Databases beyond a single machine

a.k.a. "how to be web scale"

Oliver Tan, Member of Technical Staff @ Cockroach Labs

Who am I?

- UNSW in 2011 (HS1917), 2013 - 2015 (BSc)
  - Co-created WebCMS3
    - Trivia: why is the favicon of WebCMS3 a sheep?
  - Did COMP9315 in 2014
    - Wife scored higher than me but doesn't work on databases!
- Worked in the US over the past 5 years, including ~3 years at Dropbox and over 1 year at Cockroach Labs, the makers of CockroachDB.

Agenda

- A single database machine
  - Glory of ACID
  - Can we survive on a single machine
- CAP Theorem
- Sharded PostgreSQL/MySQL
  - Sharding, 2PC
- Eventual Consistency
  - Quorums

A single database machine

What could go wrong?

ACID is Great ✔✔✔✔

make sure you google "DBMS ACID" instead of just "ACID" though

✔ Atomic

The whole transaction will commit atomically.

```
BEGIN;

SELECT value
  FROM bank_accounts
  WHERE name IN ('otan', 'jas')
  FOR UPDATE;

UPDATE bank_accounts
  SET value = value - 10000
  WHERE name = 'jas';
UPDATE bank_accounts
  SET value = value + 10000
  WHERE name = 'otan';

COMMIT
```

BOTH of these statements must run; otherwise jas is poorer for no reason if only the first statement executes

✔ Consistent

The database will only transition between valid states. Data read is always from a valid state.

assume the schema is

```
CREATE TABLE bank_accounts(
  name varchar PRIMARY KEY,
  value INT CONSTRAINT
     CHECK (value >= 0),
  ….
)
```

```
BEGIN;

SELECT value
  FROM bank_accounts
  WHERE name IN ('otan', 'jas')
  FOR UPDATE;

UPDATE bank_accounts
  SET value = value - 10000
  WHERE name = 'jas';
UPDATE bank_accounts
  SET value = value + 10000
  WHERE name = 'otan';

COMMIT
```

jas can never have a negative balance as it would not follow the CHECK CONSTRAINT

✔ Isolation

Transactions can only read values that have been written or inside the current transaction.

txn executing concurrently:

```
BEGIN
UPDATE bank_accounts
  SET value = 50000
  WHERE name = 'jas';
ROLLBACK
```

assume original value of 'jas' account is 10000

```
BEGIN;

SELECT value
  FROM bank_accounts
  WHERE name IN ('otan', 'jas')
  FOR UPDATE;

UPDATE bank_accounts
  SET value = value - 10000
  WHERE name = 'jas';
UPDATE bank_accounts
  SET value = value + 10000
  WHERE name = 'otan';

COMMIT
```

when the transaction completes, the value is 0. At no point should it read the value $50000 from t2.

✔ Durable

Once written, it stays written.

```
BEGIN;

SELECT value
  FROM bank_accounts
  WHERE name IN ('otan', 'jas')
  FOR UPDATE;

UPDATE bank_accounts
  SET value = value - 10000
  WHERE name = 'jas';
UPDATE bank_accounts
  SET value = value + 10000
  WHERE name = 'otan';

COMMIT
```

If this returns success, one can assume the data is permanently written.

But can you survive on one machine?

Spoilers: no

What if the machine is unreachable?

💩 happens:

- hard disks fail regularly
- datacenter catches on fire and is unrecoverable
- sharks eat underwater cables between continents

SEJ » News

OVH Data Center Fire Darkens Popular Sites Worldwide

Fire at an OVH Data Center in Europe takes down millions of sites, including WP Rocket and Imagify

Quarterly Hard Drive Failure Stats for Q2 2020

At the end of Q2 2020, Backblaze was using 140,059 hard drives to store customer data. For our evaluation we remove from consideration those drive models for which we did not have at least 60 drives (see why below). This leaves us with 139,867 hard drives in our review. The table below covers what happened in Q2 2020.

Backblaze Q2 2020 Annualized Hard Drive Failure Rates
Reporting period: April 1, 2020 through June 30, 2020 inclusive

MFG	Model	Drive Size	Drive Count	Drive Days	Drive Failures	AFR
HGST	HMS5C4040ALE640	4TB	2,952	266,200	1	0.14%
HGST	HMS5C4040BLE640	4TB	12,739	1,159,472	9	0.28%
HGST	HUH728080ALE600	8TB	1,000	91,000	0	0.00%
HGST	HUH721212ALE600	12TB	2,600	200,188	3	0.55%
HGST	HUH721212ALN604	12TB	10,846	986,674	19	0.70%
Seagate	ST4000DM000	4TB	19,093	1,739,577	49	1.03%
Seagate	ST6000DX000	6TB	886	80,626	0	0.00%
Seagate	ST8000DM002	8TB	9,795	890,937	17	0.70%
Seagate	ST8000NM0055	8TB	14,462	1,316,313	33	0.92%
Seagate	ST10000NM0086	10TB	1,200	109,200	2	0.67%
Seagate	ST12000NM0007	12TB	35,095	3,319,854	82	0.90%
Seagate	ST12000NM0008	12TB	15,543	1,279,568	27	0.77%
Seagate	ST12000NM001G	12TB	4,799	137,929	8	2.12%
Seagate	ST16000NM001G	16TB	59	5,431	1	6.72%
Toshiba	MD04ABA400V	4TB	99	9,009	0	0.00%
Toshiba	MG07ACA14TA	14TB	8,699	663,647	20	1.10%
		TOTALS	139,867	12,255,625	271	0.81%

BACKBLAZE

Forbes

Aug 15, 2014, 06:59pm EDT

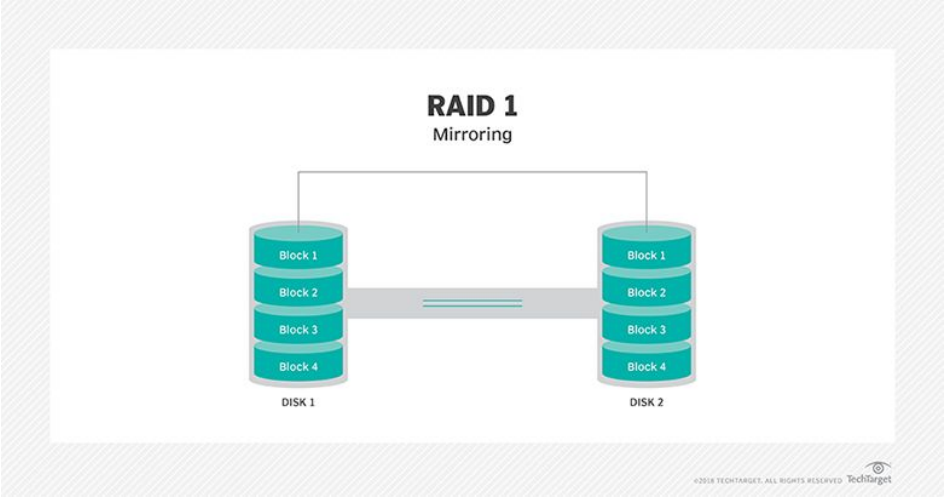
How Google Stops Sharks From Eating Undersea Cable

Amit Chowdhry Contributor
Consumer Tech
Tech enthusiast, born in Ann Arbor and educated at Michigan State

RAID

- Store the same data on multi disks
  - Controller is in charge of splitting the data and ensuring everything gets written.
  - Lots of different configurations available.
  - Can involve sacrifice of disk write latency.
- "Resolves" the hard drive issue but not the network issue.

a very old school strategy!

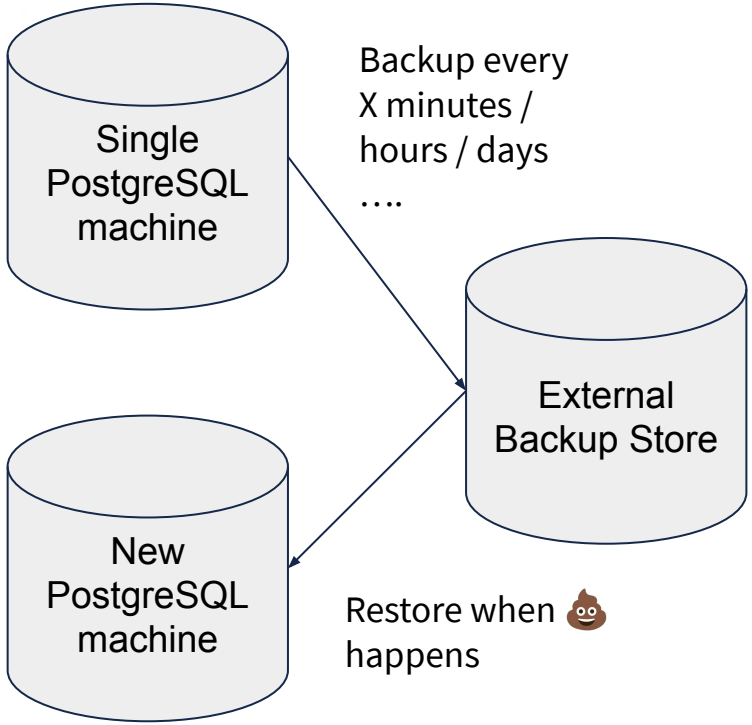


Using backups to restore to a new machine?

Not "web-scale" because:

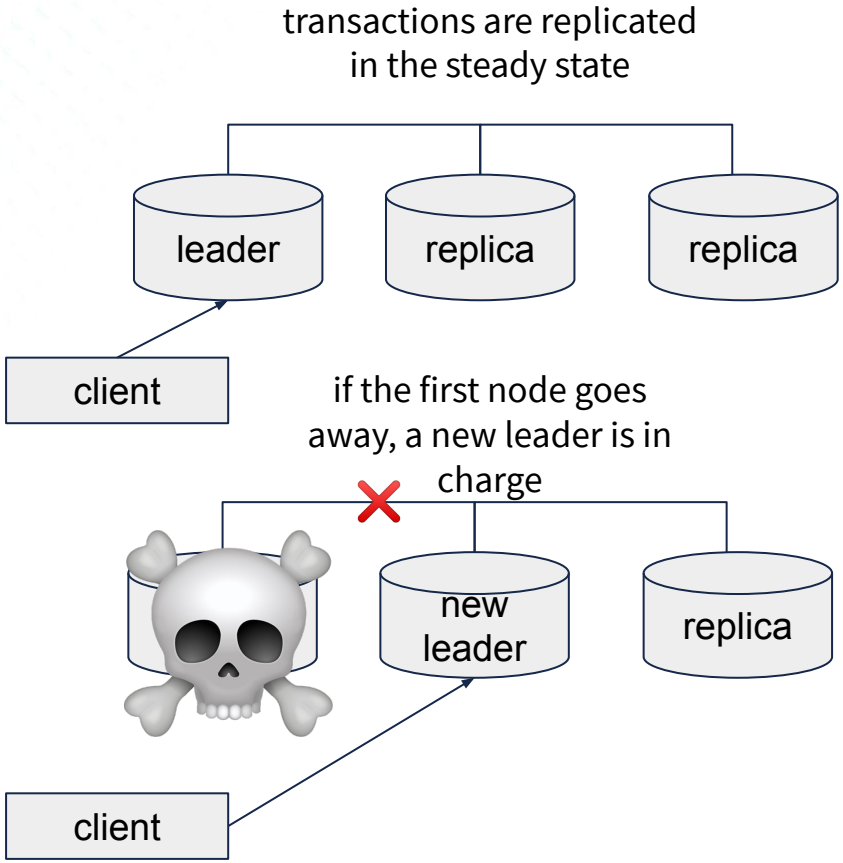
- Backups can be out of date
- Restoring machines can take significant time to reload data.
  - We want 0 downtime!

(But still important to run regular backups in case 💩 happens for other reasons, e.g. someone makes a DROP TABLE …)



Using replicas

- Have "follower" nodes keep in sync with the leader.
  - Leader serves traffic, but replicas can be used to read stale data.
  - Replication can be done synchronously or asynchronously.
  - Called "master" and "standby" nodes in PostgreSQL.
- When the leader is down, perform a "failover" in which a replicas becomes the new leader.
  - Data can be served straight away.



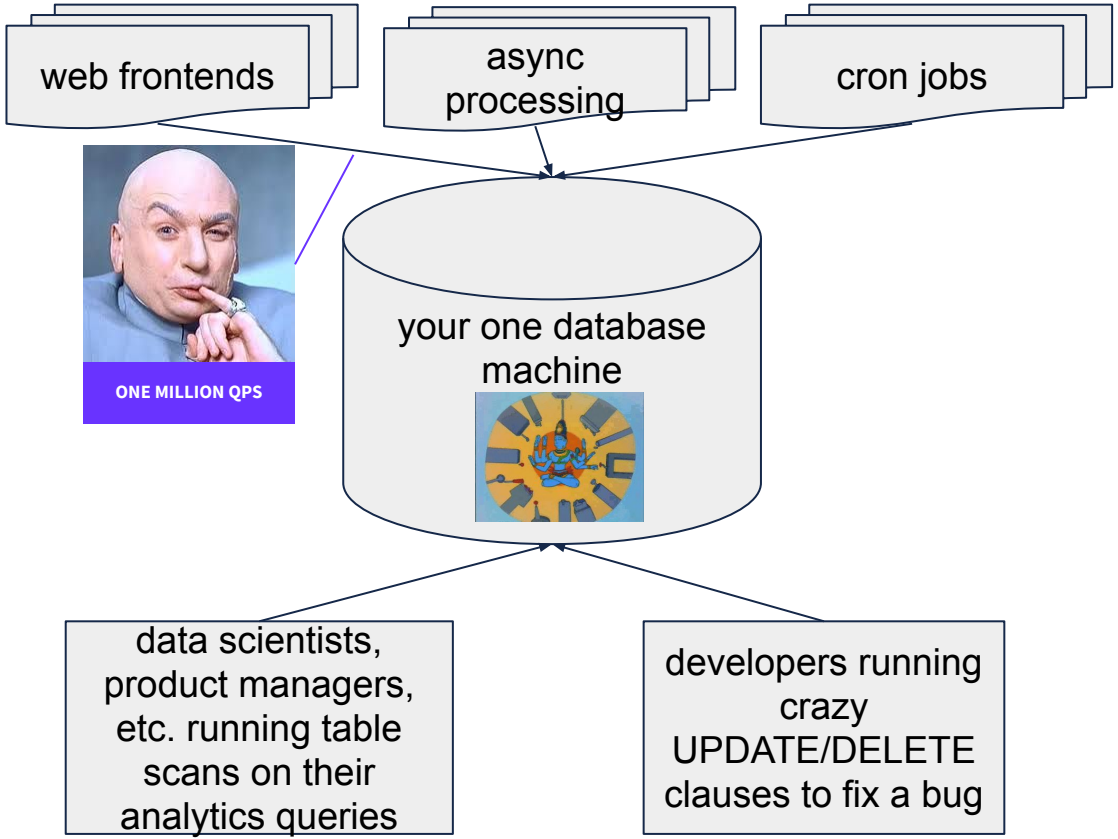
In the real world,
💩 happens
all the time

you MUST design to handle what 💩 comes at you

Can the machine handle such a load?

Going web-scale?

You'll probably handle a lot of queries, and most likely, all your code needs to access the database to retrieve data!



Up the machine specs

- Also known as "vertical scaling"
  - "throwing money at the problem"
- Simply buff up CPU / memory / disk when required
- Does not solve the "💩 happens" dilemma

High spec machines can be hard to find or very, very expensive

- AWS EC2 has some beefy machines which scales in price above linearly:
  - t4g.xlarge (4 vCPUs, 16GiB RAM costs ~$0.08/hour (~$700/year)
  - m6g.8xlarge (32 vCPUs, 128GiB RAM) costs ~$0.77/hour (~$6.8/year)
  - m5a.24xlarge (96 vCPUs, 384GiB RAM) costs ~$2.61/hour (~23k/year)
  - but there's a hardware (and software) limits here!
- IBM Mainframes (100+ cores, 100TB+ RAM) are upwards of 75k each + hundreds of thousands of dollars each year in support and maintenance
  - the latter is how they get ya

Avoid the Single Point of Failure

a single database machine is not viable

Solution: use multiple machines!

a.k.a. "horizontal scaling"

CAP Theorem

alternatively, "why we can't have nice things"

Consistency

Every read receives the most recent write or an error.

NOTE: different to C in ACID - in this case, which refers to a "database" level consistency.

```
BEGIN;

SELECT value
  FROM bank_accounts
  WHERE name IN ('otan', 'jas')
  FOR UPDATE;

UPDATE bank_accounts
  SET value = value - 10000
  WHERE name = 'jas';
UPDATE bank_accounts
  SET value = value + 10000
  WHERE name = 'otan';

COMMIT
```

If this statement succeeds, when querying a node with "jas", jas will always have $10k less.

If this statement succeeds, when querying a node with "otan", otan will always have $10k more.
note: does NOT mean the transaction has to be atomic!

Availability

Every request receives a non-error response.

```
BEGIN;

SELECT value
  FROM bank_accounts
  WHERE name IN ('otan', 'jas')
  FOR UPDATE;

UPDATE bank_accounts
  SET value = value - 10000
  WHERE name = 'jas';
UPDATE bank_accounts
  SET value = value + 10000
  WHERE name = 'otan';

COMMIT
```

none of these statements will error

Partitioning

The system continues to respond even if some nodes cannot be reached.

```
BEGIN;

SELECT value
  FROM bank_accounts
  WHERE name IN ('otan', 'jas')
  FOR UPDATE;

UPDATE bank_accounts
  SET value = value - 10000
  WHERE name = 'jas';
UPDATE bank_accounts
  SET value = value + 10000
  WHERE name = 'otan';

COMMIT
```

even if some part of the database is down, some/all statements will succeed.

note this is NOT true in vanilla PostgreSQL!

Everything is a trade-off

your job writing software is to find the one that works best for you

CAP Theorem

Consistency; Availability; Partitioning
pick two

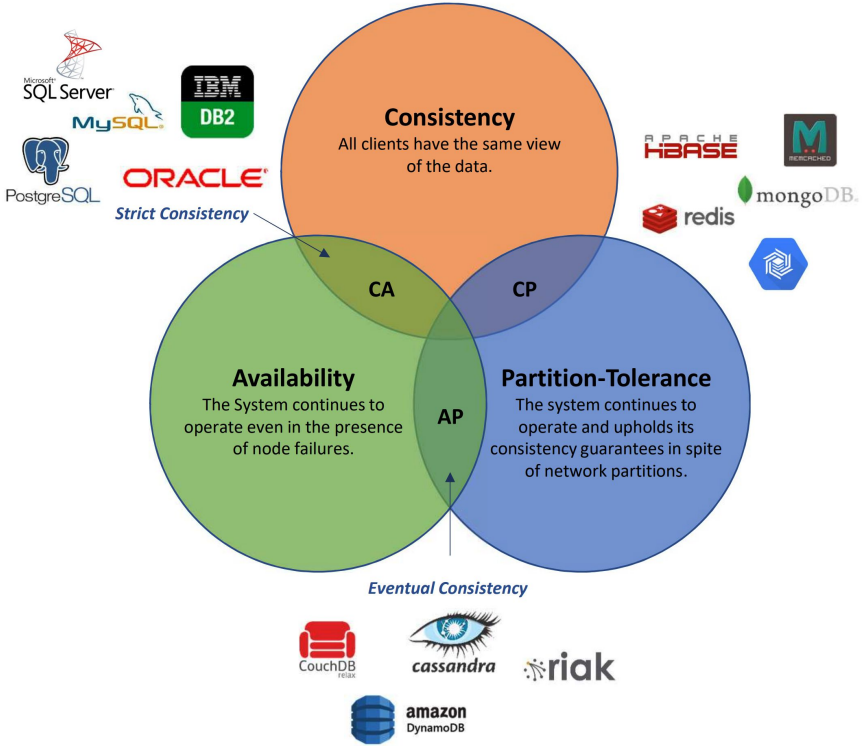
(or less)

We want to survive a network partition, so....

We can either:

- cancel the operation, **decreasing the availability but ensure consistency**
- proceed with the operation, **providing availability but risking inconsistency**

Databases by CAP Theorem



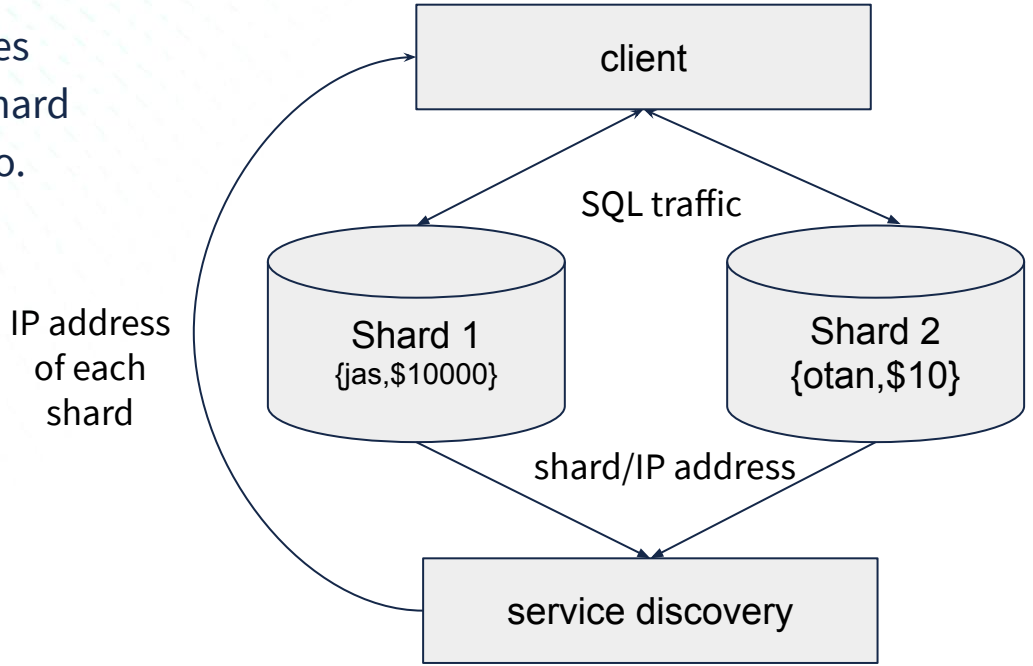
(source)

Sharded MySQL/PostgreSQL

e.g. vitess, lots of in-house DBMSs

What is sharding?

- Split data between different machines
- Need a mechanism to know which shard the data you are accessing belongs to.

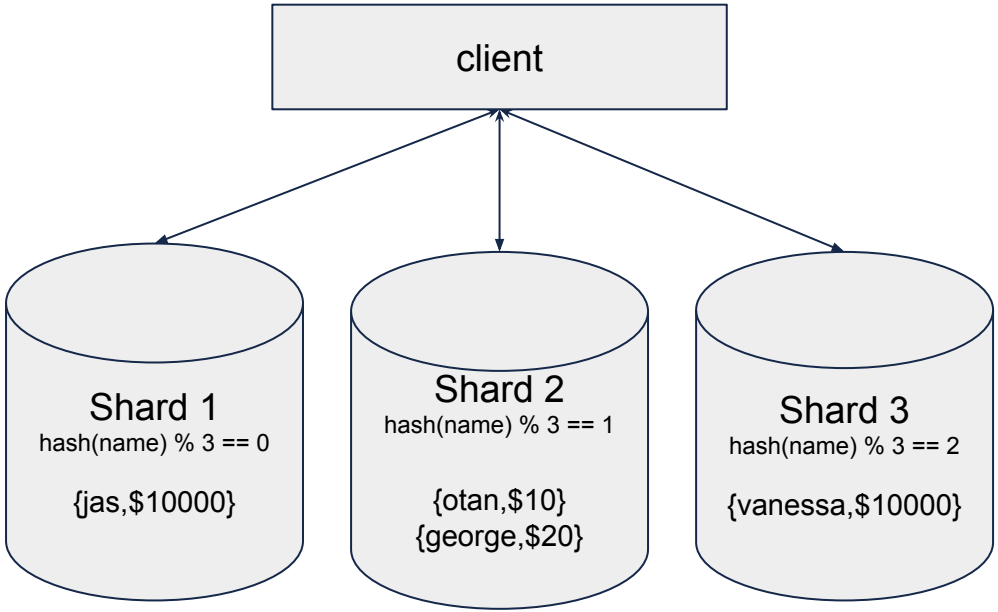


simplified example of client talking to shards directly using some service discovery mechanism

Sharding Strategy: modulus

Take modulus of the hash of the PRIMARY KEY and route appropriately

✓ easy to reason about
× expensive to split / recombine shards
× easy to get a "hot shard"
× cannot perform a range scan



Sharding Strategy: range based sharding

Shard by ranges of a PRIMARY KEY

✓ range scans
✓ relatively cheap to split / recombine shards
✓ easy to split shards if they get too busy
× more chance of getting a "hot shard" if a range is hot

	PRIMARY KEY	DATA
shard 1 (a-j)	george	...
	jas	...
shard 2 (k-u)	otan	...
shard 3 (v-z)	vanessa	...
	zebra	...

Sharding Strategy: consistent hashing

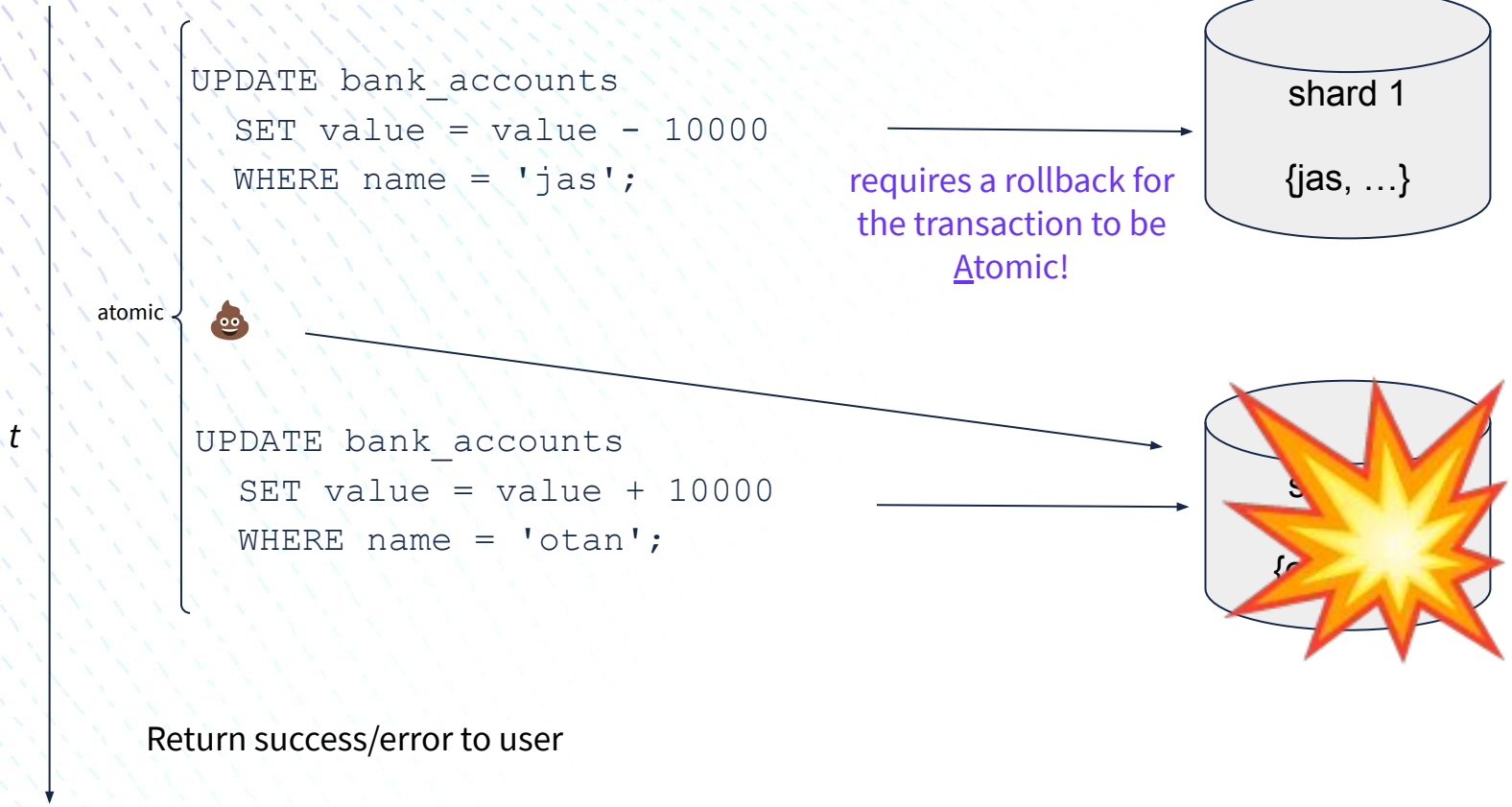
Hash the PRIMARY KEY and assign "ranges" on the hash.

✓ relatively cheap to split / recombine shards
✓ easy to split shards if they get too busy
✓ avoid "hot shards" by nature of randomness
× expensive to do "range scans"

	HASH(PRIMARY KEY)	PRIMARY KEY	DATA
shard 1 (a-j)	aaaa	otan	...
	bbbb	vanessa	...
shard 2 (k-u)	kkkk	jas	...
shard 3 (v-z)	vvvv	zebra	...
	zzzz	george	...

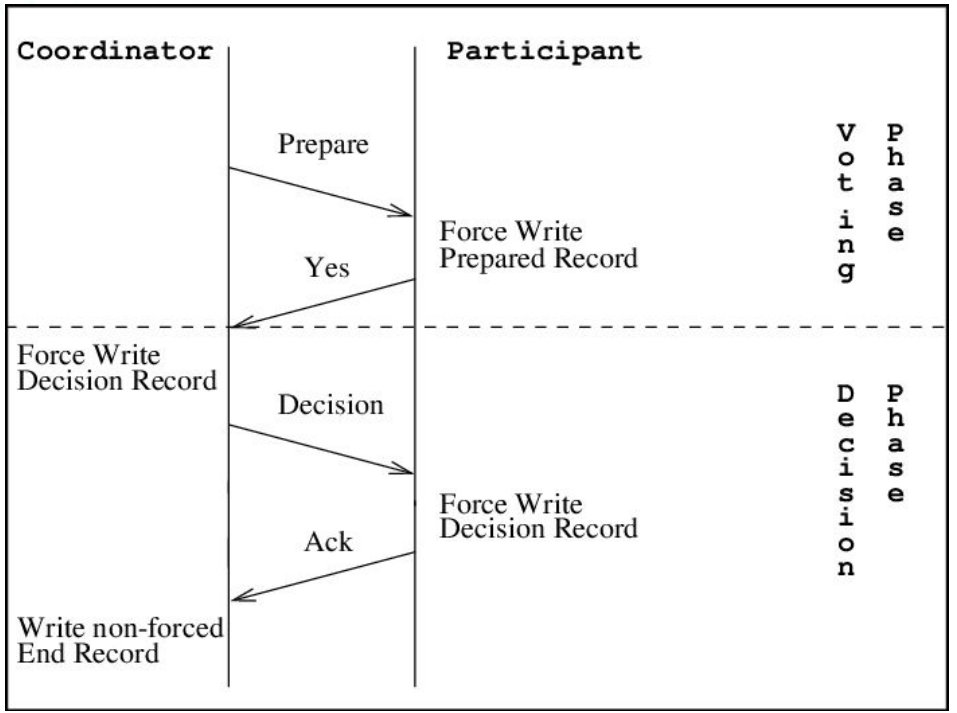
Can we make sharded MySQL/PostgreSQL ACID?

Naive Sharded Transaction



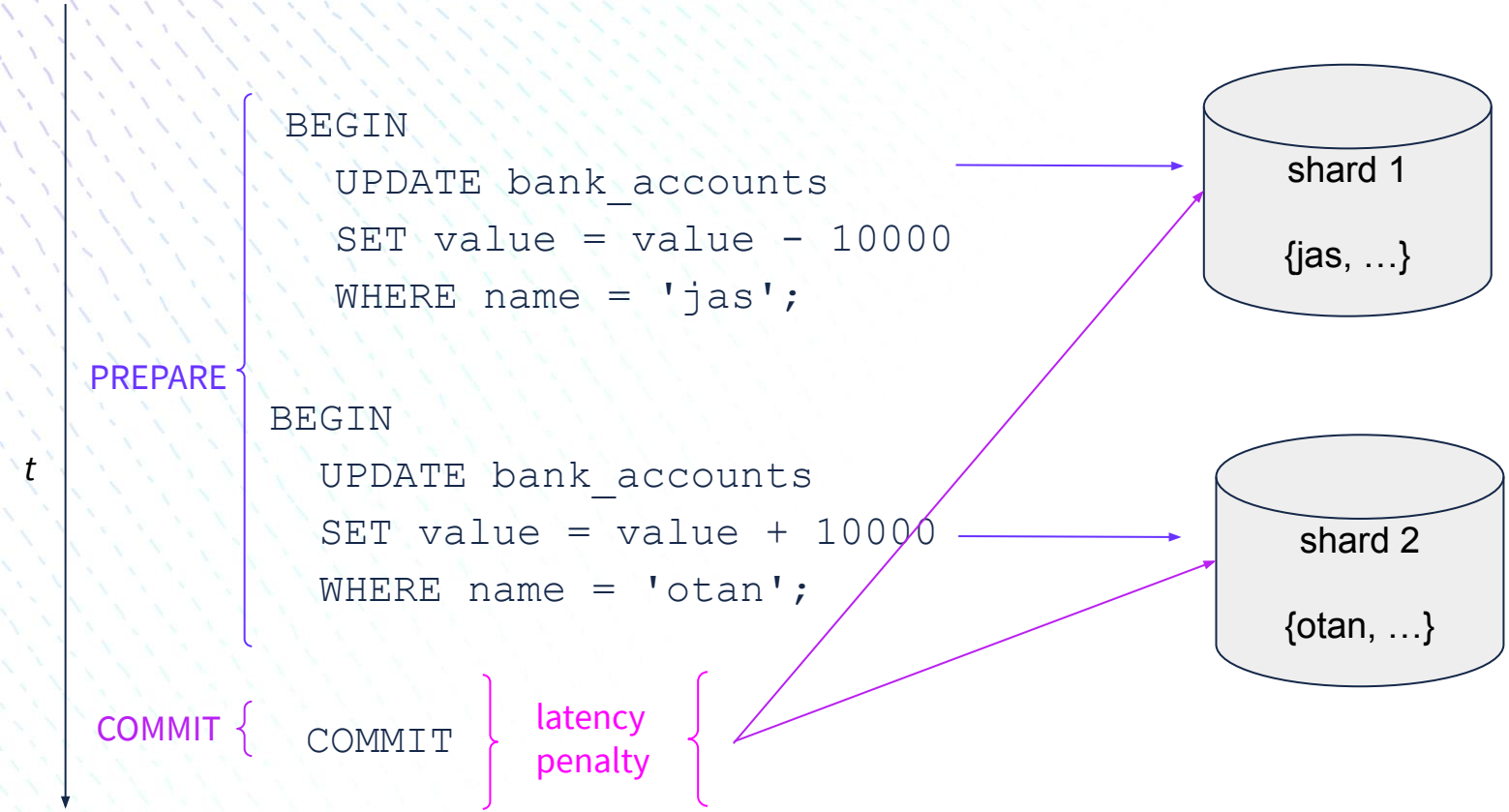
Two-Phased Commit (2PC) to the Rescue!

- **Prepare Phase (Voting Phase)**
  - Coordinator tells all relevant shards (participant) that a commit is about to take place.
  - Shards respond whether the commit is ok.
- **Commit Phase (Decision Phase)**
  - If all shards agree, the coordinator tells each shard to COMMIT.



(source)

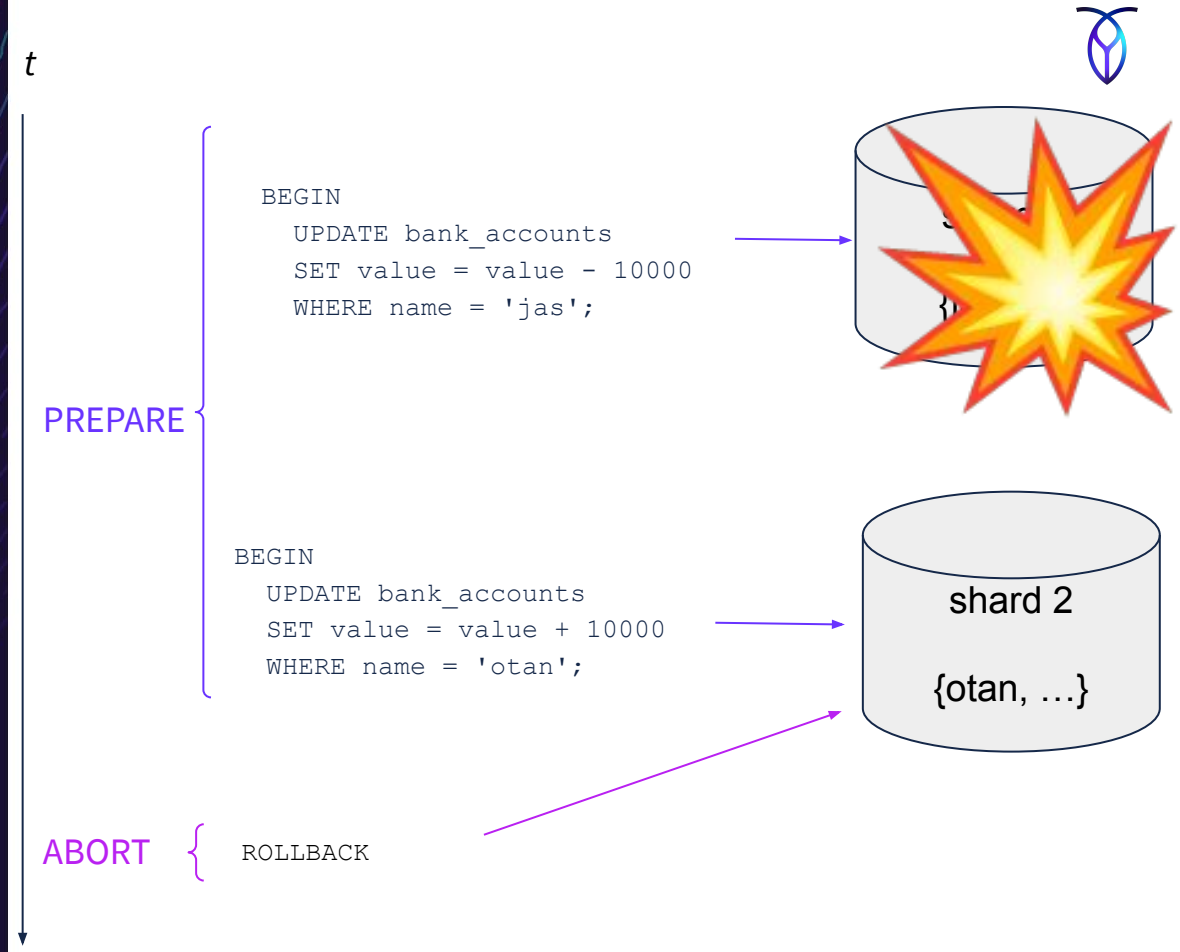
Simplified 2PC model using sharded SQL



What if a participant dies during the prepare phase?

The coordinator will be unable to get a successful response from the participant, and so the transaction will not be committed.

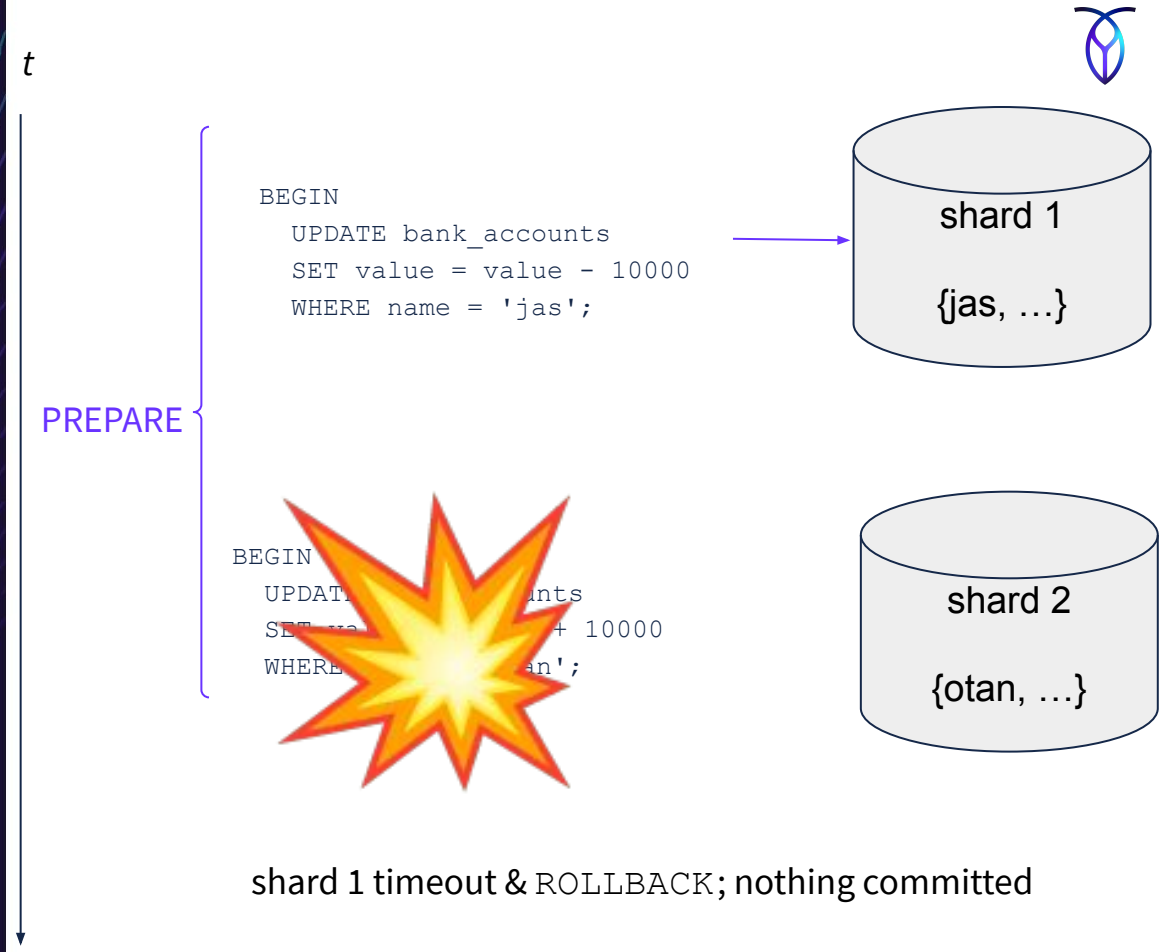
The coordinator tells everyone else to rollback.



What if the coordinator dies during the prepare phase?

The participants will never have their transactions committed, as that message is never sent!

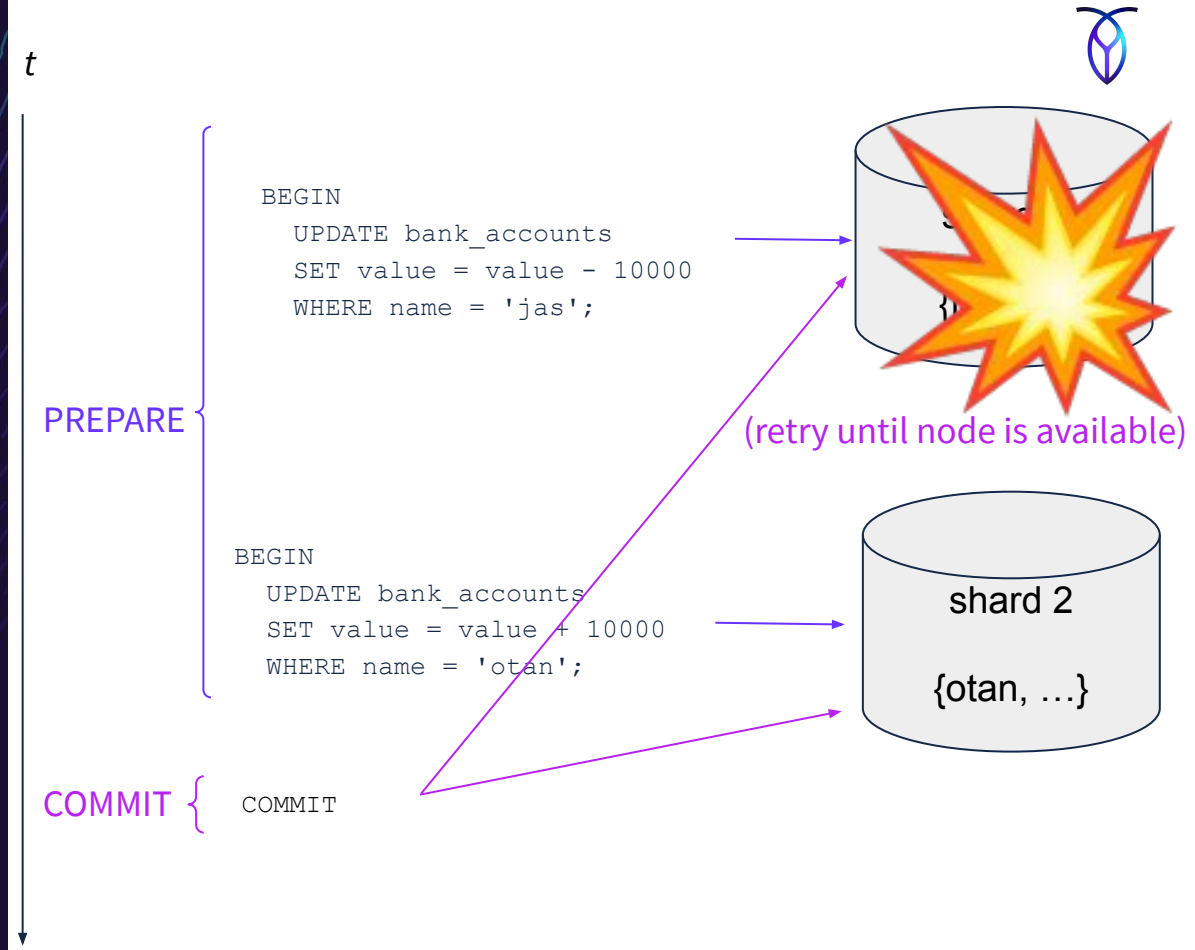
Timeouts can be built into the protocol to ensure long-running transactions do not remain and the participants do not block.



shard 1 timeout & `ROLLBACK`; nothing committed

What if a participant dies during the commit phase?

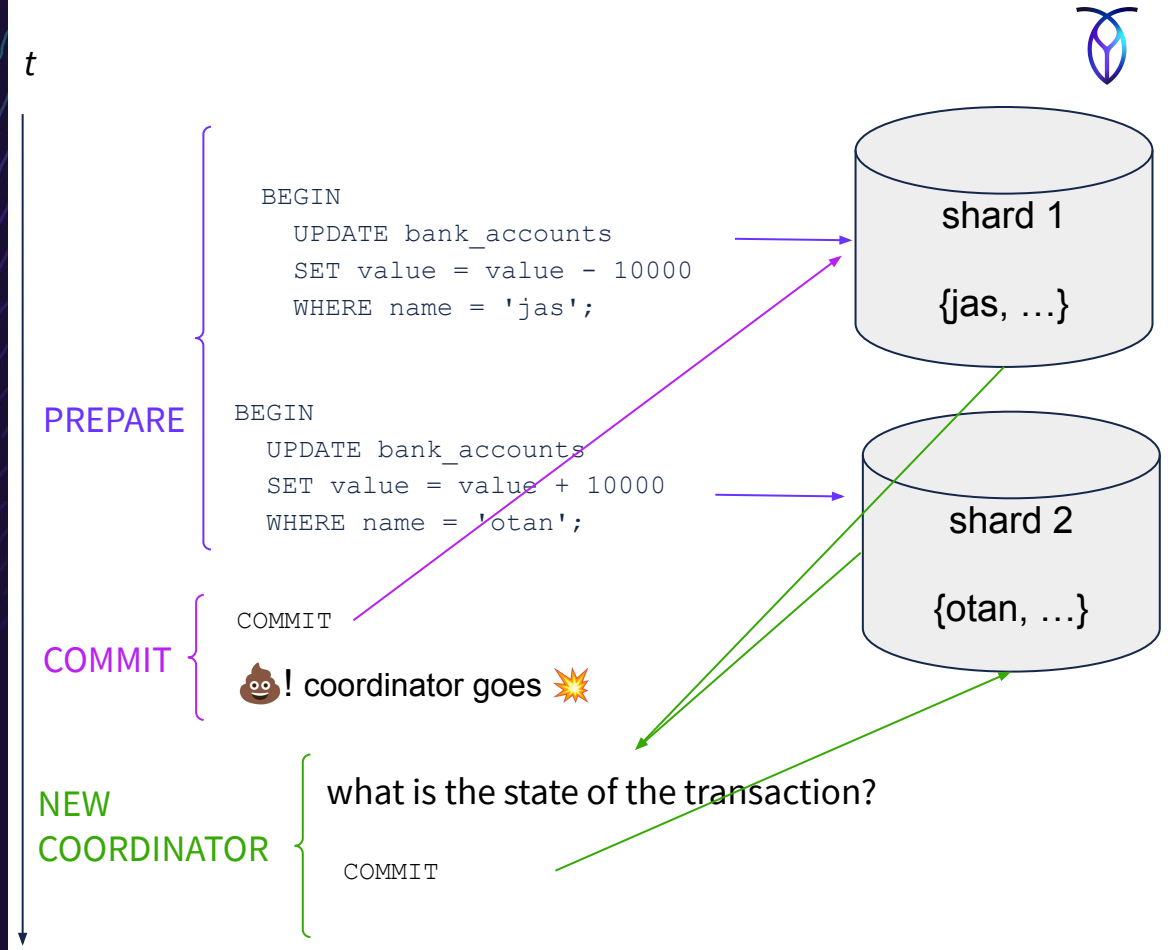
The coordinator will be responsible for ensuring that the participant commits the transaction when it comes back.



What if the coordinator dies during the commit phase?

Some participants are stuck. Another coordinator must pick up where the coordinator left off.

When a new coordinator comes in, state may have to be re-read off existing participants to catch up.



Are we being too pedantic about the error case?

- Amazon EC2 promises 99.99% availability - otherwise it will pay you money!
- **At 99.99%, one machine can be down for up to 1 minute a week!**
- Let's say you have 10 machines - that's ((99.99%)^10)=99.9% chance of a machine being down at any moment, or 10 minutes a week! That's even scarier!
- **The more you scale, the more 💩 happens. Databases are important and cannot be giving incorrectly persisted data - so no - this is not pedantic!**
    - AVOID SINGLE POINTS OF FAILURE!

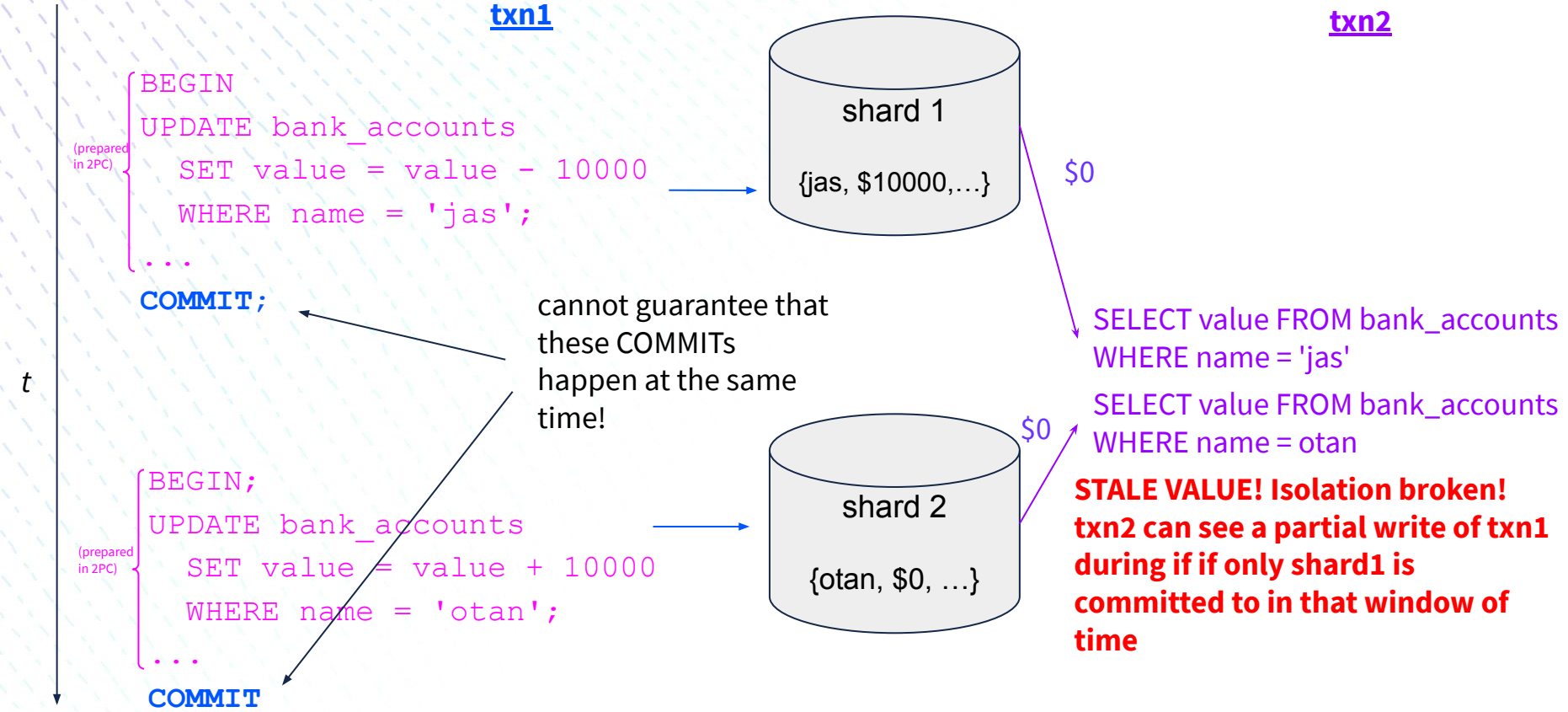
Service Credits

Service Credits are calculated as a percentage of the total charges paid by you (excluding one-time payments such as upfront payments made for Reserved Instances) for the individual Included Service in the affected AWS region for the monthly billing cycle in which the Unavailability occurred in accordance with the schedule below.

Monthly Uptime Percentage	Service Credit Percentage
Less than 99.99% but equal to or greater than 99.0%	10%
Less than 99.0% but equal to or greater than 95.0%	30%
Less than 95.0%	100%

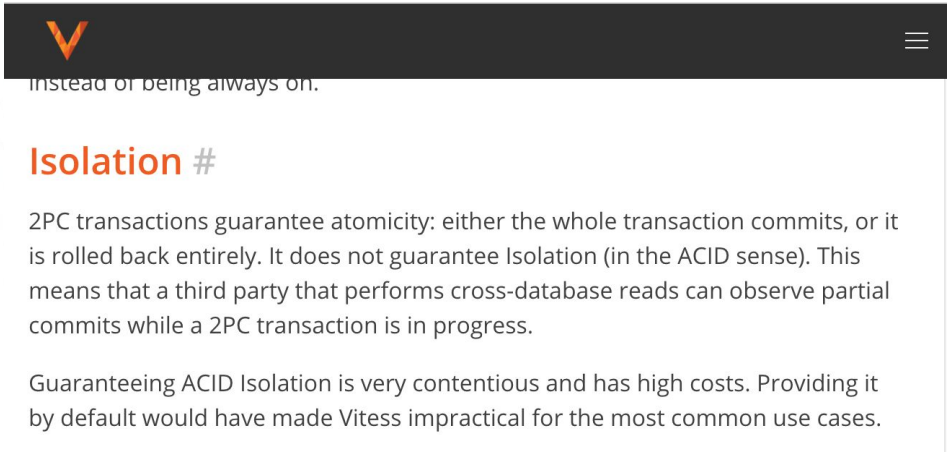
(source)

What about Isolation?



Resolving Isolation on Distributed DBMSs

- That's at least another hour of magical mind blowing detail involving time and clocks!
  - for the nerds, here's a good intro into a solution to this problem
- tl;dr: it's complicated, contentious on the system and requires extra synchronization between shards on reads and writes (hence a latency penalty)
  - even vitess says it's too hard and would be impractical*, so they made a **trade-off** was made to not even offer it


instead of being always on.

Isolation

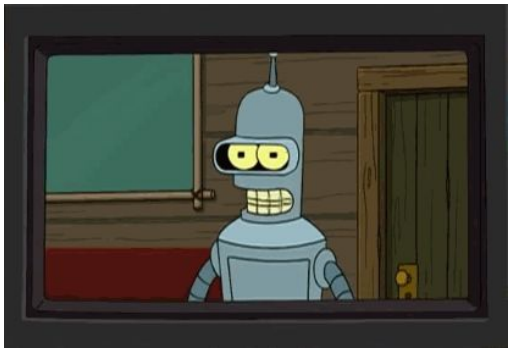
2PC transactions guarantee atomicity: either the whole transaction commits, or it is rolled back entirely. It does not guarantee Isolation (in the ACID sense). This means that a third party that performs cross-database reads can observe partial commits while a 2PC transaction is in progress.

Guaranteeing ACID Isolation is very contentious and has high costs. Providing it by default would have made Vitess impractical for the most common use cases.

(from vitess documentation)

Can we make sharded MySQL/PostgreSQL ACID?

yes, but it's mostly a trade-off of **latency** and **implementing it yourself***

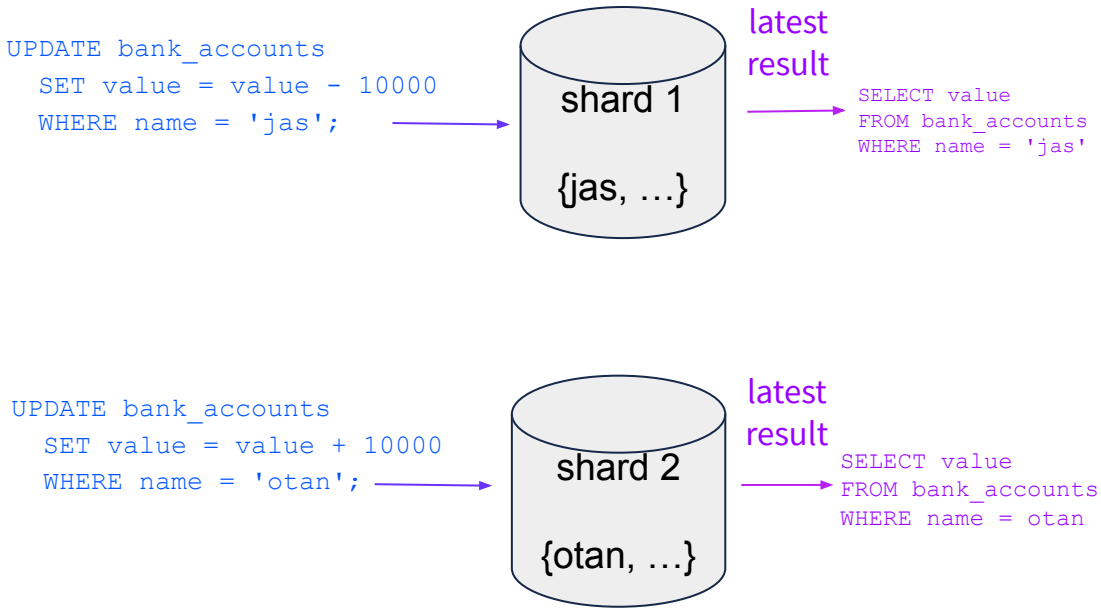
* 

Where are we under CAP Theorem?

✔ Consistent

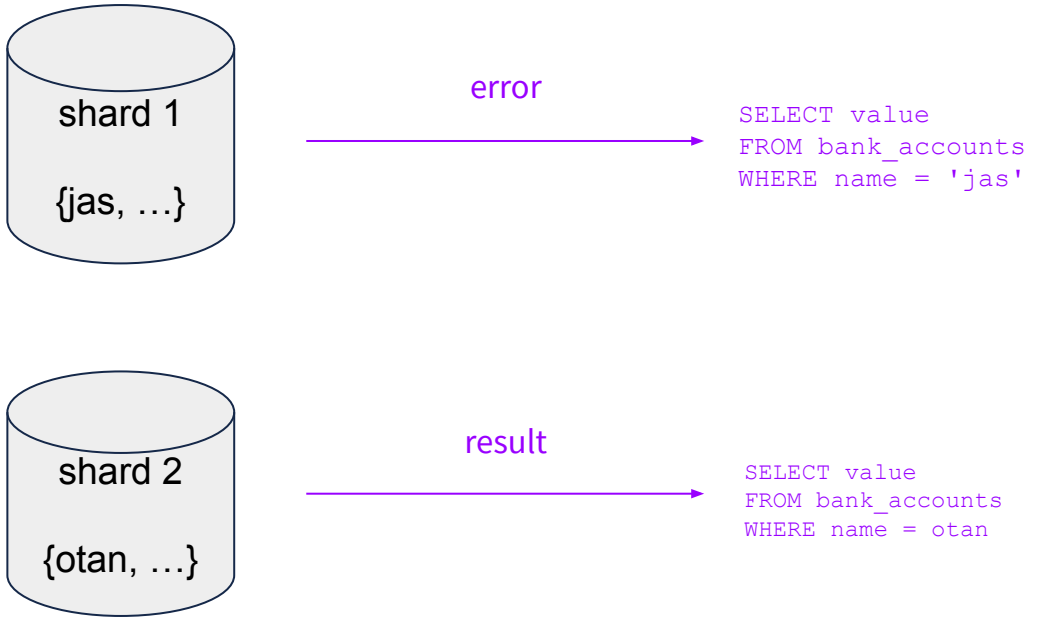
Even without 2PC, we are still deemed consistent.

This is because the latest write to a shard can always be read back.



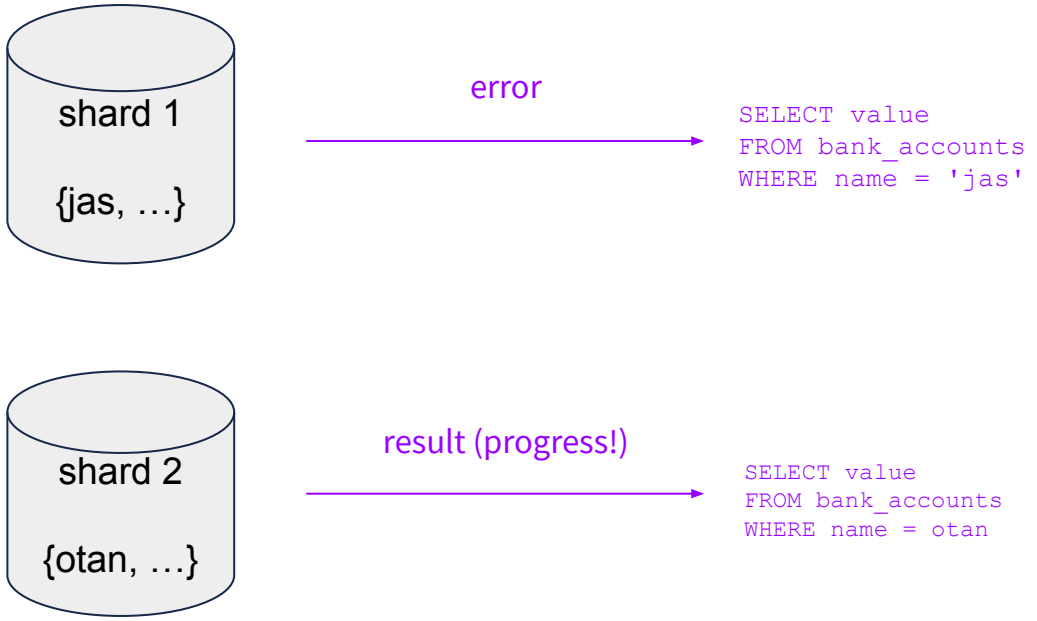
× Available

If a shard containing the data you are looking for is down, your query will return an error.



✔ Partitioning

Even if a shard is unavailable, the system can otherwise continue to make progress.



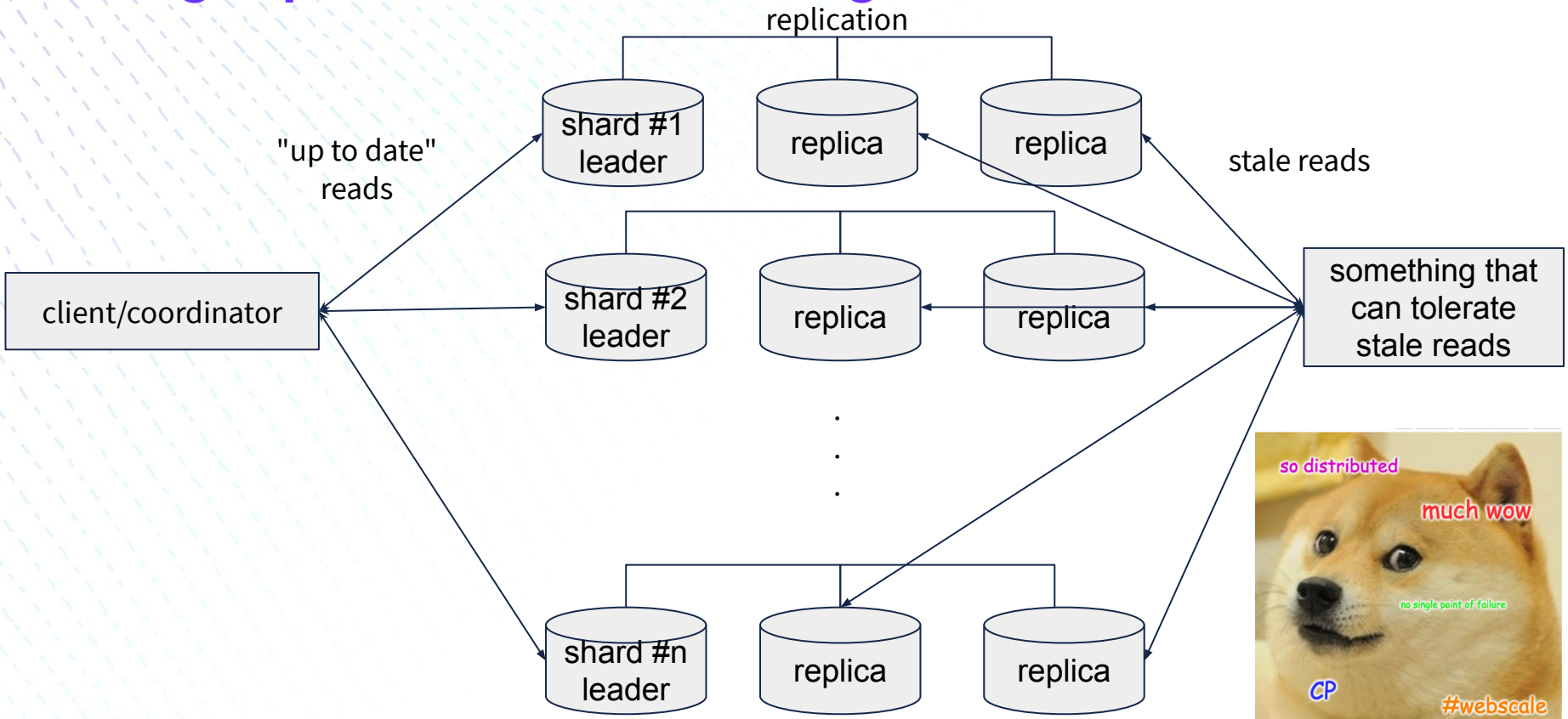
Where are we under CAP Theorem?

Answer: CP

Other things lost with sharding MySQL/PostgreSQL

- Foreign Keys
  - Corollary: CASCADES, certain triggers
- Uniqueness of PRIMARY KEYs/unique indexes
- Schema Changes (e.g. ADD/DROP COLUMN, CREATE/DROP INDEX) may not apply atomically across all shards in the cluster and require coordination
  - can have performance impacts if some shards have an index whilst others do not.
- User-defined functions may be out of sync between shards

Putting replicas and shards together...



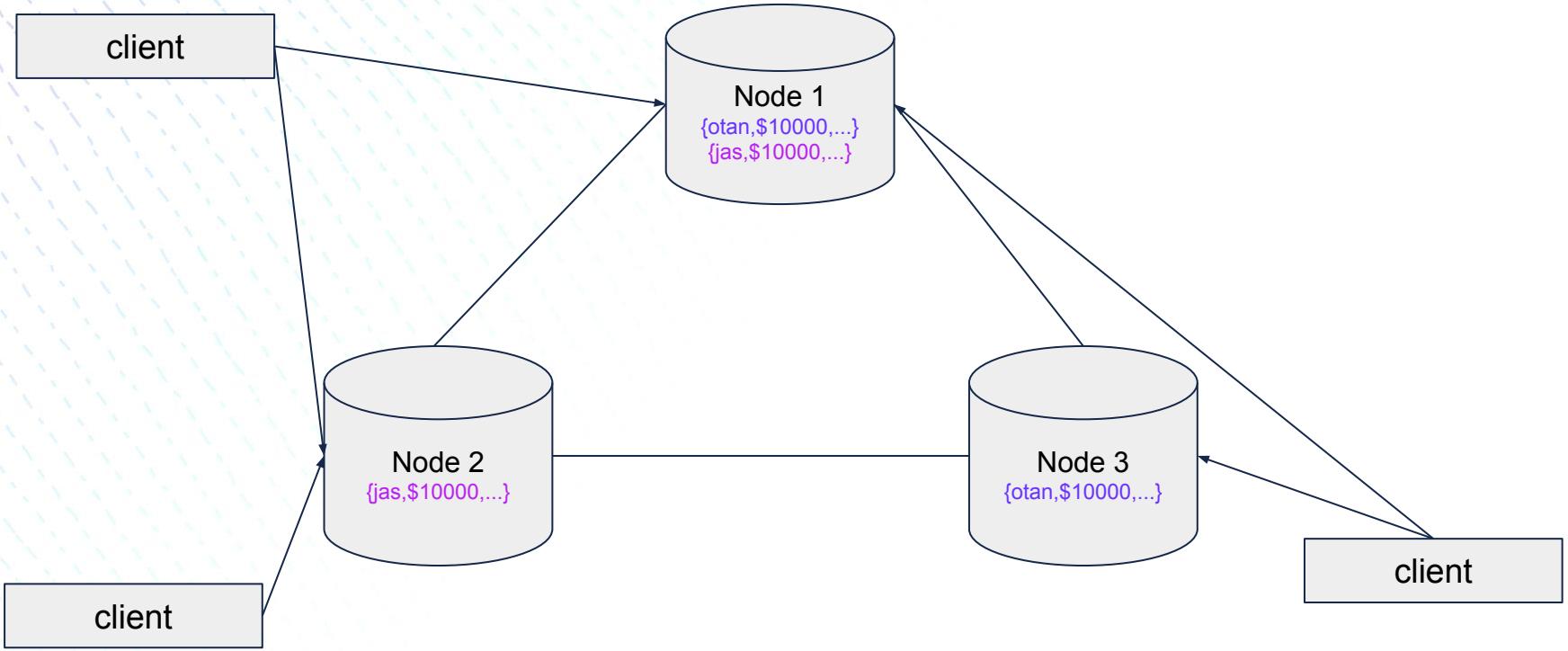
Takeaways

- Sharding gives us parallelism and enables us to still operate under a partition
- In a distributed system, we pay latency penalties to maintain ACID transactions:
  - latency on 2PC to maintain atomicity
  - latency on cross-shard dependencies to maintain isolation
- This is not implemented out of the box for MySQL/PostgreSQL!
- Don't skimp on handling the error case, because 💩 WILL happen
- **Distributed database systems are hard**

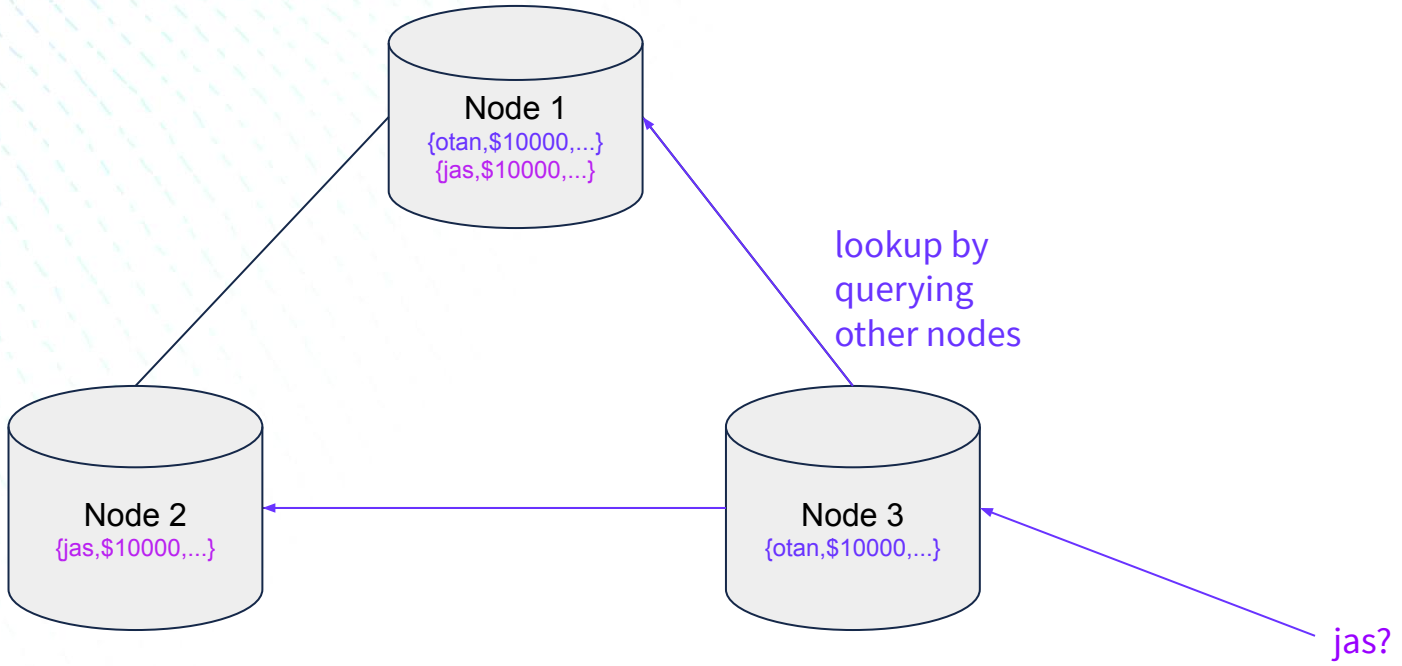
Eventual Consistency & AP DBMSs

e.g. Apache Cassandra, Amazon DynamoDB

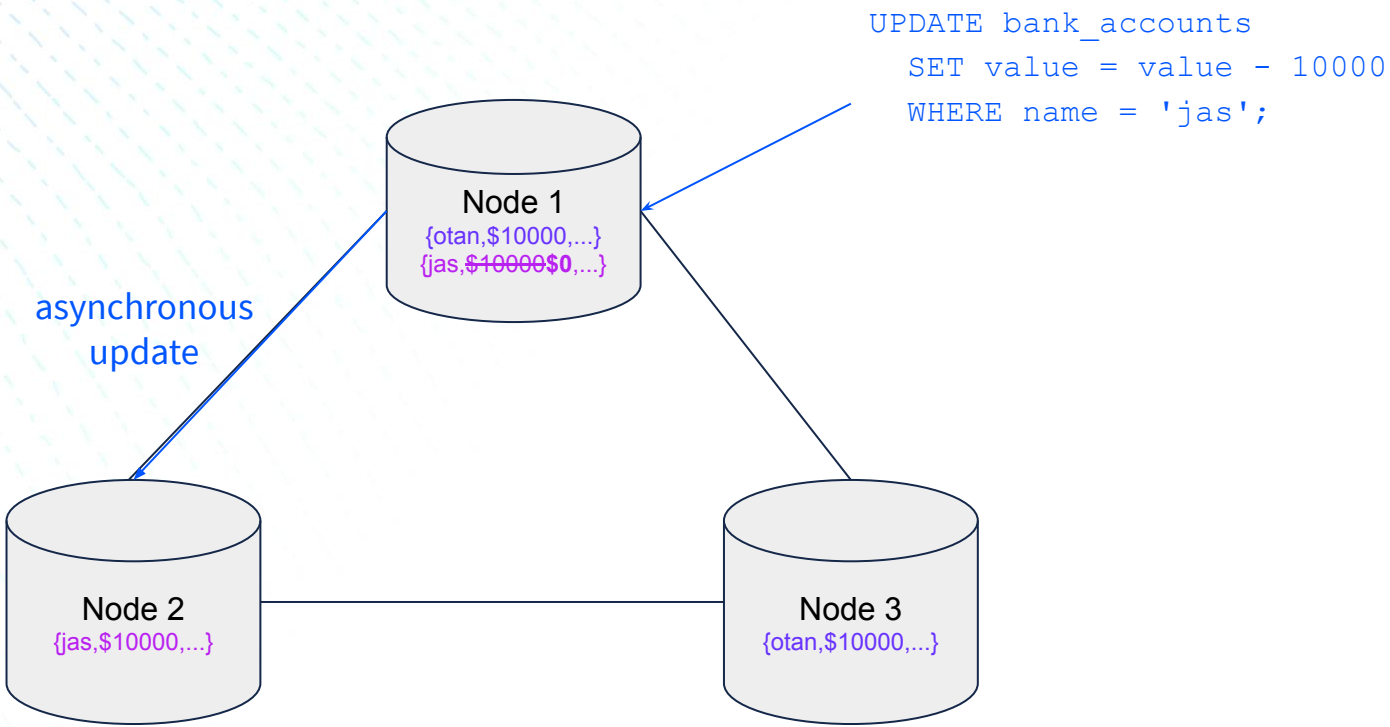
A new architecture...



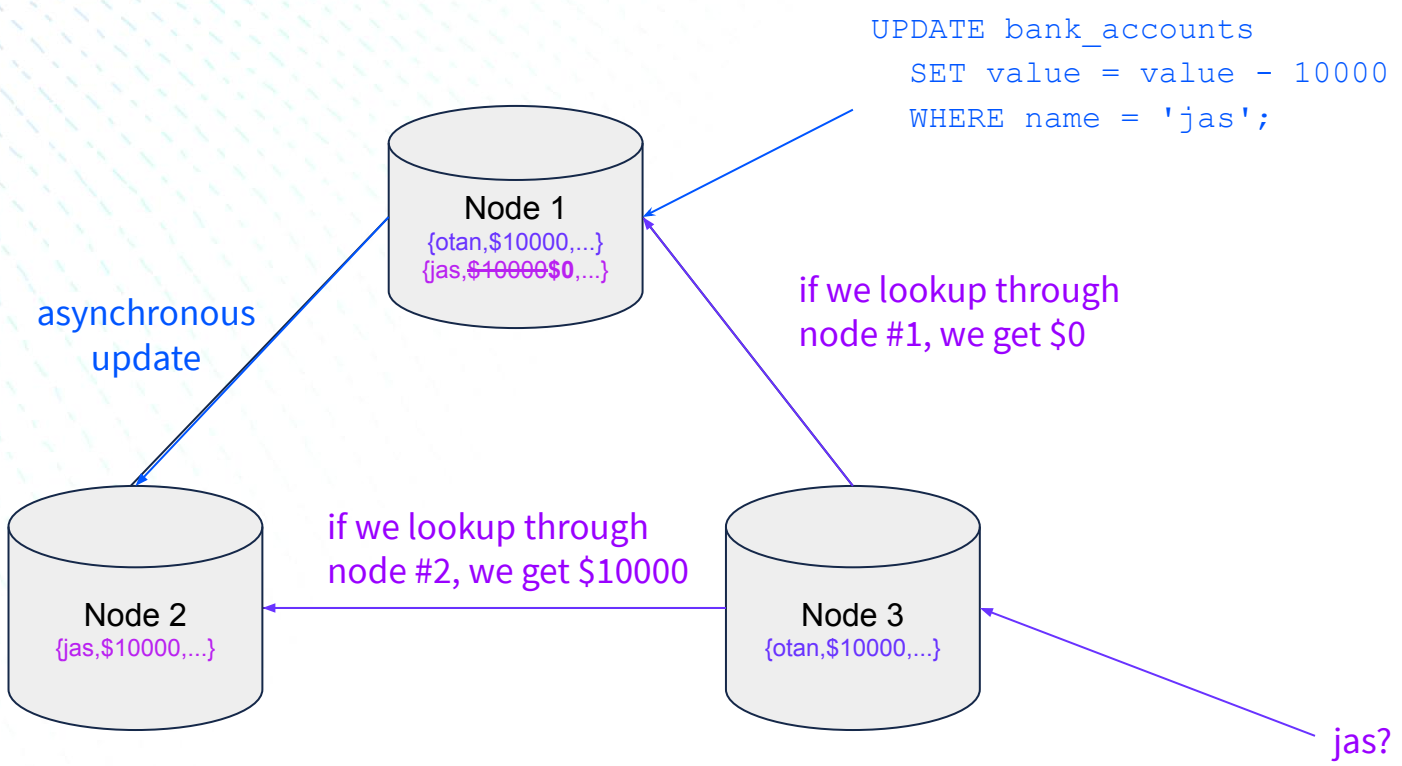
Querying a row



Updating replicas asynchronously



A racy read

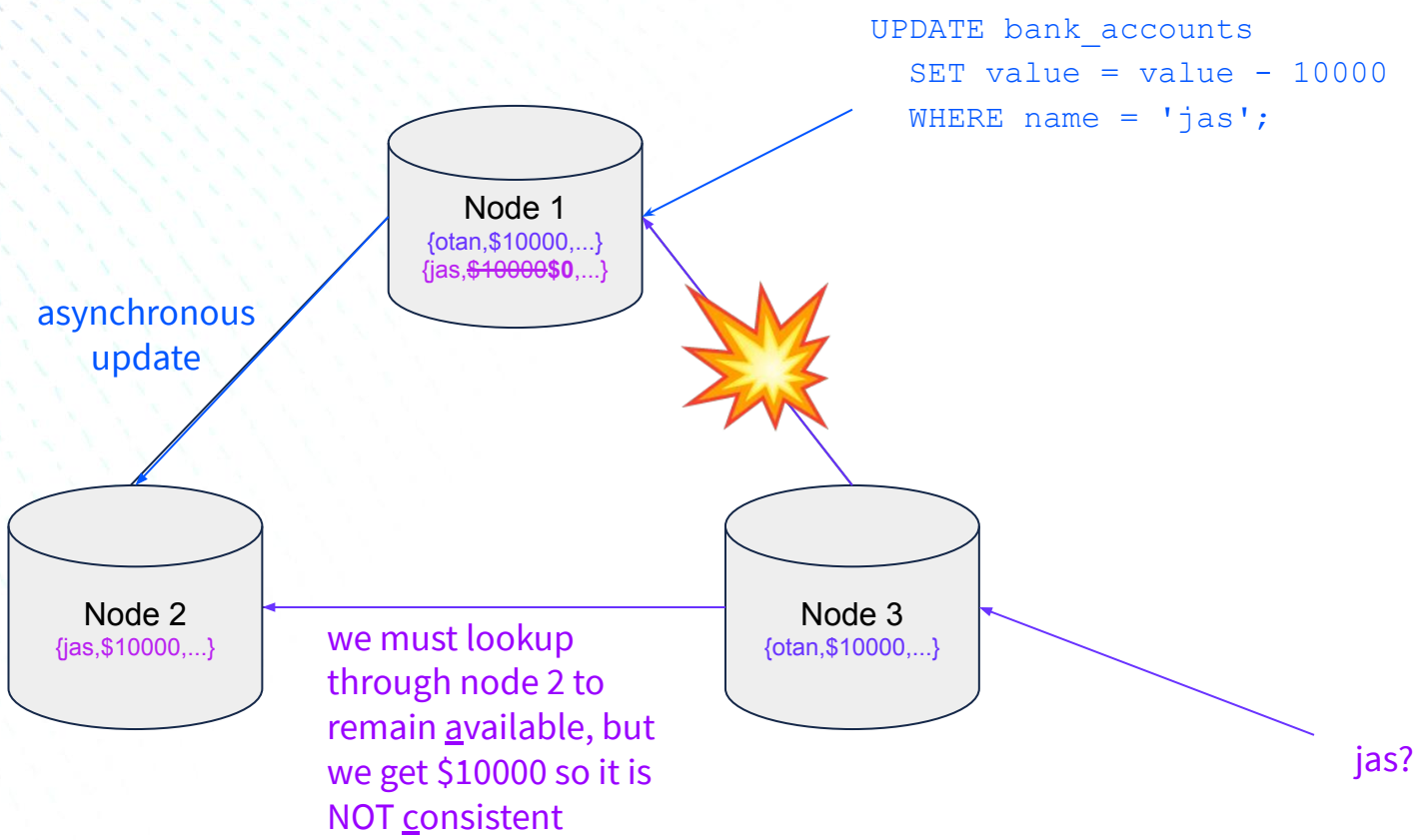

UPDATE bank_accounts
SET value = value - 10000
WHERE name = 'jas';
Node 1
{otan,$10000,...}
{jas,$10000$0,...}
asynchronous update
if we lookup through node #1, we get $0
Node 2
{jas,$10000,...}
if we lookup through node #2, we get $10000
Node 3
{otan,$10000,...}
jas?

Eventual Consistency

Nodes will eventually return the most "up-to-date" value.

(more concretely, we're throwing ACID out the window)

On Partitioning, it is Available but not Consistent



Eventual Consistency is a **trade-off...**

Possibly acceptable:

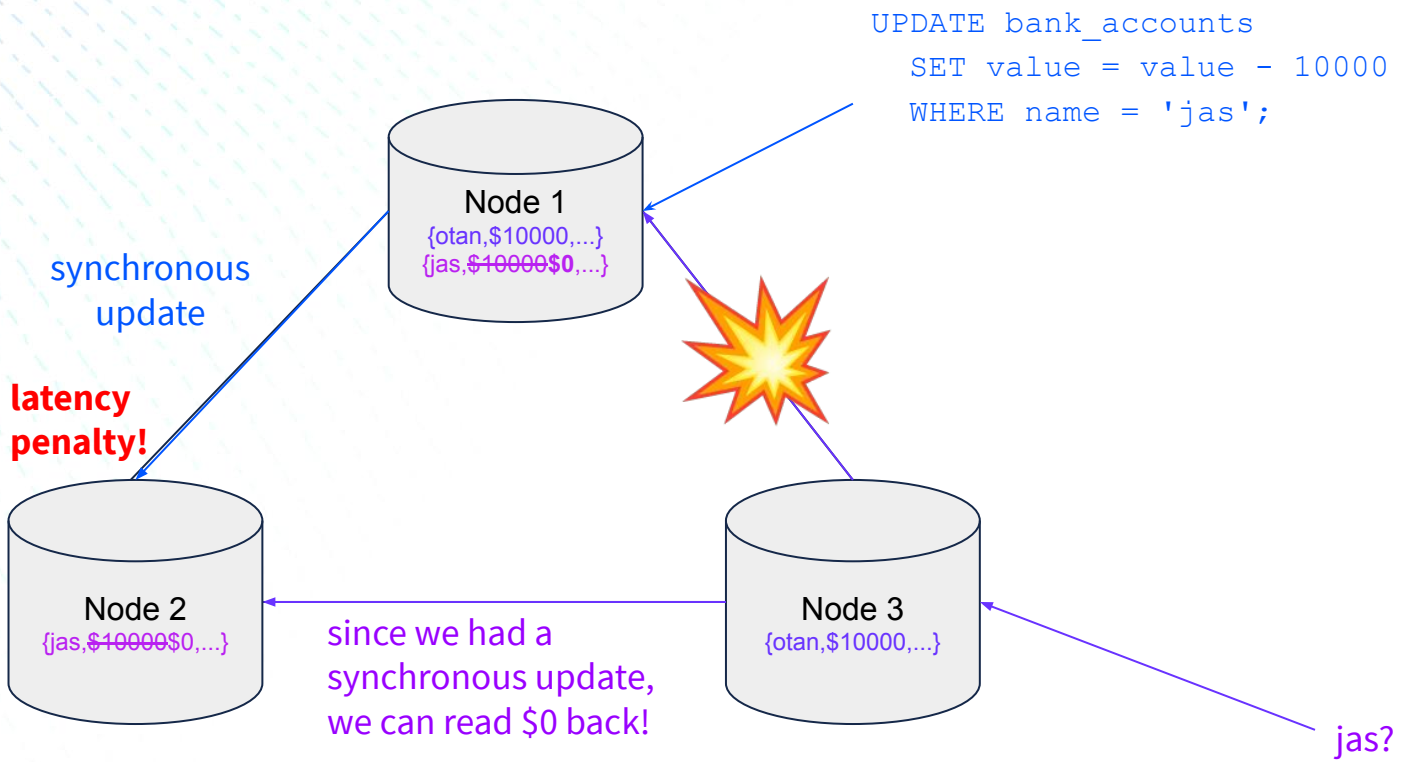
- News Feeds
- Social Media Comments
- Website Reviews

Not acceptable:

- Finance, e.g. bank accounts balances, portfolio worth, transactions
- Bookings
- Updating a file
- anything where ACID matters!

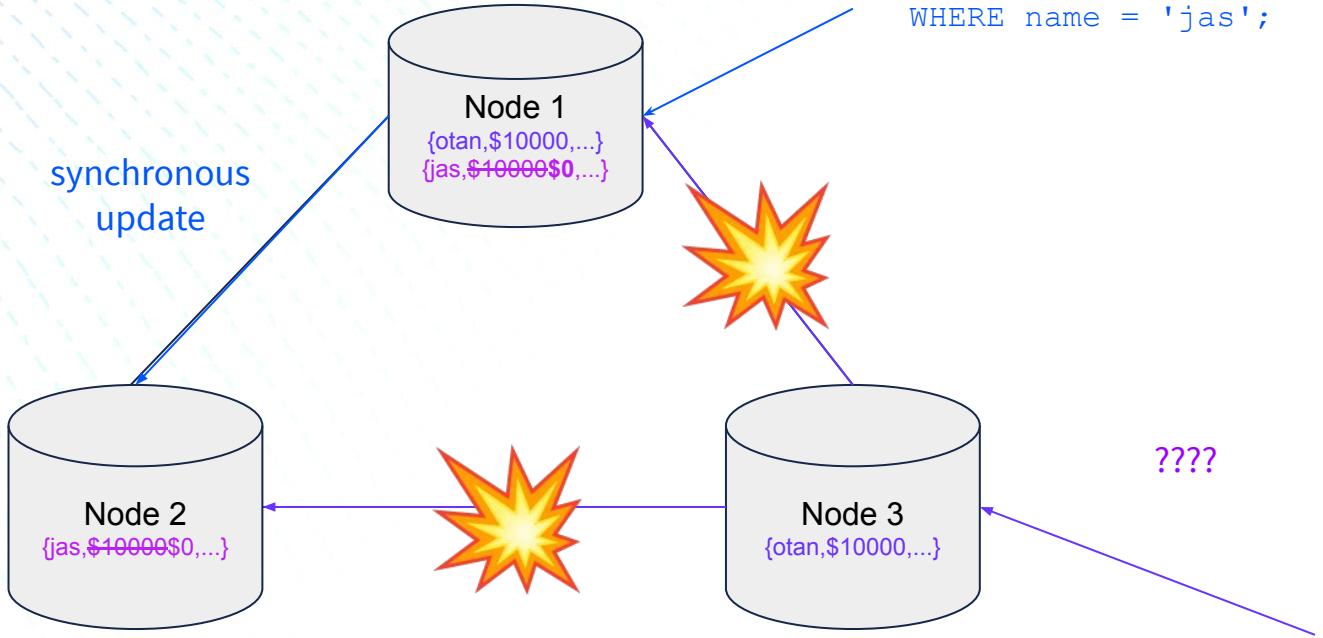
otan opinion: your brain will hurt when dealing with eventual consistency with business critical applications!

Can we get better consistency? What if we do synchronous updates?



What if two nodes are down?

```
UPDATE bank_accounts
  SET value = value - 10000
  WHERE name = 'jas';
```

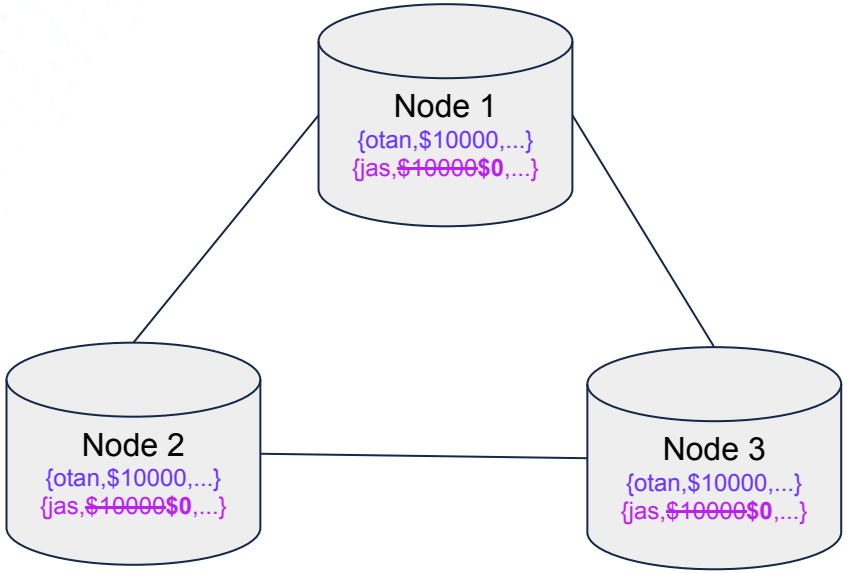


if we reject traffic, we are unavailable but consistent.
if we return NULL, we are inconsistent but available!

Question 1: how many nodes do I synchronise writes to?
Question 2: can I use synchronisation to trade availability for consistency?

How many nodes to synchronise writes to?

- One node (two nodes total)
  - survive one node being down but still have strict consistency on the other node
  - in a 3 node cluster, if two nodes are down I can choose to be inconsistent or unavailable
- Two nodes (three nodes total)
  - survive two nodes being down but still have strict consistency on the third node
  - in a 3 node cluster, if I synchronise **all** nodes, I have **strict consistency** (using 2PC) but lose **availability on writes!**
- **...what if I have five nodes in my system?**

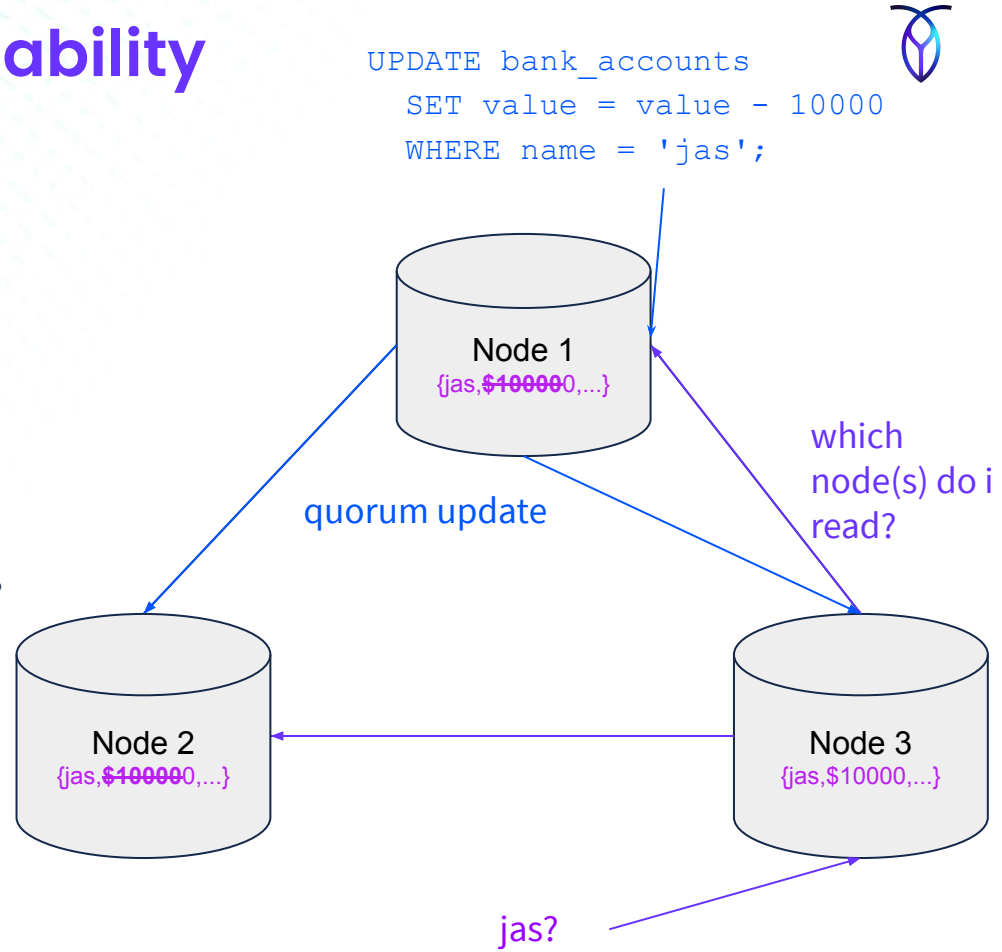


A Happy Medium: a quorum (majority) of nodes

corollary: we must have at least 3 nodes to have a clear majority

Read consistency vs availability

- How many nodes should you read for data? Which node(s) should you read off?
- The trade-offs:
  - Reading only one gives you lower latency but can be inconsistent under certain replication schemes.
  - Reading multiple/a consensus of nodes heighten latency and mean unavailability, but gives stronger consistency



tl;dr: can we swap availability for consistency?

yes, if the some portion of nodes do not respond, do NOT serve traffic.

Make your trade-off with Apache Cassandra!

Write consistency levels

This table describes the write consistency levels in strongest-to-weakest order.

Write Consistency Levels

Level	Description	Usage
`ALL`	A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition.	Provides the highest consistency and the lowest availability of any other level.
`EACH_QUORUM`	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in each datacenter.	Used in multiple datacenter clusters to strictly maintain consistency at the same level in each datacenter. For example, choose this level if you want a write to fail when a datacenter is down and the `QUORUM` cannot be reached on that datacenter.
`QUORUM`	A write must be written to the commit log and memtable on a quorum of replica nodes across *all* datacenters.	Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Use if you can tolerate some level of failure.
`LOCAL_QUORUM`	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in the same datacenter as the coordinator. Avoids latency of inter-datacenter communication.	Used in multiple datacenter clusters with a rack-aware replica placement strategy, such as NetworkTopologyStrategy, and a properly configured snitch. Use to maintain consistency locally (within the single datacenter). Can be used with SimpleStrategy.
`ONE`	A write must be written to the commit log and memtable of at least one replica node.	Satisfies the needs of most users because consistency requirements are not stringent.
`TWO`	A write must be written to the commit log and memtable of at least two replica nodes.	Similar to `ONE`.
		datacenters if an offline node goes down.
`ANY`	A write must be written to at least one node. If all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, an `ANY` write is not readable until the replica nodes for that partition have recovered.	Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability.

(source)

This applies to PostgreSQL replication too!

26.2.8. Synchronous Replication

PostgreSQL streaming replication is asynchronous by default. If the primary server crashes then some transactions that were committed may not have been replicated to the standby server, causing data loss. The amount of data loss is proportional to the replication delay at the time of failover.

Synchronous replication offers the ability to confirm that all changes made by a transaction have been transferred to one or more synchronous standby servers. This extends that standard level of durability offered by a transaction commit. This level of protection is referred to as 2-safe replication in computer science theory, and group-1-safe (group-safe and 1-safe) when `synchronous_commit` is set to `remote_write`.

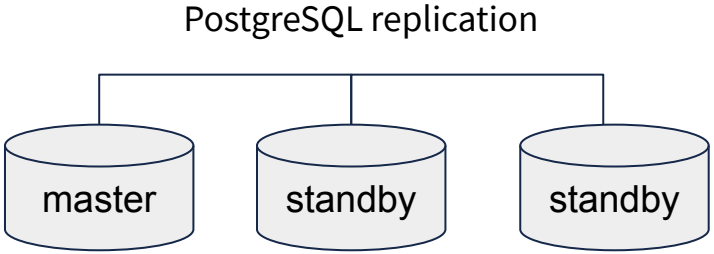
When requesting synchronous replication, each commit of a write transaction will wait until confirmation is received that the commit has been written to the write-ahead log on disk of both the primary and standby server. The only possibility that data can be lost is if both the primary and the standby suffer crashes at the same time. This can provide a much higher level of durability, though only if the sysadmin is cautious about the placement and management of the two servers. Waiting for confirmation increases the user's confidence that the changes will not be lost in the event of server crashes but it also necessarily increases the response time for the requesting transaction. The minimum wait time is the round-trip time between primary to standby.

26.2.8.3. Planning For Performance

Synchronous replication usually requires carefully planned and placed standby servers to ensure applications perform acceptably. Waiting doesn't utilize system resources, but transaction locks continue to be held until the transfer is confirmed. As a result, incautious use of synchronous replication will reduce performance for database applications because of increased response times and higher contention.

26.2.8.4. Planning For High Availability

`synchronous_standby_names` specifies the number and names of synchronous standbys that transaction commits made when `synchronous_commit` is set to `on`, `remote_apply` or `remote_write` will wait for responses from. Such transaction commits may never be completed if any one of synchronous standbys should crash.



(source)

Concluding Remarks

tl;dr

Covered Concepts

- ACID
- Leaders and Replicas
- CAP Theorem
- Two-phase commit (2PC)
- Eventual Consistency
- Quorums

Lessons

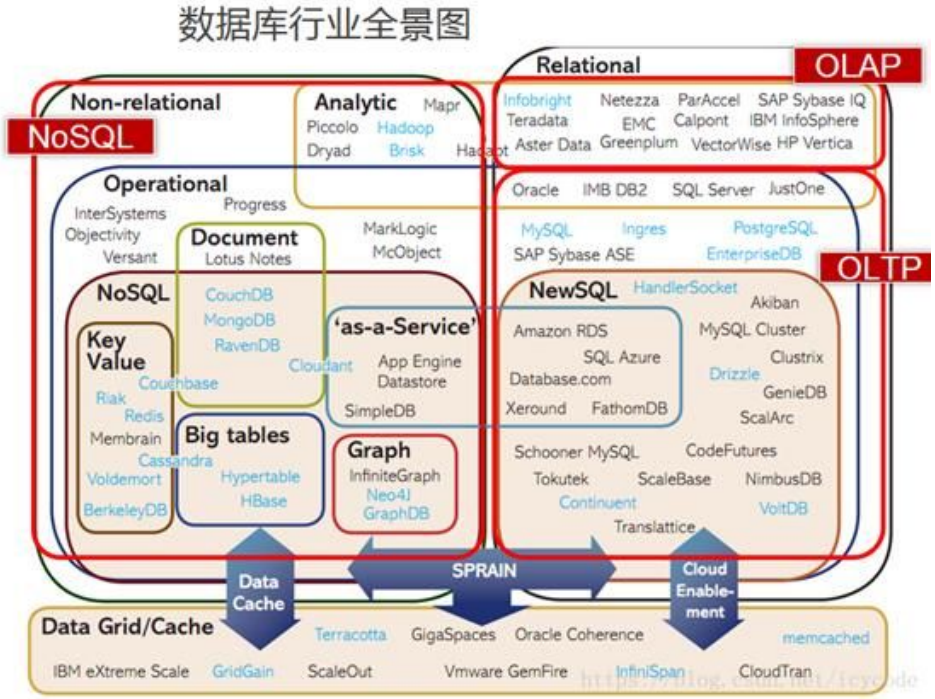
- A single database machine can fail (it's not web-scale!)
  - 💩 happens
  - **avoid single points of failure**
- the CAP theorem
  - choose two of: consistency, availability and partitioning.
  - for distributed databases, we want to survive partitioning, so we must sacrifice one of the other two
  - latency is a cost to factor in too
- As a result, using software developers must make a **tradeoff** when deciding on their database. There's no silver bullet*
- **Distributed database systems are hard**

yes, it's worth using the same meme again

Much more out there with different trade-offs...

- Document stores (e.g. MongoDB)
- Key Value stores (e.g. redis)
- Graph-based databases (e.g. Neo4j)
- Columnar database stores (e.g. HBase)
- **NewSQL databases (e.g. CockroachDB!)**
- Search databases (e.g. ElasticSearch)
- …. most of which are in jas' course notes if you're interested….

though may work and look different, the trade-offs between the concepts CAP are going to be the same!



Distributed DBMSs are on the up

364 systems in ranking, March 2021

Rank Mar 2021	Rank Feb 2021	Rank Mar 2020	DBMS	Database Model	Score Mar 2021	Score Feb 2021	Score Mar 2020
1.	1.	1.	Oracle	Relational, Multi-model	1321.73	+5.06	-18.91
2.	2.	2.	MySQL	Relational, Multi-model	1254.83	+11.46	-4.90
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1015.30	-7.63	-82.55
4.	4.	4.	PostgreSQL	Relational, Multi-model	549.29	-1.67	+35.37
5.	5.	5.	MongoDB	Document, Multi-model	462.39	+3.44	+24.78
6.	6.	6.	IBM Db2	Relational, Multi-model	156.01	-1.60	-6.55
7.	7.	↑ 8.	Redis	Key-value, Multi-model	154.15	+1.58	+6.57
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	152.34	+1.34	+3.17
9.	9.	↑ 10.	SQLite	Relational	122.64	-0.53	+0.69
10.	↑ 11.	↓ 9.	Microsoft Access	Relational	118.14	+3.97	-7.00
11.	↓ 10.	11.	Cassandra	Wide column	113.63	-0.99	-7.32
12.	12.	↑ 13.	MariaDB	Relational, Multi-model	94.45	+0.56	+6.10
13.	13.	↓ 12.	Splunk	Search engine	86.93	-1.61	-1.59
14.	14.	14.	Hive	Relational	76.04	+3.72	-9.34
15.	↑ 16.	15.	Teradata	Relational, Multi-model	71.43	+0.53	-6.41
16.	↓ 15.	↑ 23.	Microsoft Azure SQL Database	Relational, Multi-model	70.88	-0.41	+35.44
17.	17.	↓ 16.	Amazon DynamoDB	Multi-model	68.89	-0.25	+6.38
18.	↑ 19.	↑ 21.	Neo4j	Graph, Multi-model	52.32	+0.16	+0.54
19.	↓ 18.	↑ 20.	SAP Adaptive Server	Relational, Multi-model	52.17	-0.07	-0.59
20.	↑ 21.	↓ 18.	SAP HANA	Relational, Multi-model	51.00	+0.77	-3.27

(source)

Distributed database systems are hard

But that's what makes it fun!*

*unless you're a startup and you
just want to build a product
without worrying about the infra
(you know, most companies)

Join us for Part II: CockroachDB Architecture

Questions? You can contact me - otan@cockroachlabs.com