1 Overview

In CockroachDB ACID transactions are fully supported in order to provide consistency. To ensure this every statement in CockroachDB is basically a transaction because by default 'autocommit mode' is enabled. Furthermore there is an own protocol to handle those transactions cross-range and cross-table. 'Parallel commit' is the name of the atomic commit protocol and in combination with transaction pipelining it reduces significantly the latency of such operations.

2 Writing and reading

To provide consistency in distributed transactions CockroachDB creates write intents and a transaction record.

Write intents provide the uncommited transaction writes to other tables/ranges like the multi-version concurrency control (MVCC) protocol and additionally a pointer to the transaction record. If there is a write intent which handles the same key there's a transaction conflict. Those write intents can be encountered by future operations while looking for MVCC values. The operation knows how to handle the write intent value by checking its transaction record. If there is no transaction record it decides based on the timestamp the validity of the write intent.

Transaction record stores the state of the transaction (PENDING, STAGING, COMMITED, ABORTED) and an array of writes from this transaction. Thus other transactions can check if this one is already commited and therefore in an consistent state.

3 Parallel commits

Depending on the state of the transaction record CockroachDB commits or restart the transaction. If the transaction record says it is in 'STAGING' state the transaction gets checked on whether it is successfully replicated across the cluster. If that is true the cleanup phase begins by changing the transaction record state to 'COMMITED', resolving the write intents to MVCC values and deleting afterwards these writes.

4 Isolation levels

Even though there are many ANSI transaction levels, CockroachDB decided to only support the 'SERIALIZABLE' level. This decision is based on a research paper from 2017, named 'ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications' which shows the vulnerability of concurrent ACID transactions.
Previously the 'SNAPSHOT' level was also supported but the security aspect was valued higher to transaction throughput.

5 Transaction conflicts

CockroachDB handles a transaction conflict by the metrics priority and transaction record.

When there are for example two write intents which use the same key firstly the priority of there transaction gets compared. The one with the lower one gets in this case aborted. When the priorities don't differ the transaction which has either no transaction record or wasn't heartbeated within the transaction liveness threshold gets aborted. If there is still no decision the write intent which encountered the other one enters the TxnWaitQueue.

Heartbeat timestamps are there to evaluate if a transaction is abandoned. Active transactions are responsible for periodically updating it on its central transaction record.

6 TxnWaitQueue

As already mentioned in Transaction conflicts if there is a write/write conflict the last option is to push one transaction into the TxnWaitQueue. It is a simple map which keeps track of blocking transaction IDs and their blocked companions.

```
txnA -> txn1, txn2
txnB -> txn3, txn4, txn5
```

The TxnWaitQueue gets manipulated only by the leaseholder which holds the transaction records. The queue receives a signal when a transaction got comitted or aborted. Based on this signal it let's particular transactions execute again.

direct link to leaseholder

Deadlocks are handled by choosing and aborting a random affected transaction.

7 Transaction pipelining

To reduce latency when making transaction with multiple writes CockroachDB is pipelining those transactions. Pipelining in this context means all of the affected ranges receive the write intents and write it to the disc in parallel when the transaction got commited.
More specifically the gateway node sends the statement to the leaseholders at first and then the leaseholders convert those statements into write intents. After spreading the write intents within the particular range and receiving confirmations from all nodes the transaction is considered as commited.

In the example below the resulting write intents are because of their UUIDs probably in different ranges. This would be an ideal case because it would be fully parallelized.

```
BEGIN;
INSERT into kv (key, value) VALUES ('apple', 'red');
INSERT into kv (key, value) VALUES ('banana', 'yellow');
INSERT into kv (key, value) VALUES ('orange', 'orange');
COMMIT;
```