of the EventStore is that of a “commit” consisting of a series of events. By focusing on a commit we are able to liberate ourselves from the high requirements introduced when using a two-phase commit protocol. At the same time, by reordering slightly the sequence of operations when performing a commit to the configured persistence engine, the EventStore is able to avoid using transactions at all and yet still maintain transactional integrity, all of which has been outlined in the architectural overview document. Our purpose here is to understand how we are able to maintain transactional integrity **even when** it is possible, such as when using a relational database as the configured persistence mechanism.

First and foremost, in a relational database there is often a concept of foreign key constraints. These constraints are used as a guard to protect relational integrity. Fortunately, in our model this integrity is not necessary. As silly and ridiculous as this may sound, the reason for this surrounds several factors implicit in the model. The primary factory is there is only one central table in our relational database—the Commits table. The other tables, such as Dispatch and Streams are actually only heuristics to aid in a few simple queries. The other reason is that, by removing these constraints, we are able to reorder the sequence of operations.

The Commits “table” is our central source of truth. Without it, there is no event store. It is also what enables and facilitates optimistic concurrency control. By numbering each commit for a particular stream we are assured that no two commits will ever step on the other’s respective toes. This simplistic form of optimistic concurrency control is the foundation upon which we can build using alternate persistence mechanisms. If a commit does not make it into this table, the commit never occurred. No matter what other tables might say, the commit never took place.

When employing a relational database, we can also use a concept known as a Dispatch table. The Dispatch table contains weak references to a set of commits which have not been dispatched or put on the wire for interested subscribers. Typically the lifespan of an entry in this table is very small, perhaps a few dozen milliseconds. Generally, this same concept could be structured as a simple Boolean field on the commit row itself, which is how is implemented in document persistence engines. But in a relational model, indexes on a Boolean are seldom utilized because of their low cardinality. As such, a separate table is employed. Furthermore, this table is only ever used for one purpose—for when a process fails to dispatch or succeeds in dispatching but hasn’t marked the commit as dispatched. When the application starts, it can query the Dispatch table to determine what work has not yet been performed. Generally speaking, an entry in the Dispatch table indicates that there is an outstanding commit to be dispatched.

The Streams “table” or concept—both in a relational model and document-oriented model—is also interesting in that it doesn’t authoritatively dictate the attributes of a particular stream. Typically the Streams table will have information such as diagnostic/name info about what kind of stream it is, e.g. a .NET Type. Furthermore, it will have several values which indicate the sequence or revision of the most recent commit as well as the revision of the most recent snapshot. It is for this purpose that we utilize a Streams table—to be able to determine how far the most recent revision is ahead of the most recent snapshot. Beyond this, the table does not serve any purpose. In contrast to Greg Youg’s event store document, the Streams table does not aid in optimistic concurrency control, that’s what the Commits table does.

Interestingly enough, by making a “commit” the central concept, and yet still employing the stream identifier, e.g. StreamId as the single identifier for all things related to a stream, we’re able to take advantage of horizontal sharding capabilities of a number of storage engines such as MongoDB, MySQL Cluster, and others. Furthermore, because we’ve removed all foreign key constraints, the storage engine doesn’t need to enforce them, such as MySQL Cluster (NDB) which would otherwise handle this at the database level rather than at the storage engine level, e.g. InnoDB. In other words, we have sacrificed very little, if anything at all, to have massive gains in scalability.

The current source code master/trunk commit of the EventStore has reordered slightly the sequence of operations to take advantage of all of the above understanding thus avoiding the overhead of transactional integrity along with the complexity of isolation levels. But we now need to consider failure conditions and potential race conditions to see how they are handled elegantly by the order of operations when using a relational database.

The current sequence of operations is thus:

1. Insert a weak reference to the commit using StreamId and CommitSequence into the Dispatch table.
2. Insert the actual commit into the Commits table.
3. Insert or update the Streams table—insert for the first commit; update for subsequent commits.

What happens if something fails between the first and second steps? The way the current SQL query is written, a Dispatch is only inserted if one doesn’t already exist for that particular commit sequence. In other words, if a commit is attempted and fails just after the first step, we have a dangling reference to a commit. But this dangling reference isn’t a problem as we shall see.

In most cases, the commit will be retried, but even if it’s not retried the next commit related to the stream in question will attempt to perform an insert into the Dispatch table and see that a corresponding row already exists and it will simply skip the insert for that commit into the Dispatch table. In other words, it will assume control of the Dispatch row. The bottom line is that whether our commit succeeds after retry or another commit succeeds first, a corresponding Dispatch row (which before was dangling) will now point to a commit. And once the commit is dispatched, it will remove the row from the Dispatch table. With the row removed, we no longer have a dangling row in the Dispatch table. Interestingly enough, we don’t have foreign key constraints, but we have perfect referential integrity.

In a race condition between two commits, both will attempt to write to the table. But the way the insert is written, only a single row will ever result. Following that, the first process to insert into the Commits table wins the optimistic concurrency “race”, while the other fails. This means that the Dispatch table is correctly pointing to the victorious insert in the Commits table. The failed insert has nothing to roll back because it never actually inserted anything into the Commits table and the associated Dispatch entry is correctly pointing to the successful Commit. As we can see, race conditions are also gracefully handled.

But what about a failure scenario after the Commit row has been inserted and the Streams table is never updated? As mentioned previously, the Streams table is only a heuristic to determine how far the most recent stream revision is away from a snapshot. In a failure condition, the Streams table is never updated. This will also only happen in extreme circumstance such as database failure (process termination, hardware failure, power loss, etc.) because the application has already sent the entire batch of SQL statements as a single transfer across the wire to the database process.

In the event the database somehow does fail to update the associated row in the Streams table, the worst possible consequence is that a snapshot that might have otherwise been taken is not taken. At the same time, the very next commit to occur when the database comes back online will properly update the Streams table.

The above can even be taken to an extreme whereby the Streams table is only ever updated every X commits, thus avoiding the overhead of an additional insert for extremely low-latency systems. By revising the database insert statement slightly, the Streams table could be updated every 5 commits, for example by performing a modulus the StreamRevision by 5, e.g. (WHERE @StreamRevision % 5 = 0).

There are several other performance tuning techniques that can be done at the application level to reduce the number of IO operations such that the application becomes bound by CPU, memory speed, and network bandwidth, not to mention all of the streamlining that can be done for infrastructure services, operating system tweaks, and hardware.

So there we have it, full database consistency and integrity along with massive horizontal scalability—all without the use of foreign key constraints or transactions, etc. Event sourcing is an extremely powerful model that solves a number of significant problems in a very elegant and easy-to-understand manner.