ArchitectureInternals

General

- Configuration file is parsed by DatabaseDescriptor (which also has all the default values, if any)
- Thrift generates an API interface in Cassandra.java; the implementation is CassandraServer, and CassandraDaemon ties it together (mostly: handling commitlog replay, and setting up the Thrift plumbing)
- CassandraServer turns thrift requests into the internal equivalents, then StorageProxy does the actual work, then CassandraServer turns the results back into thrift again
    - CQL requests are compiled and executed through QueryProcessor. Note that as of 1.2 we still support both the old cql2 dialect and the cql3, in different packages.
- StorageService is kind of the internal counterpart to CassandraDaemon. It handles turning raw gossip into the right internal state and dealing with ring changes, i.e., transferring data to new replicas. TokenMetadata tracks which nodes own what arcs of the ring. Starting in 1.2, each node may have multiple Tokens.
- AbstractReplicationStrategy controls what nodes get secondary, tertiary, etc. replicas of each key range. Primary replica is always determined by the token ring (in TokenMetadata) but you can do a lot of variation with the others. SimpleStrategy just puts replicas on the next N-1 nodes in the ring. NetworkTopologyStrategy allows the user to define how many replicas to place in each datacenter, and then takes rack locality into account for each DC – we want to avoid multiple replicas on the same rack, if possible.
- MessagingService handles connection pooling and running internal commands on the appropriate stage (basically, a threaded executorservice). Stages are set up in StageManager; currently there are read, write, and stream stages. (Streaming is for when one node copies large sections of its SSTables to another, for bootstrap or relocation on the ring.) The internal commands are defined in StorageService; look for `registerVerbHandlers`.
- Configuration for the node (administrative stuff, such as which directories to store data in, as well as global configuration, such as which global partitioner to use) is held by DatabaseDescriptor. Per-KS, per-CF, and per-Column metadata are all stored as parts of the Schema: KSMetadata, CFMetadata, ColumnDefinition. See also ConfigurationNotes.

Some historical baggage

- Some classes have misleading names, notably ColumnFamily (which represents a single row, not a table of data) and, prior to 2.0, Table (which was renamed to Keyspace).

Write path

- StorageProxy gets the nodes responsible for replicas of the keys from the ReplicationStrategy, then sends RowMutation messages to them.
    - If nodes are changing position on the ring, "pending ranges" are associated with their destinations in TokenMetadata and these are also written to.
    - ConsistencyLevel determines how many replies to wait for. See WriteResponseHandler.determineBlockFor. Interaction with pending ranges is a bit tricky; see https://issues.apache.org/jira/browse/CASSANDRA-833
    - If the FailureDetector says that we don't have enough nodes alive to satisfy the ConsistencyLevel, we fail the request with UnavailableException
    - When performing atomic batches, the mutations are written to the batchlog on two live nodes in the local datacenter. If the local datacenter contains multiple racks, the nodes will be chosen from two separate racks that are different from the coordinator's rack, when possible. If only one other node is alive, it alone will be used, but if no other nodes are alive, an UnavailableException will be returned unless the consistency level is ANY. If the cluster has only one node, it will write the batchlog entry itself. The batchlog is contained in the system.batchlog table.
    - If the FD gives us the okay but writes time out anyway because of a failure after the request is sent or because of an overload scenario, StorageProxy will write a "hint" locally to replay the write when the replica(s) timing out recover. This is called HintedHandoff. Note that HH does not prevent inconsistency entirely; either unclean shutdown or hardware failure can prevent the coordinating node from writing or replaying the hint. ArchitectureAntiEntropy is responsible for restoring consistency more completely.
    - Cross-datacenter writes are not sent directly to each replica; instead, they are sent to a single replica with a parameter in MessageOut telling that replica to forward to the other replicas in that datacenter; those replicas will respond diectly to the original coordinator.
- On the destination node, RowMutationVerbHandler calls RowMutation.apply() (which calls Keyspace.apply()) to make the mutation. This has several steps. First, an entry is appended to the CommitLog (potentially blocking if the CommitLog is in batch sync mode or if the queue is full for periodic sync mode.) Next, the Memtable, secondary indexes (if applicable), and row cache are updated (sequentially) for each ColumnFamily in the mutation.
- When a Memtable is full, it is asynchronously sorted and written out as an SSTable by ColumnFamilyStore.switchMemtable
    - "Fullness" is monitored by MeteredFlusher; the goal is to flush quickly enough that we don't OOM as new writes arrive while we still have to hang on to the memory of the old memtable during flush
    - When Memtables are flushed, a check is scheduled to see if a compaction should be run to merge SSTables. CompactionManager manages the queued tasks and some aspects of compaction.
        - Making this concurrency-safe without blocking writes or reads while we remove the old SSTables from the list and add the new one is tricky. We perform manual reference counting on sstables during reads so that we know when they are safe to remove, e.g., ColumnFamilyStore.getSSTablesForKey.
        - Multiple CompactionStrategies exist. The original, SizeTieredCompactionStrategy, combines sstables that are similar in size. This can result is a lot of wasted space in overwrite-intensive workloads. LeveledCompactionStrategy provides stricter guarantees at the price of more compaction i/o; see http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra and http://www.datastax.com/dev/blog/when-to-use-leveled-compaction
- See ArchitectureSSTable and ArchitectureCommitLog for more details

Read path

- StorageProxy.fetchRows() creates a ReadExecutor for each of the read commands.
  - Depending on the query type, the read commands will be SliceFromReadCommands, SliceByNamesReadCommands, or a RangeSliceCommand. Secondary index queries are covered by RangeSliceCommand.
  - The ReadExecutor determines the replicas (endpoints) to read from by processing the row (partition) key with the ReplicationStrategy for the keyspace
  - Endpoints are filtered to contain only those that are currently up/alive
    - If there are not enough live endpoints to meet the consistency level, an UnavailableException reponse is returned
  - Endpoints are sorted by "proximity".
    - With a SimpleSnitch, proximity directly corresponds to proximity on the token ring.
    - With implementations based on AbstractNetworkTopologySnitch (such as PropertyFileSnitch), endpoints that are in the same rack are always considered "closer" than those that are not. Failing that, endpoints in the same data center are always considered "closer" than those that are not.
    - The DynamicSnitch, typically enabled in the configuration, wraps whatever underlying snitch (such as SimpleSnitch and PropertyFileSnitch) so as to dynamically adjust the perceived "closeness" of endpoints based on their recent performance. This is an effort to try to avoid routing more traffic to endpoints that are slow to respond.
  - The closest node (as determined by proximity sorting as described above) will be sent a command to perform an actual data read (i.e., return data to the co-ordinating node).
  - As required by consistency level, additional nodes may be sent digest commands, asking them to perform the read locally but send back the digest only.
    - For example, at replication factor 3 a read at consistency level QUORUM would require one digest read in additional to the data read sent to the closest node. (See ReadCallback, instantiated by StorageProxy)
    - If read repair is (probabilistically) enabled (depending on read_repair_chance and dc_local_read_repair_chance), remaining nodes responsible for the row will be sent messages to compute the digest of the response.
  - A specific implementation of AbstractReadExecutor is created depending on whether or not speculative retry should be applied. In the normal case, a SpeculatingReadExecutor will be created.
- On the data node, ReadVerbHandler gets the data from CFS.getColumnFamily, CFS.getRangeSlice, or CFS.search for single-row reads, seq scans, and index scans, respectively, and sends it back as a ReadResponse
  - For single-row requests, we use a QueryFilter subclass to pick the data from the Memtable and SSTables that we are looking for.
  - If the row cache is enabled, it is first checked for the requested row (in ColumnFamilyStore.getThroughCache). The row cache will contain the full partition (storage row), which can be trimmed to match the query. If there is a cache hit, the coordinator can be responded to immediately.
  - The set of SSTables to read data from are narrowed at various stages of the read by the following techniques:
    - If a row tombstone is read in one SSTable and its timestamp is greater than the max timestamp in a given SSTable, that SSTable can be ignored
    - If we're requesting column X and we've read a value for X from an SSTable at time T1, any SSTables whose maximum timestamp is less than T1 can be ignored
    - If a slice is requested and the min and max column names for a given SSTable do not fall within the slice, that SSTable can be ignored
    - The BloomFilter for each SSTable can be checked to definitively determine that a given row is not present in the SSTable. A small percentage of checks will result in a false positive (claiming that the row does exist when it actually does not). The approximate rate of false positives is configurable in order to control the size of bloom filters.
  - To locate the data row's position in SSTables, the following sequence is performed:
    - The key cache is checked for that key/sstable combination
    - If there is a cache miss, the IndexSummary is used. The IndexSummary is a sampling of the primary on-disk index; by default, every 128th parition (storage row) gets an entry in the summary. A binary search is performed on the index summary in order to get a position in the on-disk index to begin scanning for the actual index entry.
    - The primary index is scanned, starting from the above location, until the key is found, giving us the starting position for the data row in the sstable. This position is added to the key cache. In the case of bloom filter false positives, the key may not be found.
  - Some or all of the data is then read:
    - If we are reading a slice of columns, we use the row-level column index to find where to start reading, and deserialize block-at-a-time (where "block" is the group of columns covered by a single index entry) so we can handle the "reversed" case without reading vast amounts into memory
    - If we are reading a group of columns by name, we use the column index to locate each column
    - If compression is enabled, the block that the requested data lives in must be uncompressed
  - Data from Memtables and SSTables is then merged (primarily in CollationController)
    - The column readers provide an Iterator interface, so the filter can easily stop when it's done, without reading more columns than necessary
    - Since we need to potentially merge columns from multiple SSTable versions, the reader iterators are combined through a ReducingIterator, which takes an iterator of uncombined columns as input, and yields combined versions as output
  - If row caching is enabled, the row cache is updated in ColumnFamilyStore.getThroughCache()
- Back on the coordinator node, responses from replicas are handled:
  - If a replica fails to respond before a configurable timeout, a ReadTimeoutException is raised
  - If responses (data and digests) do not match, a full data read is performed against the contacted replicas in order to guarantee that the most recent data is returned
  - If the read command is a SliceFromReadCommand and at least one replica responded with the requested number of cells (or cql3 rows), but after merging responses fewer than the requested number of cells/rows remain, the query will be retried with a higher requested cell/row count.
  - Once retries are complete and digest mismatches resolved, the coordinator responds with the final result to the client

In addition:

- At any point if a message is destined for the local node, the appropriate piece of work (data read or digest read) is directly submitted to the appropriate local stage (see StageManager) rather than going through messaging over the network.
- The fact that a data read is only submitted to the closest replica is intended as an optimization to avoid sending excessive amounts of data over the network. A digest read will take the full cost of a read internally on the node (CPU and in particular disk), but will avoid taxing the network.

Deletes

- See DistributedDeletes

Gossip

- based on "Efficient reconciliation and flow control for anti-entropy protocols:" http://www.cs.cornell.edu/home/rvr/papers/flowgossip.pdf
- See ArchitectureGossip for more details

Failure detection

- based on "The Phi accrual failure detector:" http://vsedach.googlepages.com/HDY04.pdf

Further reading

- The idea of dividing work into "stages" with separate thread pools comes from the famous SEDA paper: http://www.eecs.harvard.edu/~mdw/papers/seda-sosp01.pdf
- Crash-only design is another broadly applied principle. Valerie Henson's LWN article is a good introduction
- Cassandra's distribution is closely related to the one presented in Amazon's Dynamo paper. Read repair, adjustable consistency levels, hinted handoff, and other concepts are discussed there. This is required background material: http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html. The related article on article on eventual consistency is also relevant. Jeff Darcy's article on Availability and Partition Tolerance explains the underlying principle of CAP better than most.
- Cassandra's on-disk storage model is loosely based on sections 5.3 and 5.4 of the Bigtable paper.
- Aaron Morton gave a talk on Cassandra Internals at the 2013 Cassandra Summit.
- Facebook's Cassandra team authored a paper on Cassandra for LADIS 09, which has now been annotated and compared to Apache Cassandra 2.0.