1 Overview

To support working with relational data CockroachDB provides a SQL layer.

With the support of the SQL API developers are able to run familiar SQL queries. When running those statements the cluster gets an request, which initiates to write or read the particular data ultimately as key-value (KV) pairs into the storage layer. Therefore the layer converts beneath the surface these SQL statement into a load of KV operations, which get processed by the next layer called transaction layer.

Developers are even capable of choosing to which node a request should be sent. The receiving node can either process the request or it acts just as so-called gateway node. This is possible because CockroachDB's nodes behave symmetrically. Thus the use of load balancers has ideal prerequisites.

2 Components

2.1 Relational Structure

One of the headline features of CockroachDB is the relational structured data storage. The data is organized in rows, columns, tables and even multiple databases. This ensures that relational key features like constraints are supported. Application developers don't need to built data validation into the application logic because the database already guarantees consistent data.

2.2 SQL API

To keep CockroachDB simple to use it implements a SQL API. The API makes use of a large portion of the ANSI SQL standard, but as usual in other databases too, some SQL features are differently implemented and not all features are supported.
Besides the basic CRUD operations CockroackDB offers constraints like 'primary key', 'foreign key', 'unique', 'not null' and many more.

2.2.1 Transaction

Even transactions are available to keep the database consistent. Until now CockroachDB has tried many approaches to guarantee ACID transactions. The previous concept was a serializable, lockless and distributed strategy. The problem with the lockless design was that it couldn't sufficiently guarantee consistency. Therefore this design was discarded. To provide serializable isolation it's now a more locking and lock-like structure. More information to this topic is in the transaction layer chapter.

direct link to transaction layer

2.2.2 Joins

As implementing efficient join algorithms is a big challenge in distributed SQL CockroachDB released since 2016 constantly new and improved implementations. CockroachDB currently supports 'merge joins', 'hash joins' and 'lookup joins' algorithms in production and additionally 'lateral joins' (a type of correlated subquery) since version 20.1 alpha. The launch of the hash join algorithm in 2017 reduced the asymptotic time complexity from previously O(N*M) to O(N+M). Later on the merge join algorithm got vectorized because the traditional sort-merge-join

> fails to take into account the overhead of integrating this algorithm into a system. Due to the nature of how data is stored and the interfaces used to process the data within the database, there are a few key factors that make this algorithm behave slower in practice than in theory.
>
> In a database, each row of data is encoded in a format that is optimized for a specific purpose, which may not necessarily be the same purpose across all cases. To be specific, the current state of the row by row engine stores each column of every row in something called a datum. To be able to compare this datum to another datum, there is extra overhead in each call to retrieve the value, as first the type of that value has to be determined after which the value has to be decoded. Now if this process happens for every single column of every single row, it is clear that isn't the most efficient use of processor time.
>
> Instead, it would make sense if the type check and conversion could happen once per column, since every value in the column has the same type. This is called vectorized execution, since the idea is to operate on values a column at a time. (https://www.cockroachlabs.com/blog/vecto
> the-merge-joiner-in-cockroachdb/)

Supported Joins

Inner joins

```
<table expr> [ INNER ] JOIN <table expr> ON <val expr>
<table expr> [ INNER ] JOIN <table expr> USING(<colname>, <colname>,
...)
<table expr> NATURAL [ INNER ] JOIN <table expr>
<table expr> CROSS JOIN <table expr>
```

Left outer joins

```
<table expr> LEFT [ OUTER ] JOIN <table expr> ON <val expr>
<table expr> LEFT [ OUTER ] JOIN <table expr> USING(<colname>, <colname>,
...)
<table expr> NATURAL LEFT [ OUTER ] JOIN <table expr>
```

Right outer joins

```
<table expr> RIGHT [ OUTER ] JOIN <table expr> ON <val expr>
<table expr> RIGHT [ OUTER ] JOIN <table expr> USING(<colname>, <colname>,
...)
<table expr> NATURAL RIGHT [ OUTER ] JOIN <table expr>
```

Full outer joins

```
<table expr> FULL [ OUTER ] JOIN <table expr> ON <val expr>
<table expr> FULL [ OUTER ] JOIN <table expr> USING(<colname>, <colname>,
...)
<table expr> NATURAL FULL [ OUTER ] JOIN <table expr>
```

Merge Joins

Hash Joins

Lookup Joins The whole table of supported features is here.

2.3 SQL Parser, Planner, Ececutor