Distributed Database and NoSQL Database

CS 4750
Database Systems

Pattamsetti, “Distributed Computing in Java 9,” Ch. 6
Recap

- Core Database Systems
  - Database design: E-R, Normalization
  - Database programming
  - SQL and RA
  - Database security
  - Indexing
  - Query cost estimation (part of optimization)
  - Transactions – ACID

- Distributed relational databases – CAP

- Non-relational database: NoSQL – BASE
Distributed Database

Distributed database (DDB)

- A collection of multiple, logically interconnected databases that are distributed over a computer network
- Data are physically stored across multiple sites, managed by a DBMS that is independent of the other site
- Data at any site available to users at other sites
- Sites may be far apart, linked by some forms of telecommunication lines (secure lines or Internet)
- Sites that are close together may be linked by a local area network (LAN)
Distributed Database Management System

Distributed database management system (DDBMS)

• A centralized software system that manages the DDB
• Synchronizes the databases periodically
• Provides an access mechanism that makes the distribution transparent to the users (as if it were all stored in a single location)
• Ensures that the data modified at any remote site is universally updated
• Supports a huge number of users simultaneously
• Maintains data integrity of the databases
Distributed Database Architecture

[Ref: Pattamsetti, “Distributed Computing in Java 9,” p. 168]
Why Distributed Database

- **Reliability**: if a failure occurs on one site of the distribution, the entire database does not experience a setback.

- **Availability**: if there is a failure in another site of the distribution, users can still get access to the local database.

- **Scalability**: additional nodes can be added to the distribution, supporting growth of business and increase of processors.

- **Security**: permissions can be set for individual database.

- **Cost-effective**: Users access remote data less frequently; thus reducing bandwidth usage.

- **Speed and resource efficiency**: requests and interactivities with the database are performed at a local level, decreasing remote traffic.
Challenges

• **Security**: due to the Internet usage

• **Consistency issues**: databases must be synchronized periodically to ensure data integrity

• **Increased storage requirements**: due to replication of databases

• **Multiple location access**: transactions may access data at one or more sites
Distributed Strategies

Based on the organizational needs and information split and exchange requirements, the distributed database environment can be designed in two ways:

• **Homogeneous**
  • Use the same DBMS for all database nodes that take part in the distribution

• **Heterogeneous**
  • May use a diverse DBMS for some of the nodes that take part in the distribution
Homogeneous Distributed DB

- Information is distributed between all the nodes
- The same DBMS and schema are used across all the databases
- The distributed DBMS controls all information
- Every global user must access the information from the same global schema controlled by the distributed DBMS
- A combination of all the individual DB schemas makes the global schema

[Ref: Pattamsetti, “Distributed Computing in Java 9,” p. 161]
Heterogeneous Distributed DB

- Information is distributed between all the nodes
- Different DBMS and schemas may be used across the databases
- Local users (interacting with one of the individual database) can access the corresponding DBMS and schema
- Users who want to access the global information can communicate with the distributed DBMS, which has a global schema (a combination of all the individual DB schemas)

[Ref: Pattamsetti, “Distributed Computing in Java 9,” p. 163]
Distributed DB Setup Method

• The process of setting up the distributed DB environment involves a thorough analysis and design

• Ongoing and future information maintenance must be determined
  • Synchronous: information across all nodes should be kept in sync all the time
  • Asynchronous: information is replicated at multiple nodes to make it available for other nodes

• Once the analysis for a specific distributed DB environment is made, the setup can be performed in one of the following ways:
  • Replication
  • Fragmentation/partitioning (horizontal or vertical)
  • Hybrid setup
Replication

- Maintain multiple copies of the database instances, stored in different sites

- Easy and minimum risk process as the information is copied from one instance to another without a logical separation

- Each individual node has the complete information

- Efficient in accessing the information without having network traversals and reduces the risk of network security

- Require more storage space

- Take longer to synchronize all the nodes when the information across all the nodes needs to be updated
Horizontal Fragmentation

• Splitting the **rows** of a table (or a relation between two or more nodes, containing databases) to form a distributed database – “split by region”

• Each individual database has a set of rows that belong to the table or relation that belongs to the specific database

\[
\begin{array}{|c|c|c|}
\hline
PK & A & B \\
\hline
\hdashline
PK & A & B \\
\hline
... & ... & ... \\
\hline
PK & A & B \\
\hline
\end{array}
\]

\[
N \text{ nodes}
\]

\[
R_1, \ -\inf \ < \ PK \ <= \ v_1 \\
R_2, \ v_1 \ < \ PK \ <= \ v_2 \\
R_N, \ v_N \ < \ PK \ < \ \inf
\]
Horizontal Fragmentation

• The information access is efficient

• Best if partitions are uniform

• Optimal performance as the local data are only stored in a specific database

• More secure as the information belonging to the other location is not stored in the database

• If a user wants to access some of the other nodes or a combination of node information, the access latency varies.

• If there is a problem with a node or a network, the information related to that node becomes inaccessible to the users
Vertical Partitioning

- (aka normalization process in distributed database setup)

- Splitting the **columns** of a table (or a relation between two or more nodes, containing databases) to form a distributed database while keeping a copy of the base column (primary key) to uniquely identifying each record – “**split by purpose**”

Each node contains all rows of a table.
Vertical Partitioning

- Appropriate if each of the organizational units located in different geographies have separate operations
- Partition based on behavior and function that each node performs
- Best if partitions are uniform
- Poorly chosen columns to split can lead to node bottleneck
- The aggregation of the data involves complex queries with joins across the location database, as no replication is made for non-primary keys
Correctness of Fragmentation

• **Completeness**
  
  Decomposition of a relation $R$ into $R_1, R_2, ..., R_n$ is complete if and only if each data item in $R$ can also be found in some $R_i$

• **Reconstruction**
  
  If a relation $R$ is decomposed into $R_1, R_2, ..., R_n$, reconstructing $R_1, R_2, ..., R_n$ should result in the original $R$

• **Disjointness**
  
  If a relation $R$ is decomposed into $R_1, R_2, ..., R_n$ and data item $d$ is in $R_i$, then $d$ should not be in any other fragment $R_j$ where $i <> j$
Hybrid Setup

• Involve a combination of replication and fragmentation
• Relation is partitioned into several fragments
• Some information is replicated across the database nodes
• Data administrators play a crucial role to choose the right combination to ensure data integrity and security
Threats on ACID Properties

• While distributed database system has many advantages, it imposes a threat on ACID properties

• Consistency in database (ACID)
  • Database relies on a set of integrity constraints
  • DBMS executes each transaction to ensure Atomicity and Isolation and thus maintaining a consistent state

• Consistency in distributed database system with replication
  • Strong consistency:

Final state from a schedule with read and write operations on a replicated object = Final state from a schedule on a single copy of the object with order of operations from a single site preserved

• Weak consistency: (several forms)
CAP Theorem

- **Consistency** -- All copies have some value
- **Availability** -- System can still run via replication even if some parts have failed
- **Partition tolerance**
  - Network can break into two or more parts, each with active systems that communicate with the other parts
  - If some parts fail, no setback on the service
- Must have **exactly two** of the three properties for any system
- Very large system will partition by default, thus choose one of consistency or availability
  - Traditional database – choose **consistency**
  - Most web apps – choose **availability** (except some specific/important parts such as order/payment processing)
**CAP: Example Combination**

- **Consistency/Partition tolerance**
  - Queries are executed on one site. Then they are passed to all other sites, which then execute the queries.
CAP: Example Combination

- **Availability/Partition tolerance**
  - Each site provides services independently. No impact if a network goes down or other sites fail. – resulting in inconsistent DBs
CAP: Example Combination

- **Consistency/Availability**
  - Each site provides services independently as its own system.
Threats on CAP

• Only two of the three properties are guarantees:
  • Consistency – every read receives the most recent write or an error
  • Availability – every request must respond with a non-error
  • Partition tolerance – continued operation in presence of dropped or delayed message

• Distribute RDBMS – partition tolerance + consistency

• NoSQL systems – partition tolerance + availability

(As failures can occur all the time, partition tolerance is essential to distributed systems)
NoSQL

• Loose data model

• Give up built-in OLAP/analysis functionality
  • OLAP (Online Analytical Processing) – allows users to execute complex query to extract multidimensional data for analysis purpose

• Give up built-in ACID consistency

• Rely on BASE consistency model
NoSQL Data Models

**Key-value**

- **Table**
  - Key Space
  - #id1 { Key: Value, Key: Value }
  - #id2 { Key: Value, Key: Value }
  - #id3 { Key: Value1, Value2, Valu3, Value4 }

**Graph**

- Diagram showing relationships between Tom Hanks, Mike Nichols, Penny Marshall, and Ron Howard.

**Document**

- Database
  - Collection
    - Document#1: Key: Value
    - Document#2: (Key: Value, Key: Value)
    - Document#3: Key: Value
    - Document#4: Key: { Value, Key: Value }

**Column-family**

- Table
  - Column Family 1
    - Column 1: #1 { Key: Value, Key: Value }
    - Column 2: #1 { Key: Value, Key: Value }
    - Column 3: #2 { Key: Value, Key: Value }
    - Column 4: #2 { Key: Value, Key: Value }
  - Column Family 2
  - Column Family 3
Example NoSQL Data Models Implementation

Key-value
- Amazon DynamoDB
- Redis
- RocksDB

Graph
- Neo4j
- Amazon Neptune

Document
- AsterixDB
- MongoDB

Column-family
- Google Cloud Bigtable
- Apache HBase
- Cassandra
NoSQL: Key-Value

- (key, value) pairs
- Key can be string, integer, …, unique for the entire data set
- Value can be any type
- Basic operations:
  - get(key) – returns value
  - put(key, value) – add (key, value) pair to the data set

- Example flight information as key-value pairs

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>flightNumber</td>
<td>Complete record of a particular flight</td>
</tr>
<tr>
<td>date</td>
<td>All flight records on a particular date</td>
</tr>
<tr>
<td>(origin, destination, date)</td>
<td>All flight records between the origin and the destination on a particular date</td>
</tr>
</tbody>
</table>
NoSQL: Document

• Data set can be any kinds of files that are parsable
  • Structured document: CSV
  • Semi-structured document: XML, JSON

• Human-readable, may be unordered, heterogeneous data, fields may be skipped

• Example friend information as XML and JSON

XML

```xml
<?xml version="1.0" encoding="utf-8" ?>
<friends>
  <friend>
    <name>Humpty</name>
    <email>humpty@uva.edu</email>
    <phone>434-111-1111</phone>
    <photo>images/friend1.png</photo>
  </friend>
  <friend>
    <name>Dumpty</name>
    <email>dumpty@uva.edu</email>
    <phone>434-222-2222</phone>
    <photo>images/friend2.png</photo>
  </friend>
</friends>
```

JSON

```json
{
  "friends": [
    {
      "name": "Humpty",
      "email": "humpty@uva.edu",
      "phone": "434-111-1111",
      "photo": "images/friend1.png"
    },
    {
      "name": "Dumpty",
      "email": "dumpty@uva.edu",
      "phone": "434-222-2222",
      "photo": "images/friend2.png"
    }
  ]
}
```
Relational DB vs. Semi-Structured Documents

Relational Model
• Fixed schema
• Flat data
• Well-defined

Semi-Structured
• Self-described schema
• Tree-structure
• More flexible
• Extensible

Retrieve table
Scan through rows
Return data

Retrieve document
Parse document tree
Return data
BASE Consistency Model

- With the enormous growth in data, achieving ACID or CAP becomes very difficult.

- A more relaxed set of properties is BASE

- **Basically Available, Soft state, Eventually consistent**

  Most failures do not cause a complete system outage

  System is not allows write-consistent

  Data will eventually converge to agreed values

- Key idea:
  - Databases may not all be in the same state at the same time ("soft state")
  - After synchronization is complete, the state will be consistent
Wrap-Up

• Distributed Database Systems → database scaling

**Replication**
- Multiple copies of each database partition
- Improves fault tolerance
- Read performance ok
- Write performance suffers

**Partitioning**
- Multiple machines to distribute data
- Write performance ok
- Read performance suffers

• RDBMS scaling makes consistent harder
  - Replication: need to prevent inconsistent versions
  - Partitioning: need to coordinate server actions
  - ACID is hard to maintain
Wrap-Up (2)

• RDBMS – intended to be highly consistent (boost availability by sacrificing some consistency)

• NoSQL – intended to be highly available (boost consistency by sacrificing some availability)

• Relational database systems – ACID
• Distributed database systems – CAP
• NoSQL systems – BASE

• Most applications compromise, depending business logic
  • Consistency / availability
  • Scalability
  • Usability
  • Analysis requirements

• No silver-bullet !!