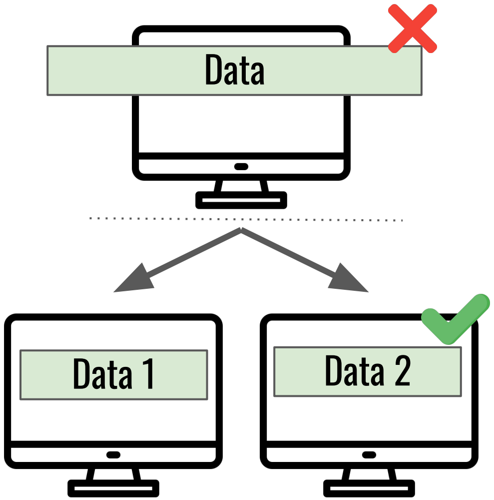
# Part 1: RDBMS and NOSQL

Valerie Hayot-Sasson

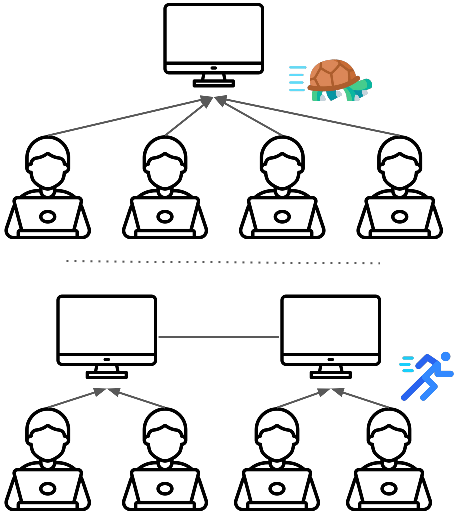
# Why distribute data ?

## Size



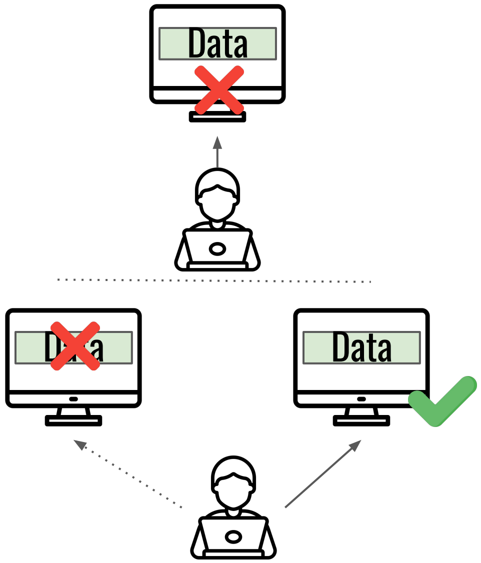

Data too large for a single node

## Performance



Satisfy the needs of many clients

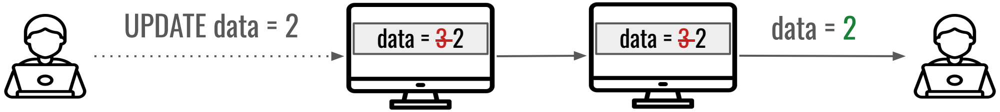
## Reliability



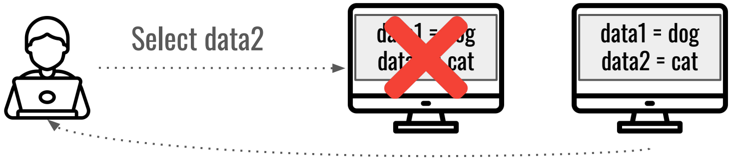

Redundancy in case of failure

# Desirable properties for distributing data

**C**onsistency : All entities see the **same data**



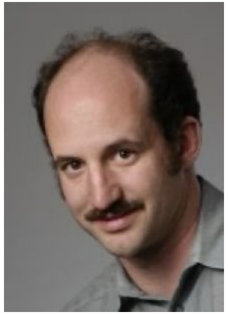

**A**vailability: Every request received a **non-error response for every operation**



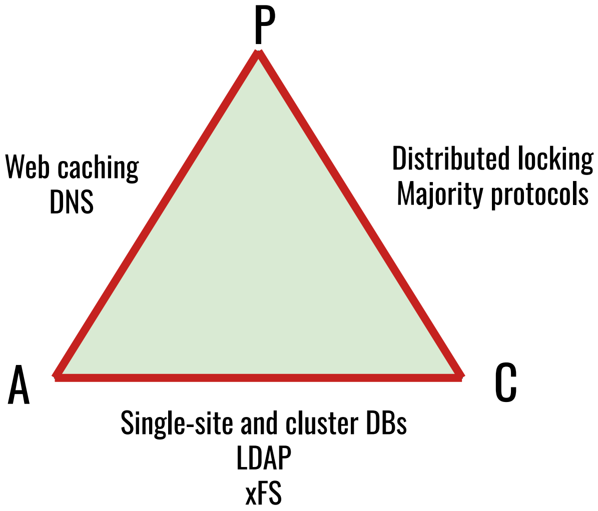

**P**artition tolerance: The system **continues to operate** even when connections between nodes are down/slow

# The CAP Theorem

A network-shared data system can only have **two** out of the three desirable properties



Eric Brewer



# Takeaways

- Network partitions are inevitable: must choose between C and A
  - Forfeit consistency: process the operation even though it might be wrong/lead to inconsistency
  - Forfeit availability:  return an error if we cannot guarantee that data is up to data
- Properties are continuous: do not need to forego one all the time
- Network partitions are rare: only make tradeoff when partitions occur

[Brewer, E. (2012). CAP twelve years later: How the" rules" have changed. Computer, 45(2), 23-29.]

# Consistency versus Availability

Databases (C over A)

- **A**tomic
- **C**onsistent*
- **I**solation
- **D**urability

NoSQL (A over C)

- **B**asically **A**vailable
- **S**oft state
- **E**ventual consistency

* Consistency in databases includes preserving all databases rules. CAP's consistency involves single-copy consistency

# RDBMS to NoSQL

Many current workloads do not require ACID properties, allowing us to loosen consistency requirements (eventual consistency) to improve performance and availability

- Benefits: simple scale out, faster read/writes, maybe easier to use?

Some slides based on: https://courses.engr.illinois.edu/cs425/fa2016/L9-11.FA16.pdf

# Relational Database Management Systems (RDBMS)

- Schema oriented
- Data organized into tables
  - Defined columns
  - Constraints (e.g., foreign key)
  - Types
- Supports joins between tables
- Structured Query Language (SQL)
- Highly Optimized Query Planners

| ID | Name | Email |
|---|---|---|
| 1 | Jane | jane@ |
| 2 | Bob | bob@ |
| 3 | Mary | mary@ |

```
SELECT *
FROM USERS
WHERE name = ‘Mary’
```

| ID | Name | Email |
|---|---|---|
| 3 | Mary | mary@ |

| grade_id | homework | student_id | grade |
|---|---|---|---|
| 1 | 1 | 1 | 60 |
| 2 | 1 | 2 | 40 |
| 3 | 2 | 1 | 75 |

```
SELECT users.name, grades.grade
FROM users JOIN grades
ON users.id = grades.student_id
```

| name | grade |
|---|---|
| Jane | 60 |
| Bob | 40 |
| Jane | 75 |

# ACID Properties

- **Atomic**: Transactions execute in “all-or-nothing” manner
  - Transactions can be one or more operations
  - Example: buying a plane ticket
- **Consistency**: Transition from one valid state to another.
  - All DB rules must be preserved following a transaction.
- **Isolation**: No transaction sees uncommitted changes of concurrent transactions.
  - Read phenomena: dirty reads, non-repeatable reads, phantom reads
- **Durability**: State changes caused by committed transactions are preserved.

# Scaling RDBMS

- ACID properties make scaling out difficult. More common to scale up (add more resources).
- Distribution approach depends on needs:
  - **Read heavy**: add replicas
    - Add as many replicas needed to handle the workload
    - Balance queries across replicas in Round-Robin fashion
  - **Write heavy:** Partition database across many servers
    - Put different tables on different machines or shard one table across many machines
    - Horizontal fragmentation: split selection query
    - Vertical fragmentation: split by projection queries

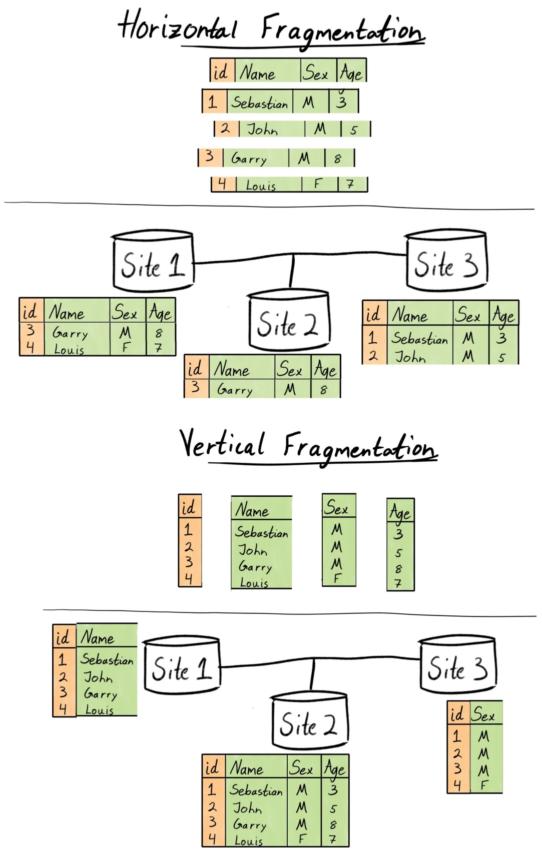
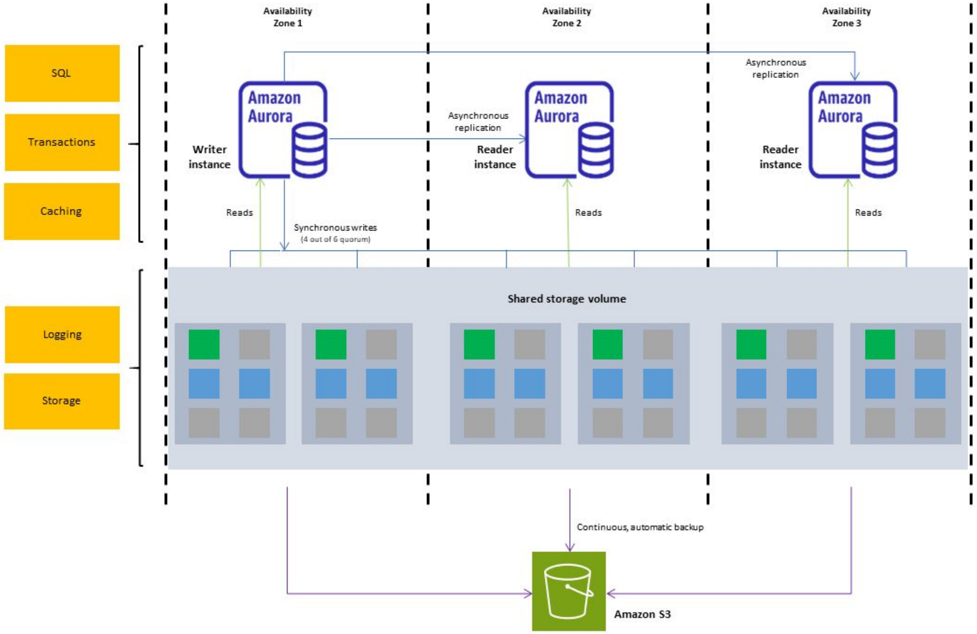



Image credit: https://stackoverflow.com/a/61680342

# Amazon Aurora

MySQL and PostgreSQL-compatible relational database in the cloud



# Adapting to evolving workload needs

- Newer workloads:
  - Data are large and unstructured
  - Random access read/writes
  - Joins are infrequent

- NoSQL Goals:
  - Speed
  - Avoid single point of failure
  - Reduce ownership costs
  - Fewer system administrators
  - Elastic scalability
  - Scale out

# Not Only SQL (NoSQL)

- Unstructured
- No schema
- Sparse columns/rows
- No foreign keys (joins might not be supported)
- Does not use SQL as a query language

| | |
|---|---|
| 1 | Name: kyle, grade: 100 |
| 2 | Name: bob, grade: 60 |
| 3 | Name: jane, grade: 95 |

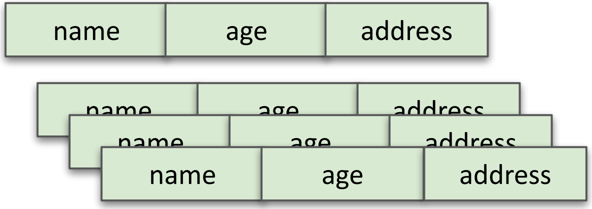
| ID | Document |
|---|---|
| 1 | Name: Kyle |
| 2 | Name: bob, email: bob@ |
| 3 | Email: jane@ |

# NoSQL advertised benefits

- Simplicity (from a design/interaction perspective)
- Horizontal scaling
- High availability
- Performance (depending on the use case)

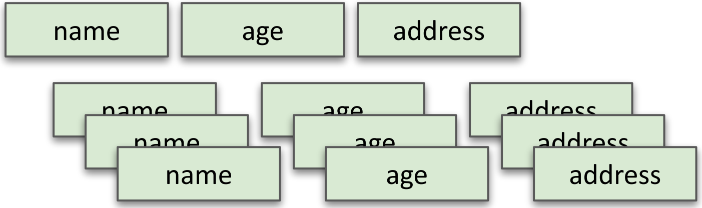
# Rows vs Columns

- Row store: Store all attributes of a tuple together
  - Advantage: easy to add/modify records
  - Disadvantage: may read unnecessary data




➔ Column store: Store all rows of for an attribute together

- ◆ Advantage: only read relevant attributes
- ◆ Read/write requires multiple accesses



# NoSQL variations

- Key-value stores
  - Store simple, flat lists of key-value pairs
  - Expose get and put operations
  - Redis, Memcached
- Document stores
  - Each key is paired with a whole document (e.g., JSON)
  - MongoDB, CouchDB
- Column family
  - Each row is addressed by a key and contains one or more columns
  - Columns are key-value pairs (tuple: name, value, timestamp)
  - Cassandra, HBase
- Graph
  - Neo4J

# Organizing NoSQL stores

- Most NoSQL stores group key-value pairs into something that looks like a table
  - Cassandra: Column families
  - MongoDB: Collections
  - HBase: Tables
  - DynamoDB: Tables

- NoSQL stores typically store a column (or group of columns) together
  - Index entries in a column for easy retrieval
  - Supports range queries so you do not need to get entire database

# Eventual consistency

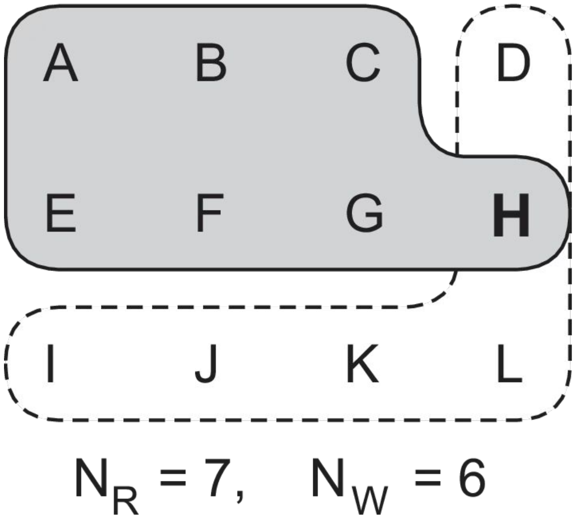
- Tradeoff between consistency and availability when there is a network partition
  - RDBMS: strong consistency over availability
  - NoSQL: availability over consistency (eventual consistency)

- If writes to a key stop, then all values (replicas) will eventually converge
  - If writes continue, we will keep trying to converge
  - Maintain set of updated values lagging latest values sent by clients

- Implication: may return stale values to clients (e.g., during many back-to-back writes)
- Work well with periods of low writes – system converges quickly

# Tunable consistency levels

- Client is allowed to choose consistency level for each operation (read/write)
- Strong consistency: $R + W > N$
- Eventual consistency: $R + W \leq N$
  - `ANY:` write written to at least one node
    - **Fastest**: node will cache write and reply to client quickly
  - `ALL:` write written to all nodes
    - Ensures strong consistency, but **slow**
  - `ONE:` At least one replica
    - Faster than `ALL` but cannot tolerate more than one failure
  - `QUORUM:` written to a configurable quorum across nodes




https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlConfigConsistency.html

- Created in 2007

- Document-oriented database
  - Schema-less
  - Stores arbitrary documents
    - Documents are given a unique ID (`_id`)
    - Documents contain arbitrary keys/values
  - Provides its own query language
  - Leader-worker replication across replica sets
  - No joins/transactions

# MongoDB architecture

- Data is sharded across machines
  - Scale and performance
- Documents are distributed across shards using an immutable “shard key” (from keys in documents)
  - Knowledge of data/queries can optimize use
- Supports different sharding strategies
  - E.g., Hashing and range-based
- Query router sends queries to appropriate shards

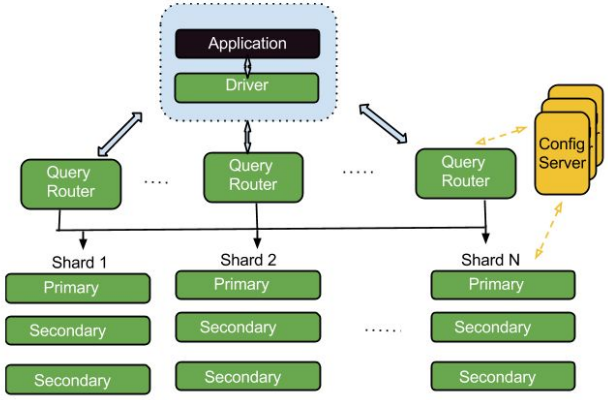



Image credit:
https://medium.datadriveninvestor.com/mastering-mongodb-understanding-and-utilizing-the-nosql-database-41ec3890f97

# BSON

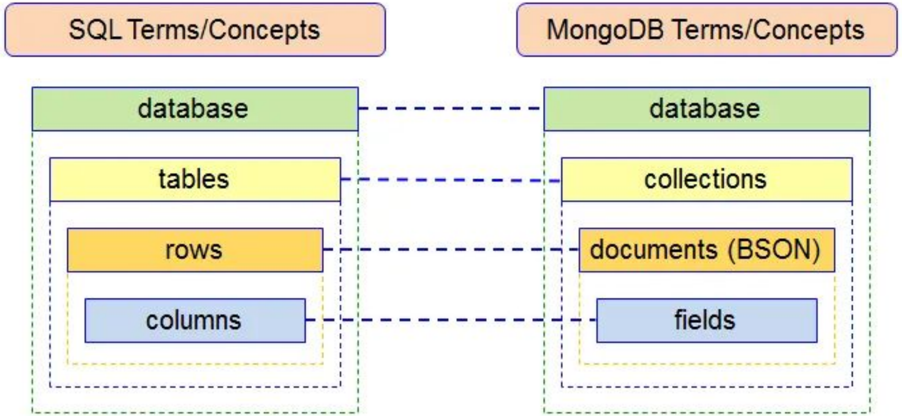
- Mongo wanted to support users and JSON is pervasive

- Challenges of JSON:
    - Limited data types (no binary, date)
    - Objects do not have fixed length (slows processing)

- BSON (Binary JSON)
    - Adds data types and fixed lengths
    - Anything in JSON can be stored as BSON

- MongoDB stores data in BSON internally and over the network

# MongoDB concepts




https://medium.com/nerd-for-tech/all-basics-of-mongodb-in-10-minutes-baddaf6b6625

# MongoDB exercise

- Installation
  a. Docker database server

  ```
  docker run -p 27017:27017 mongo:latest
  ```

  b. Python client (in your terminal)

  ```
  pip install pymongo
  ```

- Docs: https://pymongo.readthedocs.io/en/stable/

# Create database/column

```
import pymongo
mongo_client = pymongo.MongoClient("mongodb://localhost:27017/")
db = mongo_client["mydatabase"]
col = db["customers"]

# Insert records
l = [{"name":"Kyle","office":"303"},
     {"name":"Tyler","office":"312"},]

x = col.insert_many(l)
```

# Query things

Search for “kyle”

```
for x in col.find({'name': 'Kyle'}):
     print(x)
```

Get only some attributes

```
for x in col.find({},{ "_id": 0, "office": 1 }):
  print(x)
```

# Exercise

1. Create a database that stores students and their grades for MPCS course
   - E.g., `student_name, student_id, hw1, hw2, hw3`

2. Insert 5+ records

3. Find number of total records (hint: collection count_documents)

4. Find a student by name and get only their name and hw1 grades

5. Find students who received more than 50 on hw1 (read the docs for query syntax)

6. (Optional/advanced) Compute each student’s grade for the course

# Submission

https://classroom.github.com/a/Z8_TBszc