Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #25

Final Review & Systems Potpourri

FALL 2023 » Prof. Andy Pavlo • Prof. Jignesh Patel

ADMINISTRIVIA

Project #4 is due **Sunday Dec 10th @ 11:59pm**

→ Extra Office Hours: **Saturday Dec 9th @ 3:00-5:00pm**

→ Location: GHC 4407

SPRING 2024

Jignesh is recruiting impressionable TAs for 15-445/645 in Spring 2024.

→ All BusTub projects will remain in C++.

→ You are **not** expected to be like Chi.

Sign up here:

https://www.ugrad.cs.cmu.edu/ta/S24/

COURSE EVALS

Your feedback is strongly needed:
→ https://cmu.smartevals.com
→ https://www.ugrad.cs.cmu.edu/ta/F23/feedback/

Things that we want feedback on:
→ Homework Assignments
→ Projects
→ Reading Materials
→ Lectures

OFFICE HOURS

Andy:

→ Monday Dec 11th @ 9:30-10:30am
→ Zoom: https://cmudb.io/pavlo-zoom

Jignesh:

→ Monday Dec 11th @ 1:00-2:00pm ET
→ Zoom: https://cmu.zoom.us/my/jignesh

TAs will have their regular office hours up to and including Friday Dec 8th

FINAL EXAM

Who: You

What: Final Exam

Where: POS 153

When: Tuesday Dec 12th @ 8:30am
Why: https://youtu.be/8tuoIO4CxOw

Email instructors if you need special accommodations.

https://15445.courses.cs.cmu.edu/fall2023/final-guide.html

FINAL EXAM

Everyone should come to POS 153.

You will then be assigned a random location.
→ POS 153, HOA 160, HOA 107

There will be TAs stationed in each room to give you the exam and to handle questions.

Instructors will bounce around the rooms during the exam time.

FINAL EXAM

What to bring:

→ CMU ID
→ Pencil + Eraser (!!!)
→ Calculator (cellphone is okay)
→ One 8.5x11" page of handwritten notes (double-sided)

What not to bring:

→ NFT-themed Clothing

STUFF BEFORE MID-TERM

SQL

Buffer Pool Management

Data Structures (Hash Tables, B+Trees)

Storage Models

Query Processing Models

Inter-Query Parallelism

QUERY OPTIMIZATION

Heuristics
- → Predicate Pushdown
- → Projection Pushdown
- → Nested Sub-Queries: Rewrite and Decompose

Statistics
- → Cardinality Estimation
- → Histograms

Cost-based search

TRANSACTIONS

ACID

Conflict Serializability:

→ How to check for correctness?
→ How to check for equivalence?

View Serializability

→ Difference with conflict serializability

Recoverable Schedules

Isolation Levels / Anomalies

TRANSACTIONS

Two-Phase Locking
→ Rigorous vs. Non-Rigorous
→ Cascading Aborts Problem
→ Deadlock Detection & Prevention

Multiple Granularity Locking
→ Intention Locks
→ Understanding performance trade-offs
→ Lock Escalation (i.e., when is it allowed)

TRANSACTIONS

Timestamp Ordering Concurrency Control
→ Thomas Write Rule

Optimistic Concurrency Control
→ Read Phase
→ Validation Phase
→ Write Phase

Multi-Version Concurrency Control
→ Version Storage / Ordering
→ Garbage Collection
→ Index Maintenance

CRASH RECOVERY

Buffer Pool Policies:
→ STEAL vs. NO-STEAL
→ FORCE vs. NO-FORCE

Write-Ahead Logging

Logging Schemes
→ Physical vs. Logical

Checkpoints

ARIES Recovery
→ Analyze, Redo, Undo phases
→ Log Sequence Numbers
→ CLRs

DISTRIBUTED DATABASES

System Architectures

Replication

Partitioning Schemes

Two-Phase Commit

TOPICS NOT ON EXAM!

SingleStore

Details of specific database systems (e.g., Postgres) redis

REDIS (2009)

Remote **Di**ctionary **S**erver

Key-value DBMS written in C with specialized value types:

→ Values can be strings, hashes, lists, sets and sorted sets.
→ Specific commands for each value type.
→ Single-threaded execution engine.

Mostly used as an in-memory cache.

Lots of clones (commercial, hobbyist).

KeyDB Dragonfly Tendis

REDIS – DATA MODEL

Keys → Values

STRING

Key		Value
page:index.html	→	"<html><head>..."
view_count	→	12345

SET

Key		Value
users_logged_in	→	{1, 2, 3, 4, 5}

LIST

Key		Value
latest_post_ids	→	{111, 112, 119, …}

HASH

Key		Value
user:999:session	→	time => 1430086922 username => tupac

SORTED SET

Key		Value
current_scores	→	odb ~ 11 tupac ~ 12 biggie ~ 19 eazye ~ 20

REDIS – INTERNALS

In-memory storage:
→ Periodic Snapshots + WAL for persistence.
→ No buffer pool.

Single-threaded execution engine using a chained hash table to store databases.
→ No secondary indexes.
→ No schema / constraints

REDIS – INTERNALS

Supports some notion of transactions:

→ Operations are batched together and executed serially on server side.
→ Allows for compare-and-swap.
→ Does not support rollback!

Asynchronous primary-replica replication:

→ Master sends oplog to downstream replicas.
→ Primary waits until at least some replicas are available before accepting writes but still not check whether they received those writes.

CockroachDB

COCKROACHDB (2015)

Distributed relational/SQL DBMS written in Go.

→ Decentralized homogenous shared-nothing architecture using range partitioning.

→ Postgres SQL + wire protocol compatible.

→ Open-source (BSL – MariaDB)

Log-structured on-disk storage.

Pull-based vectorized query processing model.

MVCC + OCC Concurrency Control

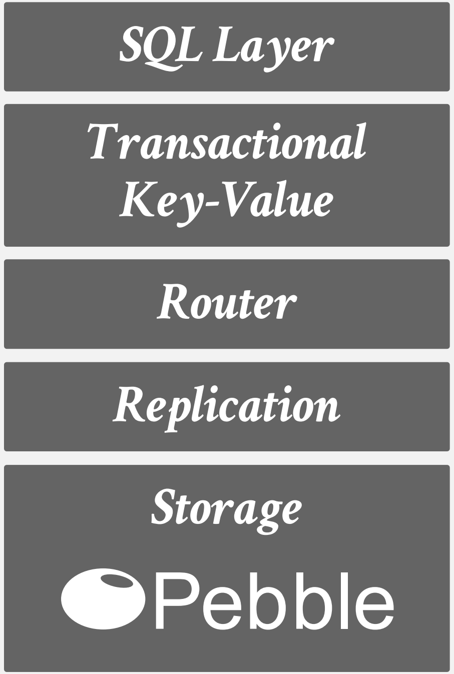
→ All txns run with Serializable isolation level (!!!)

COCKROACHDB – ARCHITECTURE

Multi-layer architecture on top of a replicated key-value store.
→ All tables and indexes are store in a giant sorted map in the k/v store.

Custom Pebble storage manager at each node (previously RocksDB).

Raft protocol (variant of Paxos) for replication and consensus.



COCKROACHDB – CONCURRENCY CONTROL

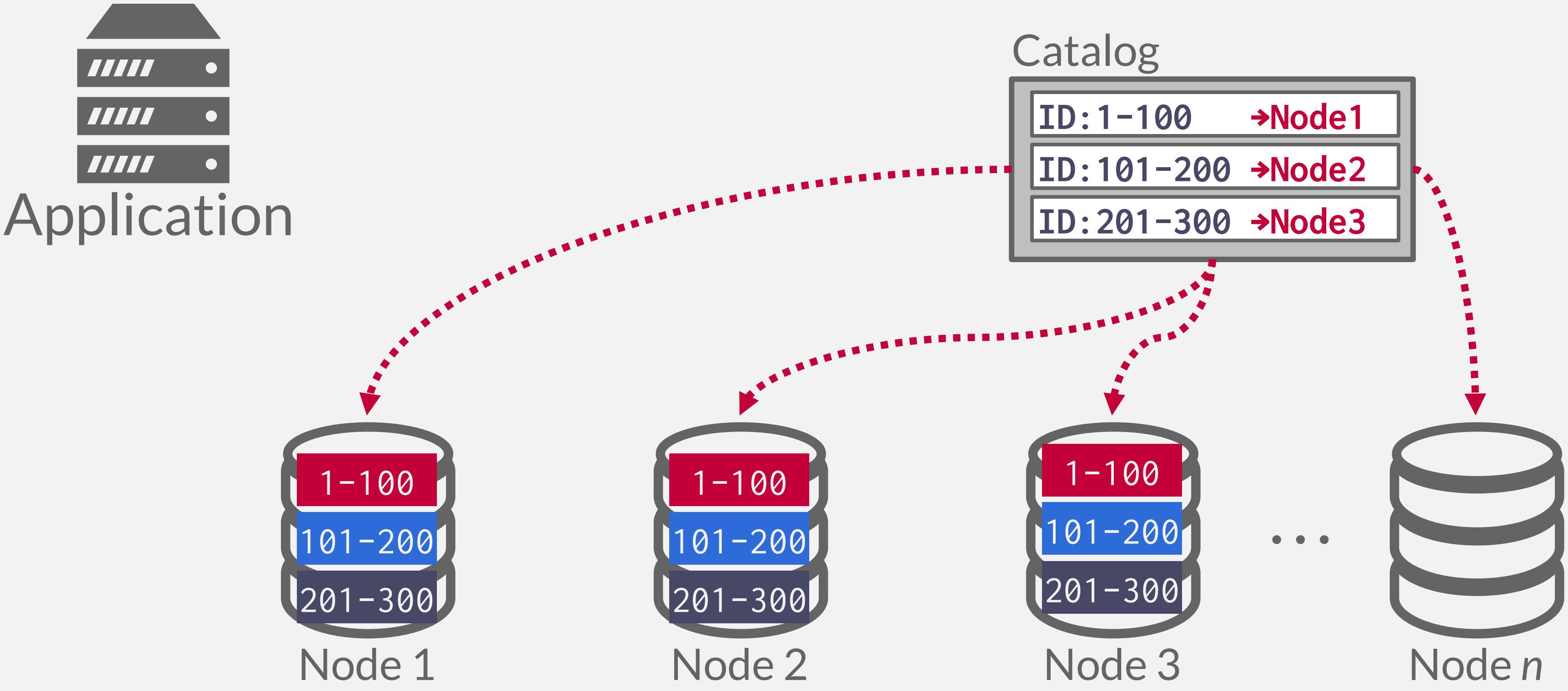
DBMS uses hybrid clocks (physical + logical) to order transactions globally.

→ Synchronized wall clock with local counter.

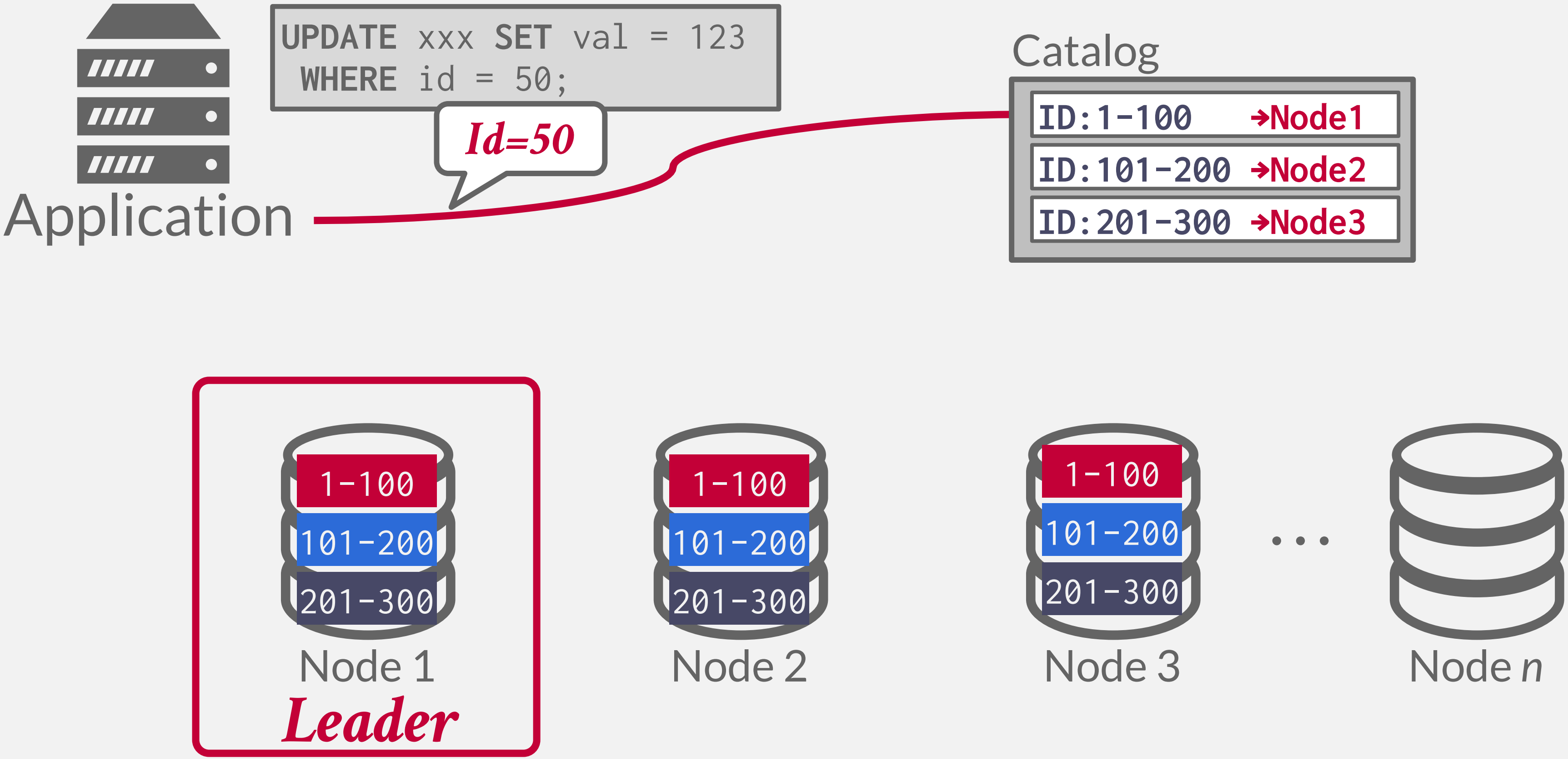
Txns stage writes as "intents" and then checks for conflicts on commit.

All meta-data about txns state resides in the key-value store.

COCKROACHDB – CONCURRENCY CONTROL



COCKROACHDB – CONCURRENCY CONTROL

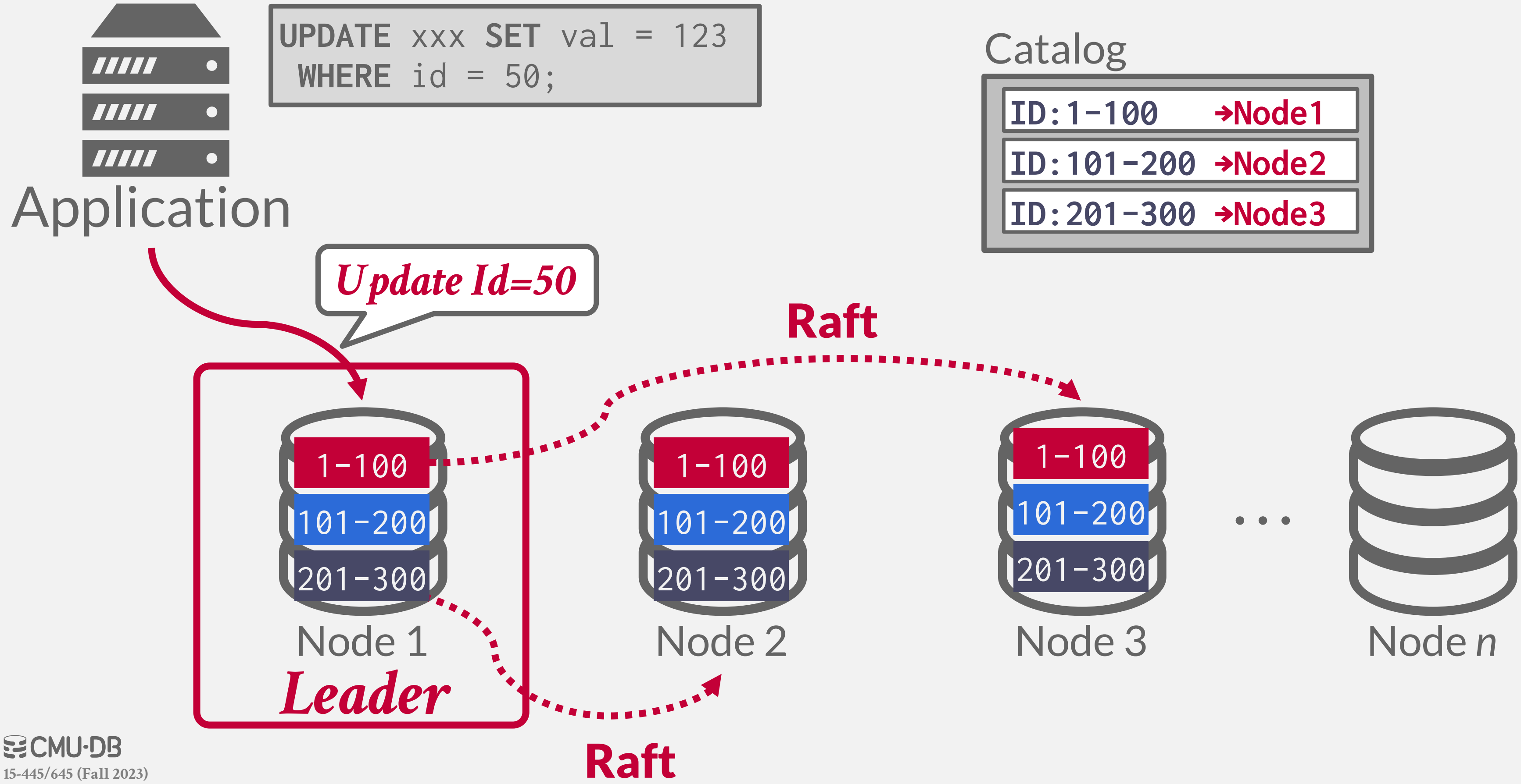


COCKROACHDB – CONCURRENCY CONTROL

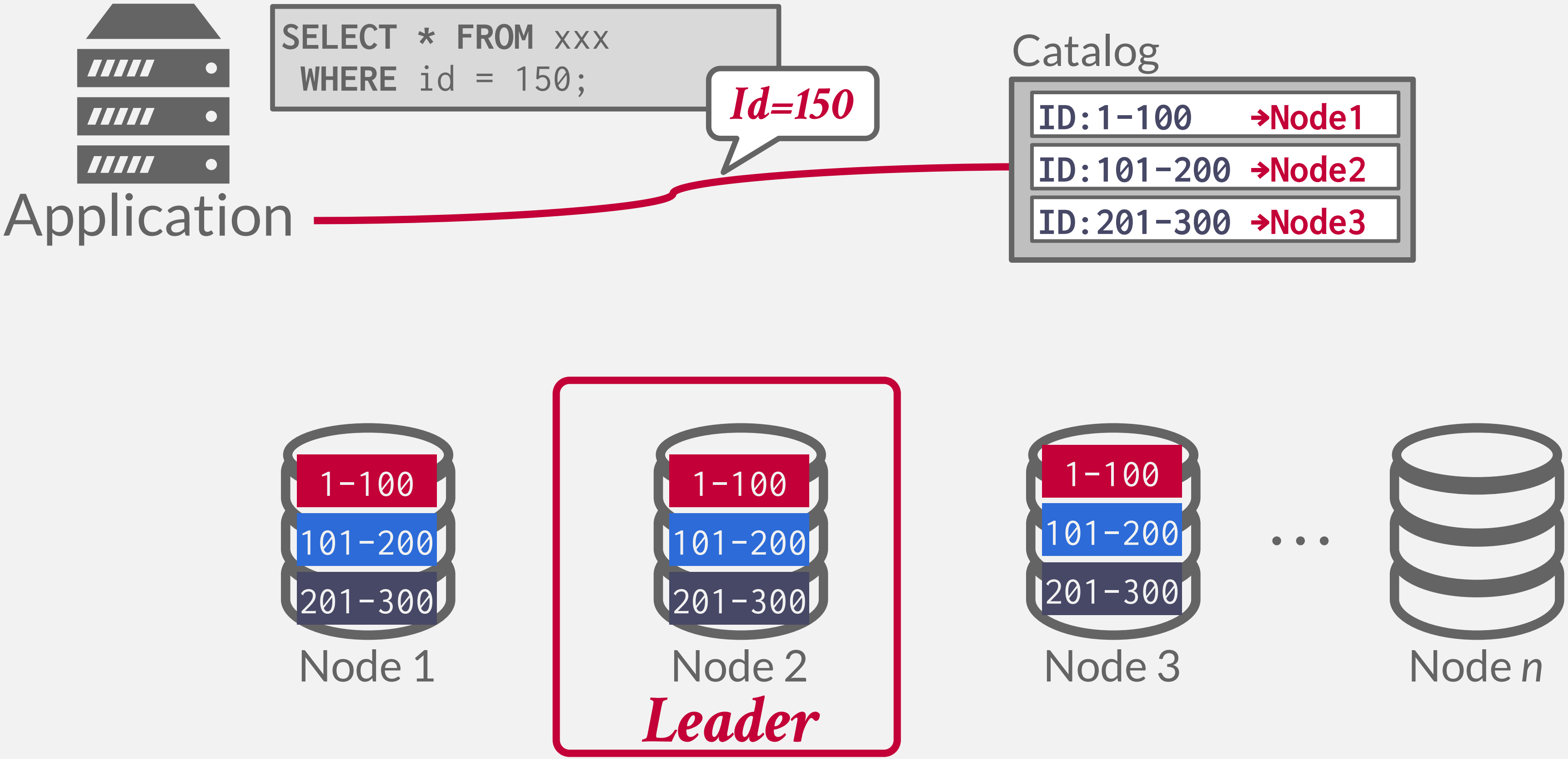
```
UPDATE xxx SET val = 123
 WHERE id = 50;
```

Catalog

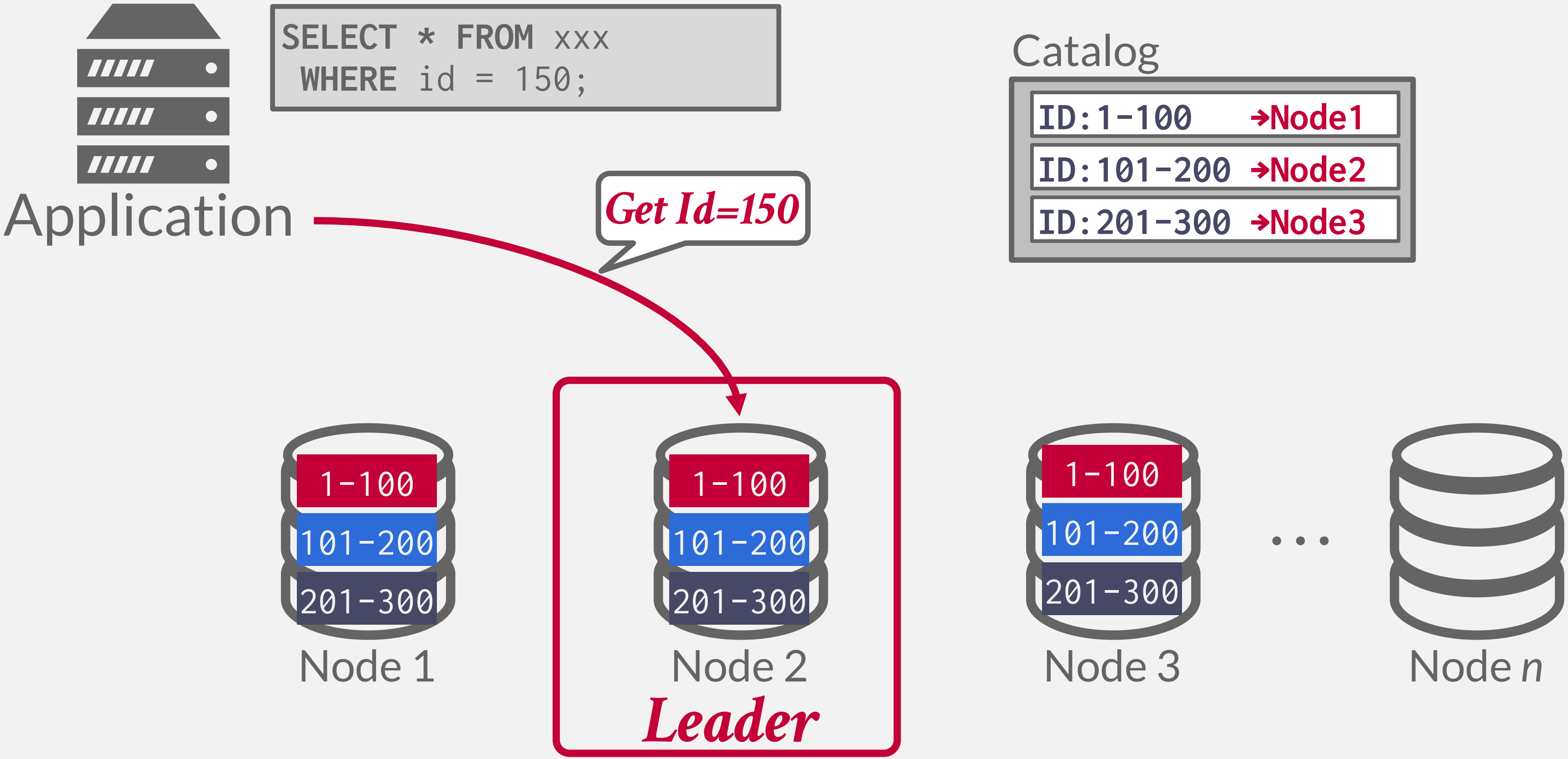
ID:1-100	→Node1
ID:101-200	→Node2
ID:201-300	→Node3



COCKROACHDB – CONCURRENCY CONTROL



COCKROACHDB – CONCURRENCY CONTROL

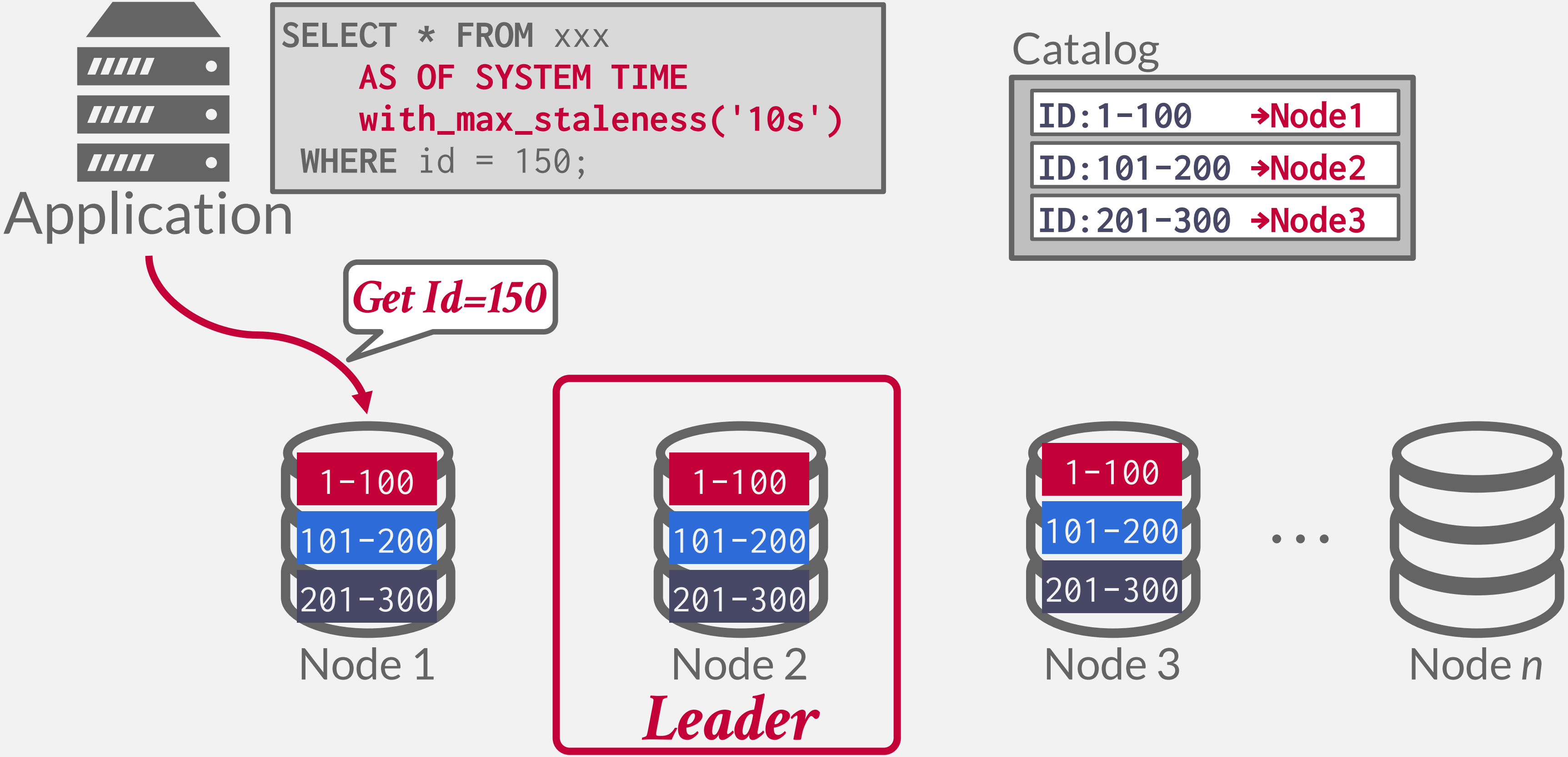


COCKROACHDB – CONCURRENCY CONTROL

```
SELECT * FROM xxx
    AS OF SYSTEM TIME
    with_max_staleness('10s')
 WHERE id = 150;
```

Catalog

ID:1-100 →Node1
ID:101-200 →Node2
ID:201-300 →Node3



snowflake

SNOWFLAKE (2013)

Cloud-native OLAP DBMS written in C++.

Shared-Disk / Disaggregated Storage

Push-based Vectorized Query Processing

Precompiled Operator Primitives

Separate Table Data from Meta-Data

No Buffer Pool

PAX Columnar Storage

SNOWFLAKE – ARCHITECTURE

Data Storage: Cloud-hosted object store
→ Amazon S3, MSFT Azure Store, Google Cloud Storage

Virtual Warehouses: Worker Nodes
→ VM instances running Snowflake software with locally attached disks for caching.
→ Customer specifies the compute capacity.
→ Added support for serverless deployments in 2022 (?).

Cloud Services: Coordinator/Scheduler/Catalog
→ Transactional key-value store (FoundationDB)

snowflake

SNOWFLAKE – EXECUTION ARCHITECTURE

Worker Node (e.g., EC2 Instance)
→ Maintains a local cache of files + columns that previous Worker Processes have retrieved from storage.
→ Simple LRU replacement policy.
→ Optimizer assigns individual table files to worker nodes based on consistent hashing. This ensures that files are only cached in one location.

Worker Process (e.g., Unix Process)
→ Spawned for the duration of a query.
→ Can push intermediate results to other Worker Processes or write to storage.

SNOWFLAKE – QUERY PROCESSING

Snowflake is a push-based vectorized engine that uses precompiled primitives for operator kernels.

→ Pre-compile variants using C++ templates for different vector data types.
→ Only uses codegen (via LLVM) for tuple serialization/deserialization between workers.

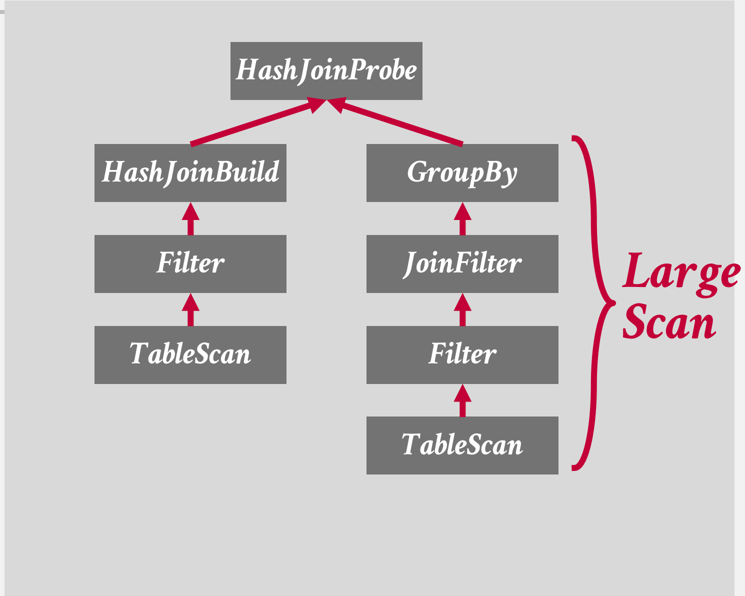
Does not support partial query retries

→ If a worker fails, then the entire query has to restart.

SNOWFLAKE – FLEXIBLE COMPUTE

If a query plan fragment will process a large amount of data, then the DBMS can temporarily deploy additional worker nodes to accelerate its performance.

Flexible compute worker nodes write results to storage as if it was a table.



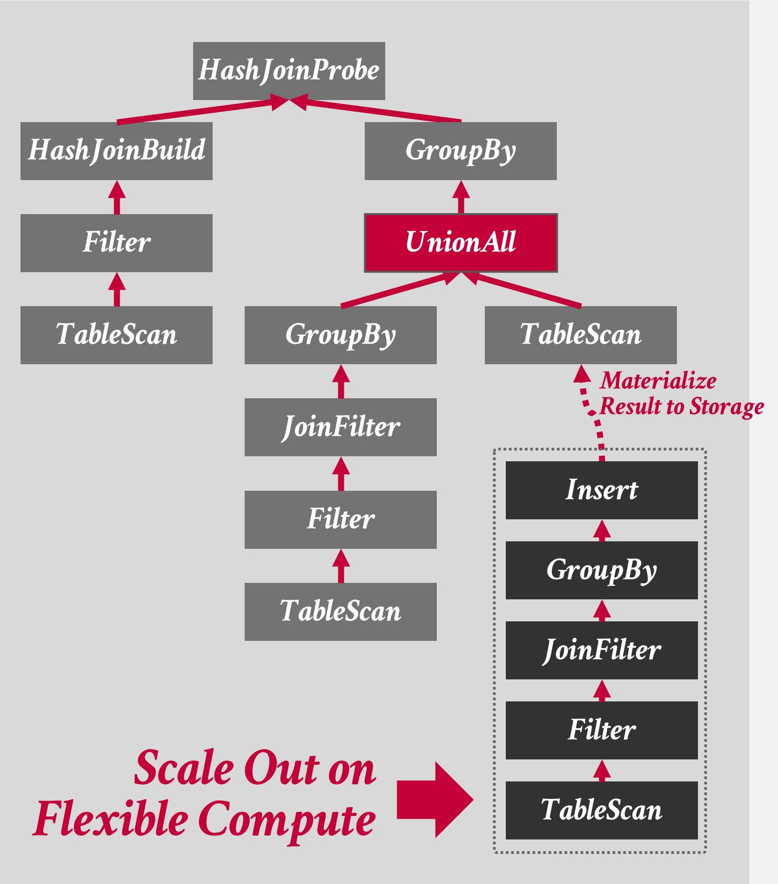
Source: Libo Wang

snowflake

SNOWFLAKE – FLEXIBLE COMPUTE

If a query plan fragment will process a large amount of data, then the DBMS can temporarily deploy additional worker nodes to accelerate its performance.

Flexible compute worker nodes write results to storage as if it was a table.



Source: Libo Wang

mangoDB

MANGODB (2012)

Single-node satirical implementation of MongoDB written in Python.

→ Only supports MongoDB wire protocol v2

→ https://github.com/dcramer/mangodb

All data is written to `/dev/null`

The joke is that original version of MongoDB would send write acknowledgements back to client before writing updates to disk.

mangoDB

MANGODB

Single-node satirical implem
written in Python.

→ Only supports MongoDB wire
→ https://github.com/dcramer/r

All data is written to /dev

The joke is that original vers
would send write acknowled
before writing updates to dis

Files | master | mangodb / server.py | Top

Code | Blame | Raw

```python
def mangodb(socket, address):
    socket.sendall('HELLO\r\n')
    client = socket.makefile()
    output = open(os.devnull, 'w')
    lock = threading.Lock()
    wait = threading.Condition(lock)
    while 1:
        line = client.readline()
        if not line:
            break
        cmd_bits = line.split(' ', 1)
        cmd = cmd_bits[0]
        if cmd == 'BYE':
            break
        if cmd == 'WAIT':
            wait.wait()
            continue
        if len(cmd_bits) > 1:
            lock.acquire(True)
            output.write(cmd_bits[1])
            if MANGODB_DURABLE:
                output.flush()
                os.fsync(output.fileno())
            data = '42' if MANGODB_EVENTUAL else \
                os.urandom(1024).encode('string-escape')
            lock.release()
            client.write('OK' + data + '\r\n')
        client.flush()
```

mangoDB

MANGODB

Single-node satirical implem
written in Python.
→ Only supports MongoDB wire
→ https://github.com/dcramer/r

All data is written to /dev

The joke is that original vers
would send write acknowle
before writing updates to dis

Files | master | mangodb / server.py | Top

Code | Blame | Raw

```python

def mangodb(socket, address):
    socket.sendall('HELLO\r\n')
    client = socket.makefile()
    output = open(os.devnull, 'w')
    lock = threading.Lock()
    wait = threading.Condition(lock)
    while 1:
        line = client.readline()
        if not line:
            break
        cmd_bits = line.split(' ', 1)
        cmd = cmd_bits[0]
        if cmd == 'BYE':
            break
        if cmd == 'WAIT':
            wait.wait()
            continue
        if len(cmd_bits) > 1:
            lock.acquire(True)
            output.write(cmd_bits[1])
            if MANGODB_DURABLE:
                output.flush()
                os.fsync(output.fileno())
            data = '42' if MANGODB_EVENTUAL else \
                os.urandom(1024).encode('string-escape')
            lock.release()
            client.write('OK' + data + '\r\n')
        client.flush()
```

mangoDB

MANGODB

Single-node satirical implem
written in Python.
→ Only supports MongoDB wire
→ https://github.com/dcramer/r

All data is written to /dev

The joke is that original vers
would send write acknowled
before writing updates to d

Files master mangodb / server.py Top

Code Blame Raw

```python
def mangodb(socket, address):
    socket.sendall('HELLO\r\n')
    client = socket.makefile()
    output = open(os.devnull, 'w')
    lock = threading.Lock()
    wait = threading.Condition(lock)
    while 1:
        line = client.readline()
        if not line:
            break
        cmd_bits = line.split(' ', 1)
        cmd = cmd_bits[0]
        if cmd == 'BYE':
            break
        if cmd == 'WAIT':
            wait.wait()
            continue
        if len(cmd_bits) > 1:
            lock.acquire(True)
            output.write(cmd_bits[1])
            if MANGODB_DURABLE:
                output.flush()
                os.fsync(output.fileno())
            data = '42' if MANGODB_EVENTUAL else \
                os.urandom(1024).encode('string-escape')
            lock.release()
            client.write('OK' + data + '\r\n')
        client.flush()
```

mangoDB

MANGODB

Single-node satirical implem
written in Python.
→ Only supports MongoDB wire
→ https://github.com/dcramer/r

All data is written to /dev

The joke is that original vers
would send write acknowled
before writing updates to dis

Files master mangodb / server.py Top

Code Blame Raw

```python

def mangodb(socket, address):
    socket.sendall('HELLO\r\n')
    client = socket.makefile()
    output = open(os.devnull, 'w')
    lock = threading.Lock()
    wait = threading.Condition(lock)
    while 1:
        line = client.readline()
        if not line:
            break
        cmd_bits = line.split(' ', 1)
        cmd = cmd_bits[0]
        if cmd == 'BYE':
            break
        if cmd == 'WAIT':
            wait.wait()
            continue
        if len(cmd_bits) > 1:
            lock.acquire(True)
            output.write(cmd_bits[1])
            if MANGODB_DURABLE:
                output.flush()
                os.fsync(output.fileno())
            data = '42' if MANGODB_EVENTUAL else \
                os.urandom(1024).encode('string-escape')
            lock.release()
            client.write('OK' + data + '\r\n')
        client.flush()
```

TDB

TABDB (2019)

TabDB is a relational DBMS that stores data in your browser's tab title fields.

It uses Emscripten to convert SQLite's C code into JavaScript.

It then splits the SQLite database file into strings and stores them in your browser tabs.

https://tabdb.io/

CONCLUDING REMARKS

Where does the name "BusTub" come from?

Why is the relational model superior?

Why do tech companies sell multiple DBMSs?

CONCLUDING REMARKS

Databases are awesome.
→ They cover all facets of computer science.
→ We have barely scratched the surface…

Going forth, you should now have a good understanding how these systems work.

This will allow you to make informed decisions throughout your entire career.
→ Avoid premature optimizations.