# Chapter 5 – Schema Design

A sound data model is the foundation of a 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.

Although CockroachDB supports mechanisms for efficiently altering schemas online, schema changes are nevertheless high impact changes. The cost of a schema change increases as an application progresses from design through to production implementation, so optimizing an applications data model is typically a task that is best undertaken early.

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 practicialities 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, keys and other characteristics of the database.

### TERMINOLOGY and concepts

* Relations and tables, tuples and rows, etc.
* Primary keys, foreign keys
* Mea Culpa, like everyone else, we often mix terms in order to be understood.

### 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 starting point.

Relational theory defines multiple “levels” of normalization. The third normal form is generally regarded as the most appropriate level of normalization for most models.

#### Definition of third normal form

#### example of third normal form model

#### BOYCE-CODD NORMAL FORM

### primary key choices

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.

Without entering into the wider debate of the merits of natural keys from a data modeling and design perspective, it is worth considering the merits of artificial keys from a performance perspective. There is little doubt that artificial keys generally result in superior performance.

In a traditional monolithic database, an artificial primary key is generally created from a auto-incrementic number via the AUTOINCREMENT datatype or a \*sequence generator\*. However, in a distributed database like CockroachDB this pattern has significant disadvantages….

Ordering of attributes in a primary key

### 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. These include:

* The Data warehousing Star and Snowflake Schemas
* Timeseries designs
* Geo-spaatial

## 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 columsn from the second table into the first to avoid the join. Other considerations might be database engine specific. 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.

### Logical to Physical

* Entities to Tables
* Choice of Primary Keys
  + Example of mapping subtypes to multiple tables
* Attributes to coluns
  + Datatype selection
  + Use of Nulls

### Primary key selection

#### Synthetic vs natural

#### Primary keys in a distributed database

#### AntiPattern: sequences

#### The SERIAL datatype

#### The UUID Datatype

#### Prefixing with non-monotonically increasing column

#### Hash sharded indexes

#### Ordering of attributes

##### For filters

##### for joins

### Datatype choices

#### Varchar vs String

#### TIMEZ vs TIMESTAMPZ

#### GeoSpatial data

#### Bytes

#### Collations

#### NULLS

### Denormalizations

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 model.

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.

#### Risks of denormalization

Downsides of denormalization:

* on maintainability
* on performance

#### Types of denomalization

##### REplicated column values

##### Rollups

##### vertical partitioning

##### Horizontal partitioning

##### Materialized views

##### multi-reg

#### Arrays

##### As child data

##### To avoid join table

### Multi-region models

The data model for a multi-regional implementation will include a location specific column as a leading column within the primary key and in as many secondary indexes as possible.

### JSON document models

The biggest challenge to relational databases in today’s database market 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, it’s worth pointing out that document databases do offer significant convienience 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 conjuction. 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.

#### foreign keys in JSONB

#### Indexing options for JSONB

##### Downsides of inverted indexes.

\* Range queries

\* index size

\* Mainteance cost

##### Using computed columns

###### For foreign keys?

#### LINKING VS EMBEDDING IN JSONB

#### Limits of JSONB

##### Maximum document sizes

##### Unbounded Arrays in JSONB

### Column families

Column families allow you to store groups of columns separately in physical storage. Consider multiple column families in the following circumstances:

### Exploiting Computed Columns

#### As an alternative to views

#### For more effective indexing

#### VIRTUAL vs StORED

## Indexing

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.

### When and what to index

* Cardinality, selectivity, scan vs index efficiency
* Concatenated indexes vs index merges
* Indexes and null values
* Covering indexes
* Inverted indexes
* Partial indexes
* Sort-optimizing indexes
* Spatial indexes

### Index overhead

* Effect on inserts, updates and deletes

### Index hotspots

### HASH Sharded indexes

#### Creating

#### updating when node topology changes

### Measuring Index effectiveness

* Explain
* Detecting unused indexes?