ArangoDB

One Engine, one Query Language.
Multiple Data Models.

json { } + r a p h g + key value

Running complex queries in a distributed system

About me

¡Hola, me llamo Jan!

I am programming since the mid-80s, moving from 8 bit computers, assembly language and other nice things to C++

As a former user of other databases I wrote a lot of SQL queries, especially for MySQL, Oracle and Microsoft products

Since January 2012 I am working for ArangoDB Inc. in Colonia, DE

ArangoDB Inc. is a database vendor, producing the distributed, multi-model NoSQL database named “ArangoDB”

About ArangoDB

ArangoDB is a multi-model NoSQL database

Runs in single-server mode or as distributed database

Allows working with JSON documents, graphs and key / values

Provides a rich query language (AQL) for getting your data back

Database data can be made accessible via custom REST APIs

Agenda

- Relational databases
- NoSQL databases
- Distributed ACID transactions
- Databases with distributed ACID transactions
- Q & A

Disclaimer

My view may be biased

Due to the complexity of the topic, I will only include a few selected databases (no Hadoop, sorry!)

I may be generalizing too much

And I may go into details too much – stop me if it gets boring!

Relational databases

Relational databases

Relational databases originated in the 1970s, now very mature

SQL (structured query language) came up as a standard means for querying and administrating relational databases

The vast majority of currently running databases are still relational

Examples: MySQL, ProgreSQL, SQLite, Oracle, SQL Server

ACID transactions – properties

Most relational databases provide ACID transactions:

- Atomicity:
  transactions either fully commit or fully abort
- Consistency:
  the database transitions from one valid state to another
- Isolation:
  concurrent transactions do not interfere with each other
- Durability:
  data modifications applied by transactions do not disappear

Transaction isolation levels

To produce consistent results, operations that access the same data need to be put into some order

In SQL this is controlled with the transaction isolation levels

The strongest isolation level in SQL is “serializability”, which guarantees that there are no amolies
A consequence of “serializability” is that any transactions that affect each other must be executed serially or be aborted (or retried)

ACID transactions with high isolation levels relieve developers from handling concurrency and consistency issues in their applications

All good?

Wait…

What if scalability and high availability are required?

Relational databases – scalability

Most relational databases are designed to run well on a single server

ACID transactions are really hard to scale out

In practice, scaling out was (and still is) not a good option with most relational databases

This leaves scaling up as a way to get more database capacity, but the scale-up path will end quickly

Relational databases – custom sharding

To scale out anyway, many power users implemented custom sharding on top, in their client application layer

Ebay, Yahoo, Google, Facebook etc. all did this

That approach sacrifices the relational database’s ACID guarantees for any cross-shard operations/transactions

Relational databases – availability

Making a relational database resilient to server failure(s) can be challenging

Client application write operations are normally sent to the database master, which then replicates these writes to one or multiple slave servers:

- Asynchronous replication:
  slave may lag behind, reads from slave may produce stale data, potential data loss in case of master failure
- Synchronous replication:
  no slave lag, but puts an extra delay on all write operations

NoSQL databases

NoSQL databases – background

As relational databases failed to meet the requirements of big data companies, they began implementing their own databases

These databases – termed “NoSQL” later on – addressed the two major issues that relational databases had: scalability and availability

NoSQL databases will take care of automatically distributing data, and also handle the failover automatically

Client applications do not need to implement client-side sharding anymore

Sharding

Most NoSQL databases scale horizontally by dividing the entire dataset into “shards”

The following two ways of sharding are used in practice to map the keys of the dataset unambiguously to shards:

- hash sharding
- range sharding

The variant of sharding in use determines the types of queries that can be supported efficiently (without consulting all shards)

Hash sharding

A hash function is applied on each key

The hash value is reduced to a bounded value (e.g. number of shards) by a modulo operation:

```
shard(key) = hash(key) % number of shards
```

Very simple to implement, low overhead, normally balances well

Finding the shard for a given key is straightforward

Other operations (e.g. range lookups) need to check all shards

Range sharding

The dataset is split into sorted key ranges

Each range has a lower and upper bound of the keys it contains

To find the shard for a given key or range, a lookup table needs to be consulted:

```
shards(range) = lookup(range)
```

Needs an extra indirection via a lookup table – to remain balanced, ranges need to be split or merged when getting too big or too small

NoSQL databases – design choices

NoSQL implementations focused on the operations that are easy to scale out and make highly available

Hard-to-scale features – especially ACID transactions – were omitted intentionally in NoSQL databases

These design choices simplified the implementations and made these databases scale well

NoSQL databases – guarantees

Most NoSQL databases provide strong guarantees only for single-key operations

Guarantees are weak or don’t exist for multi-key operations

Transactions from client applications normally contain operations that affect multiple keys

These will be executed as if they were single, unrelated operations

Client applications need to work around the lack of transactions!

NoSQL databases – issues

Multi-key operations in NoSQL databases are normally

- non-atomic:
  no commit or rollback commands – they can fail somewhere in the middle without the database reverting them
- non-consistent:
  database may be inconsistent temporarily (eventual consistency), client applications are supposed to resolve conflicts or clean up
- non-isolated:
  operations will see modifications from other parallel write operations, last write wins

Multi-key example

An example for a multi-key operation is to transfer a value from one record (key) to another

Two records (keys) participate in this operation:

- key “account_1”: start value: 25
- key “account_2”: start value: 50

We will transfer an amount of 10 from one account to the other

As it is just a transfer, the sum of values from both accounts should not change

Multi-key example (all good)

(account_2 → account_1)

Read account_1:	*25*	
Write account_1:	*25 + 10*	*→ 35*
Read account_2:	*50*	
Write account_2:	*50 – 10*	*→ 40*

Start state: account_1: 25, account_2: 50, sum: 75
End state: account_1: 35, account_2: 40, sum: 75

Transfer succeeded!

Multi-key example (with crash)

(account_2 → account_1)

Read account_1: 25
Write account_1: 25 + 10 → 35

(server crash here, and restart)

Start state: account_1: 25, account_2: 50, sum: 75
End state: account_1: 35, account_2: 50, sum: 85

Data lost, no way to recover from there!

Multi-key example (with concurrency)

time

(account_2 → account_1)	(account_1 → account_2)
Read account_1: 25	*Read account_1: 25*
Write account_1: 25 + 10 → 35	
	Write account_1: 25 – 10 → 15
Read account_2: 50	*Read account_2: 50*
	Write account_2: 50 + 10 → 60
Write account_2: 50 – 10 → 40	

Start state: account_1: 25, account_2: 50, sum: 75
End state: account_1: 15, account_2: 40, sum: 55

Data is inconsistent now due to lost updates!

Compare-and-swap (CAS) operations

To mitigate the problems with read-then-update operations,
several NoSQL databases provide atomic CAS operations

These only modify a value if a certain precondition is satisifed

With compare-and-swap, reading and updating a value
is fused into a single operation, which is guaranteed to be atomic

“Real world” multi-key example

From the MongoDB manual:

“For situations that require multi-document transactions, you can implement two-phase commit in your application to provide support for these kinds of multi-document updates. Using two-phase commit ensures that data is consistent and, in case of an error, the state that preceded the transaction is recoverable. During the procedure, however, documents can represent pending data and states.”

https://docs.mongodb.com/manual/tutorial/perform-two-phase-commits/

NoSQL databases – achievements

NoSQL databases solved the problems of scalability and availability to a great extent

But they put back the burden of handling consistency and isolation (i.e. transaction management) on client application developers

Distributed ACID transactions

NewSQL databases

Newer attempts try to bring together the best of both worlds:

- scalability and availability improvements brought by NoSQL
- ACID guarantees and transactions from relational databases

The databases that try to achieve this are often called “NewSQL” (though some of them do not provide a SQL interface at all)

Examples include VoltDB and FoundationDB (not covered here), CockroachDB, Google Spanner, FaunaDB

CockroachDB’s view on consistency

From the CockroachDB docs:

“Above all else, CockroachDB believes consistency is the most important feature of a database – without it, developers cannot build reliable tools, and businesses suffer from potentially subtle and hard to detect anomalies.”

https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer.html

Spanner’s take on transactions

From the Google Spanner paper:

“We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.”

https://research.google.com/archive/spanner.html

ACID transactions – now distributed

In a distributed system, ACID transactions need to take into account that there are multiple nodes participating

- Atomicity:
  all nodes agree on the transaction status (e.g. committed / aborted)
- Consistency:
  create a consistent snapshot of data across multiple nodes
- Isolation:
  hide ongoing concurrent transactions until commit
- Durability:
  gracefully handle node failures and stay available

Distributed ACID transactions

The implementation of distributed ACID transactions requires:

- a single (but replicated) place that controls transaction statuses
- use of MVCC (or similar means) with transaction timestamps for concurrency control and conflict resolution
- replication of all writes to all replica nodes or a majority of those

In order to be resilient and handle node failures, a distributed database always needs to write transaction data to multiple nodes

The overhead of distributed ACID transactions is unavoidably higher than the overhead of plain local transactions

Consensus

In a distributed database, the different nodes of a cluster need to agree on things, e.g. if a specific transaction has committed or not

Consensus is relatively easy if there are no failures, but it’s hard if

- the network has outages or partitions
- the network drops, duplicates or delays packages
- servers fail and do not come back again (fail-stop)
- servers fail and come back with (old) data (fail-recover)

Consensus protocols

Different protocols are in use for reaching consensus

The most commonly used protocols are

- two-phase commit (2PC) (197x)
- three-phase commit (3PC) (1983)
- the Paxos consensus protocol (1998)
- the RAFT protocol (2013)
- modified variants of those

Paxos and RAFT are proven to be correct, 2PC and 3PC have issues

Synchronization of time

In a distributed database, the nodes in a cluster also need to agree on the sequence of events, e.g. for transaction ordering and conflict resolution

Ordering transactions is a requirement to linearize concurrent transactions that access the same data, but start and commit on different nodes

What is required is an unambiguous global (cross-node) sequence of events, e.g. to order all transactions into a lineralizable sequence

Synchronization of time

Ordinary mortals use NTP to synchronize the clocks of different servers, which may produce a clock skew of two- or three-digit milliseconds

Google works around large clock skews in their datacenters by using GPS receivers and atomic clocks

Even that does not completely eliminate clock skew, but allows to keep it really low (single-digit milliseconds values)

If clock skew is bounded, two timestamps from different nodes can be compared if they are farther apart that the uncertainty window (maximum tolerable clock difference)

Timestamp comparsions

Extra measures are needed to remove uncertainty about timestamps from different nodes that are very close together:

- wait (and / or retry with an adjusted timestamp) until the uncertainty window has passed
- use clock values that contain causilities (happens-before relationships)

Hybrid logical clocks (HLC)

The idea of hybrid logical clocks is to use timestamps that have

- a physical component: current wall clock time
- a logical component: used to distinguish between events with same physical component

Normally one uses a single integer value to store both components, devoting the majority of bits to physical time, e.g.

HLC timestamp = 48 bits for physical time + 16 bits for logical time

HLC timestamp >= wall clock time

Hybrid logical clocks – causality

Whenever a message is exchanged between nodes, it contains the sender’s HLC timestamp

The receiver makes sure its own HLC timestamp is higher than the one of the sender

That way the causality is also preserved in the HLC timestamps:
the cause will have a smaller HLC timestamp than the effect

By comparing the HLC timestamps of causally linked events from different servers it is possible to determine which one was earlier (happens-before relationship)

Databases with distributed ACID transactions

CockroachDB – overview

CockroachDB is an open-source distributed SQL database

Acts like a relational database to a user, as it provides databases, tables and columns, constraints etc.

Internally it is a key / value store based on RocksDB

Provides distributed ACID transactions with isolation level “serializable”

It uses HLC timestamps for transaction timestamps and bringing them into an order

CockroachDB – ranges

Data is partionitioned by key into non-overlapping “ranges”

Each range contains the sorted keys plus the mapped values

Range -∞..es	Range et..in	Range iu..∞
ar → Argentina	et → Ethiopia	jm → Jamaica
at → Austria	fj → Fiji	pm → Panama
be → Belgium	in → India	sk → Slovakia
es → Spain	it → Italy	

Ranges are automatically split into subranges when getting too big, and replicated to a configurable amount of replicas for resilience

CockroachDB – transaction processing

Transactions are started by client applications at a coordinator node

A transaction is assigned the current HLC timestamp of the coordinator node as its provisional commit timestamp

A transaction is then executed in the following phases:

- execution of write and read operations
- commit / rollback
- garbage collection / cleanup

CockroachDB – write and read operations

Write transactions first store a transaction record with the current transaction’s status (“PENDING”) in the first range written to

All write operations are stored as “write intents” via MVCC, along with the transaction id

Write intents get converted into “real” values when the transaction commits – they are provisional until the commit

Writes are written to multiple servers using RAFT instances

CockroachDB – conflicts

Writes and reads need to check for conflicts that would violate serializability of transactions

Before storing write intents, there will be a conflict detection with other write intents and successfully committed newer writes for same keys

Read operations also check for conflicts with write intents

In case of a conflict, it is resolved and / or the transaction is retried

CockroachDB – conflicts (WW)

If a transaction finds another running transaction's write intent for the same key, it is a write-write conflict

Transaction 1	Transaction 2
begin	*begin*
write key account_1	
	write key account_1

If the conflicting transaction (T1) has a lower priority, it will be aborted – otherwise the current transaction (T2) will be retried with increased priority

CockroachDB – conflicts (WW)

If a transaction finds a more recent committed value for the same key, it is a write-write conflict

Transaction 1	Transaction 2
	begin
begin	
write key account_1	
commit	
	write key account_1

The later transaction (T2) will be retried with a higher timestamp

CockroachDB – conflicts (RW)

A conflict arises when a reader finds a write intent of a running transaction with a lower timestamp (HLC write < HLC read)

Transaction 1	Transaction 2
begin	
write key account_1	
	read key account_1

If the reader (T2) has a higher priority, the write transaction (T1) will be retried with a higher timestamp

Otherwise the read transaction (T2) will retry with a new timestamp

CockroachDB – conflicts (RW)

It’s a conflict when a reader finds a committed value or a write intent of a running transaction with a slightly higher timestamp (within the uncertainty interval) (HLC write > HLC read)
This can happen if the clock of the writer is ahead of the reader’s clock

Transaction 1	Transaction 2
begin	
write key account_1	
	read key account_1

It is uncertain if the read or write was first, so the read transaction (T2) will retry with a higher timestamp

CockroachDB – conflicts (RW)

It’s a conflict when a writer writes a key that was read with a higher timestamp (HLC write < HLC read)
This can happen if the clock of the reader is ahead of the writer’s clock

Transaction 1	Transaction 2
begin	
	read key account_1
write key account_1	

It is uncertain if the read or write was first, so the write transaction (T1) will retry with a higher timestamp

CockroachDB – commits

A transaction will only be commited if it’s not aborted and none of the participating nodes has bumped its timestamp

In case of a timestamp bump, the transaction is started again with the adjusted timestamp as its provisional commit timestamp

When a transaction commits, there is a final atomic flip of the transaction status in the transaction record from “PENDING” to “COMMITTED”

CockroachDB – cleanup

After the commit, the write intents of the transaction are cleaned up asynchronously and turned into “real” MVCC values

Outdated MVCC versions will be removed after a configurable interval (24h by default)

As long as older MVCC versions are present, time travel queries are supported

CockroachDB – query execution

Queries are parallelized to execute concurrently on multiple shards where possible

Results from individual shards may need to be aggregated on a coordinator in case of GROUP BY, SORT BY

Pushing as much work as possible to the shards reduces amount of data to be transferred back from the shards to the coordinator (WHERE, GROUP BY)

Spanner – overview

Distributed cloud database by Google with ACID transactions

Offered to the public as commercial service “Google Cloud spanner”

Closed source (but CockroachDB is an open-source clone of it)

Support for SQL select queries, data modification operations have custom APIs

Spanner – overview

Uses Google’s datacenter infrastructure with GPS receivers and atomic clocks to minimize the clock deviations between nodes of a cluster

Uses an internal API (TrueTime) to determine bounds for the timestamp uncertainty interval:

- lower bound: timestamp that is definitely in the past
- upper bound: timestamp that is definitely in the future

Delays commits (commit wait) during the uncertainty interval

FaunaDB – overview

Distributed, deterministic database

Based on the Calvin paper for distributed transactions:
http://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf

Does not replicate transaction results (modifications done by the transaction) but transaction inputs (operations to be carried out)

Transaction inputs are globally ordered in a deterministic way

Client applications need to describe the complete read and write operations for a transaction when submitting it

FaunaDB – sequencing

Client applications can send transaction requests to any node

A sequencer on each node will first batch all incoming transaction requests in the order they come in

This is done for every epoch (10ms by default)

All sequencers replicate their locally accumulated batches for the epoch to failover nodes using RAFT

FaunaDB – sequencing

Sequencers analyze the transaction inputs to determine the read and write sets for the transactions of an epoch

Then they send the transaction inputs to all nodes participating in the transaction for execution

The schedulers on all nodes will then put together a global transaction order (all transaction inputs) from all sequencers for the same epoch

This is done by interleaving all batches for the epoch in a round-robin, but deterministic manner

FaunaDB – transaction execution

After the transactions and their order for an epoch are determined, the scheduler will execute them one by one

Execution of transactions can be parallelized for unrelated transactions, but dependent transactions are executed in strict order, each in the following phases:

- acquisition of locks for all local keys accessed by the transaction
- performing all reads for all local keys
- serving / collecting remote reads
- execution of transaction logic and application of writes
- releasing locks

FaunaDB – transaction execution

By using the concept of epochs and delaying all incoming transaction requests until the epoch is finished, FaunaDB can get around the problem of the different nodes in the cluster having deviating clocks

The nodes in the cluster only have to agree on which is the current epoch

Transaction execution order is deterministic, so all replicas will produce the same transaction results

¡Thanks!

¿Any questions?

Links

Consensus protocols:
http://lamport.azurewebsites.net/pubs/lamport-paxos.pdf
http://the-paper-trail.org/blog/consensus-protocols-paxos/
https://raft.github.io/raft.pdf
https://www.microsoft.com/en-us/research/publication/consensus-on-transaction-commit

Hybrid logical clocks (HLC):
http://www.cse.buffalo.edu/tech-reports/2014-04.pdf

Links

CockroachDB:
https://www.cockroachlabs.com/

Google Spanner:
https://research.google.com/archive/spanner.html
https://cloud.google.com/spanner

Calvin / FaunaDB:
http://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf
https://fauna.com/