(This document is based off discussions that can be found in this forum post)

Building indexes from existing data (especially if you add the additional constraint that you want to do it from a live dataset while concurrently permitting reads and writes) can be somewhat tricky, but it also ends up being a necessary primitive that many sufficiently-advanced layers will end up needing to support schema evolution. This document would go through a sketch here of how that might be done.

The naive approach could be run a transaction which looks like "for each existing record, insert the index" and then commit that transaction. But there are several problems with this solution:

- Transaction size and time limit. Sufficiently large data sets may take longer than five seconds to read, and additionally, a transaction can do at most `10 MB` of writes (actually, probably best to keep them under `1 MB`, if possible)
- Conflicts. This transaction will necessarily have a read conflict range that spans the entire data set, so any concurrent write will cause that index-building transaction to fail.(In theory, one could get around this by writing a key in FDB to "lock" the data, i.e., you would enforce client-side that any write first check to see if the lock key is set and then fail the write in your application rather than submitting it to be committed.)

So, it becomes essentially a necessity for any layer that implements some kind of index maintenance scheme to also implement a multi-transaction index builder. How to do that? The simplest way would be (1) start a transaction, (2) read some number of keys starting from the beginning of the data range, (3) write the appropriate index keys for the keys read, (4) commit the result, (5) go back to (1), but if (4) succeeds read from the last key written rather than the beggining of the range. This will usually work, with a few caveats:

1. There must be some mechanism to avoid using the unbuilt index to answer queries while simultaneously updating the index as data are inserted and deleted from the main data set. This usually necessitates being able to make indexes enter a "write-only" mode, and this state needs to be persisted somewhere, and then when the build completes, it can mark the index as readable by mutating that state.
2. This will only do the right thing if index updates are idempotent. For simple indexes, that assumption holds, so the only issue is that index updates might end up duplicating work. One example where that won't work can be the this: consider an index that counts how many times we see a value–a histogram index. (You might do this efficiently using our atomic mutation API.)
3. If whatever is building the index dies midway through, the entire index build must be restarted.

To solve (2) and (3), the solution can be designed to be slightly more sophisticated by atomically writing a key with the the index updates that states how

far through the index build has gotten. That way, if the index building job dies part way through, it can be resumed from where it left off rather than the beginning. And then once that information is available, it can also be used to solve (2) by updating a non-idempotent write-only index if and only if the key being added is within the built range. If it's outside the built range, the index builder will pick it up. If it's within the range, then the initial write will index it. Either way, it is indexed exactly once.

So, the final question is then how to implement parallelism. To do that, the aforementioned technique can be extended to write multiple keys instead of just one saying which ranges of the main dataset have been built. Then multiple index builders can work on different ranges at once and use those keys to avoid stepping on each other. The details are a little complicated, but that's the general idea. If the distribution of the data is roughly known, it would be straightforward to pick boundaries to shard upon. FDB also offers an API that can be used to find the boundary keys of the underlying shards. Note that the boundaries are not required for correctness, just performance, so once boundaries are chosen, they can be used for the entirely of the build (or even change them at will depending on how the progress of each ranged is being tracked), and things will just "work".