UMD DATA605 - Big Data Systems

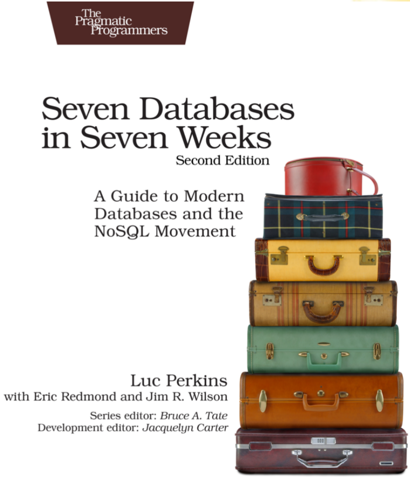
NoSQL Databases

Instructor: Dr. GP Saggese - gsaggese@umd.edu

SCIENCE ACADEMY

Resources

- Concepts in the slides
- Tons of tutorials on line
- Silbershatz Chap 10.2
- Nice high-level view:
    - Seven Databases in Seven Weeks, 2e



From SQL to NoSQL

- **DBs are central tools to big data**
  - New applications, new constraints to data / storage
  - Around 2000s NoSQL “movement” started
    - Initially it meant “No SQL” -> “Not Only SQL”
- **DBs (e.g., SQL vs NoSQL) make different trade-offs**
  - Different worldviews
  - Schema vs schema-less
  - Rich vs fast ability of query
  - Strong consistency (ACID), weak, eventual consistency
  - APIs (SQL, JS, REST)
  - Horizontal vs vertical scaling, sharding, replication schemes
  - Indexing (for rapid lookup) vs no indexing
  - Tuned for reads or writes, how much control over tuning
- **The user base / applications have expanded**
  - IMO Postgres + Mongo cover 99% of use cases
  - Any data scientist / engineer needs to be familiar with both
  - “Which DB solves my problem best?”
- **Polyglot model**
  - Use more than one DB in each project
  - Relational DBs are not going to disappear any time soon

Issues with Relational DBs

- **Relational DBs have drawbacks**
  - 1 Application-DB impedance mismatch
  - 2 Schema flexibility
  - 3 Consistency in distributed set-up
  - 4 Limited scalability
- In the next slides for each drawback we will discuss:
  - **What is the problem**
  - **Possible solutions**
    - Within relational SQL paradigm
    - With NoSQL approach

1 App-DB Impedance Mismatch: Problem

- **Mismatch between how data is represented in the code and in a relational DB**
  - Code thinks in terms of:
    - Data structures (e.g., lists, dictionaries, sets)
    - Objects
  - Relational DB thinks in terms of:
    - Tables (entities)
    - Rows (actual instances of entities)
    - Relationships between tables (relationships between entities)
- **Example of the app-DB mismatch**:
  - Application stores a simple Python map like:

    ```
    # Store a dictionary from name (string) to tags (list of strings)
    tag_dict: Dict[str, List[str]]
    ```
  - A relational DB needs 3 tables:
    - Names(nameId, name) to store the keys
    - Tags(tagId, tag) to store the values
    - Names_To_Tags(nameId, tagId) to map the keys to the values
  - One could denormalize the tables using a single table
    - Names(name, tag)

SCIENCE ACADEMY

1 App-DB Impedance Mismatch: Solutions

- **Ad-hoc mapping layer**
  - Translate objects and data structures into DB data model
    - E.g., you implement a layer that handles storing into the DB "Name to Tags" transparently
    - The code thinks in terms of a map, but there are 3 tables in the DB
  - Cons
    - You need to write / maintain code
- **Object-relational mapping (ORM)**
  - Pros
    - Convert automatically data between object code and relational DB
    - E.g., implement a Person object (e.g., name, phone number, addresses) using DB
    - E.g., SQLAlchemy for Python and SQL
  - Cons
    - Complex types (e.g., struct), polymorphism, inheritance
- '**NoSQL approach**
  - No schema
    - Every object can be flat or complex (e.g., nested JSON)
    - Stored objects (aka documents) can be different

2 Schema Flexibility

- **Problem**
  - Not all applications have data that fits neatly into a schema
  - E.g., data can be nested and / or dishomogeneous (e.g., List[Obj])
- **Within relational DB**
  - Use a schema general enough to accommodate all the possible cases
  - Cons
    - Super-complicated schema with implicit relations
    - DB tables are sparse
    - It is a violation of the basic relational DB assumption
- **NoSQL approach**
  - E.g., MongoDB does not enforce any schema
  - Pros
    - Application does not worry about schema when writing data
  - Cons
    - Application needs to deal with variety of schemas when it processes the data
    - Related to ETL vs ELT data pipelines

3 Consistency in Relational DBs

- **All systems in the real-world fail**
  - Application error (e.g., corner case, internal error)
  - Application crash (e.g., OS issue)
  - Hardware failure (e.g., RAM ECC error, disk)
  - Power failure
- **Relational DBs enforce ACID properties**
  - Need to be guaranteed for any system failure
- **Atomicity**
  - = transactions are "*all or nothing*"
  - Either a transaction (which can be composed of multiple statements) succeeds completely or fails
- **Consistency**
  - = any transaction brings the DB *from a valid state to another*
  - The "invariants" of the DB (e.g., primary, foreign key constraints) must be maintained
- **Isolation**
  - = if transactions are executed *concurrently*, the result is the same as if the transactions were executed *sequentially*
- **Durability**
  - = once a transaction has been committed, the

Application error

Hardware failure

3 Consistency in Distributed DB

- When data scales up or number of clients increases $\rightarrow$ **distributed setup**
- Goals to achieve:
  - Performance (e.g., transaction per seconds)
  - Availability (guarantee of a certain up-time)
  - Fault-tolerance (can recover from faults)
- **Achieving ACID consistency is:**
  - *Not easy* in a single DB setup
    - E.g., Postgres guarantees ACID
    - E.g., MongoDB doesn't guarantee ACID
  - *Impossible* in a distributed DB setup
    - Due to CAP theorem
    - Even weak consistency is difficult to achieve

A = Atomicity

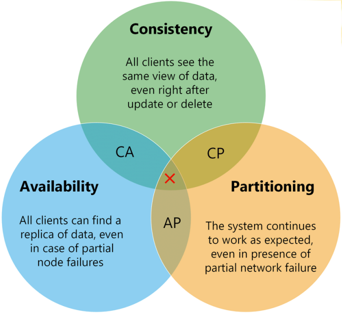
C = Consistency

I = Isolation

D = Durability

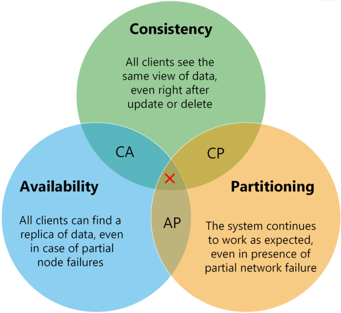
CAP Theorem

- **CAP theorem**: Any *distributed* DB can have *at most two* of the following *three* properties
  - **Consistent**:
    - All clients see the same data
    - Writes are atomic and subsequent reads retrieve the new value
  - **Available**: a value is returned as long as a single server is running
  - **Partition tolerant**: the system still works even if communication is temporary lost (i.e., the network is partitioned)
- Originally a conjecture (Eric Brewer)
- Made formal later (Gilbert, Lynch, 2002)



CAP Corollary

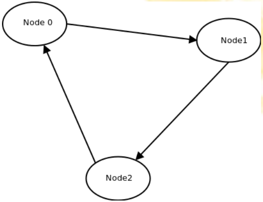
- **CAP Theorem**: pick 2 among consistency, availability, partition tolerance
- *Network partitions* cannot be prevented in large-scale distributed system
  - Minimize probability of failures using redundancy and fault-tolerance
- Need to either sacrifice:
  - *Availability* (i.e., allow system to go down)
    - E.g., banking system
  - *Consistency* (i.e., allow different views of the system)
    - E.g., social network



You are here

CAP Theorem: Intuition

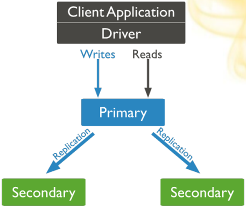
- Imagine there are
  - a client (*Node0*)
  - two DB replicas (*Node1*, *Node2*)
- **A network partition happens**
  - DB servers (*Node1*, *Node2*) can't communicate with each other
  - Users (*Node0*) can access only one of them (*Node2*)
  - **Reads**: the user can access the data of the server in the same partition
  - **Writes**: data can't be updated since multiple users might be updating the data on different replicas, leading to inconsistency
- **CAP theorem**: one needs to sacrifice consistency or availability
- **Available, but not consistent**
  - Sometimes inconsistency is fine (e.g., social networking)
  - Let updates happen on the accessible replica at cost of inconsistency
- **Consistent, but not available**



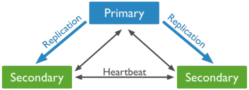
X
X
DB replica
DB replica
Client

Replication Schemes

- **Replication schemes**: how to organize multiple servers implementing a distributed DB
- **Primary-secondary replication**
  - Application only communicates with primary
  - Replicas cannot update local data, but require primary node to perform update
  - Single-point of failure
- **Update-anywhere replication**
  - Aka "multi-master replication"
  - Every replica can update a data item, which is then propagated (synchronously or asynchronously) to the other replicas
- **Quorum-based replication**
  - Let N be the total number of replicas
  - When writing, we make sure to write to W replicas
  - When reading, we read from R replicas and pick the latest update (using timestamps)



Primary-secondary replication

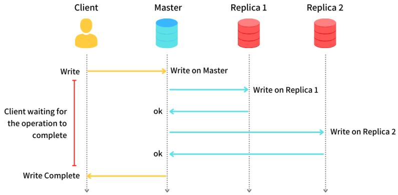


Update-anywhere replication

SCIENCE ACADEMY

Synchronous Replication

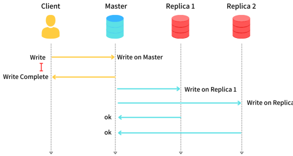
- **Synchronous replication**: updates are propagated to other replicas as part of a single transaction
- Implementations
  - **2-Phase Commit (2PC)**: original proposal for doing this
    - Single point of failure
    - Can't handle primary server failure
  - **Paxos**: more widely used today
    - Doesn't require a primary
    - More fault tolerant
  - Both solutions are complex / expensive
- **CAP theorem**: still only one among Consistency, Availability, in case of Network partition
  - Many systems use relaxed / loose consistency models



Synchronous Replication

Asynchronous Replication

- **Asynchronous replication**
  - The primary node propagates updates to replicas
  - The transaction is completed before replicas are updated (even if there are failures)
  - Commits are quick at cost of consistency
- **Eventual consistency**
  - Popularized by AWS DynamoDB
  - Consistency guaranteed only on the eventual outcome
  - "*Eventual*" can mean after the server or network is fixed
- **"Freshness" property**
  - Under asynchronous updates, a read from a replica may not get the latest version of a data item
  - User can request a version with a certain "freshness"
    - E.g., "data from not more than 10 minutes ago"
    - E.g., it's ok to show price for an airplane ticket that is few minutes old
  - Replicas version their data with timestamps
  - If local replica has fresh data, uses it, otherwise send request to primary node



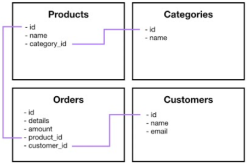
Asynchronous Replication

4 Scalability Issues with RDMS

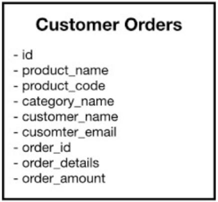
- The sources of relational SQL DB scalability issues are:
- **Locking data**
  - The DB engine needs to lock rows and tables to ensure ACID properties
  - When DB locked:
    - Higher latency $\rightarrow$ Fewer updates per second $\rightarrow$ Slower application
- **Even worse in distributed set-up**
  - Requires replicating data over multiple servers (scaling out)
  - Application becomes even slower
    - Network delays
    - To enforce DB consistency, locks are applied across networks
    - Overhead of replica consistency (2PC, Paxos)

Scalability Issues with RDMS: Solutions

- **Table denormalization**
  - = approach used to increase relational DB performance by adding redundant data
  - Pros
    - Reads become faster: Lock only one table instead of multiple ones (reducing resource contention), No need for joins
  - Cons
    - Writes become slower: There is more data to update (E.g., to update a *category name*, need to do a scan)
    - If we join the tables, we lose relations between tables (this is the main reason of using a relational DB!)
- **Relax consistency**
  - Give up on part of ACID
  - Make definition of consistency weaker (E.g., eventual consistency)
- **NoSQL**



Normalized data



Denormalized data

NoSQL Stores

- **Use cases of large-scale web applications**
  - Applications need real-time access with a few ms latencies
    - E.g., Facebook: 4ms for reads to get snappy UI
  - Applications don't need ACID properties
  - In fact, MongoDB started at DoubleClick (AdTech), acquired by Google
- **Solve previous problems with relational databases**
  - 1 Application-DB impedance mismatch
  - 2 Schema flexibility
  - 3 Consistency in distributed set-up
  - 4 Scalability
- **If you want to really scale out, you must give up something**
  - Give up consistency
  - Give up joins
    - Most NoSQL stores don't allow server-side joins
    - Instead require data to be denormalized and duplicated
  - Only allow restricted transactions
    - Most NoSQL stores will only allow one object transactions
    - E.g., one document / key

Relational DB vs MongoDB

How MongoDB solves the four RDBM problems

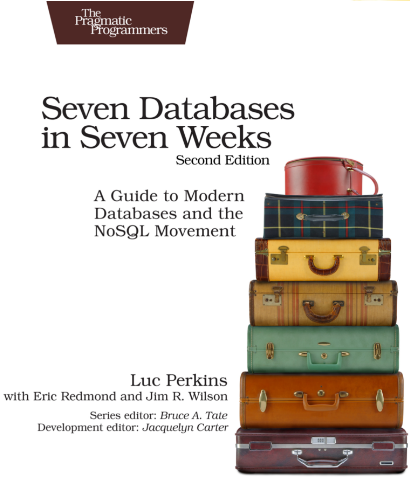
- **1 Application-DB impedance mismatch**
  - Store data as nested objects
- **2 Schema flexibility**
  - No schema, no tables, no rows, no columns, no relationships between tables
- **3 Consistency in replicated set-up**
  - Application decides consistency level
    - *Synchronous*: wait until primary and secondary servers are updated
    - *Quorum synchronous*: wait until the majority of secondary servers are updated
    - *Asynchronous*, *eventual*: wait until only the primary is updated
    - "*Fire and forget*": not even wait until the primary persisted the data
- **4 Scalability**
  - Updating data means locking only one document, and not entire collection
  - Sharding: use more machines to do collectively do more work

UMD DATA605 - Big Data Systems

- Issues with Relational DBs
- **NoSQL Taxonomy**
- (Apache) HBase

Resources

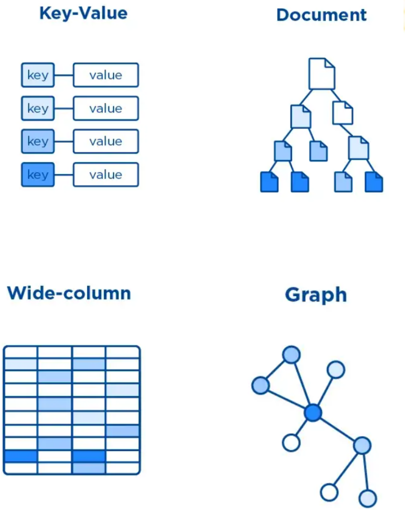
- Concepts in the slides
- Silberschatz Chapter 23.6
- Mastery:
  - Seven Databases in Seven Weeks, 2e



DB Taxonomy

- **At least five DB genres**
  - *Relational* (e.g., Postgres)
  - *Key-value* (e.g., Redis)
  - *Document* (e.g.,. MongoDB)
  - *Columnar* (e.g., Parquet)
  - *Graph* (e.g., Neo4j)

- **Criteria to differentiate DBs**
  - Data model
  - Trade-off with respect to CAP theorem
  - Querying capability
  - Replication scheme

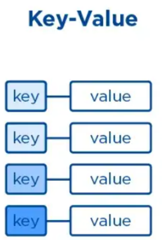


Relational DB

- E.g., *Postgres*, MySQL, Oracle, SQLite
- **Data model**
  - Based on set-theory and relational algebra
  - Data as two dimensional tables with rows and columns
  - Many attribute types (e.g., numeric, strings, dates, arrays, blobs)
  - Attribute types are strictly enforced
  - SQL query language
  - ACID consistency
- **Application**
  - Any relational tabular data
- **Good for**
  - When layout of data is known, but not the data access pattern
  - Complexity upfront for schema to achieve query flexibility
  - Used when data is regular
- **Not so good for**
  - When data is hierarchical (not a nice row in one or more tables)
  - When data structure is variable/dishomogeneous (record-to-record variation)

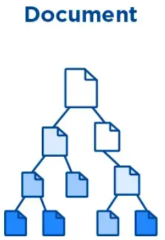
Key-Value Store

- E.g., Redis, DynamoDB, *Git*, *AWS S3*, *filesystem*
- **Data model**
  - Map simple keys (e.g., strings) to more complex values (e.g., it can be anything, binary blob)
  - Support get, put, and delete operations on a primary key
- **Application**
  - Caching data
  - E.g., store users' session data in a web application
  - E.g., store the shopping cart in an e-commerce application
- **Good for**
  - Useful when data is not "related" (e.g., no joins)
  - Lookups are fast
  - Easy to scale horizontally using partitioning scheme
- **Not so good for**
  - Not great if data queries are needed
  - Lacking secondary indexes and scanning capabilities



Document Store

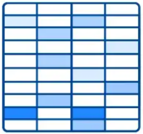
- E.g., *MongoDB*, CouchDB
- **Data model**
  - Like key-value but value is a document (i.e., a nested dict)
  - Each document has a unique ID (e.g., hash)
  - Any number of fields per document, even nested
    - E.g., JSON, XML, dict data
- **Application**
  - Any semi-structured data
- **Good for**
  - When you don't know how your data will look like
  - Map well to OOP models (less impedance mismatch between application and DB)
  - Since documents are not related, it's easy to shard and replicate over distributed servers
- **Not so good for**
  - Complex join queries
  - Denormalized form is the norm



Columnar Store

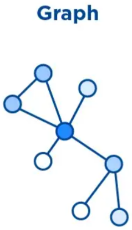
- E.g., *HBase*, Cassandra, *Parquet*
- **Data model**
  - Data is stored by columns, instead of rows like in relational DBs
  - Share similarities with both key-value and relational DBs
    - Keys are used to query values (like key-value stores)
    - Values are groups of zero or more columns (like relational stores)
- **Application**
  - Storing web-pages
  - Storing time series data
  - OLAP workloads
- **Good for**
  - Horizontal scalability
  - Enable compression and versioning
  - Tables can be sparse without extra storage cost
  - Columns are inexpensive to add
- **Not so good for**
  - Need to design the schema based on how you plan to query the data
  - No native joins, applications need to handle join

Wide-column

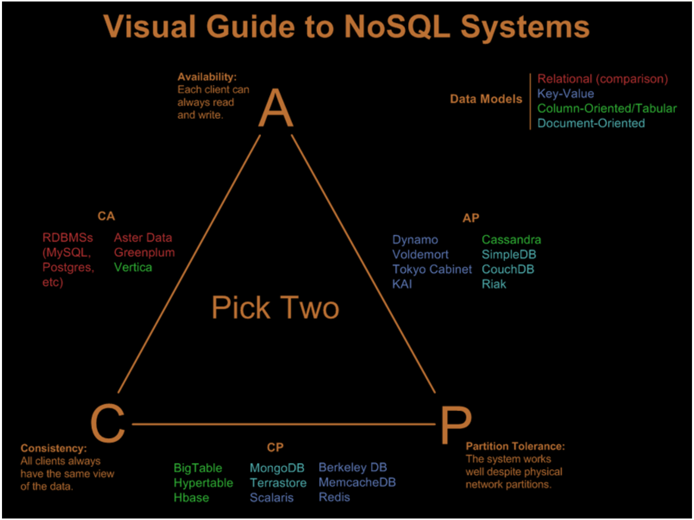


Graph DB

- E.g., *Neo4J*, GraphX
- **Data model**
  - Highly interconnected data, storing nodes and relationships between nodes
  - Both nodes and edges have properties (i.e,. key-value pairs)
  - Queries involve traversing nodes and relationships to find relevant data
- **Applications**
  - Social data
  - Recommendation engines
  - Geographical data
- **Good for**
  - Perfect for "networked data", which is difficult to model with relational model
  - Good match for OO systems
- **Not so good for**
  - Don't scale well, since it's difficult to partition graph on different nodes
    - Store the graph in the graph DB and the relations in a key-value store



Taxonomy by CAP

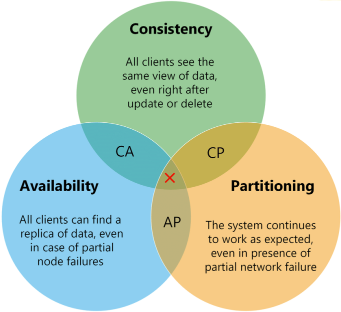


From http://blog.nahurst.com/visual-guide-to-nosql-systems

SCIENCE ACADEMY

Taxonomy by CAP

- **CA (Consistent, Available) systems**
  - Have trouble with partitions and typically deal with it with replication
  - Traditional RDBMSs (e.g., PostgreSQL, MySQL)
- **CP (Consistent, Partition-Tolerant) systems**
  - Have trouble with availability while keeping data consistent across partitioned nodes
  - BigTable (column-oriented/tabular)
  - HBase (column-oriented/tabular)
  - MongoDB (document-oriented)
  - Redis (key-value)
  - MemcacheDB (key-value)
  - Berkeley DB (key-value)



- **AP (Available, Partition-Tolerant) systems**
  - Achieve "eventual consistency" through replication and verification
  - Dynamo (key-value)
  - Cassandra (column-oriented/tabular)
  - CouchDB (document-oriented)