Apache Cassandra v2.1 Architecture Analysis

1 Topology
2 Data model
3 Partitioning
4 Replication
5 Availability
6 Storage engine
7 Durability
8 Process model
9 Write path
10 Read path
11 Memory usage

Abstract

Cassandra (C*) is a distributed, horizontally scalable, eventually consistent (letters AP within CAP theorem), hybrid between a key-value and a column-oriented database with no single points of failure (SPF). C* written in Java programming language and supported by the Apache Community. Best use is for applications that require a lot of writes and manage large streams of non-transactional data, such as application logs, real-time analytics and statistics, sequential data.

This paper provides an architectural discussion of C* DBMS design algorithms and principles, including system topology and data connections, data partitioning and replication, service availability and durability, read and write paths, memory usage, storage engine design.

1 Topology

Physical. C* system is based on decentralized structured peer-to-peer physical topology, where all *peers* (*storage hosts*, *nodes*) communicate symmetrically to each other and have equal roles: store and manage user data. In general, nodes are heterogeneous. This in pair of the data replication and availability guarantees to have no SPFs.

Some of the nodes called *seeds*. Seeds are nodes that are discovered via an external mechanism (either from static configuration or from a configuration service) and are known to all nodes. Seeds designation and only purpose is to prevent system logical partitions. Logical partition is the situation when two or more nodes join the system at the same time and they don't immediately aware of each other. Because of nodes eventually reconcile their system membership with a seed, logical partitions are highly unlikely. Seed nodes are *not* a single point of failure, nor do

they have any other special purpose in system operations beyond the bootstrapping of nodes.

Logical. There is a special configuration manager that inform C* about the network topology called s*nitch*. Snitches provide possibility to group nodes into logical units called *racks* and *data centers* as illustrated on Figure 1.1. There are many predefined snitches but administrator can define a new one.

The main goal of such structure is to provide additional levels of fault tolerance. Data centers unite nodes, for example, from one geographical region or building of computation machines. But it is not necessarily a physical data center. Logical data centers unite nodes of the same type (transactional, analytics, search, etc.) and used for replication purposes. Let's assume datacenter as a building consists of rooms. So, the idea of racks is to unite nodes according their rooms locality to reduce impact of emergency situation on the whole system state . Racks are determined by local area network links (data connections), power supply, fire protection constructions and etc. Through snitches requests are routed efficiently and C* distributes data according to data centers and racks.

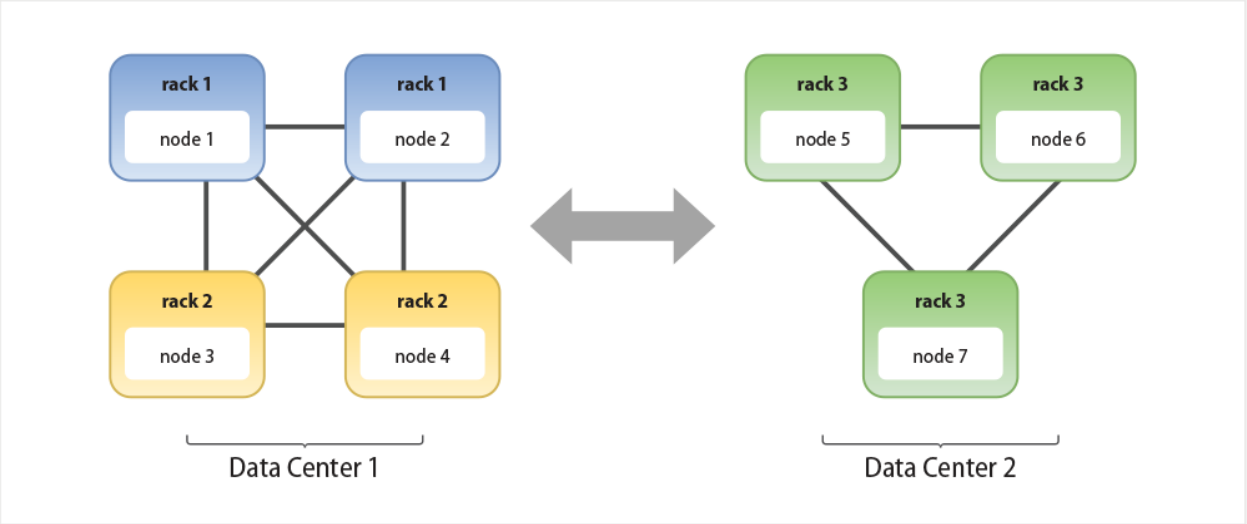

Figure 1.1. Logical topology

From a higher level of abstraction, C* system integrally is based on distributed hash table (DHT) data structure. DHT provides look up service similar to a hash table. Any $< key, value >$pair search requests is actively addressed by DHT using constant amount of time. Such requests are distributed across system nodes through a key-based routing.

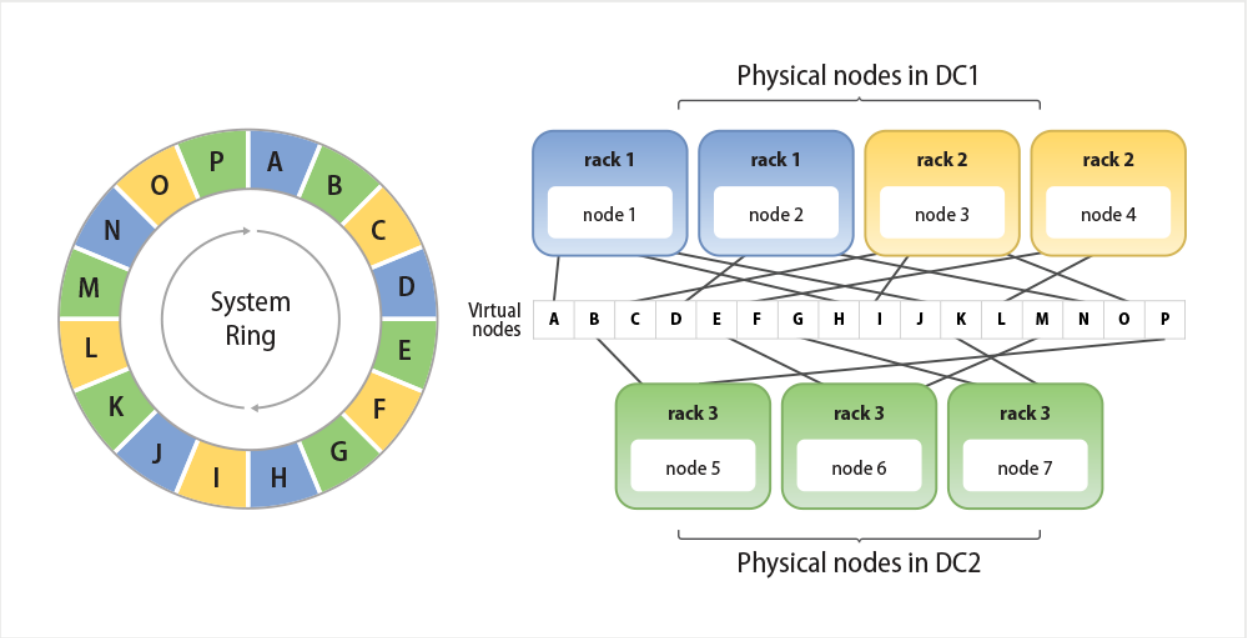

Figure 1.2. System ring topology

Each node participating in a DHT is responsible for a given ordered range of keys, where range upper-bound is called *token* and aware about the tokens of other nodes in the system. One physical node can own N distinct tokens and transparently splitted into N *virtual nodes* (*vnodes*). So, the virtual node is a pair $< node, token >$. Since the whole range of keys in hash table are bounded C* highest level topology can be visualized as ring. Ring for logical topology

(Figure 1.1) is illustrated on Figure 1.2 (letters are ranges). So, horizontal scaling is achieved by adding new nodes to an existing ring and assigning needed tokens for them. Note: more about data distribution in Section 3.

2 Data model

C* logical data model is a hybrid between a key-value and a column-oriented data models. To use such data model C* exposes to client applications a special data definition and manipulation language called CQL (Cassandra Query Language). It allows client applications to structure data in the form of such concepts as *keyspaces* and *tables*.

Keyspace is the outermost grouping container for data similar to a schema in a relational database. Keyspace consists of tables which share the same options. Typically, a system has one keyspace per application.

Tables are second level containers which organize data into *partitions* also known as *wide rows*.

A partition is a collection of *rows* which themselves are fetched data units. Client applications access one or more partitions and rows.

A row is a data structure that is a collection of alphabetically ordered (by name) columns that have the same *primary key*. All rows in partition have the same primary key. Insertion, update, and deletion operations on rows sharing the same partition key for a table are performed atomically and in isolation.

A column is the smallest increment of data which contains a name - the column string identifier, and a value - data of defined type stored in the column.

Type is a classification of data that determines the possible values for that data. There are many predefined types provided by C* (string, integer, boolean, blob, etc.) but users can define their own types.

The primary key is a one or more columns that uniquely identify a row in a table. Primary key may be single or compound. A compound primary key consists of the partition key and one or more additional columns that determine per-partition clustering and called *clustering columns*. Single, or just primary key, is consists only of the partition key.

Partition key is the first column declared in the primary key definition. Partition key determines which node stores the data. It is responsible for data distribution across the nodes (see Section 3). Partition key itself can be composite. Composite partition key is a partition key consisting of multiple columns. If, for example, composite partition key equals to $< column_1, column_2 >$than C* will store columns having the same $column\_1$ but a different $column\_2$ on different nodes, and columns having the same $column\_1$ and $column\_2$ on the same node.

Clustering is a storage engine process that physically stores multiple rows in lexicographic order on disk within a single partition. Rows is clustered by the clustering columns. These columns form logical sets inside a partition to facilitate retrieval.

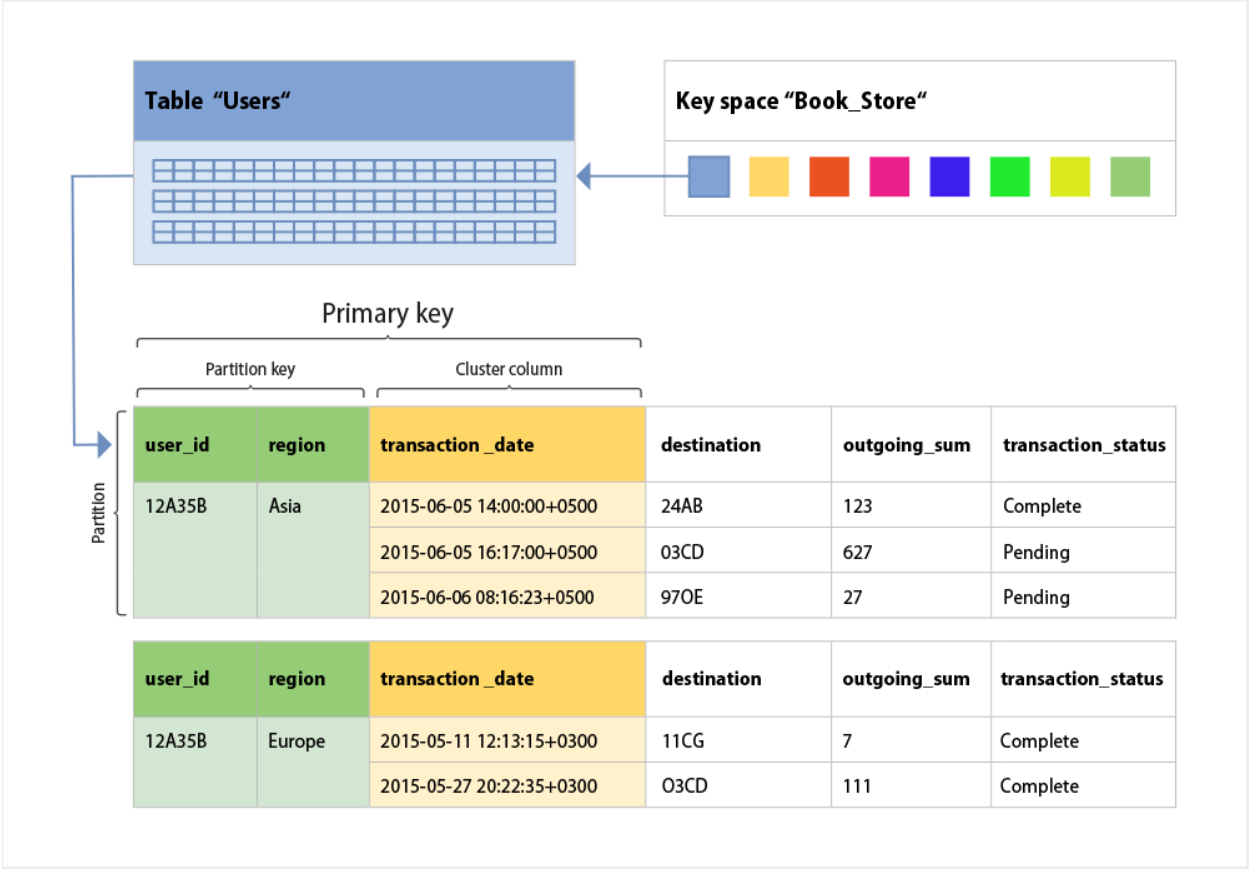

Figure 2.1. Logical data model

3 Partitioning

Data partitioning determines how data is distributed across the nodes in the system. C* system is a horizontal partitioning database. Horizontal partitioning is a transparent for client application process, performed by the system, of breaking a logically combined data set into set's elements (partitioning elements) and distributing such elements across nodes. In C*, data distribution and replication (see Section 4) go together and performed by rows (see Section 2). Replicas are copies of rows. When row is first written, it is also referred to as a replica (first replica).

As mentioned in Section 1 C* is a DHT, and therefore uses hashing to distribute and load balancing data across nodes. C* partitioning process is based on consistent hashing algorithm which determine destination node for the first replica by replica partition key and nodes tokens.

Consistent hashing algorithm is based on a predefined hash function H. This function is defined on some set of partition keys and all its values are from monotonic range $[a, z]$. The whole range $c$is divided into unique monotonic intervals $(a, b], (b, c], \ldots, (k, z]$ or $[a, k)$. Intervals are assigned to physical nodes (or virtual nodes) and its upper-bounds $b, c, \ldots, z$ or a called tokens. C* places the data on each node according to the value of the partition key and the interval that the node is responsible.

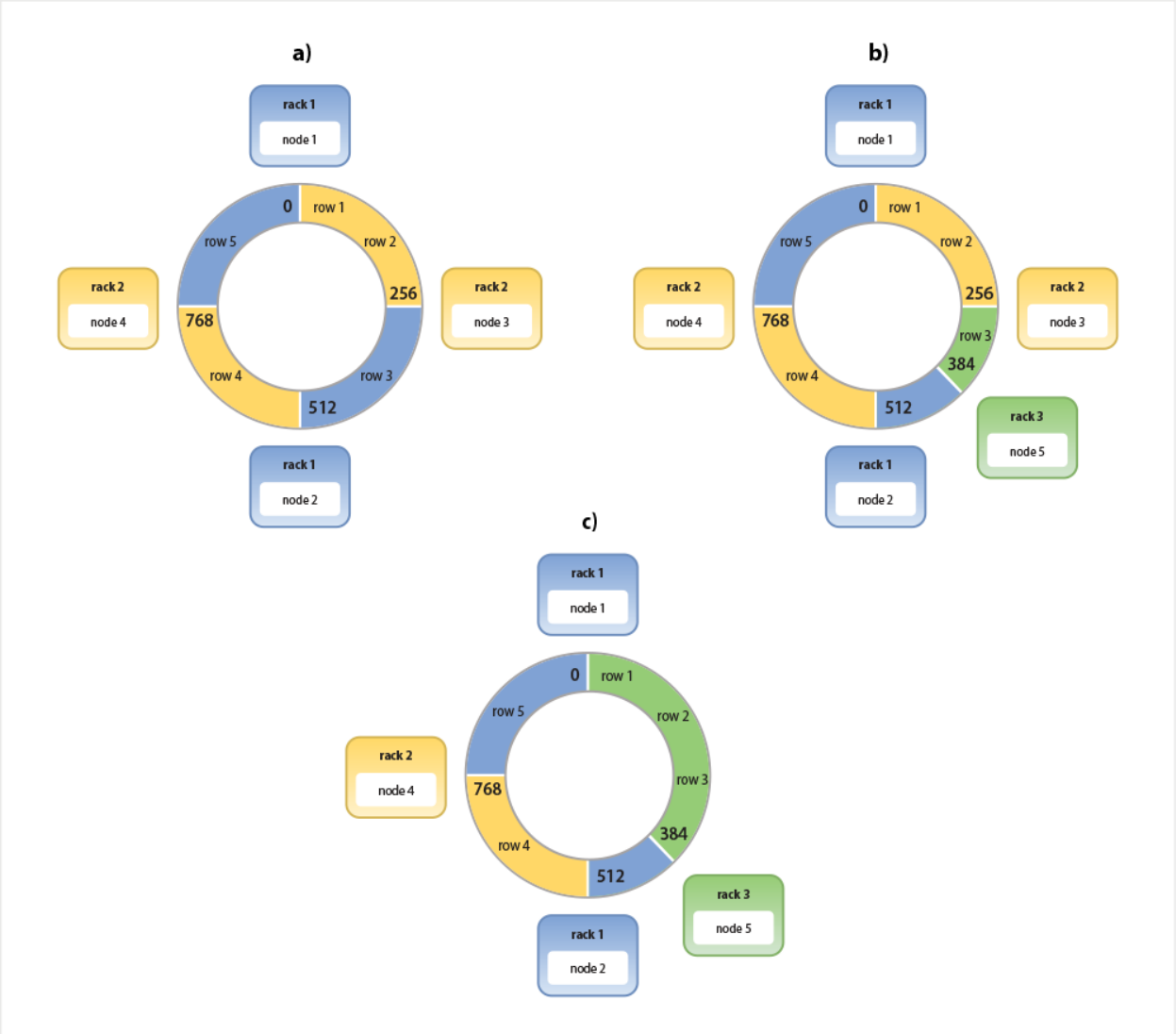

Figure 3.1. a) original ring; b) with add node; c) with removed node

Define function Pthat for each row R returns row partition key. Consistent hashing partitioning is based on mapping each first replica R to some node by calculating replica hash value: $H(P(R)) = h$. To find where a row R should be placed, the system walks clockwise the ring until encounters the first node with a token $t \in [a, z]$ such that $t \geq h$or $t = a$. The result is that each node contains all the rows located between its token and the previous node token.

On the Figure 3.1a is illustrated system ring of four nodes n_1, n_2, n_3, n_4 with tokens $t_1 = 0, t_2 = 256, t_3 = 512, t_4 = 768$. Node stores next rows:

n_1: $R_5, H(P(R_5)) = 842$;

n_2: $R_1, H(P(R_1)) = 128$ and $R_2, H(P(R_2)) = 255$;

n_3: $R_3, H(P(R_3)) = 300$;

n_4: $R_4, H(P(R_4)) = 666$;

When a new node $n_5, t = 384$ (Figure 3.1b) is added into the ring then a part of data will be redistributed and row R_3will be remapped to node n_5.

When an existing node $n_2$is removed (Figure 3.1c) from the ring then a part of data will be redistributed and rows R_1, R_2will be remapped to node n_5.

In C* function H is called partitioner. C* offers the following partitioners:

- *Murmur3Partitioner*: uniformly distributes rows across the system based on MurmurHash hash values.
- *RandomPartitioner*: uniformly distributes rows across the system based on MD5 hash values.
- *ByteOrderedPartitioner*: keeps an ordered distribution of rows lexically by key bytes and allows range scans over rows.

Virtual nodes paradigm helps to distribute data among the nodes more evenly since the placement of a row is determined by the partition key hash within many smaller ring ranges belonging to each node.

4 Replication

Replication is the process of storing copies of rows on different nodes. Rows copies are called *replicas*. When row is first written, it is also referred to as a replica (*first replica*). Set of nodes that store replicas with the same partition key called *replica nodes*. The number of replicas is defined separately for each keyspace and each data center. It can be M replicas in one data center and N replicas in other for the same keyspace. Such approach called *asymmetrical replication grouping*. The total number of replicas across the system is referred to as the *replication factor*. Replication factor is less than the total number of system nodes but greater than zero. But it is possible to increase the replication factor and add the desired number of nodes afterwards. Replication factor 1 mean that there is only first replica. When replication factor is greater than the total number of system nodes, writes are rejected, but reads are served as long as the desired *consistency level* can be met. Consistency level specifies on how many replicas the write/read must succeed before returning an acknowledgement to the client application (see Sections 5, 6, 7). There is no leading, or master, replica - all replicas have equals roles. In addition to the replication factor system uses *replica placement strategy.* Strategy sets the distribution of the replicas across the nodes in the system based on topology snitch.

Partitioning and replication go together. And there are two strategies to place replicas: SimpleStrategy and NetworkTopologyStrategy.

SimpleStrategy is used for simple single data center systems. First replica are placed on a node determined by the partitioner. Remaining replicas are placed on the next nodes clockwise in the ring without considering rack or data center location (Figure 4.1).

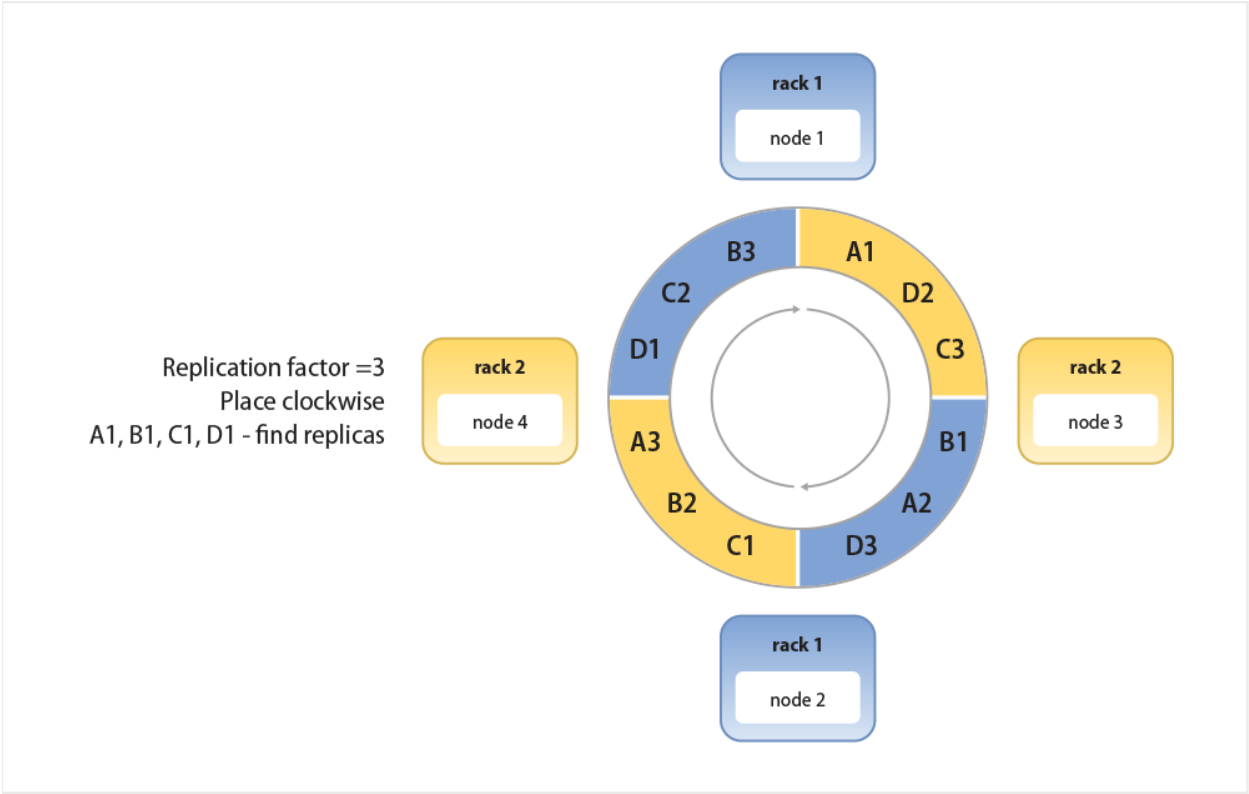

Figure 4.1. Simple replication strategy

NetworkTopologyStrategy determines replica placement independently within each data center. First replica are placed on a node determined by the partitioner. Remaining replicas are placed by walking the ring clockwise until a node in a different rack is found. If no such node exists, additional replicas are placed in different nodes in the same rack (Figure 4.2).

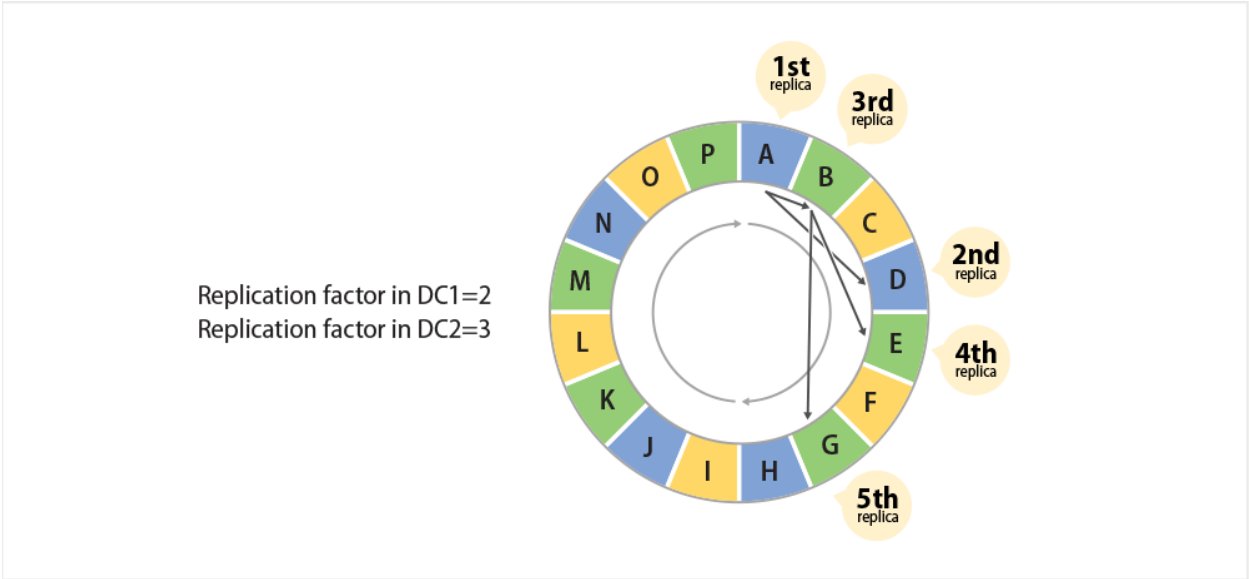

Figure 4.2. Simple replication strategy

5 Availability

C* values availability and partitioning tolerance (AP in CAP theorem). Tradeoffs between consistency and latency are tunable in C*. It is possible to get strong consistency with an increased latency. AP is achieved by the techniques discussed below in this section.

Membership & fault tolerance. All nodes in C* system have equal roles: all nodes accepts requests to manipulate data. Each node know about the system view, so client application can connect to any node and discover information about all system nodes (up/down nodes) and system structure (data centers, racks, nodes tokens). Once client application gets such information it proceed to communicate with node that host requested data. Such node is called coordinator (for more about writes and reads see Sections 9, 10).

The naive way to share system view among nodes is to send messages from each node to every other nodes. But such approach loads physical communication channels. To keep information about the system view up-to-date C* uses *gossip* protocol. Gossip is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know about, so all nodes quickly learn about all other nodes in the system. Such process is called *gossip process*.

Gossip protocol allows C* next:

- to bootstrap new nodes;
- to establish nodes memberships;
- to detect failures;
- to avoid routing client application requests to unreachable nodes;
- to avoid routing requests to nodes that are alive, but performing poorly.

Each node runs gossip process every second and exchanges state messages with up to three other random live nodes in the system. A gossip message has a version associated with it, so that during a gossip exchange, older information is overwritten with the most current state for a particular node.

Gossip process is illustrated on Figure 5.1. Node A starting gossip exchange sends *gossip digest sync message* to node B. A gossip digest consists of node endpoint address, generation number and maximum version that has been seen for the node. Generation monotonically increases every time node is rebooted. Version increases for generation every time node state is changed, so greater version means most recent node state. Sync message contains digests of all nodes known to node A. Node B receiving sync message will examine it and reply with *gossip digest ack message*, which includes two parts: digests of nodes for which node B does not have actual information, and states of nodes about B know more recent information than A. Receiving ack message from B, node A sends *gossip digest ack2 message* to B with synced states of nodes known to both of nodes $B$and A and gossip round is complete.

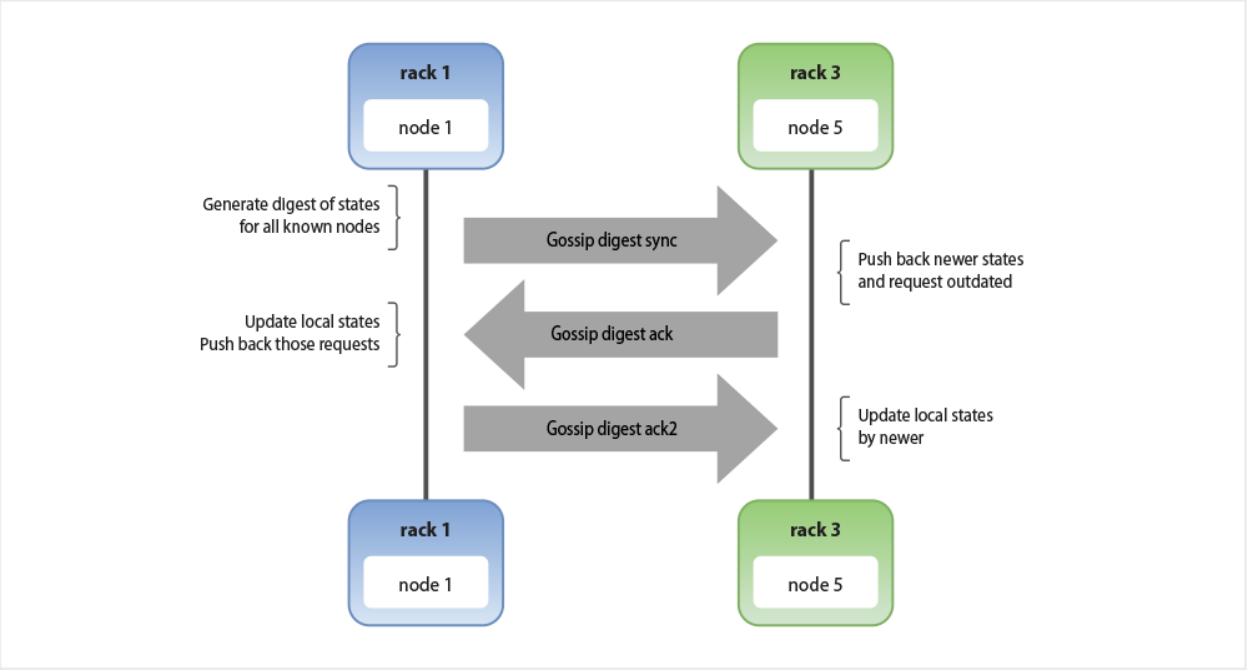

Figure 5.1. Gossip process between two nodes

Eventual consistency. Node failures can result from various causes such as hardware failures and network outages, which in pair of replication affects on data *consistency*. Consistency refers to how up-to-date and synchronized a row is on all of its replicas. If all replicas is the same than they are *converged*, or achieved *replica convergence.* C* offers *eventual consistency* model and extends it by offering *tunable consistency* feature.

Eventual consistency model achieves data high availability that informally guarantees that, if no new updates are made to a given row, eventually all accesses to replicas will return the last updated value. In eventual consistency system writes may not update all replicas at the same time, but this can be done lately.

In order to ensure replica convergence, C* must reconcile differences between replicas. This is achieved by the following techniques:

- *hinted handoff* process;
- *anti-entropy node repair* process;
- *read repair.*

To reconcile differences C* uses approach called "last writer wins" and uses timestamp associated with each column. Reconciliation is performed by comparing columns timestamps. Older column is replaced by younger. Thus, it requires all system nodes have the same time, synchronized in some way.

Tunable consistency. Tunable consistency is a feature that allows client applications to specify the number of replica nodes which must acknowledge the write or read success to the client application. This number is called *consistency level.* It is possible to configure consistency level globally on a system, data center, or individual I/O operation basis. Client application can tune consistency either for writes or reads operations.

The read/write consistency level specifies how many replicas must respond to a read/write request before returning data to the client application.

To get fully consistent reads client application must use such read and write consistency levels that $(nodes\_written + nodes\_read) > replication\_factor$.

Anti-entropy node repair. Anti-entropy (AE) node repairing ensures that all replicas is consistent and repair inconsistencies on a node that has been down for a while. Thus, repairing should be performed manually during normal operation as part of regular system maintenance, or during node recovery after a failure or on a node that has been down for a while, or on a nodes that contain data that is not read frequently. In general, this process is I/O and CPU intensive.

So, AE means comparing all the replicas of each piece of data that exist (or are supposed to) and updating each replica to the newest version. To detect the inconsistencies between replicas faster and to minimize the amount of transferred data, C* uses Merkle trees. A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children. The principal advantage of Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire replica node data set.

C* builds Merkle trees per table, and they are not maintained for longer than it takes to send them to neighboring replica nodes. Instead, the trees are generated as snapshots of the dataset during major compactions (see Section 6): this means that excess data might be sent across the network, but it saves local disk IO, and is preferable for very large datasets.

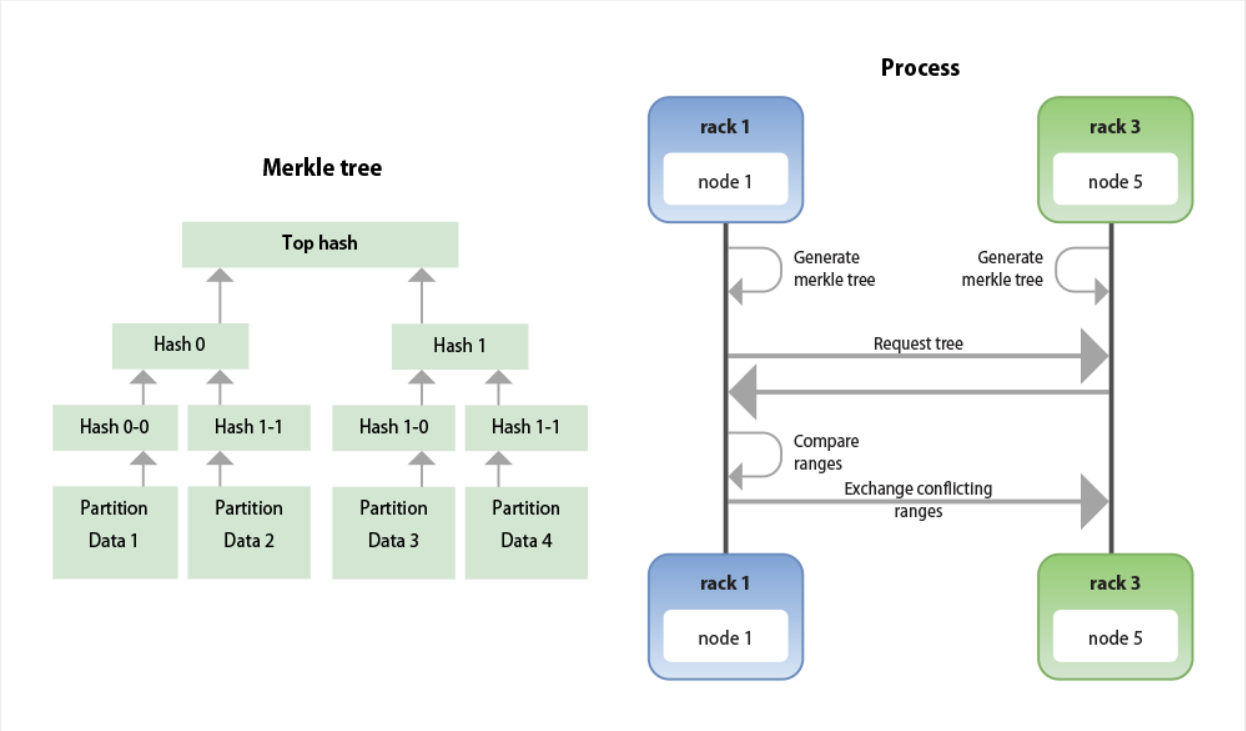

Figure 5.2. Anti-entropy process

To optimize anti-entropy process and increase data consistency two other techniques are used: hinted handoff and read repair. They are performed in automatic regime during processing client application requests.

Hinted handoff. Hinted handoff (HH) is a feature that improve data consistency when a replica node is not available, due to network issues or other problems, to accept a replica from a successful write operation. C* allows possibility to enable or disable HH.

HH process is illustrated on Figure 5.3. During a write operation, when HH is enabled and consistency can be met, the coordinator stores a hint about dead replicas in the local system hints table under either of these conditions:

- A replica node for the row is known to be down ahead of time;
- A replica node does not respond to the write request.

When the system cannot meet the consistency level specified by the client, C* does not store a hint.

A hint indicates that a write needs to be replayed to one or more unavailable replica nodes. The hint consists of:

- The location of the replica that is down;
- Version metadata;
- The actual data being written.

Hints are not saved for forever but for a certain amount of time which is configurable.

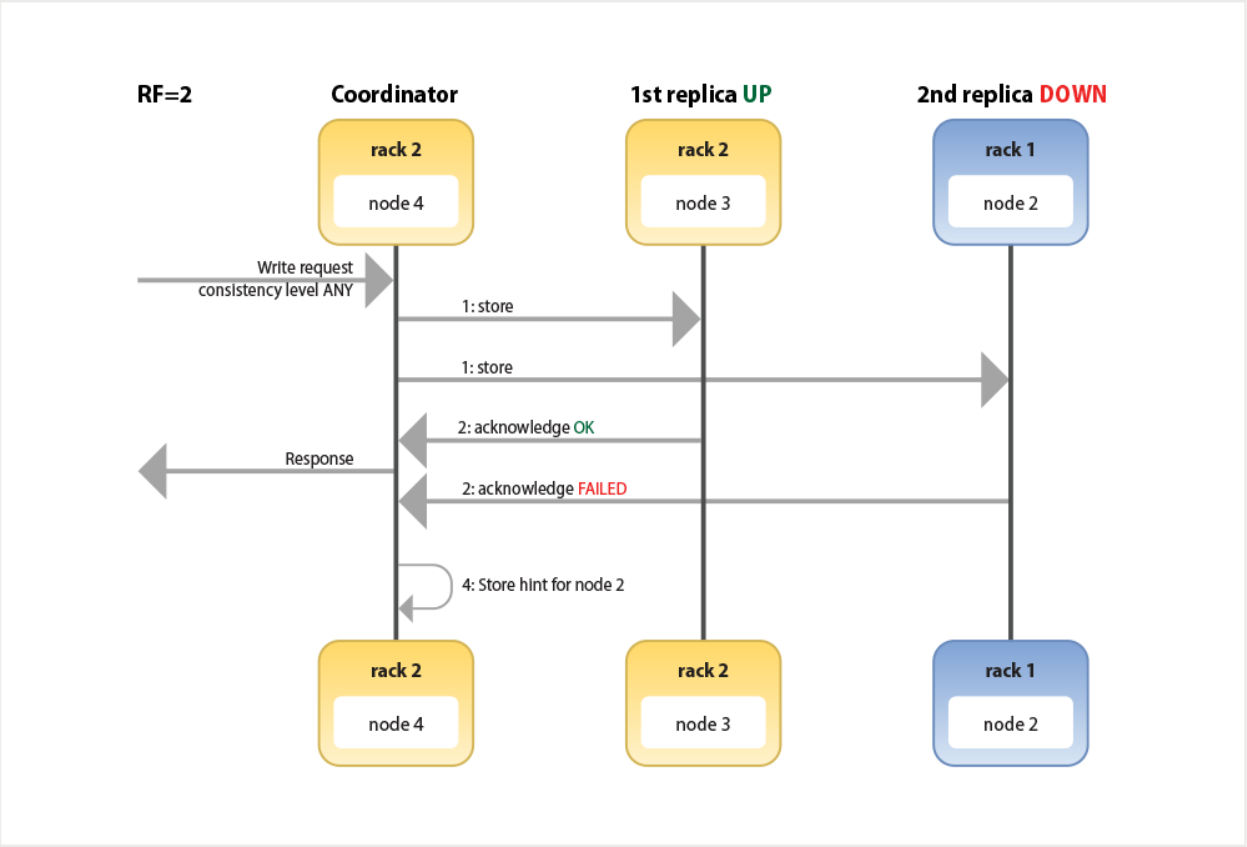

Figure 5.3. Hinted handoff process

HH is not a process that guarantees successful write operations, except when a client application uses a consistency level of *ANY*.

ANY level means that write must be written to at least one node. If all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, and ANY write is not readable until the replica nodes for that partition have recovered.

Read repair. Read repair can be treated as a real-time anti-entropy process. Read repair means that when a query is made against a given key, C* perform a digest query against all the replica nodes of the partition key and push the most recent version to any out-of-date replicas. If consistency level is lower than the number of all replicas was specified, this is done in the background after returning the data from the closest replica nodes to the client application,; otherwise, it is done before returning the data.

Coordinator sends request to replica nodes to get digests for partition key. Then coordinator compares digests and If any mismatch it re-requests the full data set from replica nodes, reconcile differences between replicas, updates replica nodes and blocks until out-of-date replica nodes respond (if performed not in background). After that coordinator return merged result to the client application.

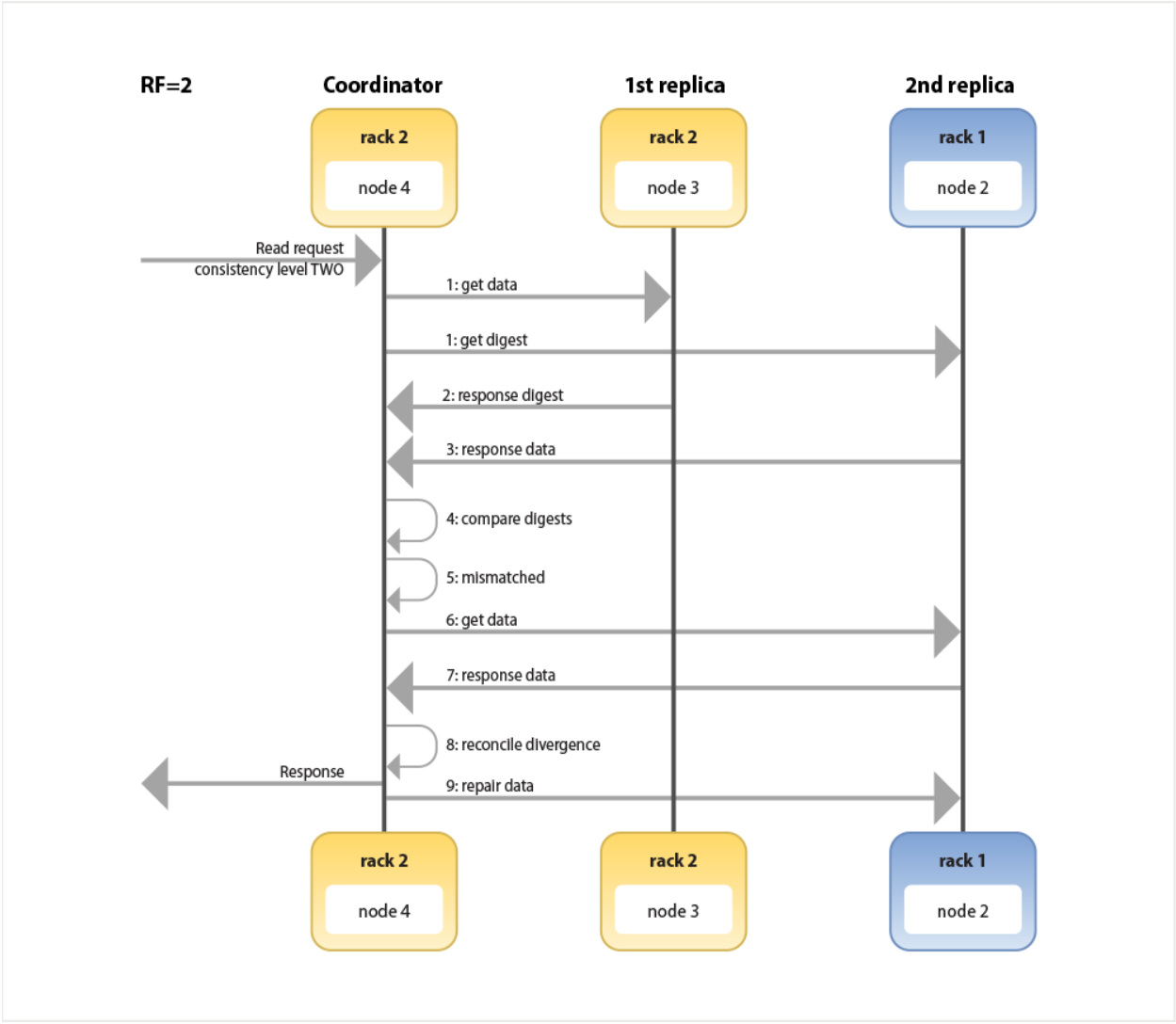

Figure 5.4. Read repair process

6 Storage engine

C* storage engine is based on Log-Structured Merge-Tree (LSM-tree) technique to store and manage data, and consists of the components illustrated on Figure 6.1.

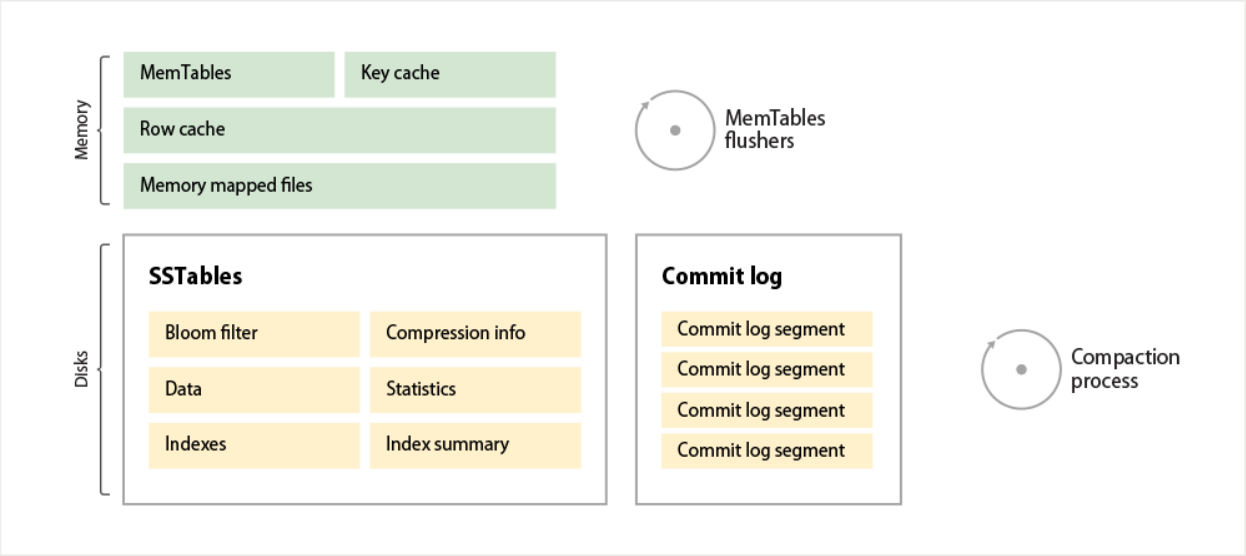

Figure 6.1. LSM-Tree components

All components is divided into two types: in-memory and disk allocated. Storage is based on four key concepts:

- *Memory table (MemTable)*
- *Sorted string table (SSTable)*
- *Memtable flushing process*
- *Compaction process*

In addition, storage implementation uses optimizations to improve operations performance:

- *Key cache*
- *Row cache*
- *Indexes*
- *Secondary indexes*
- *Bloom filters*

The idea of LSM-tree technique is the following:

- MemTables only allocated in memory;
- SSTables respond for disk reads and writes;
- All writes go in memory and are held by MemTables;
- All reads firstly go to MemTables, and if read is not successful - go to SSTables;
- MemTable periodically are flushed to disk and stored as new SSTable there;
- Periodically SSTables are united into new one, old SSTables are removed.

Durability is achieved by write-ahead-logging (see Section 7) and stands along the LSM-tree data structure.

It is possible to specify a multiple data locations where C* will store keyspaces and tables.

All concepts, their relationships and interactions will be discussed below in details.

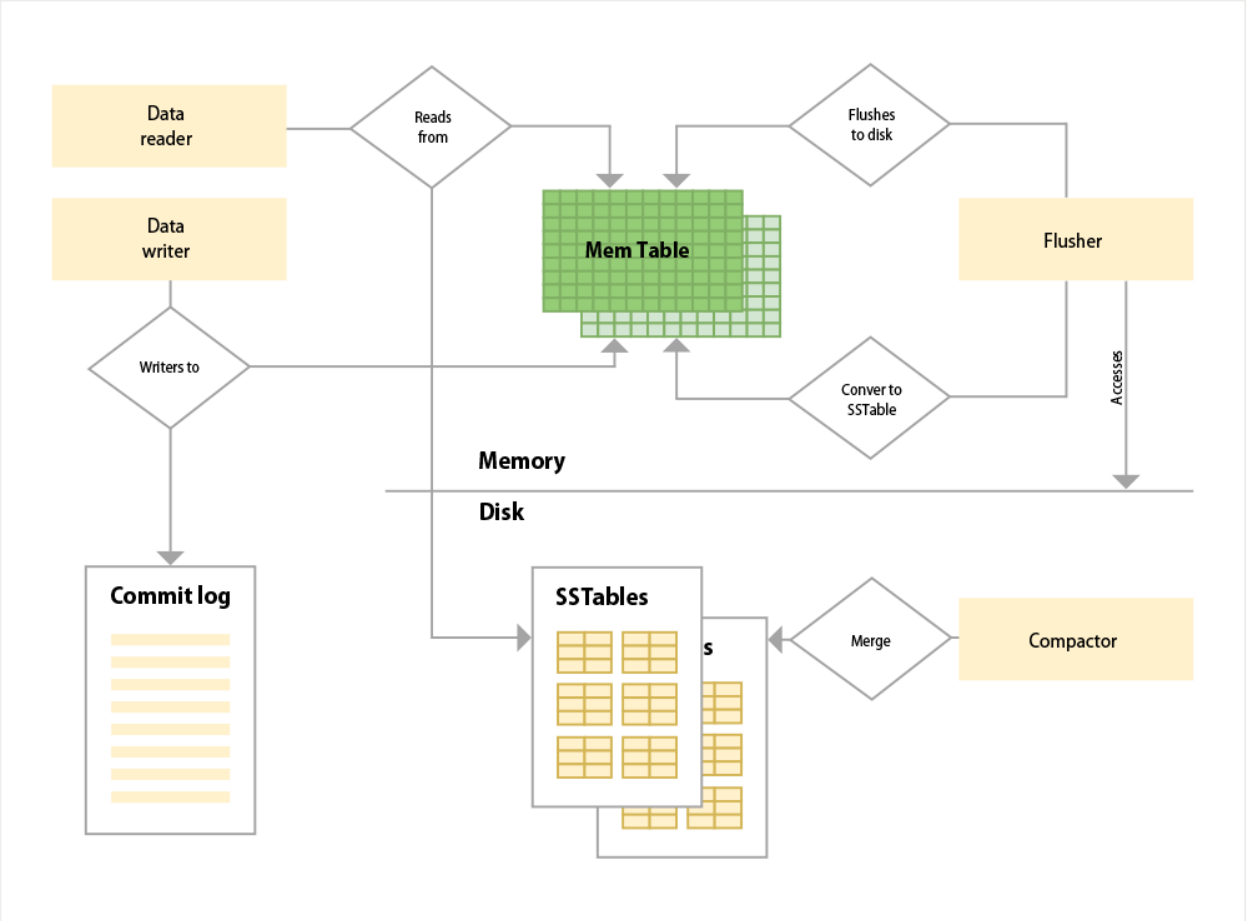

Figure 6.2. LSM-tree working components interactions

Memory tables. C* writes are first written to the commit log (if durability is enabled), and then to a per-table structure called a MemTable. A MemTable is basically a write-back cache of data rows that can be looked up by key. It contains recently read/modified data. Whenever a data from a table is read from a node, it will first check if latest data is present in MemTable or not. If latest data is not present, it will read data from SSTable and cache.

MemTable is created separately for each table so there is no blocking of read or write for individual tables. Multiple updates on single column will result in multiple entries in commit log, and single entry in MemTable. MemTable is a map that based on skip list data structure. Key of that map is a key of row. Value is a sorted map of row columns stored in a thread-safe and atomic B-tree. So, all operations on columns map are atomic and isolated (in the sense of ACID).

MemTables are kept in a bounded memory area and when occupied part of that area exceeds the threshold the biggest MemTable is written to disk as an SSTable. The process of turning a MemTable into a SSTable is called flushing. MemTables are sorted by key and then written out sequentially. Thus, writes are fast, costing only a commit log append and an amortized sequential write for the flush.

Sorted string table. SSTable is a data format, which allows to store a huge number of $< key, value >$ pairs in efficient manner. This format is optimized to provide high throughput for sequential write and read operations which is disks friendly. $keys$ and $values$ can be of any sizes, but all $< keys, value >$ pairs must be order by key. SSTable allows to create sorted index of entries just reading data sequentially from disk and calculating offsets between $keys$. Thus, any $keys$ can be find in $O(1)$ time.

C* SSTable integrally composes from a set of separate immutable files:
- *Data* file - stores rows ordered by key;
- *Index (primary index)* file - stores index using primary keys;
- *Index summary (partition summary)* file - stores only ranges of indexes from index file;
- *Bloom filter* file - stores information about rows presence in data file;
- *Compression offset maps* file - info about data compression;
- Other special files (checksum, statistics).

Figures 6.3 and 6.4 illustrate index and data files formats and connections between files.

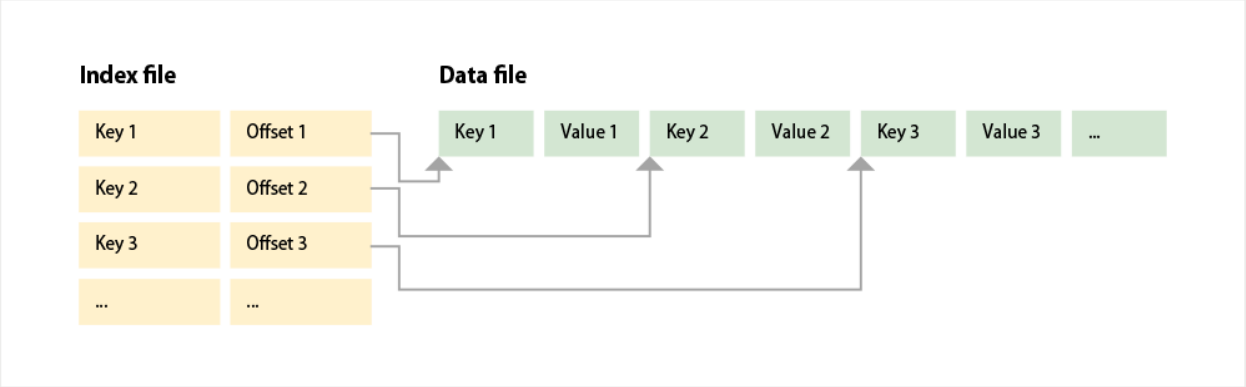

Figure 6.3. Index file linked to data file

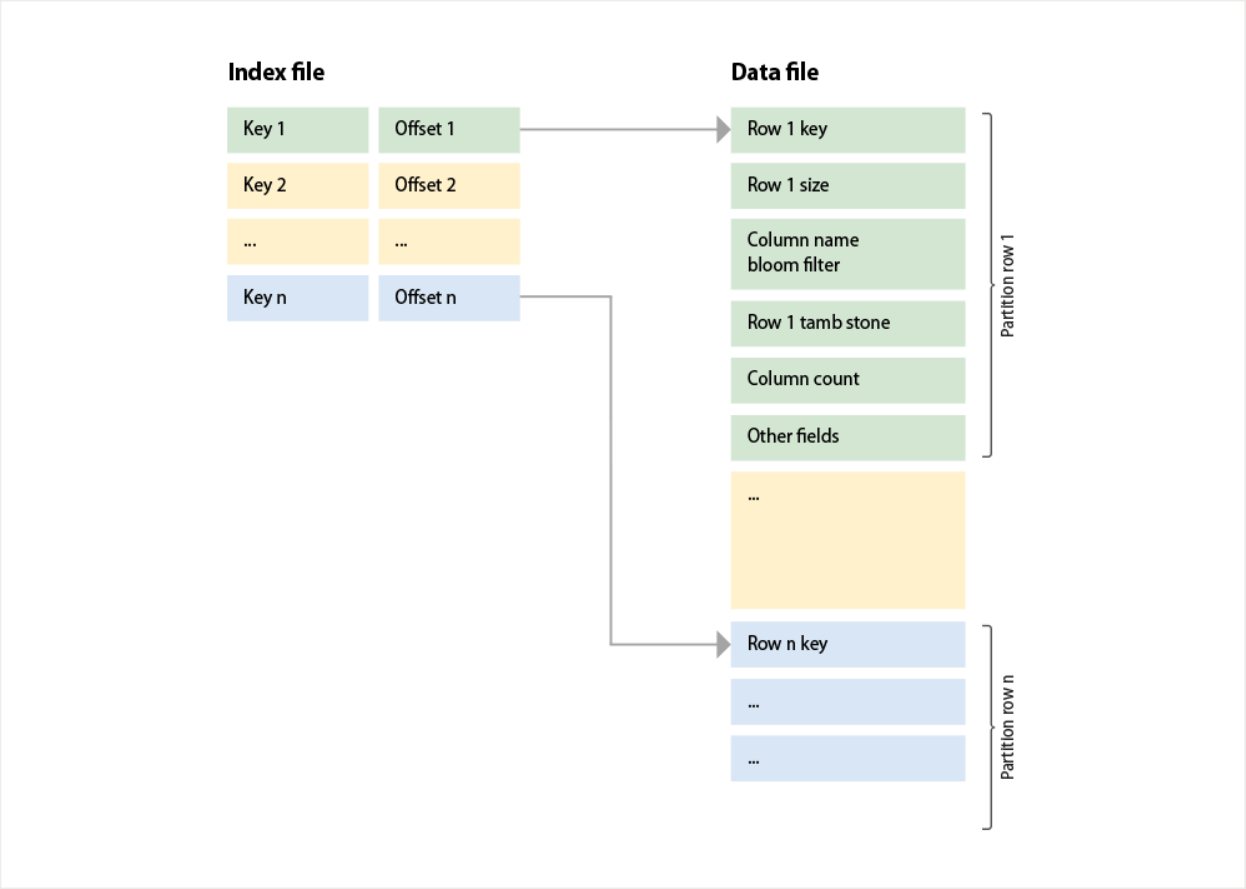

Figure 6.4. Detailed data file format

Row tombstone field is used to indicate that row is deleted (see Section 9).

Index summary is used to speed up the access to index on disk. Index summary stores only one index record for every 128 records (default value) of indexes in an index file. Each of these records of index summary will hold key value and offset position in index. Read requests comes to index summary, offset is checked for that key in index file on disk. Since all index records are not stored in index summary, it gets a rough estimate of offset it has to check in index file and reduces disk seeks.

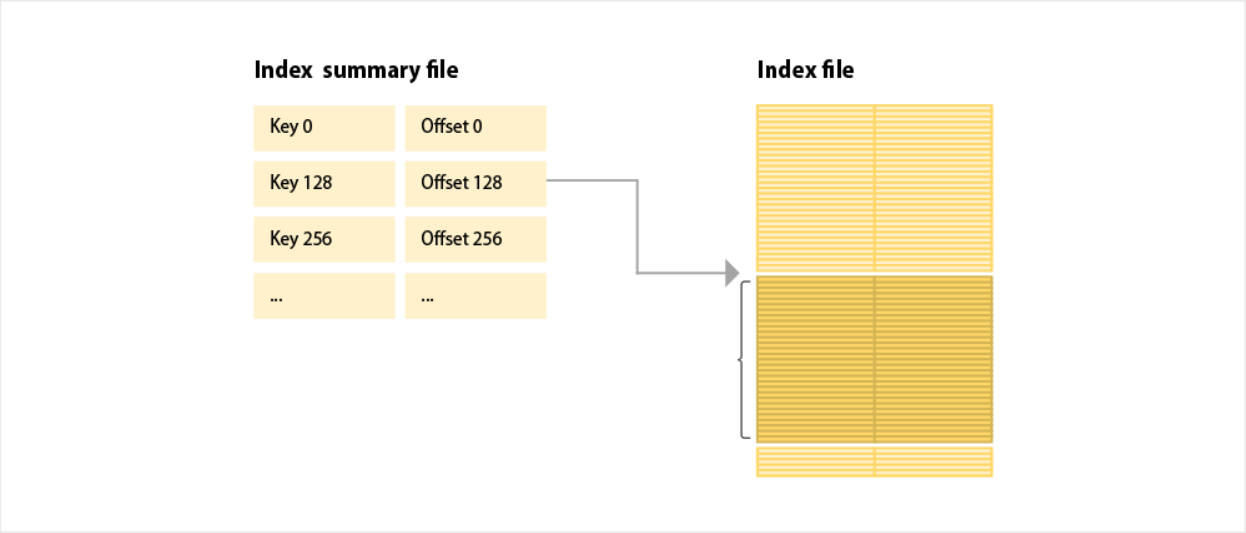

Figure 6.5. Connection between index and index summary

Compression offset maps holds the offset information for compressed blocks. By default all tables in C* are compressed and more the compression ratio larger the compression offset table. When Cassandra needs to look for data, it looks up the in-memory compression offset maps and unpacks the data chunk to get to the columns. Both writes and reads are affected because of the chunks that have to be compressed and uncompressed.

Obviously, indexes allow to reduce disk seeks to one and improve read operation performance. But generally, many SSTables is correspond to one table, and to find requested row system need to seek number of times equals to the number of SStables. To avoid such behaviour and dramatically improve read performance, each SSTable has Bloom filter (BF). Bloom filter file store BF for each data file. BF is space-efficient probabilistic data structure which typically is a bit-array. BF allows by row partition key find out is a given SSTable does not contains this row (true negative answer) or maybe it contains this row (false positive answer) and reduce the number of tables to explore. Figure 6.5 illustrates filtering process of checking if element (row) belongs to the set (data file).

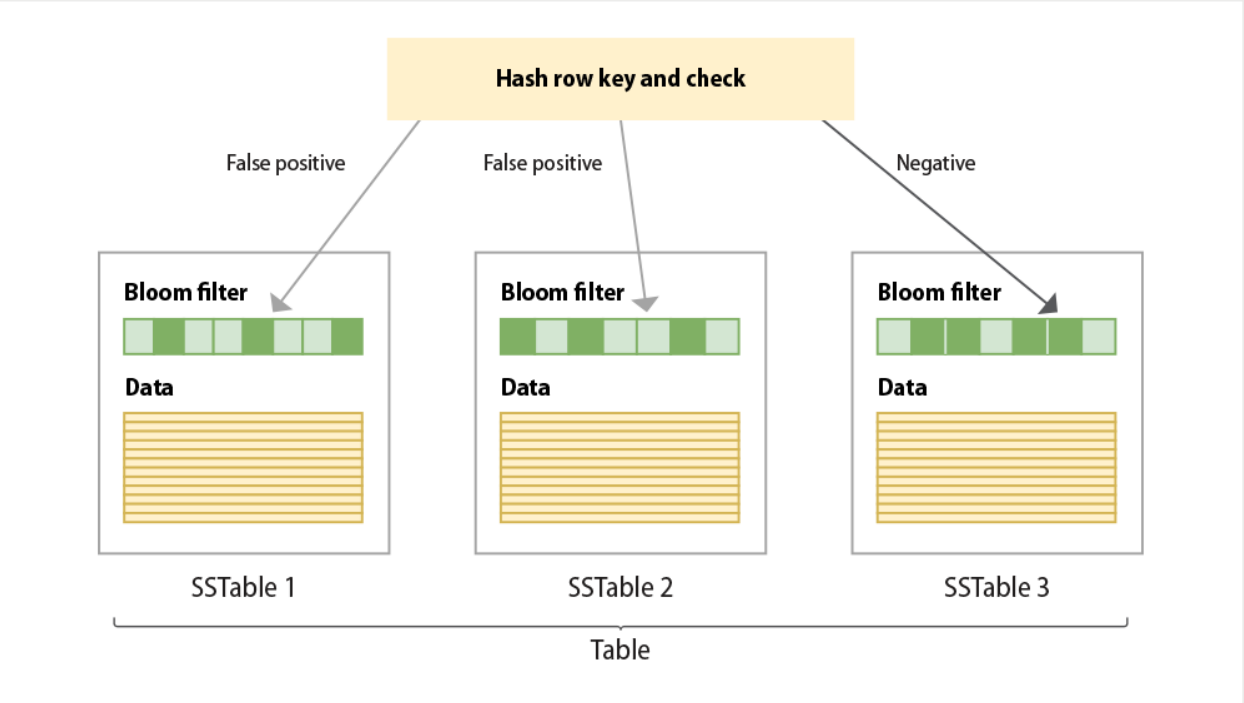

Figure 6.6. Bloom filtering

Compaction process. To bound the number of SSTable files that must be consulted on reads, and to reclaim space taken by unused data, C* performs compaction: merging multiple old SSTable files into a single new one by doing merge sort. So, in short, compaction is the process of consolidating SSTables, discarding tombstones, and regenerating the SSTable index. Since the input SSTables are all sorted by key, merging in a compaction can be done sequentially and efficiently, requiring no random disk i/o accesses. But compaction can be a fairly heavyweight operation. C* takes two steps to mitigate compaction impact on application requests performance:

- throttles compaction i/o to specified bound (default 16MB/s)
- request the operating system pull newly compacted partitions into its page cache when C* key cache indicates that the compacted partition was "hot" for recent reads

Once compaction is finished, the old SSTable files will be deleted as soon as any pending reads finish with them as well.

Compaction algorithms are pluggable and defined by compaction strategies. In general, strategies decide what SSTables to compact, how big they should be, how often to do that. Strategies get notified when adding new SSTable.

C* has two predefined strategies:

- Size-tiered compaction strategy combines SSTables based on their size;
- Leveled compaction strategy:
    - keeps levels of non-overlapping SSTables;
    - each level is 10x the size of the previous one;
    - all sstables in levels 1+ are about the same size (160MB).
- Date-tiered compaction strategy: for time series data and expired data.

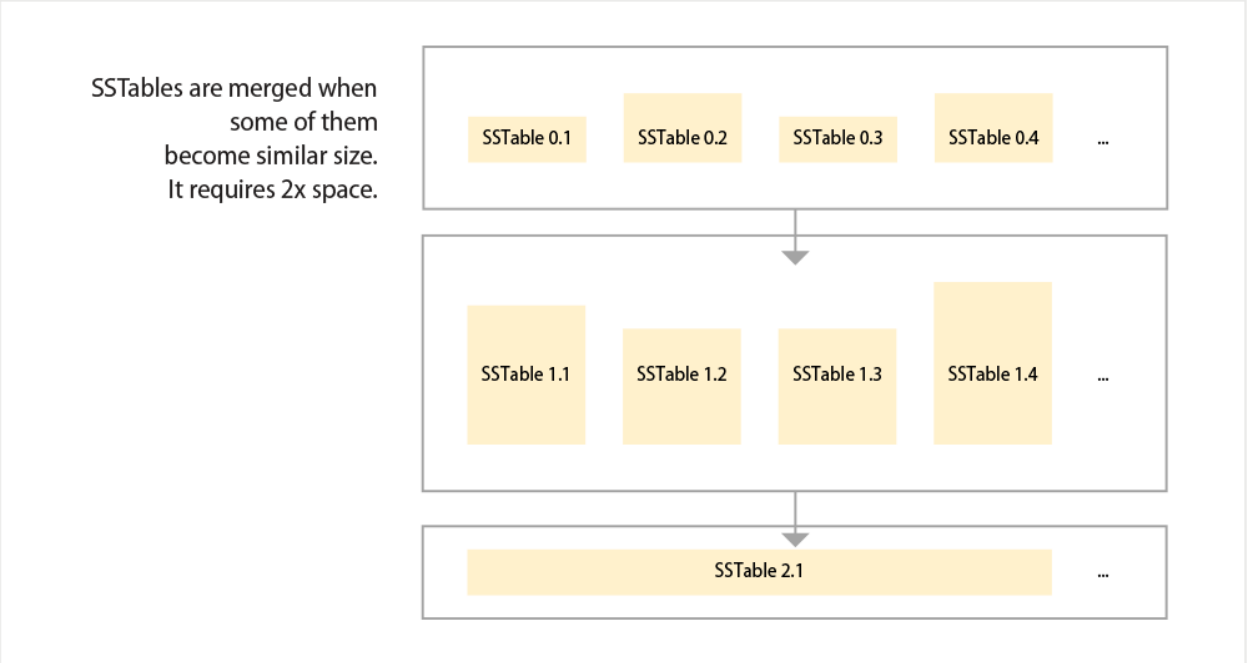

Figure 6.7. Size-tiered compaction strategy schema

Size-tiered is better for write-intensive workloads and Leveled better for read-intensive due to disk usage. Figures 6.7 and 6.8 illustrate how mentioned strategies work.

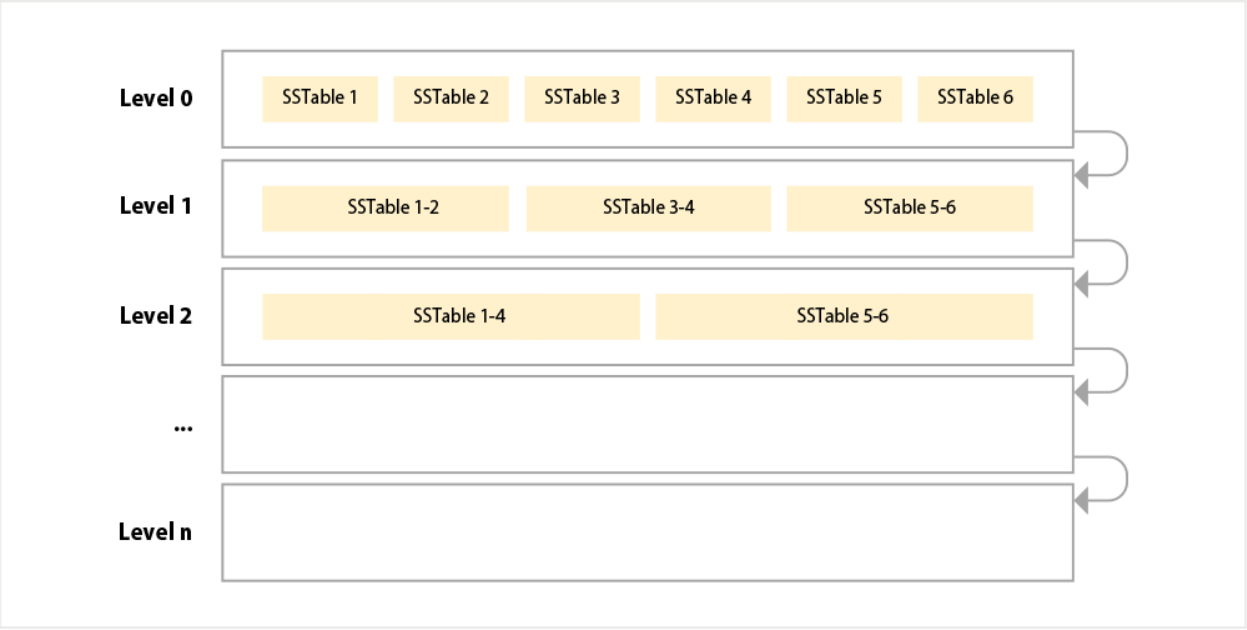

Figure 6.8. Leveled compaction strategy schema

There are two types of compaction:

- *minor*;
- *major*.

A minor compaction performed for one tables and is triggered automatically, when at least N SStables of similar sizes have been flushed to disk.

A major compaction performed for all node's tables and is triggered either manually, or automatically:

- When manually triggered - node sends "Get Merkle tree request" messages to all its neighbors: when a node receives a such request, it will perform a read-only compaction to immediately validate the table (for Anti-entropy, Section 5).
- Automatic compactions will also validate a table and broadcast "Merkle tree responses", but since request messages are not sent to neighboring nodes, repairs will only occur if two nodes happen to perform automatic compactions within certain time window of one another.

During compaction, there is a temporary spike in disk space usage and disk I/O because the old and new SSTables co-exist. A major compaction can cause considerable disk I/O.

Key cache & Row cache. The key cache is essentially a cache of the SSTable index part. It saves CPU time, disk seeks and memory over relying on the OS page cache for this purpose.

The row cache is more similar to a traditional cache like: when a row is accessed, the entire row is pulled into memory (merging from multiple SSTables, if necessary) and cached so that further reads against that row can be satisfied without hitting disk. C* allows to configure the number of rows to cache in a partition. To cache rows, if the row key is not already in the cache, C* reads the first portion of the partition, and puts the data into the cache. If the newly cached data does not include all rows, C* performs another read.

Key and row caches is configurable at table level. It is possible to vary key cache and row cache sizes or disable it at all.

Secondary index. It is possible to improve read operation performance for those table columns that are not in partition key. For that secondary indexes are created. Secondary index is a separate SSTable hidden for client applications, but ordered by column that is not the key column. Secondary indexes allow querying by value and can be built in the background automatically without blocking reads or writes.

7 Durability

Durability is the property that writes, once completed, will survive permanently, even if the node is killed or crashes or loses power.

Writes in C* can be durable or not and applied to the whole keyspace. It's possible to enable/disable durability for any user keyspace and system traces keyspace (where C* automatically saves trace sessions for reference and keep it for some period of time). But writes to other system tables are always durable and can't be disabled.

C* runs on top a modern operating systems (OS) and uses disk file system to store data. This requires making *fsync* system calls to tell the OS to flush its write-behind cache to disk and

guarantee durability. But calling fsync on each write is a very performance leak, because the disk needs to do random seeks to write the data to the write location on the physical platters. It immediately affects on write latencies since each seek costs 5-10 ms on rotational devices.

To make writes durable and avoid fsyncing each write C* uses write-ahead logging (WAL) technique. All writes to a node are recorded both in MemTables and in a memory mapped files called *commit log segments* (CLS) on disk before they are acknowledged as a success. It is done by *commit log* subsystem. If a crash or node failure occurs before the data in MemTables are flushed to disk, the commit log is replayed on restart to recover any lost writes. Figure 7.1 illustrates the path of durable write operation.

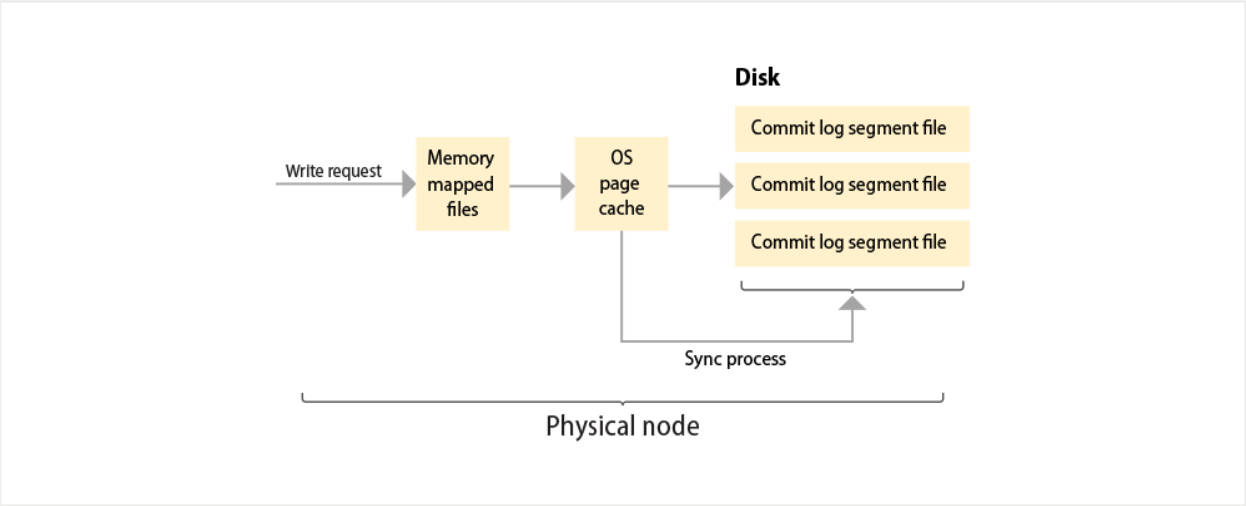

Figure 7.1. Durable write path

Commit log implements append-only algorithm, so all writes to CLS are performed sequentially to avoid disk seeking and minimize write latencies and go through operating system (OS) page cache. Since fsync is a costly operation C* uses two strategies: call fsync for a group of writes (batch mode), or fsync CLS periodically (periodic mode).

To use periodic mode sync period in milliseconds must be specified. In periodic mode data potentially can be lost all replicas with this data crash within that period.

In batch mode C* guarantee that it syncs before acknowledging writes. To use batch mode batch window period must be specified. C* groups multiple writes over a time period window and executes them in batches.

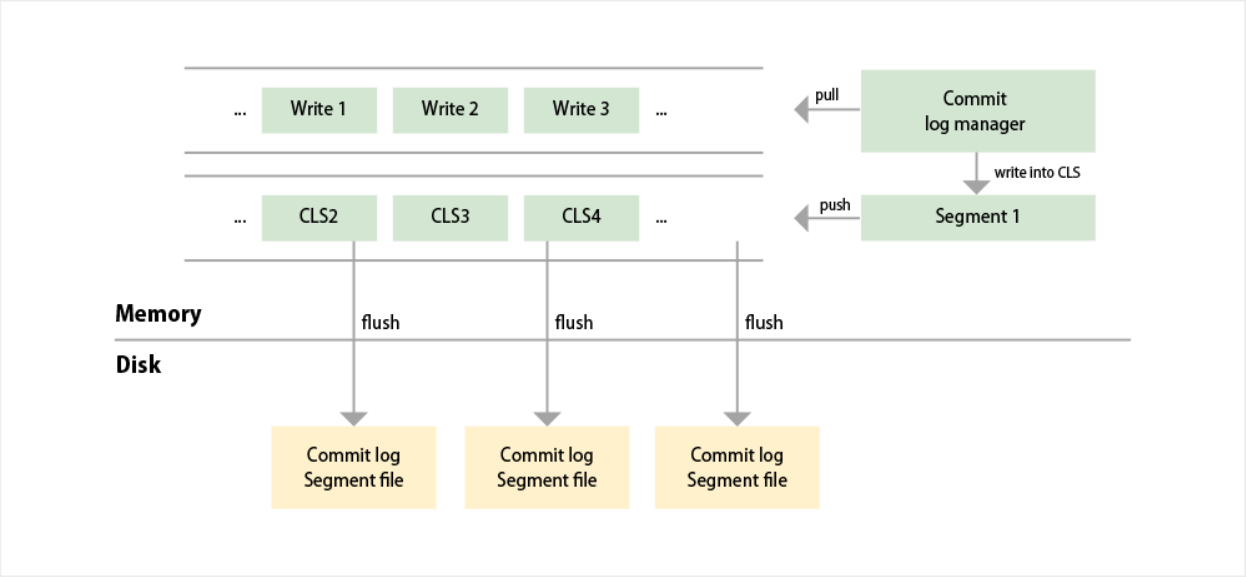

Figure 7.2. Commit log process

CLS always has a limited size (default is 32MB) and checksum to avoid errors, however CLS size is tunable. Also, it is possible to archive CLS (to keep history) where finer granularity of CLS is reasonable. Figure 7.2 shows the underlying structure of CLS file.

Commit log keeps CLS at a specified disk directory. C* supports handling policies to react on commit log disk failures. There are next policies:

- *die*: shut down gossip and kill the JVM, so the node can be replaced;
- *stop*: shut down gossip, leaving the node effectively dead, but can still be inspected via JMX;
- *stop_commit*: shutdown the commit log, letting writes collect but continuing to service reads;
- *ignore*: ignore fatal errors and let the requests fail

8 Process model

Process model is a technique of how the system handle client requests and perform other tasks. C* uses staged-based event driven architecture. Such approach decomposes all work into into a set of stages connected by queues where every queue if served by separate thread-pool. It avoids the high overhead associated with thread-based concurrency models (one thread per one client request) and decouples event and thread scheduling from application logic. By performing admission control on each event queue, C* can manage load for each stage, preventing resources from being overcommitted.

Each stage is responsible for a set of events that pushed into the stage queue and handled in separate thread by needed handler. Very simple staged handling process is displayed on Figure 8.1.

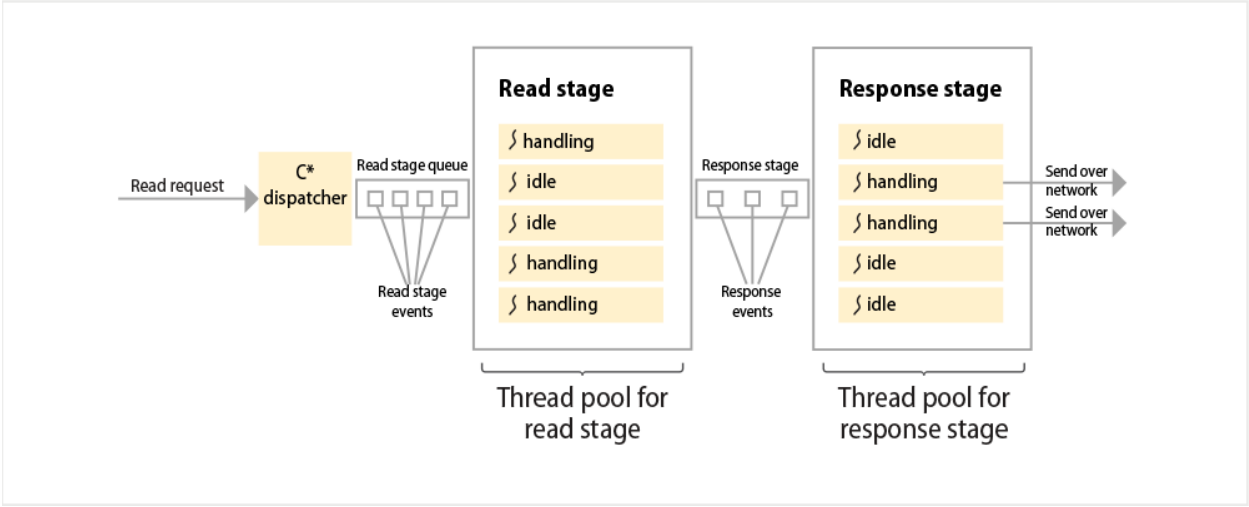

Figure 8.1. Staged events processing

9 Write path

Section describe insert, update and delete write operations. Insert operation adds new row into database. Update operation updates an existing rows and supports counters (fields that can be only incremented). Insert and update operation are identical internally. Delete operation removes rows from database. Generally, they all pass the same path described below.

Write operations use stage-based approach described in Section 8. Write path begins from the client request sent to coordinator node.

Write request

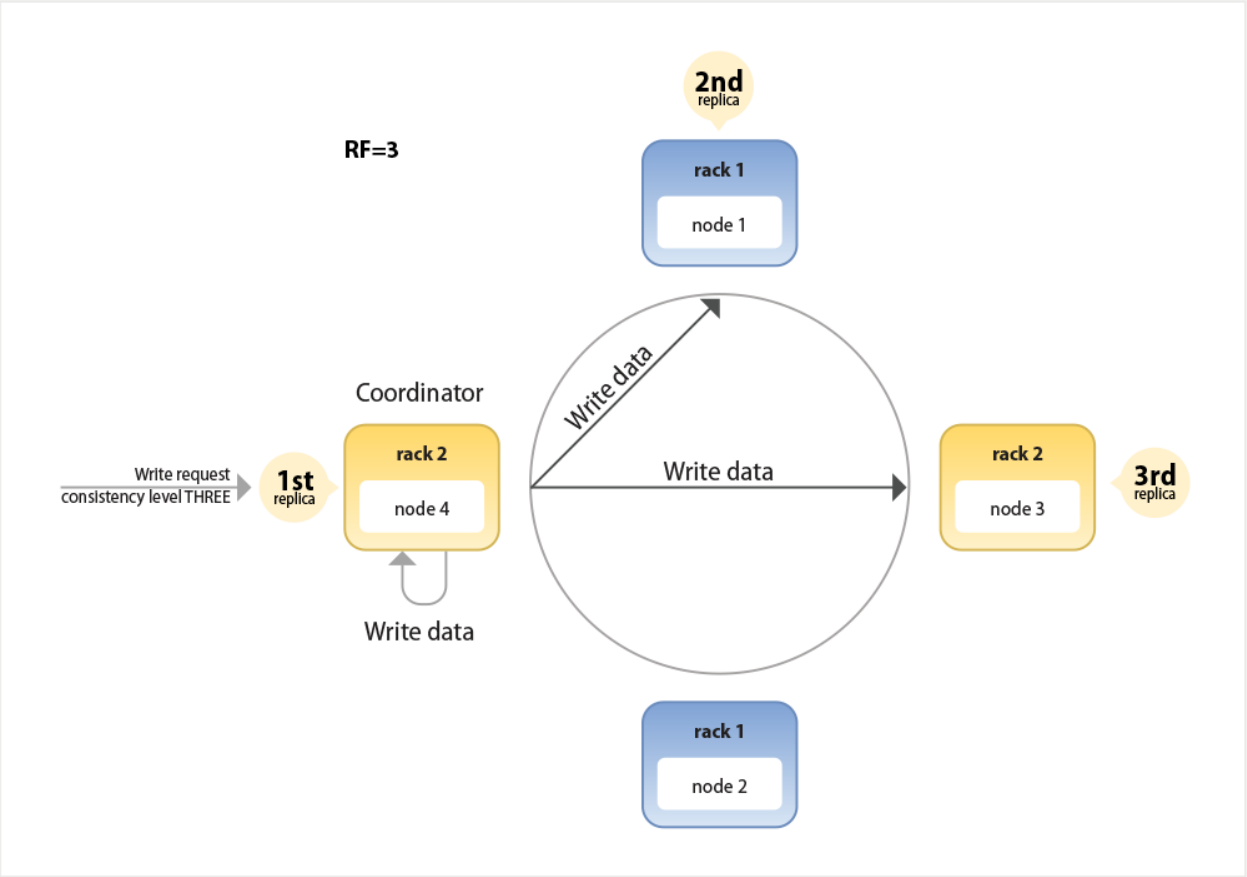

Figure 9.1. Write request distribution

Coordinator gets the nodes responsible for replicas of the keys and then sends row mutation messages to them. Coordinator waits for responses from the number of replicas specified by consistency level. If there are not enough nodes alive to satisfy the consistency level, C* fail the request with *UnavailableException*. If there are no failures but writes time out anyway because of a failure after the request is sent or because of an overload scenario, C* will write a "hint" locally to replay the write when the replica(s) timing out recover. Hinted handoff process described in Section 5. Cross-datacenter writes are not sent directly to each replica; instead, they are sent to a single replica with a parameter in message saying that replica to forward to the other replicas in that datacenter; those replicas will respond directly to the original coordinator.

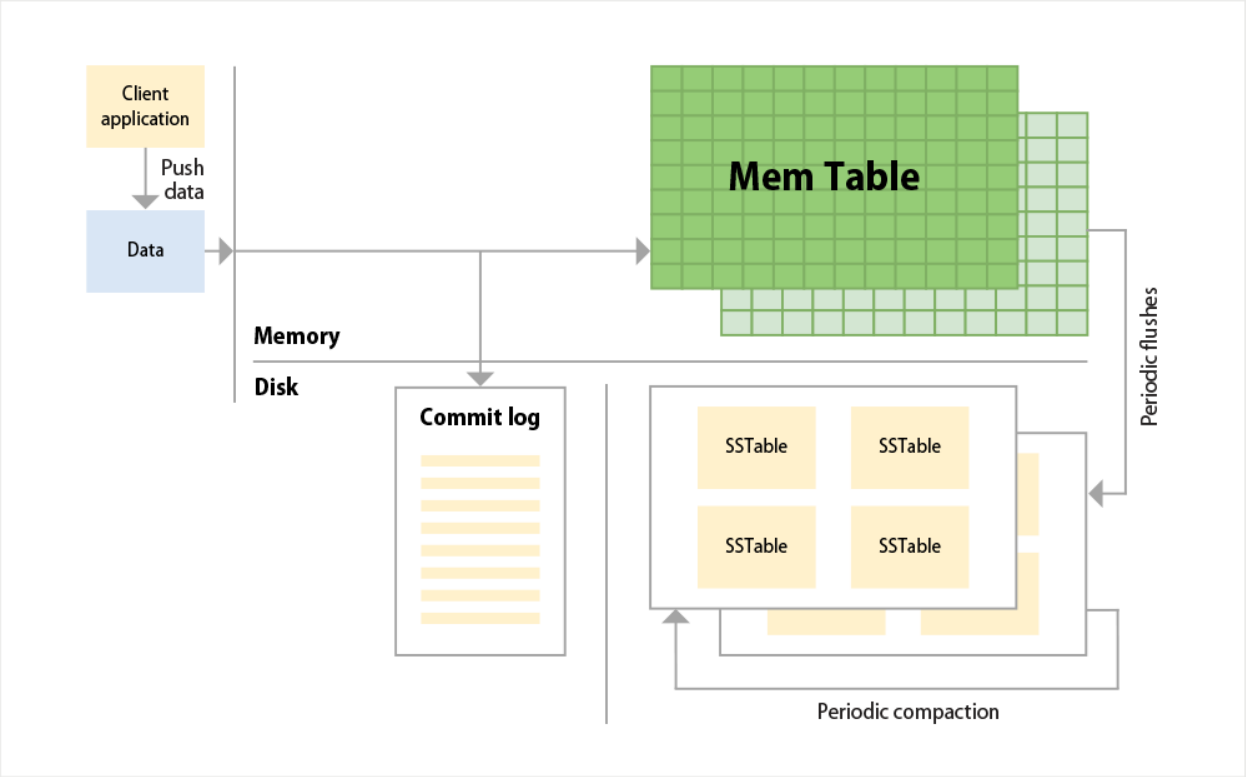

Figure 9.2. Replica node high level write process

On the replica node, write is performed. This has several steps. First, an entry is appended to the commit log (potentially blocking if the commit log is in batch sync mode or if the queue is full for periodic sync mode). Then write go to the MemTable. Also, C* updates secondary indexes and invalidate row cache for each row in the write message.

When a MemTable is full, it is asynchronously sorted and written out as an SSTable.

"Fullness" is monitored by the system. The goal is to flush quickly enough that C* does not meet out of memory exception as new writes arrive while C* still must 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. Compaction manager manages the queued tasks and some aspects of compaction. Making this concurrency-safe without blocking writes or reads while C* removes the old SSTables from the list and add the new one is tricky. C* performs manual reference counting on SSTables during reads so that C* knows when they are safe to remove.

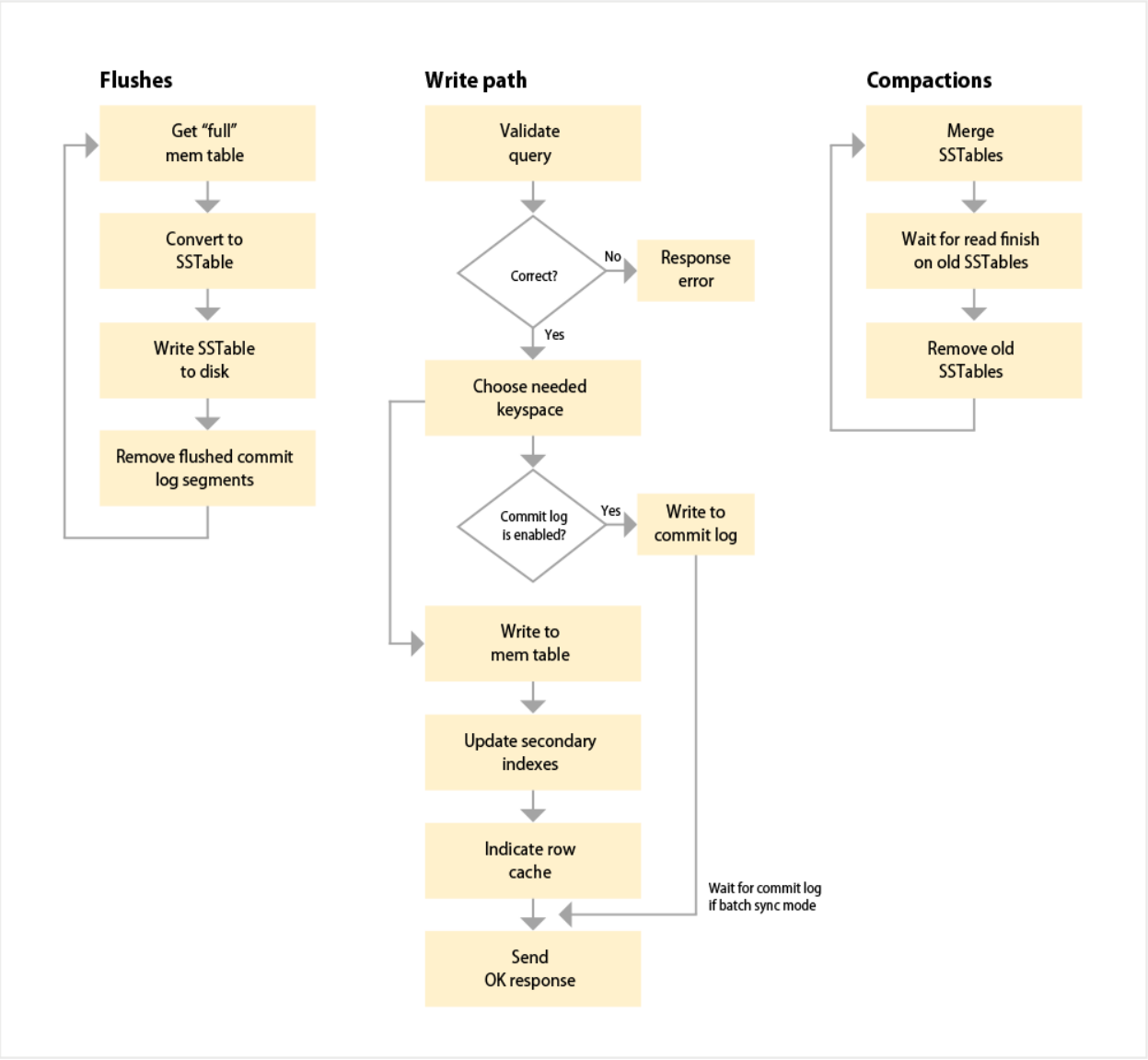

Figure 9.3. Replica node write path

Delete operation is a little bit tricky. A delete operation can't just wipe out all traces of the data being removed immediately: if C* did, and a replica did not receive the delete operation, when it becomes available again it will treat the replicas that did receive the delete as having missed a write update and repair them. So, instead of wiping out data on delete, C* replaces it with a special value called a tombstone. The tombstone can then be propagated to replicas that missed the initial delete request.

To remove rows completely from disk C* defined a constant, *GCGraceSeconds* (default 10 days), and had each node track tombstone age locally. Once it has aged past the constant, it can be GC'd during compaction.

10 Read path

Read operation begins the same as the write. Coordinator determines the replica nodes to read from using replication strategy and row (partition) key for the keyspace.

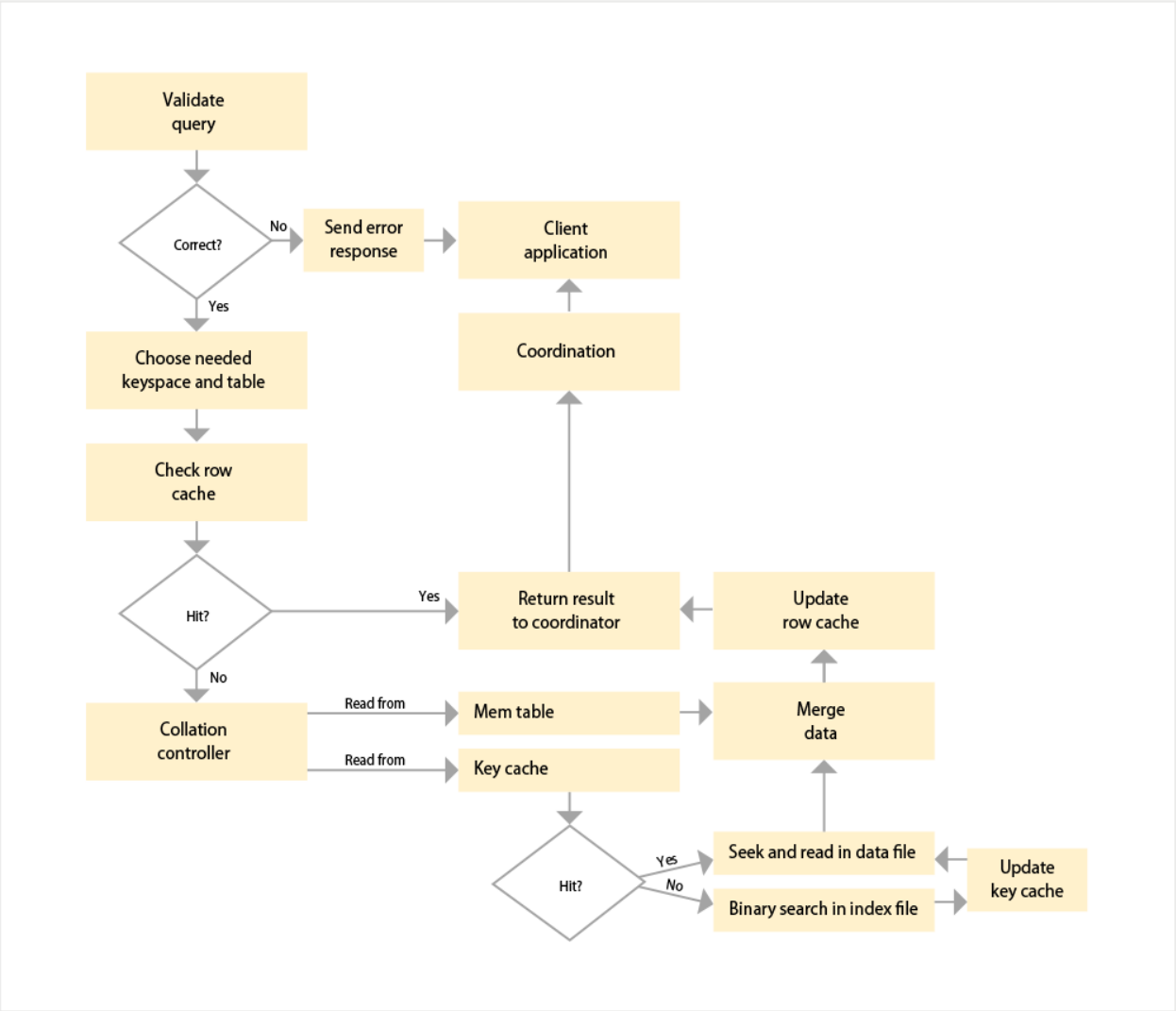

Figure 10.1. Replica node read path

Replica nodes are filtered to contain only those that are currently up/alive. If there are not enough live replica nodes to meet the consistency level, an UnavailableException response is returned. The closest node (in network, local data center) will be sent a command to perform an actual data read (i.e., return data to the coordinating 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. If needed read repair process is performed.

On the replica node C* gets the data for single row reads, sequential scans, and index scans, respectively, and sends it back as a read response. If the row cache is enabled, it is first checked for the requested row. The row cache will contain the full partition, which can be

trimmed to match the query. If there is a cache hit, the node can be responded to immediately. Otherwise C* must check SSTables to find requested rows. Bloom filter is used to skip those SSTables that do not contain requested rows.

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 index summary is used. 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 compression is enabled, the block that the requested data lives in must be uncompressed. Data from MemTables and SSTables is then merged by collation controller. Then, if row caching is enabled, the row cache is updated.

Responses from replica nodes are backed to the coordinator. If a replica node fails to respond before a configurable timeout, a *ReadTimeoutException* is raised. Once retries are complete and digest mismatches resolved, the coordinator responds with the final result to the client application.

If consistency level is less than the total number of replicas, than rapid reads can be performed. For example, if one of the requested replica nodes is failed during request processing, coordinator should re-request another available replica node. To avoid time overhead of coordinator send request to extra replica nodes not after failure detection but at the same time it sends requests to replica nodes according consistency level. This process is performed with probability and based on nodes latencies statistics. For example, rapid read can be performed if replica node does not respond for a time that is under 90 percentile of all performed requests.

11 Memory usage

C* is written in Java programming language and should be run under Java HotSpot Virtual Machine. Thus, C* uses Java heap space and off-heap space to allocate new objects. Java heap space is controlled by garbage collector (GC) which is a form of automatic memory management. Due to GC process Java virtual machine (JVM) stops execution of hosted application (C*) that in turn affects on response latencies.

Therefore, JVM has to be tuned for any special use case to achieve low latencies and high throughput while executing C*. Also, C* in pair of heap area uses off-heap memory, which is raw memory available for process and need to be manage by programmers.

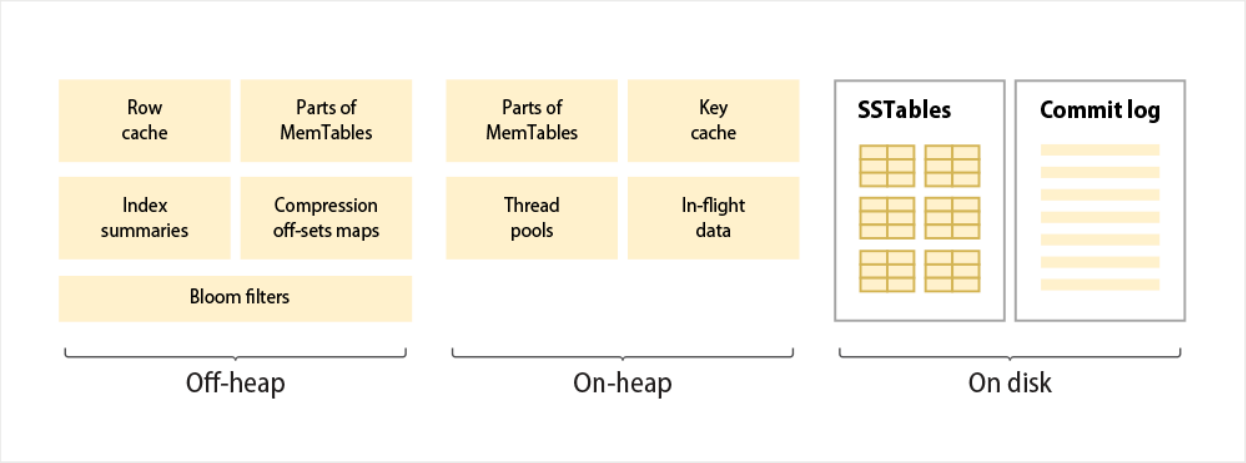

Figure 11.1. Objects allocation areas

The vast majority of objects in C* are ephemeral short lived objects. This means that the majority of created objects are created and should be garbage collected when request is processed. The only thing that should live for a long time the key cache. So, it is possible to reduce GC pauses by tuning heap size and other GC options according C* use-cases, handled workloads.

C* can store cached rows, index summary, compression offsets map, bloom filters and partly MemTables in native memory, outside the Java heap. This results in both a smaller memory footprint and reduced JVM heap requirements, which helps keep the heap size in the sweet spot for JVM GC performance.

MemTables are not moved entirely into off-heap space, but two options are provided to reduce their footprint:

- *off-heap buffers*: moves the column name and value;
- *off-heap objects*: moves the entire column off heap, leaving only reference containing a pointer to the native (off-heap) data.

MemTables are stores in a bounded area of memory specified by administrator (separately for on-heap and off-heap spaces). If is used part of this area exceeds threshold MemTables are flushed to disk. There are next rules to setup memory size to hold MemTables:

- *Larger MemTables take memory away from caches*: Since MemTables are storing actual column values, they consume at least as much memory as the size of data inserted. However, there is also overhead associated with the structures used to index this data;
- *Larger MemTables don't improve write performance:* Increasing the MemTable capacity will cause less-frequent flushes but doesn't improve write performance directly: writes go directly to MemTables regardless;
- *Larger MemTables do absorb more overwrites:* If write load sees some rows written more often than others a larger memtable will absorb those overwrites, creating more efficient sstables and thus better read performance;

- *Larger MemTables do lead to more effective compaction:* Since compaction is tiered, large sstables are preferable: turning over tons of tiny memtables is bad (this impacts read performance).

The bloom filter grows to approximately 1-2 GB per billion partitions. In the extreme case, it is possible have one partition per row, so you can easily have billions of these entries on a single machine. The Bloom filter is tunable to trade memory for performance.

The compression offset map grows to 1-3 GB per terabyte compressed. The more compress data, the greater number of compressed blocks and the larger the compression offset table. Having compression enabled makes the page cache more effective, and typically, almost always pays off.

C* uses memory mapped files to do zero-copy reads. That is, C* uses the OS virtual memory system to map the SSTable data files into the C* process' address space. This will "use" virtual memory; i.e. address space. Also, C* relies on OS page cache to improve read performance and avoid disk accesses.

C* maintains distinct thread pools for different stages of execution. Each of the thread pools provide statistics on the number of tasks that are active, pending, and completed. So, memory consumed by threads created to handle different system events.

Also C* exposes a number of statistics and management operations via Java Management Extensions (JMX). Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring Java applications and services. Any statistic or operation that a Java application has exposed as an MBean can then be monitored or manipulated using JMX.

Cassandra 2.1 vs. Couchbase 3.0

(summaries)

1 Topology

C* has no SPF as well as Couchbase. Both databases easy to setup and understand. C* provides flexible logical topology configuration by organizing nodes into racks and data centers while Couchbase is only rack-aware and to support data centers nodes must be physically in separate Couchbase systems. C* allows to tune each node precisely and nodes hardware can be heterogeneous. Couchbase support only homogeneous hardware. All nodes in C* are of the same roles, while Couchbase has master nodes which responsible for writes and reads, but masters can switch their roles with replica nodes.

2 Data model

C* in compared to Couchbase has reacher data model with special query language. C* data model is not just a single key-value store. It allows to organize and operate complex data conveniently, operate on a single columns or a set of columns of the specified types not just the whole document like in Couchbase.

3 Partitioning

C* uses hash partitioning based on consistent hashing algorithm with possibility to choose partitioning strategies on the user level. This with the virtual nodes concept provides possibility to easy rebalancing the system (limits data size to stream from node to node). Couchbase is also hash based partitioning system but the whole system must be rebalanced even if the only node is removed or added.

4 Replication

C* and Couchbase support single datacenter and cross-datacenter replication, distributes across racks, takes into account the physical topology. C* supports customizable replication strategies. In general, both databases do replication asynchronously but in C* clients can do it synchronously by choosing highest write consistency level. Couchbase does it always asynchronously and client must ask to Couchbase periodically if data is replicated.

5 Availability

C* and Couchbase clients choose between availability and consistency by tuning request consistency level. C* uses gossip protocol to keep system information up to date. Couchbase uses system-wide service to do the same job. All C* replica nodes are equivalent and client can read data from any of that nodes. Couchbase client read only from master nodes and because of this fact Couchbase has guaranteed downtime if master node is failed (see automatic failover). C* uses such techniques as hinted handoff to provide data availability (to increase probability of availability) while if Couchbase node responsible for a certain range of data does not respond this range will be not available for a time.

6 Storage engine

C* storage engine allows to write fast but reads are more costly due to multiple SSTables. Compactions use disk throughput and require free space capacity up to 2x of the occupied part.

Couchbase has async storage engine which provides superfast reads and writes while memory is available. Occupied memory is freed by a special process. Since data is flushed to disk async client does not know for sure when data will be on disk.

7 Durability

C* has flexible durability options (batches and periodic) and uses write-ahead-logging technique in local machine. Commit log requires separate disk for its activities. Couchbase relies on data replication rather than commit log.

8 Write path & Read path

Read and write can be memory-centric as well as disk-centric for both C* and Couchbase. Performance of these operations depends on consistency level and data location. In C* columns updated by key, clients do not need to read the whole row and update columns on server side in Couchbase client has to read the whole document to update only one field and then rewrite old document with newly updated.

C* clients can connect to any system node using the same network port while in Couchbase client must use different ports for situation when they are aware of system topology and when they do not.

9 Memory usage

C* requires memory for caches and MemTables. Memory is managed by JVM memory management system but it's also possible to use off-heap memory so C* can use all available memory. C* load should be tuned to meet disks speed and available memory for MemTables.

Couchbase provides two policies of memory usage. The first one is when all metadata is memory. If metadata size is greater than the available memory, then Couchbase system will not work. Such approach provides efficient disk I/O. The second policy is when all are in memory if there is not enough memory data and metadata if flushed to disk. When client creates bucket then it must be specified what amount of memory will be used for this bucket and this amount cannot be changed. If there is not enough memory than it is impossible to create new bucket.