Database Storage systems

July 2023 (Fail stop only)

Goal

- Learn building blocks of DB software systems
- Have a top level understanding of the popular storage systems
- Scope
  - Once a query is parsed, validated, authenticated, converted to plan, then the execution of that query kicks-in, We are focusing on the execution.
  - We work with mathematical abstractions than queries.
  - How to enable transactions on the planet scale DB (not public ledger like blockchain).
  - We primarily focus on simple and complex read write patterns with ACID guarantees
  - Not considering optimisation of planning of complex joins, group by… (They are fundamentally compute ops on the data)

Database overall components

Client server connection

Client connect to server using TCP protocol

Check username & password

Server checks whether the username and password are valid, if they are not valid just close the connection

Client sends SQL query

Client sends simple query like 'SELECT … FROM table JOIN WHERE … GROUP BY … ORDER BY … HAVING …'

Query → structures

Convert the query string to more tree+array like data structure, if it can't understand the query, throw error

Check access of query

Now check if it has access to specific columns, for DML/DDL statements

Run the query and get the results
(with optimisation of plan execution)

Database storage types

- Single machine with ACID properties
  - → MySQL, Postgres, Aurora, AlloyDB, MongoDB
  - → Neo4j, Graph use cases
- For in memory cache
  - → Redis, Memcache
- For config management
  - → Zookeeper, etcd
- For distributed data storage AP
  - → Cassandra, ScyllaDB, DynamoDB
- For distribute data storage with CP
  - Yugabyte, Spanner

Flow

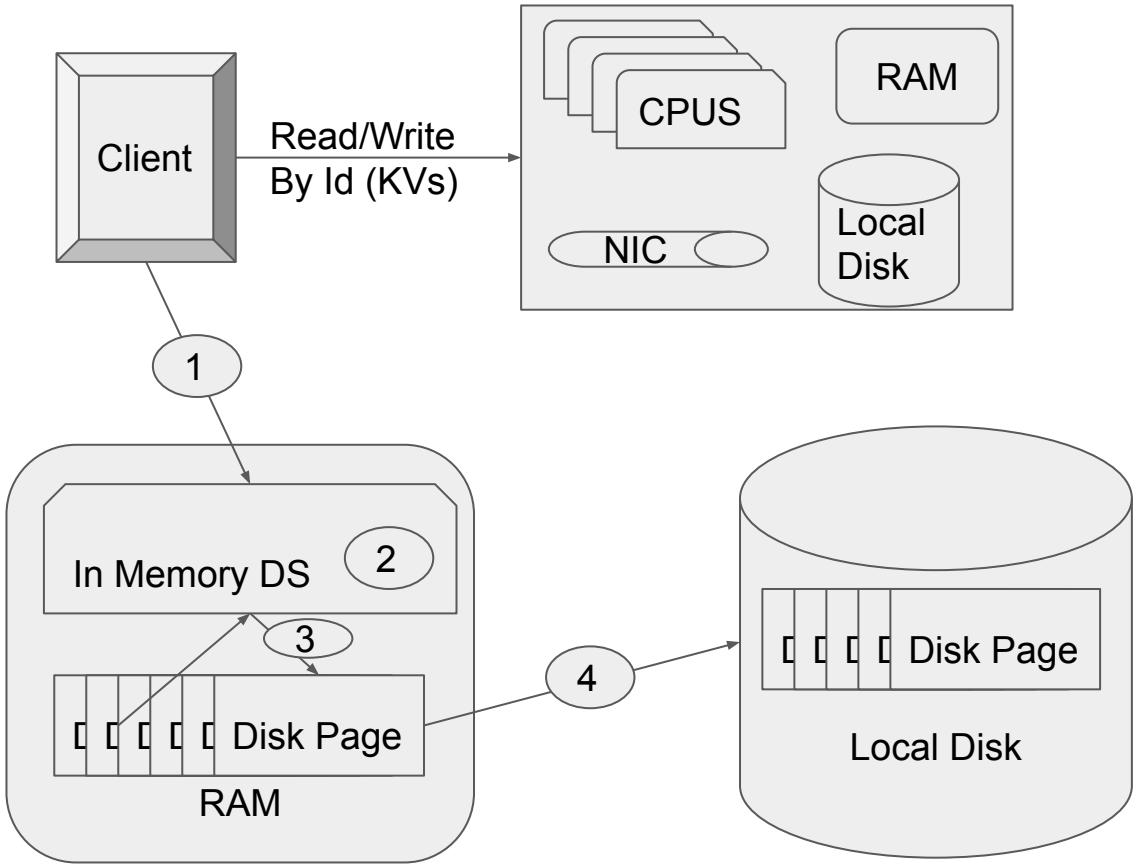
- Build a key value system, along with durability
- Add secondary index to system
- Handling restart of single machine
- Improve the fault tolerance
- Improve the read/write performance
- Use multiple machine to support more scale and fault tolerance
- Making the system consistent, available too
- Enable transactions with ACID properties
- Enable transactions with cross shards (scaled readwrites)
- Extreme performance types
    - Partition getting used a lot
    - Key getting used a lot
    - Improve performance with custom hardware

Storing rows in databases, relation with Key values

- Every row in table usually have primary key, if not databases create one internally
- Each row will be compressed into two parts (primary key → byte[], remaining row → byte[]), so it'll be key value pair with (byte[], byte[])
  - In memory parts usually differ, some implementations use (string, string)
- Store the data in memory with KV pair, and lookup KV easily
  - Hashmap → O(1) time complexity, but range queries are expensive
  - Btree → O(logn) time complexity, with range queries, but balancing requires locking many parent nodes
  - Skiplist → O(logn) randomised data structure, optimal for locking
- We usually say (x=A), means row has primary key = x, remaining columns data is compressed and denoted with A.

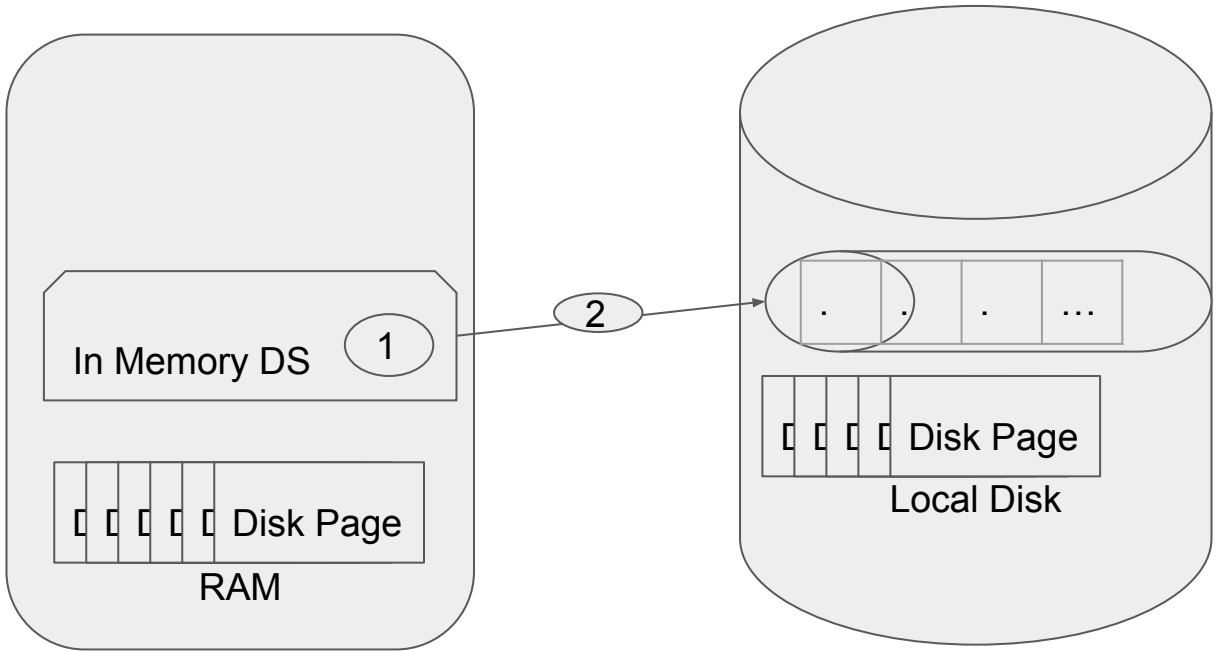
Key value system

- Maintain in memory data structures to maintain key values, regularly write them to Disk
- Possible options for In Memory
  - HashMap (point lookup)
  - Btree / B+tree (range)
  - Skiplist (Locking)
- Possible options for storage
  - B+ tree
  - LSM tree
    - SSTable
    - Ressi (column oriented)

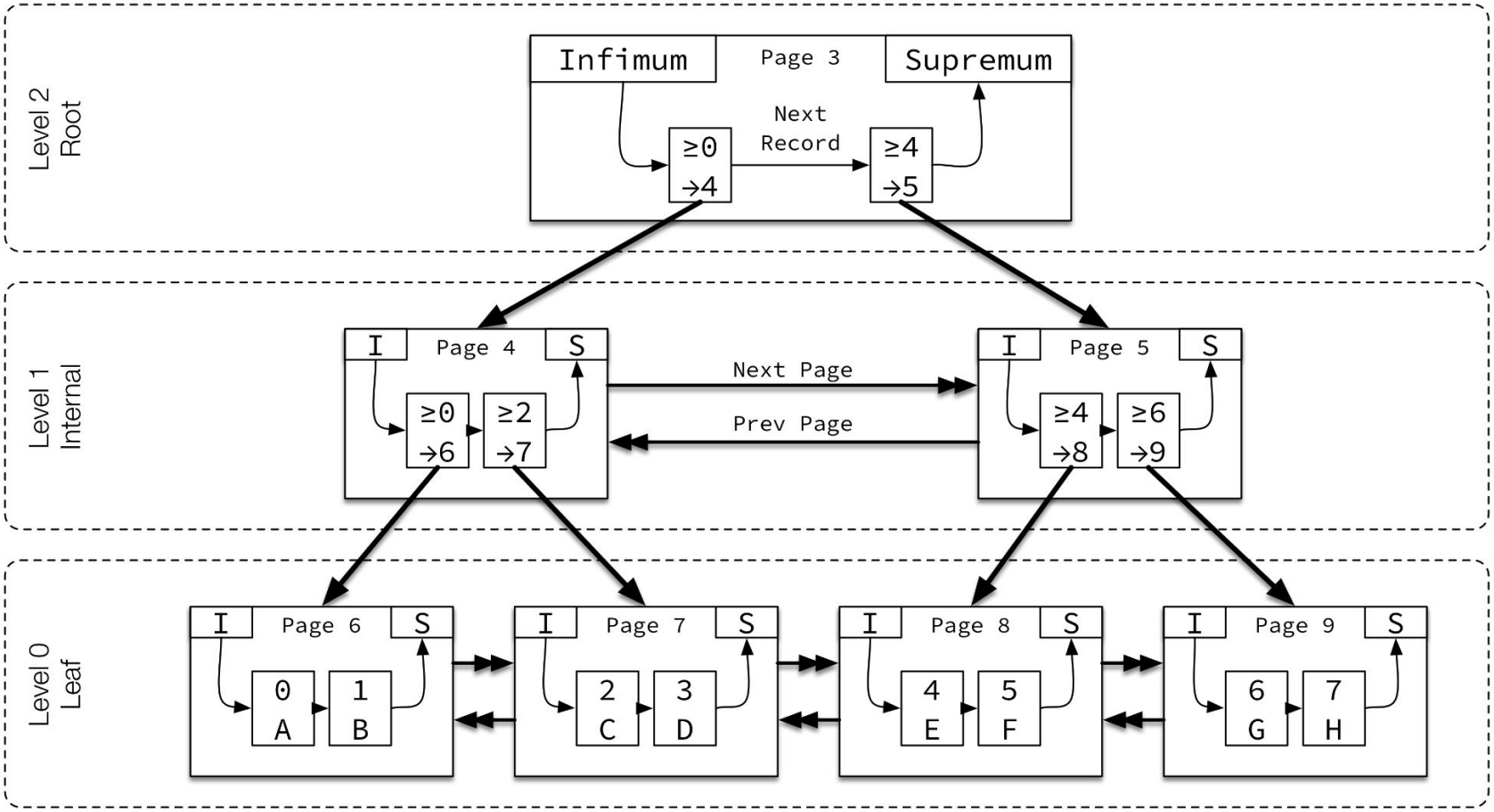


Handle restart of server (first step of fault tolerance)

- We can’t write disk page for every write, Disk bottleneck
  - Disk → 100 access/sec
  - SSD → 5K access/sec
  - Still not good enough for writes, sometimes, we need to find the B+ tree leaf node to insert, might be having 5-20 access/req
- Idea is to write to disk in sequential order, just describing the operation(Ex: INSERT row), called log, avoid searching and writing the main disk pages
  - 1000X faster than random read/write
  - Only 1 access/req



B+Tree Structure



Summary of B+tree

- O(log(n)) structure representing a Map<K, V>
  - Where K could be primary key or unique row id
  - V is the complete row except the primary key
- B+tree often stores, extra information like
  - Row_id → unique id of row in case of no primary key
  - Txn_id → last txn id, updated the row
  - Rollback_ptr_id → last version of the row value, to recover in case of TXN ROLLBACK
  -
- In the definition we often see “BTREE”
- Sometimes, to create a secondary index, we create another B+tree, we’ll see another database

Flows understanding

- Try to understand the simplified flows step by step (executing single instructions with ACID properties)

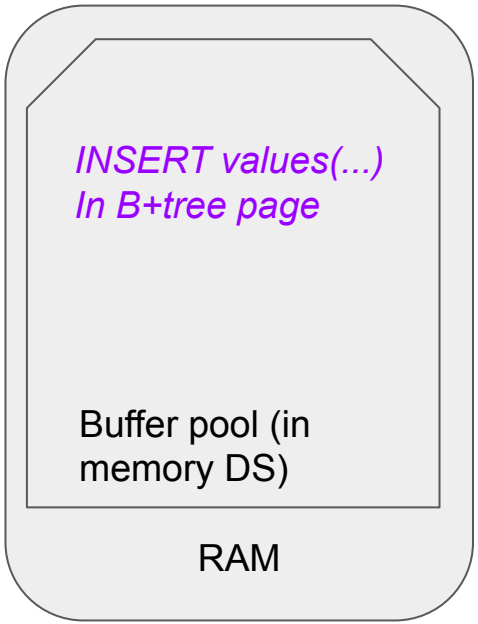
- Single thread, primary key
  - INSERT, SELECT, UPDATE, DELETE,
  - SELECT on non primary key
- Single thread, secondary key
  - INSERT, then SELECT
  - UPDATE and DELETE

Writes handling

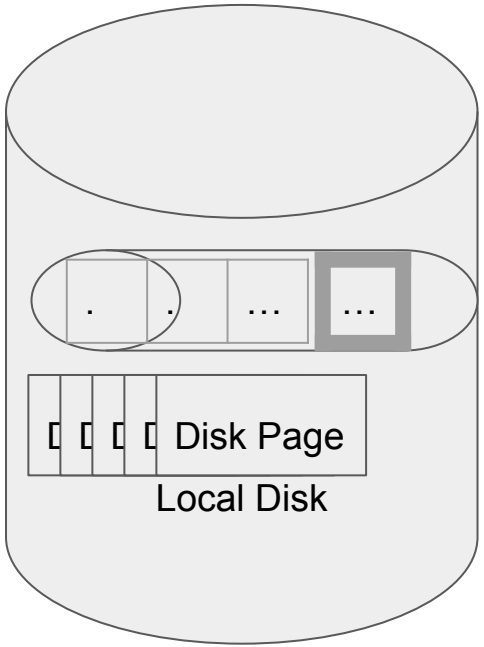
- How does it work for bulk inserts ?
- What happens if MySQL system restarted after generating Auto INCREMENT key ?
- What if buffer pool is full ?

What happens to disk usage for fresh created server and table ?

- Client sends INSERT
- Pre insert steps
  - Generate Auto inc key



Write to the redo log with sequential write



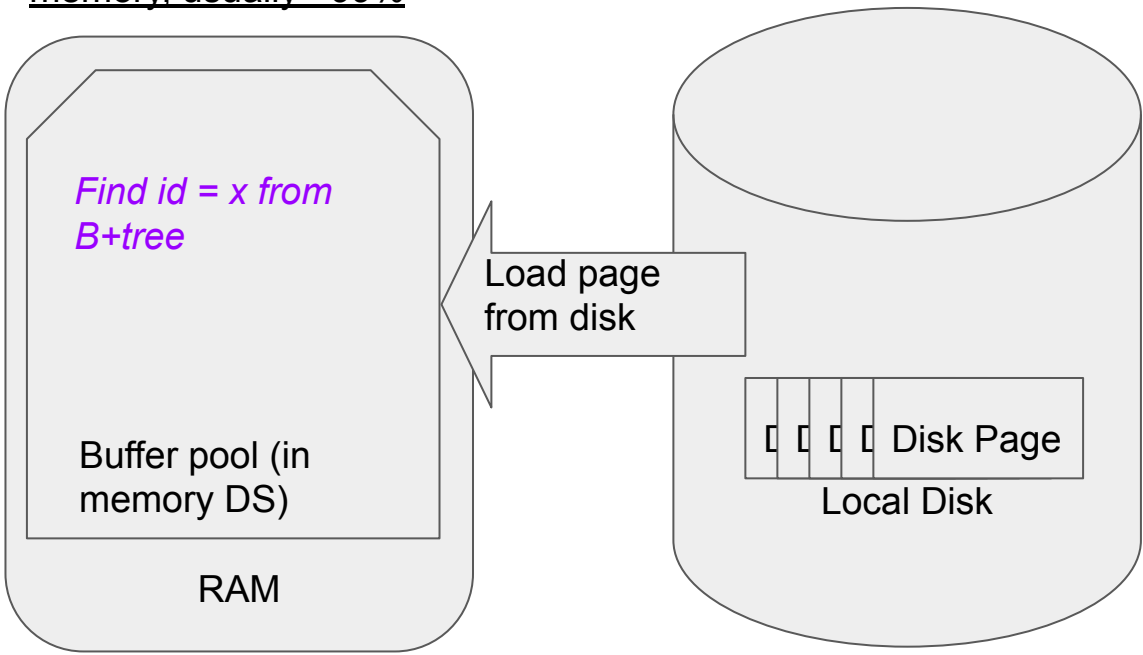
Flush to disk before returning to client

Read from Database

1. SELECT … FROM TABLE WHERE id = x (primary key case)

- Can we improve the O(log(n)) search time from the Buffer pool (in memory DS)
- Yes, enable adaptive_hash cache (maintains hashtable with primary keys)

Buffer hit ratio is, how many reads from memory, usually >99%

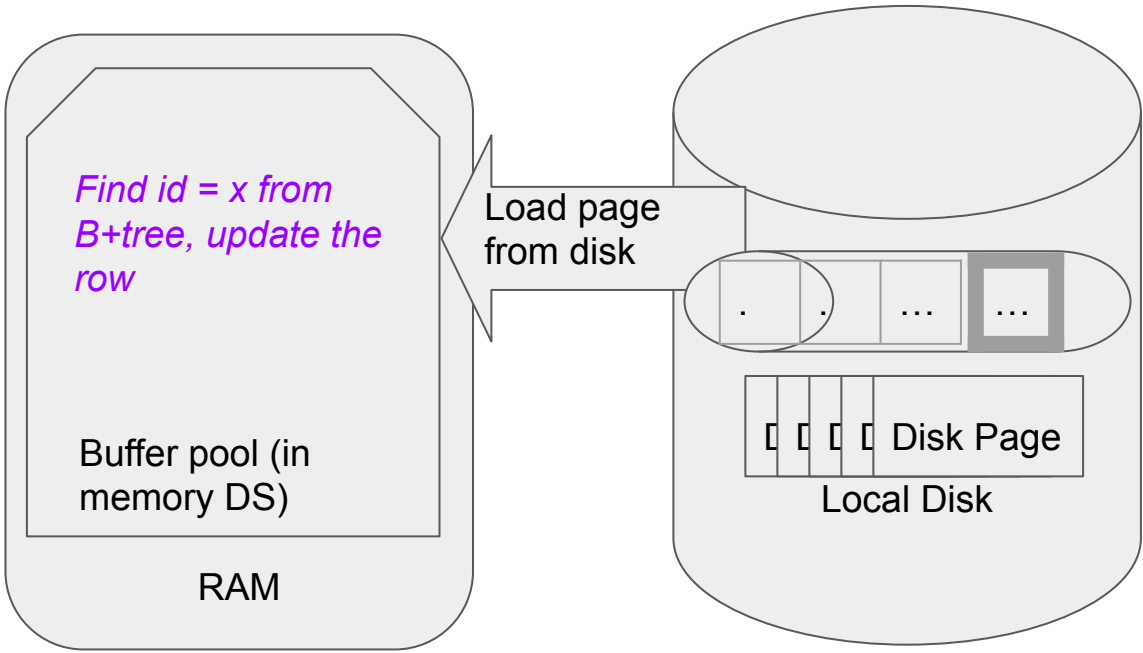


UPDATE to row

1. UPDATE TABLE SET … WHERE id = x (primary key case)

1. How do we handle deletes ?

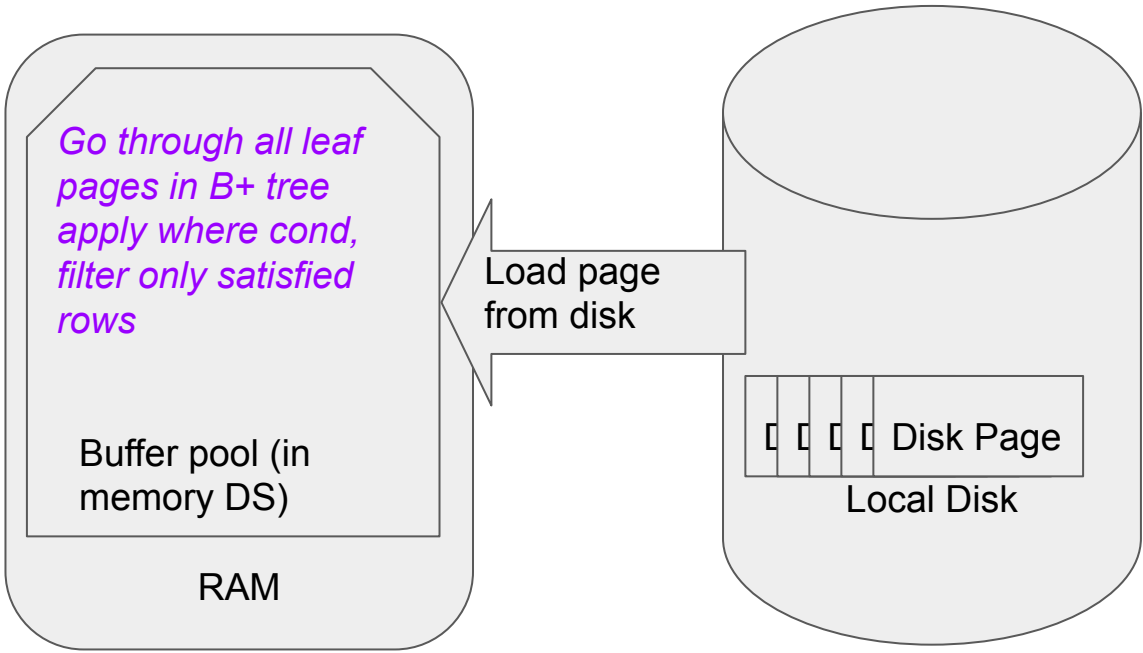
- Persist in the WAL log (redo log), before returning



Read from Non primary key

SELECT … FROM TABLE WHERE age > 30 and department = ‘xyz’

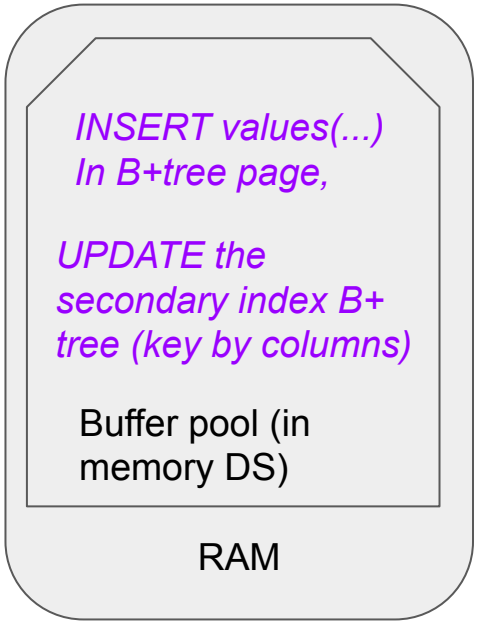
This is called table scan,often needs to be avoided



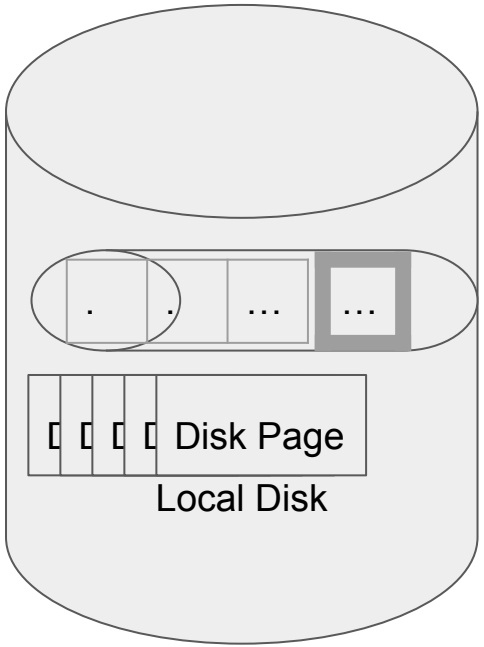
Secondary index INSERT

- Client sends INSERT
- Pre insert steps
  - Generate Auto inc key

- How does secondary b+ tree page looks like ?
- How many additional updates we need to do ?
- How many buffer pool searches we need to do ?



Write to the redo log with sequential write



Flush to disk before returning to client

- How many WAL entries will be created if the table has 1 primary key, 2 secondary keys ?

Secondary index B+ tree

- Key is simply column of table on which we created index
- Value is array of primary keys where this column particular value is present

Search by index column first searches this “Index (B+tree)”, find primary keys, then searches in B+ tree

Range query on secondary index, will generate non sequential primary keys, so in main B+ tree you need to search many leafs, those are not consecutive

Q: How does it handle secondary index with multiple columns ?

A: Separates by “-” ?

Q: For multiple columns how does it works on range queries

A: It searches same way like value but does range search, supports partial range searches too

Secondary index - Examples

Table (Id, Name, Age) → 100 rows, with Index (Name, Age) name_age_idx

40 diff names, 30 diff ages

(Archit-25) → [46, 87] (key → Value mapping)

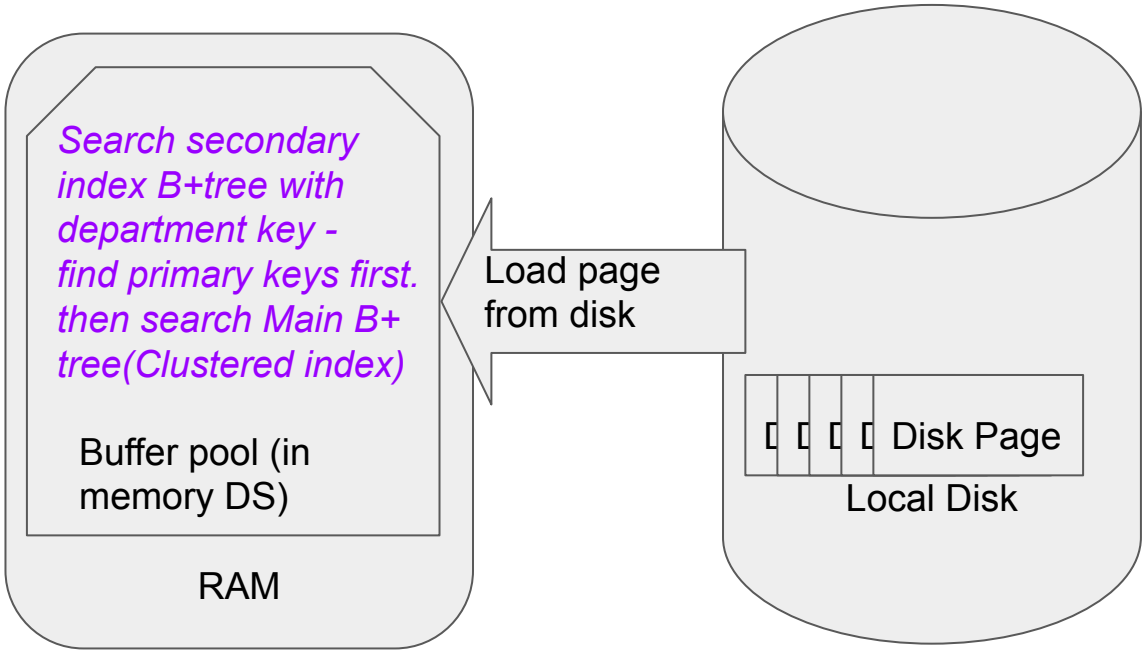
Archit-25 < Archit-30

- WHERE Name = Archit and age >= 25 and age <= 30
  - [(Archit-25), (Archit-30)]
- WHERE age = 25 and name >= ‘Araa’ and name <= ‘Arha’
  - [(Araa-*), (Arha-*)]
- WHERE age = 25
  - Won’t be able to search on the index [(*-25), (*-25)] ~ [*]
  - The solution is create new index on ‘age’ column

Read from secondary index

SELECT … FROM TABLE WHERE department = ‘xyz’

we avoided scanning whole table now!

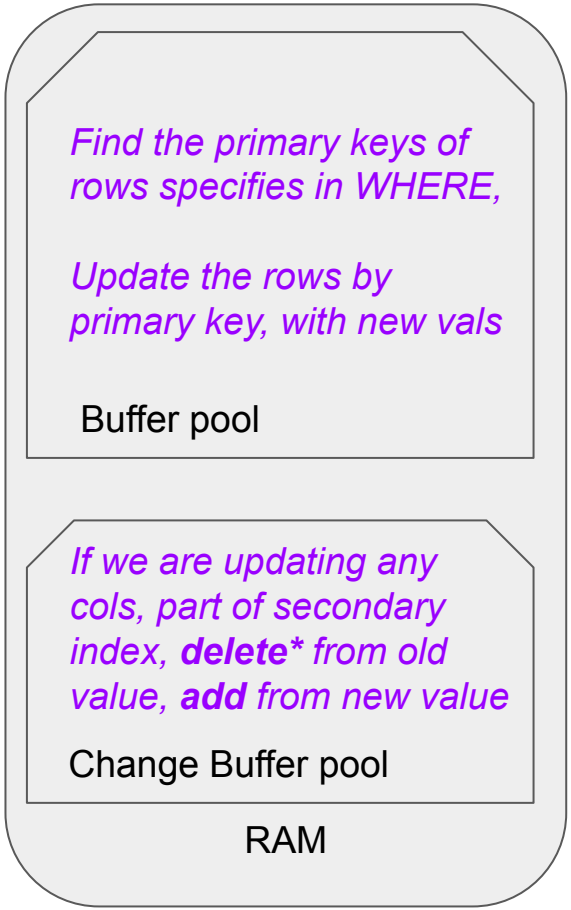


- How does range scans on `department` column works ?
- What’s impact on latency ?

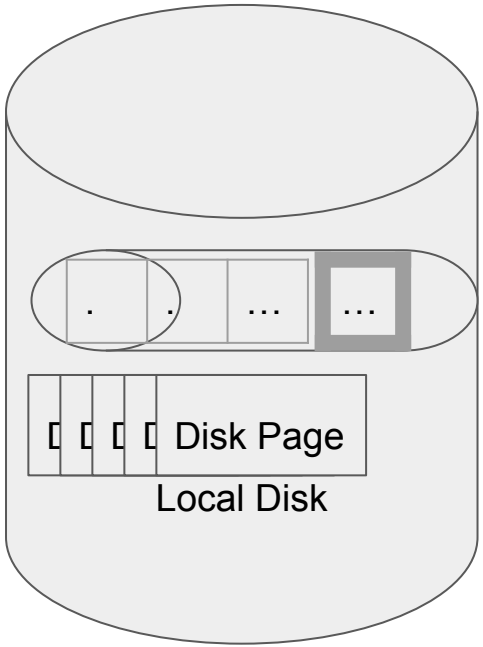
Secondary index - Effect on Update

- UPDATE table SET cols WHERE condition

- How does these deletes executed in the secondary index ?



Write to the redo log with sequential write



Flush to disk before returning to client

Summary of Single thread and single statements

- Have one B+tree (clusterted index) to store the rows, since this is a Map<>, it’s keyed by primary key or row id
- INSERT/UPDATE/DELETE or SELECT statements happen on the B+tree, often loads the data from disk to RAM (called buffer pool in MySQL)
- If you add secondary index, usually we create another Map<> (B+tree), keyed by secondary index column(s), value is array of primary keys*
- If we make on update, first finds the primary keys, updates the main B+tree, then do delete + add to secondary indexes (in case if we are updating the columns on secondary index)

Support of Multiple cores/threads

- UPDATE/DELETE statements
  - LOCK the primary keys of the rows updating, secondary indices will be locked as per requirement.
  - This called is called X lock in MySQL
- What happens if we are updating/deleting half of the table ?

Multi threads SELECT

- Simple solution is LOCK like UPDATE
  - It’ll impact the performance in case we want to read the same row by multiple threads
- We need to allow to read by multiple threads,
  - We’ll create a lock called S
  - All select statements will acquire this lock before SELECT*
- We prevent the UPDATE / SELECT by conflicting the locks
  - If we have an UPDATE it should have X lock
  - If we have SELECT(s) they should have S lock
  - If we have X lock, only X lock is possible on row
  - Else we can have as may S locks we want.

For SELECT, locking only row is not good enough - why ?

- If we `SELECT … FROM table WHERE id between (10 and 20)`
  - Assume we have rows with ids (10, 15, 20)
  - Txn ‘A’ run this query **2 times**, but in b/n the two SELECTs another txn INSERT the data at 17
  - How to prevent the INSERTs b/n two rows ? Or how to prevent INSERT in the **GAP**
- **Gap lock (++)**
  - Lock on pointer from previous row (imagine linkedlist), to prevent insert before the ROW.
  - A Gap lock can be taken by multiple txns, they don’t conflict with each other.
  - You can take GAP lock on row, without taking lock on ROW (S or X Lock).

Multi threads for INSERT

- MySQL gap locks prevent INSERTs
- If another Txn want to insert the data it’ll do INSERT_INTENTION locks
  - ("X,INSERT_INTENTION",WAITING,supremum pseudo-record)
  - It’ll be in waiting stage
- As soon as it gets the chance it’ll do the INSERT
- Idea is to support not keeping the INSERT waits forever.

Transactions

- A great property of databases, also solving the problem of SQL statements
- Apply a programming level logic to databases, that can span multiple statements.
- For performance reasons, they need to run concurrently.
- Yet provides
  - Atomicity → Execute all or none
  - Consistency → For all clients, ability to show only meaningful data, especially avoid showing partial updates.
  - Isolation → Need to execute the statements, as if it's the only set! Idea is that one txn shouldn't see the effects of execution of other, the data that's supposed to be rolled back, or interim data…
  - Durability → Once committed, it's guaranteed to be stored in MySQL forever*

Trade Off performance with isolation

- Sometimes, it’s okay to relax some constraints (based on application requirement)
- For example, showing stock of the items in the product view page
  - It’s okay to read little older, you could read
  - READ UNCOMMITTED → read the data written by another transaction, might rollback later as well.
  - READ COMMITTED → read the data written + committed by another transaction.
  - You can get much faster response with this, better scaling of database

Example of READ UNCOMMITTED/COMMITTED

Transaction 1 (Product view page showing stock count as quantity)

```
BEGIN;
SELECT quantity FROM products WHERE id = 1;
-- retrieves 20
```

Transaction 2 (cancel order)

```
BEGIN;
UPDATE products SET quantity = 21 WHERE
id = 1;
-- no commit here
```

```
SELECT quantity FROM products WHERE id = 1;
-- READ UNCOMMITTED retrieves 21 (dirty read)
-- READ COMMITTED retrieves 20 (dirty read has been avoided)
COMMIT;
```

```
ROLLBACK;
```

Limitation of READ COMMITTED

Transaction 1

```
BEGIN;
SELECT quantity FROM products WHERE id = 1;
-- retrieves 20
```

Transaction 2

```
BEGIN;
UPDATE products SET quantity =
21 WHERE id = 1;
COMMIT;
```

```
SELECT quantity FROM products WHERE id = 1;
-- READ UNCOMMITTED retrieves 21 (non-repeatable
read)
-- READ COMMITTED retrieves 21 (non-repeatable
read)
COMMIT;
```

Possible problems & solution

- Same query could return the different results could be inconsistent
- Imagine bank accounts, debit and credit transactions, this could lead to problems,
- Even though it looks contrived example, when you query by secondary indices or other columns, this is quite possible scenario.
- Solution
  - Do REPEATABLE READ

REPEATABLE READ

Transaction 1

```
BEGIN;
SELECT quantity FROM products WHERE id = 1;
-- retrieves 20
```

Transaction 2

```
BEGIN;
UPDATE products SET quantity =
21 WHERE id = 1;
COMMIT;
```

```
SELECT quantity FROM products WHERE id = 1;
-- READ UNCOMMITTED retrieves 21 (non-repeatable
read)
-- READ COMMITTED retrieves 21 (non-repeatable
read)
-- REPEATABLE READ retrieves 20 (non-repeatable
read has been avoided)
COMMIT;
```

Are we done, no not yet! Example

Transaction 1

```
BEGIN;
SELECT name FROM products WHERE quantity > 17;
-- retrieves Phoskill and Dost
```

Transaction 2

```
BEGIN;
INSERT INTO products VALUES (3,
'Spruce', 26);
COMMIT;
```

```
SELECT name FROM products WHERE quantity > 17;
-- READ UNCOMMITTED retrieves Phoskill, Dost and Spruce
(phantom read)
-- READ COMMITTED retrieves Phoskill, Dost and Spruce (phantom
read)
-- REPEATABLE READ retrieves Phoskill, Dost and Spruce
(phantom read**)
-- SERIALIZABLE retrieves Phoskill and Dost (phantom read has
been avoided)
  COMMIT;
```

MySQL specials

- READ UNCOMMITTED is reading dirty values, could rollback later
- READ COMMITTED is reading only committed values
  - Lock only rows which are selected by query, (X,REC_NOT_GAP)
  - You can’t update the rows from another transaction if you do SELECT … FOR UPDATE
- REPEATABLE READ is reading the same data
  - Lock rows and prevents insertions from other transactions (X)
  - You can’t update the rows from another transaction if you do SELECT … FOR UPDATE
  - You can’t add rows from another transactions as well.
  - This will prevent phantom reads* (special quality of MySQL)
- SERIALIZABLE is same as REPEATABLE read
  - If you do SELECT query without FOR SHARE, it’ll add that clause automatically

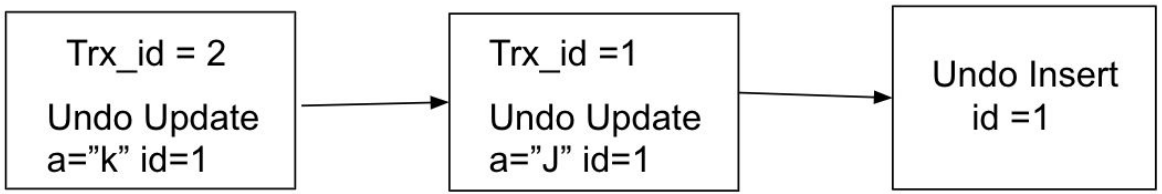
How to support all the isolation levels

- Isolation levels are by session, summary is we need to support all types of isolations in the system.
- For reading it’s simple, just lock the rows for read, if necessary block insertions with GAP locks
- For writes/updates
  - We need rollbacks
  - We need to read old reads (READ COMMITTED or REPEATABLE READS)
  - Many databases including MySQL uses MVCC, stores previous (multiple) versions, and use them for reading and writing

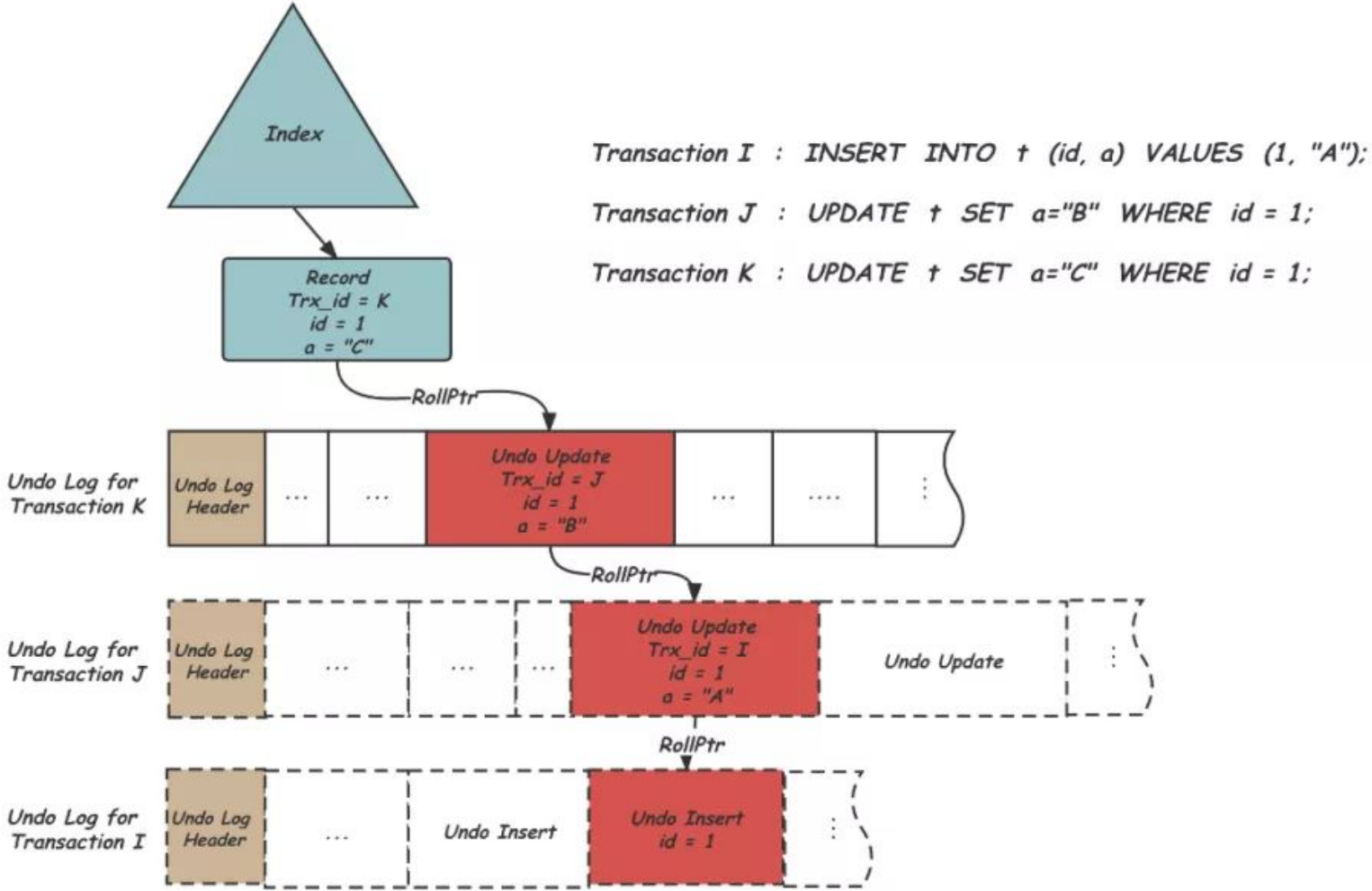
How to maintain multiple versions

- Option 1 → Read the existing record, copy to new place and update the new one in the B+tree, and add pointer the old version
- Option 2 → Read the existing record, append the new row in the B+ tree, add pointer to old row.
-

Option 1 is MySQL way, Option 2 is PostgreSQL way



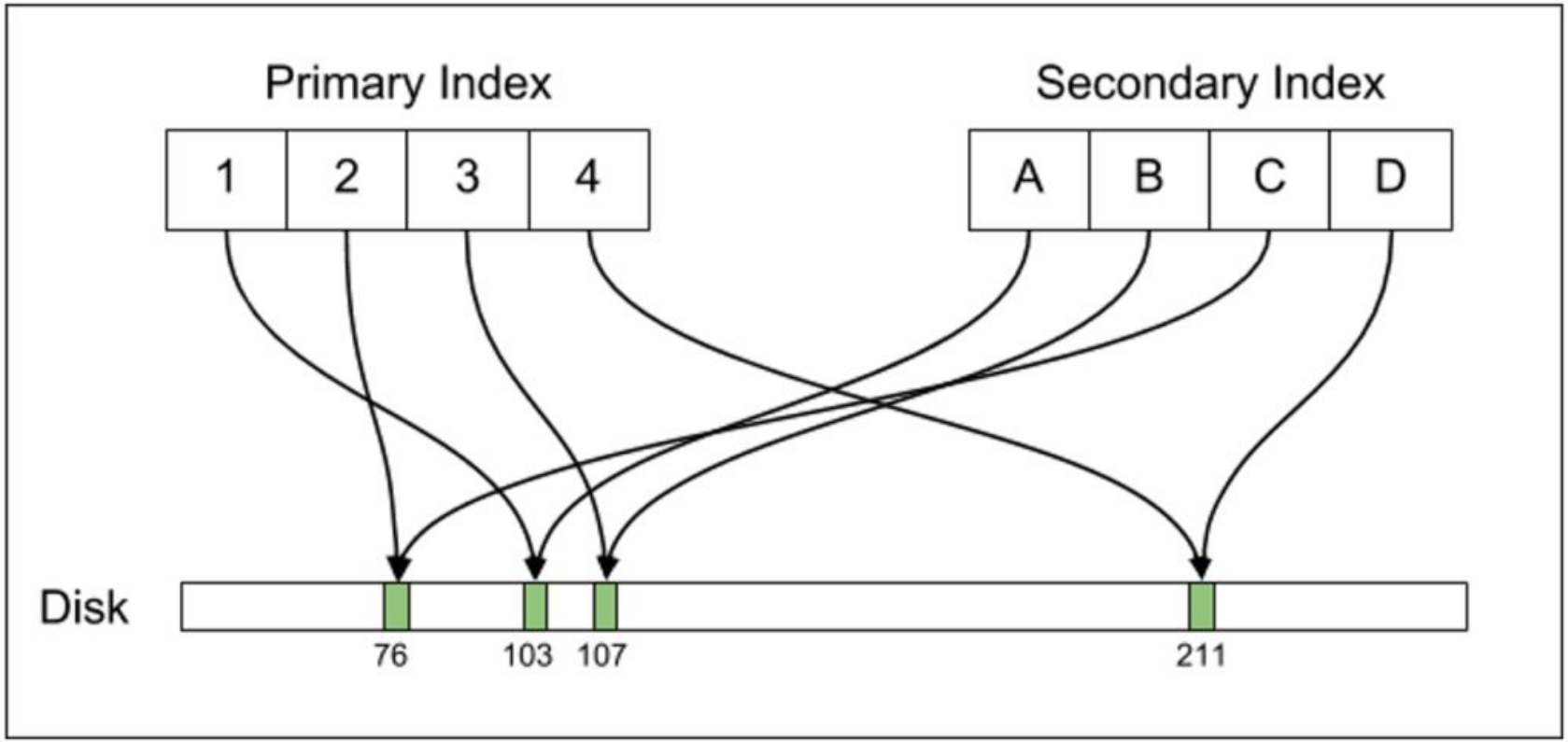
Undo Log Chain


Index
Record
Trx_id = K
id = 1
a = "C"
RollPtr
Undo Log for Transaction K
Undo Log Header
Undo Update
Trx_id = J
id = 1
a = "B"
RollPtr
Undo Log for Transaction J
Undo Log Header
Undo Update
Trx_id = I
id = 1
a = "A"
Undo Update
RollPtr
Undo Log for Transaction I
Undo Log Header
Undo Insert
Undo Insert
id = 1
Transaction I : INSERT INTO t (id, a) VALUES (1, "A");
Transaction J : UPDATE t SET a="B" WHERE id = 1;
Transaction K : UPDATE t SET a="C" WHERE id = 1;

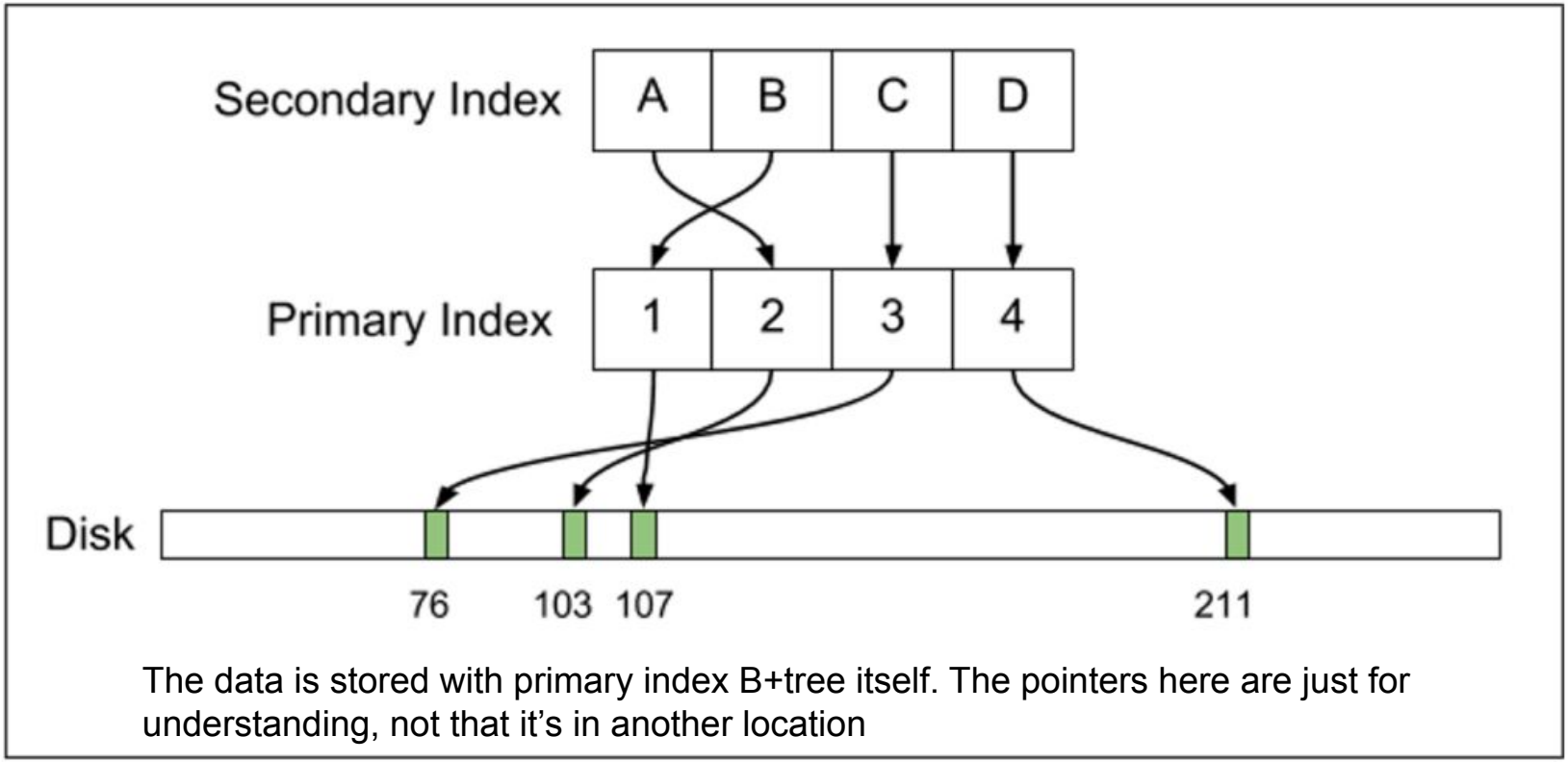
Update in PostgreSQL

ctid	prev	id	first	last	birth_year
A	null	1	Blaise	Pascal	1623
B	null	2	Gottfried	Leibniz	1646
C	null	3	Emmy	Noether	1882
D	null	4	Muhammad	al-Khwārizmī	780
E	null	5	Alan	Turing	1912
F	null	6	Srinivasa	Ramanujan	1887
G	null	7	Ada	Lovelace	1815
H	null	8	Henri	Poincaré	1854
I	D	4	Muhammad	al-Khwārizmī	770

PostgreSQL data structures (logical)



MySQL data structure (logical)



The data is stored with primary index B+tree itself. The pointers here are just for understanding, not that it's in another location

Impact on read writes on MySQL vs PostgreSQL (due to version management)

- SELECT by primary key
  - Find primary key page in B+tree, MySQL immediately returns the result,
  - but PostgreSQL needs to find the ctid and searches another B+tree (you can optimise this with CLUSTER command in PostgreSQL)
- UPDATE by primary key
  - Find primary key page in B+tree, MySQL updates the row and while writing to the disk, B+tree page might overflow and need to split page or worse do the rotations in MySQL, for the case of multiple writes, this leads to updating multiple pages
  - For PostgreSQL, read the row as above, writes new columns and appends to the B+tree, so page can’t overflow and still chance of B+tree rotation! For the case of multiple writes we are adding few pages so faster!
- So for SELECTs MySQL, for UPDATEs PostgreSQL is better!!!!!!??????!!!!!

How is the analysis with secondary indexes

- SELECT by secondary key
  - In MySQL, traverse secondary B+tree to find primary key and search main B+tree
  - In Postgres Traverse secondary B+tree, to find ctid, and search across main B+tree
- UPDATE by primary key/secondary key with secondary indices
  - Once we find the primary keys, we need to update the main B+tree, and update secondary indices, if we are updating the cols in the UPDATE query.
  - For PostgreSQL, once we find ctids, we append in B+tree, which is faster, but need to update the new ctid in PrimaryKey B+tree (maps to ctid's), and update all secondary indexes B+tree with new ctid, sometimes this makes it overall slower than the benefit of appending in B+tree.
- So for SELECTs it's almost same, For UPDATEs if there are too many secondary indexes you need to update all of them, it's slower!

Overhead of maintaining old versions

- In MySQL, if the transaction commits, you can ignore the old data, conversely if the transactions taking longer, we need to hold very old data, impacts the performance often, more like **C++ cleaning, MySQL started using purge threads too.**
- In PostgreSQL, nothing happens to old right after txn commits, there is regularly VACUUM process cleans old data, more like Golang, Java GC.

Examples

- Example table → employees
- No secondary index, primary key on id column
- We have 5 rows already inserted

id	name	age	joined
1	Abdullah	23	2022-07-01 00:00:00
2	Anand	29	2022-01-01 00:00:00
3	Mohd	25	2021-12-01 00:00:00
4	Saurabh	22	2022-07-01 00:00:00
5	Surbhi	22	2022-07-01 00:00:00

Examples

- (`SELECT * FROM employees`) → **No locks**
- (`SELECT * FROM employees WHERE id = 3 FOR UPDATE`) → X Lock on id = 3, table IX

LOCK_TYPE	LOCK_MODE	LOCK_STATUS	LOCK_DATA
TABLE	IX	GRANTED	*<null>*
RECORD	X,REC_NOT_GAP	GRANTED	3

- (`SELECT * FROM employees WHERE id between 2 and 3 FOR UPDATE`) → X lock and 2, 3 and prevent any insertion b/n 2 and 3 (GAP lock)

INDEX_NAME	LOCK_TYPE	…	LOCK_MODE	LOCK_STATUS	LOCK_DATA
<null>	TABLE	…	IX	GRANTED	*<null>*
PRIMARY	RECORD	…	X,REC_NOT_GAP	GRANTED	2
PRIMARY	RECORD	…	X	GRANTED	3

Examples…

(SELECT * FROM employees WHERE id between 2 and 4 **FOR UPDATE**)
→ X Lock on id 2, 3, 4 and prevent INSERT before 3 & 4 as well.

INDEX_NAME	…	LOCK_TYPE	LOCK_MODE	LOCK_STATUS	LOCK_DATA
<null>	…	TABLE	IX	GRANTED	*<null>*
PRIMARY	…	RECORD	X,REC_NOT_GAP	GRANTED	2
PRIMARY	…	RECORD	X	GRANTED	3
PRIMARY	…	RECORD	X	GRANTED	4

(SELECT * FROM employees WHERE age = 25 FOR UPDATE) → **All rows locks**

LOCK_TYPE	LOCK_MODE	LOCK_STATUS	LOCK_DATA
TABLE	IX	GRANTED	*<null>*
RECORD	X	GRANTED	supremum pseudo-record
RECORD	X	GRANTED	1
RECORD	X	GRANTED	2
RECORD	X	GRANTED	3
RECORD	X	GRANTED	4
RECORD	X	GRANTED	5

How to avoid locking on whole table ?

- Secondary index, let's create a secondary index on 'age' column, and let's see the locks taken for the same query, Expectations ?
  - Take lock on row with id = 3, take table lock (IX)
  - Take lock on index item where age = 25, id = 3
  - How to stop records INSERT'ed with age = 25 by another threads/sessions ?

(`SELECT * FROM employees WHERE age = 25 FOR UPDATE`) → **Index + primary key locks**

LOCK_TYPE	LOCK_MODE	LOCK_STATUS	LOCK_DATA
TABLE	IX	GRANTED	*<null>*
RECORD	X	GRANTED	25, 3
RECORD	X,REC_NOT_GAP	GRANTED	3
RECORD	X,GAP	GRANTED	29, 2

Same Example 2 (Primary key + Secondary key on Age)

What if we query with same secondary index on the table, with age = 22 (FOR UPDATE)

INDEX_NAME		LOCK_TYPE	LOCK_MODE	LOCK_STATUS	LOCK_DATA
<null>	…	TABLE	IX	GRANTED	*<null>*
idx_age	…	RECORD	X	GRANTED	22, 4
idx_age	…	RECORD	X	GRANTED	22, 5
PRIMARY	…	RECORD	X,REC_NOT_GAP	GRANTED	4
PRIMARY	…	RECORD	X,REC_NOT_GAP	GRANTED	5
idx_age	…	RECORD	X,GAP	GRANTED	23, 1

How it’s for secondary **unique** indices

No index (`SELECT * FROM employees WHERE name = 'Mohd' FOR UPDATE`) → **All row locks**

INDEX_NAME		LOCK_TYPE	LOCK_MODE	LOCK_STATUS	LOCK_DATA
<null>	…	TABLE	IX	GRANTED	*<null>*
PRIMARY	…	RECORD	X	GRANTED	supremum pseudo-record
PRIMARY	…	RECORD	X	GRANTED	1
PRIMARY	…	RECORD	X	GRANTED	2
PRIMARY	…	RECORD	X	GRANTED	3
PRIMARY	…	RECORD	X	GRANTED	4
PRIMARY	…	RECORD	X	GRANTED	5

Assume we have secondary **unique** index on ‘name’ column, how does the same^ would work ?

INDEX_NAME		LOCK_TYPE	LOCK_MODE	LOCK_STATUS	LOCK_DATA
<null>	…	TABLE	IX	GRANTED	*<null>*
idx_name	…	RECORD	X,REC_NOT_GAP	GRANTED	'Mohd', 3
PRIMARY	…	RECORD	X,REC_NOT_GAP	GRANTED	3

For secondary unique index

Simple secondary index (`SELECT * FROM employees WHERE name = 'Mohd' FOR UPDATE`)
→ **index + primary key locks**

INDEX_NAME		LOCK_TYPE	LOCK_MODE	LOCK_STATUS	LOCK_DATA
<null>	…	TABLE	IX	GRANTED	*<null>*
idx_name	…	RECORD	X	GRANTED	'Mohd', 3
PRIMARY	…	RECORD	X,REC_NOT_GAP	GRANTED	3
idx_name	…	RECORD	X,GAP	GRANTED	'Saurabh', 4

Assume we have secondary unique index on ‘name’ column, how does the same^ would work ?

INDEX_NAME		LOCK_TYPE	LOCK_MODE	LOCK_STATUS	LOCK_DATA
<null>	…	TABLE	IX	GRANTED	*<null>*
idx_name	…	RECORD	X,REC_NOT_GAP	GRANTED	'Mohd', 3
PRIMARY	…	RECORD	X,REC_NOT_GAP	GRANTED	3

Example locks

```
> CREATE TABLE t(id INT PRIMARY KEY);
> insert into t values (1),(2),(3),(4);
> delete * from t where id=3;
> insert into t values (5);
> BEGIN;
> SELECT * FROM t FOR SHARE;
+----+
| id |
+----+
|  1 |
|  2 |
|  4 |
|  5 |
+----+
> SELECT OBJECT_INSTANCE_BEGIN,INDEX_NAME,LOCK_TYPE,LOCK_DATA,LOCK_MODE
  FROM performance_schema.data_locks WHERE OBJECT_NAME='t';
+----------------------+------------+-----------+-----------------------+-----------+
| OBJECT_INSTANCE_BEGIN | INDEX_NAME | LOCK_TYPE | LOCK_DATA             | LOCK_MODE |
+----------------------+------------+-----------+-----------------------+-----------+
|         3011491641928 | NULL       | TABLE     | NULL                  | IS        |
|         3011491639016 | PRIMARY    | RECORD    | supremum pseudo-record | S         |
|         3011491639016 | PRIMARY    | RECORD    | 1                     | S         |
|         3011491639016 | PRIMARY    | RECORD    | 2                     | S         |
|         3011491639016 | PRIMARY    | RECORD    | 5                     | S         |
|         3011491639016 | PRIMARY    | RECORD    | 4                     | S         |
+----------------------+------------+-----------+-----------------------+-----------+
```

How to handle TABLE level operations

- We know how to provide safety, while concurrently accessing the rows. We have READ/WRITE locks to improve the performance even better.

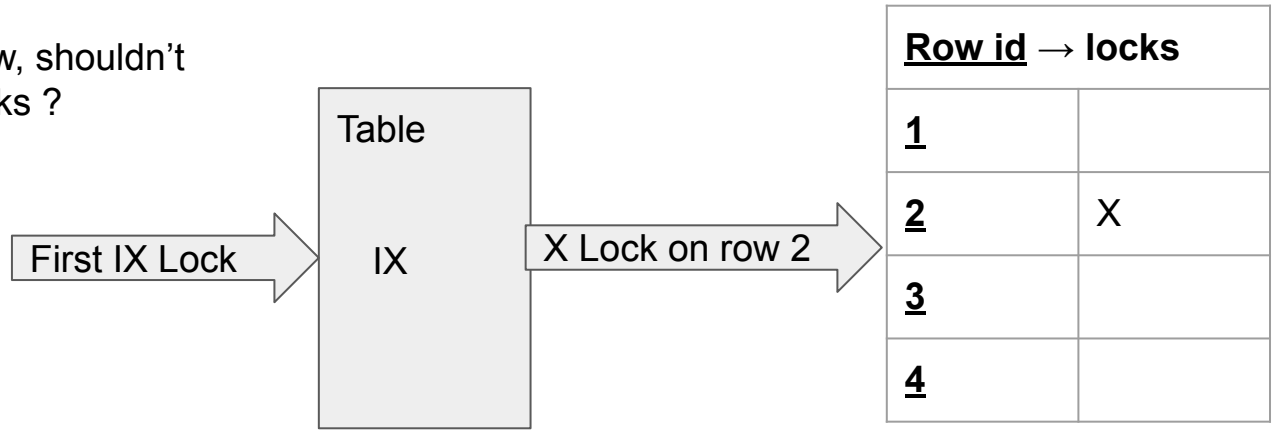
- Sometimes we need to lock the table, but can lock only if there is no ***conflicting*** lock, on any row of the table,
    - If there is no X lock on any row, then we can take S lock on table
        - We need to know if there is an easy way of finding any X lock on any row
- Similarly if there is no X lock on the table, then we can take S lock on any row,
    - We need to know if there is an easy way of finding any X lock on the table, then only we can take S lock on any row.

Row id → **locks**					
1					
2	S				
3	S	S			
4	X				
5					
6	X				
7	S	S	S		
8	S	S			
9					
10					

Solution of table + row locks support with performance

- Any TXN want to take a S lock on any row, first take IS lock on table
  - Similarly TXN needs to take IX lock on table for any X lock on any row.
- Now how to read/write a row
  - Try to take IS on table, if there is X lock on table, then don't grant lock immediately
  - Try to take IX on table, if there is X/S lock on table, then don't grant lock immediately
- How to read/write a table
  - Try to take S, if there is IX/X lock on table, then don't grant lock
  - Try to take X, if there is IX/X/IS/S on table, then don't grant lock

For read/write a row, shouldn't we check other locks ?



Some nomenclature

- S,REC_NOT_GAP → Just read/shared lock on record
- X,REC_NOT_GAP → Just write/exclusive lock on record
- *,GAP → only lock on INSERT to add just before the record
- S or X → read or write lock on record + lock on INSERT to add just before the record (GAP lock)
  - Also called Next-key lock
- X,INSERT_INTENTION → Waiting for INSERT, mostly due to GAP lock
- IS → Intent lock on table → Means somebody is reading a row on table
- IX → Intent lock on table → Means somebody is writing a row on table

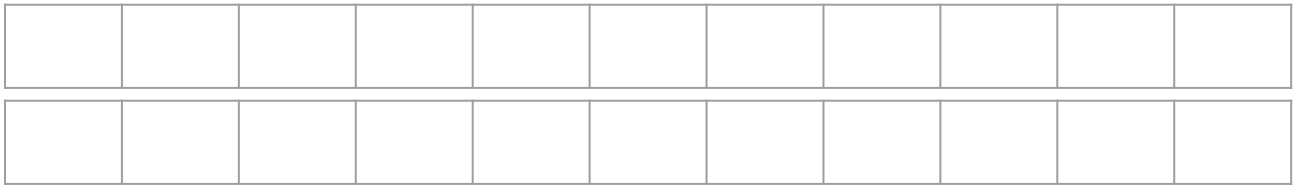
Queue system for locking

Row id: 97, write_flag: false

Serving queue:

Waiting queue:

Entering mutex: mu



```
RequestForLock(type, txnid) {
If (type == s)
    mu.lock()
    waiting_queue.append(txnid)
    mu.unlock()
    wakeup()
}
```

```
wakeup() {
If (write_flag == false) {
    For (req: waiting_queue) {
        If (type == 'X') break;
        serving_queue.append(req)
        grant(req)
    }
    Else {
    condWait()
    }
}
```

MySQL locks description

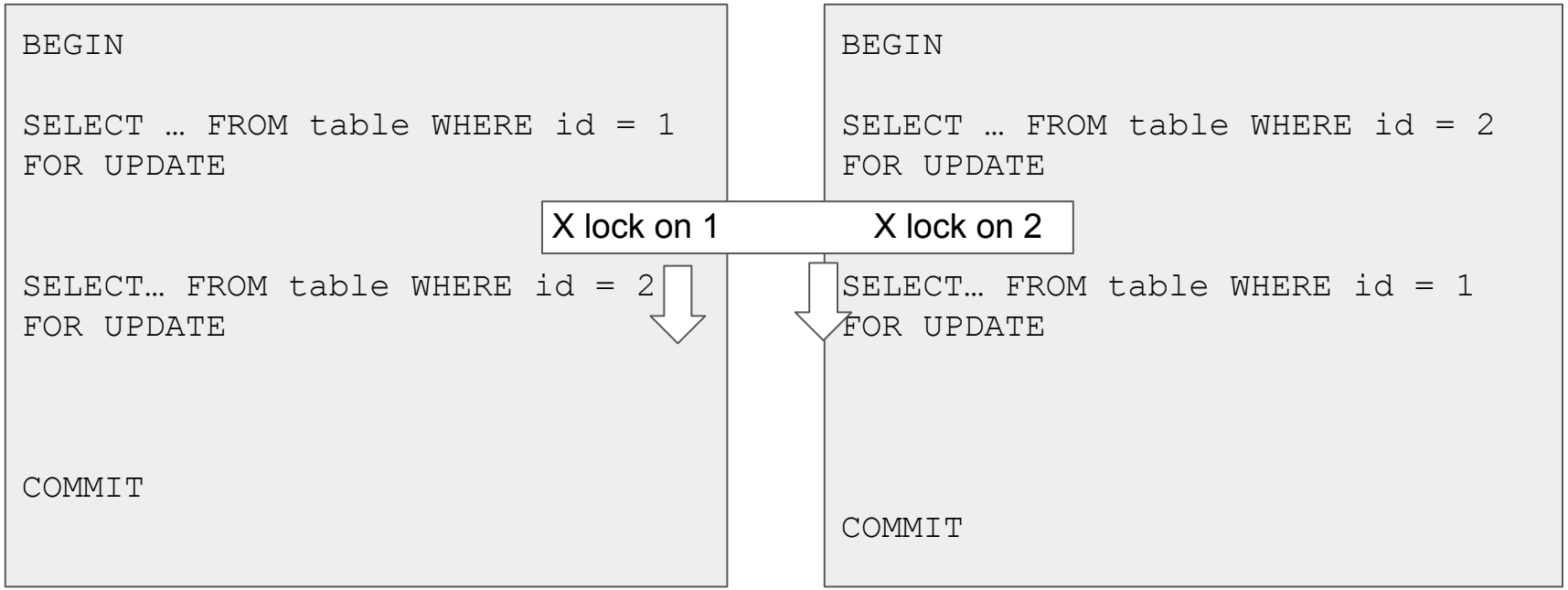
	1	2	3	4
ENGINE	INNODB	INNODB	INNODB	INNODB
ENGINE_LOCK_ID	5376838784:1232:488324…	5376838784:171:4:1:489…	5376838784:171:4:2:489…	5376838784:171:4:3:48
ENGINE_TRANSACTION_ID	1657676	1657676	1657676	1657676
THREAD_ID	91	91	91	91
EVENT_ID	301	301	301	301
OBJECT_SCHEMA	lock_test	lock_test	lock_test	lock_test
OBJECT_NAME	employees	employees	employees	employees
PARTITION_NAME	<null>	<null>	<null>	<null>
SUBPARTITION_NAME	<null>	<null>	<null>	<null>
INDEX_NAME	<null>	PRIMARY	PRIMARY	PRIMARY
OBJECT_INSTANCE_BEGIN	4883242984	4890691608	4890691608	4890691608
LOCK_TYPE	TABLE	RECORD	RECORD	RECORD
LOCK_MODE	IX	X	X	X
LOCK_STATUS	GRANTED	GRANTED	GRANTED	GRANTED
LOCK_DATA	<null>	supremum pseudo-record	1	2

Locks summary

1. If we `SELECT … FOR SHARE/UPDATE` or `UPDATE … ` MySQL will lock rows
2. If we filter (WHERE condition) by primary key, it'll only lock the rows with id
3. If we filter by column without index (`name = 'Mohd'`)
   a. For REPEATABLE READ isolation lock all rows in table
   b. For READ COMMITTED isolation lock only filtered rows
4. If we filter by column with secondary index (`name = `Mohd``)
   a. For REPEATABLE READ isolation - lock only filtered rows + lock secondary index entries along with blocking front and back to stop any insertions with the same column value
   b. For READ COMMITTED isolation lock - only filtered rows
5. Before taking a row lock, Every txn takes intent lock on table, which helps to manage table + row level locks optimally to provide highest concurrency.

Multi locking system deadlocks

- Assume two txns



Avoid overhead of locking ?

- Locking the object every time before the row read or row update is unnecessary especially in low contention time.
  - Let’s say only <1% have conflicts, then is this type of locking is okay ?
- Optimal solution ?
  - Optimistic locking
  - Hopeful for not conflicts, but catch if there is a conflict, then restart from beginning
- Application level implementation (Some other databases have internal implementation, Example FRaM)

Optimistic locking txn example

- Assume each row has version number, inserted with 1, gets incremented for every update
  - (primary_key, version, …(cols))
  - Every update looks like ‘`UPDATE table SET` **`version = version + 1,`** `… WHERE …` ’
- Txn Start (BEGIN)
  - Read row with version, let’s say we read version as 4
  - Do work (apply logic, read other rows)
  - Update row (update only if version matches, for every update increment the version number)
    - `UPDATE table SET` **`version = version + 1,`** `… WHERE … AND` **`version = 4`**
  - Check updated row count returned from ‘UPDATE …’ statement, if it’s ‘0’, it’s more likely that rows version is changed, **restart the entire txn**.
- Commit or Rollback

MySQL Transactions output

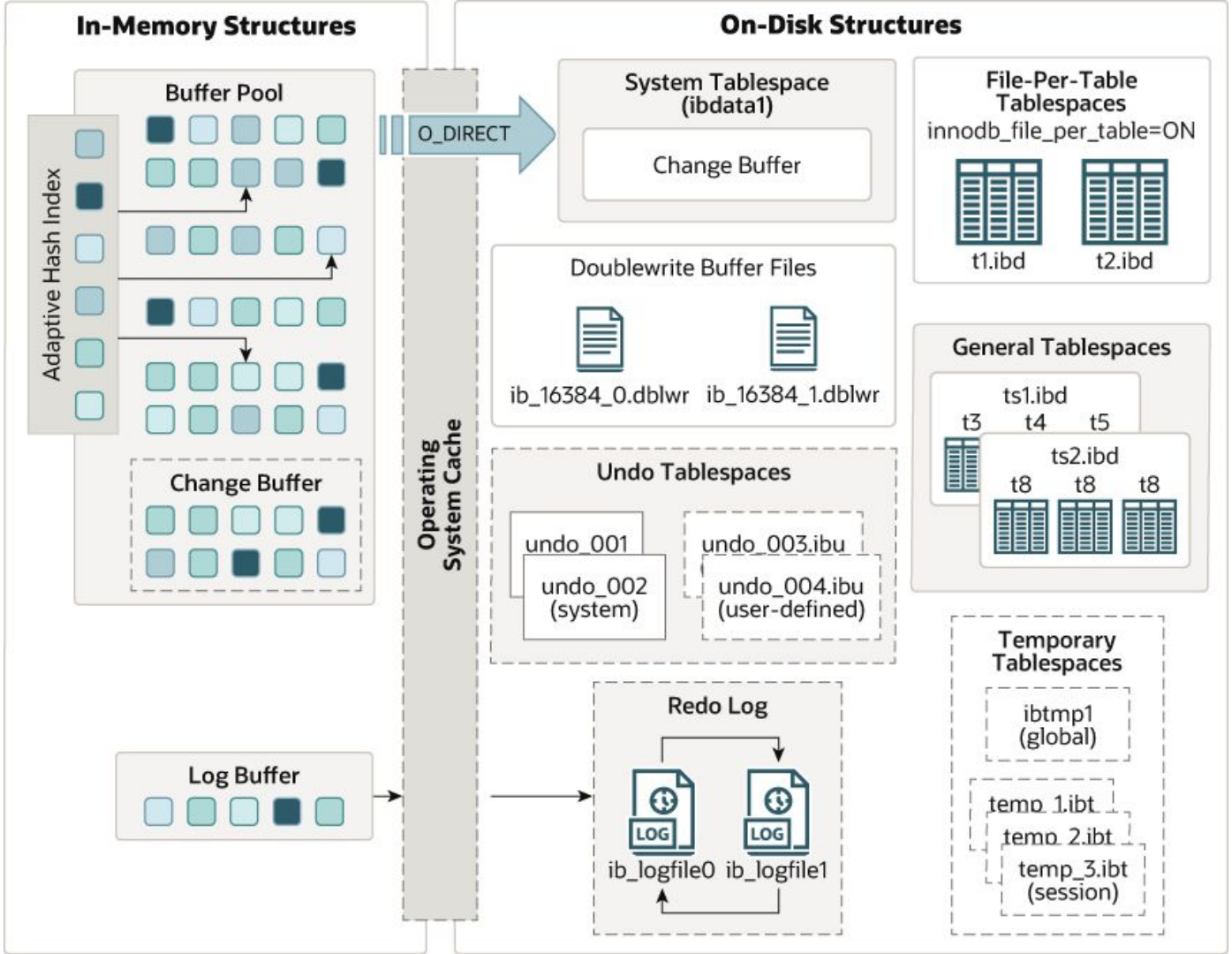
```
# ------------
# TRANSACTIONS
# ------------
# Trx id counter 1657283
# Purge done for trx's n:o < 1657277 undo n:o < 0 state: running but idle
# History list length 0

# LIST OF TRANSACTIONS FOR EACH SESSION:
# ---TRANSACTION 1657282, ACTIVE 28 sec
# 2 lock struct(s), heap size 1128, 6 row lock(s)
# MySQL thread id 9, OS thread handle 6117715968, query id 2151 localhost
127.0.0.1 root starting
```

Multi thread txn Summary

- We need to execute multiple statements together in atomic way. This is supported through transactions
- To provide consistency, we need READ_COMMITTED isolation level
- But that’s not good enough, and still txns seeing committed data after txns could break the transactions consistency
- To avoid that REPETABLE_READ, SERIALIZABLE isolation levels are provided
- Most of the databases strive to provide the locking mechanisms to support more concurrency, so try to lock only minimal (rows). However while minimal locking is not guaranteed, but consistency!
- They need additional data structures to store the previous versions data, and mechanism to merge them.

Is it the best possible isolation levels ?

- For single writer/coordinator instance like MySQL(Aurora/AlloyDB), PostgreSQL, Mongo, this is good enough
- If we have multiple machines that are writing the data to multiple partitions we get more problems
- We’ll discuss in the later part! (Cassandra, Spanner)

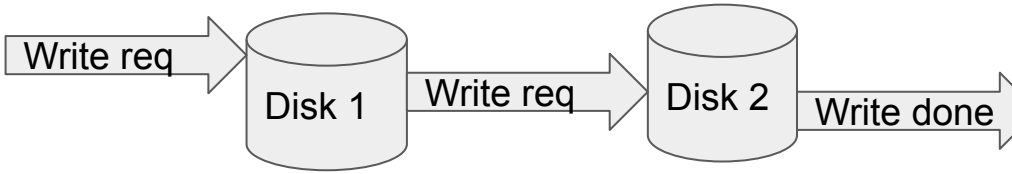

In-Memory Structures
On-Disk Structures
Buffer Pool
Adaptive Hash Index
Change Buffer
O_DIRECT
Operating System Cache
System Tablespace (ibdata1)
Change Buffer
File-Per-Table Tablespaces
innodb_file_per_table=ON
t1.ibd
t2.ibd
Doublewrite Buffer Files
ib_16384_0.dblwr
ib_16384_1.dblwr
General Tablespaces
ts1.ibd
t3
t4
t5
ts2.ibd
t8
t8
t8
Undo Tablespaces
undo_001
undo_002 (system)
undo_003.ibu
undo_004.ibu (user-defined)
Log Buffer
Redo Log
LOG
LOG
ib_logfile0
ib_logfile1
Temporary Tablespaces
ibtmp1 (global)
temp_1.ibt
temp_2.ibt
temp_3.ibt (session)

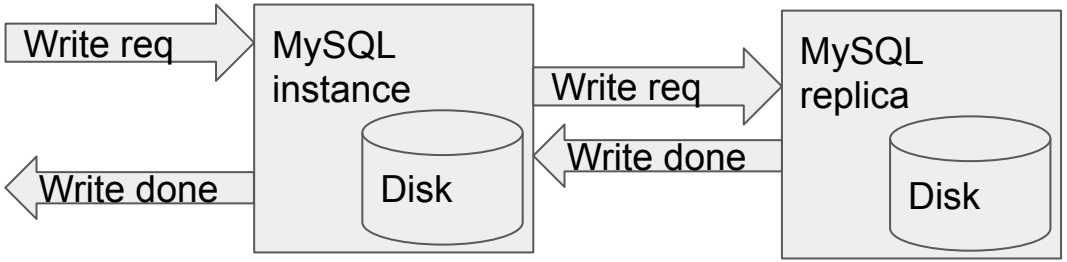
Improve Fault tolerance

Improve the fault tolerance

- Even a machine fails, keep the datasafe
- Store it in multiple places
  - Disk level replication
  - Database level replication
    - Sync
    - Async

- Disk level replication
  - EBS on AWS





- Database level replication
  - Two DB instances to use

Impact of Sync vs Async

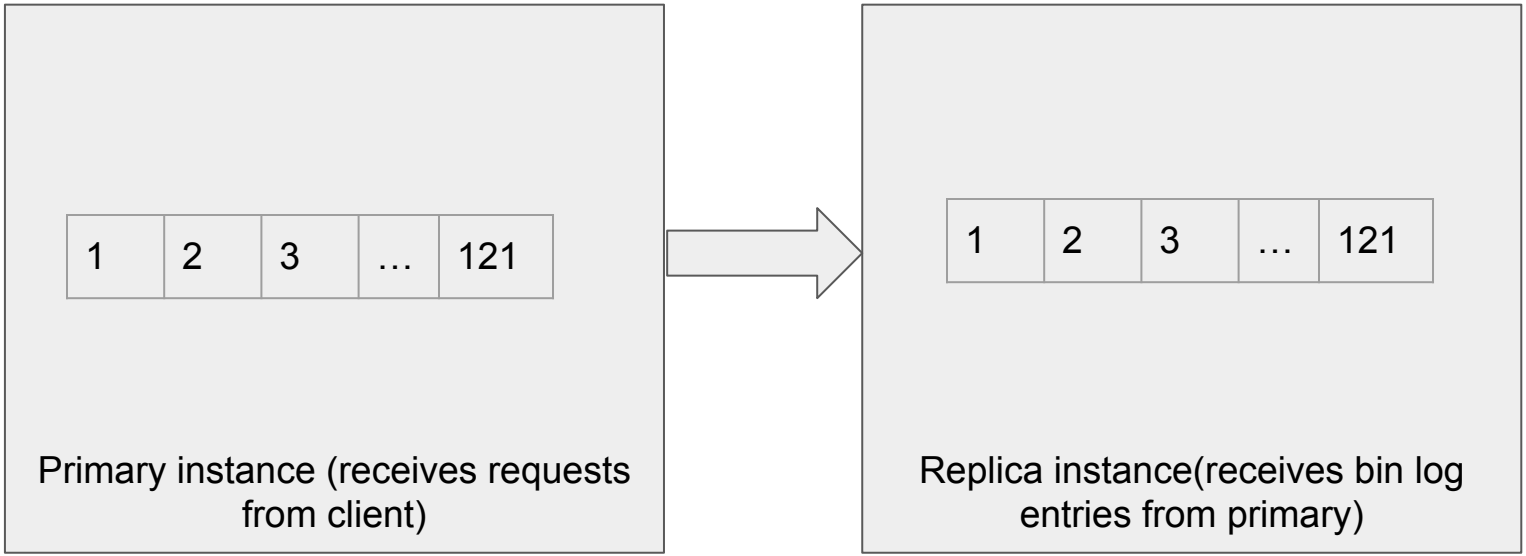
- For Async approach, there might be delay(especially if there is a problem in network), so systems won’t be in consistent way, these systems are called AP
- For sync approach, if any of the system down, we can’t continue further, but they are consistent, these systems are called CP
- First, let’s see how to build consistent systems (CP) with improving Availability.
- As usual, we add performance (resource usage + latencies) lens

Approach of consistent systems

- Start with replication of instance, it increases fault tolerance
- If few machines fails*, the system continue in consistent(single source of truth) way
- For failed machines, either restart or add replacement
- If a writer fails restart it, halts writes until we get the writer back**
    - If writer can’t be recovered we need to select one replica which is in sync with writer (primary) instance

Basics of replication, with consistency @ reading

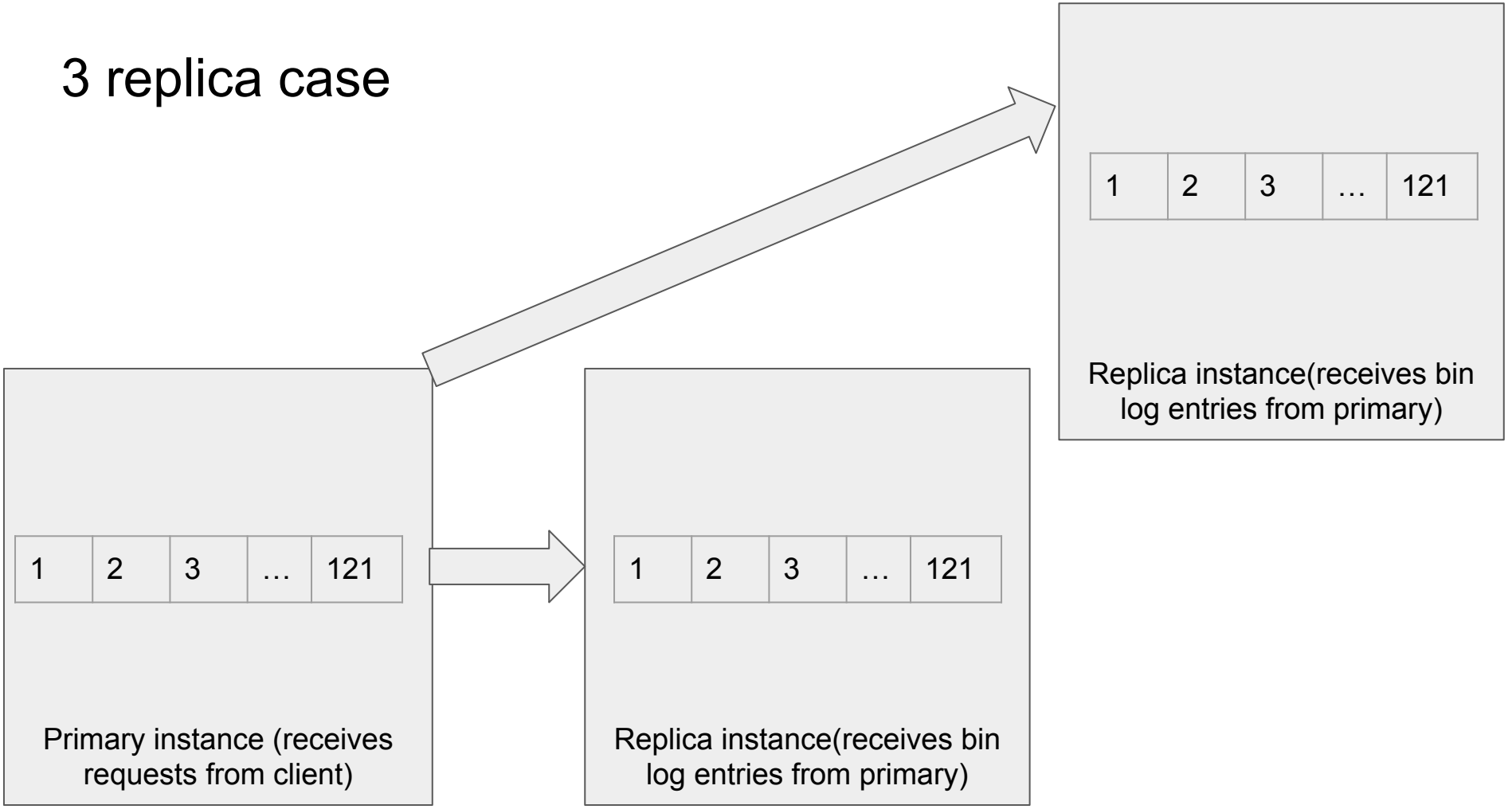
Let's start with a **primary** instance (receives requests from the client) and sends the changes (bin logs) to the replica, and sends the client success if primary instance and replica instance writes data successfully. From the log, both constructs the actual database structures. What do we improved here ?



Assess 2 replica case (including primary one)

- Writer fails, we can’t proceed further
- Let’s say the probability of one server up is 0.9, the probability of both servers up is only 0.9 * 0.9 = 0.81
- We reduced the availability
- Can we improve availability with 2 replicas ?
  - No
- Let’s add another replica, with writing requirement of only 2 replicas (including primary one)
- So even one fails, we can continue, Now the availability is ?
  - 0.9 * (0.9 * 0.9 + 0.9 * 0.1 + 0.1 * 0.9) = **0.891**
  - **We improved availability compared to 1 replica setup (>0.81)**
  - **For 5 replicas with 2 failures allowed it’s = 0.89667 (binomial theorem)**
  - **For 7 replicas with 3 failures allowed it’s = 0.898857**

3 replica case



Summary of single writer system

- Primary instance takes requests from client
- It’ll add log in it’s own system,
- Sends bin logs to all replicas,
- if n/2 replicas + 1 primary are replied success (replicas reply success only if the previous messages/logs are already in the replica machine), reply client it’s success
- The limitation is loss of availability
- If the primary fails(we have some “Health check server”), we need to restart

Primary failure handling - Split brain

- Assume 1 primary and 2 replicas with node ids (n1, n2, n3), n1 is primary
- n1 is recognised as failure by health check monitoring due network issue b/n n1 and health check server, but client is communicating with n1 properly.
  - So client is sending write requests as usual
- “Health check” server selected n2 as primary, another client is writing to n2
- Now N3 receives from both N1 and N2 - Called split brain
- Solution
  - There are many solutions to prevent
  - One good solution is before electing a primary make sure inform at least n/2 other replicas so that they reject the requests from previous primary (since you can’t have 2 * (n/2 + 1) servers.
  - Usually primary selection also involves what’s the right log number ?
  - We’ll check all other replicas, find “Good” LSN number, assigns node 2 as primary, and starts with “Good LSN”

How to maintain consistency with 1 server going down

Consider sequence

- Primary received primary keys p1, p2, p3, with log numbers (1, 2, 3), and responded to client success
- Has written to replica 1, now replica 1 has logs → (1, 2)
- Has written to replica 2, now replica 2 has logs → (1, 3)

What if primary fails, can follower 1, or follower 2 can be source of truth ?

- No right, So we are losing high availability
- How to solve ?
- If you are replicating LSN 3 to replica 2, you should also replicate LSN 2 by that time, then only you can replicate LSN 3 to replica 2, and can say that it's copied on 2 replicas so we can return success to the client,
- So follower 2, can't have (1,3), they should always have (1,2,3) - No holes allowed, if you have 'K', you have all LSN <= K

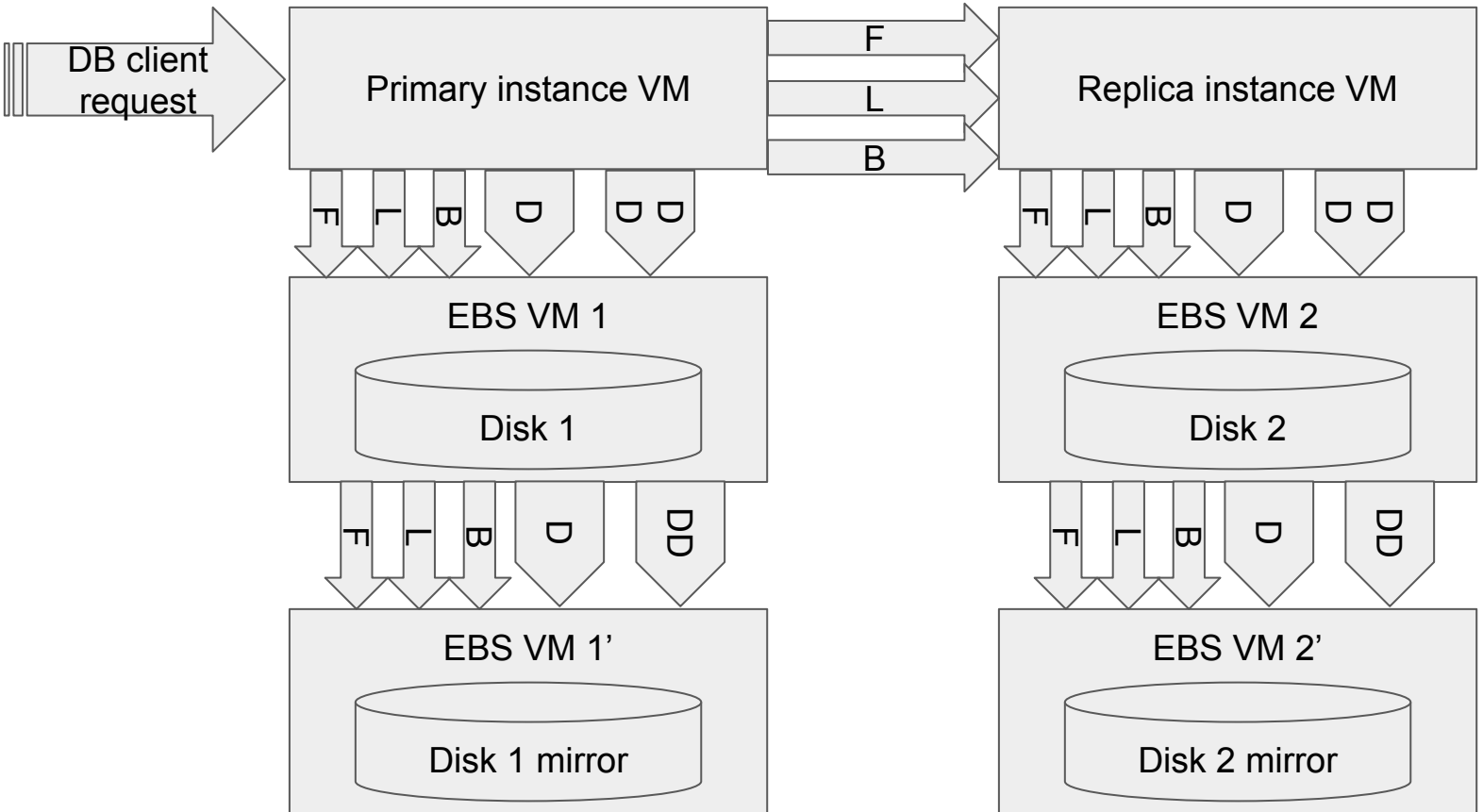
Example of Single master DB

- MySQL (Sync + Async)
- Aurora from AWS
- Alloy DB from GCP

Performance issues with single master in cloud (AWS/GCP)

- Let’s try to understand disks in cloud
- If we take any Virtual Machine(a slice of some physical machine CPU, RAM, Disk and Network), if the server goes off, we will lose the disk
  - Many cases require to keep data for longer period than machine
  - If Machine restarts it’s not guaranteed to get on the same physical server
- To improve reliability AWS introduced concept of EBS, which writes to two disks in two different servers, but in same availability zone
  - The two disks (two servers) communicate over network
  - If another VM want to use this disk we use it as NAS (Network attached storage), we mount it on partition, so we often don’t see the difference.
- For MySQL or DBs it chokes network, how ?

General MySQL replication with EBS (deep view)



F → FRM data
L → Log data
B → Binary logs
D → Data pages
DD → Double data buffers

As mentioned in aurora white paper

- Here two replicas need not communicate Double write buffers, Data buffers.

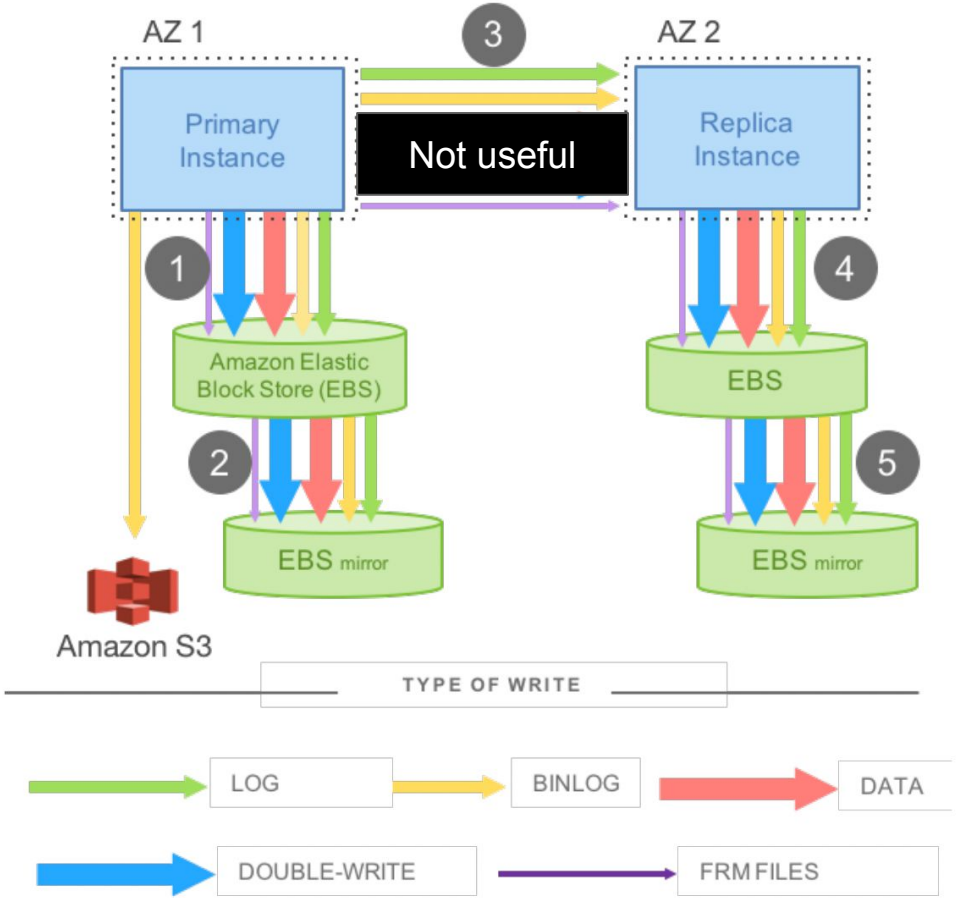
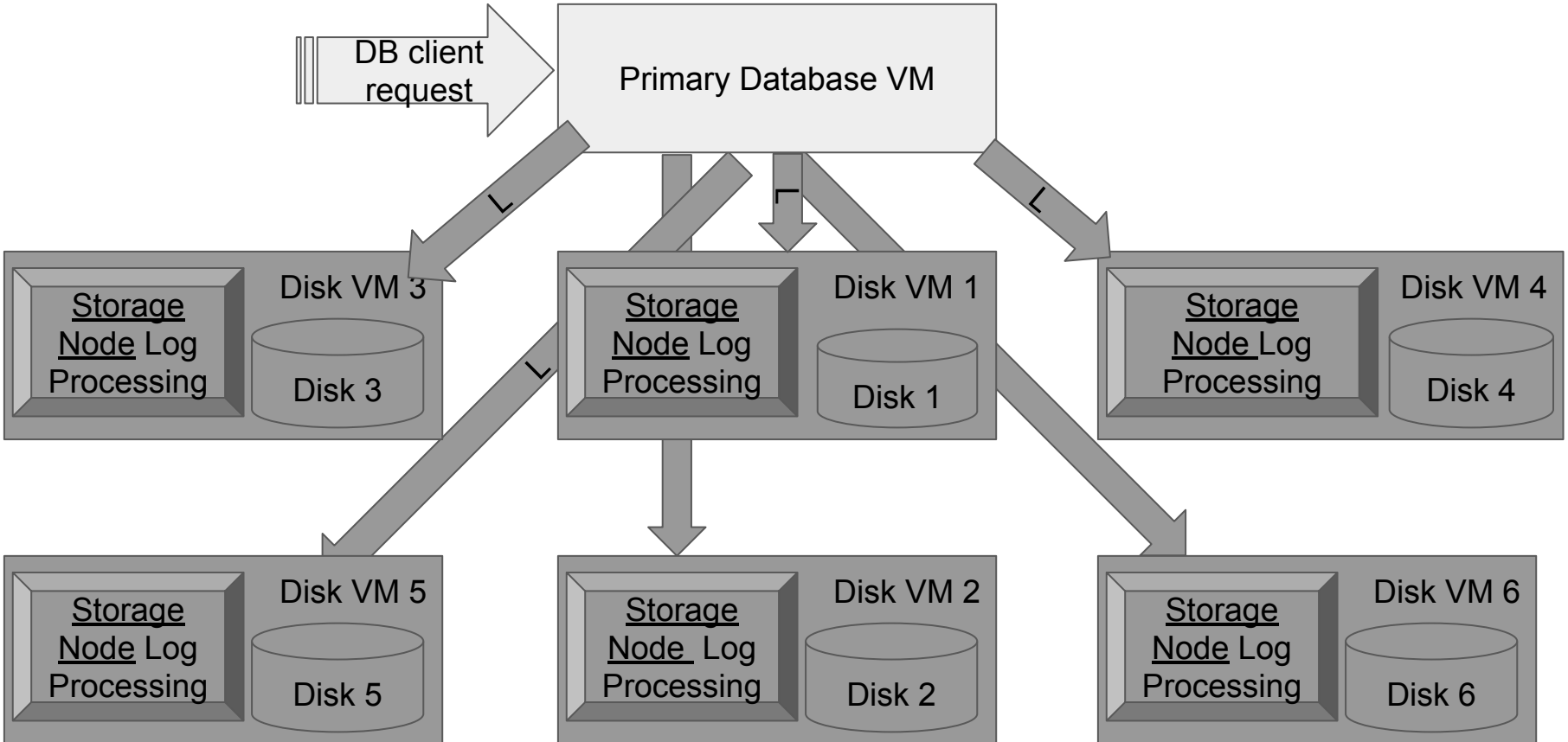


Figure 2: Network IO in mirrored MySQL

Aurora SQL replication with Disk servers(deep view)

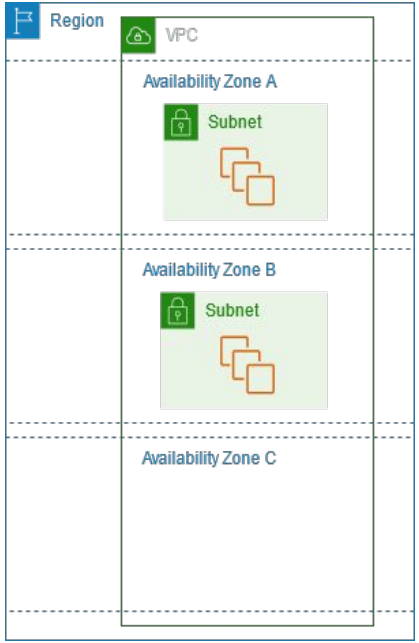


Improve IO and Network @writes

- Send only the Redo logs to Disk owned server, call it as “Storage node”
- Storage node now have logic to process the logs and store the data in optimal format
- Downside is sending to 6 storage nodes (not storing anything in the database instance), since we are waiting for only 4 replicas it won’t hit the latency significantly (overall 50% will become 67% of single server)
- Even though figure is not showing, if there are missing messages, each storage nodes communicate and fill the messages.
- They also check if any other data like B+tree part is missing or corrupted(hash checking), in such cases storage nodes communicate with each other (without main database writer) and correct the data.

Why 6 nodes to write ?

- Let's understand availability zones in AWS, they are in same region (like mumbai) but are separate centers connected by low latency network and within 100km. Usually they have independent power handling, security…*
- However if a availability zone (AZ) fails, it could stay for a 4+ hours, could be due to flooding, fire, rat…



- Let's say we have flooding in in AZ-'A', Within 4 hour period, there is very less chance of failure on another AZ, but given we have 100K+ VMs in AZ, there is a chance that few disks or networks cards or CPU's fail or there could be upgrades.
- So our system should be fault tolerant to at least 2, in reality AZ +1 failures.

Why 6 nodes to write

- Mostly we have 3 AZ per region
- Let’s start with 3 storage node replicas, with each replica in one AZ, if one AZ fails, and any one of other AZ gets some partial failures during the same time (the probability is high) then we’ll have only 1 replica, so the system will be unavailable (we can’t loose consistency)
- Try 4, with one AZ having 2 replicas, other 2 AZ’s having one replica each, if the AZ with 2 replicas fails, then we face similar problem
- Try 5, with two AZ having 2 replicas each, other 1 AZ’s having one replica, let’s say one AZ with 2 replicas fails + we’ll have one replica failure due to partial failure in AZ, during that we have only 2 replicas
    - Since Main DB Instance writer would only wait for 3 replicas response, the 2 replicas might not have latest data in worst case
    - Let’s say the Main DB Instance writer waits for 4 replicas response while writing ?
    - The latency is higher

6 nodes in practice

- If we have 6 replicas, with each AZ having 2 replicas each, even if we lose an AZ and a replica we still have 3 replicas, since the Main DB writer writes to 4 replicas at least 1 out of 3 should have latest data. We would build one more replica with this data quickly (MTTR), and will have 4 good replicas overall to proceed with writes.
- AWS build another replica using 10 GB segments using multiple network lanes, so that MTTR is so quick and significantly lower than MTTF (fix quickly before another failure happens)
- A background service automatically monitors these 10GB segments, if there are failures, it'll try to fix irrespective of any other failures.
- Group of 6 replicas with 10GB segments are called Protection Group (PG)

Can we build aurora with 5 replicas having same availability ?

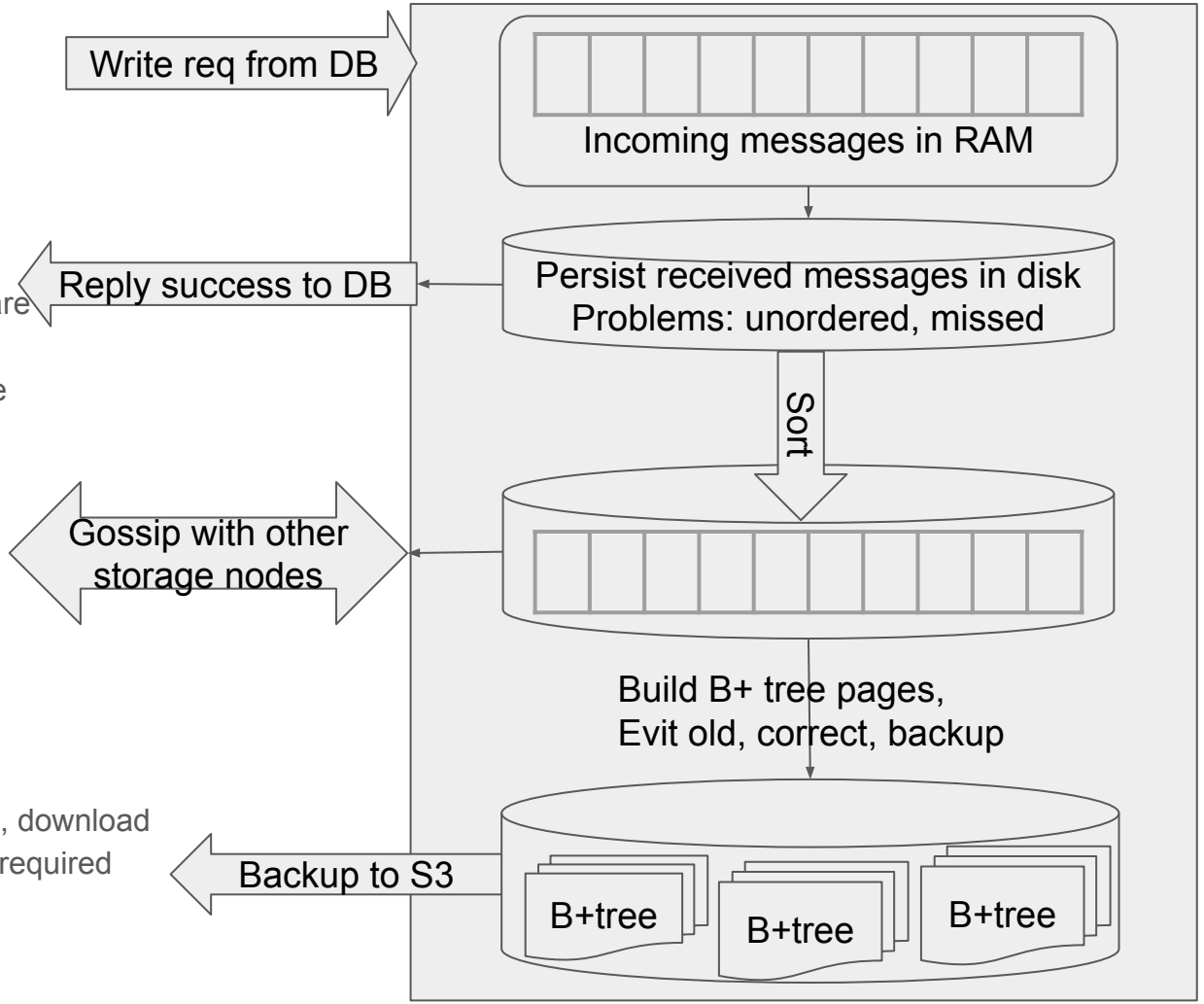
- Yes if we have 5 Availability zones with each zone having one replica.
- If one AZ fails, and another replica lost, we still will have quorum to figure out
- What about latency ?
  - The original aurora wait for 4 replicas out of 6, which are cross zone (67% time)
  - This architecture requires 3 replicas out of 5, which are cross zone, but lower latency(60%)
- What about availability ?
  - In original architecture if AZ+1 failure occurs, writes will stop.
  - In the proposed architecture if AZ + 1 failure occurs, writes will continue.
- What about MTTR ?
  - Building a replica might require more time due to cross zone, but still okay (usually have multi 100Gbps or multi Tbps links b/n two zones), For larger chunks like 10GB, the extra 5ms latency doesn't effect
- What about MTTF ?
  - The chance of going one AZ is more 5 AZ case, compared to 3 AZ case, However in reality that time is still significantly high compared to MTTR.

Storage node in detail

Write requests from Main DB writer are put in the disk and reply success to DB, However the messages might be unordered, missed. We still reply success,

From Log, periodically

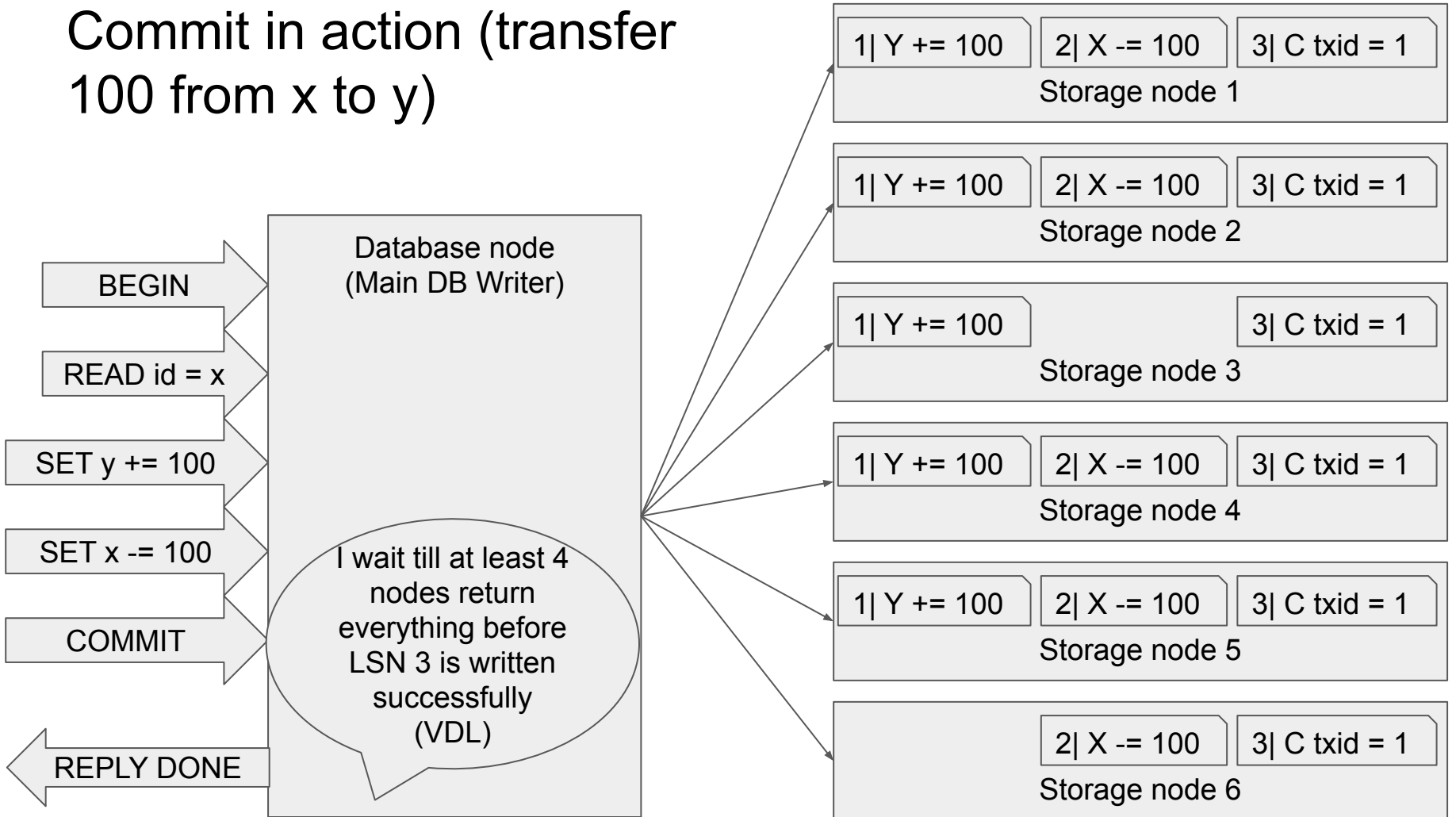
1. Build B+trees for reads
2. Clear old versions
3. Compute CRC, and cross check, download segments (10GB) from others if required
4. Backup to S3



Commit in Aurora

- Main DB writer needs to reply commit success to DB client
- We should return success to the DB client iff at least 4 storage nodes has written all the contents of the txn to the disks, so that even if the DB node restarts, and we lose 3 storage nodes, we should be able to recover.
- Since the Storage node returns success after writing message to disk, since there could be gaps before the commit message, it doesn't guarantee that whole data in the txn has been written to at least 4 nodes. Especially in case of restarts we can't recover with gaps.
- So the DBWriter needs to know & wait till at least 4 storage nodes have written all data before the txn commit log (commit is also another log message), this is happens through communicating these points VDL.

Commit in action (transfer 100 from x to y)



Logic of commit

- Database writer maintains LSN
- For each write, generate new LSN (auto increment by 1) and send message to all storage nodes, as soon as you get 4 replies move to next message
    - Each storage node maintains VCL (Volume complete log) to track, so that all logs before VCL has been stored on disk.
- Database writer sends commit message with new LSN and broadcast to all 6, however doesn't reply to DB client, and puts it in commit wait queue.
    - Each storage node call this message as CPL (Consistency point log), we can still have gaps
- All storage nodes fill gaps by communicating with each other(Gossip protocol)
    - If there are no gaps before the CPL, we take max such CPL, call it as VDL
    - Storage nodes send the VDL to main database writer
- Main database writer reads VDL, and advances it's VDL, if at least 4 nodes replied min such VDL(3rd best), and checks commit wait queue, if any clients are waiting for <= DB VDL (CPL), it takes all such client connections and reply to client

Reads in aurora

- DB writer receives request maintains InnoDB buffer pool, if the key is found return from the buffer pool
- If not fetch the page from Storage node, and puts it in buffer pool and returns
- For general MySQL, before evicting a page, you need to write to disk, but here the data is persisted in storage node. We don’t have such problem, This is one important parameter to scale
- Can DB writer read from any storage node ?
    - DB writer tracks VCL for each storage node
    - If a particular storage node as VCL of latest LSN, send the request to any storage node
    - This is another parameter to scale in Aurora.

Aurora numbers

- The claim from the paper is 35X more transaction, and use <1 IO per write

Table 1: Network IOs for Aurora vs MySQL

Configuration	Transactions	IOs/Transaction
Mirrored MySQL	780,000	7.4
Aurora with Replicas	27,378,000	0.95

Aurora scaling with data size

- For MySQL used 30K IOPS for EBS
- Used r3.8xlarge machines 32 vCPU, 244 GB RAM

Table 2: SysBench Write-Only (writes/sec)

DB Size	Amazon Aurora	MySQL
1 GB	107,000	8,400
10 GB	107,000	2,400
100 GB	101,000	1,500
1 TB	41,000	1,200

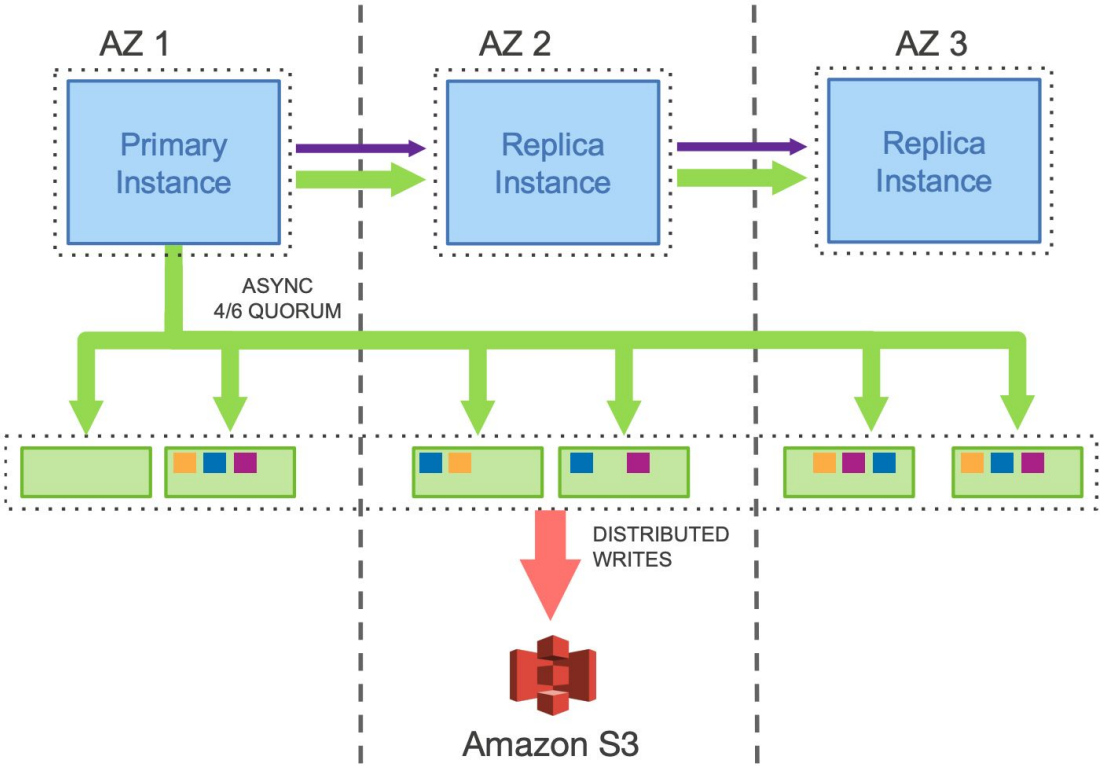
Aurora scaling with connections

- Same setup

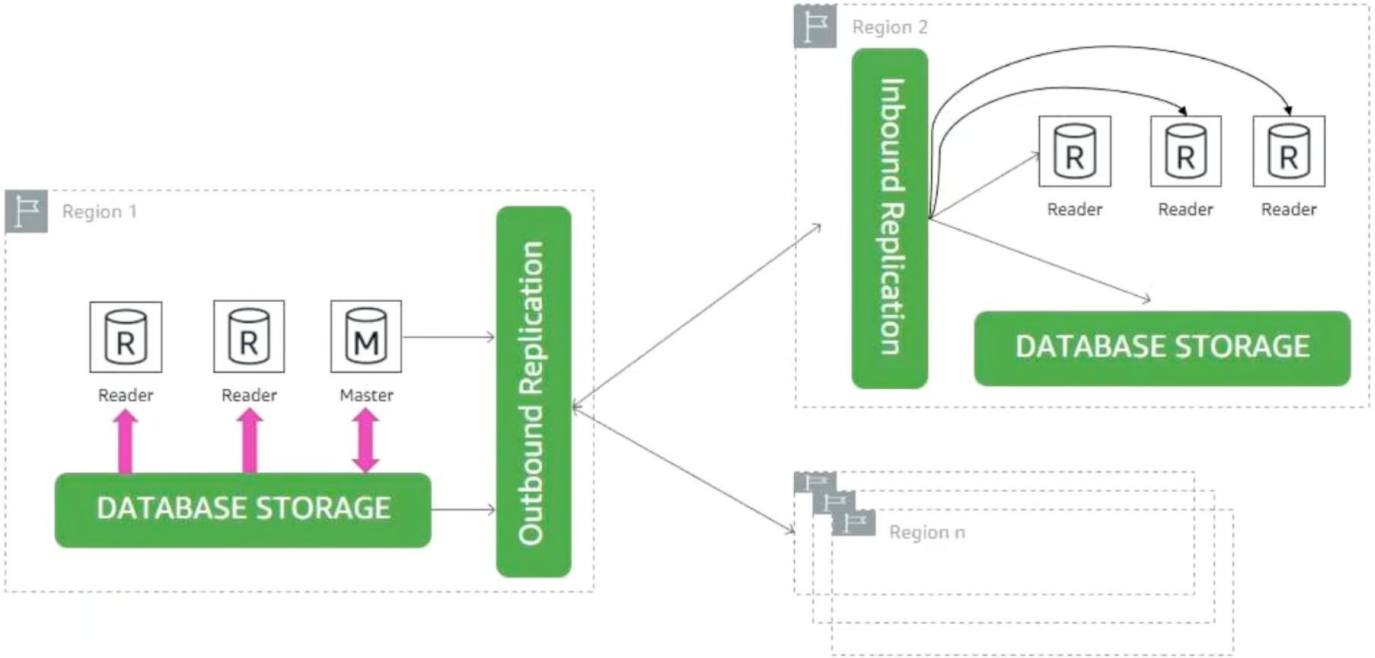
Table 3: SysBench OLTP (writes/sec)

Connections	Amazon Aurora	MySQL
50	40,000	10,000
500	71,000	21,000
5,000	110,000	13,000

What happens to storage nodes, when we add replica instances ?



You can add replicas in cross region too (Aurora Global)



High throughput: Up to 200K writes/sec – negligible performance impact

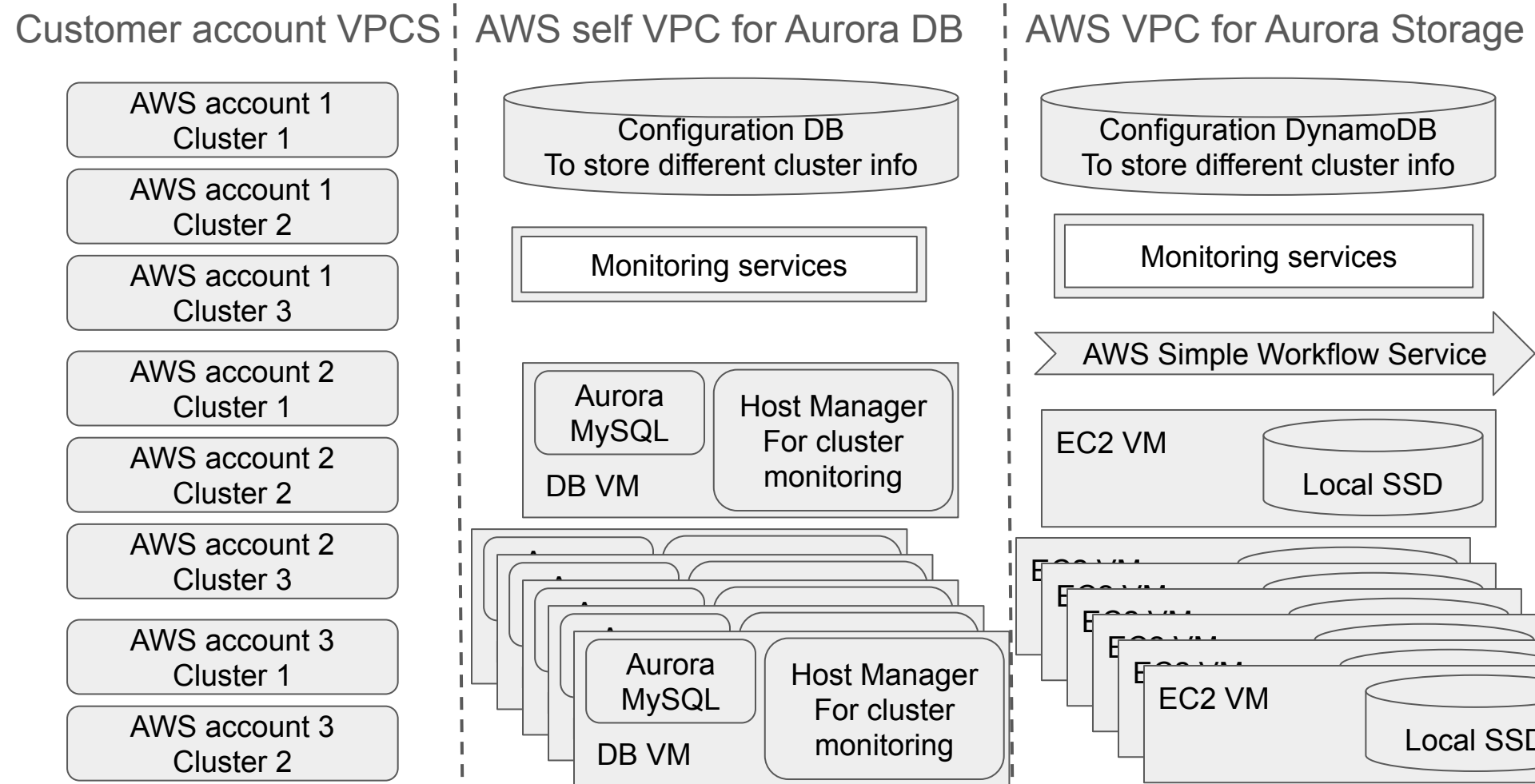
Aurora limits

- For reads two copies are maintained storage node and database node
- Network communication for B+ tree pages (why not IPC in same machine ?)
- The data needs to fit on one storage node
  - So the limit is 128TB (for older version it’s 64TB) per database
- Single writer can have limit on concurrent writes, currently it’s limited to 10M, but in reality it’s quite low.
- If the Main DB writer goes down it’s SPoF, no availability improvement there
  - In restart whole InnoDB bufferpool maintained in another process, so no warm up delay
- The DBwriter is not elastic (For example we don’t use DB in the nights still needs to pay for that instance)
  - Solution with serverless v2 is still limited

Aurora Deployment & Elasticity

- Aurora storage starts with 10GB, keeps on adding 10GB segments as we require, thus reduces devops load a lot. Charges at $0.11/GB (or $110/TB) per month.
- Aurora maintains separate process for InnoDB buffer pool, so that if the main DB writer restarts, it starts with warm cache
- Aurora IoPS cost is important too, charged at $0.22/Million IoPS
- Serverless v2 allows to scale the required main DB instance size with 0.5 CPU steps
- Aurora Global just keeps read replicas in the other region, you can’t write to DB from two regions in any case. But while doing cross region replication it’ll copy storage nodes as well.

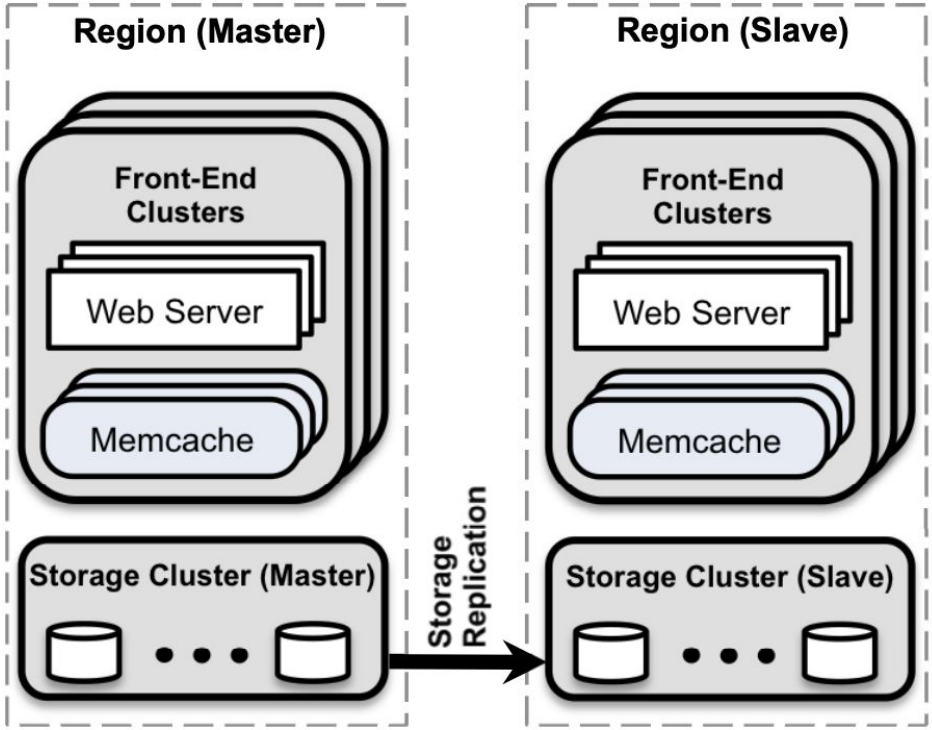
How AWS manages multiple accounts or clusters of aurora



How to solve the problem of scaling writes - Manual Sharding

- For example facebook shards data on user id, post id… (at application level routing)
- It scaled very well for facebook, they replicated the data across two regions (East and west)
- For facebook case, writer is always in US-West and they copy it to US-East sync and async
- They use coherent cache with 1% gutter servers to make sure there is no extra load on MySQL even during the cache servers failure (Memcache)
- Ended up making changes in networks and application routing too, to reduce network latency and reduce load on single MySQL cluster

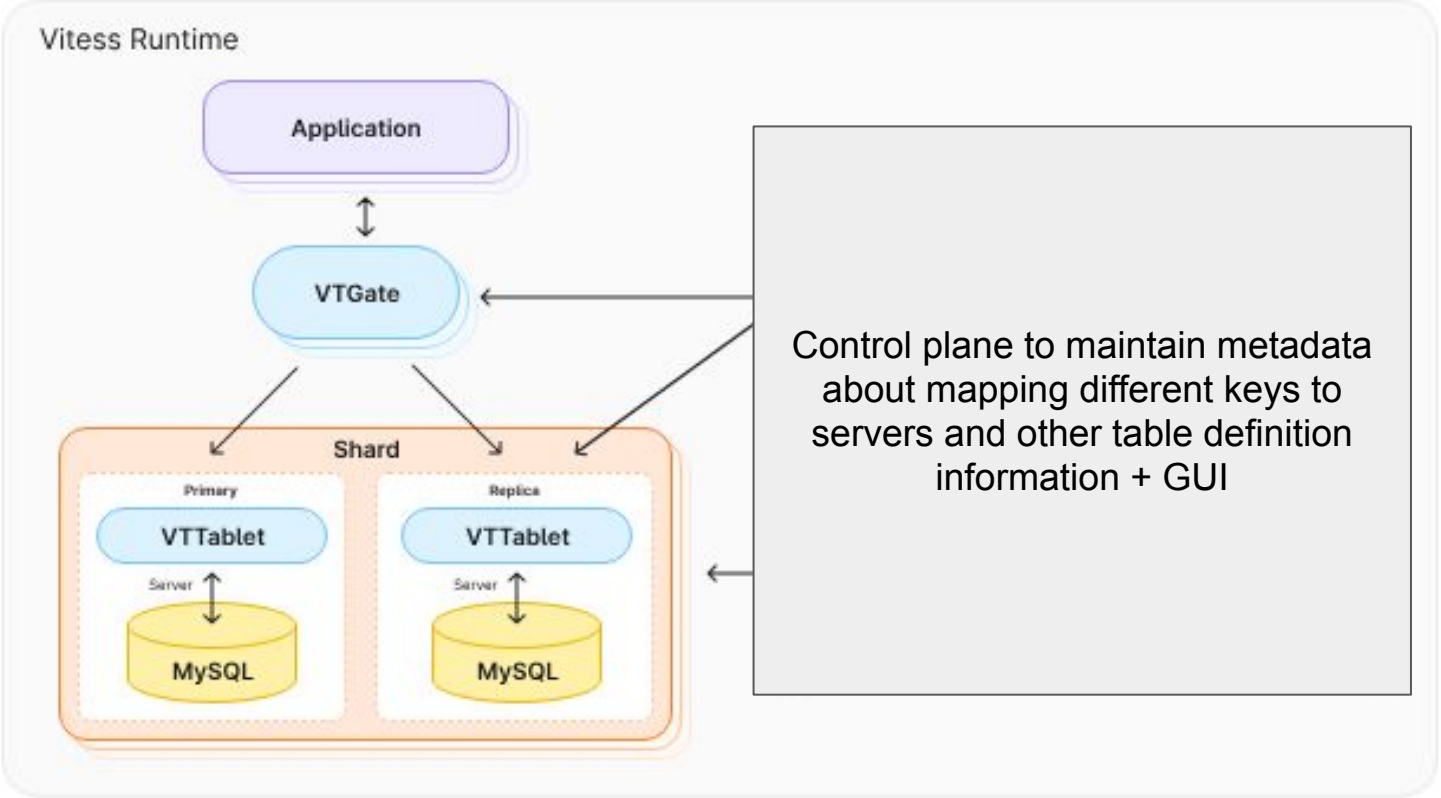
Facebook architecture



How to solve the problem of scaling writes - Automated Sharding

- Vitess is one example used by Youtube
- Applications go through a intermediate proxy server called vtgate and it routes to multiple vitess tablets (a single mysql instance)
- Distributed transactions are challenge (Recently released with alpha/beta level),
- The isolation levels are bad when txn includes cross shards.

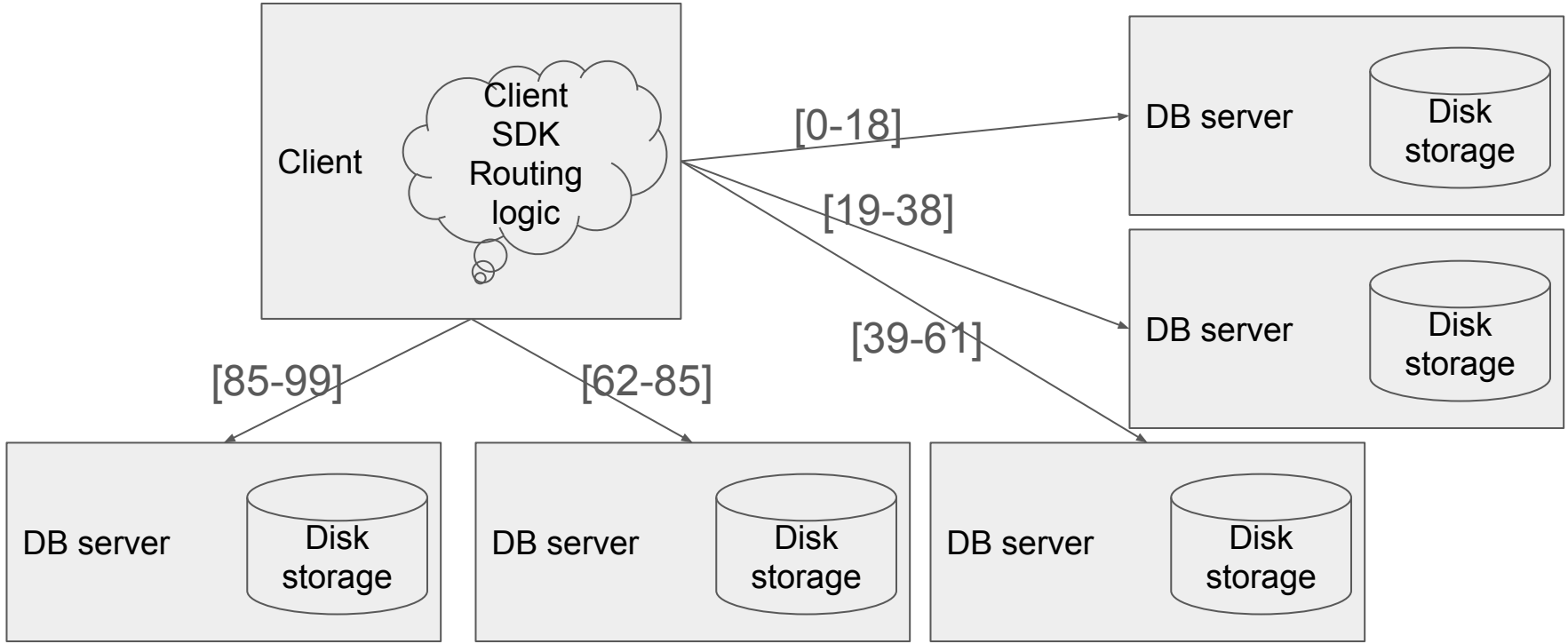
Vitess architecture



How to solve the multiple writer problem

- Distribute data across multiple servers, along with primary key, we have a concept of partition key as well
- Primary key = Partition key + Extra key part (called Cluster key in Cassandra)
- So the database automatically routes primary key by extracting partition key and computes hash and route to the server (for read and writes)
- The hash algorithm would be elastic friendly, When a node adds or gets deleted we don’t want the algorithm to distribute all keys to all nodes in random fashion
- Cassandra is example for that (ScyllaDB is C++ and Seastar optimised implementation)

Advantages of splitting writes to multiple machines

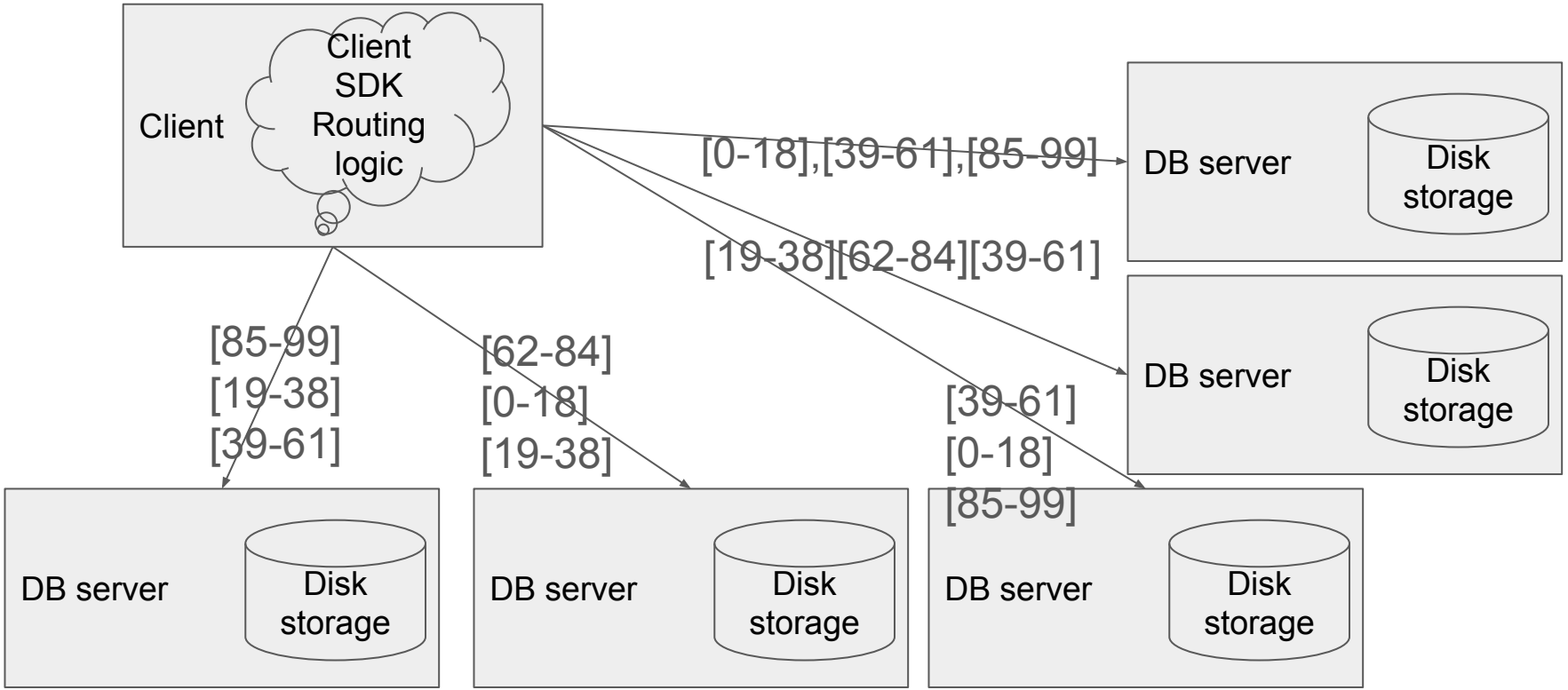


Assume partition keys are hashed to integer and we take mod(100),
Cassandra takes full integer range (-2^{63} to $+2^{63} - 1$)

Fault tolerance

- Every key range (also token range) called partition has multiple replicas, where each replica kept in different machines, often represented with RF
- It’s the responsibility of the DB server
  - to communicate with other replicas (**Coordinator role**)
  - Write to own replica
  - Might wait for reply of X replicas (X depends on the request)
- All replicas are equivalent (no primary or secondary)
- If any machine (db server node) geos down, other machine can own the token ranges for sometime, and when the server node comes back it’ll give the data back to that node - Hinted handed off

Token ranges with replication (RF = 3, 5 nodes)



Writing summary

- Whole 100 key ranges are divided among 5 servers
  - [0-18] → Node 1
  - [19-38] → Node 2,
  - [39-61] → Node 3
  - [62-84] → Node 4
  - [85-99] → Node 5 (approximately equal)
- The client SDK, calculates key (from partition key), then maps it to appropriate node using binary search
  - ≤ 18 → Node 1, 19 - 38 → Node 2, 39 – 64 → Node 3…
  - Then forwards the request to appropriate node
- The DB node, coordinates (forwards the write request) with other replica servers to write to RF replicas

Elasticity and load balancing

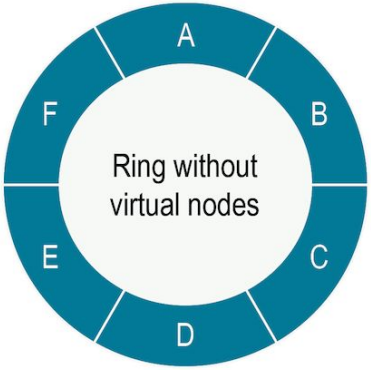
- Assume 100 keys case with 5 servers, with 5 different ranges
  - [0-18],[19-38],[39-61],[62-84],[85-99] (mapped to 1, 2, 3, 4, 5 nodes respectively)
- Let’s say the node 4 fails, now the token range would be
  - [0-18] → Node 1
  - [19-38] → Node 2,
  - [39-61] → Node 3
  - [82-99] → Node 5 (approximately double datasize, and double requests)
- This creates imbalance in the load
- The solution ? First, what’s the goal ?
- We wanted only the data on node 4 should move, but should move equally to remaining nodes.

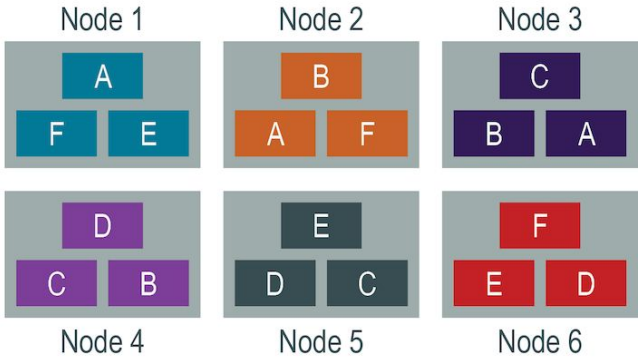
Create multiple token ranges (Vnodes) per node

- Let’s say we want to create 4 vnodes per node, each node chooses 4 random numbers b/n [0, 100)
  - [26, 30, 51, 79] → Node 1
  - [17, 77, 94, 98] → Node 2
  - [6, 22, 42, 100] → Node 3
  - [2, 44, 76, 91] → Node 4
  - [33, 40, 59, 73] → Node 5
- So Now Node 4 takes care of ranges [0-2], [41-44], [73 - 76], [80-91], approximately, equal (this case it’s 23% of keys)
- If Node 4 fails, the mapping would be
  - [26, 30, 51, 79] → Node 1
  - [17, 77, 94, 98] → Node 2
  - [6, 22, 42, 100] → Node 3
  - [33, 40, 59, 73] → Node 5
- So [0-2], [41-42] will be mapped to Node 3, [43-44] will be Node 1, [73-76],[80-91] will be Node 2, approximately equal share of keys by others.

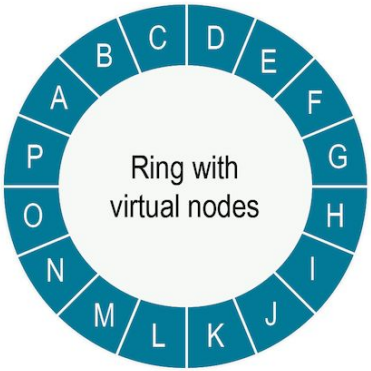
Consistent hashing summary

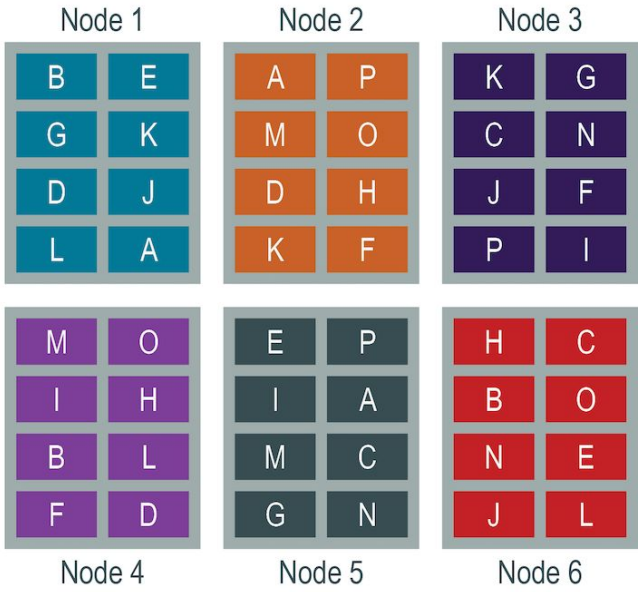
Assume A, B, C, D, E, F are 6 token ranges of 100, the node mapping includes replication factor 3





Break the same token range 100 to 16 ranges and allot multiple ranges to same node, so that even though one node goes down remaining will be distributed more equally to other nodes.



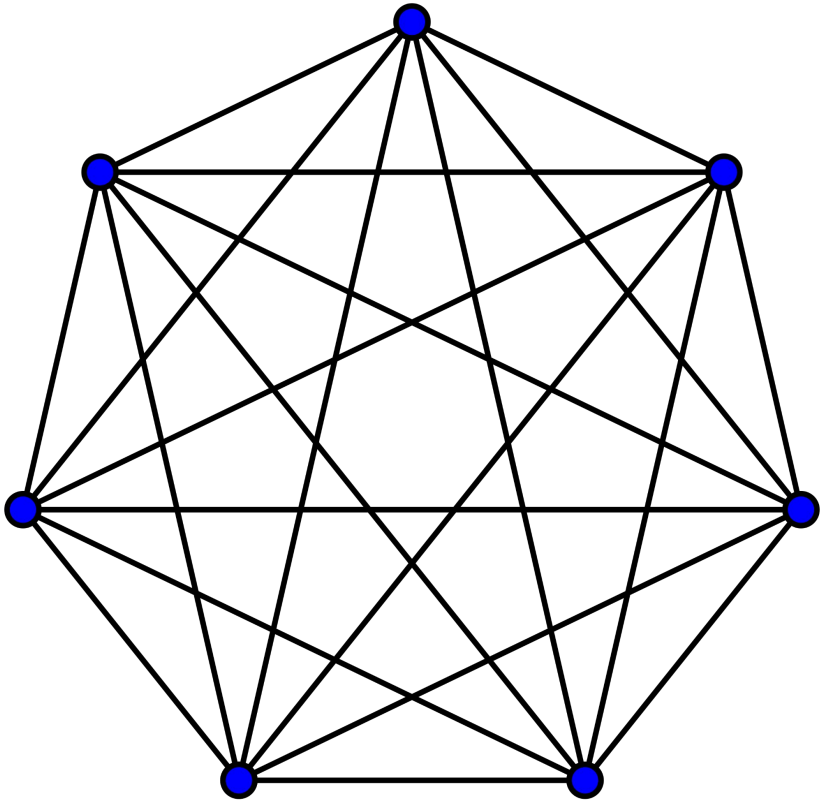


Other alternatives of load distribution

- Rendezvous hashing (HRW algorithm)
- For a given key like 47, calculate hash with all servers for example each node has ids, 1, 2, 3, 4, 5 respectively then, we calculate Hash(Key + Node id) and take maximum
  - Hash(“47” + “Node1”) → 8925
  - Hash(“47” + “Node2”) → 2425
  - Hash(“47” + “Node3”) → 1164
  - Hash(“47” + “Node4”) → 3256
  - Hash(“47” + “Node5”) → 9871
- So we chose Node 5 with highest value, with little math, it can be shown equally distributing the keys
- If a node goes up or down we calculate the hashes with only remaining servers, so if a node goes down, we won’t consider it all, (Similarly adding node will change server list), so we route to live servers only
- If the node goes down, it’ll be random chance that remaining keys while doing addition and hashes all 4 node servers will get equal share of maximum hash values.

Cassandra deployment strategies & challenges

- Decentralised architecture
- No primary or secondary nodes
- Should scale while adding or removing the nodes
- Only separation in seed (set of nodes to start with) vs non-seed nodes
- Often shown as ring, but that doesn’t make sense at all, every node can communicate with any other node in the cluster.



Cassandra deployment at large scale

- Cassandra is written in Java, often get pauses with Garbage collection (ZGC reduce the pause time at the cost of CPU usage)
- ScyllaDB is a great choice, which not only writes the code in C++, it uses optimised stack for CPU scheduling, networking and uses NUMA shards to improve the performance.
  - It adds new algorithms for SSTable compaction to reduce the variance in the latencies
  - Proven to be cheaper (⅓) and better response time (¼ for 99%) than cassandra
- While cassandra says it's completely distributed and there is no single control system, in reality we need to have control plan to do some tasks like
  - Machine going down or up
  - SSD garbage collection
- Uber built a control plane on it's own (odin)
- ScyllaDB built a control plane to manage several clusters at once

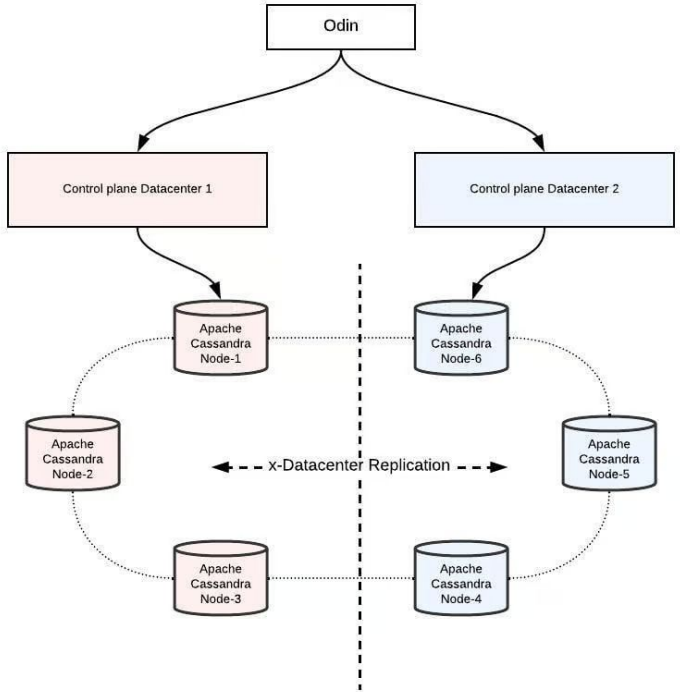
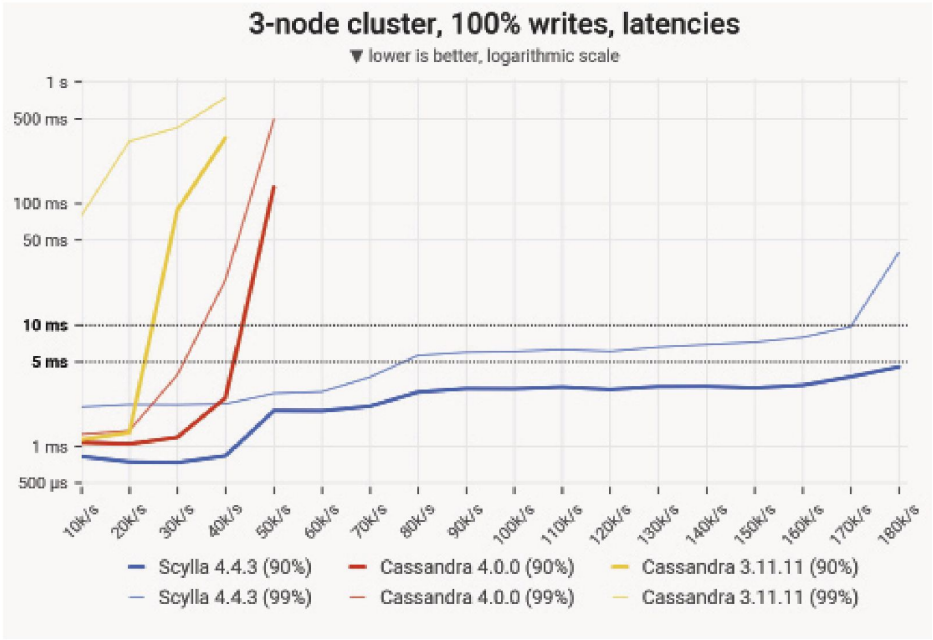
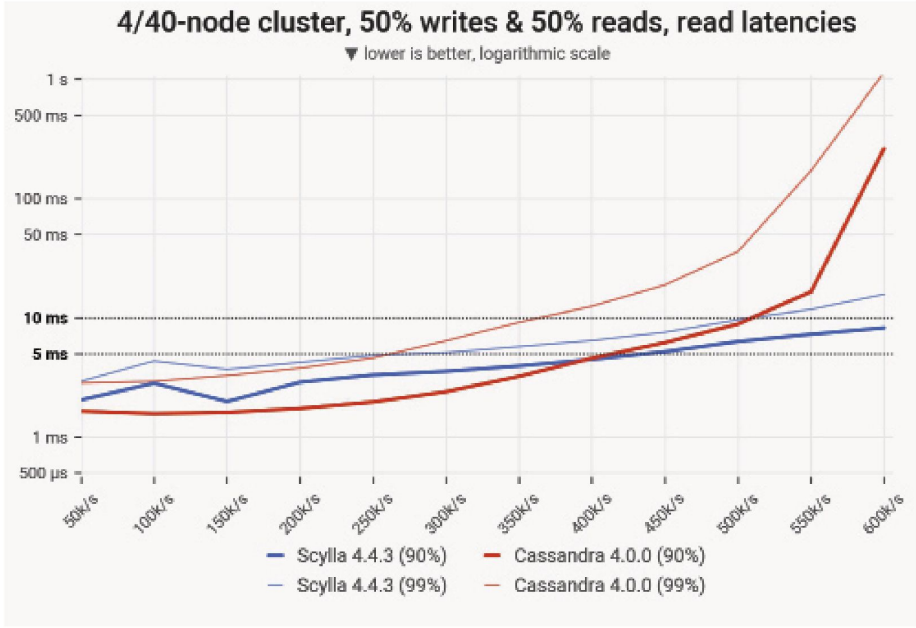


Figure 2: Apache Cassandra deployment

Cassandra vs ScyllaDB performance (from scylladb)



The 90- and 99-percentile latencies of UPDATE queries, as measured on three i3.4xlarge machines (48 vCPUs in total) in a range of load rates. ***Cassandra quickly became functionally nonoperational, serving requests with tail latencies that exceeded 1 second.***



Latencies of `SELECT` *query, as measured on 40 TB cluster on uneven hardware — 4 nodes (288 vCPUs) for Scylla and 40 nodes (640 vCPUs) for Cassandra.*

The main problem in handling multiple writes

- Assume we have 5 replicas for a partition (for a key), and we have 5 nodes
- Every write should be replicated to all replicas
- If 2 clients writes the data for the same key, and same column value (int type)
  - If they write to same replica, it can order the requests and write one after the other
  - If they write to same replica, how to choose one over the other ?
- Use timestamps along with column value
  - For conflict resolution we can use the column with higher timestamp value
  - However, timestamps across two nodes are not synchronised.
    - Assume the “real time” is 10:00 AM, 10:01 AM for writes from two clients each
    - The node 1 sees the request from client 1 (10:00AM) and the local time is 10:00 AM
    - But node 2 sees 10:01 AM request, but local time is 09:59 AM (the drift is common)
    - Even though the node 2 sees the new data, due to lower timestamp, eventually it’ll get overwritten with node 1, (with Gossip protocol)

Handle overwrites

- Overwrites causes loss of data
- Oneway is report client while reading that there is a conflict of data
- Database need to have a mechanism of identifying the conflict version of data
- DynamoDB, stores update count tuple for each row to identify the conflicts
  - Let’s say 3 nodes have 3 replicas of same primary key (p1), three different clients written to three replicas at same time concurrently, so each node stores
    - Node 1 stores p1 as P1 → (1, 0, 0)
    - Node 2 stores p1 as P1 → (0, 1, 0)
    - Node 3 stores p1 as P1 → (0, 0, 1)
  - We can make A ≤ B or A ≥ B relation of these tuples if all counters of A is ≤ counters of B or vice versa, Since you can’t make ≤ or ≥ relation for these tuple, then there is a conflict
- You send a conflict to the DB to handle it.

DynamoDB conflict identification (vector clocks)

- Assume 3 nodes (have ids: 1,2,3), each having a replica of primary key p1, And we have two clients writing to same primary key, and client 1 is writing to node 1, and client 2 is writing to node 2, Initially all nodes have (0, 0, 0) for p1 as vector clock (update counter from each node id)
- If client 1 updates primary key p1, the vectors are (1,0,0), (0,0,0),(0,0,0)
  - Node 1 sends replication requests to all replicas let's say all of them are success in this case
  - Now all nodes will have vector clocks of (1,0,0), (1,0,0), (1,0,0)
- Now, client 2 updates primary key p1, the vectors are (1,0,0), (1,1,0),(1,0,0)
  - Now, Node 2 sends replication to all replicas, let's say only node 3 is success
  - Now all nodes will have vector clocks of (1,0,0),(1,1,0), (1,1,0)
  - Observe that no conflicts so far
- Now, if client 1 updates primary key p1, the vectors are (2,0,0), (1,1,0),(1,1,0)
  - Now, Node 1 sends replication to all replicas, it also sends with old vector (1,0,0), so node 2 and node 3 will reject them
  - When a client reads DynamoDB return all version that's conflicted instead of only row
  - Client needs to update it with new value, while updating new vector clock will be created
    - It reads (2,0,0), (1, 1, 0), (1,1,0), take max of each index (2, 1, 0) and updates the node 1 with (3, 1, 0).

Handle writes without conflict generation

- Assume we have 10 nodes, we create 1000 partitions of the table and we have 5 replicas for each partition, instead of all replicas (1000 * 5) handling writes, we select a leader for each partition and only the leader will write it.
- Approximately each node will have 100 leaders, so total 1000 leaders (writers) writing, and will be load balancing
- The problem is High availability, what if the a particular leader (or node) goes down, the leader switch needs to happen automatically, and short period of time, so we need a mechanism that can do leader selection without split brain
    - Paxos (Multipaxos) for leader selection
    - Raft for leader selection
    - Bully (select node with highest id - bullying all lower ids, slow for selection, not fault tolerant)

Summary

- Multi writers can have write conflicts
- Cassandra uses timestamp to resolve the conflicts and choose the one with highest timestamp
  - Problem is timestamp drift on general purpose machines
- DynamoDB uses vector clocks to identify the conflicts and expects the clients to resolve the conflicts by storing the conflict versions
  - Client side complexity increased
  - Vector clocks take space can grow exponentially (DynamoDB paper said, they never faced issue in production)
- For all replicas in each partition select a leader, and only the leader will be able to write the data
  - Writer availability problem → Solved with consensus algorithms

Cassandra store data in consistent way (without consistency from the cassandra)

- Use conflict-free replicated data types, (CRDT), it helps solving “some” consistency problems without supporting from cassandra
- CRDT landscape is larger, but cassandra supports only few
- For example, if we want to maintain an integer which can be concurrently updated by multiple replicas, let’s say the integer could be inc only
  - Let’s say we have 5 replicas, each replica maintains an array of size 5
  - Each replica will have complete array, so we have 5 such arrays.

0	0	0	0	0

- When client asks for increment each replica owns one index, only increments that index, for replica 2, it updates array[2] only
- Each replica communicate with each other to update the other indices
- If client requests for value of integer, it sums up the local array to return. So eventually consistent
- The underlying data is not simple array type, it uses array of (shard_id, update count, final integer value) for actual implementation.

Other CRDT types

- In CRDT theory, the integer that only grows is called G-Counter
- You can support decrements on integers with two G-Counters, one for increments, and another for decrements, total value is
  - sum(Positive G-Counter) - sum(Negative G-Counter)
  - It’s called PN-Counter
- Instead of integers we can use sets, so we have G-Set to support only set additions, 2P-Set can support removals as well (Here once removed you can’t add element again)
- To avoid that limitation, we use timestamps and allow same element to be added again with new timestamp in the set, so we use the timestamp to check whether it’s deleted or added again, called LWW-Element set
- There are other CRDT types in theory, Cassandra support PN-Counter, simply called “counter” data type, and LWW-Element set, simply called “set” and “map” in cassandra.
- Cassandra “list” also maintains timestamps for each element, to maintain order. Since list supports multiple appends, and appends can happen from any other node, timestamp is ordering attribute of “list”

Cassandra Transaction support

- Let’s say if we want to transfer 100$ from x to y (primary keys x & y)
  - Requires x -= 100, y += 100 and needs to happen atomic way, if x & y are in two different partition cassandra doesn’t support atomicity containing multiple partitions
  - As we know till April 2023, there is no support of transactions
- Cassandra support row level conditional updates (called Light weight txns)
  - `UPDATE table SET balance = balance - 100 WHERE id = x` **`IF balance >= 100`**
  - If the condition is satisfied then only updates the rows, useful in optimistic locking of rows
  - `UPDATE table SET version = version + 1 … WHERE id = x` **`IF version = 5`**
- For transitions we need locking and two phase commits, Cassandra was reluctant to implement citing that this would impact performance and two phase commit requires at least two round trip among the partitions in transaction.
- In Oct-2022, there is a groundbreaking paper(from Apple engineering team), where cross partition transactions can be done with only round trip (most of the time), Cassandra planned to implement it in Cassandra 5.0 (Update: as of Jan 2024, not released yet!)

Cassandra LWT transactions

- Uses 3 round trips using Paxos, so it’s slower and resource intensive
- If some LWT txn gets timeout, the user is expected to run nodetool repair command manually, if another statement updates the row meanwhile there are in consistency issues
- In general it’s not recommended to use LWT from cassandra
- Cassandra 5.0(planned) trying to use Accord and try to provide cross shard txns
- ScyllaDB 6.0(planned) trying to use Raft for LWT txns.
- Until these things matured, don’t try to to use LWT in production

Consistency first distributed systems

Consistency first distributed system

- A distributed system which prefers consistency. I.e., in case of network/node failures, sacrifice availability, but it will provide only one view insteading of multiple replicas providing multiple values.
- For fault tolerance, we need to have distributed system, that’s having multiple replicas of same data (partition or primary key)
- There are two models
  - Master slave → Only one master writes, slaves can be used for reads (with some conditions), and usually requires 2m+1 servers, if we want to have fault tolerance of m servers.
  - Master-Master → All masters are possible to write, the models which requires consistency are in research, and usually requires, 3m+1 servers, if we want to have fault tolerant of m servers (E-Paxos is released in 2012, and Accord is latest one)

The problem of selecting a leader in distributed environment

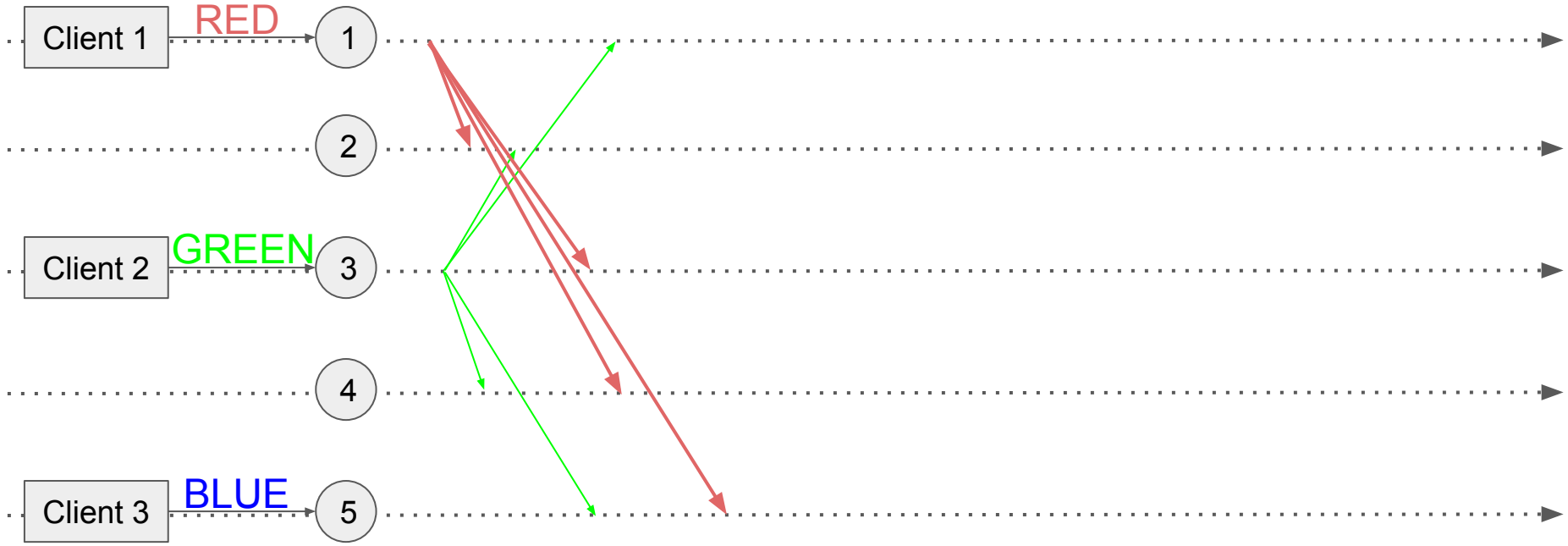
- We have multiple systems, that want to reach agreement (consensus) for leader or in general on ‘some value’
- The system should be fault tolerant, if one or few nodes dies, still we should be able to get the consensus
- We focus on getting one value (only one value) should be agreed (or accepted) by majority.
- If we have 2m+1 servers in a group, as long as we get m+1 servers agreed/accepted on only one value, we treat it as consensus.
- If some servers are down during consensus, and if they come back, the mechanism should allow them to agree on the existing value, rather than coming up with new value.

Let’s a build a distributed consensus algorithm - Paxos

- Assume we have five nodes, have ids 1, 2, 3, 4, 5.
- The clients can contact any number of these five nodes. For example three clients can send one request each to 1, 3, 5, requested different colors to paint - Red, Green, Blue
- The goal is that at least majority (3) of the servers need to get to one color
- Let’s call severs 1, 3, 5 as proposers, since they get calls from clients and they supposed to bring consensus in the group and respond success to the client. Note that here only one client can get success response, since only one value has to be chosen, Think color selection as leader election, there should be only one leader (for one term).
- Let’s evaluate options

Consensus for 5 nodes

Assume server 5 is slow to communicate with other servers & Node 1, 3 broadcast that they want be “The proposer” and have “RED” and “GREEN” values with them resp.



Goal is at least 3 servers agree to same color over a period of time (→)

Several options to try to get 3 servers one color

Option	Description	Problem
Choose first one	Let each proposer (request from client) broadcast to all servers and each server votes (or accepts) the first color it received.	There is a chance that 1 & 2 vote for RED, 3 & 4 vote for GREEN, 5th one votes for BLUE, We can't get majority servers (3) to get one value
Give a preference to proposer with higher node id	When a proposer sends a request to other servers, the server can accept it, however it can override when another proposer with higher node id sends another request (another color)	Let's say there is small delay among proposers(Nodes 1, 3, 5). Let's say 2 & 4 accepted RED color from proposer 1, but when they(2 & 4) received BLUE color from proposer 3, they will override, in this case both clients get success, that shouldn't!
Prepare first - Find who can propose, what they can propose	We need to choose one proposer for coordination, and proposer chooses value and asks everyone to agree on that. The value can be from other proposer too.	Multiple round trips, but one of the best solution available - called paxos

Consensus for 5 nodes

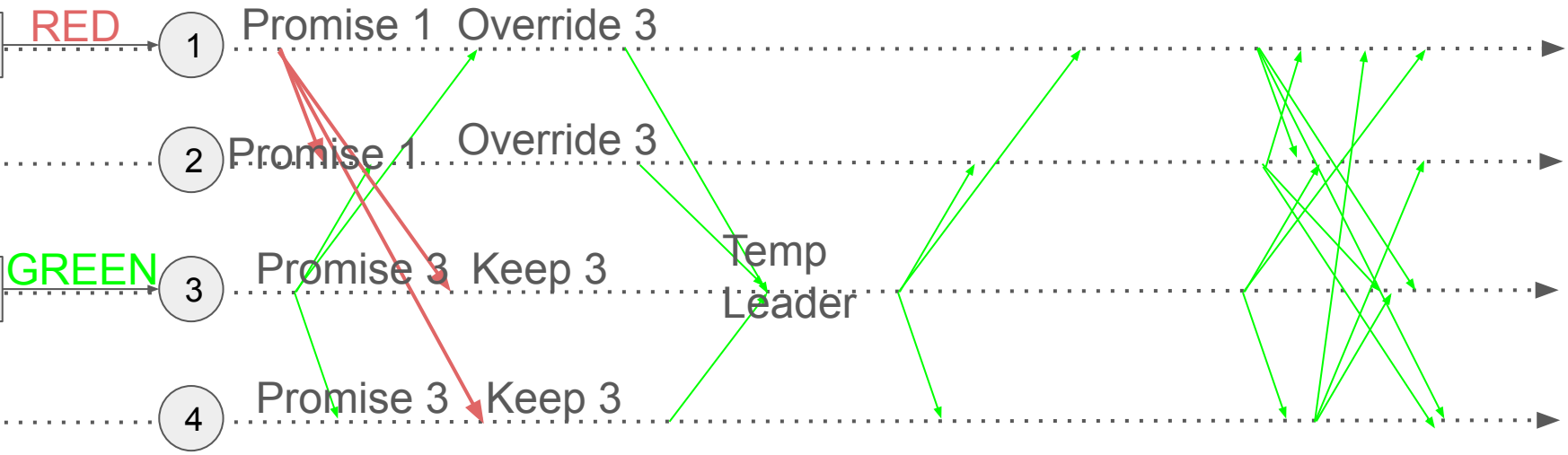
Assume server 5 is slow to communicate with other servers & Node 1, 3 broadcast that they want be “The proposer” and have “RED” and “GREEN” values with them resp.

- Node 1, 2 received RED from Node 1, and then received GREEN from Node 3,
- Node 3, 4 received GREEN from Node 3, then received RED from Node 1,
- How can we make all agree to one proposer ?
- First Node 1, 2 agree for Node 1 as “The proposer”,But If Node 3 sends before actual value is being given to them (before accept phase), they reelect Node 3 for “The proposer”

Consensus on 5 nodes

- Node 1 & 2 override Node 3 as the proposer since it came from “3”, compared to previous proposer Node of “1”
- Node 3 & 4 selects Node 3 as proposer, and won’t override since the next request is from smaller Node id “1”
- So 4 nodes reply PROMISE to Node 3, and Node 3 will become temporary leader, Note that Node 1 can’t get 3 PROMISE’s in this case.
- Only Node 3 is temporary leader, it sends the value “GREEN” to all 4 nodes, and wait for acceptance to them, Since Node 5 is down and no other requests are there, all servers will accept “GREEN” with proposal id as “3” (Node id). Since they promised for proposal id “3” earlier (if other one tries to send the value for acceptance, they would reject them)
- Every node (1,2,3,4) that accepts “GREEN” value broadcasts to all other nodes (called “learning phase”, and node role is called “learner”). So that everyone once it receives the 3 accepts, it knows “GREEN” is choosen

Consensus for 5 nodes



This is **prepare phase**,
select one temporary leader,
the server will choose the value.
The server is chosen by Node id

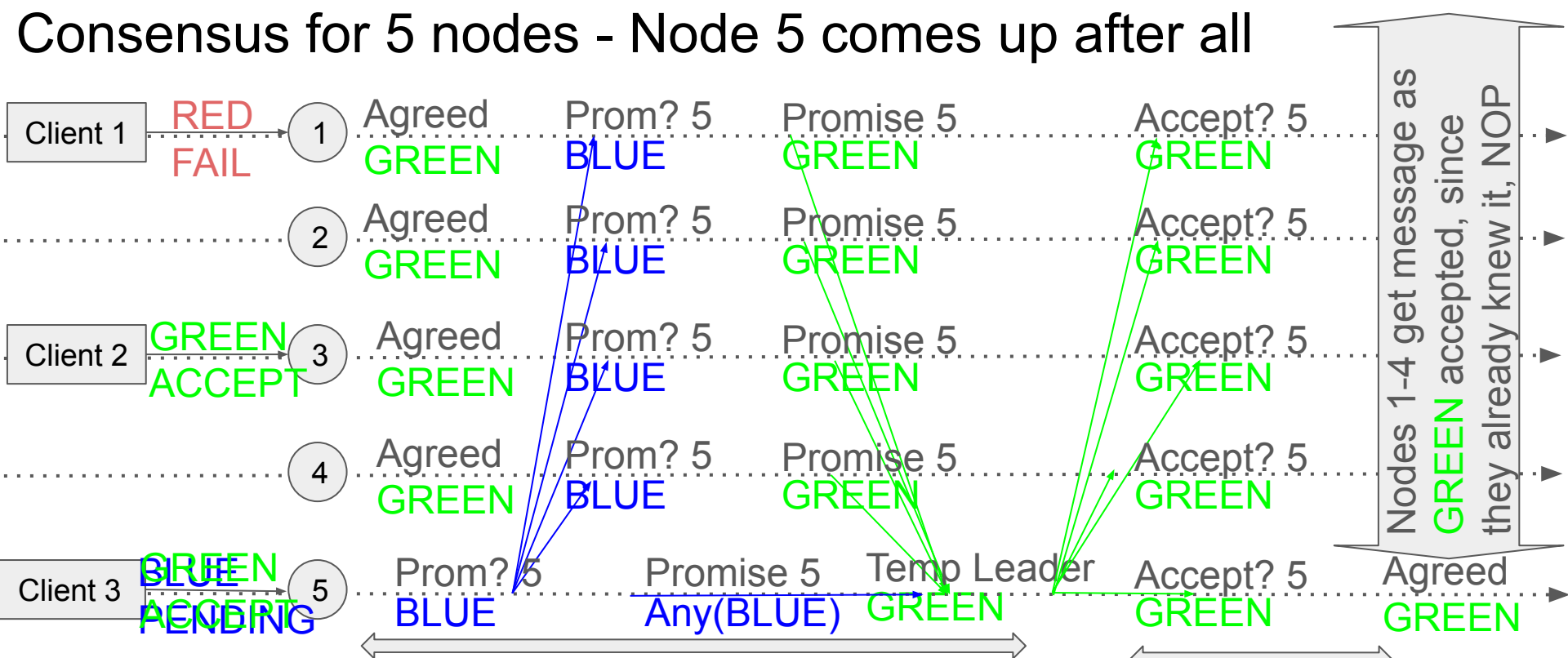
accept phase,
GREEN value
accepted by
Node 1, 2, 3, 4

learn phase,
Every node listens for 2
other acceptances, and
confirms we reached
consensus on GREEN

Consensus on 5 nodes

- Assume Node 5 comes up now(after they reached consensus) and started sending PREPARE messages to everyone
- Will other Nodes reject the PREPARE messages ?
  - No, They say they PROMISE, but
  - They will only accept “GREEN” value (since they have accepted)
- Node 5 would get 5 PROMISES, but all replies contains they will accept only “GREEN” value.
- Even though Node 5 becomes temporary leader, it can’t propose “BLUE”, it’ll send “GREEN” in accept messages.
- And every acceptor will send a broadcast to everyone else that “GREEN” is chosen, and other servers already got majority so they don’t care about this broadcast
- Summary is “GREEN” stayed once it’s selected, So only Client 2 get SUCCESS response, Node 5 returns to client 3, that “GREEN” has been chosen.

Consensus for 5 nodes - Node 5 comes up after all



This is **prepare phase**,
Node 5 asked all nodes for temp leadership
All nodes agree, Node 5 is **okay with Any**
Node 1-4 says they can take only **GREEN**
So node 5 can ask to accept only **GREEN**

accept phase,
GREEN value accepted by Node 5 now, others already accepted, but they broadcast to every other node

Consensus on 5 nodes - Case of Node 5 didn’t come up, Node 3 lost after elected as temporary leader

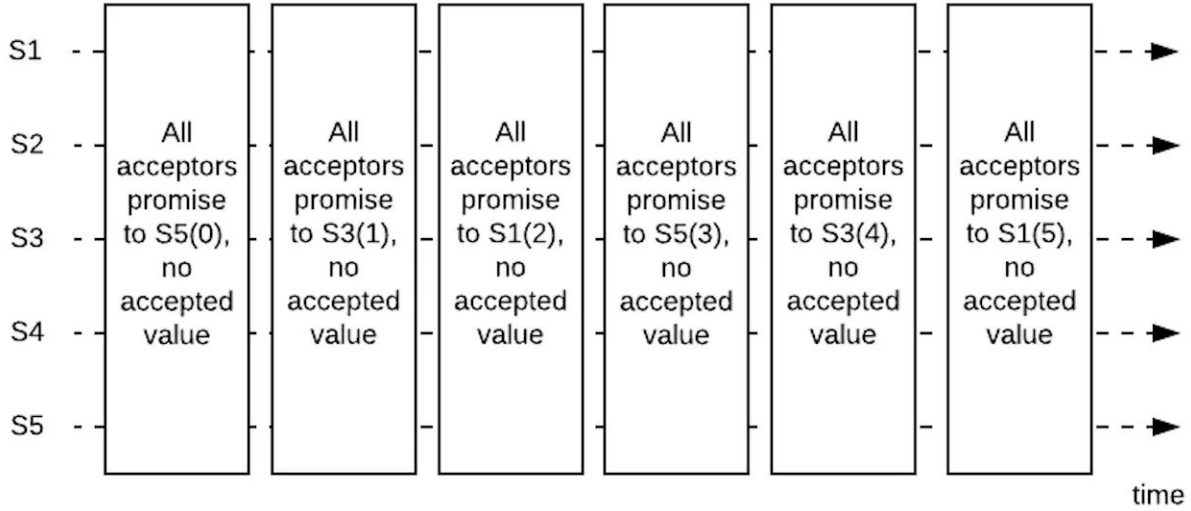
- Assume the first case with 4 nodes, where Node 3 elected as temporary leader elected by Node 1,2,3,4. But node 3 went down. Remember everybody rejected Node 1.
- Instead of leaving Node 1 aside, it’s better to get consensus on Node 1 value - RED. So how can we achieve this ?
- Node 1 can retry after sometime, But wait, Node 1 can’t be elected as leader since Node 1-4 chosen id - 3, Now they can’t accept anything below that ? How to solve the problem ?
- Add a timestamp as prefix* to each id, so instead of nodes sending only node id as their proposal id, send “timestamp”.”nodeid”, this will be used as proposal id.
- So, when node 3 sends proposal id initially; it would use 10:00:00.3 as proposal id, assume node 1 sends proposal after 5 sec, it would use 10:00:05.1 as proposal id, since this id is larger than previous id, Nodes 1,2,4 (3 is down) would accept this proposal and continue.
- There are other options for prefix like monotonic counter, slotted timestamp, but idea is same

Summary of paxos

- **Prepare phase**
  - Proposer choses a proposal id, and send the proposed value to all nodes (called acceptors)
  - Acceptors tracks proposal id, if this is not the highest so far, they'll reject it. If this is highest
    - Check if there is a value already accepted, prefer this accepted over proposed value
    - Send PROMISE success response to proposer with proposal id, accept id, accept value
  - If proposer gets majority success, the proposes goes to accept phase, if not wait and retry.
- **Accept phase**
  - Proposer sends the chosen value (either from it's proposed or from prev accept value)
  - Acceptors will check if this is same proposed (there is chance that other proposer might override this with higher proposal, either retry time or another client requested another node)
  - If it's the same proposal they PROMISEd in the prepare phase, they accept it. If not they reject it.
  - As soon as they accept, they send ACCEPT message to proposer and all nodes too (called learners)
- **Learner phase**
  - Every node counts the accept counts, and they know the total number of nodes in the system. If it crosses majority (at least m+1 in 2m+1 nodes), they learn that it's a "consensus value"

Is paxos 100% safe ?

- If proposer fail before getting acceptances, they'll lose the data too
- Bigger one is live locks, Assume we have 5 servers as shown (s1, s2, s3, s4, s5)



- Let's s5 chosen proposal id = 0 (complete id is 0), and completes prepare phase with id = 0, before sending accepted value, let's say s3 chosen proposal id = 1, and completes prepare phase.
- Even before s3 starting accept phase, s1 becomes temporary leader with proposal id = 2
- Again s5 could come (with retry) and proposal id = 3, and before any accept phase, and completes prepare phase, this goes on…
- Can happen forever, but with random ness (with exponential backoffs) in retries, the probability is low (not zero)

Paxos in real world

- For concurrent writers, you can paxos so that all replicas will agree to only one client request, Cassandra and Scylla uses them for LWT transactions (they have bad implementation though)
- For leader writers, you still need paxos for leader election. Having an automated algorithm like paxos can bring up leader in 100s of milliseconds.
  - Spanner selects leader with paxos, and leader will take all writes and replicate write log in sequential manner across all replicas, they call it as multi paxos
- Paxos is first one to be mathematically proven to have consensus, but difficult to understand and implement it.
- Raft is introduced in 2012, and have simple and understanding logic for leader election, the algorithm or variants used in several distributed systems
  - Zookeper → ZRaft
  - Kafa → KRaft
  - etcd (Kubernetes store for objects) → Raft

Beyond Paxos & Raft

- Raft still needs leader election, and a single writer algorithm. Since we have distributed systems break into partitions, scale is not a problem (in general)
- Consider a partition with 5 replicas in Spanner (called paxos group), the replicas are geo distributed, US-West, US-East, Europe, India, Singapore. Assume this partition leader is in US-West
- Let’s say user(in this partition) in India writes to a Google doc, the request first goes to Data center in India, Even though India data center has the replica, it is only read replica, for write you need to go to US-West (the data travelled from India to US-West)
- US-West needs to write data to at least 3 replicas (5 replicas in most cases), then the writer in US-West writes to India replica, (the data travelled from India to US-West), once the US-West get a success response from India’s replica, it sends success response to User.
- So two round trips have been consumed. The latency b/n India and US-West is at least 65 ms, (usually 80+ms), limited by light speed. How to do it in one round trip is research area.

@danheller 14 years ago

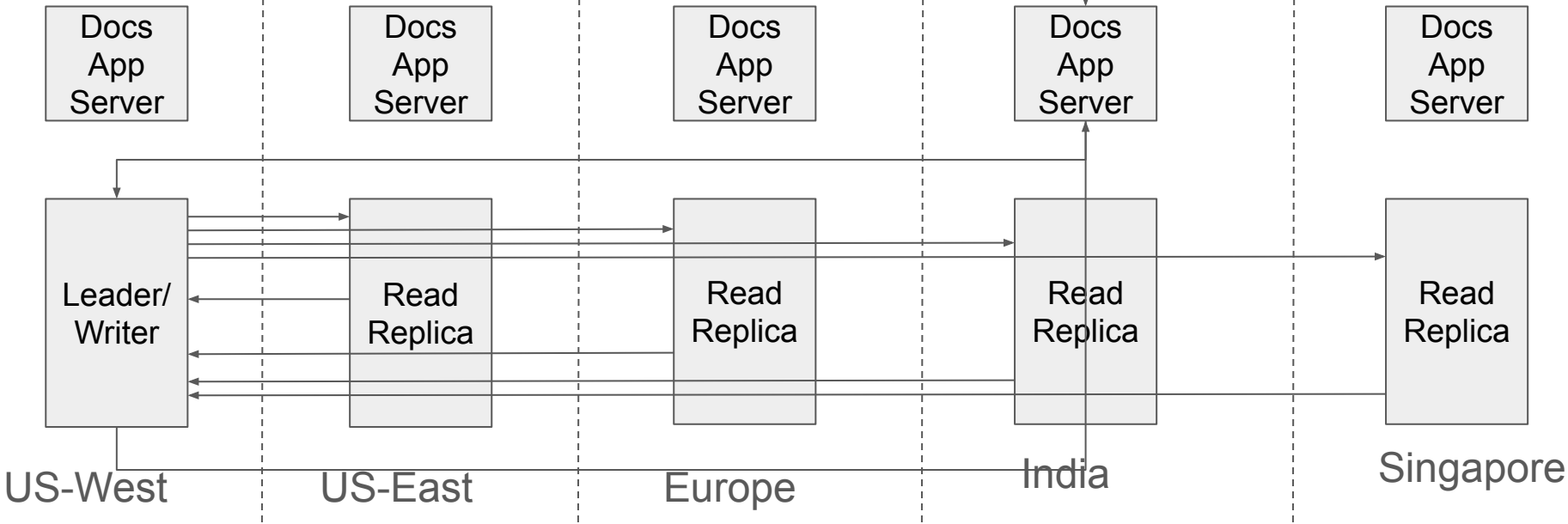
Only Google complains about how slow the speed of light is...

94 **Reply**

Single writer replicating log - two round trips

Too slow, the latency is 80 + 60 + 60 + 80 = 280ms

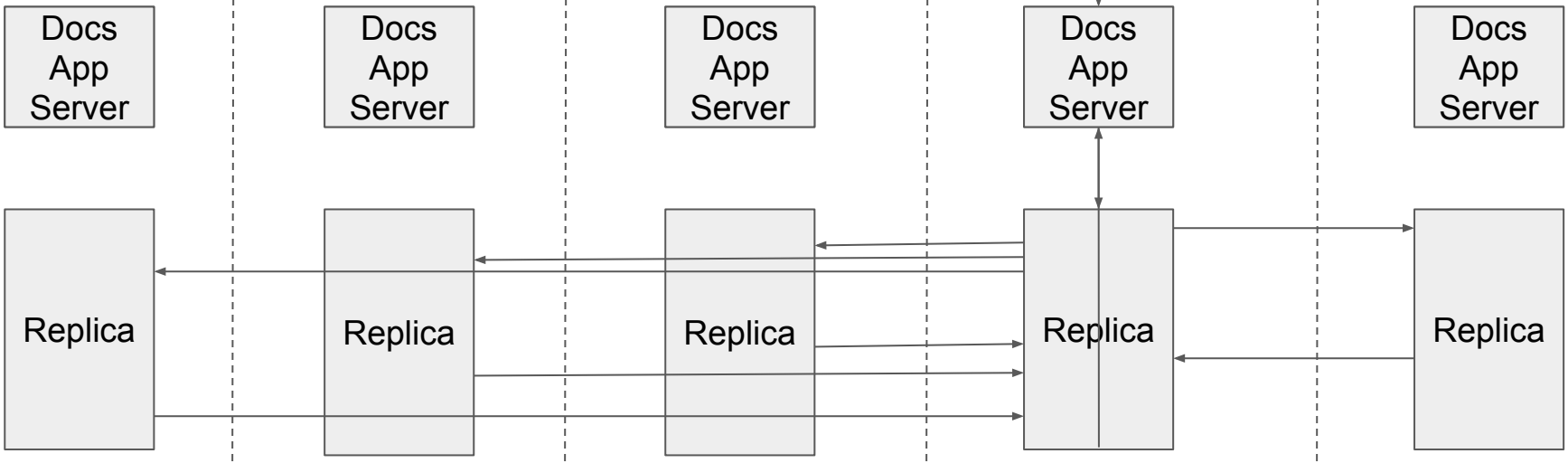
Write req to doc



Single writer replicating log - two round trips

Quite better, the latency is 0 + 70 + 70 + 0 = 140ms

Write req to doc



Beyond paxos and raft

- Everybody should be able to write as shown in the previous slide, For example Indian users will use India’s replica to write, US users will use US replica to write
- E-Paxos is the first known algorithm to solve that, but comes with a cost, we need to have 3m+1 replicas to have fault tolerance of m replicas
- There are several protocols like Tempo, Time Order Queue (TOQ) tried to improve the fault tolerance (have few extra nodes for fault tolerance) and round trips.
- Recently a groundbreaking paper called Accord tried to solve most of the problems (We often don’t see such papers especially in distributed system), that’s trying to solve multiple things at once.
    - If we don’t have conflicts in the requests, Accord can complete the data write in 1 round trip.
    - Requires only 2m+1 replicas to have fault tolerance of m servers
    - Requires very simple clock (not sophisticated one we find it in Spanner)

Building spanner

- As discussed earlier, it’s data is partitioned and replicated across multiple machines.
- Each partition and it’s replicas called paxos group, each paxos group has a single leader, which takes writes
- Since the cluster has multiple machines, and each machine has multiple partitions and leader selection is random, it’s expected to have equal number of writers for each machine - so writer load is distributed
- It provides SQL interface for CRUD operations and DDL to create/alter tables
- Spanner partition data by ranges and ranges to partition id mapping is stored in metastore, idea is that it will help in range queries.
- Since spanner stores consecutive ids in same partition, having a primary key with auto increment will always writes to same partition. So spanner want to discourage that - and didn’t implement auto increment primary key

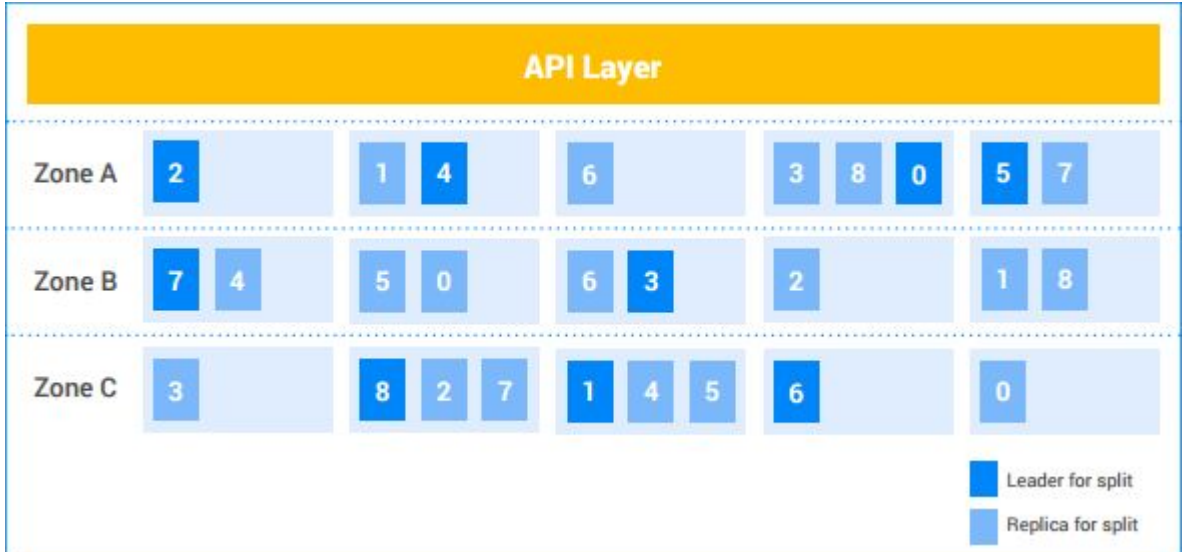
Spanner storage layout

Split	KeyRange
0	[-∞,3)
1	[3,224)
2	[224,712)
3	[712,717)
4	[717,1265)
5	[1265,1724)
6	[1724,1997)
7	[1997,2456)
8	[2456,∞)

Assume a simple table is created, with id, value columns, and id is an integer column and primary key

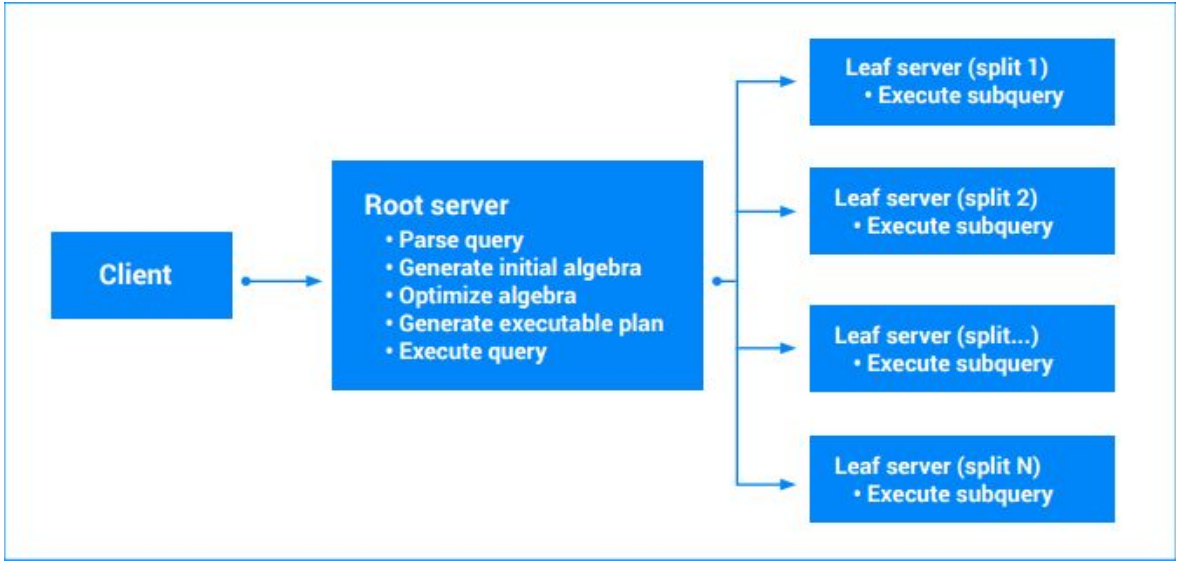
```
CREATE TABLE ExampleTable (

 Id INT64 NOT NULL, Value STRING(MAX),) PRIMARY KEY(Id);
```

Assume we have 5 nodes, and want to create 9 partitions and have 3 replicas, and one replica in each availability zone. Spanner brings API layer to run the queries and automatically route requests to corresponding split (partition)



Spanner query execution

- A DB Client (Usually application server) submits the request to root server (api layer) and Root server executes the query.
- Leaf servers executes subqueries and assist the root server to execute full query
- Assume a txn, where we need to transfer $100 from x to y, and assume x is in split 1, and y in split 2, leaf server 1 executes update on x, leaf server 2 executes update on y
- Root server make sures either both of these sub queries get executed, or none of them, to get consistency guarantees.



Transactions in Spanner

- As we have seen in transactions, we have following isolation modes
  - READ_COMMITED → Read only committed rows in transactions
  - REPETABLE_READ → Read same data if query repeatedly in the same transaction
  - SERIALIZABLE → Execute transactions as if they are executed serially (in single flow)
- Spanner doesn't allow transactions to choose isolation levels, It's by default more stricter isolation level EXTERNAL_CONSISTENCY.
  - Two transactions looks like they're executed in some serial order (SERIALIZABLE) + they executed in the order of the timestamps.
- Spanner uses MVCC (keeping previous version of data) to read only committed rows (READ_COMMITED semantics), read same dataset or dataview (REPEATABLE_READ + SERIALIZABLE).
- Spanner supports Read only txns and Read write txns
- For read only txns - At the start of the transaction spanner takes timestamp from Truetime (same time across all machine with sophisticated hardware), and uses the timestamp as version to read the data and avoids phantom reads and locks too.
- Spanner transactions are done at particular column of a row, called cell (think of cell in excel), it tries to lower the congestion as much as possible.

A Locking mechanism to get Serializability

- When two txns T1 and T2 gets executed in distributed (or multithreaded) environment the output should be one of the serial order of execution, it could be either T1 → T2 or T2 → T1, but you can't have some extra effects being executed in concurrently underhood.
- As we have seen, to achieve serializability, it's important to
  - Acquire locks before reading the rows, and don't release the locks till commit is completed.
  - We can allow concurrent reads, while preventing any writes. It won't impact the serial order.
  - For write take exclusive lock, only one write is allowed, and keep till commit is completed.
  - We need to avoid phantom rows, especially for range queries, we should avoid new rows inserted (in another transaction) after the transaction started.
    - MySQL uses GAP locks to avoid additional insertions
    - Spanner user Range locks to avoid additional insertions
- This make sure one transaction executed without seeing the effects of the other txn
  - If any other txn writes on the current txn items, they should be completed, till that completion, current txn can't start due to locks on the items.
  - If the current txn gets locks, other txns can't modify the data items (and query results), so the effect of the txn is as it is completed on a single execution. So we can ensure serialization.

External consistency transaction semantics understanding

Transaction flow (ideal world)

- If txn does any reads, take just *ReadShared* Lock, so it'll stop any writes to the cells
- If txn writes, keep the changes in buffer
  - If it's read before - take *ExclusiveLock*, so it'll stop any access to that cell
  - If it's writing without read (blind writes) - take *WriterShared*, stop all reads to that cell, writes allowed
- Before commit get ts := Truetime.now()
- Commit all writes with the above timestamp.
- Release all locks acquired in txn only now.

Observations

- For all cells we read or read + write in txn, prevent writing to it during the txn, and this guarantees that cells won't change during txn (as in REPEATABLE READ), if we query by range spanner takes lock on range to avoid phantom reads (as in SERIALIZABLE)
- Same cell can't have same txn timestamp for two blind writes. So it keeps both versions in the history.
- If two transactions T1, and T2 and timestamp(T1) < timestamp(T2), T2 will see all effects (writes) of T1. flow would be
  - Timestamp noted by T1, then written
  - Released locks by T1
  - T2 acquire the locks, Timestamp noted by T2
  - T2 writes the data and releases the locks
  - So T2 is guaranteed to see the all the writes of T1

But the world is not ideal

- Even though Truetime is using sophisticated hardware + software, it won't be able to give the accurate time across cluster of nodes across globe.
- However it reduces the uncertainty to <1ms (for 99% of the time) across large cluster, compared to NTP, which will be 10-20 mins (for 99% of time).
- We need to redesign the algorithm, which guarantees accuray, at the loss performance
- We have seen that locking mechanism is good to achieve serializable, but to get external consistency we need a mechanism, where each transaction gets unique id (preferably monotonic integer) and two txns ids should follow the same order as time.
- And we need to guarantee that two txns with txn_id1 < txn_id2, the txn with txn_id2 should see all the data of txn_1.

Solve for uncertainty → Use cluster's earliest time and latest time

- Truetime can't provide exact time, but can provide global cluster earliest time and latest time. ⇒ Truetime.now() returns [earliest time, latest time], note that 99% of the time, the window size is <1ms.
- During txn, at the commit time, if we take *Truetime.now().latest*, as txn timestamp (ts), and wait till the time passes and whole cluster is guaranteed to have this timestamp, i.e., wait till the ts < *Truetime.now().earliest*
- This is called commit wait and every txns wait this period, since it's in <1ms for 99% of time and spanner usually have 3-7 replicas the time taken for replication will overshadow this time, so in general we don't see the effect of commit wait time.
- But wait, How does this guarantee timestamp of txns are really ordered in order of execution ?

Reasoning for latest timestamp as timestamp for txn

- Assume we are doing a txn T1, and the Truetime.now() interval is [10:06, 10:08] and real world time is 10:07, assume we have take some middle value like 10:07 as txn timestamp and released the clocks realworld time @10:08
- Assume another txn T2, and the Truetime.now() interval is [10:00, 10:10], and real world time is 10:09 and reads the data written by txn t1 and writes with timestamp of 10:05 (middle of 10:00 and 10:10),
- So we have T1@10:07, T2 @10:05. However the txn T2 has seen the writes of T1 but having lower timestamp, this violates the external consistency, you need to have transaction timestamps reflecting the order of data update history.
- If we choose largest timestamp for T1, we would choose T1@10:08, and for txn T2 the timestamp would be T2@10:10, it follows external consistency and when T2 chooses 10:10, it knows that all the other transactions started before this txn were…
    - If they have smaller timestamp, it's guaranteed that time passed already across the whole cluster, so it can read them
    - If they have larger timestamp, it won't see the effects of the txn.
- What if we have T1 Truetime interval is [10:06 10:14] and the real world time is 10:07, we write with T1@10:14, and since we wait till truetime passes 10:14, other txn won't see the effects till 10:14.
- Meanwhile if T2 with Truetime interval [10:00 10:10] comes at real world time @10:09, it won't be able to see the effects of T1, and it's timestamp will be T2@10:10, so casualty follows.

Reasoning for waiting till cluster min time passes txn ts

- Assume the real world time is 10:01 at commit time for txn T1, however the Truetime has returned [10:00, 10:10] (10 sec interval). The txn time is 10:10
- Assume next txn (T2) came, and real world time is 10:02, hower the true time said it's [10:01, 10:04] (3 sec interval). The txn time would be 10:04 and this txn shouldn't see the effects of prev txn (T1).
- To avoid all such txns seeing the effect of transaction T1, we can't release the locks till 10:10, since we wait till Truetime.now().earliest crosses 10:10, after that it's guaranteed that all transaction timestamps across the cluster will be more than 10:10, hence only later txns can see this data. Essentially commit wait is to hide data written till whole cluster timestamps will be higher than the current txn
- Note that even though in real time, two transactions have been started at 10:01 and 10:02, their timestamps would be 10:10 and 10:04 (reverse order),
- However if the two transactions data cells depend on each other, then T2 can't get locks at all till 10:11 and will be able to commit after that, so external consistency is maintained.

Engineering marvel - Truetime

Truetime in 2012 paper from Google, Now a days it's improved and 99% of the time is in order of ~200µs.

The data is collected from thousands of servers spread across 2200 km apart, 2 *ϵ is time different b/n clusters latest time and earliest time.

The example is special case of spike (production issue)

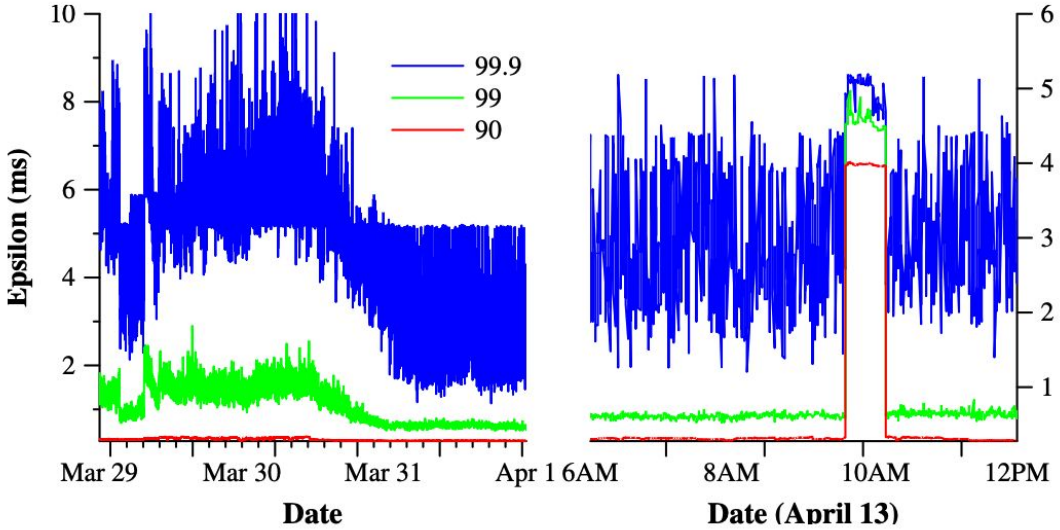
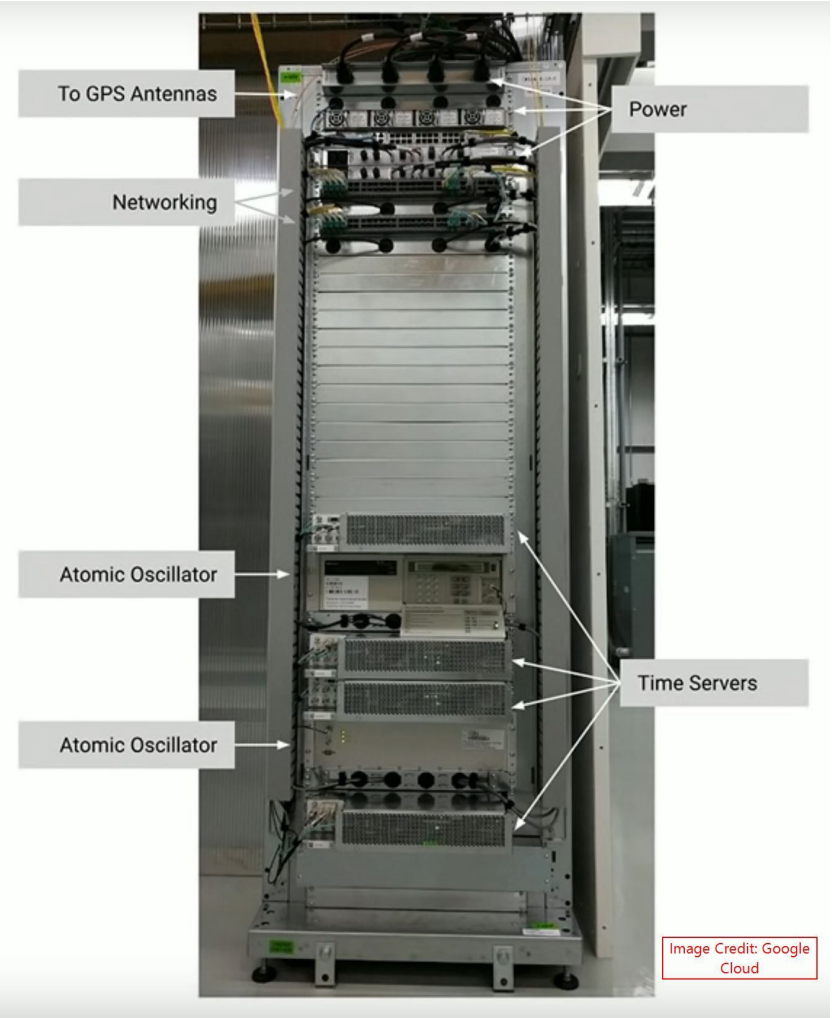


Figure 6: Distribution of TrueTime ϵ values, sampled right after timeslave daemon polls the time masters. 90th, 99th, and 99.9th percentiles are graphed.

Truetime server implementation

- For each server (or wrack) add GPS Antennas
- GPS is sometimes unreliable (can be hacked too)
- So add Atomic Oscillator (atomic clocks)
- Servers will connect with GPS antennas or atomic oscillators to get the Time.
- However the equipment is expensive, even then you can connect the hardware to 4 machines on the wrack, you can’t install all these antennas and oscillators.
- Call these special servers as “Time Servers”
- Other servers simply use daemons to connect with multiple time servers and use local clock to calculate the Truetime interval bound



Global level setup - Truetime

- 10’s of data centers or zones across the globe
- In each data center we have some “Time Master” whose racks equipped with special hardware, we have seen two types of servers with sources
  - GPS Time Master
  - Armageddon Time Master (atomic clocks)
- All the other servers connect to the Time masters across multiple data centers.
- Each server gets multiple values and removes any outlier and use local clock to calculate uncertainty
- If local server clock is misbehaving or GPS or Atomic clock is misbehaving, there are detection algorithms to identify such servers (and remove them)

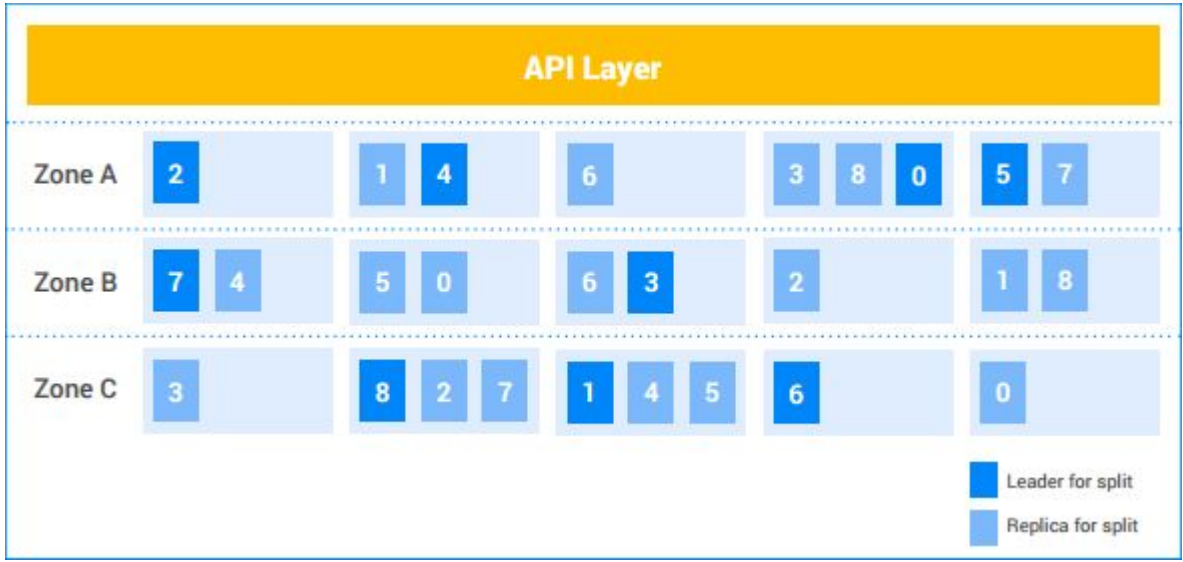
Truetime use cases in spanner

- We have seen that it helps in temporal ordering of transactions in distributed environment
- Paxos leader automated election without split brain
  - Paxos leader will send heartbeat to all the replicas with ts = current timestamp + 10s
  - All followers will see this timestamp, if the timestamp is after Truetime.now().earliest, they realise that leader is down and elect new leader.
  - So no two leaders will be elected at anytime
- Read only transactions without locks
  - `BEGIN RO;`
  - Read Truetime.now().latest as ts
  - Read the data from database with timestamp <= ts
  - `CLOSE;`
  - So no locks needs to be taken, since we are reading some data which won't change (past history won't change, and spanner being MVCC, maintains old data with timestamps for 4-8 hours)
- Consistent reads without reading from multiple replicas
  - Once reads choose a timestamp, and they check if the nearby replica (need not to be leader, leader could be far away in USA, where client is in India) has the all the data till that timestamp, if the replica have that it'll read from replica.

Spanner transactions - beyond one partition

Spanner can do atomic transactions while writing data to multiple partitions (called tablets), without losing external consistency

Spanner replicas one tablet (partition) data across multiple servers, together called as paxos group. We have 9 paxos groups (0-9) in the group

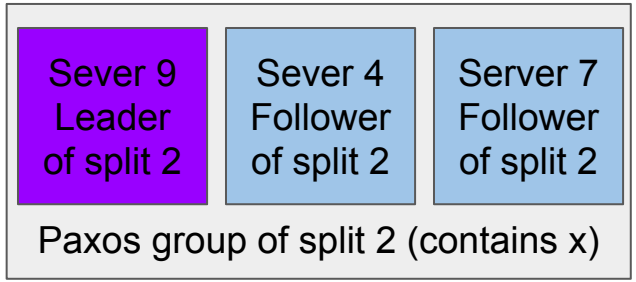


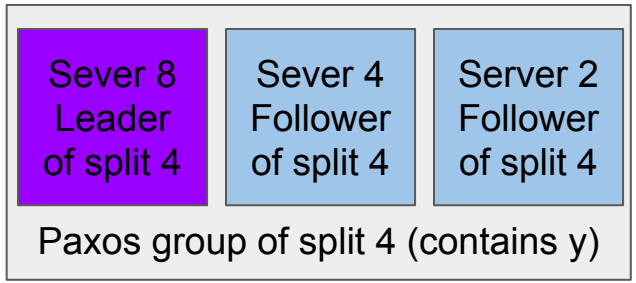
Assume a txn, where we want to transfer $100 from account x → y assume account x is in split 2, and account y is split 4.

Goal is running txn, that either completely does the transfer or nothing, it can't be case of only debit (in split 2) or only credit (in split 4),

Transactions across multiple splits

- All the reads are done with ReadShared Locks, and writes are buffered without locks
- At the time of COMMIT statement, One of the leader is elected as coordinator(TC), assume server 9 (split 2 leder)
  - Prepare phase
    - TC sends prepare message to all paxos groups (participants)
    - All participants will acquire write locks and make sure read locks acquired earlier are not lost.
    - All participants do consistency checks and store it local buffer/disk.
    - Response success or failure to coordinator
  - TC decides to commit only if all participants return success
  - TC calculates commit ts := Truetime.now().latest
  - TC stores the abort/commit decision on it's paxos group
  - TC broadcast abort/commit ts and metadata to all participants (incl. self)
  - Second-Phase
    - All participants apply the writes to cells
    - Release locks
  - Reply success or aborted (if it can't commit) to the client





Two phase commit limitation and spanner solution

- In two phase commit in the prepare phase if any participant returns it can’t commit (write), it will return failure and txn aborts and client receives failure.
- After prepare phase the TC stores that this particular txn is committed.
- So if any participant goes down in the commit phase, when the participant restarts it’ll check with TC to get status and use that information to abort or commit later.
- However if the coordinator (TC) fails in prepare phase, it’ll throw error to client
- If TC fails after local commit (middle phase), then the txn is done
- It’s expected that one of follower in paxos group of TC, will come up and sends commit/abort message to all participants in the group
- Since spanner has good availability with very low turnaround time, this is okay, if systems don’t have such high availability two phase commit is a problem

Spanner two phase commit scaling and latencies

- Spanner can scale with complex case when multiple participants involved.
- As we can see even if there are 50 participants in the txn, the 99% is around 100ms.

participants	latency (ms)	
	mean	99th percentile
1	17.0 ±1.4	75.0 ±34.9
2	24.5 ±2.5	87.6 ±35.9
5	31.5 ±6.2	104.5 ±52.2
10	30.0 ±3.7	95.6 ±25.4
25	35.5 ±5.6	100.4 ±42.7
50	42.7 ±4.1	93.7 ±22.9
100	71.4 ±7.6	131.2 ±17.6
200	150.5 ±11.0	320.3 ±35.1

Table 4: Two-phase commit scalability. Mean and standard deviations over 10 runs.

Beyond two phase commit

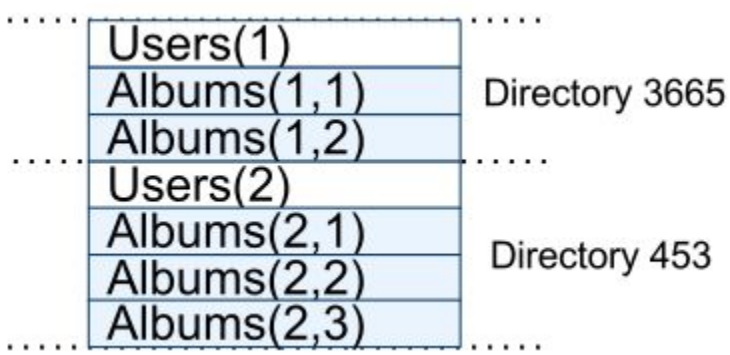
- Two phase commit takes two round trips (without considering follower communication, consider only leader to leader communication)
- There are some efforts in which people support un-abortable distributed transactions in one round trip
- There are some efforts, people add another phase(three phase commit) so that
    - If TC fails after 1st phase, participants timeout and aborts the txn without coordinator
    - If TC fails after 2nd phase(pre-commit), participants timeout and try to commit txn without coordinator. In the pre-commit phase the actual commit decision is sent to all (or some k) participants. Since they store the decision of pre-commit, if TC fails, the new coordinator uses this information to proceed with commit phase

Locks in Spanner

- Each split (either leader or follower) stores the lock data in memory, so if it restarts the lock data is lost and we're supposed to restart txn. However it's often the case that one of the follower gets elected to leader and will have all lock data.
- All the locks acquired are supposed to kept until commit wait time is done this is called two phase locking.
- Locks can lead to deadlocks, so spanner automatically kills (wounds) the younger transaction, the algorithm is known as wound wait. So in prepare phase in two phase commit, all the acquired locks are checked once again.
- Spanner try to avoid the locks and provides Read only transactions, which don't lock at all (not even ReadShared), instead they read the snapshot of the database at a particular time t.
- Each split (leader or follower) maintains at t_safe timestamp, which is the max timestamp of the data it contains (contains all data before t_safe timestamp). So while reading with timestamp if the follower split contains the data, it can serve from the it's own split without going to the leader

Storage layer optimisations

Store related data (even though in multiple tables) data together, mostly in a single disk block, so that the disk operations are optimised




```
CREATE TABLE Users {
uid INT64 NOT NULL,
email STRING }
PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
uid INT64 NOT NULL,
aid INT64 NOT NULL,
name STRING }
PRIMARY KEY (uid, aid),
INTERLEAVE IN PARENT Users ON
DELETE CASCADE;
```

Spanner storage additional features

- Should be able to store protobufs as tables, this is not available in GCP Spanner, even though there is demand for that!
- You can have replication groups so that some groups can be used for transactions, while others can be used for analytics
- Store columns in different stores, for example store some hot columns in SSD (like count, reaction count), where as comments history in Disk drive
- Spanner initially used SSTables with simple row optimised way. Over a period of time, they moved to row-column optimised format called Ressi

Spanner scale and similar systems

- Google announced that spanner served 3 billion QPS across the globe and serve several internal system like Google Ads, compared to 126 Million QPS from Amazon dynamoDB (By June 2023)
- Spanner biggest resource is Truetime which gives time in microseconds precision
  - Recently in Nov 2023, Amazon announced new time sync service that can give the time microsecond precision, but they didn't give info about uncertainty bounds
  - Some databases like CockroachDB and YugaByte DB don't have sophisticated locks, but for a distributed txns, they always commit with timestamp that's max of all commit timestamps of all servers involved + 1.
  - So they don't have external consistency but high serializability

Paxos and similar systems

- Paxos is one of the complex concept in computer science ever
- So people developed new algorithms, they don’t improve the time complexity or network but simpler to understand
- A popular such algorithm is Raft
- Assume we have 5 nodes in a system, a leader will be elected based on votes, votes will be given if the proposed leader will at least more log than participant
    - If a node can get majority votes then it can become the leader
    - However if there two or more nodes want to become leader at same instant, they might not get majority votes. So they’ll retry with some random wait time so that one leader election won’t conflict with others.

Raft protocol detail

- Raft goal is to create a consistent data store with fault tolerance, high availability. Since we need to have N replicas to provide fault tolerance. However Raft will have only writer (leader) to take requests from the client.
- Raft selects leader through election, all writes goes to leader and leader maintains log
- Writes are forwarded to follower and we make sure that, if a log number “L” has been replicated to follower, it’s guaranteed that all logs with log number < L are also replicated
- If a client makes a write request to the client, leader will replicate the log to at least majority of the replicas and return success
- Since it’s only majority of the replicas will have the data, In case of leader failure → leader election, we can’t select any replica

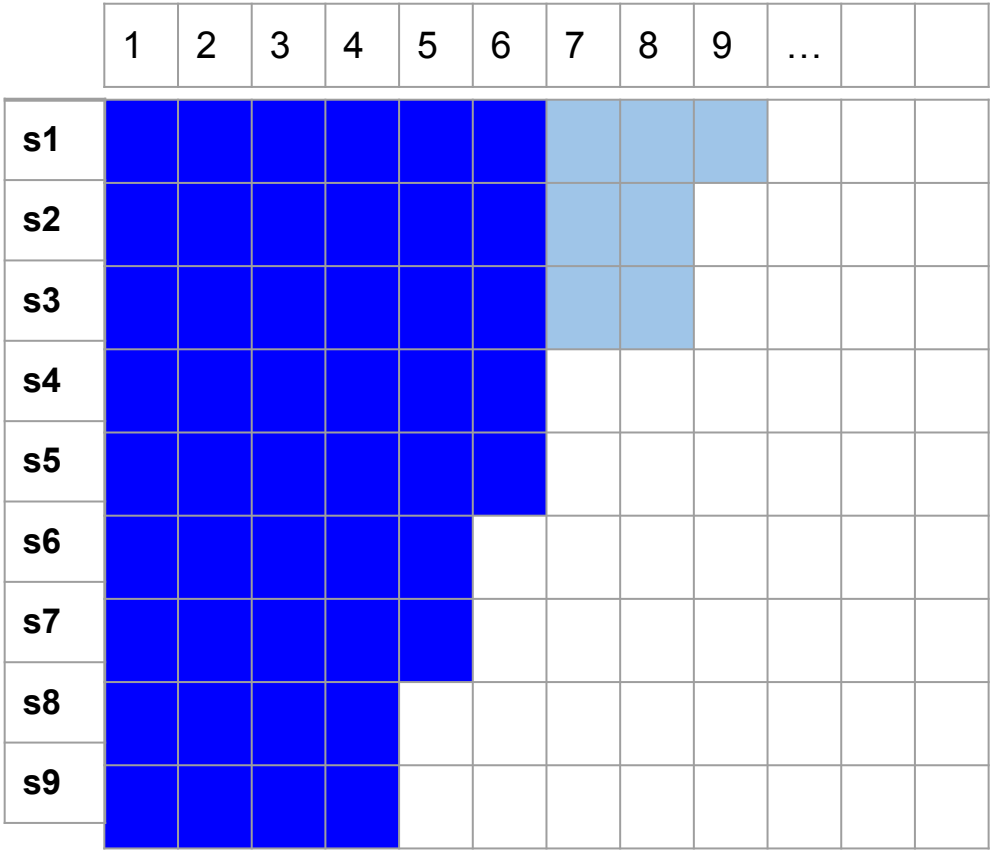
Raft leader eligibility

Assume we have 9 servers s1… s9. We have s1 as leader and written 9 log messages by client.

Client got success till log 6, since majority of them copied.

Log 7,8,9 are in progress but leader got failed, so client see failure response for log 7,8,9. But expects the data till Log 6.

however s6,s7,s8,s9 are not guaranteed to have the log till 6.



Raft leader eligibility - Have good log history

- Let a leader is crashed, once the server doesn't listen anything (no heartbeat) from server for sometime, The server tried to be election "candidate". To avoid all servers to become candidate at once, we add a random time delay before becoming a candidate
- A candidate server looks for votes from remaining servers (called followers), and broadcast vote request
- A follower will vote for candidate with more log, in the prev case,
  - S1,s2,s3,s4,s5 won't vote for any one of s6,s7,s8,s9
  - But s6,s7,s8,s9 will vote for s1,s2,s3,s4,s5
- Any server in s1,s2,s3,s4,s5 can get majority votes (including itself) and can become leader, and have the data till Log6, so the system will be consistent.

Raft leader eligibility - Have latest election

- Suppose, there is leader election going on and one candidate requested votes for all followers, it got votes, but before becoming the leader it crashed.
- Now it’s important that another server needs to be leader, after election timeout period one of the node becomes candidate, votes for it self and asks votes from others, followers would vote new candidate
- What if the old server comes up and says let me become leader, since I got votes, the follower should reject it right. So we need to reject older entries, since this is election we use “term” counter.
- So every election will have term counter, and if followers see vote requests for older term - They won’t vote candidates with older term.

Raft leader election summary

Any node just observes that there is no leader (leader lease time expired), and wait timeout happened, it'll become candidate, other nodes becomes followers

Candidate	**Followers**
Increment term counter Vote for itself Request all followers for votes	
	Check if it's not old term (if it's reject vote) Check if candidate has longer log than self Then vote for candidate
Wait for majority votes (m+1 in 2m+1) Once get the votes, become leader Broadcast to all followers	
	Followers learn the leader node id

Summary of highly available single writer systems

- A leader gets elected or selected
- Leader receives all write requests, and decides to store or not (if it’s violating unique constraint), if it’s decides to store, store it in itself and send the log request to all followers
- Once majority of followers append the write (and all prior to that), and the majority return success to the leader, the leader returns success to the client
- However, some follower logs might be deleted (we couldn’t get majority on the log entry, even though that particular follower return success)
- So leader sends commit or undo commit messages to the follower, the follower will move it’s commit pointer or delete the log correspondingly

Beyond commodity hardware

- We’ve seen all of these services are done with commodity hardware (spanner is not quite sophisticated compared to other sophisticated hardware in the industry, it used more accurate clocks)
- Assume what if CPU can access infinite memory, in practice assume a CPU can access all memory in the cluster of the nodes (natively)
- Difference b/n CPU access it’s own memory vs RPC (traditional hardware)
    - When a CPU access a memory address, it’ll send address to memory controller, and it would read the data and send the data to CPU without any other CPU knowing it
    - If an RPC is made, it’s supposed to write RPC request to kernel(context switch and copy), kernel writes to NIC transfer queue and sends the request over network to another machine, other machine’s NIC read the data and raise CPU interrupt, and CPU schedules reading from NIC, CPU reads from NIC to kernel, and kernel context switch to copy to user space and read the data, and similar thing is done to send the response back.

Can’t we do anything better than this

- Build a hardware where a CPU can read memory location of another machine without changing to kernel space, and directly writes to NIC buffer, and other machines NIC buffer directly reads from memory and send back to the origin CPU.
- This is called RDMA, and one database is created in production with this idea, (FaRMDB)
- However the efficiency gains(mostly network latencies) lost if we keep the nodes across the globe, the nodes are expected to stay closer, might be in same rack, same aisle or same floor in the datacenter.
- They achieved single read with a latency of 5-10µs (not ms), since they used RAM to store all the data.
- So overall fault tolerance of the system is lower, and it’s expensive too.