# CockroachDB architecture

The architecture of a software system defines the high-level design decisions that enable the goals of that system. As you may recall from Chapter 1, the goals of CockroachDB are to provide a scalable, highly available, highly performant, strongly consistent, geo-distrusted, SQL-powered relational database system capable of running across a wide variety of hardware platforms. The architecture of CockroachDB is aligned to those objectives.

There are multiple ways of looking at the CockroachDB architecture. From a physical cluster level, a CockroachDB deployment consists of one or more shared-nothing, masterless nodes that collaborate to present a single logical view of the distributed database system. Within each node, we can observe the CockroachDB architecture as a series of layers that provide essential database services, including SQL processing, transaction processing, replication, distribution and storage.

In this chapter, we'll endeavor to give you a comprehensive overview of the CockroachDB architecture. The aim of the chapter is to provide you with the fundamental concepts that will help you make sensible decisions regarding schema design, performance optimization, cluster deployment and other topics.

The CockroachDB architecture is sophisticated: it incorporates decades of database engineering best practice designs together with several unique innovations. However, CockroachDB doesn't require that you understand its internals in order to get things done. If you are in a hurry to get started with CockroachDB, you can skip forward to the next chapter and return to this chapter later as necessary. We will, however, assume you are broadly familiar with the key concepts in this chapter when we consider advanced topics later in the book.

## The CockroachDB Cluster Architecture

From a distance, a CockroachDB deployment consists of one or more database server processes. Each server has its own dedicated storage – the familiar "**shared nothing**" database cluster pattern. The nodes in a CockroachDB cluster are symmetrical – there are no "special" or "master nodes". This storage is often directly attached to the machine on which the CockroachDB server runs, though it's also possible for that data to be physically located on a shared storage subsystem.

Data is distributed across the cluster based on **key ranges**. Each range is replicated to at least three members of the cluster.

**Database clients** – applications, administrative consoles, the CockroachDB shell and so on – connect to a CockroachDB server within the cluster.

The communications between a database server and database client occur over the PostgreSQL **wire protocol** format. This protocol describes how SQL requests and responses are transmitted between a PostgreSQL client and a PostgreSQL server. Because CockroachDB uses the PostgreSQL wire protocol, any PostgreSQL driver can be used to communicate with a CockroachDB server.

In a more complex deployment, one or more **load balancer** processes will be responsible for ensuring that these connections are evenly and sensibly distributed across nodes. The load balancer will connect the client with one of the nodes within the cluster, which will become the **gateway server** for the connection.

The client request might involve reading and writing data to a single node or to multiple nodes within the cluster. For any given range of key values, a **Leaseholder node** will be responsible for controlling reads and writes to that range. The Leaseholder is also usually the **Raft leader**, which has the responsibility to make sure that replicas of the data are maintained correctly.

Figure 1 illustrates some of these concepts. A Database client connects to a Load Balancer (1) that serves as a proxy for the CockroachDB cluster. The Load Balancer directs requests to an available CockroachDB node (2). This node becomes the Gateway node for this connection. The request requires data in Range 4, so the Gateway node communicates with the Leaseholder node for this range (3), which returns data to the gateway, which in turn returns the required data to the database client (4).



Figure 1 CockroachDB Cluster architecture

This architecture distributes load evenly across the nodes of the cluster. Gateway duties are distributed evenly across the clusters nodes by the load balancer, leaseholder duties similarly distributed by ranges across all the nodes.

In the case of a query that requires data from multiple ranges or where data must be changed (and therefore replicated), the workflow involves more steps. We will work through a more complex example towards the end of the chapter.

### Ranges and Replicas

We'll examine the nuances of CockroachDB distribution and replication later in this chapter. But for now, there are a few concepts we need to understand.

Under the hood, data in a CockroachDB table is organized in a **Key-Value** (KV) storage system. The Key for the KV store is the table's Primary Key. The Value in the KV store is a binary representation of the values for all the columns in that row.

Indexes are also stored in the KV system. In the case of a non-unique index, the Key is the index key concatenated to the table's primary Key. In the case of a unique index, the Key is the index key, with the primary key appearing as the corresponding Value for that key.

**Ranges** store contiguous spans of key values. Figure 2 illustrates how a "dogs" table might be segmented into ranges.

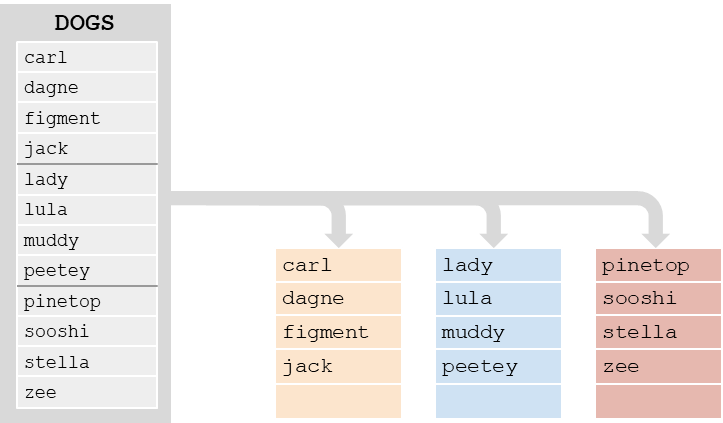


Figure 2 Ranges

As mentioned earlier, **Leases** are granted to a node giving it responsibility for managing reads and writes to a range. The node holding the lease is known as the **Leasehoder**. The same node generally also the**Raft leader** which is responsible for ensuring that replicas of the node are correctly maintained across multiple nodes.

## The CockroachDB software stack

Each CockroachDB node runs a copy of the CockroachDB software, which is a single multi-threaded process. From the Operating System perspective, the CockroachDB process might seem like a black-box, but internally it is organized into multiple logical layers, as shown in Figure 3.



Figure 3 CockroachDB software layers

We'll discuss each of these layers in turn as we proceed through the chapter.

## The CockroachDB SQL layer

The CockroachDB SQL layer is the part of the CockroachDB software stack that is responsible for handling SQL requests. Since CockroachDB is a SQL database, you would be forgiven for thinking that the SQL layer does pretty much everything. However, the core responsibility of the SQL layer is actually to turn SQL requests into requests that can be serviced by the Key-Value subsystem. Other layers handle transactions, distribution and replication of ranges and physical storage to disk.

The SQL layer receives requests from **database clients** over the **PostgreSQL wire protocol**.

A database client is any program that is using a database driver to communicate with the server and includes the CockroachDB command-line SQL processor, GUI tools such as DBEaver or Tableau or applications written in Java, Go, NodeJS, Python or any other language that has a compatible driver.

The PostgreSQL wire protocol describes the format of network packets that are used to send requests and receive results from a database client and server. The wire protocol layers on top of a transport medium such as TCP/IP or Unix-style sockets. The use of the PostgreSQL wire protocol allows CockroachDB to take advantage of the large ecosystem of compatible language drivers and tools that support the PostgreSQL database.

### SQL Optimization

The SQL layer parses the SQL request, checking it for syntactical accuracy and ensuring that the connection has privileges to perform the requested task.

CockroachDB then creates an execution plan for the SQL statement and proceeds to **optimize** that plan.

SQL is a declarative language: You define the data you want, not how to get it. Although the non-procedural nature of SQL results in improvements in programmer productivity, the database server must support a set of sophisticated algorithms to determine the optimal method of executing the SQL. These algorithms are collectively referred to as **the optimizer**.

For almost all SQL statements, there will be more than one way for CockroachDB to retrieve the rows required. For instance, given a SQL with JOIN and WHERE clauses, there may be multiple join orders and multiple access paths (table scans, index lookups, etc.) available to retrieve data. It's the goal of the optimizer to determine the best access path. CockroachDB’s SQL optimizer has some unique features relating to its distributed architecture, but broadly speaking, the Cost-based optimizer is similar to that found in other SQL databases such as Oracle or PostgreSQL.

The optimizer uses both heuristics – rules – and cost-based algorithms to perform its work.

The first stage of the SQL optimization process is to transform the SQL into a normalized form suitable for further optimization. This transformation removes any redundancies in the SQL statement and performs rule-based transformations to improve performance. The transformation takes into account the distribution of data for the table, adding predicates to direct parts of the queries to specific ranges or adding predicates that allow the use of indexed retrieval paths.

The optimization of the SQL statement proceeds in two stages – expansion and ranking. The SQL statement is transformed into an initial plan. Then the optimizer expands that plan into a set of equivalent candidate plans which involve alternative execution paths such as join orders or indexes.

The optimizer then ranks the plans by calculating the relative cost of each operation leveraging statistics that supply the size and distribution of data within each table. The plan with the lowest cost is then selected.

CockroachDB also supports a **vectorized execution** engine that can speed up the processing of batches of data. This engine translates data from a row-oriented format (where sets of data contain data from the same row) to a column-oriented format (where every set of data contains information from the same column).

We'll return to the optimizer in chapter8 when we look in detail at SQL tuning.

## From SQL to Key-Values

As we mentioned earlier, CockroachDB data ends up stored in a Key-Value storage system that is distributed across multiple nodes in ranges. We’ll look at the details of this storage system towards the end of the chapter, but since the outputs of the SQL layer are in fact Key-Value (KV) operations, the mapping of data from tables and indexes to Key Value representation is part of the SQL layer. The output of the SQL layer are Key-Value operations.

This translation means that only the SQL layer needs to be concerned with SQL syntax – all the subsequent layers are blissfully unaware of the SQL language.

Most of the time, this mapping is relatively unimportant. However, do remember that single SQL statements will often translate into multiple Key-Value operations and that each of these will have a distinct transactional scope.

### Tables as represented in the KV store

Each entry in the KV store has a Key based on the following structure:

+/<tableID>/<indexID>/<IndexKeyValues>/<ColumnFamily>+

We’ll discuss ColumnFamilies in the next section by default, all columns are included in the a single default ColumnFamily.

For a base table, the default indexID is “primary”. Figure 5 shows a simplified version of this mapping, omitting the ColumnFamily identifier.

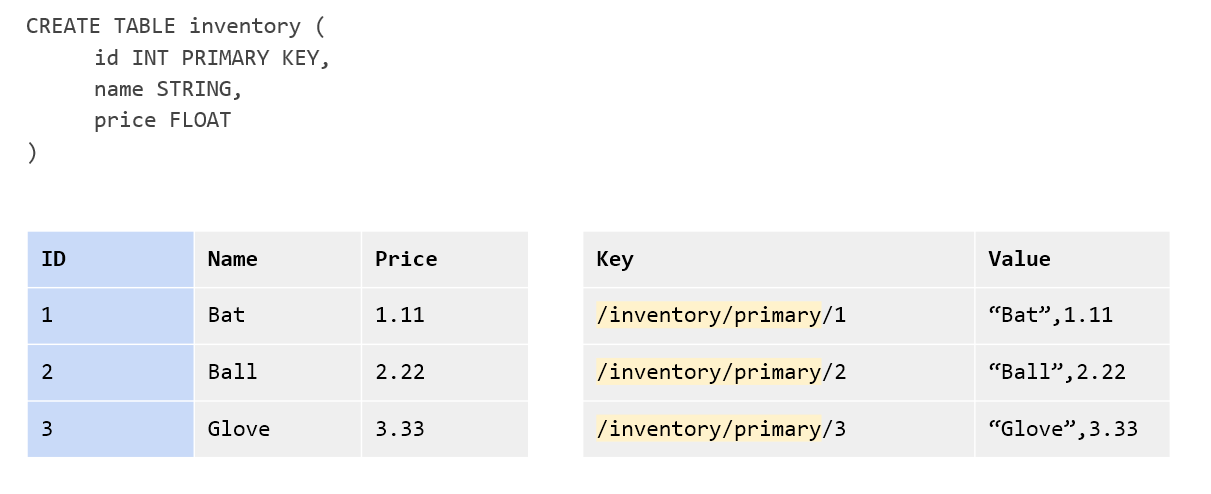


Figure 4 The table KV store contains table and index identifiers for each record.

Figure 4 shows the table name and index name (“primary”) as text, but for efficiency, these are actually represented as compact unique table and index identifiers.

### Column Families

In the above example, all the columns for a table are aggregated together in the Value section of a single KV entry. However, it’s possible to direct CockroachDB to store groups of columns in separate KV entries using \_Column Families\_. Each column family in a table will be allocated own KV entry. Figure 5 illustrates this concept – if a table has two column families, then each row in the table will be represented by two KV entries.

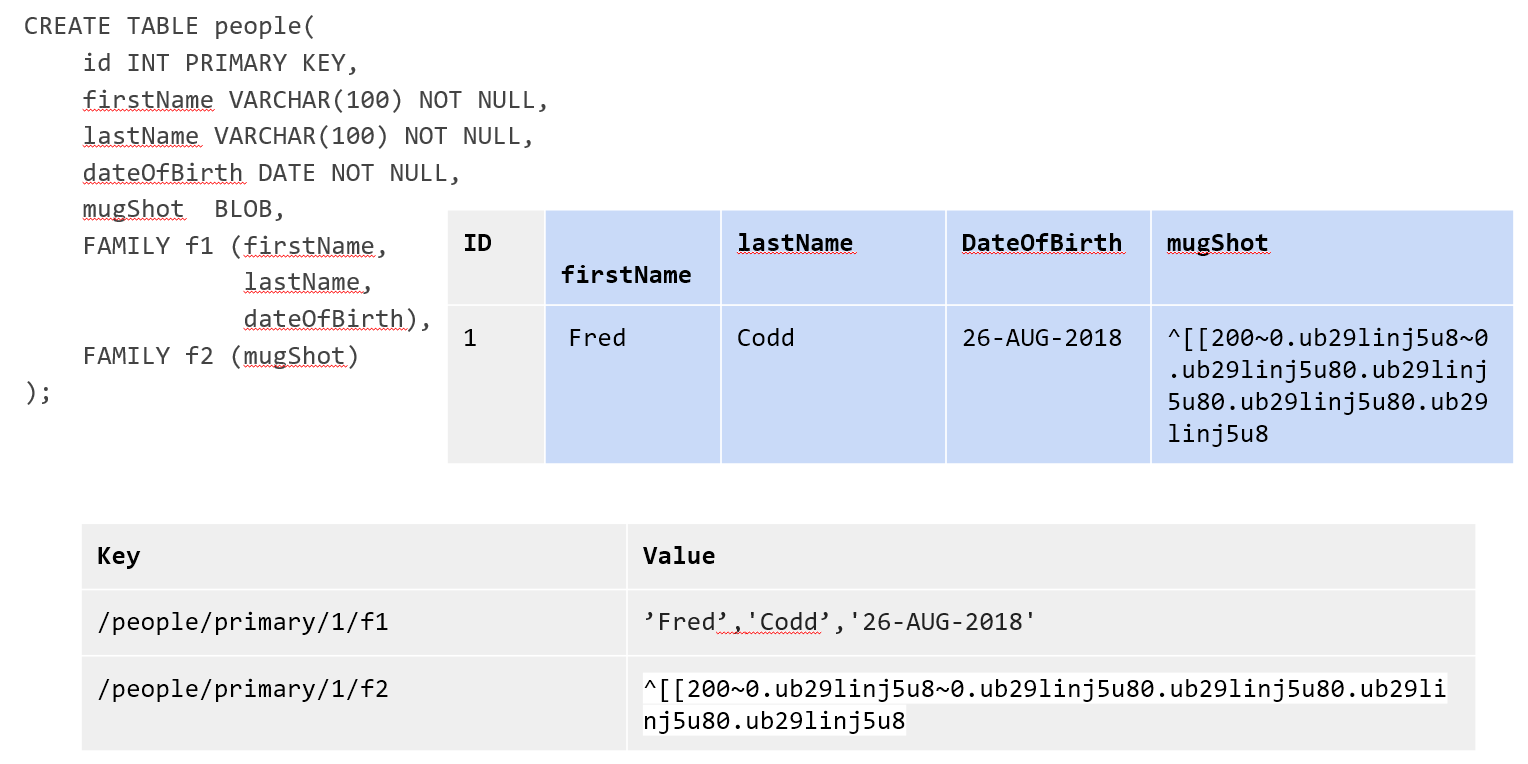


Figure 5 Column Families in the KV store

Storing columns can have a number of advantages. If infrequently accessed large columns are separated, then they will not be retrieved during row lookups which can improve the efficiency of the Key-Value store cache. Furthermore, concurrent operations on columns in separate column families will not interfere with each other.

### Indexes in the KV store

Indexes are represented by a similar KV structure. For instance, the representation of a non-unique index is shown in Figure 6

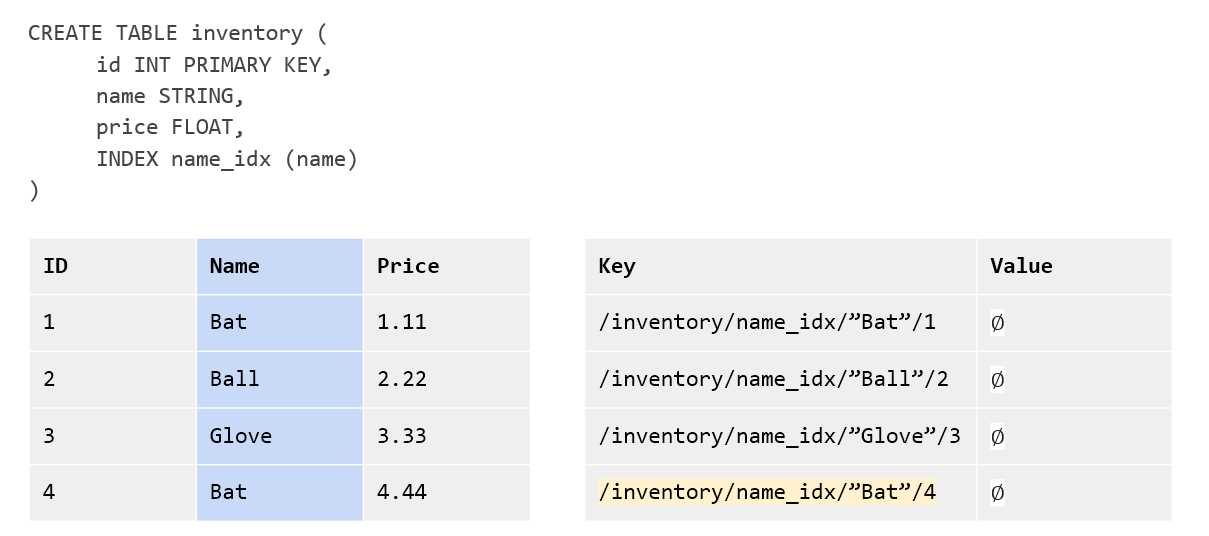


Figure 6 Non-unique index KV store representation

The key for a non-unique index includes the table and index name, the key value and the primary key value. For a non-unique index there is no “Value” by default.

For a unique index, the KV Value defaults to the value of the primary key. So, if +name+ was unique in the +inventory+ table used in previous examples, a unique index on name could be represented as shown in Figure 7.

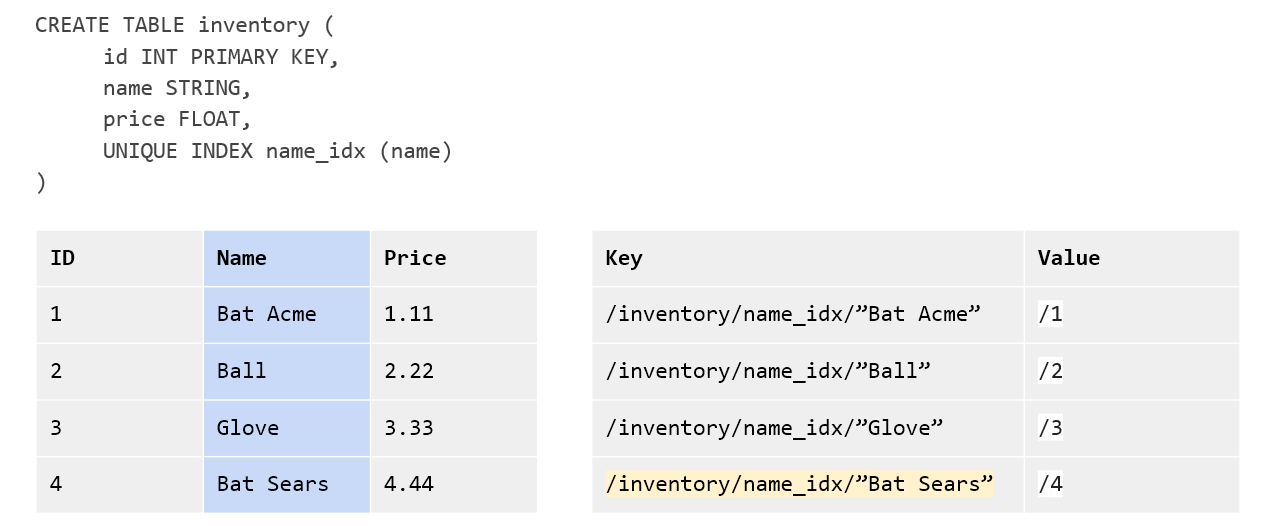


Figure 7 Unique index KV store representation

Inverted indexes allow indexed searches into values included in JSON documents. In this case, the key values include the JSON path and value together with the primary key - as shown in Figure 8.

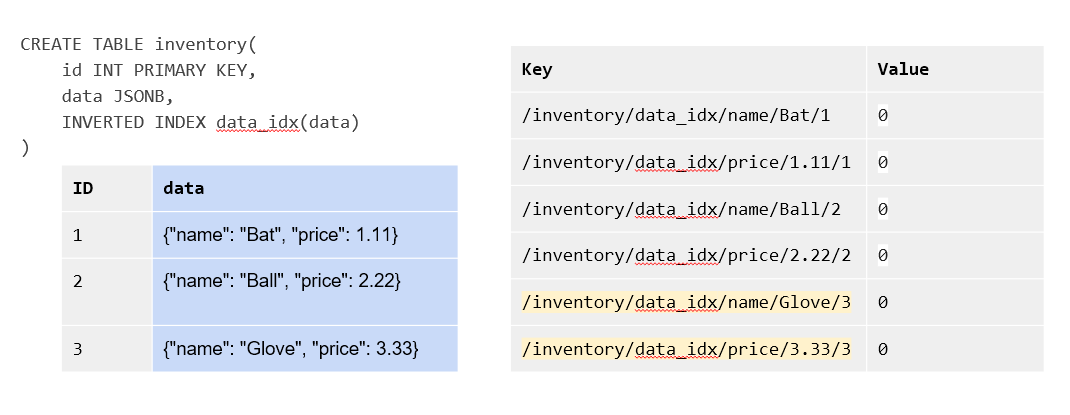


Figure 8 Inverted Index KV representation

Inverted indexes are also used Spatial indexes – see Chapter 8 for more details.

Inverted indexes can be larger and more expensive to maintain than other indexes, since single JSON document in a row will generate one index entry for each unique attribute. For very complex JSON documents, this might result in dozens of index entries for each document. We’ll also discuss this further – and consider some alternatives - in Chapter 8.

The +STORING+ clause of +CREATE INDEX+ allow us to add additional columns to the Value portion of the KV index structure. These additional columns cannot be used to optimize the search, but can streamline a query that contains a projection that includes only those columns and the index keys. For instance, in Figure 9 we see a nonunique index on name and date of birth that uses the STORING clause to add the phone number to the KV Value. Queries that seek to find the phone number using name and date of birth can now be resolved by the index without reference to the base table.

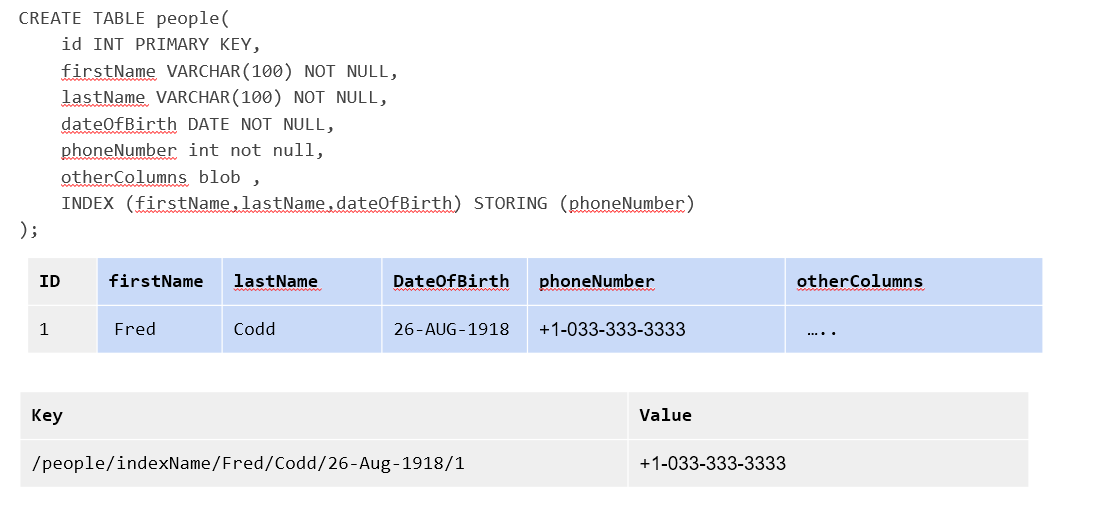


Figure 9 STORING clause of CREATE INDEX

## The CockroachDB Transactional layer

The transaction layer is responsible for maintaining the atomicity of transactions by ensuring that all operations in a transaction are committed or aborted.

Additionally, the transactional layer maintains serializable isolation between transactions – which means that transactions are completely isolated from the effects of other transactions. Although multiple transactions may be in progress at the same time, the experience of each transaction is as if the transactions were run one at a time – the **SERIALIZABLE** isolation level.

.isolation Levels

\*\*\*\*

Transaction "isolation levels" define to what extent transactions are isolated from the effects of other transactions. ANSI SQL defines four isolation levels which are, from weakest to strongest: +READ UNCOMMITTED+, +READ COMMITTED+, +REPEATABLE READ+ and +SERIALIZABLE+. Additionally, an isolation level of +SNAPSHOT+ is used by many databases as an alternative “strong” isolation level to +SERIALIZABLE+.

In some databases, users may choose a lower level of isolation in order to achieve some benefits in terms of concurrency.

However, CockroachDB supports only the +SERIALIZABLE+ level of isolation. This means that CockroachDB transactions must exhibit absolute independence from all other transactions. The results of a set of concurrent transactions must be the same as if they had all been performed one after the other.   
  
Even +SERIALIZABLE+ is arguably a compromise between performance and correctness. +LINEARIZABLE+ or +STRICT SERIALIZABLE + isolation levels place create even stronger guarantees that transactions will be sequenced in the exact order they occurred in the real world. However, in practice, +STRICT SERIALIZABLE+ isolation requires either specialized hardware (as in Spanner) or extreme limits on concurrency.

\*\*\*\*

The transactional layer processes key-value operations generated by the SQL layer. A transaction consists of multiple Key-Value operations, some of which may be the result of a single SQL statement. In addition to updating table entries, index entries must also be updated. Maintaining perfect consistency under all circumstances involves multiple sophisticated algorithms, not all of which can be covered in this chapter. For deep details, you may wish to consult the CockroachDB 2020 SIGMOD paper[[1]](#footnote-2) , which covers many of these principles in more detail.

### MVCC principles

Like most transactional database systems, CockroachDB implements the MultiVersion Concurrency Control (MVCC) pattern. MVCC allows readers to obtain a consistent view of information, even while that information is being modified. Without MVCC, consistent reads of a data item need to block simultaneous writes of that item and vice-versa. With MVCC, readers can obtain a consistent view of information even while the information is being modified by a concurrent transaction.

Figure 10 illustrates the basic principles of MVCC. At time t1, session s1 commences a transaction (1). At timestamp t2, s1 updates row r2 (2), creating a new version of that row (3). Also at timestamp t1, another database session s2 commences a transaction (4). When s2 attempts to read row r2 at time t2, it reads from the original version of the row - v1 (5). After both transactions commit, (5 & 6) session s2 will read from version v2 of the row (7).



Figure 10 MultiVersion Concurrency Control (MVCC)

The constraints of +SERIALIZABLE+ isolation limit the ability of transactions to read from previous versions. For instance, if a read transaction commences after a write transaction has commenced it may not be able to read the original version of the row because it might be inconsistent with other data already read or which will be read later in the transaction. In some cases, CockroachDB will “push” the read timestamp forward and/or restart the read to avoid blocking the read – we’ll discuss this “timestamp push” technique later in the chapter.

We'll see later on how the storage engine implements MVCC, but for now, the important concept is that multiple versions of any row are maintained by the system, and transactions can determine which version of the row to read depending on their timestamp and the timestamp of any concurrent transactions.

Transaction workflow

Distributed transactions must proceed in multiple stages. Simplistically, each node in the distributed system must lay the groundwork for the transaction and only if all nodes report that the transaction can be performed will the transaction be finalized.

Figure 11 illustrates a highly simplified flow of a transaction preparation. In this case, a two statement transaction is sent to the CockroachDB gateway node (1). The first statement involves a change to range 2, so that request is sent to the leaseholder for that range, which create a new tentative version of the range and propagates the changes to the other replicas for that range[[2]](#footnote-3). The second statement affects range 4, which likewise propagates the change to the LeaseholdeWhen all changes have correctly propagated, the transaction completes and the client is notified of success (9).



Figure 11 Basic transaction flow

### Pipelining and write intents

The Leaseholder transmits write commands to replicas in an asynchronous mode, referred to as **transaction pipelining**. The Leaseholder sends the write commands to the replicas and responds to the gateway without waiting for those replicas to acknowledge the writes

Of course, during these initial stages, it is not yet known whether the transaction will succeed, so it's premature to permanently apply the new values to the database. Instead, tentative modifications called **write intents** are created. Write intents are specially constructed MVCC-compliant versions of the records, which are marked as provisional. They serve both as tentative transaction outcomes and as locks that prevent any concurrent attempts to update the same record.

Inside the first key range to be modified by the transaction, CockroachDB writes a special **transaction record**. This transaction record contains the official status of the transaction. In the example shown in Figure 11, this transaction record would be stored in range 2 since that is the first range to be modified in the transaction.

This transaction record will record the transaction state as one of the following:

* **PENDING**: Indicates that the write intent's transaction is still in progress.
* **STAGING**: All transaction writes have been performed, but the transaction is not yet guaranteed to commit.
* **COMMITTED**: The transaction has been successfully completed.
* **ABORTED**: Indicates that the transaction was aborted and its values should be discarded.

### Parallel Commit

In a distributed database, the number of network round trips is often the dominant factor in latency. In general, committing a distributed transaction requires at least two round trips (indeed, one of the classic algorithms for this is called Two-Phase Commit). CockroachDB uses an innovative protocol called Parallel Commits to hide one of these round trips from the latency as perceived by the client.

The key insight behind Parallel Commits is that the gateway can return success to the client as soon as it becomes impossible for the transaction to abort, even if it is not yet fully committed. The remaining work can be done after returning as long as its outcome is certain. This is done by transitioning the transaction to the STAGING state in parallel with the transaction's last round of writes. The keys of all of these writes are recorded in the transaction record. A STAGING transaction must be committed if and only if all of those writes succeeded.

In the normal case, the gateway learns the status of these writes as soon as they complete, and returns to the client before beginning the final resolution of the transaction in the background. If the gateway fails, the next node to encounter the staging transaction record is responsible for querying the status of each write and determining whether the transaction must be committed or aborted (but because the transaction record and each write intent have been written durably, the outcome is guaranteed to be the same whether the transaction is resolved by its original gateway or by another node).

Note that any locks held by the transaction are not released until after this resolution process has completed. Therefore the duration of a transaction from the perspective of another transaction waiting for its locks is still at least two round trips (just like with Two-Phase Commit). However, from the point of view of the session issuing the transacation, elapsed time is significantly reduced.

### Transaction clean up

As discussed in the previous section, a COMMIT operation "flips a switch" in the transaction record to mark the transaction as committed, minimizing any delays that would otherwise occur when a transaction is committed. After the transaction has reached the COMMIT stage, then it will asynchronously resolve the write intents by modifying the write intent to become a normal MVCC record representing a new record value.

However, as with any asynchronous operation, there may be a delay in performing this cleanup. Furthermore, since a committed write intent looks just the same as a pending write intent, transactions that encounter a write intent record when reading a key will need to determine if the write intent is committed.

If another transaction encounters a write intent that has not yet been cleaned up by the transaction coordinator, then it can perform the write intent cleanup by checking the transaction record. The write intent contains a pointer to the transaction records, which can reveal if the transaction is committed.

Figure 12 illustrates the flow of a successful two-statement transaction. A client issues a DML statement (1). This creates a transaction coordinator which maintains a transaction record in PENDING state. Write intent commands are issued to the Leaseholder for the range concerned (2). The Leaseholder writes the intent markers to its copy of the data and transmits the same to all replicas. It returns success to the Transaction coordinator without waiting for the replica intents to be acknowledged (3).

Subsequent modifications in the transaction are processed in the same manner.

The client issues a COMMIT (3). The transaction co-coordinator marks the transaction status as STAGING. When all write intents are confirmed, the transaction status is set to COMMITTED, and the client is advised of success (4).

After successful commits, the transaction coordinator resolves write intents in affected ranges, which become normal MVCC records (5). At this point the transaction has released all its locks and other transactions on the same records are free to proceed.



Figure 12 Overall transaction flow

### Read/Write conflicts

So far, we've looked at the processing of successful transactions. It would be great if all transactions succeeded, but in all but the most trivial scenarios, concurrent transactions create conflicts that must be resolved.

The most obvious case is when two transactions attempt to update the same record. There cannot be two write intents active against the same Key, so either one of the transactions will wait for the other to complete, or one of the transactions will be aborted. If the transactions are of the same priority, then the second transaction – the one that has not yet created a write intent – will wait. However, if the second transaction has a high priority, then the original transaction will be aborted and will have to retry.

The **TxnWaitQueue** object tracks the transactions that are waiting and the transactions that they are waiting on. This structure is maintained within the Raft leader of the range associated with the transaction. When a transaction commits or aborts, the TxnWaitQueue is updated, and any waiting transactions are notified.

A **Deadlock** can occur if two transactions are both waiting on write intents created by the other transaction. In this case, one of the transactions will be randomly aborted.

Transaction conflicts can also occur between readers and writers. If a reader encounters an uncommitted write intent that has a lower timestamp than the consistent read timestamp for the read, then a consistent read cannot be completed. This can occur if a modification occurs between the time a read transaction starts and the time it attempts to read the key concerned. In this scenario, the timestamp for the read is "pushed" past the modification timestamp to allow for a consistent read to be returned. This will often require that the read be restarted at the higher timestamp.

<< A Diagram might help here >>

Many transaction conflicts are managed automatically, and while these have performance implications, they don't impact functionality or code design. However, there are multiple scenarios in which an application may need to handle an aborted transaction. We'll look at these scenarios and discuss best practices for transaction retries in Chapter 6.

### Clock synchronization and clock skew

You may have noticed in previous sections that CockroachDB must compare timestamps of operations frequently to determine if a transaction is in conflict. Simplistically, we might imagine that every node in the system can agree on the time of each operation and make these comparisons easily. Unfortunately, in widely distributed systems with very high transaction rates it is actually non-trivial for different nodes to agree on the exact sequence of events.

Most systems use the Network Time Protocol (NTP) to periodically update their system clock. This results in a high degree of concordance between nodes – far better than a human would require. However, when we think of transaction rates in the thousands of TPS, even a small amount of discrepancy – clock skew – makes it hard to determine the exact sequence of events

You may remember that Google Spanner attempted to solve this problem using specialized hardware – atomic clocks and GPS devices. Some NoSQL databases dispensed with the idea of physical time (e.g. the actual time as would be reported by a wall clock) altogether by using **Vector clocks**. A vector clock is a local sequence number that is incremented with every transaction. When a transaction is propagated, it includes a vector of such timestamps from all the transactions received by that node. Nodes can compare vector clock information to determine if two transactions are causally related. Unfortunately, vector clocks have proven to be unwieldy in large, widely dispersed distributed systems – the size of the vectors transmitted with each transaction grows with the number of nodes in the system.

CockroachDB uses a **Hybrid Logical Clock** (**HLC**) to provide a practical solution to time synchronization. HLC combines physical time with the Vector clock concept. As with a vector clock, nodes transmit vectors of timestamp sequences from transactions they have received. However, the node also uses the physical time to discard vector information that is outside the amount of clock skew that might be expected between nodes. This keeps the vectors from growing to an unwieldy size. Using the HLC mechanism, CockroachDB can tolerate significant clock skew. By default, CockroachDB nodes can maintain their place in the cluster, providing they are not more than 500ms out of step with other nodes.

[NOTE]

====

Hybrid Logical Clocks can't determine the exact clock time for every transaction. However, they do help CockroachDB to determine the order in which transactions have been processed across multiple nodes.

====

Even with advanced algorithms like HLC or with hardware supported mechanisms such as TrueTime, there will still be transactions that can’t be effectively ordered. Two transactions that operate on unrelated key values that still have some real-world sequencing dependency might have appear to be committed in reverse order – the **causal reverse** anomaly. This is not a violation of +SERIALIZABLE+ isolation because the transactions are not actually sequenced logically. The second transaction was not dependent on any data referenced by the first transaction.

## The distribution layer

Logically, a table is represented in CockroachDB as a monolithic Key-Value structure, in which the Key is a concatenation of the primary keys of the table, and the value is a concatenation of all of the remaining columns in the table. We introduced this structure back in Figure 2 .

The distribution layer breaks this monolithic structure into contiguous chunks of approximately 512MB. The 512MB chunk size is considered small enough to be shuffled between nodes when required but large enough to provide for "similar" records to be accessed in a small number of IOs.

### Meta Ranges

The distribution of ranges is stored in global keyspaces +meta1+ and +meta2+. +meta1+ can be thought of as a "range of ranges" lookup, which then allows a node to find the location of the node holding the +meta2+ record, which in turn points to the leaseholder and replica nodes for every range within the "range of ranges". Figure 13illustrates this two-level lookup structure.

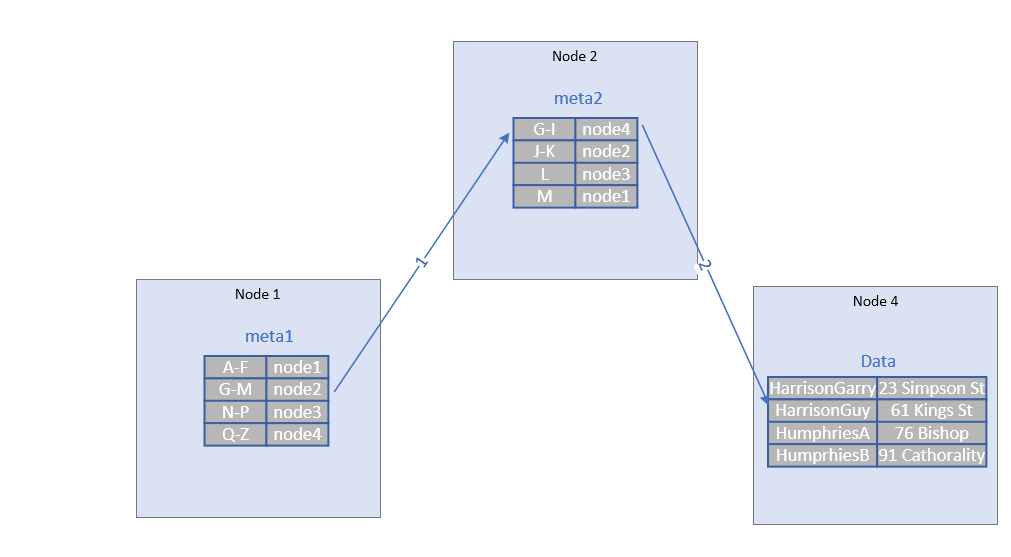


Figure 13 Meta Ranges

In Figure 13, Node1 needs to get data for the key "HarrisonGuy". It looks in its copy of +meta1+, which tells it that node2 contains the +meta2+ information for the range G-M. It accesses the +meta2+ data concerned from node2, which indicates that node4 is the Leaseholder for the range G-I, and therefore the Leaseholder for the range concerned.

Figure 13 oversimplifies these meta ranges significantly: in CockroachDB, the meta ranges include information about the replicas as well as the Leaseholders, allowing lookups to proceed if a node is unavailable and potentially to look at a local replica of the +meta2+ structure on a local node.

### Leaseholders

The Leaseholder of the CockroachDB node responsible for serving reads and coordinating writes for a specific range of keys. We discussed some of the responsibilities of the Leaseholder in the transaction section. When a transaction coordinator or gateway node wants to initiate a read or write against a range, it finds that range's leaseholder (using the meta ranges structure discussed in the previous section) and forwards the request to the Leaseholder.

Leaseholders are assigned using the Raft protocol, which we will discuss in the Replication layer section below.

### Range Splits

CockroachDB will attempt to keep a range at less than 512MB. When a range exceeds that size, the range will be split into two smaller contiguous ranges.

Ranges can also be split if they exceed a load threshold. If the parameter +kv.range\_split.by\_load\_enabled+ is true and the number of queries per second to range exceeds the value of +kv.range\_split.load\_qps\_threshold+, then a range may be split even if it is below the normal size threshold for range splitting. Other factors will determine if a split actually occurs, including whether the resulting split would actually split the load between the two new ranges and the impact on queries that might now have to span the new ranges.

When splitting based on load, the two new ranges might not be of equal sizes. By default, the range will be split at the point at which the load on the two new ranges will be roughly equal.

Ranges can also be split manually using the +SPLIT AT+ clause of the +ALTER TABLE+ and +ALTER INDEX\_ statements.

Ranges can be merged if they fall below a size threshold (which parameter?) or can be merged manually using the +UNSPLIT AT+ clause of the +ALTER TABLE+ and +ALTER INDEX\_ statements.

Figure 14 illustrates a basic range split caused with an insert causes a range to exceed the 512MB threshold. Two ranges are created as a consequence.

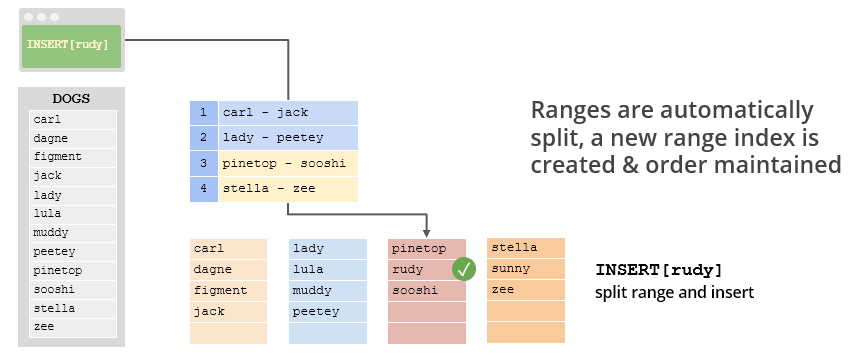


Figure 14 Range Splits

### Geo-partitioning

Geo-partitioning is a special feature of CockroachDB Enterprise that allows data to be located within a specific geographic region. This might be desirable from a performance point of view – reducing latencies for queries from a region about that region – or from a data sovereignty perspective – keeping data within a specific geographic region for legal or regulatory reasons.

Geo-partitioning can be applied to all replicas for a table or for leaseholders only.

With geo-distributed Replicas, the table is partitioned by the first column in its composite Primary Key. Nodes in the cluster are associated with regions corresponding to that first column. All ranges for that region – including replica ranges - are located within the nodes corresponding to the region.

With geo-partitioned leaseholders, only the leaseholders are pinned to the regions with replicas free to be placed in other regions. This topology pattern is more resilient than the geo-distributed replica topology but may reduce throughput since replication will have to span wider geographic distances.

(Internals? Are the partitions separate tables from the point of view of the distribution layer?)

## Replication layer

High availability requires that data not be lost or made unavailable should a node fail. This of course requires that multiple copies of data be maintained. There must be a **distributed consensus** mechanism that ensures that these multiple copies of data are correctly synchronized.

### Raft

CockroachDB employs the widely used **Raft protocol[[3]](#footnote-4)** as it’s distributed consensus mechanism. In CockroachDB each range is a distinct Raft group – distributed consensus for each range is determined independently for every range.

In Raft and in most distributed consensus mechanisms we need a minimum of 3 nodes. This is because a majority of nodes (a quorum) must always be agree on the state. In the event of a network partition, the only side of the partition with the majority of nodes can continue.

In a Raft group, one of the nodes is elected as **leader** by a majority of nodes in the group. The other nodes are known as **followers**. The Raft leader controls changes to the raft group.

Changes sent to the Raft leader are written to it’s **Raft log** and propagated to the followers. When a majority of nodes accept the change, then the change is committed by the leader.

Leader elections occur regularly or may be triggered when a node fails to receive a heartbeat message from the leader. In the later case, a follower who cannot communicate with the leader will declare itself a candidate and initiate an election. Raft includes a set of safety rules that prevent any data loss during the election process. In particular, a candidate cannot win an election unless its log contains all committed entries.

Nodes that are temporarily disconnected from the cluster can be sent relevant sections of the Raft log to re-synchronize or – if necessary – a point in time snapshot of the state followed by a catch up via Raft logs.

### Raft and Leaseholders

The CockroachDB Leaseholder and the Raft leader responsibilities serve very similar purposes. The Leaseholder controls access to a range for the purposes of transactional integrity and isolation, while the Raft Leaseholder controls access to a range for the purposes of replication and data safety.

The Leaseholder is the only node that can propose writes to the Raft leader. CockroachDB will attempt to elect a Leaseholder which is also the Raft leader so that these communications can be streamlined. The Leaseholder already knows that its writes have achieved consensus so it can service reads without passing them to the Raft leader.

## The Storage layer

We’ve touched upon the logical structure of the Key-Value store earlier in the chapter when we discussed the Key-Value store. However, we have not yet looked at the physical implementation of the Key-Value storage engine.

As of CockroachDB version 20, CockroachDB uses the Pebble storage engine – an open source Key-Value store based on inspired by the LevelDB and RocksDB storage engines. PebbleDB is primarily maintained by the CockroachDB team and is optimized specifically for CockroachDB use cases. At the time of writing, RocksDB was available as an option to ensure backward compatibility.

### Log Structured Merge (LSM) Tree

PebbleDB and RocksDB both implement the Log Structured Merge Tree (LSM) architecture.

LSM is a structure that seeks to optimize storage and support extremely high insert rates, while still supporting efficient random read access.

The simplest possible LSM tree consists of two indexed “trees”:

* An in-memory tree which is the recipient of all new record inserts - the **MemTable**.
* A number of on-disk trees which represent copies of in–memory trees which have been flushed to disk. These are referred to as **Static Sorted Tables (SSTables).**

SSTables exist at multiple levels – L0 is simply a copy of the in-memory MemTable that has been flushed to disk. Periodically, SStables are periodically compacted into larger consolidated stores. SSTables are internally sorted and indexed so lookups within an SSTable are fast.

The LSM architecture ensures that writes are always fast, since they operate at memory speed. The transfer to disk is also fast since it occurs in append only batches which allow for fast sequential writes. Reads occur either from the in-memory tree or from the disk tree; in either case reads are facilitated by an index and are relatively swift.

Of course, if a node fails while data is in the in-memory store then it could be lost. For this reason, database implementations of the LSM pattern include a **Write Ahead Log** (WAL) that persists transactions to disk. The WAL is written via sequential writes which are faster than random writes, although this performance advantage is not as critical now with Solid State Disks that support random writes more effectively.

Figure 15 illustrates LSM writes. Writes from higher CockroachDB layers are first applied to the CommitLog (1) and then to the MemTable (2). Once the MemTable reaches a certain size it is flushed to disk to create a new SSTable (3), once the flush completes WAL records may be purged (4). Periodically multiple SSTables are merged (compacted) into larger SSTables (5).



Figure 15 LSM writes

### SSTables and Bloom Filters

Each SSTable is indexed. However, there may be many SSTables on disk and this creates a multiplier effect on index lookups, since we would theoretically have to examine every index for every SSTable in order to find our desired row.

To avoid these multiple index lookups **Bloom filters** are used to reduce the number of lookups which must be performed.

Bloom filters are created by applying multiple hash functions to the key value. The outputs of the hash functions are used to set bits within the bloom filter structure. When looking up a key value within the bloom filter we perform the same hash functions and see if the bits are set. If the bits are not set, then the search value must not be included within the table. However, if the bits are set it may have been as a result of a value that happened to hash to the same values. The end result is an index which is typically reduced in size by 85% but which provide false positives only 15% of the time.

Bloom filters are compact enough to fit in memory and are very quick to navigate. However, to achieve this compression, bloom filters are “fuzzy” in the sense that they may return false positives. If you get a positive result from a bloom filter it means only that the file \_may\_ contain the value. However, the bloom filter will never incorrectly advise you that a value is not present. So, if a bloom filter tells us that a key is not included in a specific SSTable, then we can safely omit that SSTable from our look up.

Figure 16 shows the read pattern for a log structured merge tree using Cassandra terminology. A database request first reads from the MemTable (1). If the required value is not found it will consult the Bloom filters for the most recent SSTable (2). If the bloom filter indicates that no matching value is present, it will examine the next SSTable (3). If the Bloom filter indicates a matching key value may be present in the SSTable, then the process will use the SSTable index (4) to search for the value within the SSTable (5). Once a matching value is found, no older SSTables need be examined.



Figure 16 LSM reads

### Deletes and updates

SSTables are immutable - once the MemTable is flushed to disk and becomes an SSTable, no further modifications to the SSTable can be performed. If a value is modified repeatedly over a period of time, the modifications will build up across multiple SSTables. When retrieving a value, the system will read SSTables from youngest to oldest to find the most recent value of a column, or to build up a complete row. Therefore, to update a value we only need to insert the new value, since the older values will not be examined when a newer version exists.

Deletions are implemented by writing tombstone markers into the MemTable which eventually propagates to SSTables. Once a tombstone marker for a row is encountered, the system stops examining older entries and reports “row not found” to the application.

As SSTables multiply, read performance and storage will degrade as the number of bloom filters, indexes and obsolete values increase. During compaction rows that are fragmented across multiple SSTables will be consolidated, and deleted rows removed.

### The Block cache

PebbleDB and RocksDB maintain a block cache providing fast access to frequently accessed data items. This block cache is separate from the in-memory indexes, bloom filters and memtables. The block cache operates on a Least Recently Used basis.

Reading from the blockchain bypasses the need to scan multiple SStables and associated bloom filters. We’ll speak more about the cache in chapter 14 when we discuss cluster optimization.

1. <https://resources.cockroachlabs.com/guides/cockroachdb-the-resilient-geo-distributed-sql-database-sigmod-2020> [↑](#footnote-ref-2)
2. We are assuming here that the Leaseholder is also the Raft leader, which is normally the case. More on Raft later in the chapter. [↑](#footnote-ref-3)
3. <https://en.wikipedia.org/wiki/Raft_(algorithm)> [↑](#footnote-ref-4)