[[Ch05 – CockroachDB Schema Design]]

# == CockroachDB Schema Design

A sound data model is the foundation of a highly performant and maintainable application. In this chapter, we’ll review the fundamentals of relational schema design, with a particular focus on aspects of schema design that bear on distributed database operations and on advanced CockroachDB features such as column families and JSONB support. We’ll cover the creation of tables, indexes and other schema object that support a well-designed CockroachDB application.

Although CockroachDB supports mechanisms for efficiently altering schemas online, schema changes to production application are nevertheless high impact changes, typically involving co-ordinated changes to application code and production database configuration. If done poorly, there’s risk of loss of application functionalilty, availability of peformacne. Therefore, although it’s quite possible to alter CockroachDB schemas in production, its far better to get the schema right during application design.

Relational database design is a big topic and has been the subject of many books and continuing debate. We don’t want to try to cover advanced design principles here, nor do we want to engage on any debates about the purity of various design patterns. Most database models are a compromise between the mathematical purity of the relational model and the practicalities imposed by the physical database system. Therefore, in this chapter, we’ll attempt to only briefly cover the theoretical side of the relational model, but dive quite deep into the practicalities of designing a model that will work well with a CockroachDB implementation.

## === Logical Data modelling

Application data models are commonly created in two phases. Establishing the logical data model involves modeling the information that will be stored and processed by the application and ensuring that all necessary data is correctly, completely and unambiguously represented.

The logical data model is then mapped to a physical data model. The physical data model describes the tables, indexes, views that are created in the DBMS.

The logical data model typically satsifies only the functional requirements of the application. The physical data model must also satisfy non-functional requirements, most particularly performance requirements.

In practice, these two phases are often blurred together, especially in agile and other iterative development environments. Nevertheless, whether done explicitly or not, there is definitely a difference between the analysis required to determine \*what\* data an application might process, and \*how\* that data is best represented in a specific database system.

We introduced some of the core concepts of the relational model in Chapter 1. Theoretically during logical data modelling we deal with relations, tuples and attributes, while in physical design we deal with tables, rows and columns. However, outside of academia, these distinctions are often ignored, and in practice its common place to develop a logical model using the language of tables and columns.

.Mea Culpa

\*\*\*\*

Relational data modelling has generated an immense volume of research and debate over the past four decades. It’s almost impossible to say anything sensible about relational data modelling without oversimplification or misrepresentation.   
  
This is a book about CockroachDB, not about the relational theory and so we by necessity need to avoid getting bogged down in debates about the correct way to perform relational design. Our purpose here is to provide enough quick background on relational modelling to allow for us to sensibly talk about CockroachDB specific physical design principles. Our apologies to anyone who feels we’ve failed to adequately cover this complex and nuanced topic.

\*\*\*\*

### ==== Normalization

A normalized data model is one in which any data redundancy has been eliminated and in which all data is completely identifiable by primary and foreign keys. Although the normalized data model is rarely the final destination from a performance point of view, the normalized data model is almost always the best initial representation of an schema, since it minimizes redundancy and ambiguity.

Relational theory defines multiple “levels” of normalization. The third normal form is the generally accepted standard of an adequately normalized data model.

In third normal form, every attribute (column) in a tuple (row) are dependent only on the entire primary key of that tuple and not on any other attribute or key. We sometimes remember this as “The key, the whole key and nothing but the key”.

For example consider the data shown in <<StudentTestsDenormalized>>.

[[StudentTestsDenormalized]]

.Student Test Data (Denormalized)

image::images/StudentTestsDenormalized.png[Delete Statement]

Graphical user interface, table

Description automatically generated

Even if we created a primary key on studentname,testname and testdate, we would still be a long way from third normal form. Attributes such as studentdob are dependent only on part of the key (studentName) and the repeating “answer” columns are dependent on a non-key attribute (the corresponding question) attribute.

A normalized version of this data is shown in <<StudentTestsNormalized>>. Students take tests, tests have questions and students answer those questions. All attributes in each relation are now fully dependent on the primary keys for that relation.

[[StudentTestsNormalized]]

.Student Test Data (Normalized)

image::images/StudentTestsNormalized.png[StudentTestsNormalized]

Diagram

Description automatically generated

### ==== Don’t go too far

You’ll generally recognize a well-normalized data model by the absence of any redundant information. For instance, in <<StudentTestsNormalized>>, you’ll notice that student name, test names, question texts, etc are never repeated across multiple entries. There is one and only one entry for each attribute. The only thing that is repeated in a well-normalized model should be foreign key references.

That having been said, it’s often a mistake to take the normalization process too far. In a real world database, each new table in the model adds complexity to program code, and overhead in joining information during data retrieval.

For example, from time to time we see addresses “normalized” as in <<studentsOvernormalized>>.

[[studentsOvernormalized]]

.Student Address model (Normalized)

image::images/studentsOvernormalized.png[studentsOvernormalized]

Diagram

Description automatically generated

There’s nothing theoretically wrong with this model. Two students could share a flat, and the relationships between cities, states and countries are very real. We could even through in Continents, solar systems and galaxies entities into the model without violating Third Normal Form.

However, in practice, this model will require a five-way join whenever a student’s address needs to be retrieved. Since this is a pretty common operation the cost of the join across the life of the application will be high. It’s probably better just to leave the address fully embedded in the students table as shown in <<StudentTestsNormalized>>.

Purists will definitely argue that this sort of denormalization should be performed in the physical data modelling stage. And it’s also worth noting that there may be good reasons for having a CITY or STATE table in a production system – each might be associated with specific attributes of relevance. But we’d suggest that you be pragmatic when contemplating such extended relationships. Theres a lot of wasted effort in modelling logical relationships that are inevitably going to be collapsed in the physical design phase.

### ==== Primary Key choices

In CockroachDB, the choice of primary keys is critical to performance because it is the primary key which will determine the distribution of data across the nodes in the system. We’ll talk extensively about this in the next major section.

However, even from logical modelling point of view there are some factors to consider.

Third normal form requires that each relation have a primary key. However, it does not specify whether that key should be artificial or synthetic. A *natural key* is one constructed from unique attributes that occur normally within the entity. An *artificial key* is one which contains no meaningful column information and which exists only to uniquely identify the row. There is a continual debate within the database community regarding the merits of “artificial” primary keys versus the “natural” key.

In general, we are of the opinion that most fundamental entities should use an artificial key. Artificial keys are generally superior from a performance point of view, and eliminate some of the awkwardness involved when a natural primay key is changed. Furthermore, in CockroachDB, the use of an artificial key provides us with methods for ensuring a even distribution of keys throughout the cluster.

### ==== Special purpose Designs

For any given set of data, there usually exists more than one way to create a nearly correct relational model. Within the universe of possible models there exist some patterns that are particularly applicable to certain workloads. To of the most common are:

\* \*Data warehousing\* designs such as the \*star\* and \*snowflake\* schemas. These models have a large central “fact” table with foreign keys to multiple “dimension” tables. CockroachDB is not primarily intended as a datawarehousing database, and so these models are not typical of a CockroachDB deployment.

\* \*TimeSeries\* designs in which the time of origin of data is part of each data elements key, and in which data accumulates primarily as continual inserts. We’ll briefly consider some of the considerations for timeseries in the next section.

## === Physical design

Physical design involves modifying the logical design so as to improve its performance or maintainability. Some of these changes are driven by workload considerations. For instance if a table is only ever accessed in a JOIN with another table, we might replicate some columns from the second table into the first to avoid the join.

The other primary physical design driver is the capabilities and performance charactaristics of the database engine. For instance, in CockroachDB ascending primary keys cause hotspots on certain nodes and should be avoided, while in a non-distributed SQL database such as PostgreSQL ascending keys are fine.

### ==== Entities to tables

The major output of the logical design process are entities, attributes and keys. To convert the logical model to a physical model we need to convert entities to tables, and attributes to columns.

Depending on your logical model, this conversion may be close to a one-to-one mapping. However, be aware that in some cases a single entity may map to multiple table or vice-versa. For instance we might decide that the logical model shown in <<studentsOvernormalized>> should be collapsed to a single table, folding all address attributes into the Student table. Or we might collapse the addresses entity into students and collapse states and countries into cities.

In some cases a logical model may include \*subtypes\* in which an entity is defined that includes multiple “types” of tuple. For instance a PEOPLE entity might be defined as shown in <<Subtypes>>.

Graphical user interface

Description automatically generated with medium confidence

[[Subtypes]]

.Logical Model with Subtypes

image::images/subtypes.png[Subtypes]

People can be customers or employees (or both). So should there be a single PEOPLE table, a CUSTOMER and an EMPLOYEE table or even three tables: a PEOPLE table with attributes common to customers and employees and CUSTOMERS and EMPLOYEES tables with attributes unique to each type?

The answer determines on your workload and performance requirements. Each of the above solutions has a performance advantage for a certain class of query; you’ll need to think through the operations that are most important to your application. However, the solution that we’ve seen most often is the two table model (CUSTOMERS and EMPLOYEES).

### ==== Attributes to Columns

When mapping attributes to columns we are mainly concerned with selecting the best datatype for the column and defining it’s nullability correctly.

Null values are an important concept in relational databases – they distinguish between data that has a known value and data which is unknown or missing. Three valued logic: TRUE/FALSE/NULL is at the heart of SQL operations such as WHERE.

In some systems, the use of NULLS in indexed columns is discouraged and it is recommended to use NOT NULL with a DEFAULT value. This is because in some databases (CockroachDB for instance) NULL values are not included in indexes. However, CockroachDB stores NULL values in indexes and you can use an index to evaluate a IS NULL where condition.

CockroachDB datatypes generally map easily to logical datatypes. The following considerations may be considered:

\* All these CockroachDB string datatypes are equivalent: TEXT,CHAR, VARCHAR, CHARACTER VARYING and STRING.

\* All of the integer datatypes: INT, INT2,I NT4, INT8, BIGINT, SMALLINT, etc, are all stored in the same manner in the database. A BIGINT and a SMALLINT consume the same storage (providing they hold the same value). The types only serve to contrain the ranges of values that can be stored. The INT type can hold any allowable integer value (a 64-bit signed integer).

\* Similarly, FLOAT, FLOAT4, FLOAT8 and REAL datatypes all store 64-bit signed floating point numbers).

\* DECIMAL stores exact fixed point number and should be used when it’s important to preserve precision such as for monetary values.

\* BYTES, BYTEA and BLOB store binary strings of variable length. The data is stored in-line with other row data and therefore this datatype is not suitable for very large objects (a maximum of 1MB is suggested).

\* TIME stores a time value in UTC, while TIMETZ stores a time value with a timezone offset from UTZ. TIMESTAMP and TIMESTAMPZ are similar but include both date and time in the value.

Some of the other CockroachDB data types – ARRAYS, JSON, etc - will be discussed later in the chapter.

### ==== Primary key design

We’ve touched upon the importance of properly defining CockroachDB primary keys in preceding chapters: now it’s time to get serious about this very important topic.

The primary key of a table is used to distribute the ranges of that tables data across the cluster. If the primary key value is monotonically increasing, then all new data will go into a new range and will be sent to a specific node in the cluster. Most likely, this node will become a hot-spot and limit the insert throughput for the cluster.

The same phenomenon can be encountered in a timeseries database in which the primary key is prefixed with a timestamp. All “new” data will hit a single node and your cluster scalability will be compromised.

For instance, consider this implementation of the ORDERS table:

[source,sql]

----

**CREATE** **SEQUENCE** order\_seq;

**CREATE** **TABLE** orders (

salesorderid **INT** **NOT** **NULL** **PRIMARY** **KEY** **DEFAULT** **nextval**('order\_seq'),

orderdate **DATE** **NOT** **NULL** **DEFAULT** **now**() ,

duedate **DATE** **NOT** **NULL**,

shipdate **DATE** **NULL**,

customerid **INT** **NOT** **NULL**,

salespersonid **INT** **NULL**,

totaldue **DECIMAL** **NULL**

);

----

The orders\_seq sequence generator generates numbers that are guaranteed to be incrementing and – most of the time – without gaps. Since every value of ORDERID is one higher than the preceding value, new orders will be inserted into a single range which will be located on a single node. Consequently, that node will bear the burden of all INSERT operations. As new ranges are created, the responsibility of handling inserts will shift to new nodes, but at any given point in time, just a single node will be handling all of the inserts.

In the next few sections we’ll look at ways of avoiding this primary key “hot spot” anti-pattern.

#### ===== UUID based primary keys

If your application doesn’t need primary key values to be an continuously increasing value, then a UUID (Universally Unique IDentifier) primary key is the recommended solution.

A UUID is a value which is guaranteed to be unique across all systems. A UUID combines host-specific data, random numbers and timestamp data to generate an identifier that will be unique across systems and times.

The + gen\_random\_uuid()+ function generates UUIDs and can be used as the default value for a primary key, as in the example below.

[source,sql]

----

**CREATE** **TABLE** orders (

orderid uuid **NOT** **NULL** **PRIMARY** **KEY** **DEFAULT** gen\_random\_uuid(),

orderdate **DATE** **NOT** **NULL** **DEFAULT** **now**() ,

duedate **DATE** **NOT** **NULL**,

shipdate **DATE** **NULL**,

customerid **INT** **NOT** **NULL**,

salespersonid **INT** **NULL**,

totaldue **DECIMAL** **NULL**

);

----

UUIDs are unique, selective and guaranteed to be evenly distributed across all nodes of a cluster. They are therefore the preferred mechanism for CockroachDB primary keys.

#### .The SERIAL Datatype

\*\*\*\*

In PostgreSQL the SERIAL datatype is typically used to create auto-incrementing key values. It’s a handy alternative to creating a sequence as shown in an earlier example.

However, in CockroachDB the SERIAL datatype by default generated unique identifiers using the +unique\_rowid()+ function. +unique\_rowid()+ generates unique numbers that combine nodeid and timestamp. While the numbers are generally ascending, the order is not absolutely guaranteed, and gaps may occur.

You can change the behavior of SERIAL to a more PostgreSQL compatible behavior using the session variable +serial\_normalization+. However, the CockroachDB team recommends that you not use SERIAL data types unless compatibility with PostgreSQL is required.

\*\*\*\*

#### ===== Avoiding hotspots with a composite key

It may be that your application requirements really require a monotonically increasing key value. One way to avoid a hotspot in this case is to create a composite primary key that leads with a non-monotonically increasing value. For instance, in this implementation, the customerid is prefixed to the order number in the primary key:

[source,sql]

----

**CREATE** **TABLE** orders (

orderid **INT** **NOT** **NULL** **DEFAULT** **nextval**('order\_seq'),

orderdate **DATE** **NOT** **NULL** **DEFAULT** **now**() ,

duedate **DATE** **NOT** **NULL**,

shipdate **DATE** **NULL**,

customerid **INT** **NOT** **NULL**,

salespersonid **INT** **NULL**,

totaldue **DECIMAL** **NULL**,

**PRIMARY** **KEY** (customerid,orderid)

);

----

This implementation tends to send orders for a specific customer into the same ranges, but sequential orders from multiple customers should be distributed across the cluster.

There may be some upside in “clustering” customer data in this way, but the clear downside is that we now need to know the customer id when searching for an order. We’ve probably all experienced the irritation of having to provide both a customer identifier AND an order identifier to a sales associate so this downside is potentially significant. Of course, we could create a secondary index just on ORDERID, but then we’d create a secondary index with a hotspot.

What we need is a way to index monotonically increasing key values without creating unscalable insert hot-spots. The solution is \*hash sharded indexes\*.

[[.hashSharding]]

#### ===== Hash Sharded Primary Keys

Hash sharded indexes add a hashed value to the prefix of a primary key. These hash values are unique, but non-sequential. Consequently, if the primary key of a table is based on a hash sharded index, then it’s values will be distributed evenly across all the ranges in the cluster. The result should be (statistically) a perfect distribution of writes across nodes.

At time of writing hash-sharded indexes required the +experimental\_enable\_hash\_sharded\_indexes+ be set to true. So to create a hash sharded primary key we would use this syntax:

[source,sql]

----

**SET** experimental\_enable\_hash\_sharded\_indexes=**on**;

**CREATE** **TABLE** orders (

orderid **INT** **NOT** **NULL** **DEFAULT** **nextval**('order\_seq'),

orderdate **DATE** **NOT** **NULL** **DEFAULT** **now**() ,

duedate **DATE** **NOT** **NULL**,

shipdate **DATE** **NULL**,

customerid **INT** **NOT** **NULL**,

salespersonid **INT** **NULL**,

totaldue **DECIMAL** **NULL** ,

**PRIMARY** **KEY** (orderid) **USING** **HASH** **WITH** **BUCKET\_COUNT**=6

);

----

The hash sharding is transparent to the application: you’ll never see the hashed values and all filters against existing primary keys will work as normal. However, the new index cannot be used to find ranges of primary keys or to sort output by primary key. For instance, with a traditional primary key, we could retrieve the last 1000 orders in their orderid sequence with a query like this and the primary key index could be used to perform the range scan and sort the results:

[source,sql]

----

**SELECT** \* **FROM** orders

**WHERE** ORDERID>0

**ORDER** **BY** ORDERID;

----

With a hash sharded index a full scan and a sort operation would be required.

The WITH BUCKET\_COUNT clause determines how many “shards” of the index are created. CockroachDB creates a computed column for each bucket, creates an index shard for each bucket and then stores that index into the Key-Value store the appropriate computed column's hash as its prefix. Setting the number of buckets to twice the number of nodes as are in the cluster is a sensible default.

#### .Gaps in sequential keys

\*\*\*\*

Although sequences provide for guaranteed ascending key values, they cannot guarantee that there will be no missing values in the ordered sequence. For performance reasons, sequence number increments are not within scope of an application transaction. Therefore, if a transaction issues a ROLLBACK after a sequence number is consumed, that number is lost. Furthermore, cached sequence numbers may be lost on a cluster restart.

If an application needs absolutely gap-free numbers (“no missing orders” for instance), then the application will need to implement its own sequence generating logic. Balancing performance and functionality in this case is not trivial – we’ll look at this more in Chapter 6.

\*\*\*\*

#### ===== Ordering of primary key attributes

For a multi-column primary key, the order of attributes has significant implications for performance. You should follow the guidelines for composite indexes which we’ll outline later in the chapter. Generally, the more often a column is used independently of other columns, the more you’ll want to place that column first in the primary key. Likewise, the appearance of primary key columns in ORDER BY clauses should also influence the sequencing.

#### ===== Summary of Primary key performance

We’ve spent a fair bit of time on Primary key mechanisms in CockroachDB for good reason. The effect on scalability can be dramatic, and practices that worked fine in traditional monolithic SQL databases can backfire in CockroachDB.

<<PKPerformance>> shows just how significant these effects can be. In <<PKPerformance>> we see that inserts per second are increased by roughly 100 times when using UUID vs a traditional sequence based approach.

[[PKPerformance]]

.Insert Performance with various Primary key schemes

image::images/PKPerformance.png[PKPerformance]

Chart, bar chart

Description automatically generated

### === Denormalization

One of the outcomes in the development of a normalized data model is the removal of redundancies in data representation. In a well normalized model, a data element is represented in just one place within the model. This eliminates the possibility of inconsistent information within the database.

De-normalization is the process of reintroducing redundant, repeating or otherwise non-normalized structures into the physical model—almost always with the intention of improving performance.

Denormalizing data is a common practice and one that you should not feel guilty about. However, do remember that denormalization has potential downsides:

\* Denormalized data can create inconsistences. These might be transitory (waiting for a materialized view to refresh) or permanent (a derived value is not updated due to a program error). You need to be sure that you have robust mechanisms in place to preserve data integrity.

\* Denomalization has a performance overhead. Although denormalization exists to improve performance, most denormaliaations have a down side. Typically, you improve query performance at the expense of DML performance. Make sure you understand and accept these trade-offs.

The best types of denormalizations are those that can be maintained by the database system automatically and transparently. For instance, in some databases you might be tempted to vertically partition a table so that you can separate frequently accessed and rarely accessed columns. In CockroachDB column families provide this capability without the need to change application code.

#### ==== Replicating columns to avoid joins

JOIN operations magnify the overhead of retrieving data. Over-enthusiastic normalizaiton can often result in even the most trival SELECT operations requiring multi-table joins. For instance, consider the partial schema shown in <<adventurelandAddress>>footnote:[This is part of the Microsoft Adventureland sample database. You can find a version of Adventureland ported to CockroachDB at ???]

[[adventurelandAddress]]

.Overnormalized Address data model

image::images/adventurelandAddress.png[adventurelandAddress]

Diagram

Description automatically generated

To retrieve the address for a person (something we presumably do a lot), we need at five-table join:

[source,sql]

----

**SELECT** p.firstname,p.lastname, a.addressline1,a.city, s2.**name**,c2.**name**

**FROM** person p

**JOIN** businessentity b2 **ON** (p.businessentityid=b2.businessentityid )

**JOIN** businessentityaddress b3 **ON** (b3.businessentityid=b2.businessentityid)

**JOIN** address a **ON** (b3.addressid=a.addressid)

**JOIN** stateprovince s2 **ON** (s2.stateprovinceid=a.stateprovinceid)

**JOIN** countryregion c2 **ON** (c2.countryregioncode=s2.countryregioncode)

**WHERE** p.businessentityid =1

----

Because this join follows primary key values it’s going to be reasonably efficient, but it’s still clearly going to involve five times as many lookup operations as would occur if all the columns where in the base table. So the solution is obvioius: replicate the address directly into the person table. When a persons address changes, you may need to perform two UPDATE (one to ADDRESS, one to PERSON), but you will not have to perfom a five way join everytime you want an address.

#### ==== Summary tables

A summary table typically contains aggregated information that is expensive to compute on the fly. For instance, in the MOVR application we might have a dashboard that shows revenue trends by city based on the following query:

[source, sql]

----

**SELECT** **cast**(r.start\_time **AS** **date**) **AS** ride\_date,u.city,**SUM**(r.revenue)

**FROM** rides r

**JOIN** **users** u **ON** (u.id=r.rider\_id)

**GROUP** **BY** 1,2

----

Since revenue for previous days rarely changes, it’s wasteful to continually reissue this expensive query every time the dashboard requests. Instead, we create a summary table from this data and reload the data at regular intervals (perhaps once an hour).

We can create such a summary table manually, but \*materialized views\* exist for this very purpose. We’d create a materialized view as follows:

[source, sql]

----

**CREATE** **MATERIALIZED** **VIEW** ride\_revenue\_by\_date\_city **AS**

**SELECT** **cast**(r.start\_time **AS** **date**) **AS** ride\_date,u.city,**SUM**(r.revenue)

**FROM** rides r

**JOIN** **users** u **ON** (u.id=r.rider\_id)

**GROUP** **BY** 1,2;

----

The resulting table is far smaller than the source tables and we can comfortably refresh it whenever we like. One of the advantage of a materialized view is that we an also update it whenever we like with the REFRESH command:

[source, sql]

----

**REFRESH** **MATERIALIZED** **VIEW** ride\_revenue\_by\_date\_city;

----

#### ==== Vertical partitioning

Vertical partitioning involves breaking up a table into multiple tables, each of which contains a different set of rows. This is typically done to reduce the amount of work that needs to be done when updating a row, and can also reduce the conflicts that occur when two columns are subject to high concurrent update activity.

For instance, consider an IoT application in which a cities current temperature and air pressure are updated multiple times a second by weather sensor devices across the city:

[source,sql]

----

**CREATE** **TABLE** cityWeather (

city\_id uuid **NOT** **NULL** **PRIMARY** **KEY** **DEFAULT** gen\_random\_uuid(),

city\_name **varchar** **NOT** **NULL**,

currentTemp **float** **NOT** **NULL**,

currentAirPressure **float** **NOT** **NULL**);

----

The temperature values and airPressure readings come from different systems, and we’re concerned that they will cause transaction conflicts when they attempt to change the same row simultaneously. We could partition the table into two tables to avoid this conflict:

[source,sql]

----

**CREATE** **TABLE** cityTemp (

city\_id uuid **NOT** **NULL** **PRIMARY** **KEY** **DEFAULT** gen\_random\_uuid(),

city\_name **varchar** **NOT** **NULL**,

currentTemp **float** **NOT** **NULL**);

**CREATE** **TABLE** cityPressure (

city\_id uuid **NOT** **NULL** **PRIMARY** **KEY** **DEFAULT** gen\_random\_uuid(),

city\_name **varchar** **NOT** **NULL**,

currentTemp **float** **NOT** **NULL**,

currentAirPressure **float** **NOT** **NULL**);

----

However, in CockroachDB \*Column Families\* provide a solution that does not require us to modify our data model. We simply add each measurement to it’s own family:

[source,sql]

----

**CREATE** **TABLE** cityWeather (

city\_id uuid **NOT** **NULL** **PRIMARY** **KEY** **DEFAULT** gen\_random\_uuid(),

city\_name **varchar** **NOT** **NULL**,

currentTemp **float** **NOT** **NULL**,

currentAirPressure **float** **NOT** **NULL**,

**FAMILY** f1 (city\_id,city\_name),

**FAMILY** f2 (currentTemp),

**FAMILY** f3 (currentAirPressure)

);

----

#### ==== Horizontal Partitioning

Horizontal Partitioning (usually just referred to as partitioning) allows a table or index to be comprised of multiple segments.

• Queries can read only partitions that contain relevant data, reducing the number of logical reads required for a particular query. This technique – known as partition elimination – is particularly suitable for queries which read too great a portion of the table to be able to leverage an index, but which still do not need to read the entire table.

• By splitting tables and indexes into multiple segments, parallel processing can be significantly improved, since operations can be performed on partitions concurrenly.

• Deleting old data can sometimes be achieved by deleting old partitions, rather than needing to perform expensive delete operations on large numbers of individual rows.

Partitioning is not available in all CockroachDB editions: The partitioning option is currently available only in the CockroachDB Enterprise Edition.

Partitioning can be performed by range or list:

• Range partitioning allows rows to be allocated to partitions based on contiguous ranges of the partition key. Range partitioning on a time-based column is common since it allows us to purge older data by dropping a partition

• List partitioning allows rows to be allocated to partitions based on nominated lists of values. This is similar but more flexible than range partitioning and allows non-adjacent partition key rows to be stored in the same partition.

If you don’t have an Enterprise license, you can achieve a similar effect by manually creating multiple tables for the ranges or lists of values selected. However, the application logic required to manually manage such a DIY partitioning scheme is cumbersome.

CockroachDB’s multi-region capabilities eliminate one of the possible motivations for partitioning data. \*Regional by row tables\* are effectively transparently partitioned in such a way as to optimize access to those rows from a particular region. In other databases, explicit partitioning might be required to realize this goal. We’ll look at Multi-region topologies in Chapter 10.

#### ==== Repeating groups

The relational model abhors repeating groups since in any such repeating group the attributes are not fully dependent on the primary key alone. For instance, an array element is identified by the primary key and the array index.

However it can be extremely tedious to perform joins to retrieve small groups of data elements of the same type. For instance, in the beginning of the chapter (see <<StudentTestsNormalized>>) we defined a testanswers entity which contains one row for each answer on a test. If a test has 100 questions we need to access 100 rows to see all the results.

The ARRAY type provides an alternative mechanism. CockroachDB arrays are one-dimensional collections of data of the same datatype. For instance, we could store the answers to the test in such an array:

[source,sql]

----

**CREATE** **TABLE** studentTest (

student\_id uuid **NOT** **NULL** ,

test\_id uuid **NOT** **NULL**,

testDate **date** **NOT** **NULL**,

testAnswers **varchar**[] **NOT** **NULL**

);

----

We can set the results in a single update as follows:

[source,sql]

----

**UPDATE** studenttest s

**SET** testAnswers=**array**['a','b','c','d']

**WHERE** student\_id='2fdaadf5-ff3e-45c4-bc92-cc0d566e1ad9'

**AND** test\_id='dca69ac4-6c53-4efb-8c7e-bca9f412e2ee'

----

Now we only need to access a single row to get all the test results, which is a significant reduction in logical IO.

Array datatypes do have some downsides – the query syntax is awkward and it can be hard to perform analytic queries. For instance, finding the sum or average of all elements in an array is not directly supported.

Inverted indexes and allow us to directly efficiently retrieve data from an array datatype: we’ll elaborate on inverted indexes later in the chapter.

[[JSONdocmodels]]

## === JSON document models

The biggest challenge to relational databases over the past decade has come from “document databases” such as MongoDB and Couchbase. These databases store all data in the form of JavaScript Object Notation (JSON) documents. JSON documents are self describing, so there need be no formal implementation of a schema in the DBMS. One simply retrieves the JSON from the database and examines the JSON to decode the structure.

Without entering into any sort of religious debate about the obvious heresy involved in abandoning the relational model in favor of JSON documents, it’s worth pointing out that document databases do offer significant conveniences for the developer:

\* Modern object oriented programming practices involve the modelling of program objects that have a internal structure that allows for inheritance, polymorphism and subclassing, all of which are core features of modern programming languages. These program objects are typically highly denormalized and when stored in an RDBMS must be unpacked. OO programmers used to say “A relational database is like a garage that forces you to take your car apart and store the pieces in little drawers”. In contrast, a document database allows the objects to be stored directly.

\* Modern DevOps practices involve continuous integration in which the entire application can be built directly from code and tested upon any significant change. RDBMS makes this difficult, because a code change and a database change will need to be co-ordinated – ALTER TABLE statement and code commit need to be applied in conjunction. Document databases avoid this issue.

\* Document databases allow you to shoot yourself in the foot without the assistance of a DBA, and most developers want to shoot themselves in the foot whenever they like.

If these document database advantages are attractive to you, then you’ll probably be drawn to the idea of storing all or some of your data in a JSONB datatype.

JSON objects are self-describing and can contain nested JSON objects and arrays. It’s common place in document databases to embed child data within parent objects to avoid the need to perform joins. For instance, a video streaming database might include all of the titles a customer has viewed within a nested array:

[source, javascript]

----

{

"\_id" : ObjectId("5a0518aa5a4e1c8bf9a53761"),

"Address" : "1913 Hanoi Way",

"City" : "Sasebo",

"Country" : "Japan",

"District" : "Nagasaki",

"FirstName" : "MARY",

"LastName" : "Smith",

"Phone" : 886780309,

"dob" : ISODate("1982-02-20T13:00:00Z"),

"views" : [

{

"viewDate" : ISODate("2013-03-02T05:26:17.645Z"),

"filmId" : 611,

"title" : "MUSKETEERS WAIT"

},

{

"viewDate" : ISODate("2015-07-05T20:06:58.891Z"),

"filmId" : 308,

"title" : "FERRIS MOTHER"

},

{

"viewDate" : ISODate("2012-08-04T19:31:51.698Z"),

"filmId" : 159,

"title" : "CLOSER BANG"

}

],

"dateOfBirth" : ISODate("1982-02-20T13:00:00Z")

}

----

However, in CockroachDB this sort of deep nesting is an antipattern. Because the JSON data is stored in-line within the row data within the underlying Key-Value store, CockroachDB recommend that you keep the size of the JSON documents fairly small – under 1MB. Given all the video streaming that has been going on lately it’s quite likely that for some users, the 1MB limit would be exceeded.

You should also keep primary and foreign keys outside of JSONB. You can’t create primary or secondary keys on JSONB data on virtual columns. So at a minimum, your primary and secondary keys should be explicitly defined in your CREATE TABLE statement.

As mentioned in previous chapters, you can create inverted indexes on JSONB columns. These inverted indexes allow you to search for attribute value matches within the JSON object. However, inverted indexes index every attribute in the JSON object and so can have many more entries in the index than rows in the table. A better alternative is to create computed columns on the JSONB attributes and index on those attributes.

So let’s say we have decided to store our customer details as a JSONB document. The basic customer details look like this:

[source,javascript]

----

{ "Address" : "1913 Hanoi Way",

"City" : "Sasebo",

"Country" : "Japan",

"District" : "Nagasaki",

"FirstName" : "Mary",

"LastName" : "Smith",

"Phone" : "886780309",

"dob" : "1982-02-20T13:00:00Z",

"likes": ["Dinosaurs","Dogs","People"] }

----

We know that we want to search on LastName, Firstname, and we also know we want to have a foreign key out to an existing CITIES table. Our CREATE TABLE might look like this:

[source,sql]

----

**CREATE** **TABLE** people (

personId UUID **PRIMARY** **KEY** **NOT** **NULL** **default** gen\_random\_uuid(),

cityId UUID ,

personData **JSONB**,

FirstName **STRING** **AS** (personData->>'FirstName') **VIRTUAL**,

LastName **STRING** **AS** (personData->>'LastName') **VIRTUAL**,

**FOREIGN** **KEY** (cityId) **REFERENCES** cities(cityid),

**INDEX** (LastName,Firstname)

);

----

This design allows us to perform index searches on Last and Firstname, and to retrieve those attributes from the JSON without the awkward JSON dereferencing that we saw in the last chapter. We can, however, add attributes without needing to issue an ALTER TABLE statement, and programmers can load the personData JSON type directly into a JSON object in their application code.

## === Indexes

In CockroachDB, an index is an alternative representation of a subset of table columns that provide a fast access path for retrieving selected values. We looked at the structure of indexes in Chapter 2. You might recall that in CockroachDB, indexes and tables share a common fundamental storage structure. A table is essentially a structure indexed by the primary key. Secondary indexes are tables indexed by the index key, with the payload typically containing the primary key values associated with the secondary key.

Indexes exist to optimize performance and enforce uniqueness. Indexes can generally be added to a system without requiring any change to application code, so compared with other options for physical implementation they are fairly cheap. Creating an optimal set of indexes is arguably the most important factor in ensuring database performance.

### ==== Index selectivity

The \_*selectivity*\_of a column or group of columns is a common measure of the usefulness of an index on those columns. Columns or indexes are selective if they have a large number of unique values and few duplicate values. For instance, a Date\_of\_birth column will be quite selective while a Gender column will be not at all selective.

Selective indexes are more efficient than non-selective indexes because they point more directly to specific values. The CockroachDB optimizer will determine the selectivity of the various indexes available to it, and will generally try and use the most selective index.

### ==== Index break-even point

When you want to look up just a few things in a text book, you go to the index. When you want to assimilate all or most of the content, you bypass the index and go direct to the text. It’s the same with database indexes – we generally only want to use them when we retrieve a relatively small amount of a tables data.

But exactly at want point does a index outperform a full table scan? Unfortunately, there’s no single correct answer. It will depend largely upon the distribution of data in the table. If we are using an index to retrieve data that is highly clustered – appearing in the same ranges, then indexes will be more effective. However, if we are retrieving data that is more or less randomly distributed across the ranges of a table, then full table scans will be more relatively effective.

For instance, in the following example we see the relative elapsed times for retrieving data depending on the relative amount of the table being accessed. In once case the data was highly clustered (based on an ascending primary key). In the other case the data was distributed across the cluster relatively randomly (UUID key).

### ==== Index overhead

Although indexes can dramatically improve query performance, they do reduce the performance of DML. All of a table’s indexes must normally be updated when a row is inserted or deleted and an index must also be amended when an update changes any column which appears in the index.

It is therefore important that all our indexes contribute to query performance , since these indexes will otherwise needlessly degrade DML performance. In particular, you should be especially careful when creating indexes on frequently updated columns. A row can only be inserted or deleted once, but may be updated many times. Indexes on heavily updated columns or on tables that have a very high insert/delete rate will therefore exact a particularly high cost. <<IndexOverhead>> illustrates the overhead on DML that occurs as more indexes are added to a table.

[[IndexOverhead]]

.Index overhead – time to insert 120,000 rows

image::images/IndexOverhead.png[IndexOverhead]

### ==== Composite indexes

A composite index is simply an index created from more than one column. The advantage of a composite key is that it is often more selective than a single key index. The combination of columns will point to a smaller number of rows than indexes composed of the individual columns.

For instance, if we know that we frequently perform searches on FIRSTNAME and LASTNAME, then it makes sense to create an index on both of those columns:

[source,sql]

----

**CREATE** **INDEX** flname\_idx **ON** person (lastname,firstname);

----

Such an index will be far more effective than an index on LASTNAME alone, or separate indexes on LASTNAME and FIRSTNAME. We’ll provide some performance comparisons a bit later in the chapter.

If a composite index could only be used when all of its keys appeared in the WHERE clause, then composite indexes would probably be of pretty limited use. Luckily, a composite index can be used very effectively providing any of the initial or leading columns are used. Leading columns are those that are specified earliest in the index definition.

So for instance, the index we just created can optimize this query:

[source,sql]

----

**SELECT** \* **FROM** person **WHERE** lastname='Wood';

----

But not this query:

[source,sql]

----

**SELECT** \* **FROM** person **WHERE** firstname='John' ;

----

### ==== Covering indexes

A covering index is one that is capable of satisfying a query without reference to the base table. For instance, in the following query:

[source,sql]

----

**SELECT** phonenumber

**FROM** people

**WHERE** lastname='Smith'

**AND** firstname='Samantha'

**AND** state='California' ;

----

An index on LASTNAME, FIRSTNAME, STATE and PHONENUMBER would not only be able to \_find\_ the data requested but would also be able to \_return\_ the PHONENUMBER. Only a single index access would be needed.

In CockroachDB, we can use the STORING clause to store data elements that we might use in the SELECT clause, but not in the WHERE clause. This provides a more efficient mechan for

### ==== Composite and Covering index performance

<<ConcatenatedIndexChart>> illustrates the performance advantages offered by composite and covering indexes. The chart shows the number of KeyValue store options necessary to satisfy the query we introduced above with various indexing options in place.

[[ConcatenatedIndexChart]]

.CompositeIndex Performance

image::images/ConcatenatedIndexChart.png[ConcatenatedIndexChart]

<<ConcatenatedIndexChart>> shows that without indexing, the query requires 18,798 KV operations – we have to read every row in the table. Single indexes on LASTNAME or FIRSTNAME improve the performance somewhat, and having both a LASTNAME and FIRSTNAME index is better than just having either of the two indexes alone.

However, it’s not until we use a composite index that we see truly efficient indexing. Only six KV reads are needed if we have an index on LASTNAME and FIRSTNAME and only two KV operations are needed for an index on LASTNAME, FIRSTNAME and STATE. If we STORE the PHONENUMBER in the index then only a single KV operation is needed.

==== Guidelines for composite indexes

As we’ll see a bit later on, the performance improvements from indexes don’t come without a cost – each index ads overhead to DML operations, so we can’t usually create every possible index that we might like.

The best strategy is to create composite indexes that cover the broadest possible ranges of queries. Since a composite index can be used if any if it’s leading columns are included in a WHERE clause, the ordering of columns in composite indexes is very important.

The following guidelines might help when deciding which indexes to create.

\* Create composite indexes for columns which appear together in the WHERE clause.

\* If columns sometimes appear on their own in a WHERE clause, place them at the start of the index.

\* The more \_selective\_ a column is, the more useful it will be at the leading end of the index.

\* A composite index is more useful if it also supports queries where not all columns are specified. For instance LASTNAME, FIRSTNAME is more useful that FIRSTNAME,LASTNAME because queries against LASTNAME only are more likely to occur than queries against FIRSTNAME only.

### ==== Indexes and null values

In many relational database, NULL values are not included in indexes, and consequently in these systems it is often recommended not to use NULL values if you might want to search for those values. However, in CockroachDB null values are included in indexes and can be found using an index in the normal way.

### ==== Inverted indexes

We discussed inverted indexes in Chapter 2 and earlier in this chapter in the <<JSONdocmodels>> section. An inverted index creates an index for all elements in an array and for all attributes in a JSONB column. As useful and flexible as these inverted indexes may be, they are expensive from a storage and maintenance point of view. We recommend whenever possible creating an computed column from the JSON and indexing on that column. See <<JSONdocmodels>> for an example.

### ==== Partial indexes

A partial index can be created only only a subset of rows in the table. A partial index is created by adding a WHERE clause to the CREATE INDEX statement.

Partial indexes can have a lower maintenance overhead, require less storage in the database, and be more efficient for suitable queries. They are therefore a very useful type of index.

The key limitation with a partial index is that they can only be used when CockroachDB can be certain that the partial index contains all the necessary entries to satisfy the query. In practice this means that a partial index is normally used to optimize queries that contain the same WHERE clause filter condition that was included in the index definition.

### ==== Sort-optimizing indexes

Indexes can be used to optimize ORDER BY operations in certain circumstances. When CockroachDB is asked to return data in sorted order, it must retrieve all the rows to be sorted and perform a sort operation on those rows before returning any of the data. However, if an index exists on the ORDER BY columns, then CockroachDB can read the index and retrieve the data directly in sorted order.

Using the index to retrieve data in sorted order is usually only worthwhile if you are optimizing for the some small number of “top” rows. If you read the entire table in sorted order from the index then you’ll be reading from the base table and the index and the total number of IO operations may be higher than otherwise. However, if you are just getting the first “page” of data or a “top 10” then the index will be much faster, since you never have to read the rest of the table rows at all.

<<IndexSorts>> illustrates the effect of an index to optimize a sort like this:

[source,sql]

----

**SELECT** \*

**FROM** orderdetails

**ORDER** **BY** modifieddate;

----

[[IndexSorts]]

.Indexes and sort performance

image::images/IndexSorts.png[IndexSorts]

When a LIMIT clause was added to the query, then the index reduced execution time from 123ms to just 2ms – a fantastic improvement. However, if we force CockroachDB to use the index to retrieve all rows (something it won’t do by default) then execution time increased from 296ms to 4,000 ms.

### ==== Spatial indexes

A spatial index is a special type of inverted index that supports operations on the GEOMETRY and GEOGRAPHY two-dimensional spatial datatypes.

Spatial indexing is a complex topic and we aim only to introduce you to some key concepts here. For more details consult the CockroachDB documentation setfootnote:[ https://www.cockroachlabs.com/docs/stable/spatial-indexes.html].

To create a spatial index, we add the USING GIST(geom) clause:

[source,sql]

----

CREATE INDEX geom\_idx\_1 ON some\_spatial\_table USING GIST(geom);

----

We can furthermore fine-tune the index using various spatial index tuning parametersfootnote:[ https://www.cockroachlabs.com/docs/latest/spatial-indexes.html#index-tuning-parameters]:

[Source,sql]

----

CREATE INDEX geom\_idx\_1 ON geo\_table1 USING GIST(geom) WITH (s2\_level\_mod=3);

CREATE INDEX geom\_idx\_2 ON geo\_table2 USING GIST(geom) WITH (geometry\_min\_x=0, s2\_max\_level=15)

CREATE INDEX geom\_idx\_3 ON geo\_table3 USING GIST(geom) WITH (s2\_max\_level=10)

CREATE INDEX geom\_idx\_4 ON geo\_table4 USING GIST(geom) WITH (geometry\_min\_x=0, s2\_max\_level=15);

----

We don’t recommend that you change these default tuning parameters lightly; the default values will generally provide the best performance.

### ==== HASH Sharded indexes

Earlier in this chapter (<<HashSharding>>), we showed how in a distributed database monotonically increasing primary keys can lead to “hot spots” in a distributed database. We recommended using Hash Sharded indexes as a way of avoiding such an issue for monotonically increasing primary keys.

These sorts of hot spots don’t occur just in primary keys. Any indexed column that is monotonically increasing might end up with all new values in a single range, and thus creating a scalability and throughput issue.

If you have indexed columns where the value is constantly increasing (timestamps are a good example) and you want to avoid such an insert hostspot, then you should consider hash sharding the index. The syntax is the same as for the primary key example we showed in <<HashSharding>>. For instance, to create a hash sharded index on the MODIFIEDDATE column, we might do the following:

[source,sql]

----

**SET** experimental\_enable\_hash\_sharded\_indexes=**on**;

**CREATE** **INDEX** orderdetails\_hash\_ix

**ON** orderdetails(modifieddate)

**USING** **HASH** **WITH** **BUCKET\_COUNT**=6;

----

Note that while CockroachDB will not usually optimize sort with a hash sharded index, it still might provide good enough performance for a “top 10” type of query. We can force the use of the hash sharded index to perform an ORDER BY using an \*index hint\* (more on this in chapter 8):

[source,sql]

----

**SELECT** \*

**FROM** orderdetails@orderdetails\_hash\_ix

**ORDER** **BY** modifieddate **LIMIT** 10;

----

CockroachDB will retrieve the top 10 from each “bucket” and amalgamate the results on the gateway node. The result might still be a marked improvement over a full scan.

### ==== Measuring Index effectiveness

Obviously, we’d like to be sure that an index is being used to optimize a query and it would also be good to know exactly how much benefit we have achieved. We can do this using the EXPLAIN and EXPLAIN ANALYZE commands.

EXPLAIN reveals to us the CockroachDB optimizer “plan” for an SQL statement

We’ll dig into EXPLAIN in great detail within Chapter 8, but for now let’s just quickly see how they work to explain your query performance.

EXPLAIN reveals the optimizers plan for resolving a query. For instance, if we created an in dex on PEOPLE and wanted to see if the query would use it, we could issue the following command:

[source,sql]

----

EXPLAIN

SELECT phonenumber

FROM people

WHERE lastname='Smith'

AND firstname='Samantha'

AND state='California';

info

--------------------------------------------------------------------------------------

distribution: local

vectorized: true

• filter

│ estimated row count: 63

│ filter: state = 'California'

│

└── • index join

│ estimated row count: 5

│ table: people@primary

│

└── • scan

estimated row count: 5 (0.02% of the table; stats collected 5 hours ago)

table: people@people\_lastfirst\_ix

spans: [/'Smith'/'Samantha' - /'Smith'/'Samantha']

----

We can see that the PEOPLE\_LASTFIRST\_IX will be used to resolve the query.

However, in some cases we might still not be sure if the index improved execution time. If we use EXPLAIN ANALYZE then CockroachDB will execute the operation and will report on the amount of IO and other operations that occurred:

[source,sql]

----

EXPLAIN analyze

SELECT phonenumber

FROM people

WHERE lastname='Smith'

AND firstname='Samantha'

AND state='California';

info

--------------------------------------------------------------------------------------

planning time: 2ms

execution time: 4ms

distribution: local

vectorized: true

rows read from KV: 6 (598 B)

cumulative time spent in KV: 3ms

maximum memory usage: 30 KiB

network usage: 0 B (0 messages)

• filter

│ cluster nodes: n1

│ actual row count: 1

│ estimated row count: 63

│ filter: state = 'California'

│

└── • index join

│ cluster nodes: n1

│ actual row count: 3

│ KV rows read: 3

│ KV bytes read: 430 B

│ estimated row count: 5

│ table: people@primary

│

└── • scan

cluster nodes: n1

actual row count: 3

KV rows read: 3

KV bytes read: 168 B

estimated row count: 5 (0.02% of the table; stats collected 5 hours ago)

table: people@people\_lastfirst\_ix

spans: [/'Smith'/'Samantha' - /'Smith'/'Samantha']

----

EXPLAIN has some additional advanced features that we’ll learn about in Chapter 8. However, you can see how easy it is to use EXPLAIN to simply determine index utilization and effectiveness.

## === Summary

In this chapter we looked at design principles for a CockroachDB database schema. A sound data data model right is an essential foundation for a performant and maintainable CockroachDB database.

Database modelling typically proceeds in two stages: logical modelling followed by physical modelling. The aim of the Logical modelling phase is to identify the data required for application functionality. The physical modelling phase attempts to construct a data model that can meet functional requirements together with performance and availability requirements. The physical model should almost never be a direct copy of the logical model.

Database design for a distributed SQL database like CockroachDB creates some unique challenges when compared with a traditional monolithic database. In particular, primary keys should be constructed so that new rows are distributed equitably across the nodes in the cluster. The UUID datatype can achieve this, but if an ascending primary key is required, then using hash sharded primary key indexes are indicated.

We looked also at indexing choices for a CockroachDB database design. Creating the least number of composite indexes to support common filter conditions is our objective. We may also want to create some indexes to support sort operations.