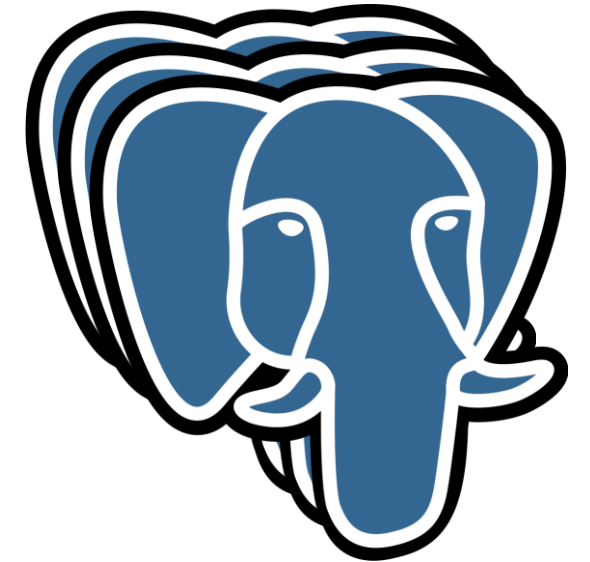


Distributed PostgreSQL



Marco Slot - marco.slot@microsoft.com

Principal software engineer on the Citus team at Microsoft

My career timeline



Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin



2009

2014

2019

My background in Distributed PostgreSQL

Developing Citus since 2014 at <https://github.com/citusdata/citus>

Citus is a PostgreSQL *extension* that adds the ability to transparently distribute or replicate tables across a cluster of PostgreSQL servers.

Paper: “Citus: Distributed PostgreSQL for Data-Intensive Applications” - SIGMOD '21

Many other Distributed PostgreSQL systems have appeared.

Distributed PostgreSQL landscape



AlloyDB (Google)



Citus (Microsoft)



Spanner (Google)



TimescaleDB



Greenplum (VMWare)



Aurora (Amazon)



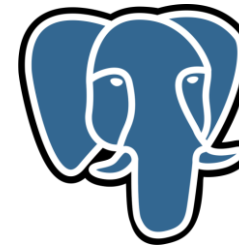
CockroachDB



Yugabyte



PolarDB (Alibaba)



PostgreSQL



TBase (Tencent)

Distributed PostgreSQL

PostgreSQL

Extremely versatile data storage and processing tool,
but limited to a single server

Distributed PostgreSQL

Extremely versatile data storage and processing tool(s)
not limited to a single server, but still under development, trade-offs to consider

Don't you need NoSQL to scale?

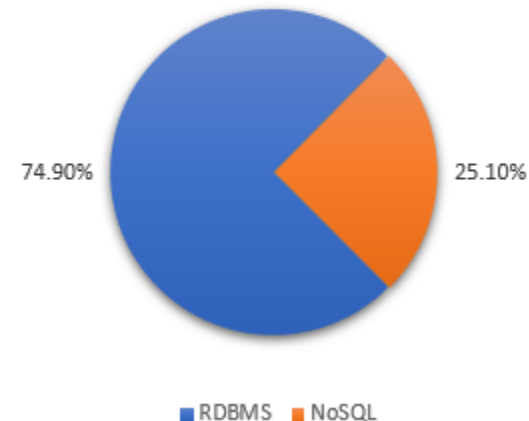
No

Winning start-up ↔ RDBMS

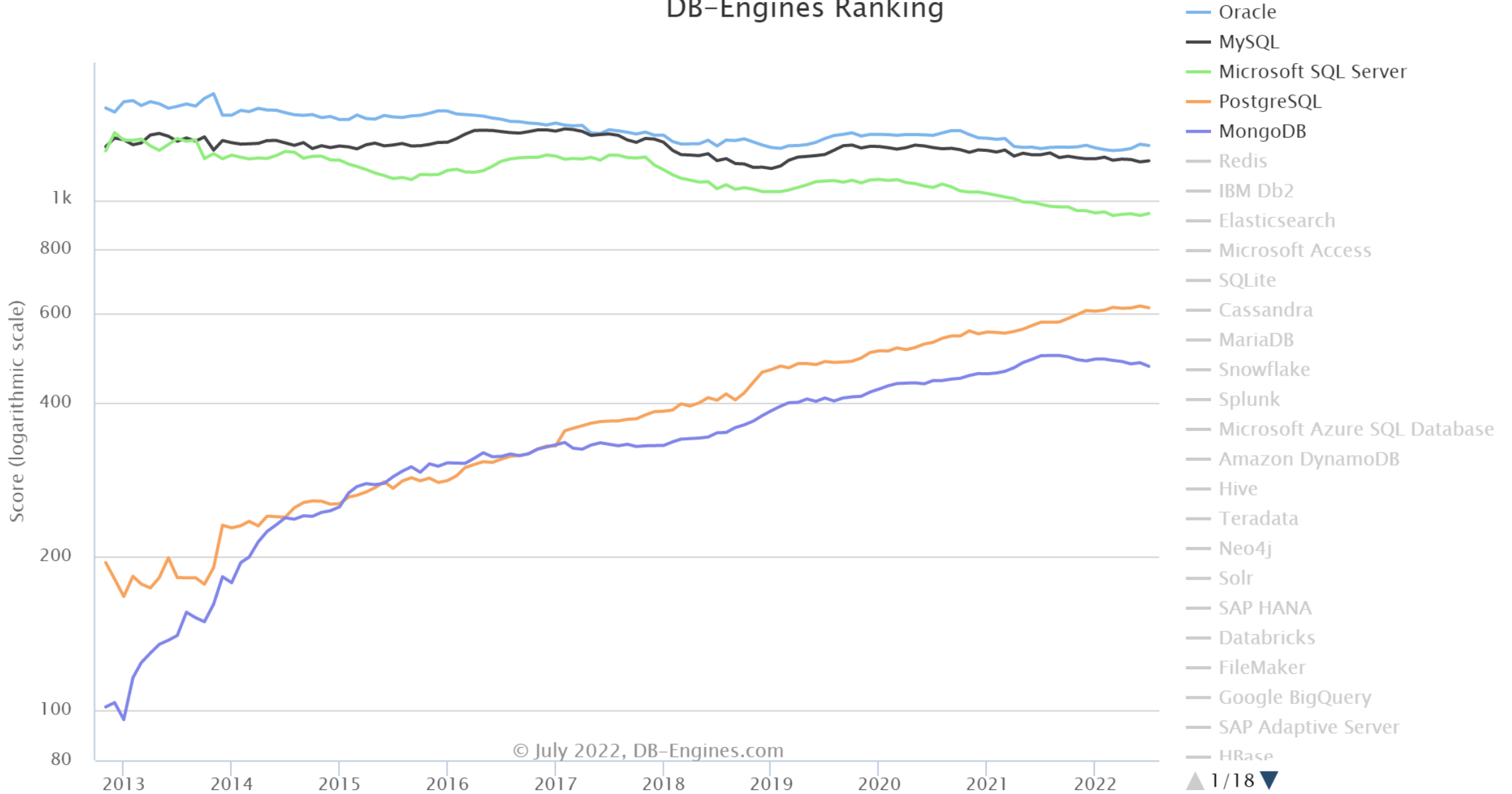
What was the main database successful tech companies used during their hypergrowth phase?

Amazon	- Oracle
Facebook	- MySQL
Gitlab	- PostgreSQL
Google	- MySQL
Instagram	- PostgreSQL
Netflix	- Cassandra
Reddit	- PostgreSQL
Salesforce	- Oracle
Skype	- PostgreSQL
Stack Overflow	- SQL Server

Usage of database by type



DB-Engines Ranking



Today's lecture

Part I: PostgreSQL

Build an intuition for the internals of PostgreSQL / a state-of-the-art RDBMS

Understand what problems need to be solved when storing & manipulating data

Know when PostgreSQL is the right tool for the job

Part II: Distributed PostgreSQL

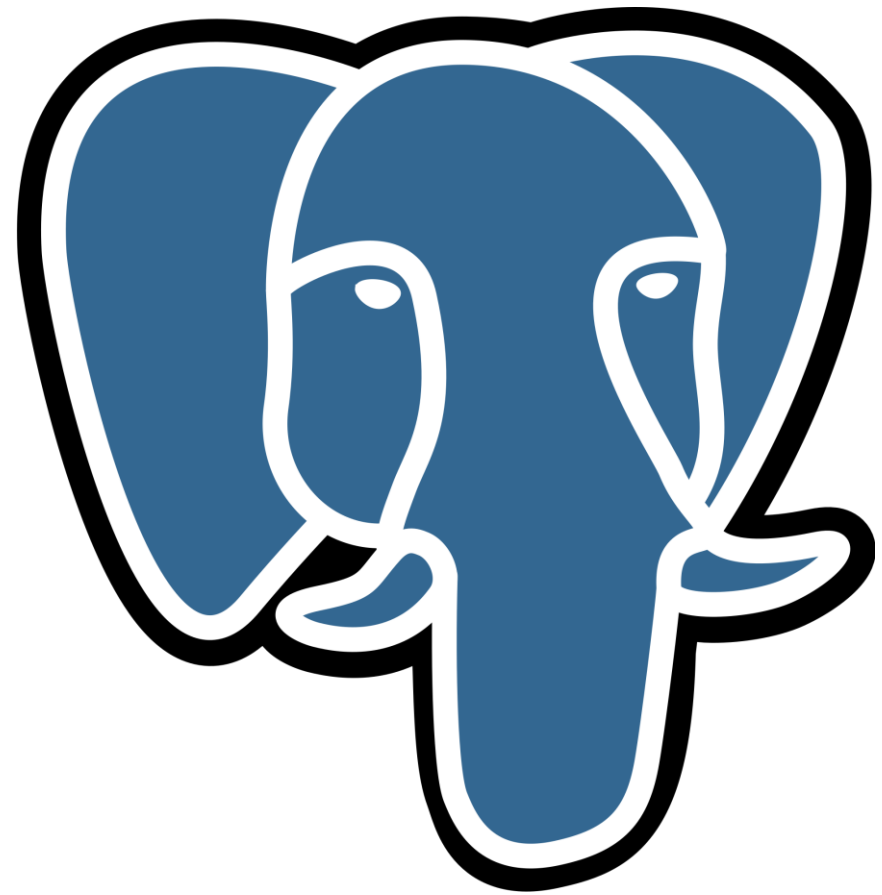
Introduction to Distributed database concepts

Look at how different Distributed PostgreSQL vendors apply these concepts

Know how to navigate the Distributed PostgreSQL landscape

PostgreSQL

Just use it



PostgreSQL

Relational Database Management System (RDBMS)

- ACID transactions, SQL, Schema management, Constraints, Procedures, Indexes, ...

Started as “postgres” by Michael Stonebraker in 1985 at Berkeley

- PostgreSQL since 1996 when SQL support was added

Community open-source project – no company behind it

- All development on pgsql-hacker@postgresql.org (patches sent as attachments...)

Extensions can add new database features:

- Types, Functions, Access methods, Foreign data wrappers, Custom query planning & execution

THE DESIGN OF POSTGRES

Michael Stonebraker and Lawrence A. Rowe

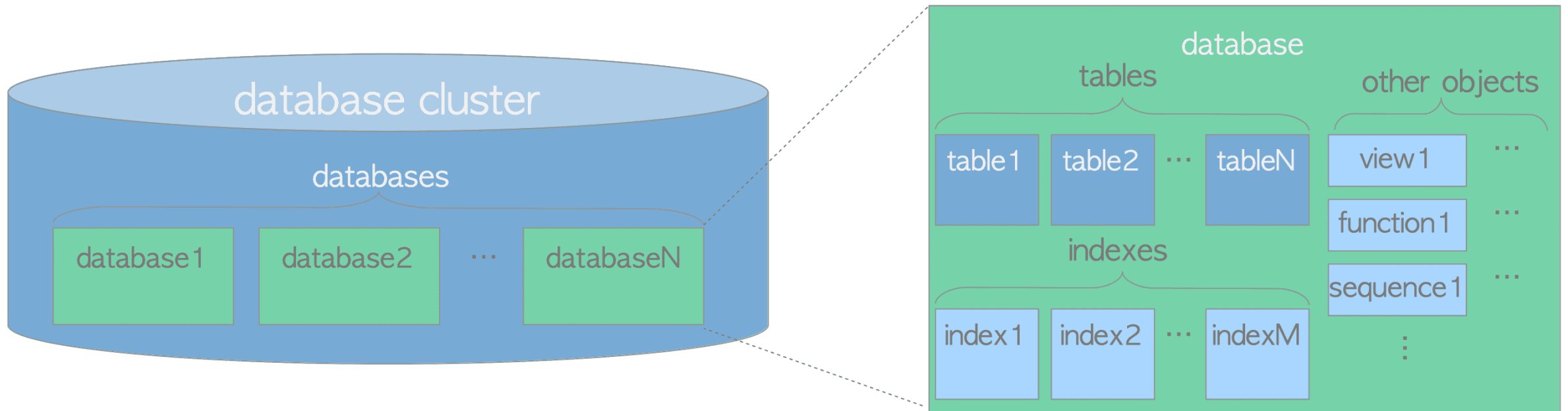
*Department of Electrical Engineering
and Computer Sciences
University of California
Berkeley, CA 94720*

Abstract

This paper presents the preliminary design of a new database management system, called POSTGRES, that is the successor to the INGRES relational database system. The main design goals of the new system are to:

- 1) provide better support for complex objects,
- 2) provide user extendibility for data types, operators and access methods,
- 3) provide facilities for active databases (i.e., alerters and triggers) and inferencing including forward- and backward-chaining,

PostgreSQL database model

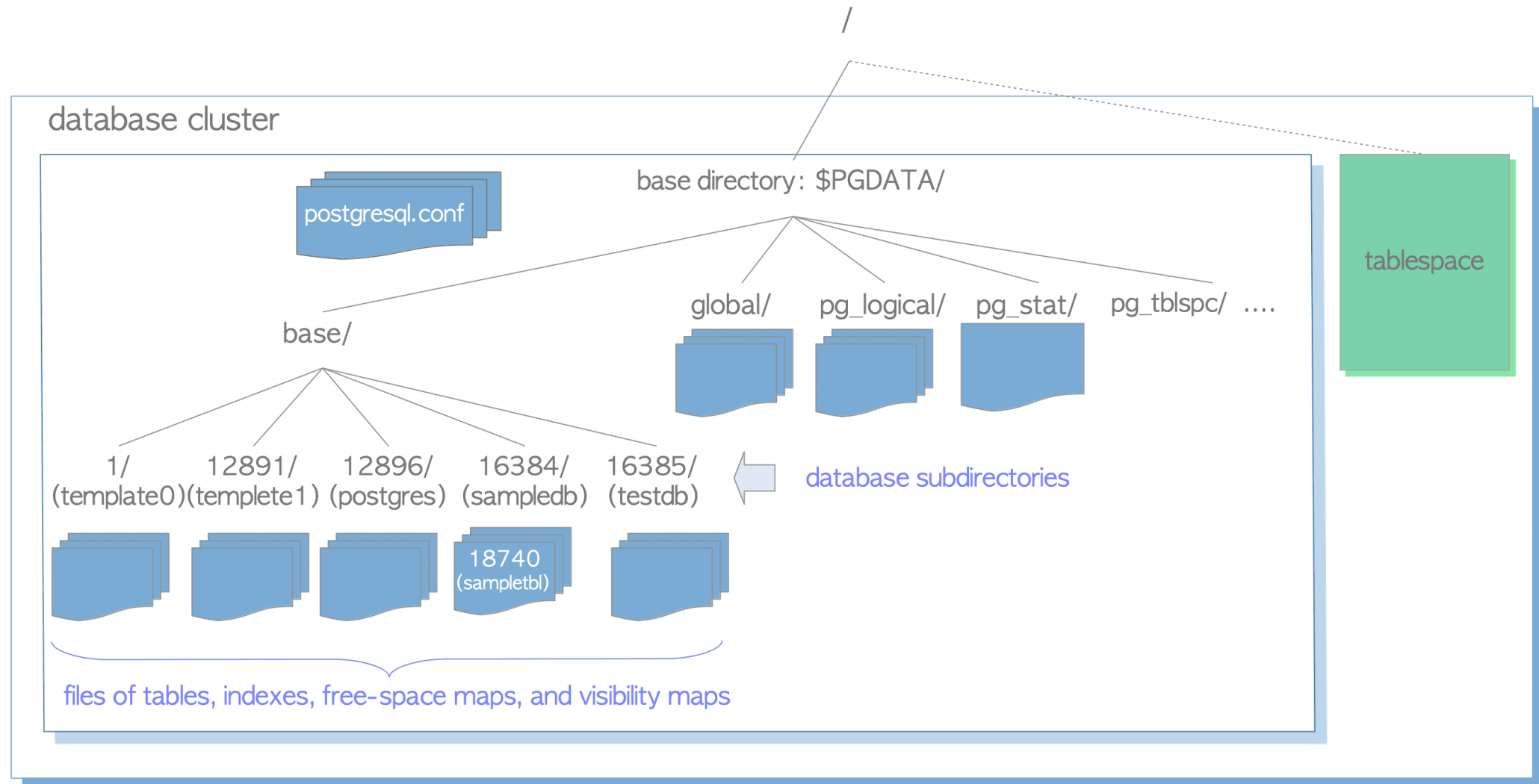


Getting started with PostgreSQL

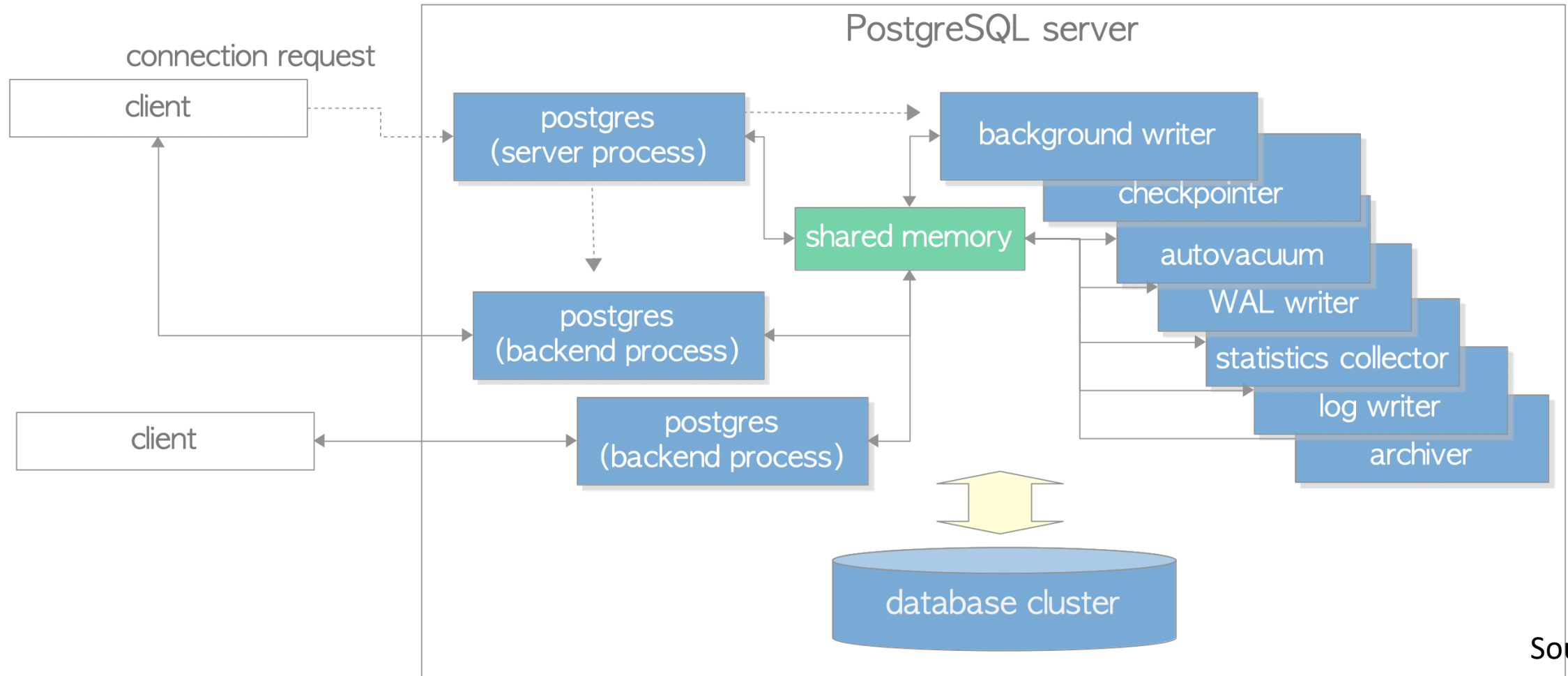
```
# Create the database directory  
initdb -D $PGDATA
```

```
# Start the postgres server  
pg_ctl -D $PGDATA start
```

PostgreSQL database directory



PostgreSQL server architecture



THE IMPLEMENTATION OF POSTGRES

*Michael Stonebraker, Lawrence A. Rowe and Michael Hirohama
EECS Department
University of California, Berkeley*

A last aspect of our design concerns the operating system process structure. Currently, POSTGRES runs as one process for each active user. This was done as an expedient to get a system operational as quickly as possible. We plan on converting POSTGRES to use lightweight processes available in the operating systems we are using. These include PRESTO for the Sequent Symmetry and threads in Version 4 of Sun/OS.

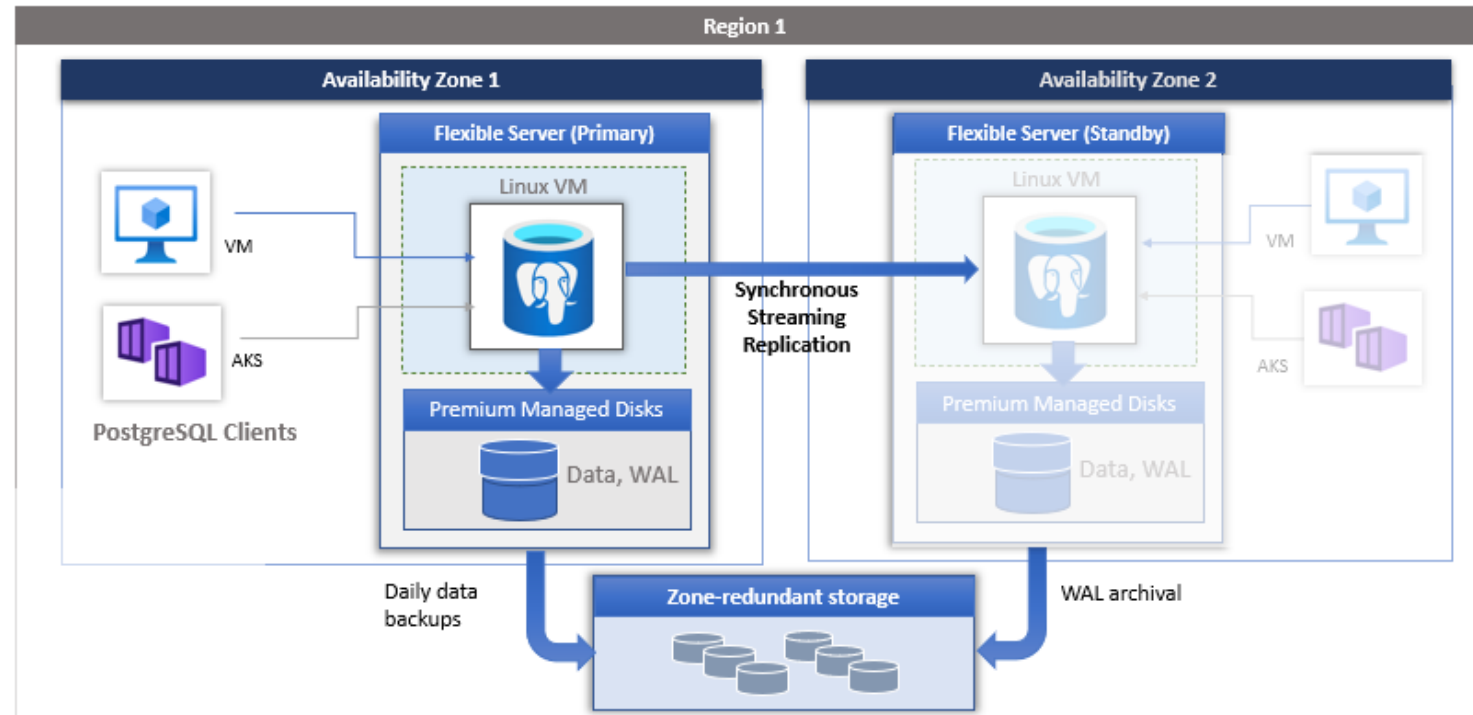
Productions set ups

Use a hot standby that can take over in case of failure.

Network-attached disks
simplify operations.

Backups = Disk snapshots +
WAL archival into cheap storage

AWS, Azure, GCP, and others can
run it for you.



PostgreSQL Getting Started

```
# Connect to the database
```

```
psql
```

```
-- Create a table
```

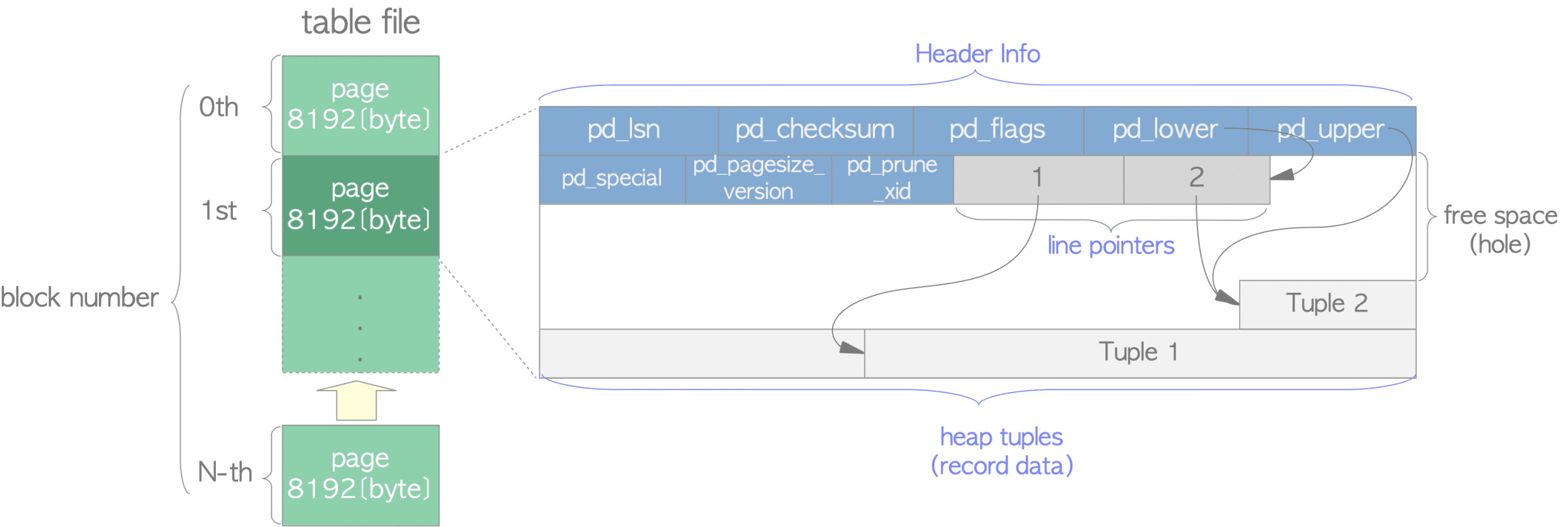
```
create table items (key text, value text);
```

```
-- Insert 2 records
```

```
insert into items values ('hello', 'world');
```

```
insert into items values ('ciao', 'cesena');
```

PostgreSQL table layout (heap)



On-disk representation
Individual pages are also cached in shared memory

Source:
interdb.jp

PostgreSQL Query Performance

```
-- Query on a table with 2 rows  
select * from items where key = 'hello';
```

key	value
hello	world

(1 row)

Time: 1.370 ms

PostgreSQL Query Performance

```
-- Add 10 million rows  
insert into items select 'item-' || s, 'value-' || s  
from generate_series(1,10000000) s;
```

```
-- Query on a table with 10,000,002 rows  
select * from items where key = 'hello';
```

key	value
hello	world

(1 row)

Time: 2379.022 ms (00:02.379)

PostgreSQL Indexes

The actual right answer to coding interviews, which the interviewer won't accept

PostgreSQL Indexes

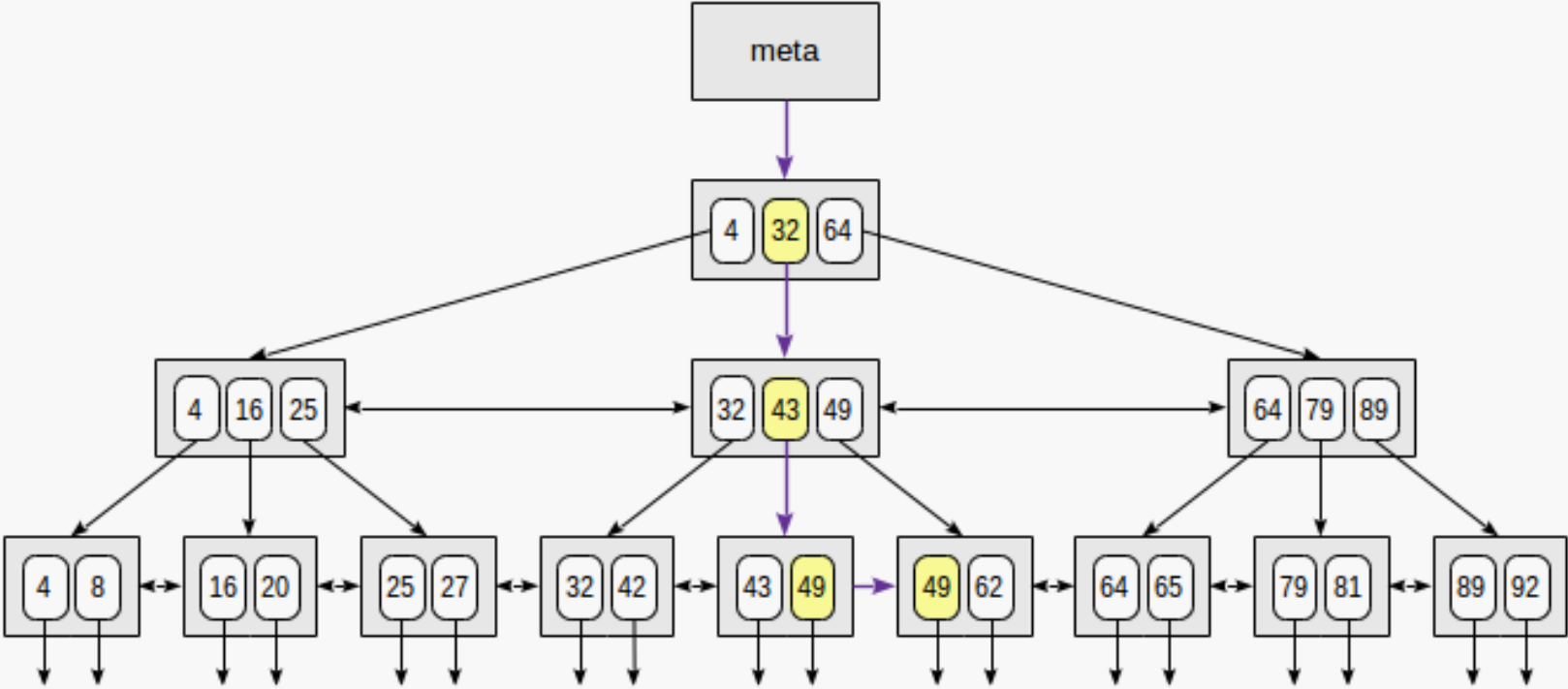
```
-- Create a btree index  
CREATE INDEX key_idx ON items (key);  
  
-- Query on a table with 10,000,002 rows  
select * from items where key = 'hello';
```

key	value
hello	world

(1 row)

Time: 0.937 ms

Btree index

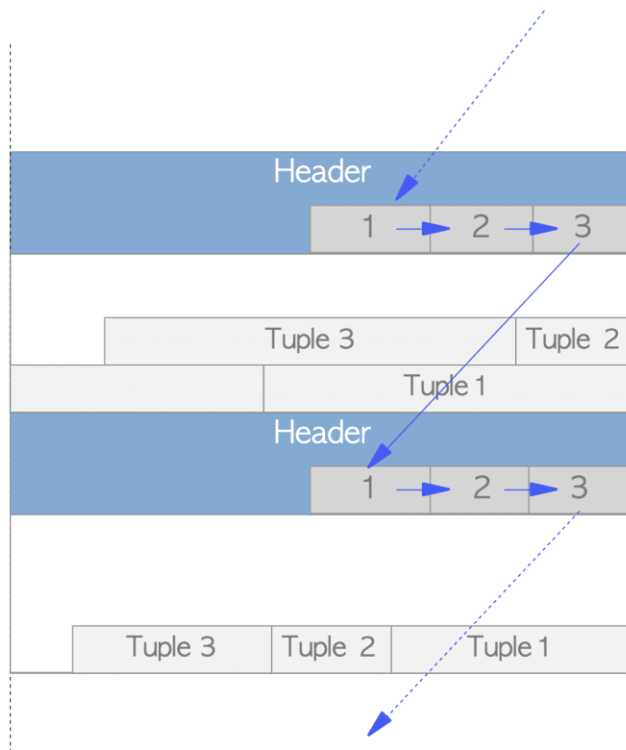


Tuple identifiers (page index, tuple index) pointing at heap

PostgreSQL Index Scan

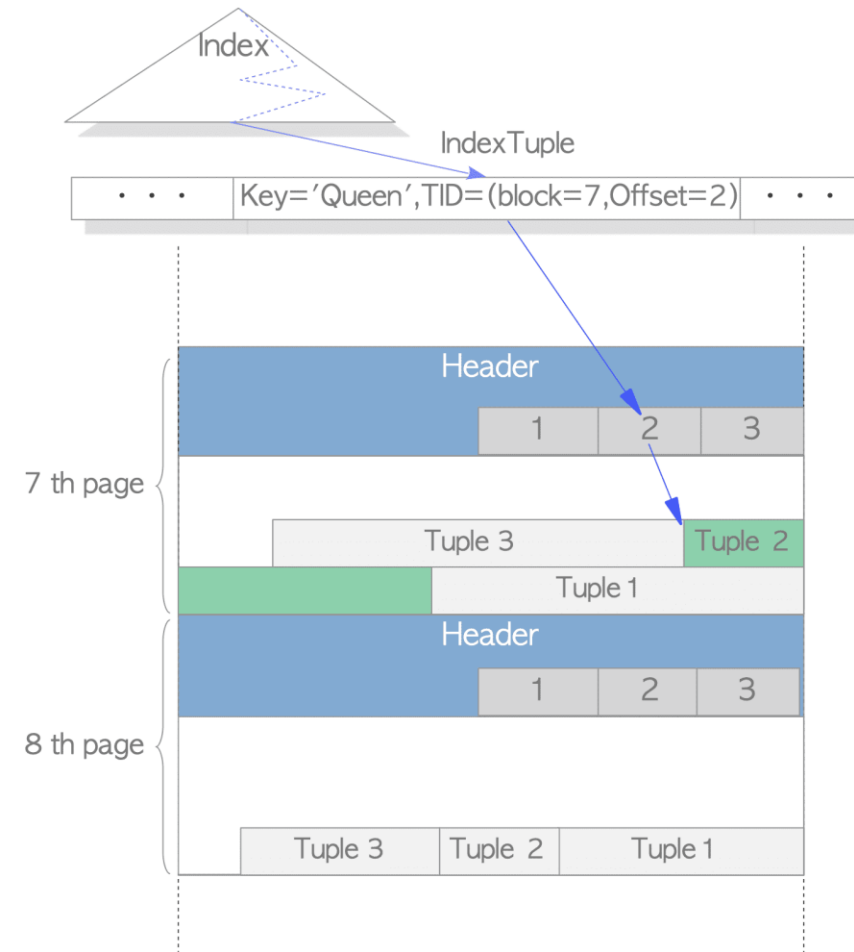
(a) Sequential Scan

SELECT * FROM tbl;



(b) Index Scan

SELECT * FROM tbl WHERE col = 'Queen';



PostgreSQL Index types

PostgreSQL index types:

- Btree - Good for key-value and small range lookups
- Hash - Good for key-value lookups, slow-changing data
- BRIN - Good for time range queries on ordered time series data
- GiST - Good for geospatial queries
- GIN - Good for document & text search
- ...

Orthogonal: Partial indexes, expression indexes

PostgreSQL Partitioning

Indexes give no guarantees about data ordering on disk.

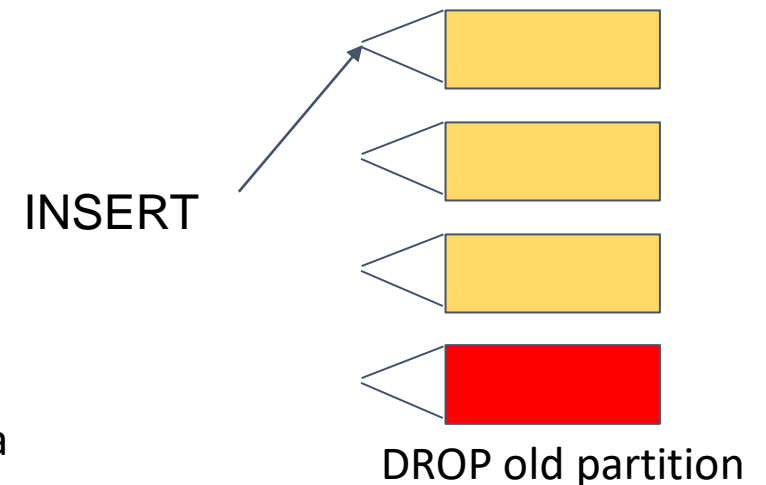
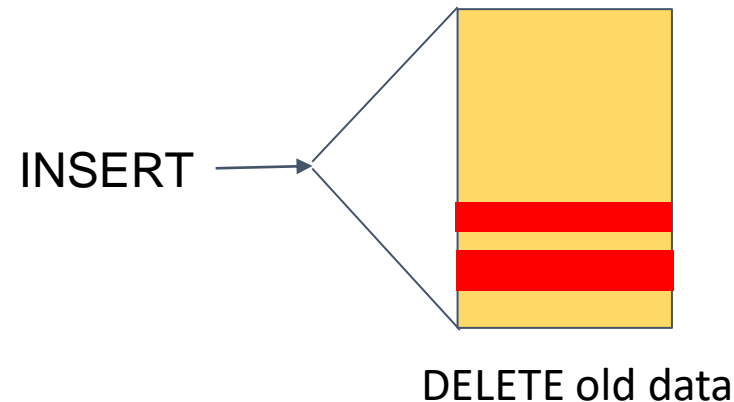
Partitioning can be used to group data by (time) range on disk.

```
CREATE TABLE events (... , event_time timestamptz default now())  
PARTITION BY RANGE (event_time);
```

```
CREATE TABLE events_1 PARTITION OF events  
FOR VALUES FROM ('2022-07-04') TO ('2022-07-11');
```

Automate using:

- TimescaleDB
- Citus
- pg_partman



ACID Transactions

Because storing stuff on computers is haaaard

ACID Transactions

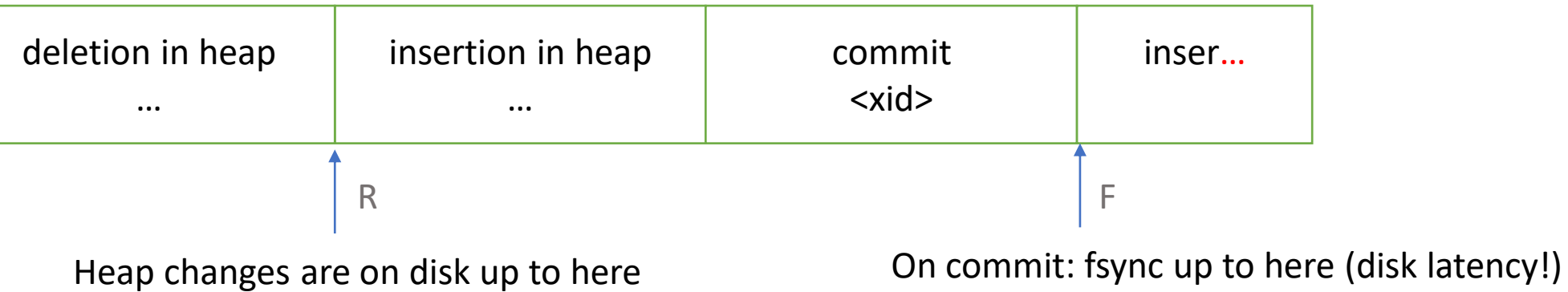
A transaction is a set of read/write operations that are bundled together and have the following properties:

- Atomicity - Either all changes are performed, or none
- Consistency - Constraints are preserved before & after
- Isolation - Intermediate state is invisible to other transactions
- Durability - Changes persist, even in case of a system failure

Multiple processes reading and writing concurrently.

Write ahead log

Write ahead log is the authoritative record of all changes to the database.



On restart (e.g. after crash): Replay all changes from R to F

Multi-version concurrency control (MVCC)

Every PostgreSQL transaction has a 32-bit transaction ID (xid)

Every tuple stores the following information:

- xmin – transaction ID which created the tuple
- xmax – transaction ID which updated/deleted the tuple
- ctid – location of the latest version of a tuple (for updates)

```
postgres=# select xmin, xmax, ctid, * from items;
```

xmin	xmax	ctid	key	value
4215173	0	(0,1)	hello	world

(1 row)

Snapshots

Every PostgreSQL transaction has a snapshot consisting of:

- xmin – no more transactions with a lower transaction ID remain
- xmax – we cannot see any higher xids because they started later
- xip_list – we cannot see these either because they are in progress

From the snapshot, we can determine whether another transaction ID is “in progress”.

Reads skip tuples written by transactions which are “in progress”.

Snapshot implementation in Postgres

```
bool XidInProgressInMVCCSnapshot(TransactionId xid, Snapshot snapshot)
{
    /* Any xid < xmin is not in-progress */
    if (TransactionIdPrecedes(xid, snapshot->xmin))
        return false;
    /* Any xid >= xmax is in-progress */
    if (TransactionIdFollowsOrEquals(xid, snapshot->xmax))
        return true;

    ...
    for (i = 0; i < snapshot->xcnt; i++)
    {
        if (TransactionIdEquals(xid, snapshot->xip[i]))
            return true;
    }

    return false;
}
```

Tuple visibility

Can determine whether tuple is visible from xmin, xmax:

```
HeapTupleSatisfiesMVCC(tuple, snapshot)
    !XidInProgress(tuple.xmin, snapshot) && IsCommitted(tuple.xmin) &&
    (tuple.xmax == 0 ||
     XidInProgress(tuple.xmax, snapshot) || !IsCommitted(tuple.xmax))
```

Separate global data structure (clog) keeps track of which transactions are committed.

MVCC Benefits

The MVCC semantics in PostgreSQL:

- Reads see everything that was committed before (read-your-writes)
- Reads and writes do not block each other
- Database appears as if no changes are happening (isolation)

Default isolation level (read committed) has some anomalies...

Update concurrency

Two concurrent updates on a counter that starts at 0:

```
WITH v AS (  
    SELECT value FROM counters  
    WHERE id = 19376  
)  
UPDATE counters  
SET value = v.value + 1  
FROM v;
```

```
WITH v AS (  
    SELECT value FROM counters  
    WHERE id = 19376  
)  
UPDATE counters  
SET value = v. value + 1  
FROM v;
```

If neither sees the other in its snapshot, what's the result?

Update concurrency

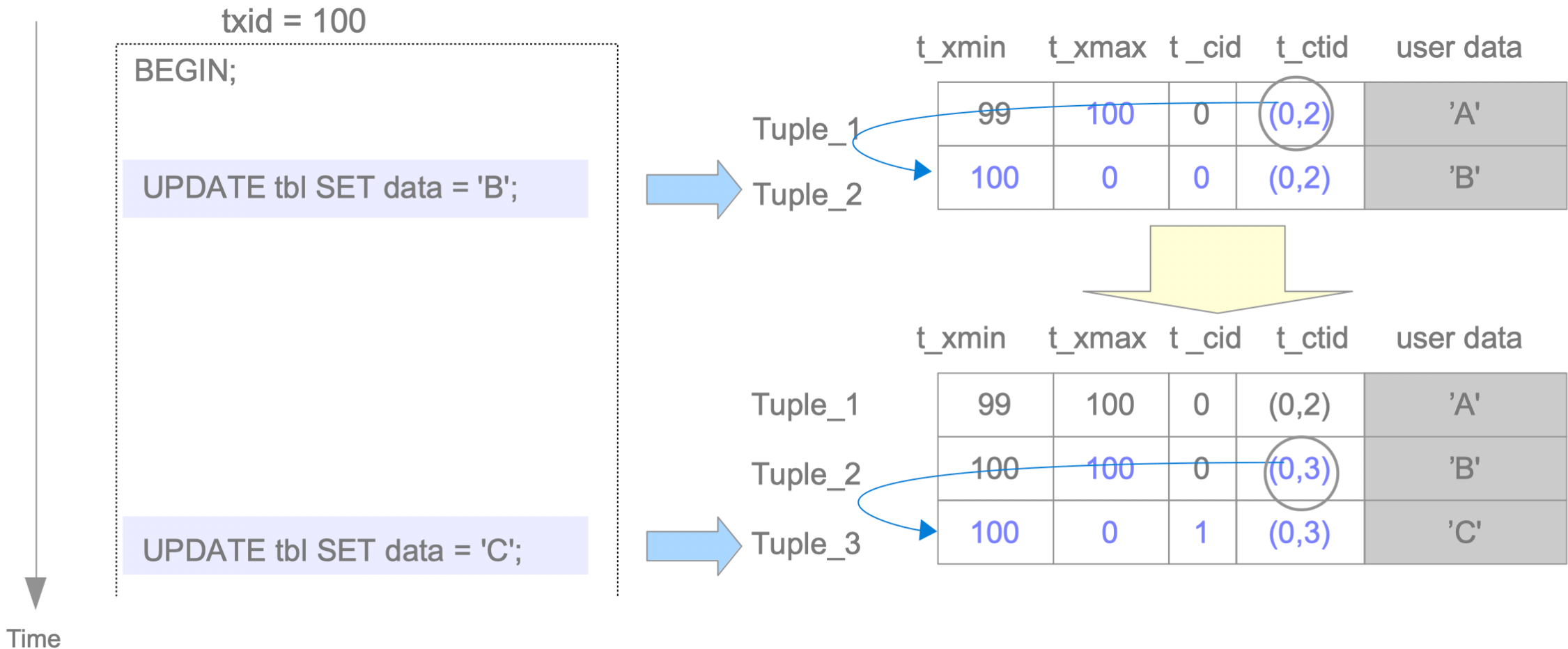
Two concurrent updates on a counter that starts at 0:

```
UPDATE counters  
SET value = value + 1  
WHERE id = 19376;
```

```
UPDATE counters  
SET value = value + 1  
WHERE id = 19376;
```

If neither sees the other in its snapshot, what's the result?

Update concurrency



Update concurrency

Two concurrent updates are serialized by row-level locks:

UPDATE counters

SET value = value + 1

WHERE id = 19376;

1. obtain row-level lock (write xmax)
2. read current tuple, write new tuple
3. update ctid of old tuple
4. commit
5. release row-level lock

UPDATE counters

SET value = value + 1

WHERE id = 19376;

1. wait for row-level lock
2. obtain row-level lock (write xmax)
3. follow update chain
4. obtain row-level lock
5. read current tuple, write new tuple
6. update ctid of old tuple
7. commit

Explicit locking

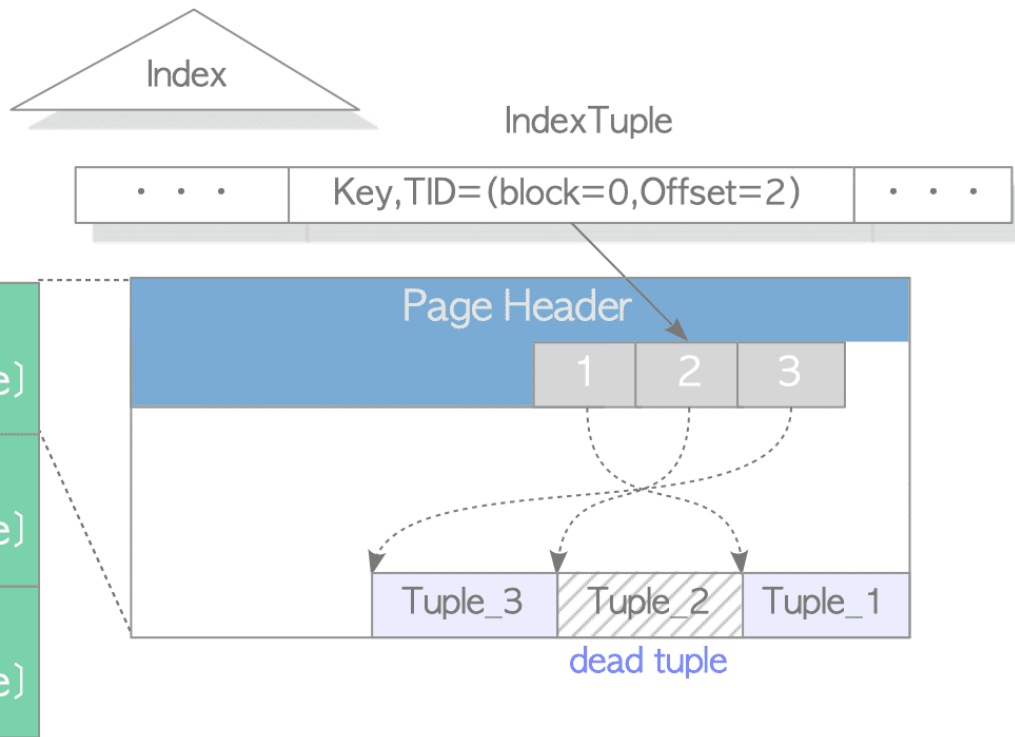
Can obtain row-level locks explicitly with SELECT .. FOR UPDATE:

```
WITH v AS (  
    SELECT value FROM counters  
    WHERE id = 19376  
    FOR UPDATE  
)  
UPDATE counters  
SET value = v.value + 1  
FROM v;
```

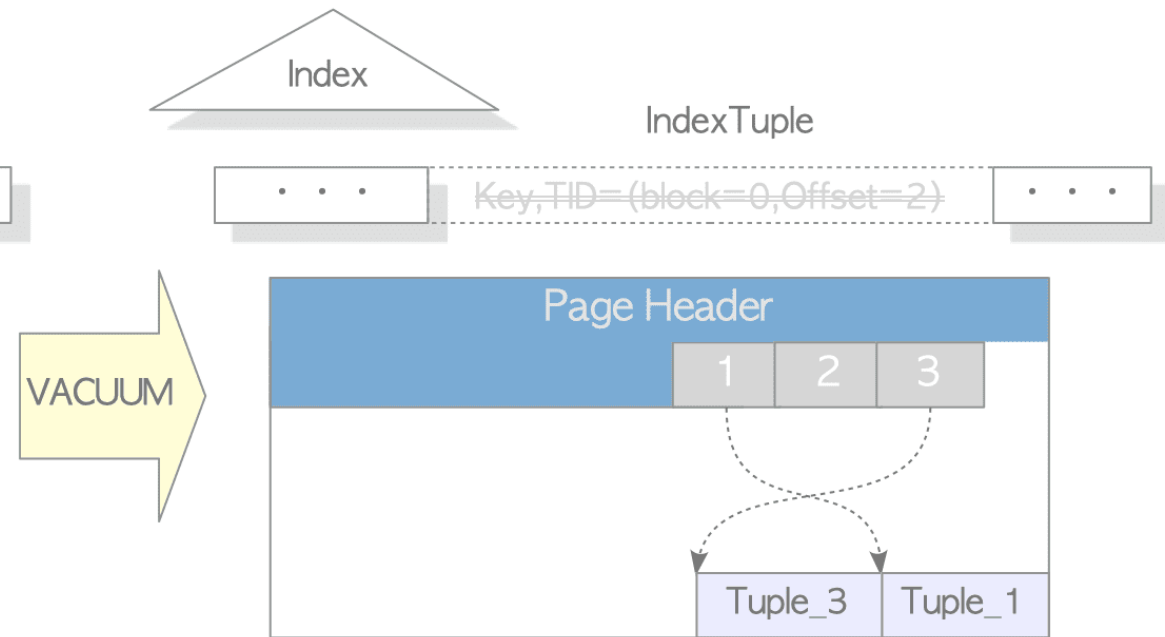
```
WITH v AS (  
    SELECT value FROM counters  
    WHERE id = 19376  
    FOR UPDATE  
)  
UPDATE counters  
SET value = v. value + 1  
FROM v;
```

What about dead tuples?

(1) Before VACUUM



(2) After VACUUM



Summary: ACID Transactions

Overall, transactions in PostgreSQL:

- Keep your data safe
- Keep your query results clean
- Work well concurrently (few locks, except in update/delete on the same row)

Read committed has anomalies

- No well-defined order between transactions
- Simple updates usually do a nice thing
- Complex updates might not, but can be corrected with FOR UPDATE

Update/delete have hidden cost of vacuuming.

SQL

Declarative data retrieval that somehow still requires a lot of tuning

```
SELECT
    customer_id,
    sum(order_value),
    row_number() OVER ()
FROM
    orders
WHERE
    order_type = 4
GROUP BY
    customer_id
ORDER BY
    2 DESC
LIMIT
    10;
```

(find the bug!)

JULIA EVANS
@bork

SQL queries run
in this order

```
SELECT
  customer_id,
  sum(order_value),
  row_number() OVER ()
FROM
  orders
WHERE
  order_type = 4
GROUP BY
  customer_id
ORDER BY
  2 DESC
LIMIT
  10;
```

(find the bug!)

FROM + JOIN



WHERE



GROUP BY



HAVING



SELECT (window functions
happen here!)



ORDER BY



LIMIT

JULIA EVANS
@bork

SQL queries run
in this order

```
SELECT
  customer_id,
  sum(order_value),
  row_number() OVER (
    ORDER BY sum(order_value) DESC
  )
FROM
  orders
WHERE
  order_type = 4
GROUP BY
  customer_id
ORDER BY
  2 DESC
LIMIT
  10;                (fixed the bug!)
```

FROM + JOIN



WHERE



GROUP BY



HAVING



SELECT (window functions
happen here!)



ORDER BY



LIMIT

JULIA EVANS
@bork

SQL queries run
in this order

```
SELECT
  customer_id,
  sum,
  row_number() OVER ()
FROM (
  SELECT
    customer_id,
    sum(order_value)
  FROM
    orders
  WHERE
    order_type = 4
  GROUP BY
    customer_id
  ORDER BY 2 DESC LIMIT 10
) a;
```

(fixed the bug!)

FROM + JOIN



WHERE



GROUP BY



HAVING



SELECT (window functions
happen here!)

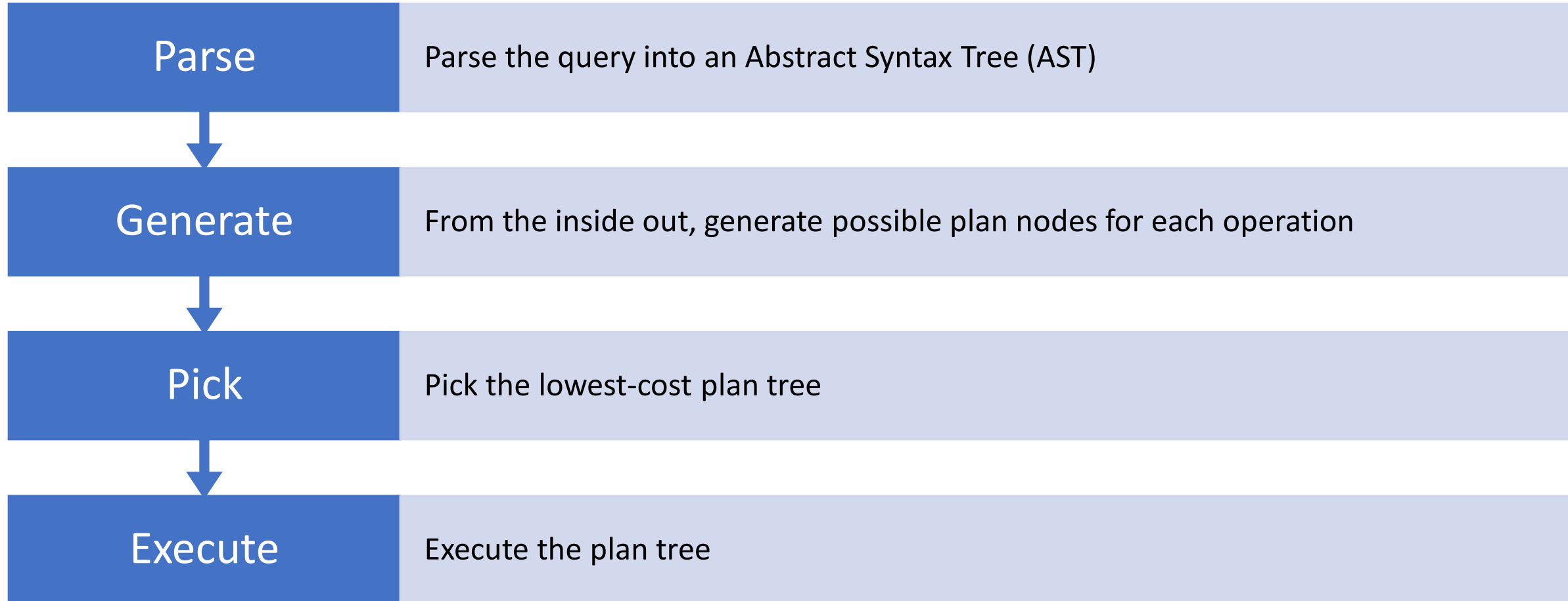


ORDER BY



LIMIT

PostgreSQL Query planner



Plan generation

FROM + JOIN



WHERE



GROUP BY



HAVING



SELECT (window functions happen here !)



ORDER BY



LIMIT

FROM+WHERE:

- sequential scan
- index scan
- bitmap index scan

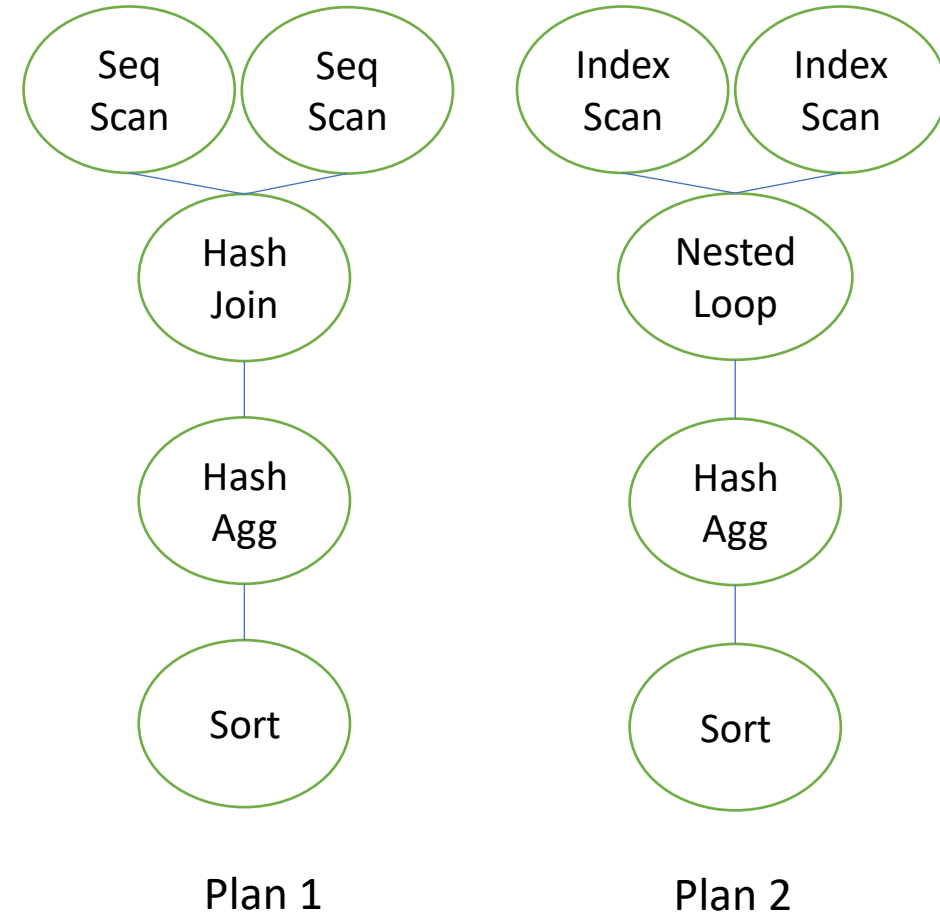
JOIN:

- nested loop
- hash join
- merge join

GROUP BY+HAVING:

- hash aggregate
- group aggregate

Window functions, ORDER BY
sort



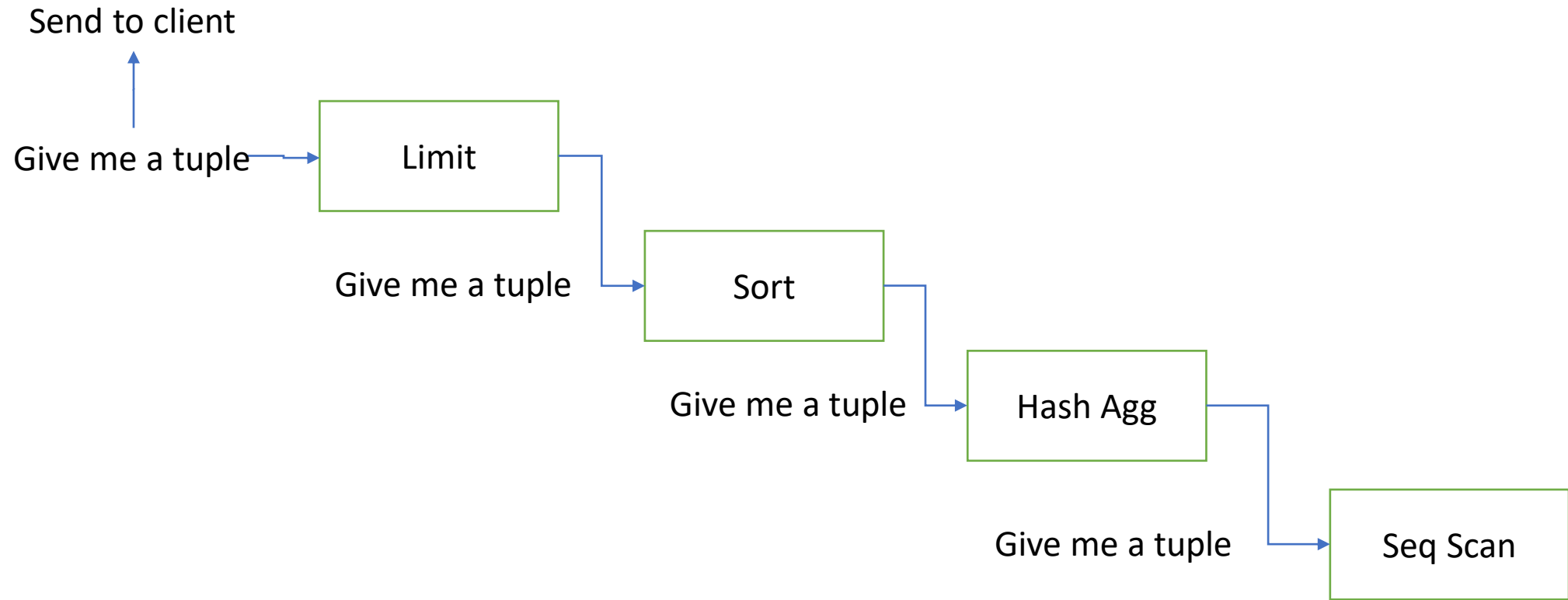
Explain query plans

```
EXPLAIN SELECT customer_id, sum(order_value) FROM orders WHERE order_type = 4 GROUP BY customer_id ORDER BY 2 DESC LIMIT 10;
```

QUERY PLAN
Limit (cost=66227.56..66227.59 rows=10 width=12)
-> Sort (cost=66227.56..68085.11 rows=743019 width=12)
Sort Key: (sum(order_value)) DESC
-> HashAggregate (cost=42741.00..50171.19 rows=743019 width=12)
Group Key: customer_id
-> Seq Scan on orders (cost=0.00..37739.00 rows=1000400 width=12)
Filter: (order_type = 4)

(7 rows)

Row-based execution



Summary: SQL

Overall, SQL in PostgreSQL:

- Uses cost-based optimization to pick a good query plan
- Row-based execution model is simple and predictable
- Supports almost all features of SQL:2016 standard

Some things to know:

- When the planner gets it wrong, it's not so easy to change
- Execution is mostly single-threaded
- Row-based executor has high overhead for analytics

Extensibility

PostgreSQL's hidden superpowers

What is an extension?

Extensions consist of:

1. SQL objects (tables, functions, types, ...)
2. Shared library

citus.sql

```
CREATE TABLE pg_dist_node (...);
CREATE TABLE pg_dist_partition (...);

CREATE FUNCTION citus_add_node(...)
RETURNS void LANGUAGE c
AS '$libdir/citus',
$function$citus_add_node$function$;

CREATE FUNCTION create_distributed_table(...)
RETURNS void LANGUAGE c
AS '$libdir/citus',
$function$create_distributed_table$function$;
```

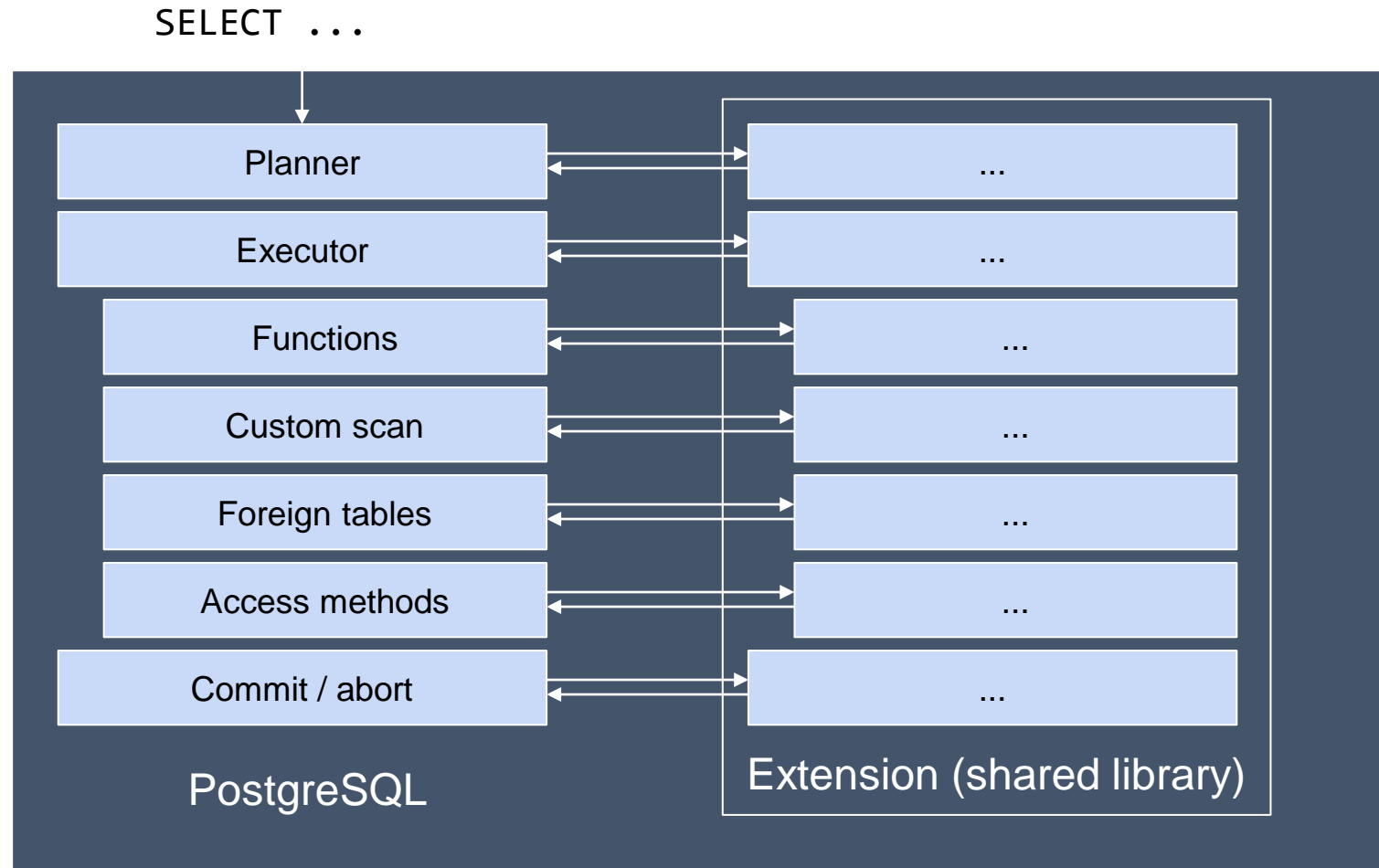
citus.c

```
#include "postgres.h"

Datum citus_add_node(...)
{
    ...
}

Datum create_distributed_table(...)
{
    ...
}
```

Almost everything is extensible



Extension hook: Planner hook

Extensions can change the way PostgreSQL plans queries

postgres.c

```
planner_hook_type planner_hook = NULL;

PlannedStmt *
planner(Query *parse, ...)
{
    PlannedStmt *result;

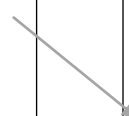
    if (planner_hook)
        result = (*planner_hook) (parse, ...);
    else
        result = standard_planner(parse, ...);
    return result;
}
```

citus.c

```
#include "postgres.h"

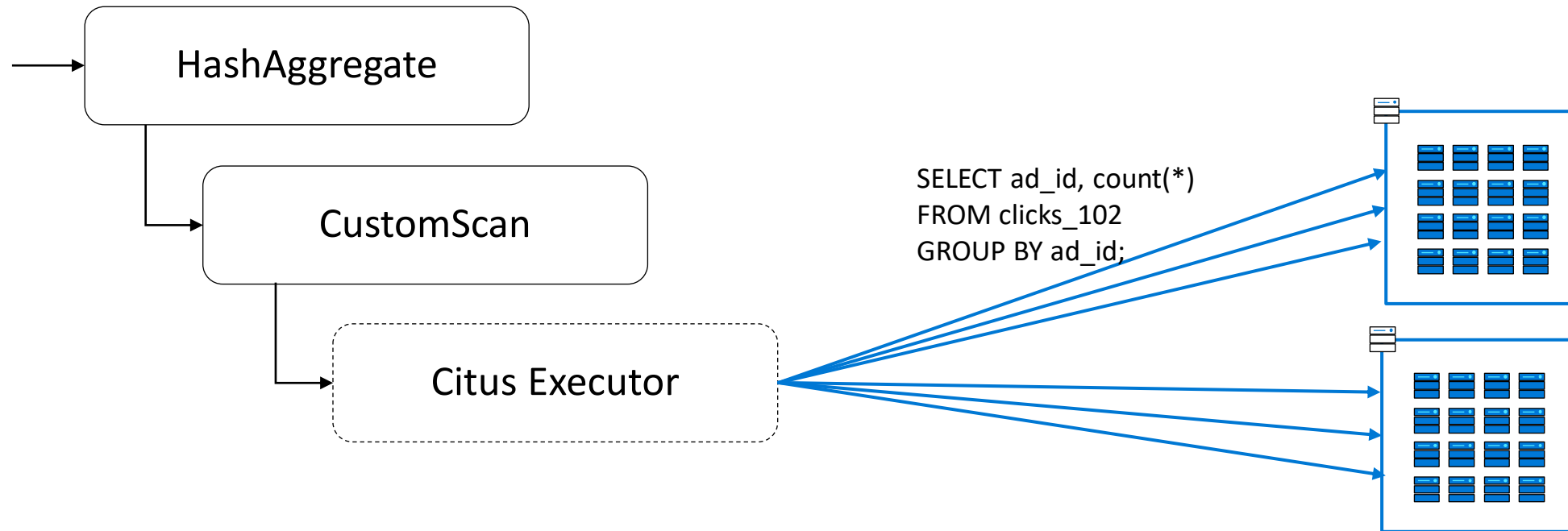
void _PG_init(void)
{
    ...
    planner_hook = distributed_planner;
    ...
}

PlannedStmt *
distributed_planner(Query *parse, ...)
{
    ...
}
```



Extension hook: CustomScan

Extensions can inject custom scan nodes into the query plan.



Extension hook: Background workers

Extensions can start background processes that operate on the database.

Example: pg_cron extension

```
SELECT cron.schedule(  
    '30 3 * * 6',  
    $$DELETE FROM events WHERE event_time < now() - interval '1 week'$$);
```

```
SELECT cron.schedule(  
    '* * * * *',  
    $$SELECT process_incoming_data()$$);
```

Common extensions

Babelfish (extension + fork)

Citus

dblink (built-in)

hll

hstore (built-in)

oracle_fdw

pg_cron

pg_partman

pg_stat_statements (built-in)

pgAudit

pgcrypto (built-in)

PL/Perl (built-in)

PL/Python (built-in)

PL/V8

PostGIS

MobilityDB

mysql_fdw

TimescaleDB

tdigest

topn

uuid-oss

wal2json

When to use PostgreSQL?

Because data storage is hard and you're probably doing it wrong

When to use PostgreSQL

System of record (OLTP) / Interactive applications

- Low latency, high throughput, good availability, transactional correctness, SQL, ...

Coordination between systems

- Transactions and locking primitives help you do the right things in your applications

Analytical applications with pre-aggregated data

- SQL, Indexes, Partitioning, Extensions, Custom Types, Data transformations, ...

Extensions make can PostgreSQL the best tool for many data types and applications:

- Time series, spatial, spatiotemporal, ...

When not to use PostgreSQL (so far)

For machine learning

- Machine learning in PostgreSQL exists, but is still in its infancy

For analytical queries over a large amount of data

- Row-based executor, limited parallelism & compression compared to data warehouse

As a low latency cache (<0.2ms)

- Btree+heap model adds relatively high overhead compared to Redis

When your data or workload does not fit on a single server...

PostgreSQL scalability challenges

Typical server limit in modern clouds:

- 64 virtual cores, 512GiB memory, 32TiB storage
- 500k reads/sec, 50k writes/sec, 5M rows/sec scans

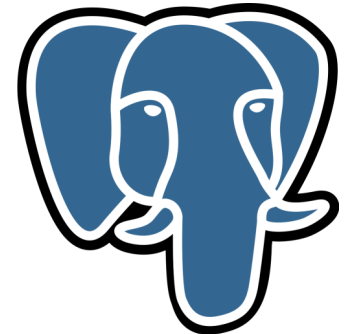
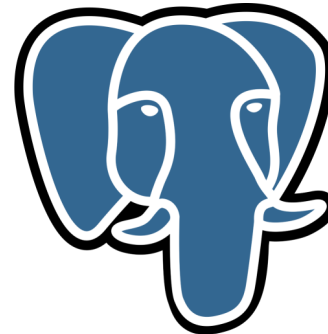
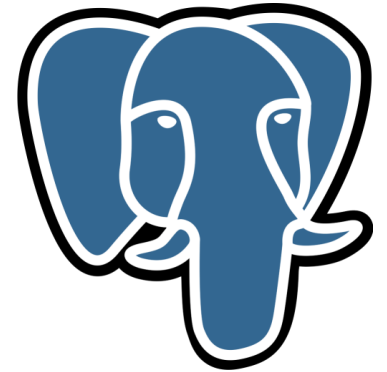
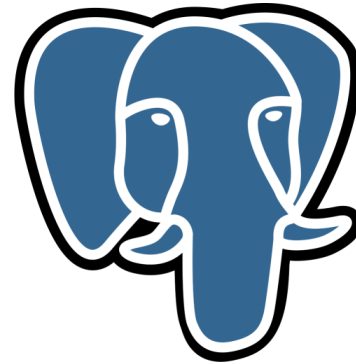
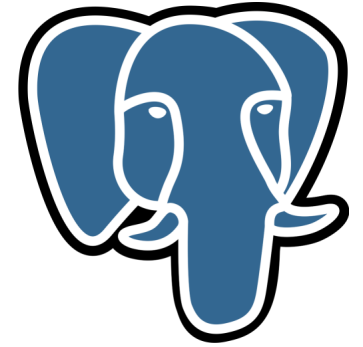
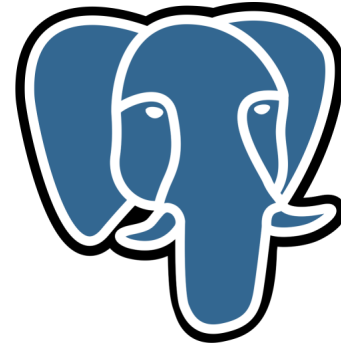
At scale, several additional pain points:

- Many operations are single-threaded
- Working set may no longer fit in memory
- Small number of operations that are $O(N)$ can dominate the workload
- Table bloat (dead tuples) can get high

Importance of availability & performance often grows with scale of application

Distributed PostgreSQL

Do all this stuff at scale



A distributed database does two things

Distribution - Place partitions of data on different machines

Replication - Place copies of (a partition of) data on different machines

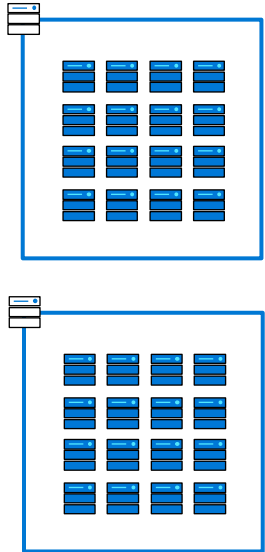
Goal: Offer same functionality and transactional semantics as an RDBMS with higher availability, durability, performance, scalability.

Reality: Concessions in terms of functionality, transactional semantics, and performance

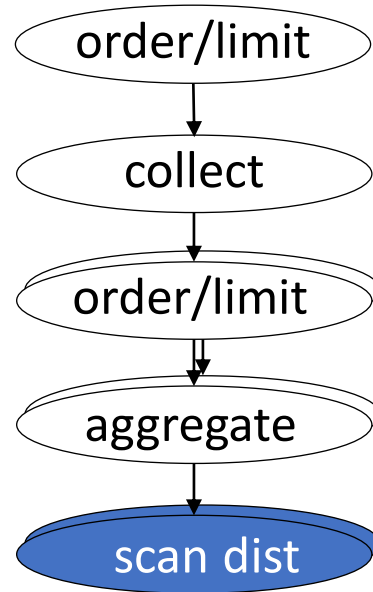
Everyone makes different concessions.

Distribution challenges

Data distribution



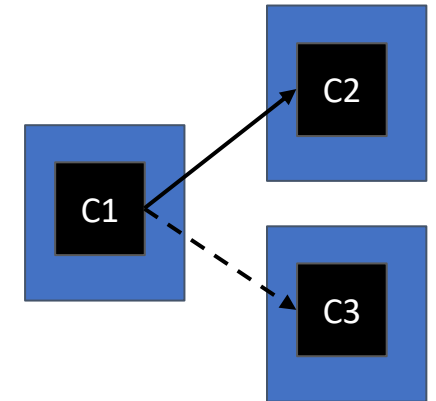
Data access (SQL)



Transactions

```
BEGIN;  
UPDATE account SET b += 20  
WHERE account_id = 1149274;  
UPDATE account SET b -= 20  
WHERE account_id = 8523861;  
END;
```

Replication



Data distribution

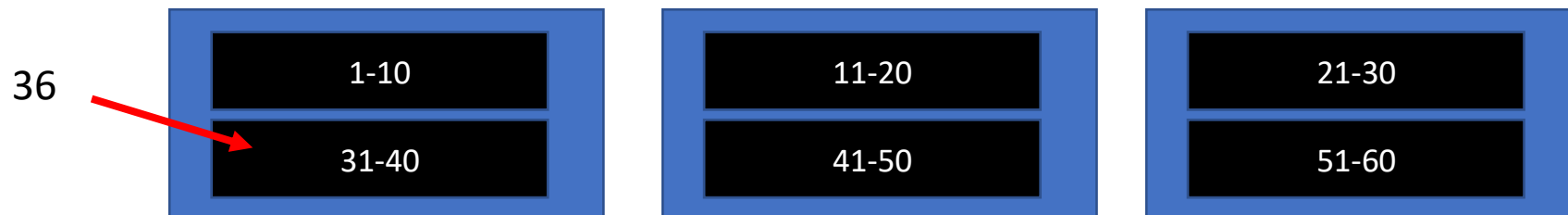
Remember that spreading out data over many machines makes things *slower*

Data distribution: Range-distribution

Tables are partitioned by a “distribution key” (part of primary key)

```
INSERT INTO dist_table (dist_key, other_key) VALUES (36, 12);
```

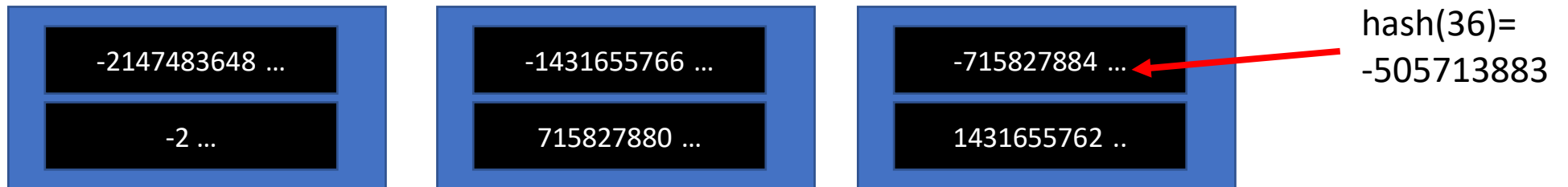
Each “shard” contains a range of values



Data distribution: Hash-distribution

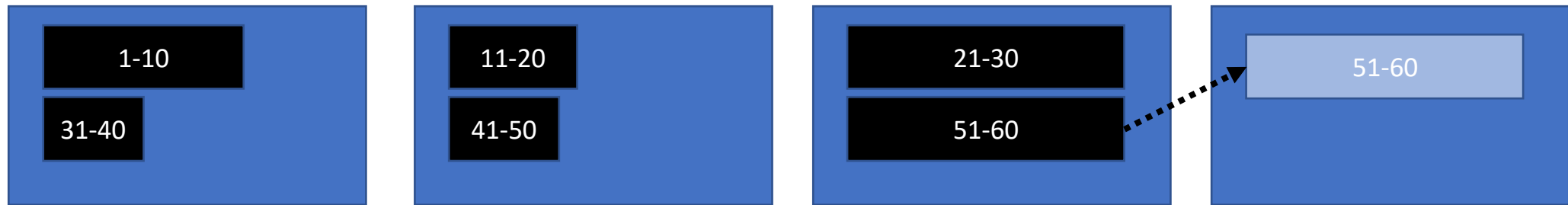
```
INSERT INTO dist_tables (dist_key, other_key) VALUES (36, 12);
```

Each shard contains a range of *hash* values

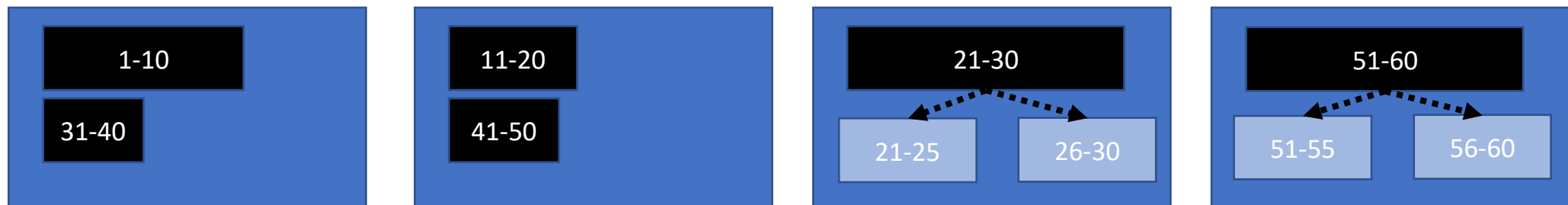


Data distribution: Rebalancing

Move shards to achieve better data distribution across nodes

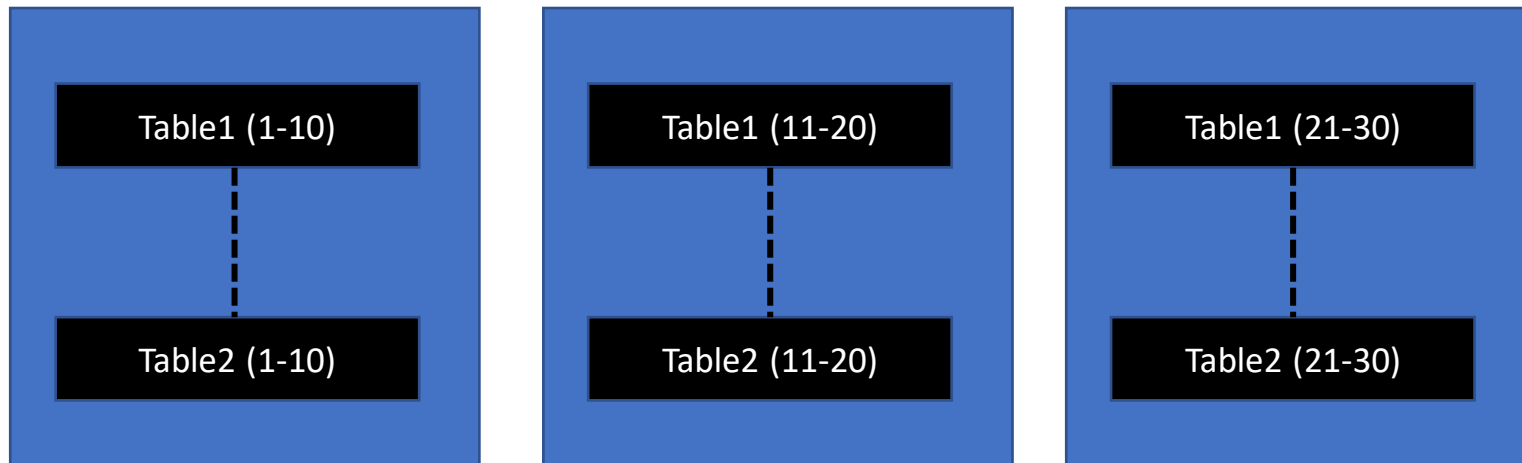


Split shards to achieve better data distribution across shards



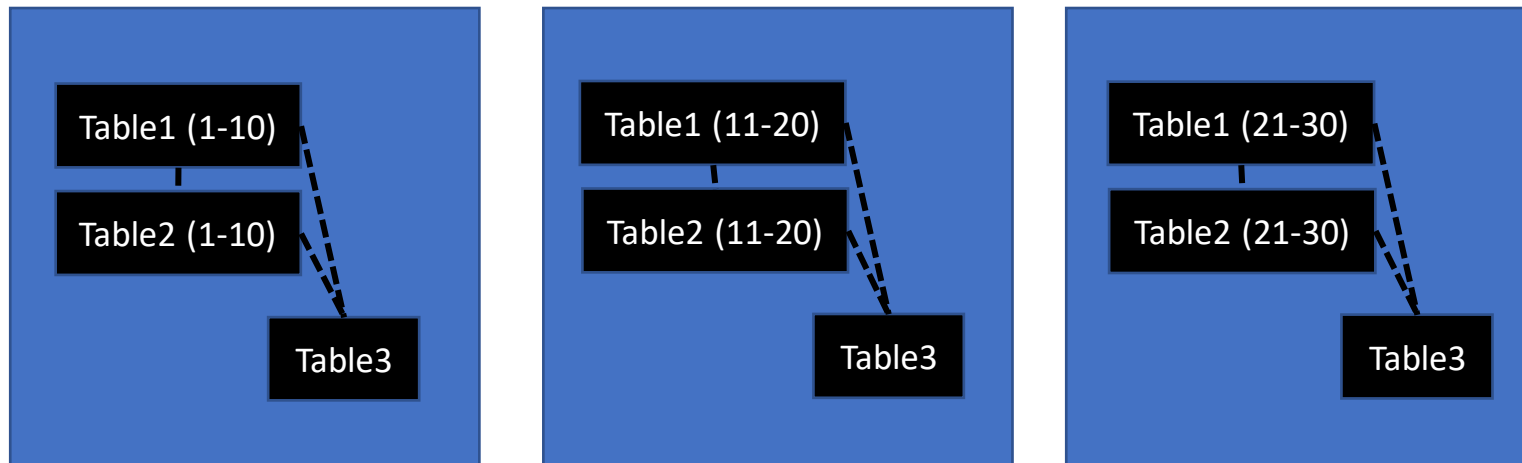
Data distribution: Co-location

Ensure same range is on same node across different tables to enable fast joins, foreign keys, and other operations on distribution key.



Data distribution: Reference tables

Replicate a small table to all nodes to enable fast joins, foreign keys, and other operations on any column.



Distributed SQL

Sometimes faster than regular SQL

Distributed SQL

SQL \approx Relational algebra

Distributed SQL \approx Multi-relational algebra

Relational algebra:

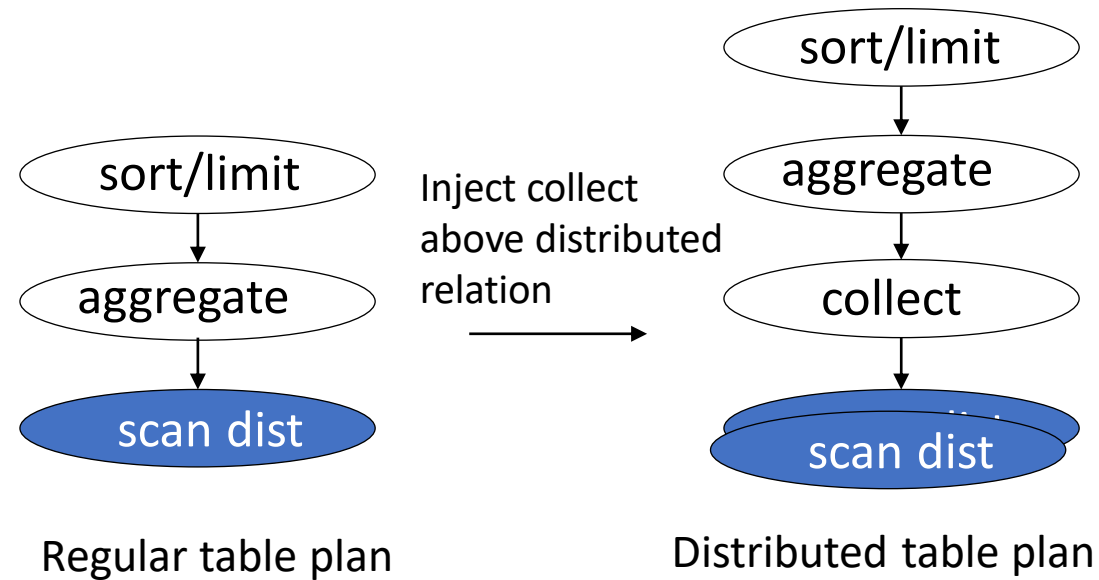
- Scan, Filter, Project, Join, (Aggregate, Order, Limit)

Multi-relational algebra:

- Collect, Repartition, Broadcast + Relational algebra

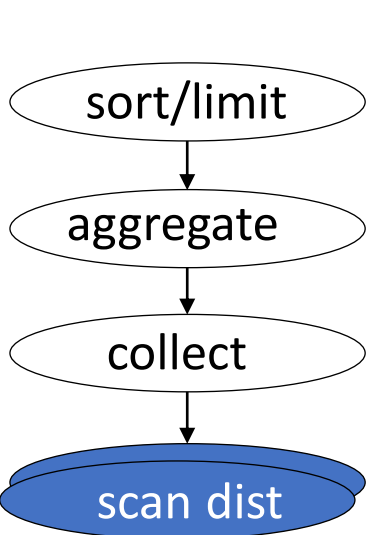
Distributed SQL: Logical planning

```
SELECT dist_key, count(*) FROM dist_table GROUP BY 1 ORDER BY 2 LIMIT 10;
```



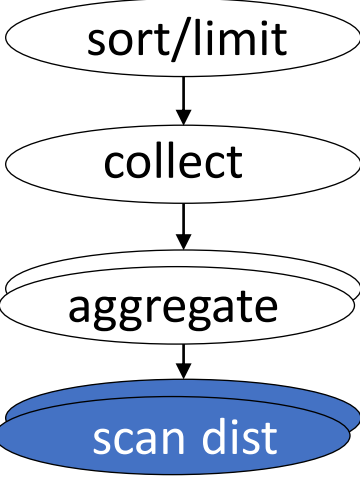
Distributed SQL: Logical optimization

SELECT dist_key, count(*) FROM dist_table GROUP BY 1 ORDER BY 2 LIMIT 10;



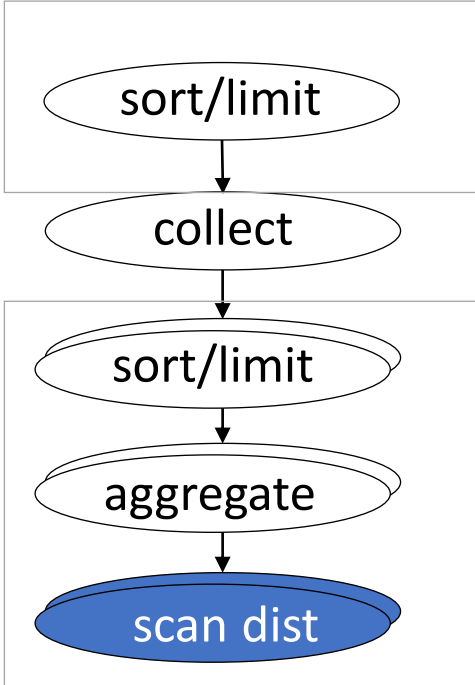
Group by dist. key is commutative with collect

$$aggregate(collect(x)) = collect(aggregate(x))$$



Order/limit can be partially pushed down

$$sort_limit(collect(x), N) = collect(sort_limit(x, N))$$

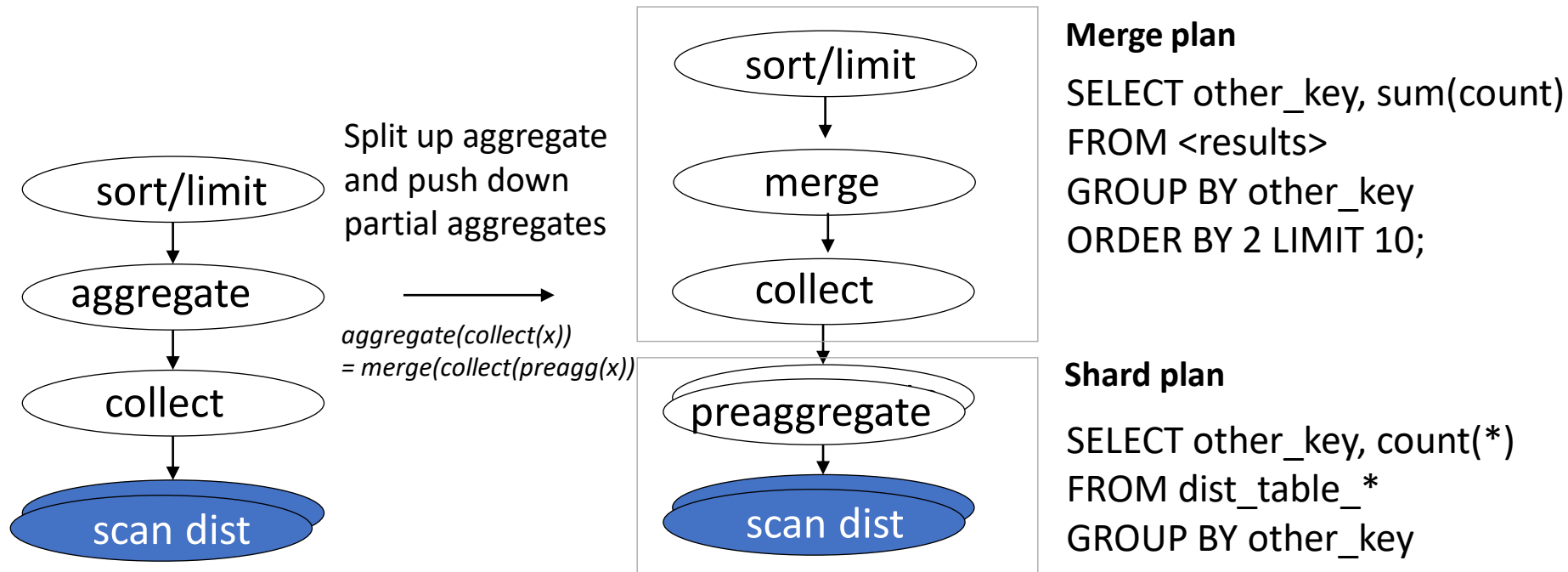


Merge plan
 SELECT dist_key, count FROM <results>
 ORDER BY 2 LIMIT 10;

Shard plan (can run in parallel)
 SELECT dist_key, count(*) FROM dist_table_*
 GROUP BY 1
 ORDER BY 2 LIMIT 10;

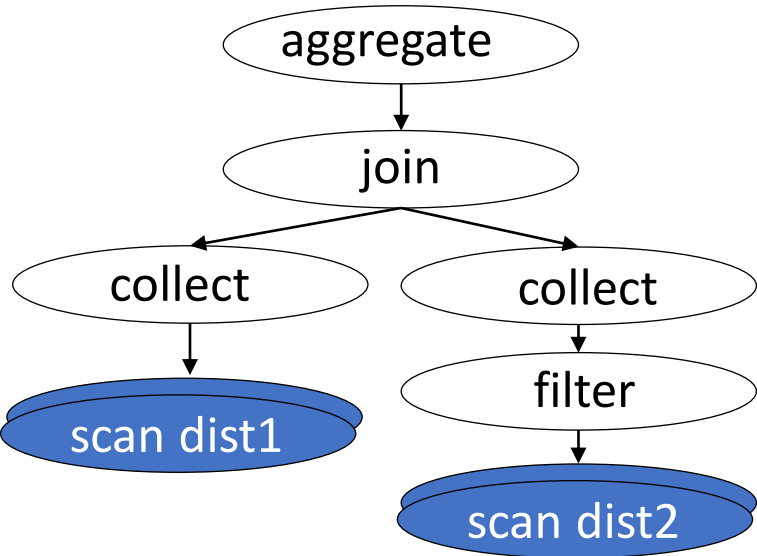
Distributed SQL: Logical optimization

SELECT **other_key**, count(*) FROM dist_table GROUP BY 1 ORDER BY 2 LIMIT 10;



Distributed SQL: Co-located joins

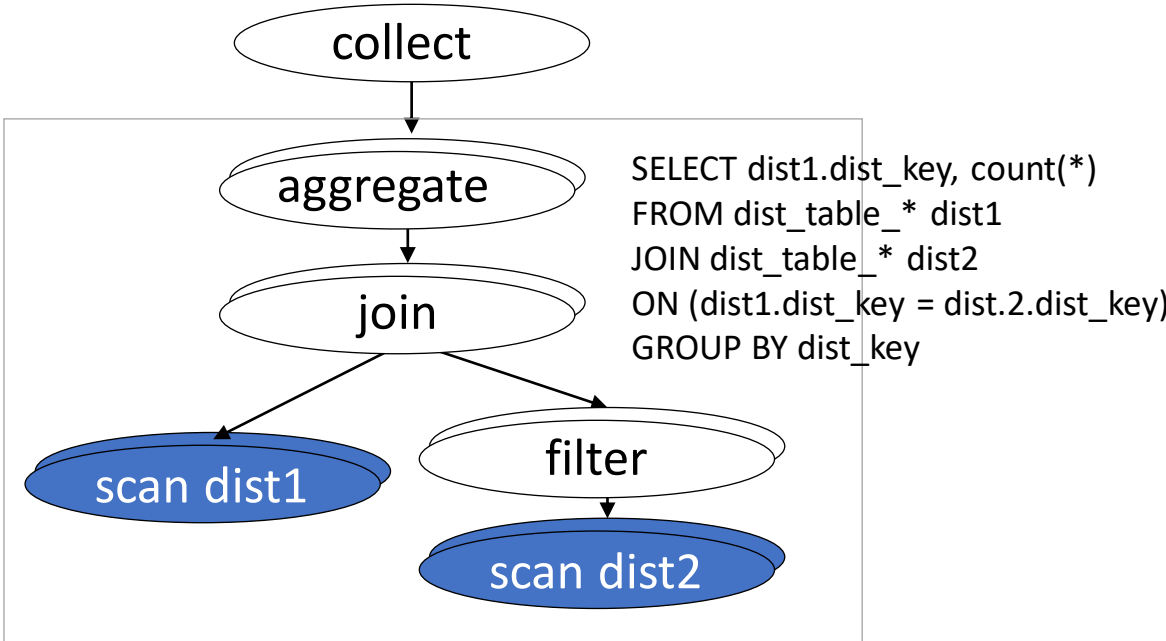
```
SELECT dist1.dist_key, count(*)  
FROM dist1 JOIN dist2 ON (dist1.dist_key = dist2.dist_key)  
WHERE dist2.value < 44 GROUP BY dist1. dist_key;
```



Filter is commutative with collect

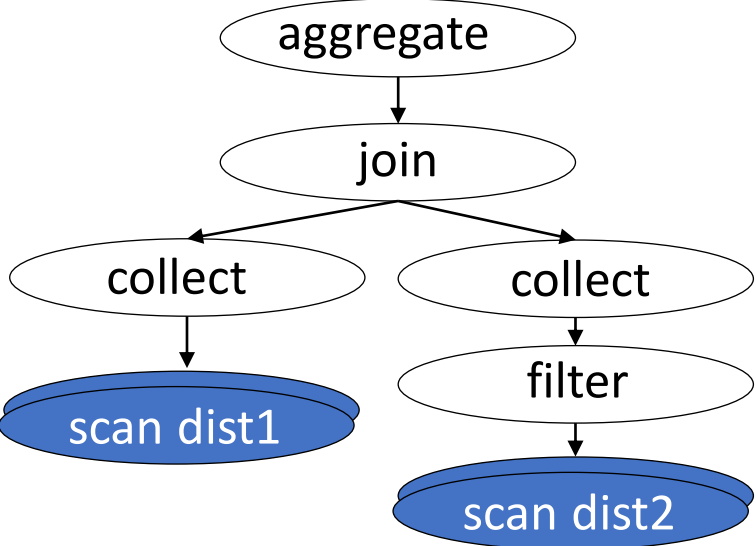
Join is co-located so distributive with 2 collect nodes

Group by dist. key is commutative with collect



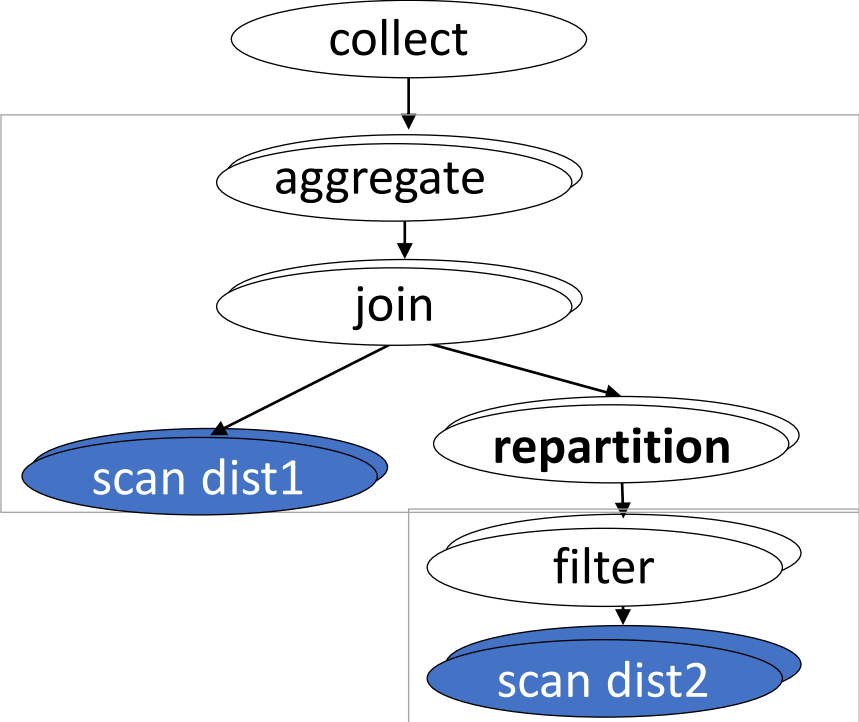
Distributed SQL: Re-partition joins

```
SELECT dist1.dist_key, count(*)  
FROM dist1 JOIN dist2 ON (dist1.dist_key = dist2.other_key)  
WHERE dist2.value < 44 GROUP BY dist1.dist_key;
```



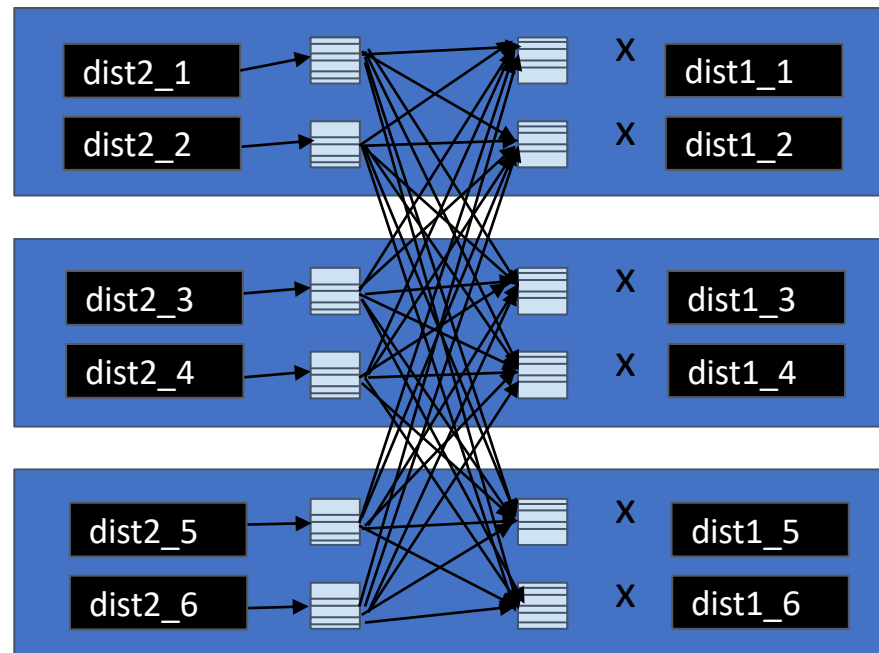
Need to re-partition data to perform join

Group by dist. key is commutative with collect



Distributed SQL: Re-partition operations

```
SELECT dist1.dist_key, count(*)  
FROM dist1 JOIN dist2 ON (dist1.dist_key = dist2.other_key)  
WHERE dist2.value < 44 GROUP BY dist1.dist_key;
```

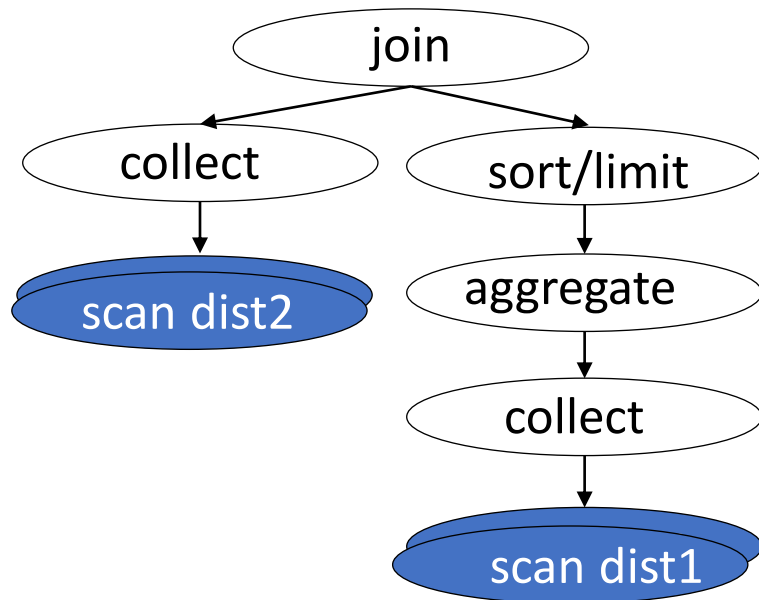


```
SELECT other_key  
FROM dist2_*  
WHERE value < 44;
```

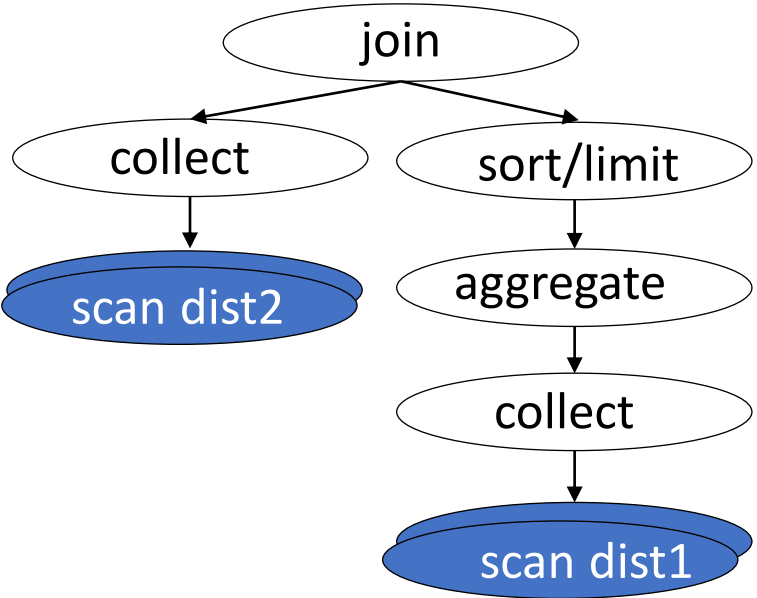
```
SELECT dist1.dist_key, count(*)  
FROM dist1_* JOIN <results>  
ON (dist1_*.dist_key = <results>.other_key)  
GROUP by dist1.dist_key;
```

Distributed SQL: Broadcast joins

```
WITH top10 AS (  
  SELECT other_key, count(*) FROM dist1 GROUP BY 1 ORDER BY 2 LIMIT 10  
)  
SELECT * FROM dist2 WHERE other_key IN (SELECT dist_key FROM top10);
```

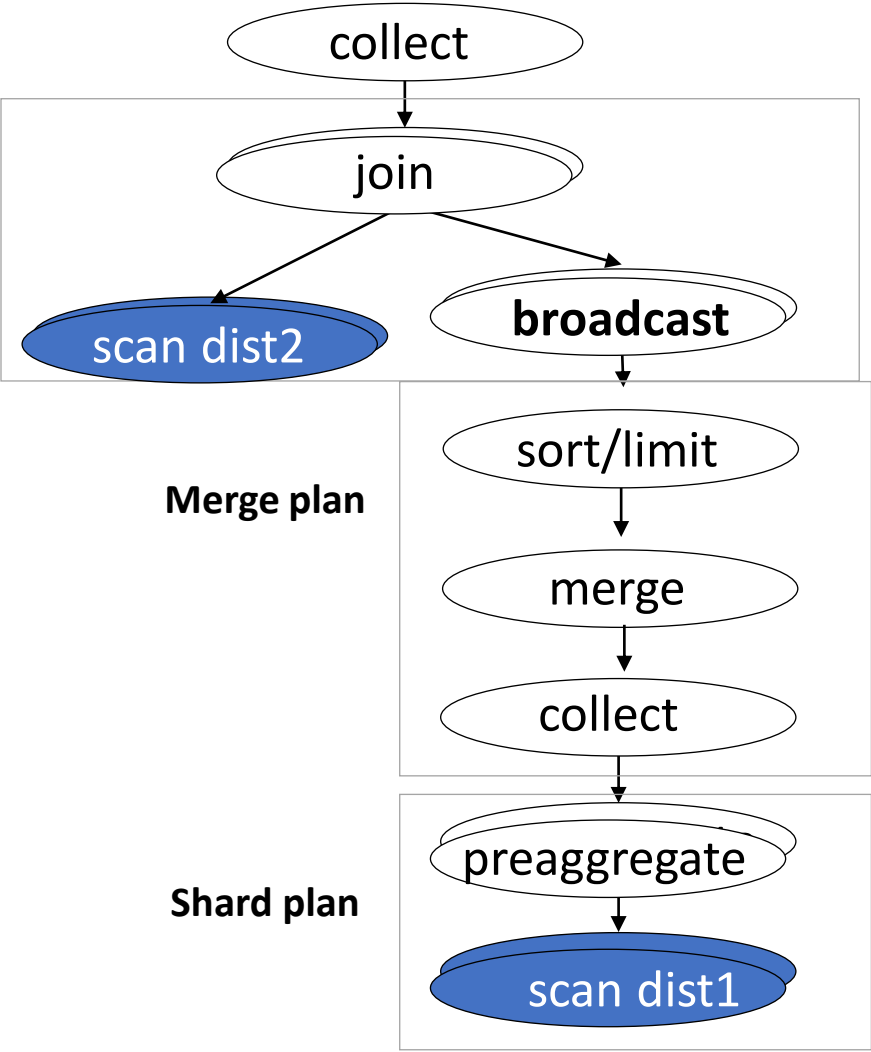


Distributed SQL: Broadcast joins



Create subplan to handle order/limit under join
→
Broadcast subplan to pull collect above the join

Shard plan



Distributed SQL: Observations

Query plans depend heavily on the distribution key.

Runtime also depends on query, data, data size (big in distributed databases), network speed, cluster size,

Distributed databases require adjusting your distribution keys & queries to each other to achieve high performance.

Distributed Transactions

Where the trouble starts...

Distributed Transactions

Ideally, we have:

Atomicity, Consistency, Isolation, Durability (ACID)

Main distribution challenges:

Atomicity - Commit on all nodes or none

Isolation - See other distributed transactions as committed/aborted

Additionally:

Distributed deadlock detection

Distributed Transactions: Atomicity

Atomicity is generally achieved through 2PC = 2-Phase Commit

- | | |
|-----------------|--|
| Phase 1: | Store (“prepare”) transactions on all nodes |
| Phase 2: | Store final commit decision and ... |
| If success, | Commit all prepared transactions |
| If error, | Abort all prepared transactions |
| Secret phase 3: | Commit/abort prepared transactions after failure |

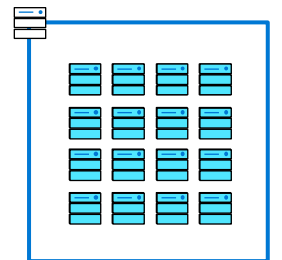
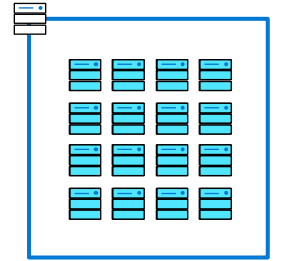
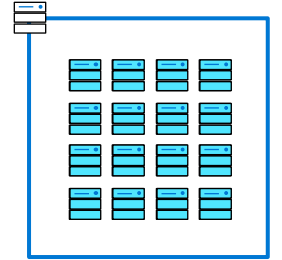
```
BEGIN;  
UPDATE campaigns  
  SET started = true  
  WHERE campaign_id = 2;  
UPDATE ads  
  SET finished = true  
  WHERE campaign_id = 1;  
COMMIT;
```



```
BEGIN ...  
assign_distributed_  
transaction_id ...  
UPDATE campaigns_102 ...  
PREPARE TRANSACTION...  
COMMIT PREPARED...
```

```
BEGIN ...  
assign distributed_  
transaction_id ...  
UPDATE campaigns_203 ...  
PREPARE TRANSACTION...  
COMMIT PREPARED...
```

WORKER NODES



2PC recovery

```
SELECT gid FROM pg_prepared_xacts  
WHERE gid LIKE 'citus_%d_%'
```

Compare

worker	Prepared xact
W1	citus_0_2413
W2	citus_0_2413

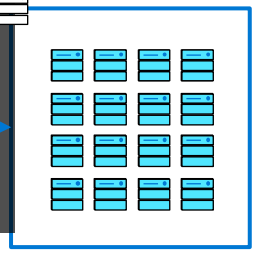
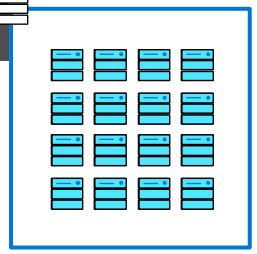
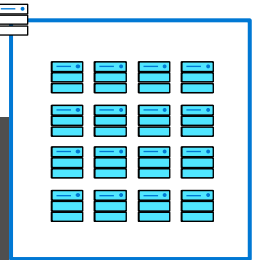


**COORDINATOR
NODE**

WORKER NODES

```
BEGIN ...  
assign_distributed_  
transaction_id ...  
UPDATE campaigns_102 ...  
PREPARE TRANSACTION citus_0_2431;  
COMMIT PREPARED...
```

```
BEGIN ...  
assign distributed_  
transaction_id ...  
UPDATE campaigns_203 ...  
PREPARE TRANSACTION citus_0_2431;  
COMMIT PREPARED ...;
```



Distributed Transactions: Isolation

If we query different nodes at different times, we may see a concurrent transaction as committed on one node, but not yet committed on another.

Distributed snapshot isolation means we have the same of view of what is committed and not committed on all the nodes.

Additional requirements:

read-your-writes consistency: Any preceding write is seen as committed.

monotonic read consistency: Subsequent reads always see newer data

Distributed Snapshot Isolation Approaches

Many different solutions, none great:

Heavy locks:	Greenplum	(low concurrency)
Hybrid logical clocks:	CockroachDB, Yugabyte	(slow)
Global transaction manager:	PolarDB, TBase	(limited scale)
No distributed isolation:	Citus, TimescaleDB	(anomalies)
Single primary:	AlloyDB, Aurora	(limited scale)
TrueTime:	Spanner	(slow)

Replication

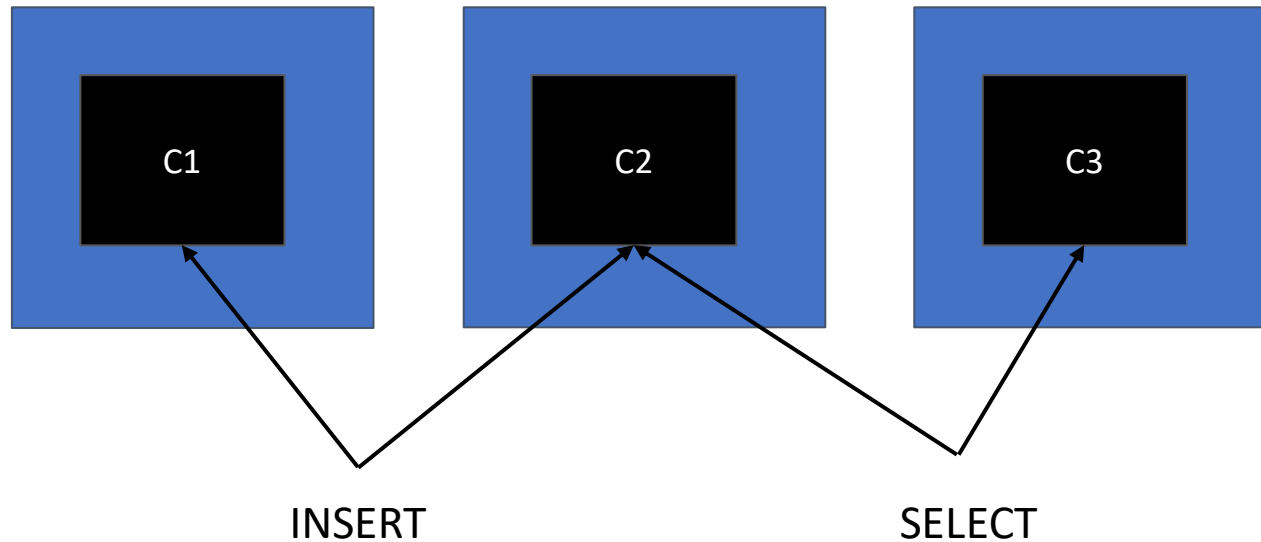
Trade-offs all the way down

Why replication?

- availability - resume from replica in case of node failure
- durability - no data loss in case of node failure
- read throughput - divide reads across read replicas
- read latency - local/nearby replica gives lower read latency
- write latency - local/nearby replica gives lower write latency

Replication: Quorums

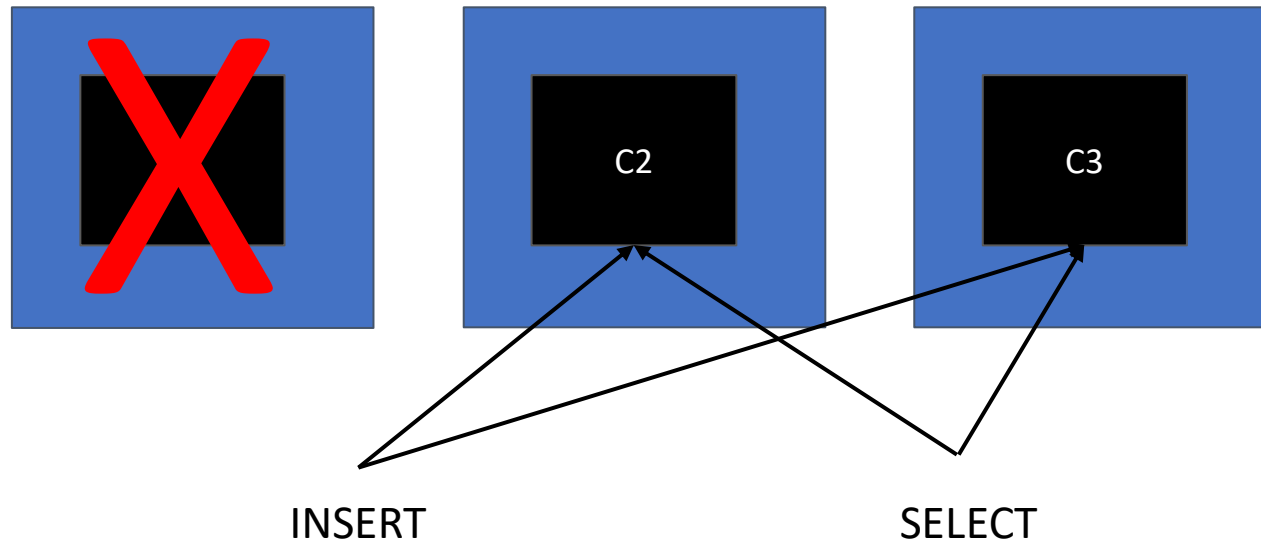
Basic idea: Read from R nodes, Write to W nodes, $R + W > N$



Challenge: Applying events in same order everywhere

Replication: Quorums

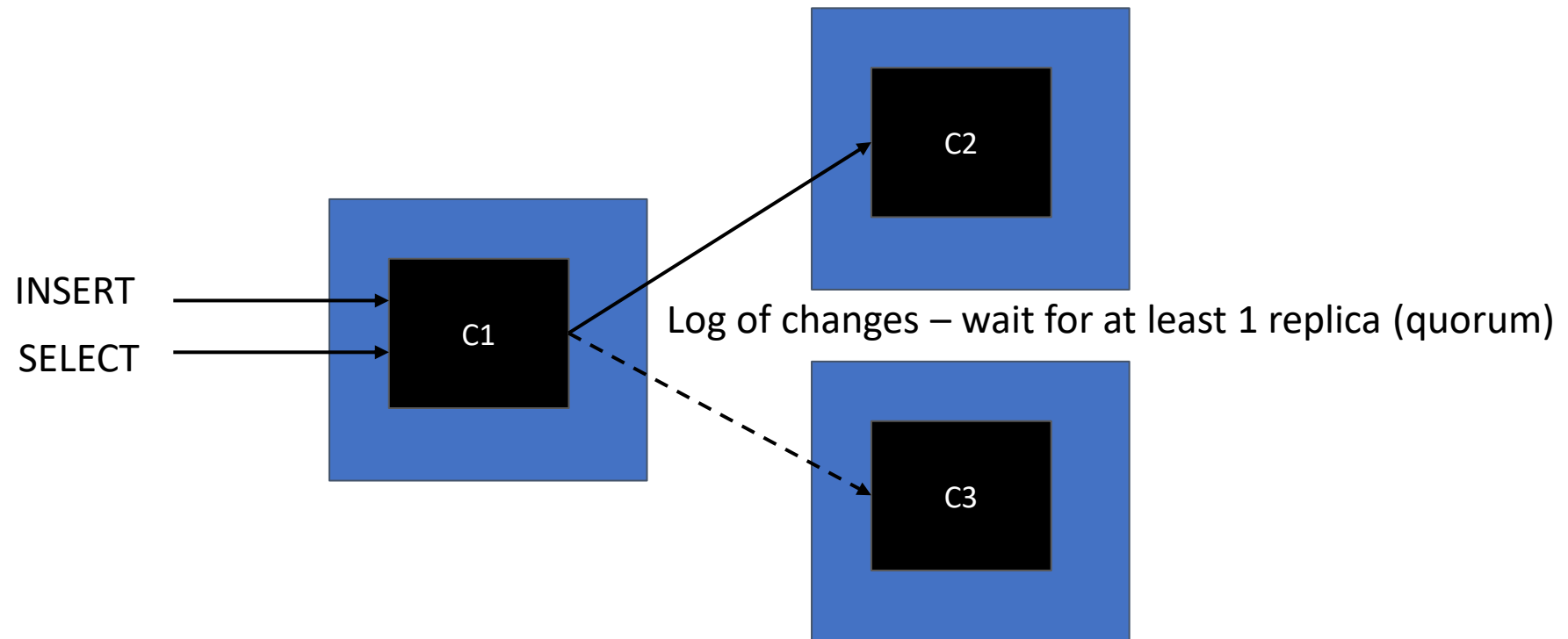
Basic idea: Read from R nodes, Write to W nodes, $R + W > N$



Challenge: Applying events in same order everywhere

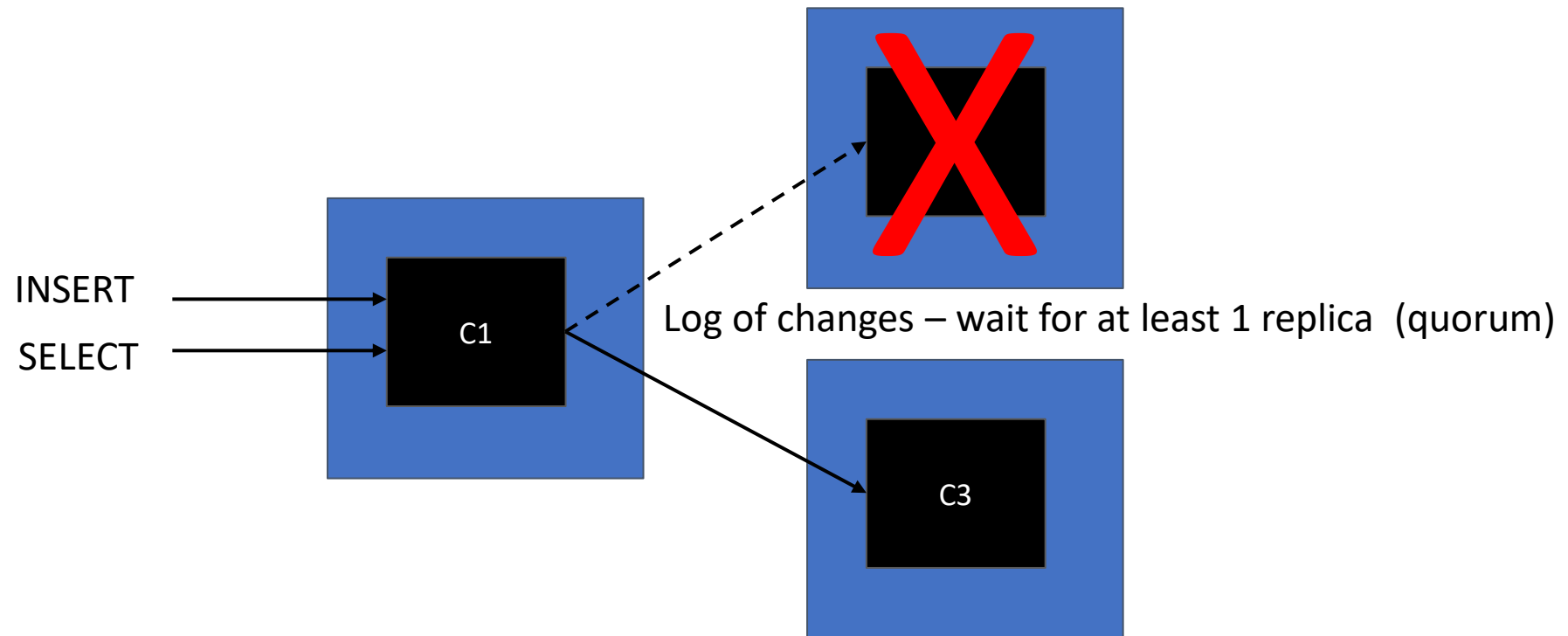
Replication: Active-passive (follow the leader)

Assign temporary leader to serialize writes efficiently



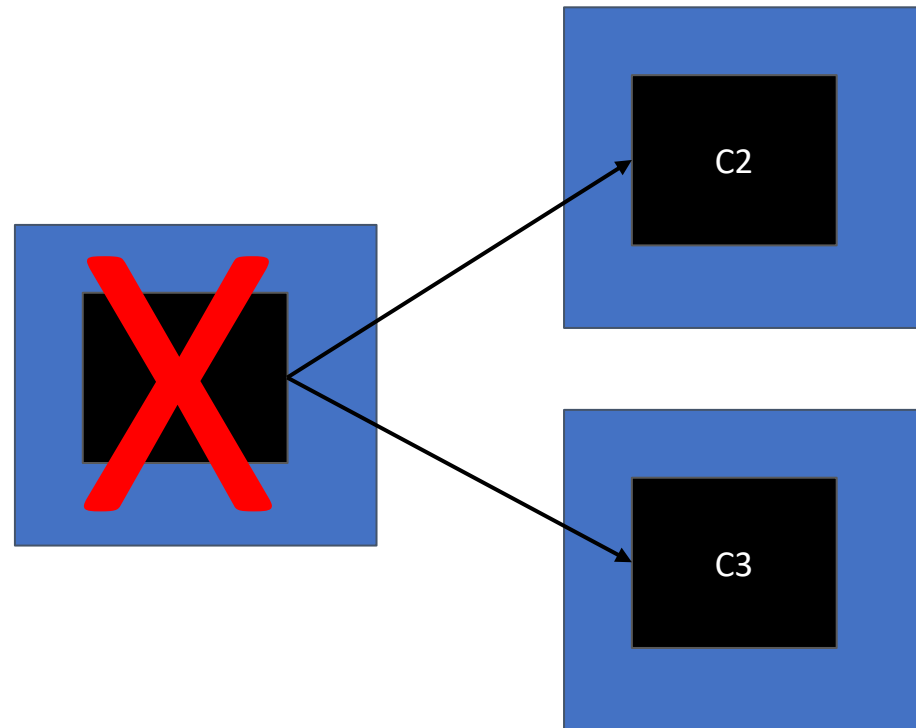
Replication: Active-passive (follow the leader)

Standby fails: Continue writing to other replica



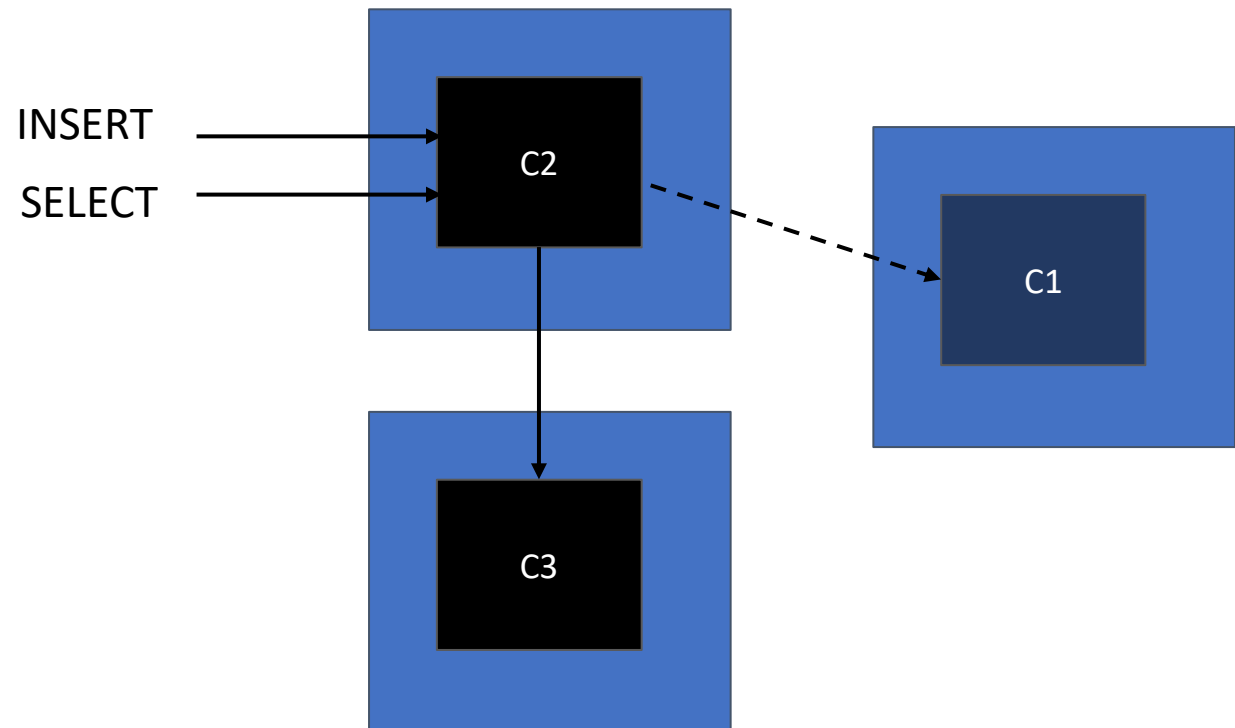
Replication: Active-passive (follow the leader)

Primary fails: Initiate a failover (choose new one through Paxos/Raft)



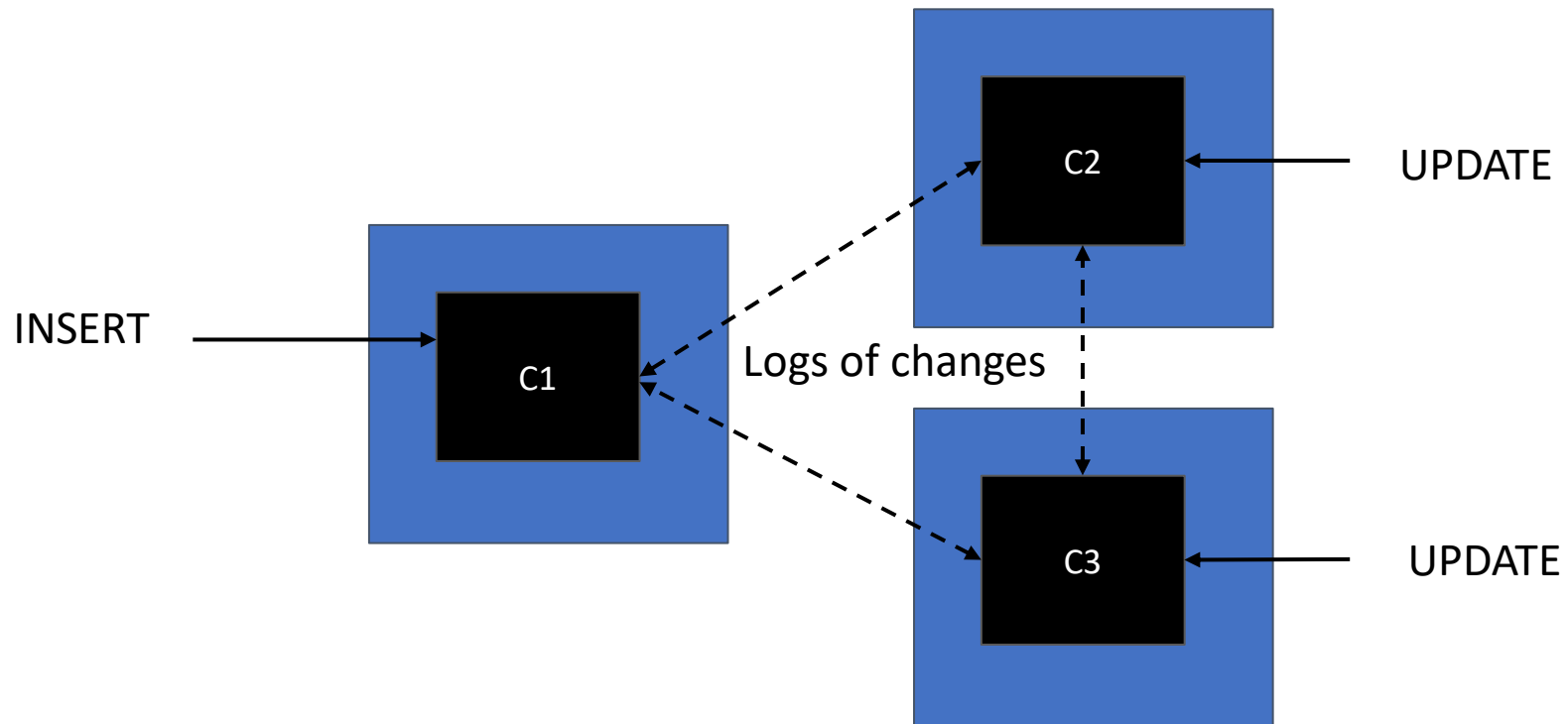
Replication: Active-passive (follow the leader)

Replica is promoted to leader, other replicas follow new leader.



Replication: Active-active (n-directional)

All nodes accept writes, somehow reconcile conflicting changes.



CAP theorem

Choose **C**onsistency vs **A**vailability when in a (minority) network **P**artition

Availability (A) = Keep writing to minority of nodes, majority does not see it

Consistency (C) = Writes/reads unavailable, consistency must be preserved

Very incomplete picture of distributed database trade-offs

Distributed PostgreSQL is generally CP

PACELC theorem

Slightly better, but still oversimplified:

If Network **P**artition: choose **A**vailability vs. **C**onsistency

Else: choose **L**atency vs. **C**onsistency

Other distributed database trade-offs

Consistency

- Read-your-writes
- No lost updates
- Linearizability

Availability

- For Reads
- For Writes
- Handle availability zone failure

Partition-tolerance

- For Reads
- For Writes

Durability

- Node failure does not result in data loss
- Writes are archived in a timely manner

Low latency

- Low read latency
- Low write latency
- Global vs. local

Complexity

- Dependencies on other systems
- Multiple node types
- Many optimizations

The Distributed PostgreSQLs

All distributed databases are bad, some are less bad than others for your use case.

Distributed PostgreSQL landscape



AlloyDB (Google)



Citus (Microsoft)



Spanner (Google)



TimescaleDB



Greenplum (VMWare)



Aurora (Amazon)



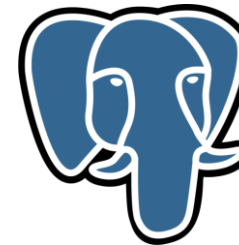
CockroachDB



Yugabyte



PolarDB (Alibaba)



PostgreSQL



TBase (Tencent)

Amazon Aurora

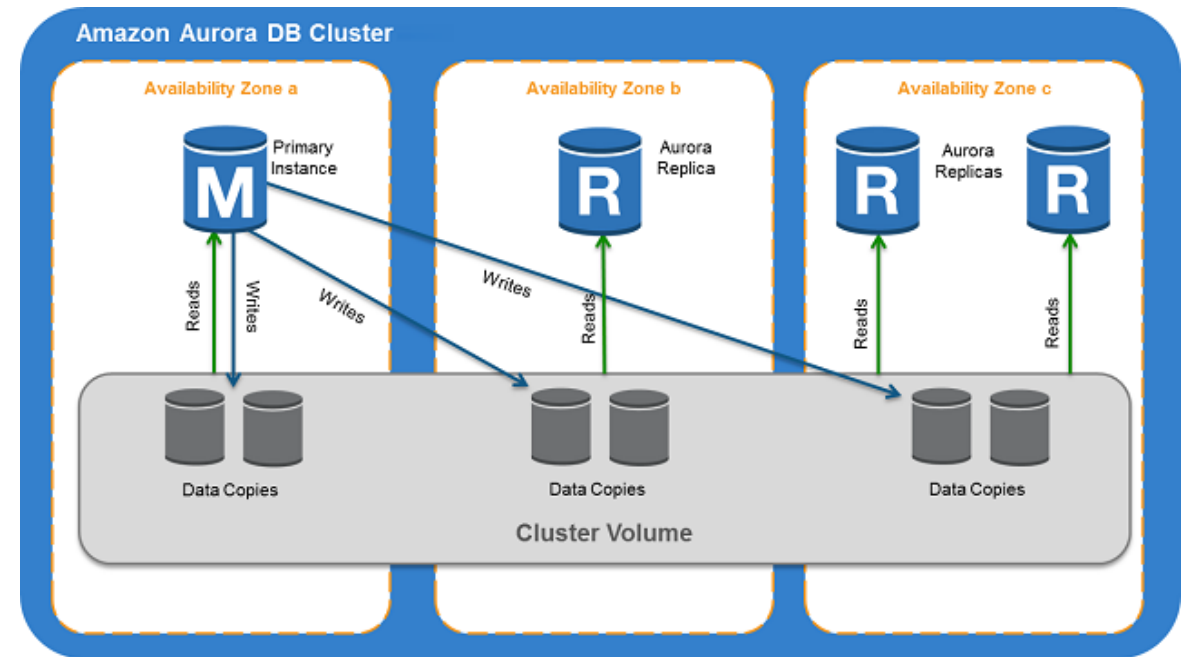
PostgreSQL fork that distributes and replicates storage for higher performance (IOPS), better fault tolerance.

Replicas read from storage layer without load on primary, have low lag.

Fast backup/crash recovery.

Does not scale writes, large working sets, large queries.

Proprietary, only runs in AWS.



Source:
docs.aws.amazon.com

Citus (Microsoft)

PostgreSQL extension that adds distributed tables, reference tables & columnar storage.

Scales query throughput, large queries.

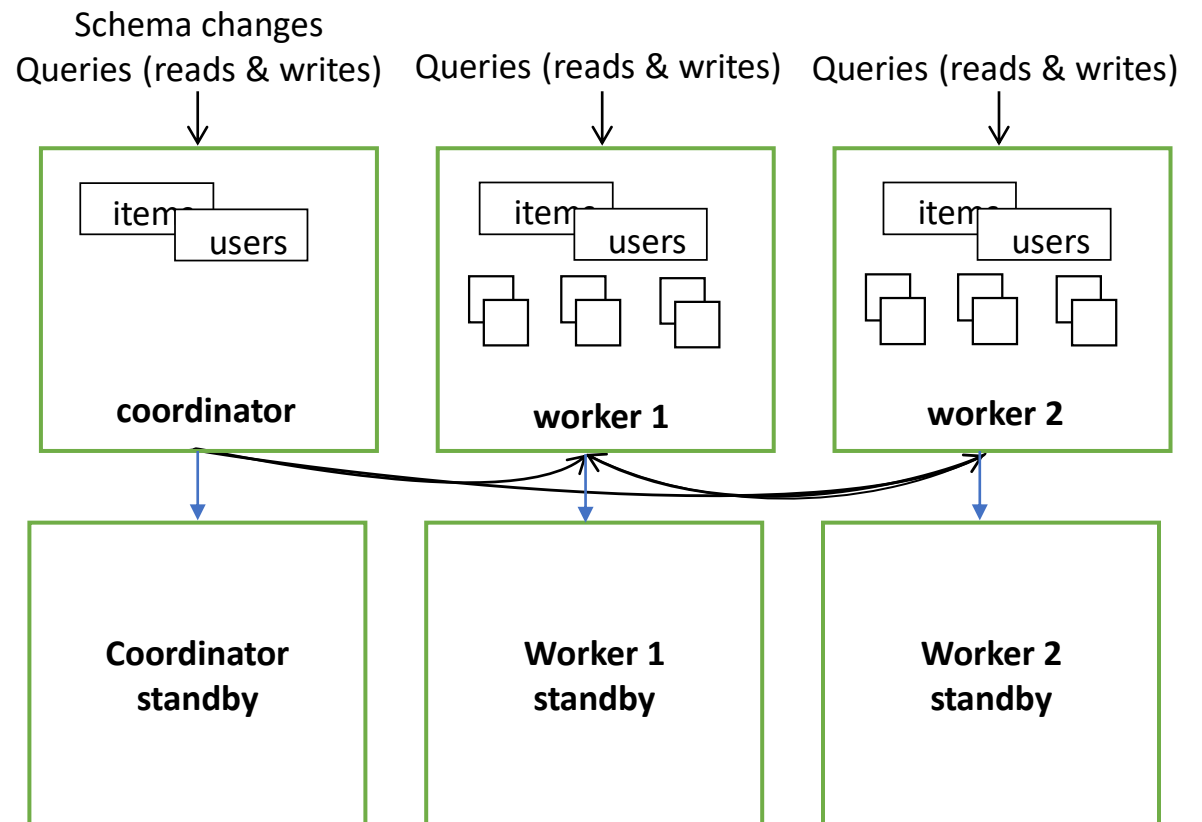
High performance via co-location, reference tables.

Always up-to-date with PostgreSQL.

No distributed snapshot isolation.

Limitations on foreign keys, joins..

Open source (AGPL)



CockroachDB

Distributed key-value store that speaks the PostgreSQL protocol.

High availability using Raft.

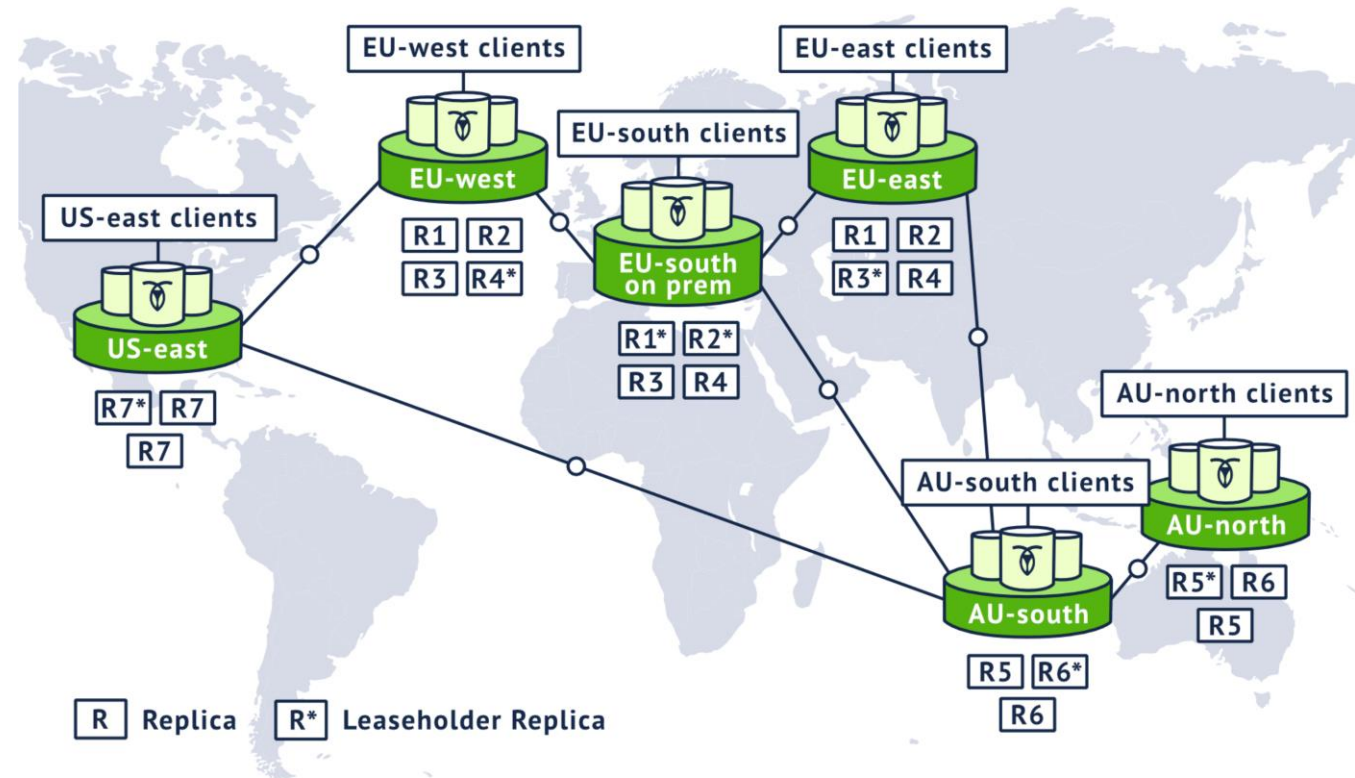
Snapshot isolation using HLCs.

Geo-replication & geo-partitioning.

Not very fast.

Limited PostgreSQL compatibility.

Open source (complex license)



Greenplum

PostgreSQL fork for data warehousing started in 2003, caught up to PostgreSQL 12

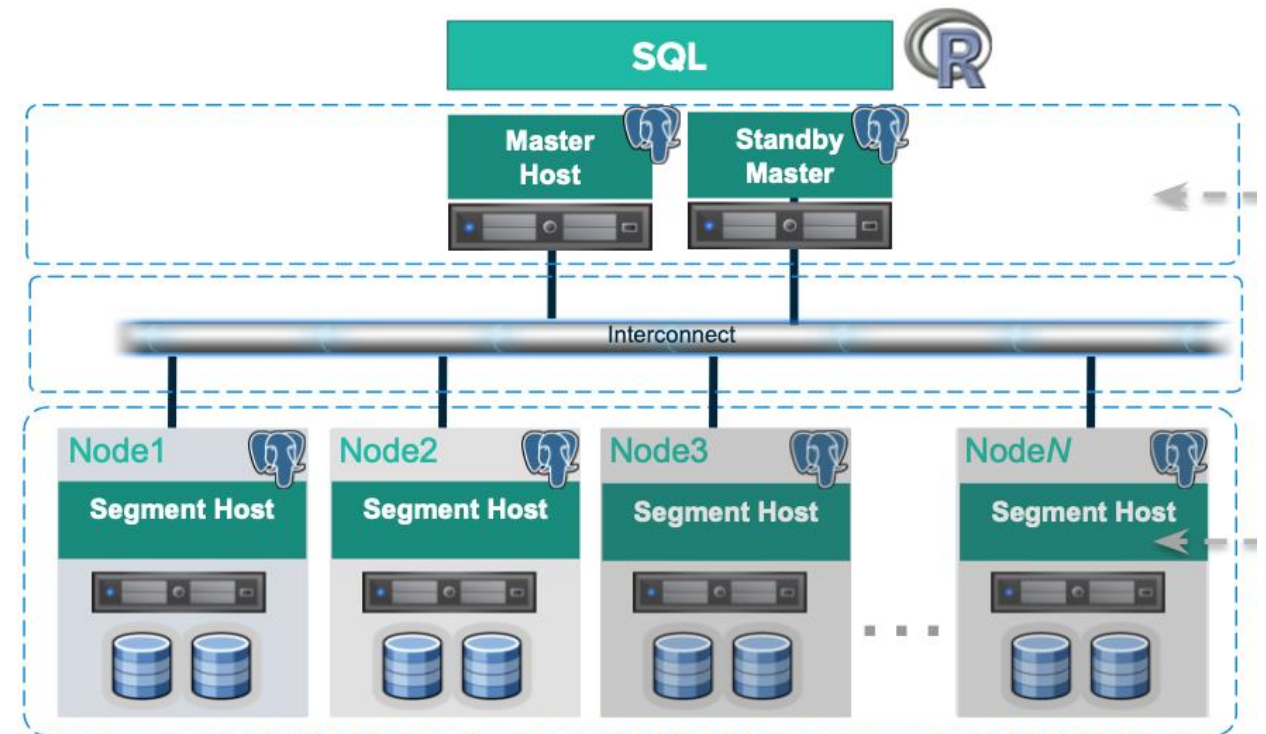
Advanced query planner for complex analytical queries on large data sets.

Complex architecture.

Transactional capabilities still slow.

Less relevant given Snowflake, Spark, ...

Open source (Apache 2.0)



PolarDB for PostgreSQL (Alibaba)

Derived from Postgres-XL (defunct PostgreSQL fork from 2014), caught up to PostgreSQL 11.

Good PostgreSQL compatibility.

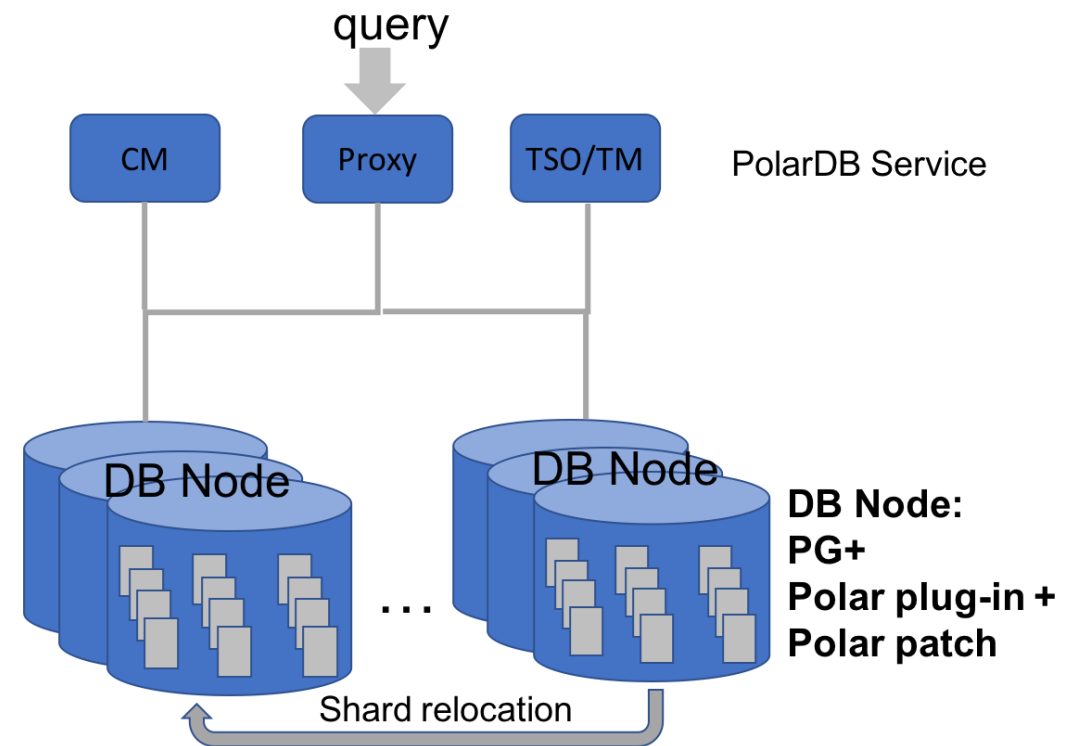
Snapshot isolation using timestamp server.

Also has an Aurora-like variant.

Complex architecture.

Old PostgreSQL version.

Open source (Apache 2.0)



Yugabyte

Distributed key-value store that includes a fork of PostgreSQL 11.

High availability using Raft.

Snapshot isolation using HLCs.

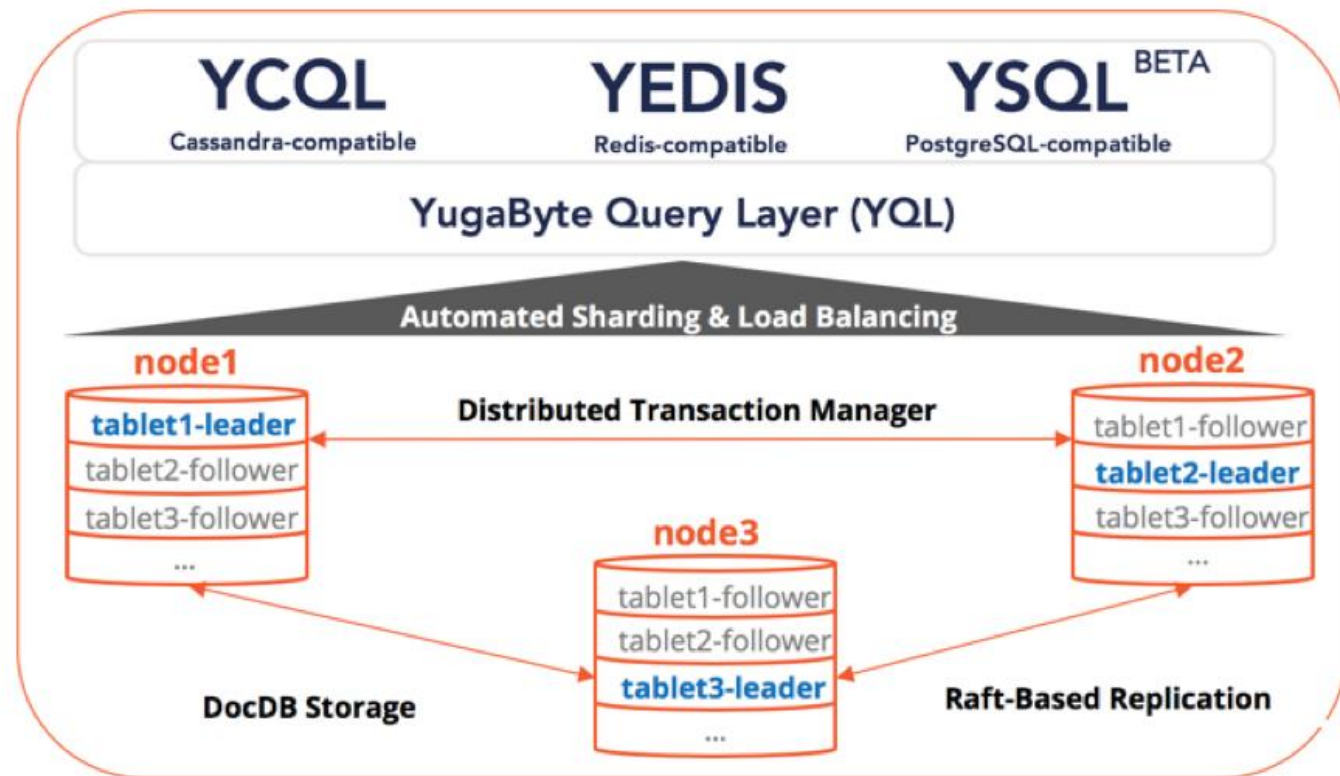
Geo-replication & geo-partitioning.

Relatively good PostgreSQL compat.

Not very fast or stable.

Old PostgreSQL version.

Open source (Apache 2.0)



Navigating Distributed PostgreSQL

For analytical dashboards:

Citus

For analytical reporting:

Greenplum (or non-PG-based systems)

For transactional workloads, if your main concern is:

• Availability:

CockroachDB

• Price-performance:

Aurora

• Write scalability:

Citus

• Data model flexibility:

Aurora, Yugabyte

One to watch:

PolarDB

Ongoing developments

Auto-distribution

Auto-indexing

Branching

Built-in compute runtime

Decoupled storage & compute

Disaggregated memory

Faster snapshot isolation

Geo-distributio

In-database machine learning

Integration with cloud services

Learned indices

Multi-modal databases

NVME drives

Self-driving databases

Serverless

High write-scalability

Why use Distributed PostgreSQL?

Either you have a really challenging data problem, or you buy into the dream

When to use Distributed PostgreSQL

Key-value storage

- Scales to very high throughput, SQL & transactions

Multi-tenant applications

- Distribute by tenant ID, co-locate data by tenant ID, can efficiently handle complex queries

Site-facing analytics

- Distributed SQL, Indexes, Partitioning, Views, Distributed insert..select, Extensions, ...

Analytics with n-dimensional data types

- Time series, spatial, spatiotemporal, ...

When not to use Distributed PostgreSQL

Complex / normalized data models

- Complex join and foreign key graphs slow down any distributed PostgreSQL system

Analytical reporting on a large data lake

- State-of-the-art is Spark, Snowflake, Synapse, Presto, ...

My personal bias: Citus

PostgreSQL extension is a huge benefit

- Always up-to-date with PostgreSQL developments
- Users can take advantage of mature implementations of PostgreSQL features, other extensions
- Start on single node PostgreSQL, scale out later

Designed to offer high performance at scale

Open source

Good traction in very large-scale software-as-a-service, site-facing analytics / IoT

Microsoft investing in it for the long run

Preferred by Stonebraker 😊 [\[1\]](#) [\[2\]](#) [\[3\]](#)

Questions?

Marco Slot - marco.slot@microsoft.com

Principal software engineer at Microsoft