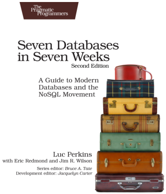
UMD DATA605 - Big Data Systems

NoSQL Databases

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

- Online tutorials
- Silbershatz Chap 10.2
- High-level view:
  - Seven Databases in Seven Weeks, 2e



SCIENCE ACADEMY

From SQL to NoSQL

- **DBs are central tools to big data**
  - New applications, new data/storage constraints
  - ~2000s NoSQL "movement" started
    - Initially "No SQL" → then "Not Only SQL"
- **DBs (e.g., SQL vs NoSQL) make different trade-offs**
  - Different worldviews
  - Schema vs schema-less
  - Rich vs fast query ability
  - Strong consistency (ACID), weak, eventual consistency
  - APIs (SQL, JS, REST)
  - Horizontal vs vertical scaling, sharding, replication
  - Indexing (for rapid lookups) vs no indexing
  - Tuned for reads or writes, control over tuning
- **User base/applications have expanded**
  - IMO Postgres + Mongo cover 99% of use cases
  - Data scientists/engineers need familiarity with both
  - "Which DB solves my problem best?"
- **Polyglot model**
  - Use more than one DB per project
  - Relational DBs won't disappear 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
- For each drawback:
  - **Problem**
  - **Solutions**
    - Within relational SQL paradigm
    - With NoSQL approach

1 App-DB Impedance Mismatch: Problem

- **Mismatch between data representation in code and relational DB**
  - Code uses:
    - Data structures (e.g., lists, dictionaries, sets)
    - Objects
  - Relational DB uses:
    - Tables (entities)
    - Rows (instances of entities)
    - Relationships between tables
- **Example of app-DB mismatch**:
  - Application stores a Python map: # Store a dictionary from name (string) to tags (list of strings). tag_dict: Dict[str, List[str]]
  - Relational DB needs 3 tables:
    - Names(<u>nameId</u>, name) for keys
    - Tags(<u>tagId</u>, tag) for values
    - Names_To_Tags(<u>nameId</u>, <u>tagId</u>) to map keys to values
  - Denormalize 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 model
    - E.g., implement a layer for "Name to Tags" storage
    - Code uses a simple map, but DB has 3 tables
  - Cons
    - Need to write and maintain code
- **Object-relational mapping (ORM)**
  - Pros
    - Automatic data conversion between object code and DB
    - E.g., implement Person object using DB
    - E.g., SQLAlchemy for Python and SQL
  - Cons
    - Complex types (e.g., struct), polymorphism, inheritance
- **NoSQL approach**
  - No schema
    - Objects can be flat or complex (e.g., nested JSON)
    - Stored objects (documents) can vary

2 Schema Flexibility

- **Problem**
  - Data may not fit neatly into a schema
  - E.g., nested or dishomogeneous data (e.g., List[Obj])
- **Within relational DB**
  - Use a general schema for all cases
  - Cons
    - Complicated schema with implicit relations
    - Sparse DB tables
    - Violates basic relational DB assumptions
- **NoSQL approach**
  - E.g., MongoDB does not enforce schema
  - Pros
    - No schema concerns when writing data
  - Cons
    - Handle various schemas during data processing
    - Related to ETL vs ELT data pipelines

3 Consistency in Relational DBs

- **All systems 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**
  - Guarantee for any system failure
- **Atomicity**
  - Transactions are "all or nothing"
  - Transaction (with multiple statements) succeeds completely or fails
- **Consistency**
  - Transaction brings DB from valid state to another
  - Maintain DB invariants (primary, foreign key constraints)
- **Isolation**
  - Concurrent transactions yield same result as sequential execution
- **Durability**
  - Committed transaction content preserved for any system failure

Application error

Hardware failure

3 Consistency in Distributed DB

- Scale data or clients → **distributed setup**
- Goals:
  - Performance (e.g., transactions per second)
  - Availability (e.g., up-time guarantee)
  - Fault-tolerance (recover from faults)
- **Achieving ACID consistency:**
  - *Not easy* in single DB
    - E.g., Postgres guarantees ACID
    - E.g., MongoDB doesn't
  - *Impossible* in distributed DB
    - Due to CAP theorem
    - Even weak consistency is difficult

A = Atomicity

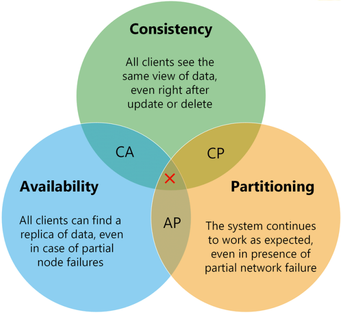
C = Consistency

I = Isolation

D = Durability

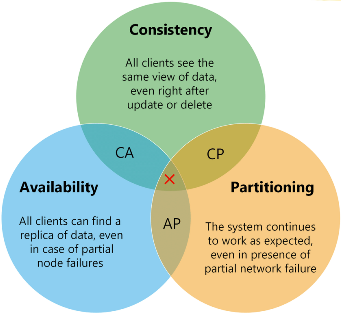
CAP Theorem

- **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**: Returns a value if a single server is running
  - **Partition tolerant**: System works even if communication is temporarily lost (network partitioned)
- Originally a conjecture (Eric Brewer)
- Made formal later (Gilbert, Lynch, 2002)



CAP Corollary

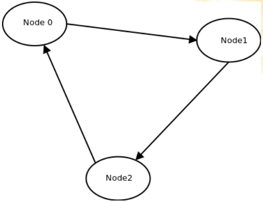
- **CAP Theorem**: pick 2 among consistency, availability, partition tolerance
- *Network partitions* cannot be prevented in large-scale distributed systems
  - Minimize failure probability using redundancy and fault-tolerance
- Sacrifice either:
  - *Availability* (allow system downtime)
    - E.g., banking system
  - *Consistency* (allow different system views)
    - E.g., social network



You are here

CAP Theorem: Intuition

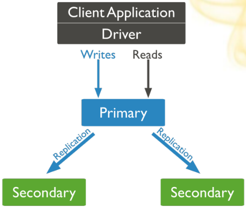
- Imagine:
  - Client (*Node0*)
  - Two DB replicas (*Node1*, *Node2*)
- **Network partition occurs**
  - DB servers (*Node1*, *Node2*) can't communicate
  - Users (*Node0*) access only one (*Node2*)
  - *Reads*: Access data on the same partition
  - *Writes*: Can't update due to potential inconsistency
- **CAP theorem**: Sacrifice consistency or availability
- **Available, not consistent**
  - Inconsistency acceptable (e.g., social networking)
  - Allow updates on accessible replica
- **Consistent, not available**
  - Inconsistency unacceptable (e.g., banking)
  - Stop service to maintain consistency



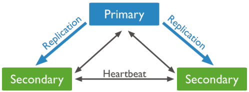
X
X
DB replica
DB replica
Client

Replication Schemes

- **Replication schemes**: Organize multiple servers for a distributed DB
- **Primary-secondary replication**
  - Application only communicates with primary
  - Replicas cannot update local data, but require primary for updates
  - Single-point of failure
- **Update-anywhere replication**
  - Aka "multi-master replication"
  - Every replica can update data, propagated to others
- **Quorum-based replication**
  - N: Total replicas
  - Write to W replicas
  - Read from R replicas, pick latest update (timestamps)



Primary-secondary replication

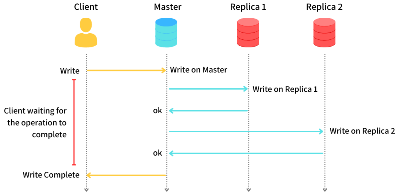


Update-anywhere replication

SCIENCE ACADEMY

Synchronous Replication

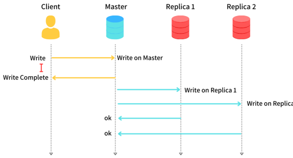
- **Synchronous replication**: updates propagate to replicas in a single transaction
- Implementations
  - **2-Phase Commit (2PC)**: original method
    - Single point of failure
    - Can't handle primary server failure
  - **Paxos**: widely used
    - No primary required
    - More fault tolerant
  - Both are complex/expensive
- **CAP theorem**: only one of Consistency or Availability during Network partition
  - Many systems use relaxed consistency models



Synchronous Replication

Asynchronous Replication

- **Asynchronous replication**
  - Primary node updates replicas
  - Transaction completes before replicas update
  - Quick commits, less consistency
- **Eventual consistency**
  - Popularized by AWS DynamoDB
  - Consistency only on eventual outcome
  - “Eventual” may mean after server/network fix
- **"Freshness" property**
  - Read from replica may not be latest
  - Request version with specific “freshness”
    - E.g., “data from not more than 10 minutes ago”
    - E.g., show airplane ticket price a few minutes old
  - Replicas use timestamps for data versioning
  - Use local replica if fresh, else request primary node



Asynchronous Replication

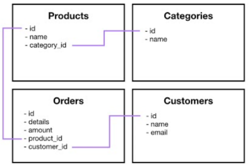
SCIENCE ACADEMY

4 Scalability Issues with RDMS

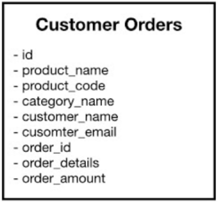
- Sources of SQL DB scalability issues:
- **Locking data**
  - DB engine locks rows/tables for ACID properties
  - When locked:
    - Higher latency → Fewer updates/second → Slower application
- **Worse in distributed set-up**
  - Requires data replication over multiple servers (scaling out)
  - Slower application due to:
    - Network delays
    - Locks across networks for DB consistency
    - Overhead of replica consistency (2PC, Paxos)

Scalability Issues with RDMS: Solutions

- **Table denormalization**
  - Increase performance by adding redundant data
  - Pros
    - Faster reads: Lock one table, no joins
  - Cons
    - Slower writes: More data to update
    - Lose table relations
- **Relax consistency**
  - Compromise on ACID
  - Weaken consistency (e.g., eventual consistency)
- **NoSQL**



Normalized data



Denormalized data

SCIENCE ACADEMY

NoSQL Stores

- **Use cases of large-scale web applications**
  - Real-time access with ms latencies
    - E.g., Facebook: 4ms for reads
  - No need for ACID properties
  - MongoDB started at DoubleClick (AdTech), acquired by Google
- **Solve problems with relational databases**
  - Application-DB impedance mismatch
  - Schema flexibility
  - Consistency in distributed setup
  - Scalability
- **To scale out, give up something**
  - Consistency
  - Joins
    - Most NoSQL stores don't allow server-side joins
    - Require data denormalization and duplication
  - Restricted transactions
    - Most NoSQL stores allow one object transactions
    - E.g., one document/key

Relational DB vs MongoDB

- How MongoDB solves four RDBM problems
- **1 Application-DB impedance mismatch**
  - Store data as nested objects
- **2 Schema flexibility**
  - No schema, tables, rows, columns, or table relationships
- **3 Consistency in replicated set-up**
  - Application decides consistency level
    - *Synchronous*: wait for primary and secondary updates
    - *Quorum synchronous*: wait for majority of secondary updates
    - *Asynchronous*, *eventual*: wait for primary update
    - *"Fire and forget"*: no wait for primary persistence
- **4 Scalability**
  - Lock only one document, not entire collection
  - Sharding: use more machines for more work