Introduction to Database Systems IDBS - Spring 2024

Lecture 11 - Scale & NoSQL

Scaling Up/Out
NoSQL
Eventual Consistency
CAP Theorem

Readings: PDBM 11

Omar Shahbaz Khan

General Info

HOMEWORK 4 - OUT NOW!

- Deadline: December 2, 2024 23:59
- Remember: 3/4 Homework Assignments need to be passed to be eligible for the exam!

FEEDBACK FOR HOMEWORK 3

- Will be out ASAP

Last Time in IDBS…

-- TODO -> DONE

- ✓ Course Evaluation!

- ✓ Transactions
  - ✓ ACID Properties
  - ✓ Buffer Management
  - ✓ Logging
  - ✓ Recovery
  - ✓ Locking

Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels

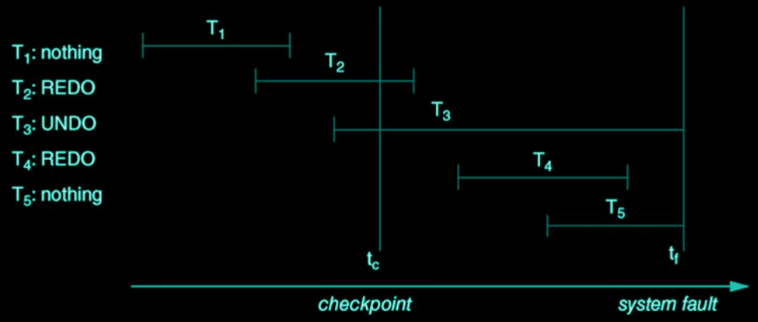
Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels

- Write-Ahead-Logging (WAL)
  1. Before any changes are written to disk we force the corresponding log record to disk

     Is this necessary for MMDBs?

  2. Before a transaction is committed we force all log records for the transaction to disk

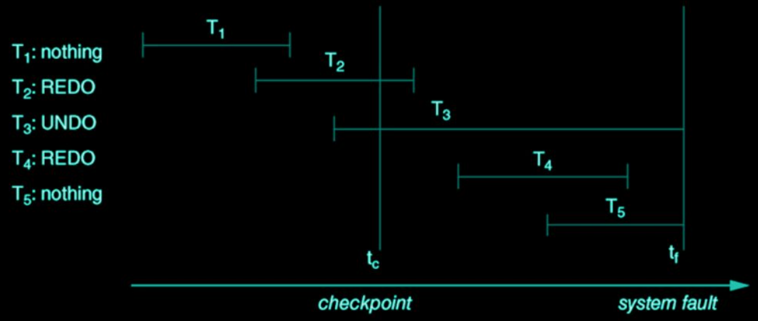


Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels

- Recovery
  1. Analyze Log
  2. REDO Committed Transactions
  3. UNDO Uncommitted Transactions

Is this necessary for MMDBs?



Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels

- Constraints
  - Primary Key, Foreign Key
  - NOT NULL, UNIQUE, CHECK[1]
  - IDENTITY[2], SERIAL
  - In DDL
- Triggers
  - Functions
  - Complex Requirements
  - Beyond DDL

1: https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-check-constraint/
2: https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-identity-column/

Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels
- Serializability
- Two-Phase Locking
  - Growing Phase: Acquire Locks
  - Shrinking Phase: Release Locks
  - S2PL: Release only Shared Locks during shrinking phase
  - R2PL: Release locks at COMMIT/ABORT/ROLLBACK
- Deadlocks
  - Detected through wait-for-graphs
  - Resolved by picking a victim aka. aborting one of the transactions
- Phantoms

Transactions

- Atomicity & Durability
  - Logging
  - Recovery
- Consistency
  - Constraints
  - Triggers
- Isolation
  - Locking
  - Isolation Levels

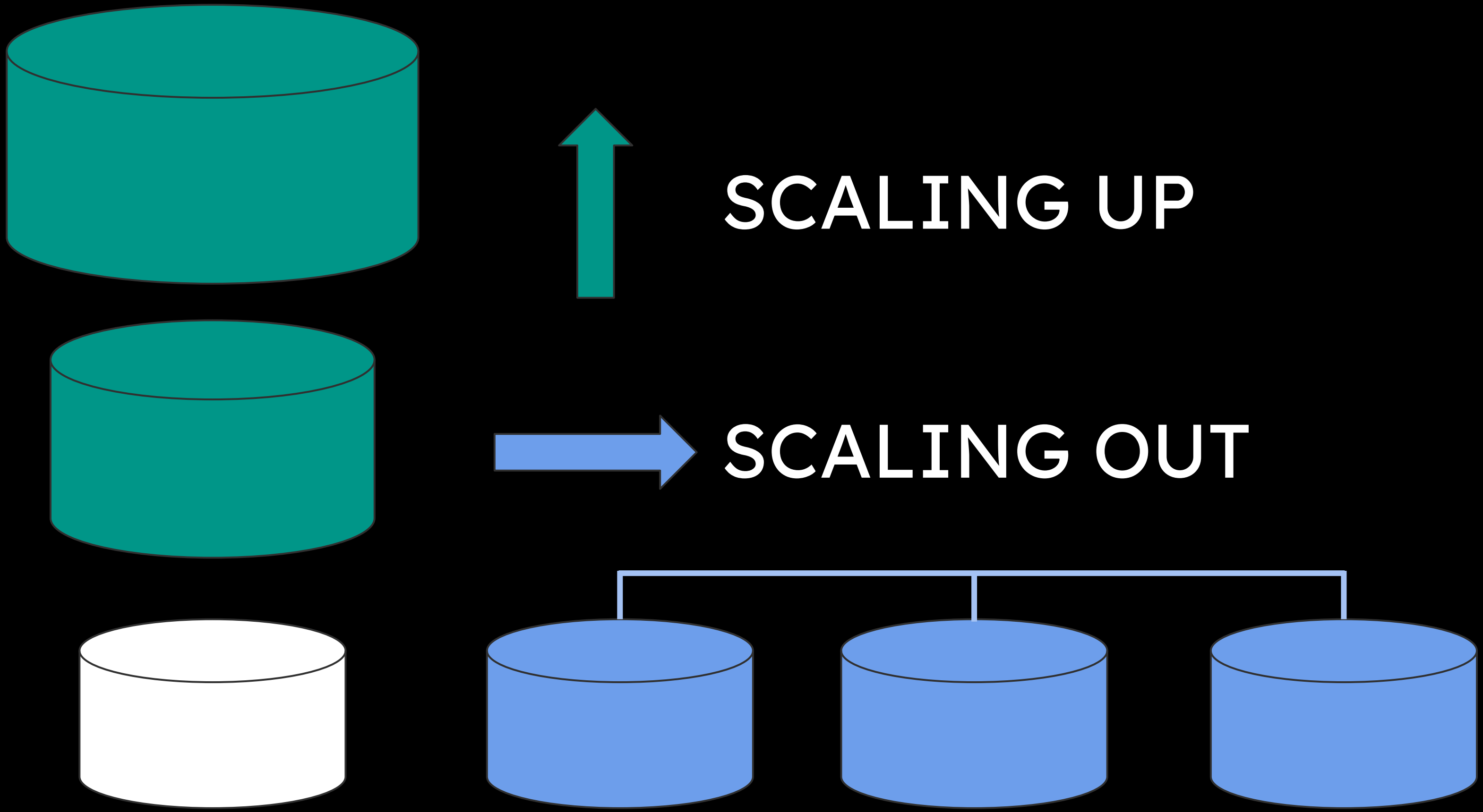
- Isolation Levels
  - Read Uncommitted (RU)
  - Read Committed (RC)
  - Repeatable Read (RR)
  - Serializable (S)
  - Snapshots (Si)

This Time…

-- TODO

- Scaling Up/Out

- NoSQL
  - Data Model
  - Distributed Architecture
  - Consistency/Availability trade-off
    - Eventual Consistency
    - CAP Theorem


SCALING UP
SCALING OUT

The End of an Architectural Era (It's Time for a Complete Rewrite)

Michael Stonebraker
Samuel Madden
Daniel J. Abadi
Stavros Harizopoulos
MIT CSAIL
{stonebraker, madden, dna, stavros}@csail.mit.edu

Nabil Hachem
AvantGarde Consulting, LLC
nhachem@agdba.com

Pat Helland
Microsoft Corporation
phelland@microsoft.com

ABSTRACT

In previous papers [SC05, SBC+07], some of us predicted the end of "one size fits all" as a commercial relational DBMS paradigm. These papers presented reasons and experimental evidence that showed that the major RDBMS vendors can be outperformed by 1-2 orders of magnitude by specialized engines in the data warehouse, stream processing, text, and scientific database markets.

Assuming that specialized engines dominate these markets over time, the current relational DBMS code lines will be left with the business data processing (OLTP) market and hybrid markets where more than one kind of capability is required. In this paper we show that current RDBMSs can be beaten by nearly two orders of magnitude in the OLTP market as well. The experimental evidence comes from comparing a new OLTP prototype, H-Store, which we have built at M.I.T., to a popular RDBMS on the standard transactional benchmark, TPC-C.

We conclude that the current RDBMS code lines, while attempting to be a "one size fits all" solution, in fact, excel at nothing. Hence, they are 25 year old legacy code lines that should be retired in favor of a collection of "from scratch" specialized engines. The DBMS vendors (and the research community) should start with a clean sheet of paper and design systems for tomorrow's requirements, not continue to push code lines and architectures designed for yesterday's needs.

All three systems were architected more than 25 years ago, when hardware characteristics were much different than today. Processors are thousands of times faster and memories are thousands of times larger. Disk volumes have increased enormously, making it possible to keep essentially everything, if one chooses to. However, the bandwidth between disk and main memory has increased much more slowly. One would expect this relentless pace of technology to have changed the architecture of database systems dramatically over the last quarter of a century, but surprisingly the architecture of most DBMSs is essentially identical to that of System R.

Moreover, at the time relational DBMSs were conceived, there was only a single DBMS market, business data processing. In the last 25 years, a number of other markets have evolved, including data warehouses, text management, and stream processing. These markets have very different requirements than business data processing.

Lastly, the main user interface device at the time RDBMSs were architected was the dumb terminal, and vendors imagined operators inputting queries through an interactive terminal prompt. Now it is a powerful personal computer connected to the World Wide Web. Web sites that use OLTP DBMSs rarely run interactive transactions or present users with direct SQL interfaces.

In summary, the current RDBMSs were architected for the business data processing market in a time of different user

Major Trends

- More hardware
  - Better servers
  - More servers ➔ Cloud
- More data
  - More quantity of data
  - More types of data
  - Still want fast systems
  - Less structure ➔ less need for complexity

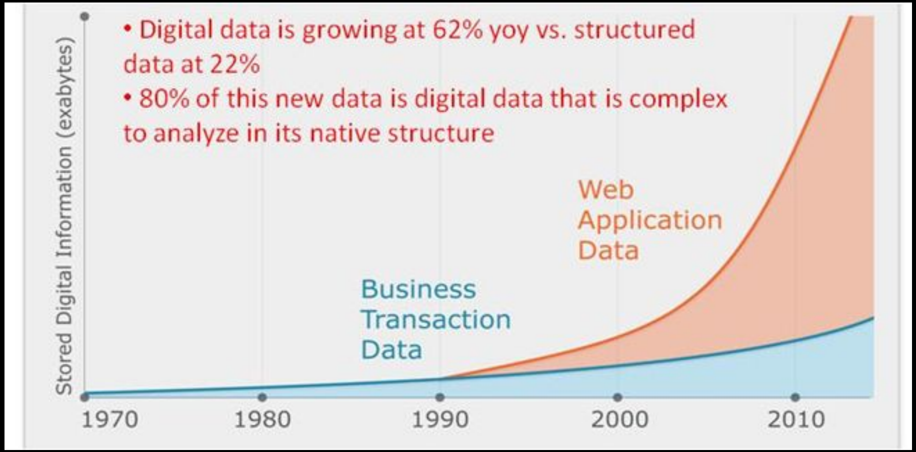


Image source: https://practicalanalytics.wordpress.com/2011/11/06/explaining-hadoop-to-management-whats-the-big-data-deal/

Scaling UP

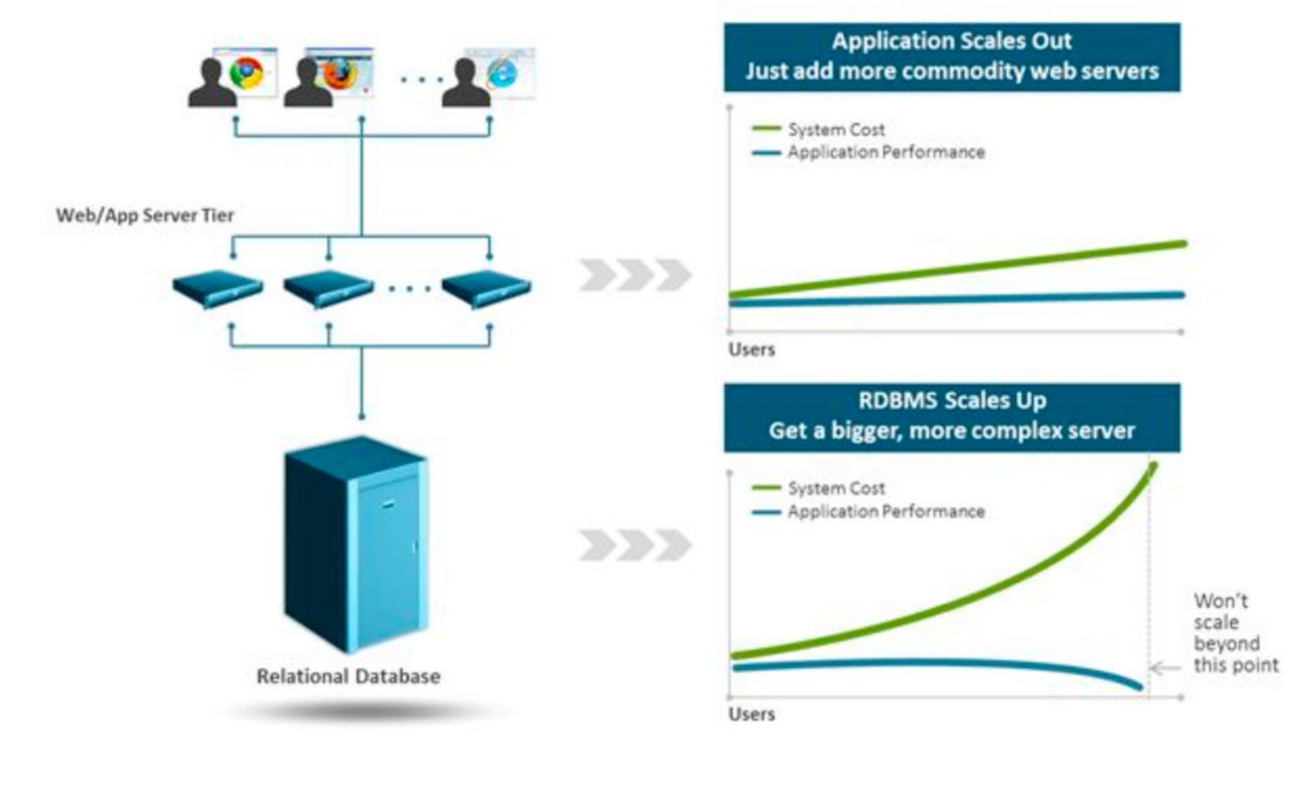


Image source: https://sungsoo.github.io/2014/07/08/nosql-background.html

Scaling OUT

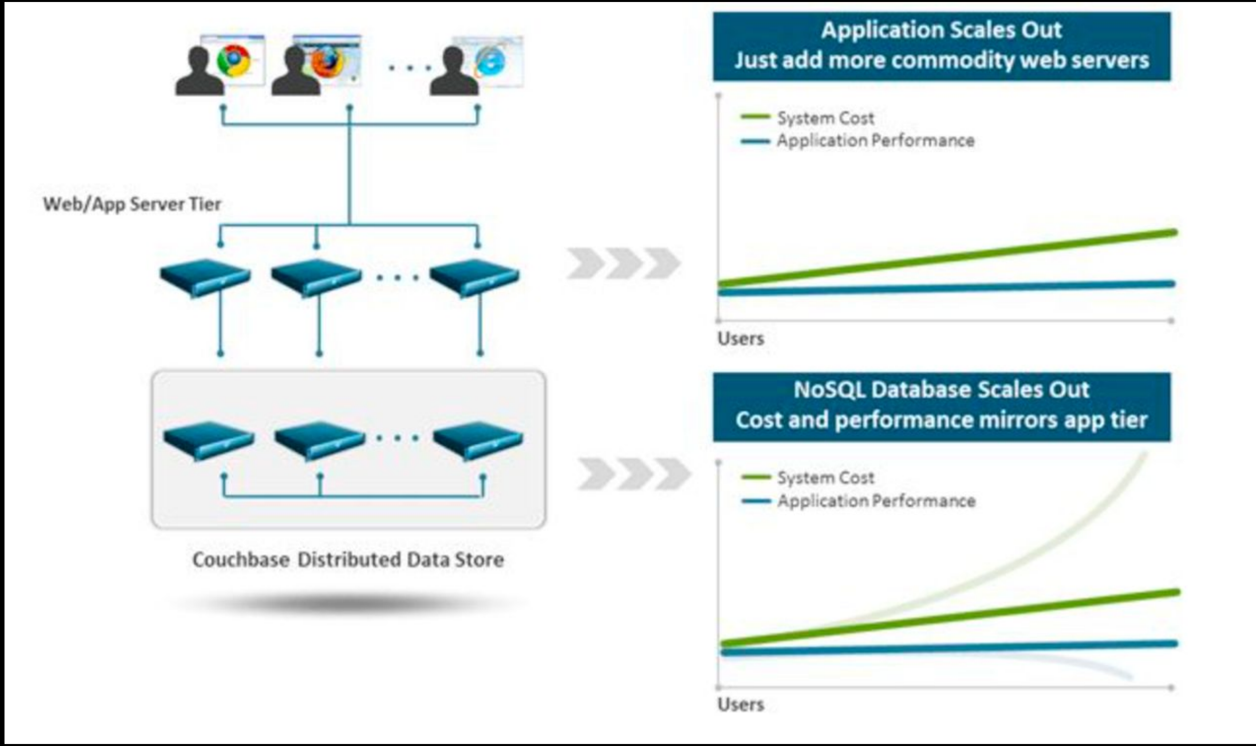
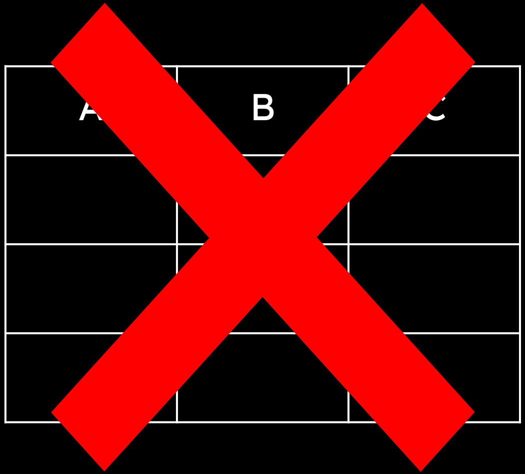
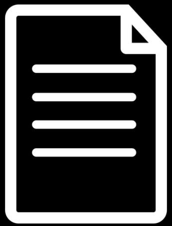
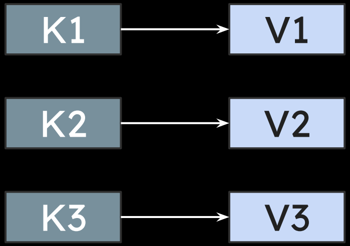
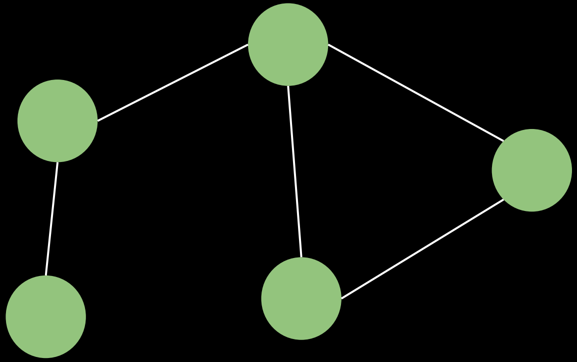


Image source: https://sungsoo.github.io/2014/07/08/nosql-background.html


B




K1
V1
K2
V2
K3
V3



NoSQL in a Nutshell

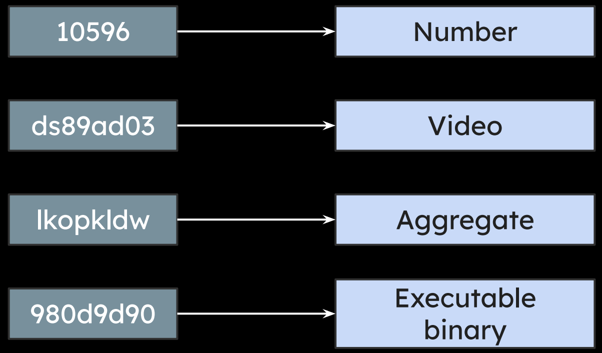
- Data model
  - Not relational
  - No formally described schema
- Interface
  - Not only SQL (→ NoSQL name)
  - Proprietary, REST, etc.
- Architecture
  - Usually distributed
- Mostly not ACID compliant
  - Consistency/Availability trade-off (CAP theorem)
- Mostly open source

Key-Value Stores

- Associative Array
  - Unique key points to a value
  - Value contents unknown

- Can not be queried
  - GET / PUT only
  - Value can be an aggregate structure

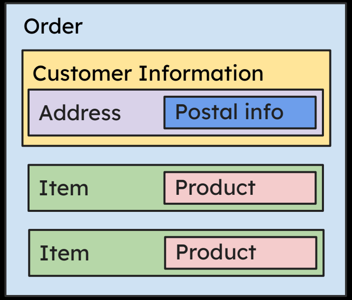
- Examples
  - Riak, Apache Cassandra, Redis

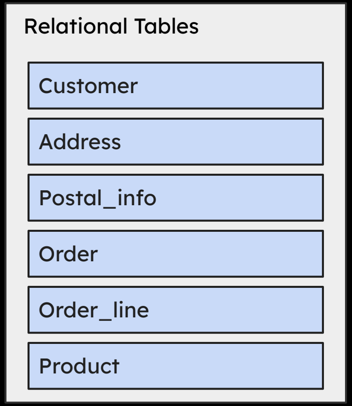


Document Stores

- Each value is a document
  - Most often JSON
  - Unique keys used for retrieval
- You can query into the document
  - More transparent than key-value stores
- The document is an aggregate structure
- Examples: MongoDB, Couchbase, Amazon DocumentDB

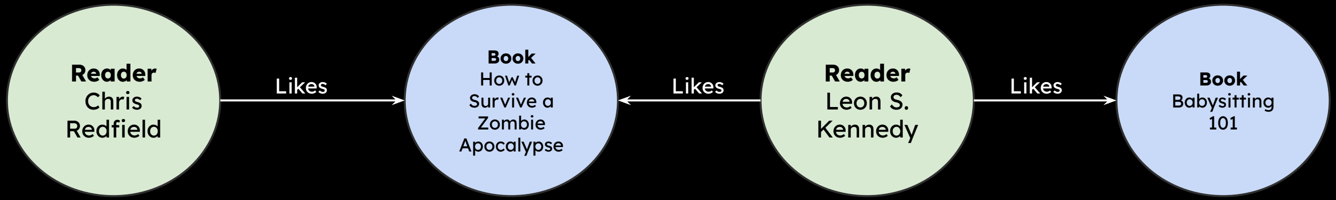
```
{
    "order_id": 3294,
    "customer": {
        "ssn": "123456789",
        "name": "John Doe",
        "address": "123 Random Street",
        "postal_code": "98765"
    },
    "line-items": [
        {"product": "Sofa", "price": 2500},
        {"product": "Table", "price": 1000},
    ]
}
```





Graph Stores

- Nodes = Entities
- Edges = Relationships, directional
- Properties = Entity descriptors
- Examples – neo4j, Allegro, InfiniGraph, OrientDB



-- TODO

✓ Scaling Up/Out

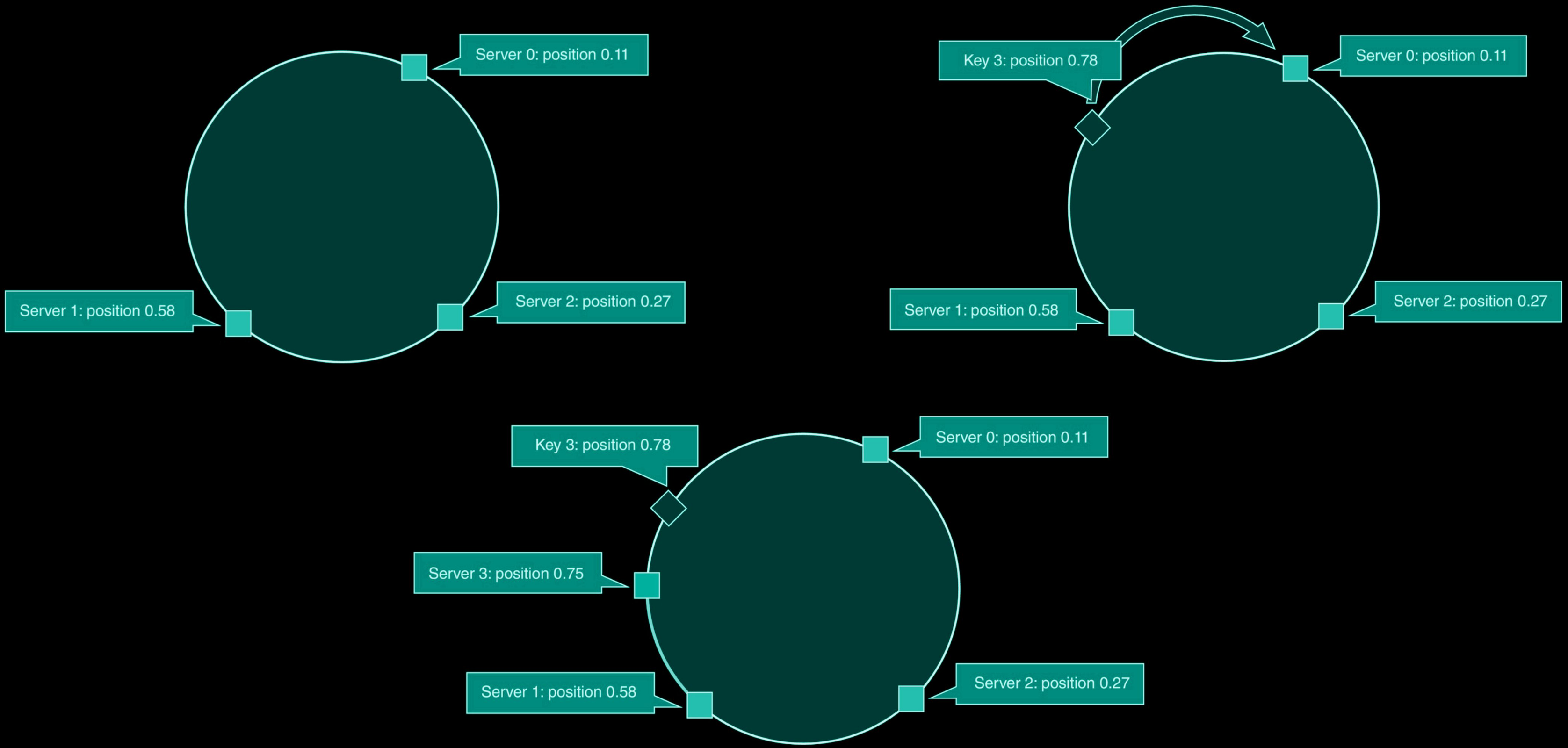
- NoSQL
  - ✓ Data Model
  - Distributed Architecture
  - Consistency/Availability trade-off
    - Eventual Consistency
    - CAP Theorem

Goals of a Distributed Storage

- Workload sharing – balance
- Redundancy (replicas) – failure handling

What does a "cloud" mean in this context?

Consistent Hashing: Balance

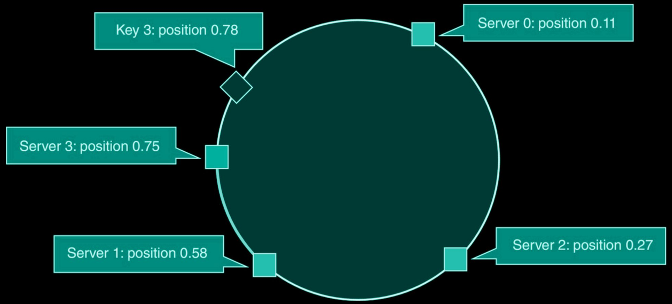


Consistent Hashing: Problems

- Random assignments can lead to skewed distributions

- Does not reflect a server's capabilities
  - Not all servers have the same hardware

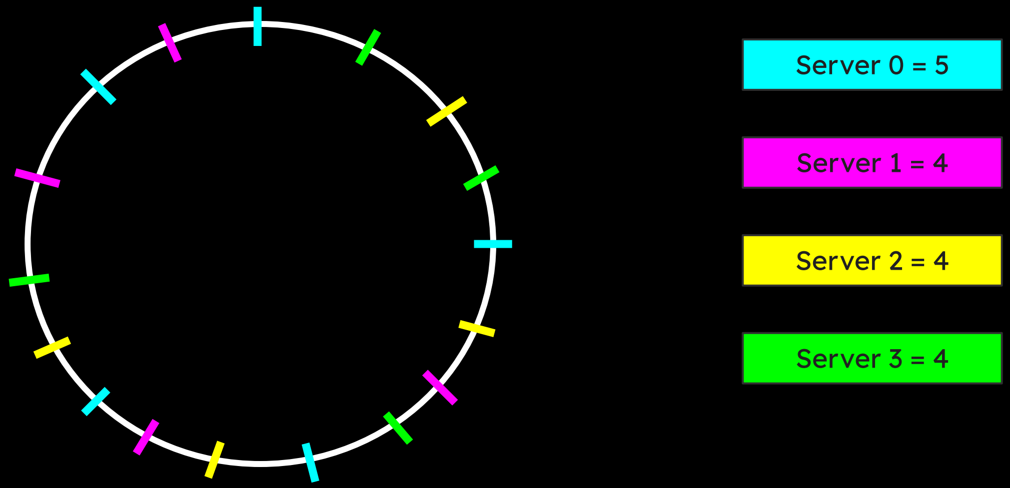
- Adding new server only takes high load from 1 server



Consistent Hashing: Virtual Servers

- 1 Server = multiple smaller (virtual) servers

- Random assignments of virtual servers



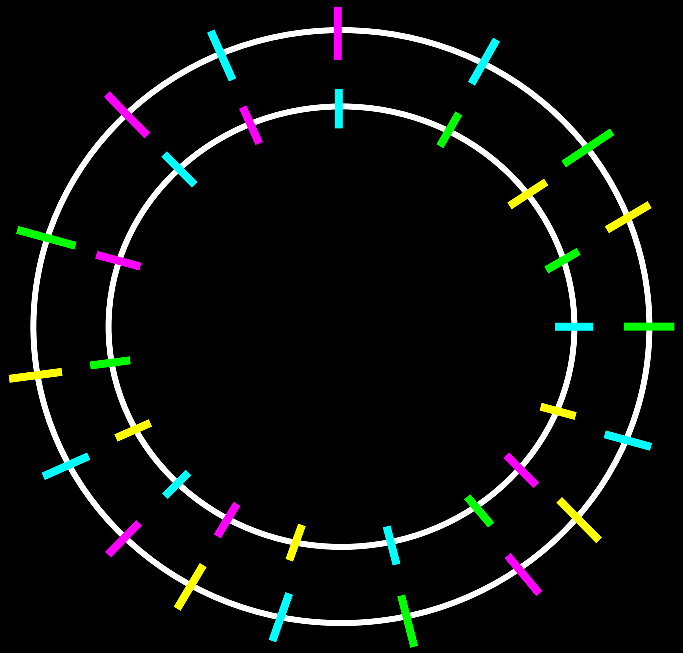
Consistent Hashing: Redundancy

Server 0

Server 1

Server 2

Server 3



-- TODO

✓ Scaling Up/Out

- NoSQL
  ✓ Data Model
  ✓ Distributed Architecture
  - Consistency/Availability trade-off
    - Eventual Consistency
    - CAP Theorem

Replica Consistency

- Sequential (or strong) consistency: All updates are seen by all processes in the same order. As a result, the effects of an update are seen by all observers. There is no inconsistency.
  - Roughly the same as Isolation in ACID

- Weak consistency: Observers might see inconsistencies among replicas

- Eventual consistency: A form of weak consistency, where at some point, in case there is no failure, all replicas will reflect the last update.

Tunable Consistency

- Not a binary system
  - N replicas, R read quorum, W write quorum
- $R = W = 1$ gives eventual consistency
- $R + W > N$ gives strong consistency
- Example:
  If the replication factor is 3, then the consistency level of the reads and writes combined must be at least 4 to achieve strong consistency.
  As in, read operations use 2 out of 3 replicas to verify the value and write operations use 2 out of 3 replicas to verify the value.

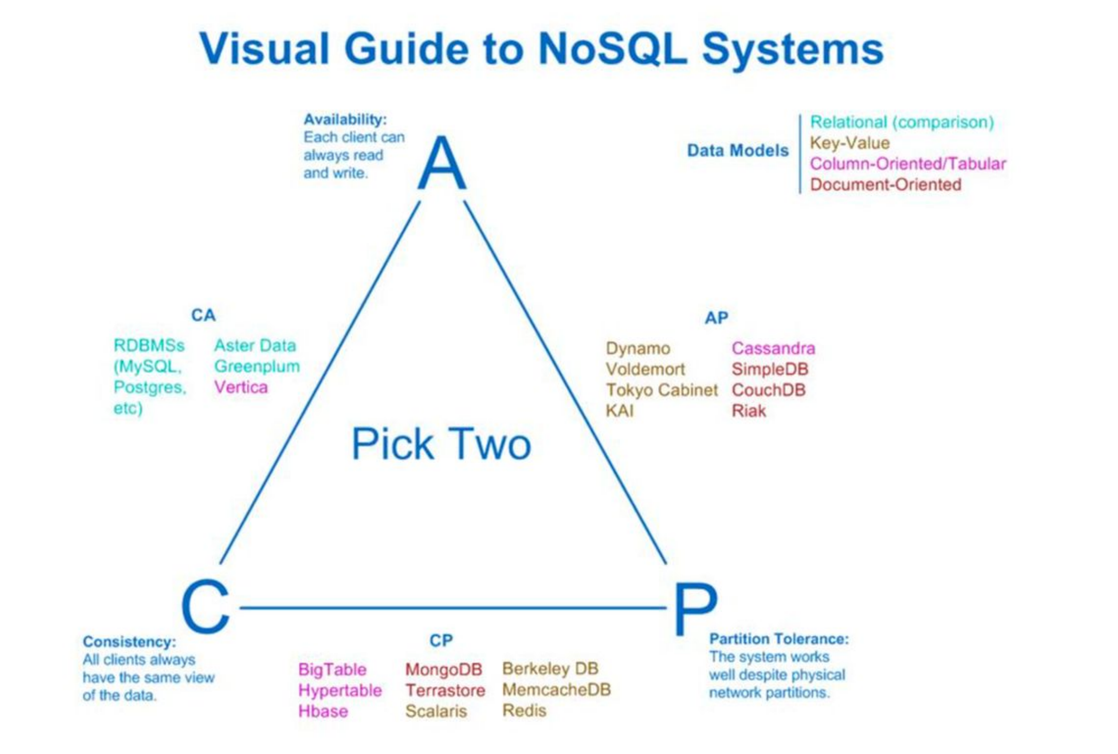
CAP Theorem

- C = Consistency
  - Readers read the most recent update
- A = Availability
  - A valid answer is returned, even if one or more nodes are down
- P = Partition Tolerance
  - Partition: The network becomes disconnected
  - A distributed system works despite the network failure

Incorrect (but Typical) Formulation

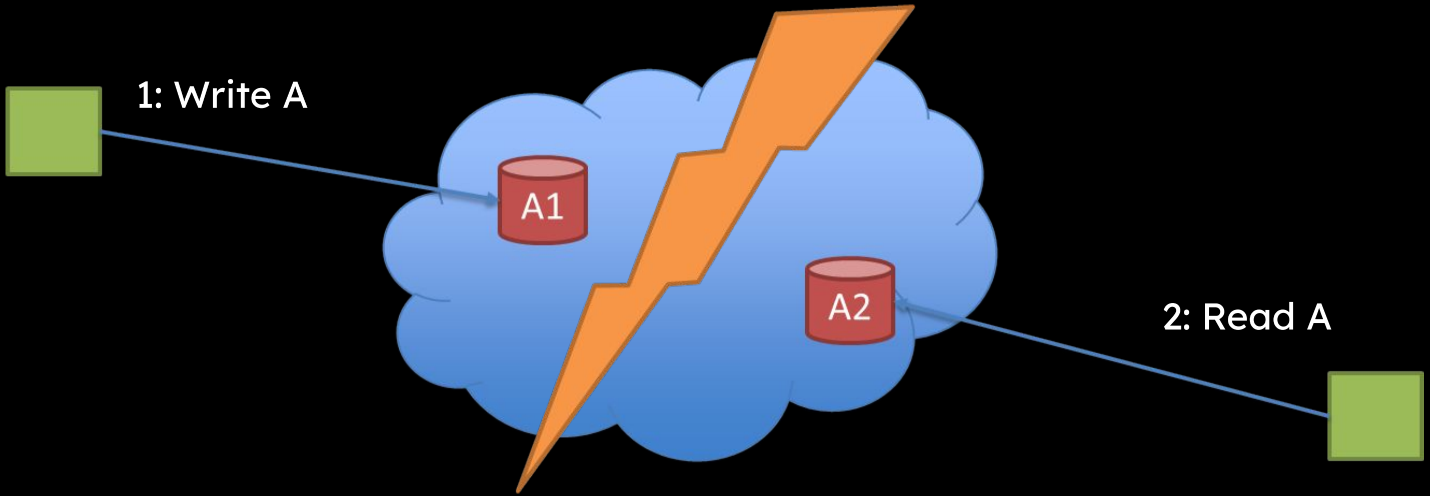
- You can only get **two** of **Consistency**, **Availability**, and **Partition Tolerance**

Resulting Classification



Correct (but Useless) Formulation

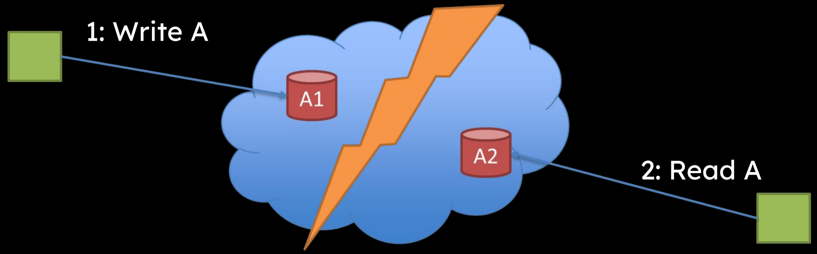
- In a **partitioned** network, choose between **Consistency** and **Availability**
- Proof: Simple thought experiment



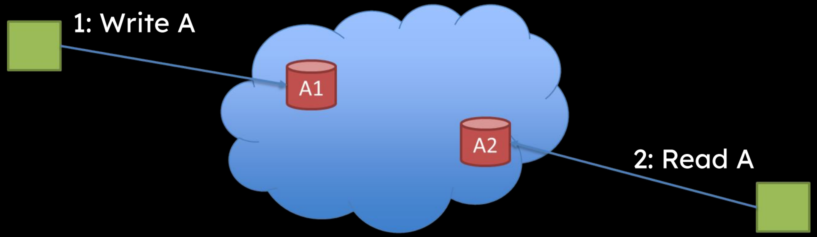
Seth Gilbert and Nancy Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, Volume 33 Issue 2 (2002)

PACELC (useful!) Formulation

- In a **partitioned** network, choose between **Availability** and **Consistency**



- **Else** (regular operation), choose between **Latency** and **Consistency**



Daniel Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story", IEEE Computer 45(2): 37-42 (2012)

2012: Google Spanner

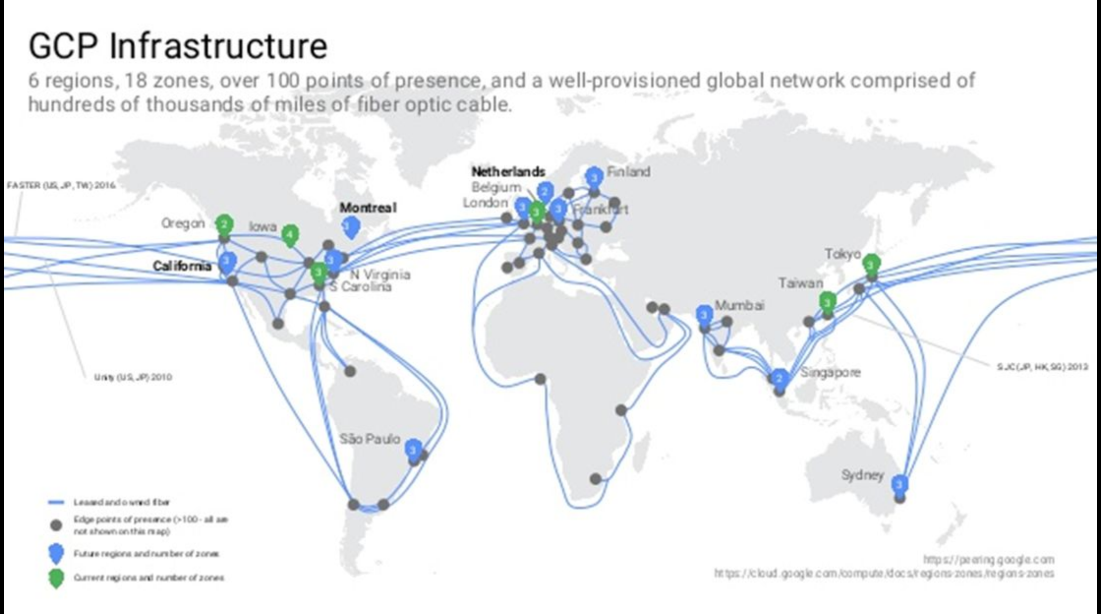
Cloud Spanner: The best of the relational and non-relational worlds

	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ Yes	✓ Yes	✗ No
SQL	✓ Yes	✓ Yes	✗ No
Consistency	✓ Strong	✓ Strong	✗ Eventual
Availability	✓ High	✗ Failover	✓ High
Scalability	✓ Horizontal	✗ Vertical	✓ Horizontal
Replication	✓ Automatic	Configurable	Configurable

- Does it break the CAP theorem?

https://cloud.google.com/spanner/

What is High Availability?



- “Cloud Spanner […] serves data with low latency while maintaining transactional consistency and industry-leading 99.999% (five 9s) availability - 10x less downtime than four nines (<5 minutes per year).”

https://cloud.google.com/spanner/

Image source: https://www.slideshare.net/MichelleHolley1/google-cloud-networking-deep-dive

2012: Google Spanner

Cloud Spanner: The best of the relational and non-relational worlds

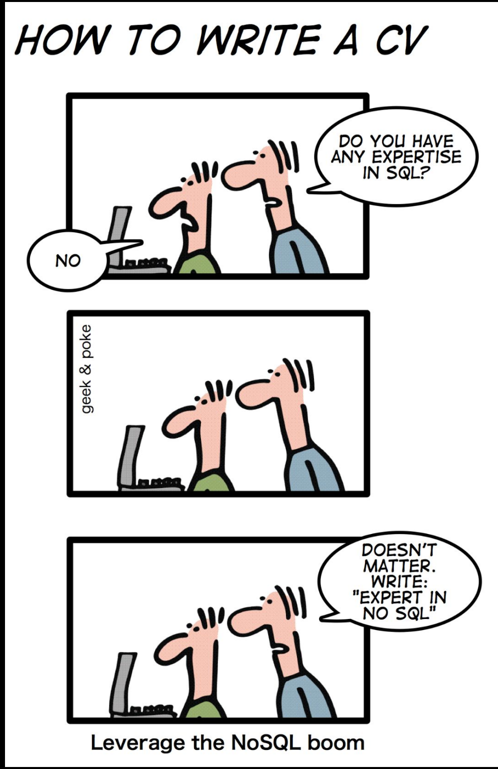
	CLOUD SPANNER	TRADITIONAL RELATIONAL	TRADITIONAL NON-RELATIONAL
Schema	✓ **Yes**	✓ Yes	✗ No
SQL	✓ **Yes**	✓ Yes	✗ No
Consistency	✓ **Strong**	✓ Strong	✗ Eventual
Availability	✓ **High**	✗ Failover	✓ High
Scalability	✓ **Horizontal**	✗ Vertical	✓ Horizontal
Replication	✓ **Automatic**	Configurable	Configurable

- Does it break the CAP theorem? NO!

https://cloud.google.com/spanner/

NoSQL

- A result of two trends:
  - More hardware
  - More data
- Data model
  - Not relational
  - No formally described schema
- Architecture
  - Usually distributed
- Mostly not ACID compliant
  - CAP = Consistency/Availability trade-off



-- TODO

- ✓ Scaling Up/Out
- ✓ NoSQL
  - ✓ Data Model
  - ✓ Distributed Architecture
  - ✓ Consistency/Availability trade-off
    - ✓ Eventual Consistency
    - ✓ CAP Theorem

Takeaways

Why do NoSQL devs eat alone?

…

They don’t know how to join tables.

Scale Up vs <u>Scale Out</u>

NoSQL

- Non-relational Data Models
- Horizontal Scaling
- MongoDB, Redis, DynamoDB, Neo4j, etc.

CAP Theorem

- Consistency
- Availability
- Partitions
- Pick between Consistency and Availability in case of network partitions

Next Time in IDBS…

Introduction to Database Systems IDBS - Autumn 2024

Lecture 12 - Big Data

Big Data Analytics
Distributed Computing Frameworks

Readings: PDBM 19.1-19.2, 19.4, 20.1-20.3

Martin Hentschel