[[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 objects that support a well-designed CockroachDB application.

Although CockroachDB supports mechanisms for efficiently altering schemas online, schema changes to production applications are nevertheless high impact changes, typically involving coordinated changes to application code and production database configuration. If done poorly, there's the risk of loss of application functionality, availability of performance. Therefore although it's quite possible to alter CockroachDB schemas in production, it is 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 in 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 Data Base Management System (DBMS).

The logical data model typically satisfies 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 modeling 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, it's commonplace to develop a logical model using the language of tables and columns.

.Mea Culpa

\*\*\*\*

Relational data modeling 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 modeling without oversimplification or misrepresentation.   
  
This is a book about CockroachDB, not about the relational theory, and so we have tried 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 modeling 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 a schema since it minimizes redundancy and ambiguity.

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

In the 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 the 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).

A normalized version of this data is shown in <<StudentTestsNormalized>>. Students take tests, and 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 names, 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 throw 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 modeling 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. There's a lot of wasted effort in modeling 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 that 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 the logical modeling point of view, there are some factors to consider.

The third normal form requires that each relation have a primary key. Yet, it does not specify whether that key should be artificial or synthetic. A \_*natural key\_* is one constructed from unique attributes that normally occur within the entity. An \_*artificial key\_* is one that 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 primary key is changed. Furthermore, in CockroachDB, the use of an artificial key provides us with methods for ensuring an 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. Two 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 data warehousing database, and so these models are not typical of a CockroachDB deployment.

\* \*Time-Series\* designs in which the time of origin of data is part of each data element's key and in which data accumulates primarily as continual inserts. We'll briefly consider some of the considerations for time-series 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 characteristics 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 tables 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 tuples. 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 depends 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 its 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 (PostgreSQL, for instance), NULL values are not included in indexes. However, CockroachDB does store 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 constrain 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 numbers 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 – such as ARRAYS and JSON - 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 table's 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 hotspot and limit the insert throughput for the cluster. This becomes particularly significant as your cluster grows: adding new nodes to a cluster may fail to result in higher throughput.

The same phenomenon can be encountered in a time-series 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 gapsfootnote:[Sequence generators are provided mainly for compatibility with other databases and are not recommended for most CockroachDB applications]. 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 a continuously increasing value, then a UUID (Universally Unique IDentifier) primary key is the recommended solution.

A UUID is a value that 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 large gaps will occur.

You can change the behavior of SERIAL to a more PostgreSQL compatible behavior using the session variable +serial\_normalization+. However, as with PostgreSQL, gaps in sequence numbers generated in this manner may still occur and the performance overhead is significant. The CockroachDB team recommends against using 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 hotspots. 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 its 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 the 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 the output by primary key. For instance, with a traditional primary key, the query below would be resolved efficiently by a range scan of the primary key index:

[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 the scope of an application transaction. Therefore, if a transaction issues a ROLLBACK after a sequence number is consumed, that number is lost. To achive anything like scalable distributed performance, you would using the CACHE option to give each node its own unique set of ranges – which will result in keys being inserted out of order across nodes. 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 insert throughput is severely diminished when SERIAL or Sequence generated keys are used. UUIDs are preferred, but if you need ascending primary keys, then a hash sharded primary key index is recommended.

[[PKPerformance]]

.Insert performance with various Primary key schemes

image::images/PKPerformance.png[PKPerformance]

The data in <<PKPerformance>> came from a 9-node CockroachCloud cluster. The performance penalty from ascending primary keys is proportional to the size of the cluster: the more nodes in the cluster the larger the relative penalty. So your milage may vary depending on your cluster size.

It is possible to greatly improve the performance of sequenced by creating them with the CACHE option. This avoids the blocking wait involved in acquiring the “next” sequence number. However, in a distributed system like CockroachDB, using CACHE defeats the purpose of the sequence generator. Because each node in the cluster has its own cache, sequence numbers will be generated out of order across the cluster as a whole.

### ==== Foreign key constraints

Foreign key constraints help ensure data integrity and provide internal documentation of the data model, which can be leveraged by query generators and diagramming tools. However, during Data Manipulation Language (DML) operations – particularly inserts, the validity of the foreign key must be checked by performing primary key lookups on the referenced table. These lookups can significantly increase the overhead of the operations and reduce throughput.

For the table that includes the Foreign key constraint, this shows up mostly in insert performance since it is somewhat unusual for a foreign key to be updated, and the foreign key references do not need to be validated during deletes.

For the tables that are referenced within in the foreign key constraint (e.g. the "parent" table), the overhead is felt most critically during deletes, where all child tables must be checked for "dangling" references.

The ON DELETE CASCADE clause of a CONTRAINT definition will automatically delete any child rows during the delete of a parent row. ON UPDATE CASCADE has a similar effect when a primary key is updated (which in most applications is a rare event).

Because of the overhead of foreign key constraints, it is not unusual for them to be removed in a production system. They may be left enabled only in test and development environments to catch any data anomalies.

Dropping foreign key constraints can potentially lead to data anomalies, but in some circumstances, the need to achieve maximum throughput will take priority.

### === 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 inconsistencies. 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.

\* Denormalization has a performance overhead. Although denormalization exists to improve performance, most denormalizations have overhead. 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 normalization can often result in even the most trivial 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 a 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 were in the base table. So the solution is obvious: replicate the address directly into the person table. When a person's address changes, you may need to perform two UPDATE (one to ADDRESS, one to PERSON), but you will not have to perform a five-way join every time you want an address.

As with many design decisions, there are many options between the two extreems. If you want to preserve the multiple-address-per-person design of << adventurelandAddress>> you could still consider collapsing the STATE and COUNTRY tables into the ADDRESS table to reduce the number of tables involved in the join.

#### ==== 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 it. 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 manually refresh it whenever we like. One of the advantages of a materialized view is that we can 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 completed 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 city's 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, CockroachDB \*Column Families\* provide a solution that does not require us to modify our data model. We simply add each measurement to its 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. Some examples are:

• Queries can read only the 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 that read too great a portion of the table to be able to leverage an index, but 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 concurrently.

• 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 example, at the beginning of the chapter (see <<StudentTestsNormalized>>), we defined a TESTANSWERS entity that 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 modeling of program objects that have an 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. Object-oriented 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 coordinated – ALTER TABLE statement and code commits need to be synchronously applied. 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 commonplace 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")

}

----

#### ==== JSON document anti-patterns

In CockroachDB, implementing one to many relationships in JSON documents is inadvisable. Because the JSON data is stored in-line within the row data within the underlying Key-Value store, CockroachDB recommends that you keep the size of the JSON documents fairly small – under 1MB.

For instance, in the video streaming JSON model that we introduced in the previous section we embedded all the films that a customer had viewed within a JSON array. Given all the video streaming that has been going on lately, it's quite likely that, at least 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 or on virtual columns derived from JSONB data. So at a minimum, your primary and foreign keys should be explicitly defined in your CREATE TABLE statement.

#### ==== Indexing JSON attributes

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 create an 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 LastName and FirstName, and to retrieve those attributes from the JSONB without the awkward JSON dereferencing syntax that we introduced 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.

#### ==== Using JSON or Arrays to avoid joins

We said before that "one to many" relationships should not be modeled in JSONB columns. The same is true of ARRAY columns. We want to avoid storing more data in these columns that the key-value store can process in a single operation.

However, it can be quite effective to model "one to few" relationships in JSONB or ARRAYS. For instance, consider the Students Tests schema we modeled way back in <<StudentTestsNormalized>>. We know that there can be at most only a couple of hundred questions in a test. In the normalized solution, we always have to join tables in order to get the answers for a specific test:

[source,sql]

----

**SELECT** s.student\_id,s.test\_id,question\_no,questionanswer

**FROM** studenttest s

**JOIN** testanswers t **on**(t.student\_id=s.student\_id **AND** t.test\_id=s.test\_id)

**WHERE** s.student\_id=**:student\_id**

**AND** s.test\_id=**:test\_id**

**----**

We know that there can only be a couple of hundred questions in a test, and the answers can at most be only a few kilobytes. So being sure that the 1MB limit will not be exceeded, we could collapse the test answers into a JSON document:

[source, sql]

----

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

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

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

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

answers **JSONB**

);

**INSERT** **INTO** studentTest (student\_id,test\_id,testDate,answers)

**VALUES** ('2fdaadf5-ff3e-45c4-bc92-cc0d566e1ad9',

'dca69ac4-6c53-4efb-8c7e-bca9f412e2ee',

**now**(),

'{"answers":[

{"questionNumber":1,"Answer":5},

{"questionNumber":2,"Answer":25},

{"questionNumber":3,"Answer":58},

{"questionNumber":4,"Answer":3425},

{"questionNumber":5,"Answer":432},

{"questionNumber":6,"Answer":0},

{"questionNumber":7,"Answer":673}

]}');

----

We can also use ARRAYS or JSON repeating groups to avoid joins where there is a "many to many" relationship between two tables. For instance, consider the relationship between students and classes as shown in <<StudentClasses>>.

[[StudentClasses]]

.Students and Classes

image::images/StudentClasses.png[StudentClasses]

Diagram

Description automatically generated

Whenever we want to get a list of a student's classes, we are forced to perform a three-way join:

[source, sql]

----

**SELECT** class\_name **FROM** students

**JOIN** studentClasses **USING**(student\_id)

**JOIN** classes **USING**(class\_id)

**WHERE** student\_id='000390a6-4e1d-4bc1-aad7-66b645131d54';

----

The table STUDENTCLASSES exists only to join the STUDENTS and CLASSES tables – it contains no independent information.

We could instead store foreign keys for all the classes in an ARRAY type:

[source, sql]

----

**ALTER** **TABLE** students **ADD** **COLUMN** classes UUID[];

**UPDATE** students s **SET** classes= (

**SELECT** **array\_agg**(class\_id)

**FROM** studentClasses sc

**WHERE** s.student\_id=sc.student\_id);

----

The ARRAY\_AGG function takes all the columns in a result set and converts them to an array. So in the above example, we copied all the CLASS\_ID values for each student into the CLASSES array.

Now when we want to get the classes for a particular student we can "unnest" the array and join the resulting CLASS\_ID values directly to the CLASSES table:

[source, sql]

----

**WITH** students\_classes **AS** (

**SELECT** student\_id , **UNNEST**(classes) class\_id

**FROM** students

)

**SELECT** class\_name **FROM** classes

**JOIN** students\_classes **USING**(class\_id)

**WHERE** student\_id='000390a6-4e1d-4bc1-aad7-66b645131d54';

----

This might all seem to be a bit convoluted, but in a high performance workload, reducing a three-way join to a two-way join might be necessary to achieve performance objectives, even if it complicates application code a little.

Of course, we can avoid the joins altogether if we embed all the information about a student's classes into a JSONB column, just as we did earlier for test answers. However, the array solution above doesn't duplicate any information from the CLASSES table into the STUDENTS table, so if the name of a class changes, we only have one update to perform.

Do bear in mind that by embedding foreign keys in this way, we lose the capability of defining FOREIGN KEY constraints and create some opportunities for data inconsistencies.

## === Indexes

An index is a database object that provides a fast-path to specific data within a table.

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 base table is essentially a relation indexed by the primary key. Secondary indexes are also relations but are indexed by the index key, with the payload typically containing the primary key values associated with that 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 easy to modify. Creating an optimal set of indexes is one of the most important factors in ensuring optimal 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 textbook, you go to the index. When you want to assimilate all or most of the content, you bypass the index and go directly to the text. It's the same with database indexes – we generally only want to use them when we are retrieving a relatively small amount of a table's data.

A non-covering index – one that includes the filter conditions but not all the columns in the SELECT list – is generally effective only when we are retrieving a small percentage of a table’s data. Beyond that, the overhead of going backwards and forwards from index to base table will be slower than simply reading all the rows in the table.

However, when we create a covering index using the STORING clause, the situation is very different, In this case, the index can outperform the table access even if very large proportions of data are accessed.

The optimizer will attempt to determine how much data is being accessed and choose an index or a table scan as appropriate. However, you don’t want to create an index that will never be used, so it’s important to understand the “cut off” point between an index and a table scan.

For instance, let’s say we have timeseries data where a measurement (say a temperature) was recorded every minute over the past year. The application is often asked to determine the average measurement over some recent time period. The query looks something like this:

[source, sql]

----

SELECT AVG(measurement)

FROM timeseries\_data

WHERE measurement\_timestamp > ((date '20220101')- INTERVAL '$dayFilter days'

----

The variable +${dayFilter}+ can take low or high values. We can create a non-covering index on the table as follows:

[source,sql]

----

CREATE INDEX timeseries\_timestamp\_i1

ON timeseries\_data(measurement\_timestamp)

----

However, this index will only be effective when the number of days selected is very small – probably less than a week. Alternatively, we could create a covering index that includes the MEASUREMENT column:

[source,sql]

----

CREATE INDEX timeseries\_covering

ON timeseries\_data(measurement\_timestamp) STORING (measurement)

----

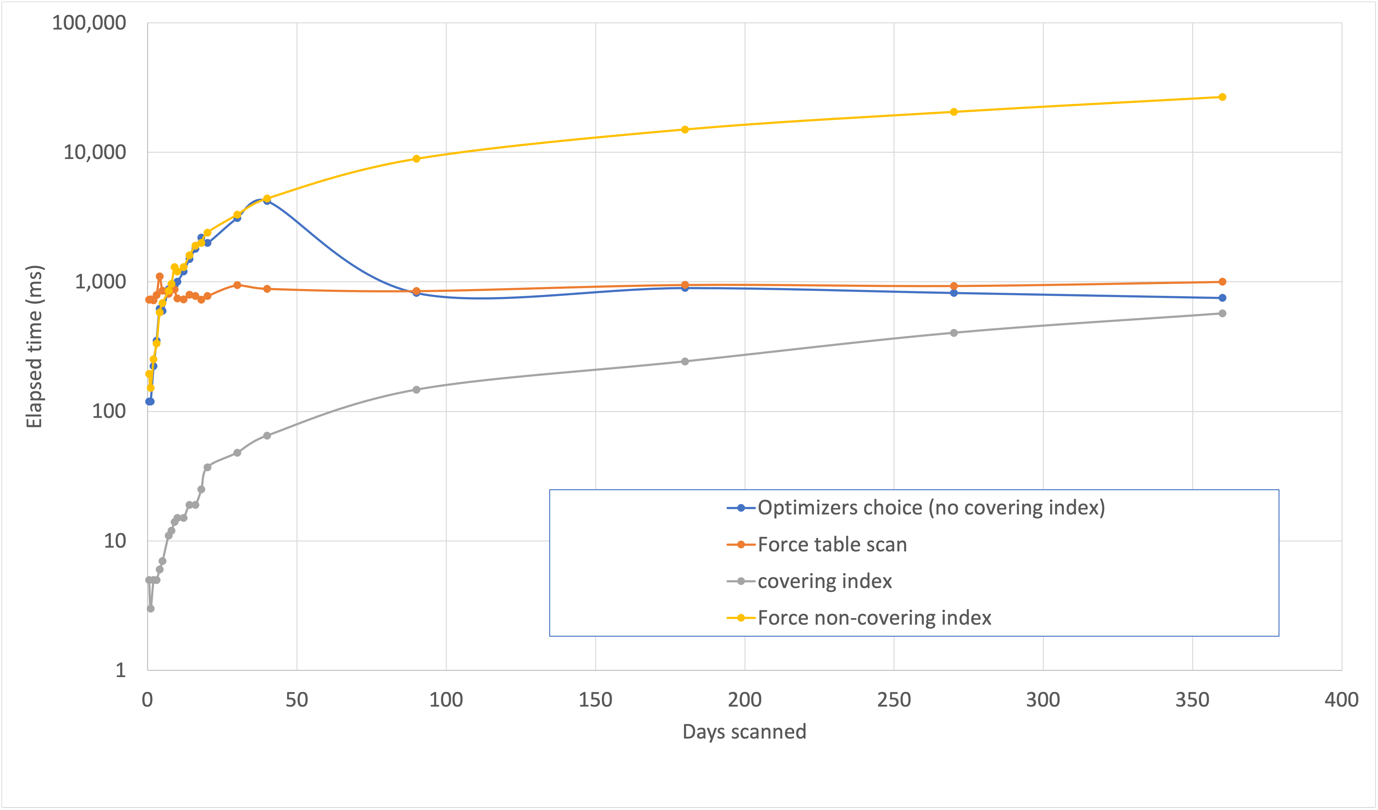
This index can be used effectively for any spans of data – from one day to the entire year’s data.

<<scanvsindex>> compares the performance of an index scan with a table scan, against the number of days of information being retrieved. The table scan must do the same amount of work regardless of the amount of data being processed, while an index scan increases in overhead the larger the amount of data being processed. For a non-covering index, a table scan is better if there’s more than about a week's worth of data retrieved (about 2% of the total data). However, a covering index can perform well even if we are retrieving …..

[[scanvsindex]]

.Comparison of table scan vs index scan performance

image::images/scanvsindex.png[scanvsindex]



There are a few lessons to be drawn from <<scanvsindex>>:

\* The optimizer switches from an index scan to table scan when the amount of data hits about 10-15% of total. The optimizer is a sophisticated piece of software but it isn’t magic, and it can’t always work out which access path is better. In some circumstances, creating an non-covering index will actually degrade performance.

\* A covering index is far superior in performance to a non-covering index, and can be used effectively even if all or most of the table is being accessed. Whenever possible, use a covering index.

\* Remember that in CockroachDB, indexes and tables have the same storage format: a covering index is not just a fast access mechanism – it’s also a compact representation of a subset of table columns that can be scanned far faster than the base table.

We’ll come back to index performance and tuning queries in Chapter 8.

### ==== 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 for composite indexes 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 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 on (LASTNAME, FIRSTNAME) that 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 – and no base table read - 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 mechanism for implementing a covering index. So for the above query, this index would be optimal:

[source,sql]

----

CREATE INDEX people\_lastfirststatephone\_ix ON people

(lastname,firstname,state)

STORING (phonenumber);

----

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

<<ConcatenatedIndexChart>> illustrates the performance advantages offered by composite and covering indexes. The chart shows the number of KeyValue (KV) store options necessary to satisfy this query under various indexing scenarios:

[source,sql]

----

**SELECT** phonenumber

**FROM** people

**WHERE** lastname='Smith'

**AND** firstname='Samantha'

**AND** state='California' ;

----

[[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 saw earlier, 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 of its 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 that 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 than FIRSTNAME,LASTNAME because queries against LASTNAME only are more likely to occur than queries against FIRSTNAME only.

### ==== Indexes and null values

In many relational databases, 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 a computed column from the JSONB attribute concerned and indexing on that column. See <<JSONdocmodels>> for an example.

### ==== Partial indexes

A partial index can be created on 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 it 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 rows directly from the index in sorted order.

Using the index to retrieve data in sorted order is usually only worthwhile if you are optimizing for some small number of "top" rows. If you read the entire table in sorted order from the index, then you'll be reading all the index entries as well as all the table entries, and the total number of IO operations will be excessive. However, if you are just getting the first "page" of data or a "top ten" 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, 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 hotspot, 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 a 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

Having created an index, we'd like to be sure that an index is being used to optimize our query and discover 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 detail within Chapter 8, but for now, let's just quickly see how they work to explain your query performance.

EXPLAIN reveals the optimizer's plan for resolving a query. For instance, if we created an index 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;)

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)

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 model right is an essential foundation for a performant and maintainable CockroachDB database.

Database modeling typically proceeds in two stages: logical modeling followed by physical modeling. The aim of the Logical modeling phase is to identify the data required for application functionality. The physical modeling 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 also looked 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.

Now that we've learned how to create a data model, we are in a position to start writing application code. We've already introduced CockroachDB SQL: in the next chapter, we'll see how to use CockroachDB SQL in application development frameworks.