NoSQL Databases and Big Data Storage Systems

Introduction

- NoSQL
  - Not only SQL
- Most NoSQL systems are distributed databases or distributed storage systems
  - Focus on semi-structured data storage, high performance, availability, data replication, and scalability

Introduction (cont'd.)

- NoSQL systems focus on storage of "big data"
- Typical applications that use NoSQL
  - Social media
  - Web links
  - User profiles
  - Marketing and sales
  - Posts and tweets
  - Road maps and spatial data
  - Email

24.1 Introduction to NoSQL Systems

- BigTable
  - Google's proprietary NoSQL system
  - Column-based or wide column store
- DynamoDB (Amazon)
  - Key-value data store
- Cassandra (Facebook)
  - Uses concepts from both key-value store and column-based systems

Introduction to NoSQL Systems (cont'd.)

- MongoDB and CouchDB
  - Document stores
- Neo4J and GraphBase
  - Graph-based NoSQL systems
- OrientDB
  - Combines several concepts
- Database systems classified on the object model
  - Or native XML model

Introduction to NoSQL Systems (cont'd.)

- NoSQL characteristics related to distributed databases and distributed systems
  - Scalability
  - Availability, replication, and eventual consistency
  - Sharding of files
  - High performance data access

Introduction to NoSQL Systems (cont'd.)

- NoSQL characteristics related to data models and query languages
  - Schema not required
  - Less powerful query languages
  - Versioning

Introduction to NoSQL Systems (cont'd.)

- Categories of NoSQL systems
  - Document-based NoSQL systems
  - NoSQL key-value stores
  - Column-based or wide column NoSQL systems
  - Graph-based NoSQL systems
  - Object/object relational databases
  - XML databases

Basic Concepts: Sharding

- Sharding is the process of horizontally partitioning a large dataset into a collection of smaller, more manageable datasets called shards.
- The shards are distributed across multiple nodes, where a node is a server or a machine.
- Each shard is stored on a separate node and each node is responsible for only the data stored on it.
- Each shard shares the same schema, and all shards collectively represent the complete dataset.
- Since each node is responsible for only a part of the whole dataset, read/write times are greatly improved.

Basic Concepts: Replication

- Replication stores multiple copies of a dataset, known as *replicas*, on multiple nodes.
- Replication provides scalability and availability due to the fact that the same data is replicated on various nodes.
- Fault tolerance is also achieved since data redundancy ensures that data is not lost when an individual node fails.
- There are two different methods that are used to implement replication:
  - master-slave
  - peer-to-peer

Basic Concepts: Master Slave Replication

- During master-slave replication, nodes are arranged in a master-slave configuration, and all data is written to a master node.
- Once saved, the data is replicated over to multiple slave nodes.
- All external write requests, including insert, update and delete, occur on the master node, whereas read requests can be fulfilled by any slave node.
- One concern with master-slave replication is read inconsistency, which can be an issue if a slave node is read prior to an update to the master being copied to it.
- To ensure read consistency, a voting system can be implemented where a read is declared consistent if the majority of the slaves contain the same version of the record.

Basic Concepts: Peer to Peer Replication

- With peer-to-peer replication, all nodes operate at the same level.
- Each node, known as a peer, is equally capable of handling reads and writes.
- Each write is copied to all peers.
- Peer-to-peer replication is prone to write inconsistencies that occur as a result of a simultaneous update of the same data across multiple peers.
  - Concurrency control can lock the record being updated. While this ensures consistency, availability is affected
- Like master-slave replication, reads can be inconsistent during the time period when some of the peers have completed their updates while others perform their updates.
- Reads eventually become consistent when the updates have been executed on all peers.

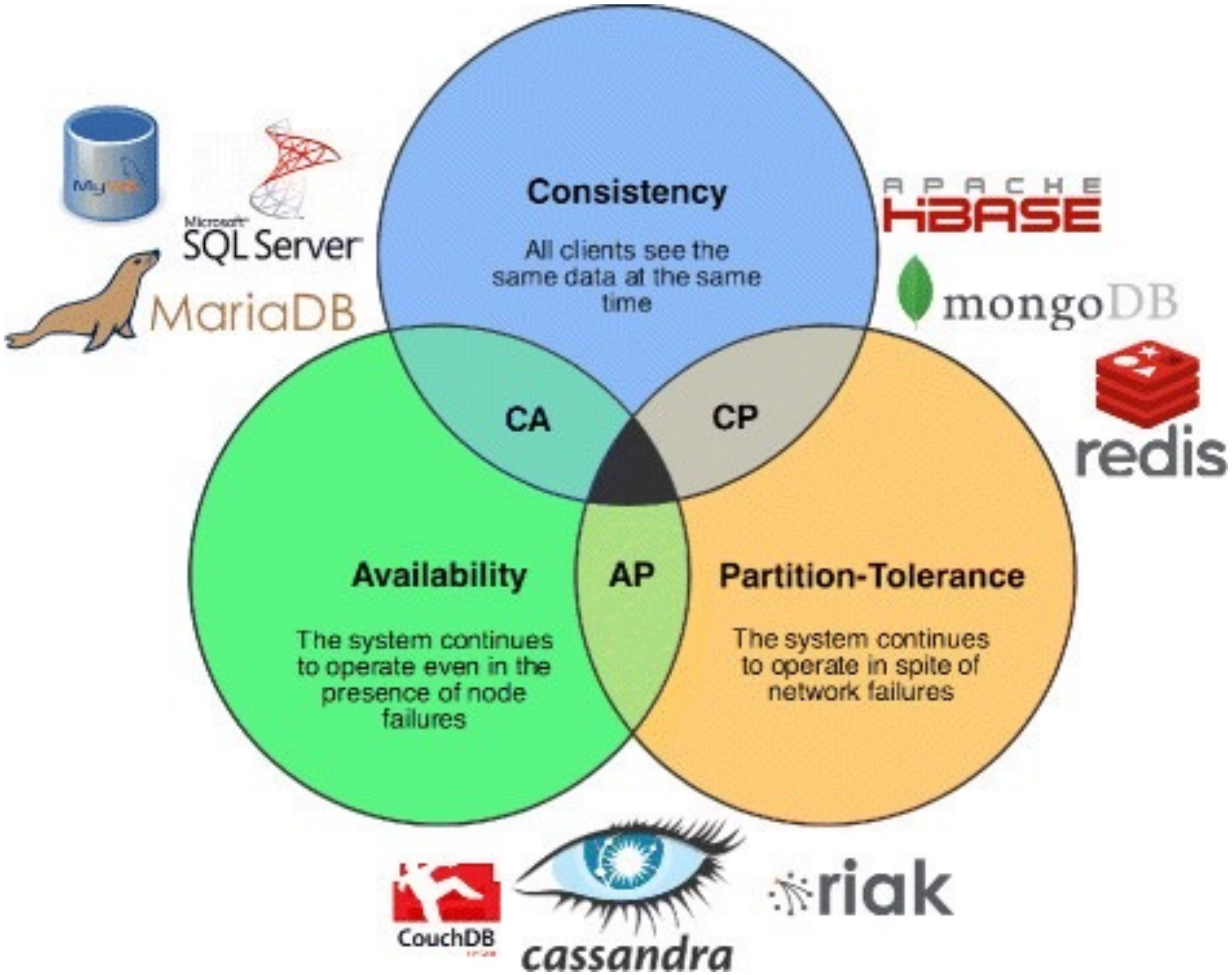
24.2 The CAP Theorem

- Consistency, availability, and partition tolerance
  - Consistency – A read from any node results in the same data across multiple nodes.
  - *Availability* – A read/write request will always be acknowledged in the form of a success or a failure
  - *Partition tolerance* – The database system can tolerate communication outages that split the cluster into multiple silos and can still service read/write requests
- Not possible to guarantee all three simultaneously
  - In distributed system with data replication

The CAP Theorem (cont'd.)

- CAP theorem demonstrates why only two of the three properties of the CAP theorem are simultaneously supportable.
  - If consistency (C) and availability (A) are required, available nodes need to communicate to ensure consistency (C). So, partition tolerance (P) is not possible.
  - If consistency (C) and partition tolerance (P) are required, nodes cannot remain available
  - If availability (A) and partition tolerance (P) are required, then consistency (C) is not possible because of the data communication requirement between the nodes.

The CAP Theorem (cont'd.)



24.3 Document-Based NoSQL Systems and MongoDB

- Document stores
  - Collections of similar documents
- Individual documents resemble complex objects or XML documents
  - Documents are self-describing
  - Can have different data elements
- Documents can be specified in various formats
  - XML
  - JSON

Document-Based NoSQL Systems

Key	Document
1001	{ "CustomerID": 99, "OrderItems": [{ "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }
1002	{ "CustomerID": 220, "OrderItems": [{ "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }

https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data

MongoDB Data Model

- Documents stored in binary JSON (BSON) format
- Individual documents stored in a collection
- Example command
  - First parameter specifies name of the collection
  - Collection options include limits on size and number of documents

```
db.createCollection("project", { capped : true, size : 1310720, max : 500 } )
```

- Each document in collection has unique ObjectID field called _id

MongoDB Data Model (cont'd.)

- A collection does not have a schema
  - Structure of the data fields in documents chosen based on how documents will be accessed
- Document creation using insert operation

```
db.<collection_name>.insert(<document(s)>)
```

- Document deletion using remove operation

```
db.<collection_name>.remove(<condition>)
```

(a) **project document with an array of embedded workers:**

```
{
      _id:                  "P1",
      Pname:                "ProductX",
      Plocation:            "Bellaire",
      Workers: [
                  { Ename: "John Smith",
                    Hours: 32.5
                  },
                  { Ename: "Joyce English",
                    Hours: 20.0
                  }
             ]
);
```

MongoDB Distributed Systems Characteristics

- Replication in MongoDB
  - Concept of replica set to create multiple copies on different nodes
  - Variation of master-slave approach
  - Primary copy, secondary copy, and arbiter
    - Arbiter participates in elections to select new primary if needed

MongoDB Distributed Systems Characteristics (cont'd.)

- Replication in MongoDB (cont'd.)
  - All write operations applied to the primary copy and propagated to the secondaries
  - User can choose read preference
    - Read requests can be processed at any replica
- Sharding in MongoDB
  - Horizontal partitioning divides the documents into disjoint partitions (shards)
  - Allows adding more nodes as needed
  - Shards stored on different nodes to achieve load balancing

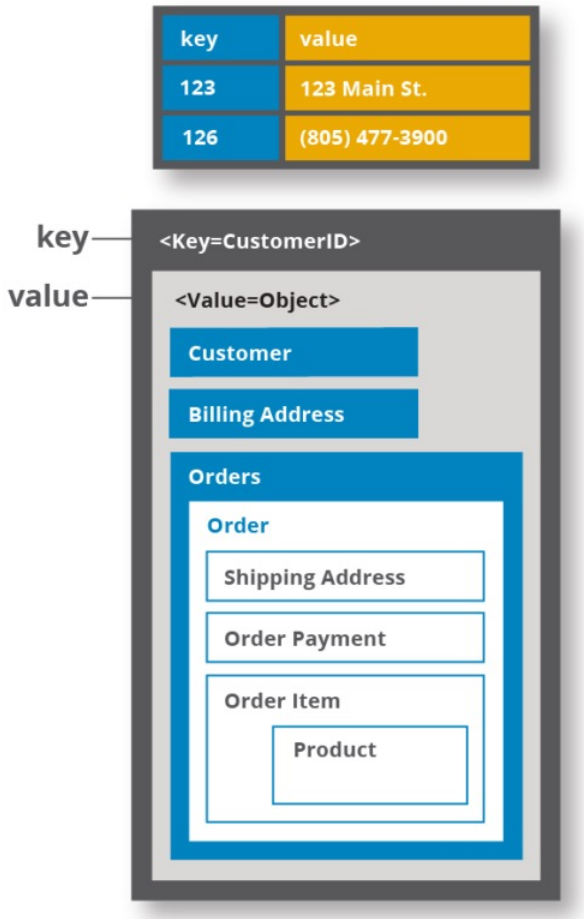
MongoDB Distributed Systems Characteristics (cont'd.)

- Sharding in MongoDB (cont'd.)
  - Partitioning field (shard key) must exist in every document in the collection
    - Must have an index
  - Range partitioning
    - Creates chunks by specifying a range of key values
  - Hash partitioning
    - Partitioning based on the hash values of each shard key

24.4 NoSQL Key-Value Stores

- Key-value stores focus on high performance, availability, and scalability
  - Can store structured, unstructured, or semi-structured data
- Key: unique identifier associated with a data item
  - Used for fast retrieval
- Value: the data item itself
  - Can be string or array of bytes
  - Application interprets the structure
- No query language

NoSQL Key-Value Stores



https://hazelcast.com/glossary/key-value-store/

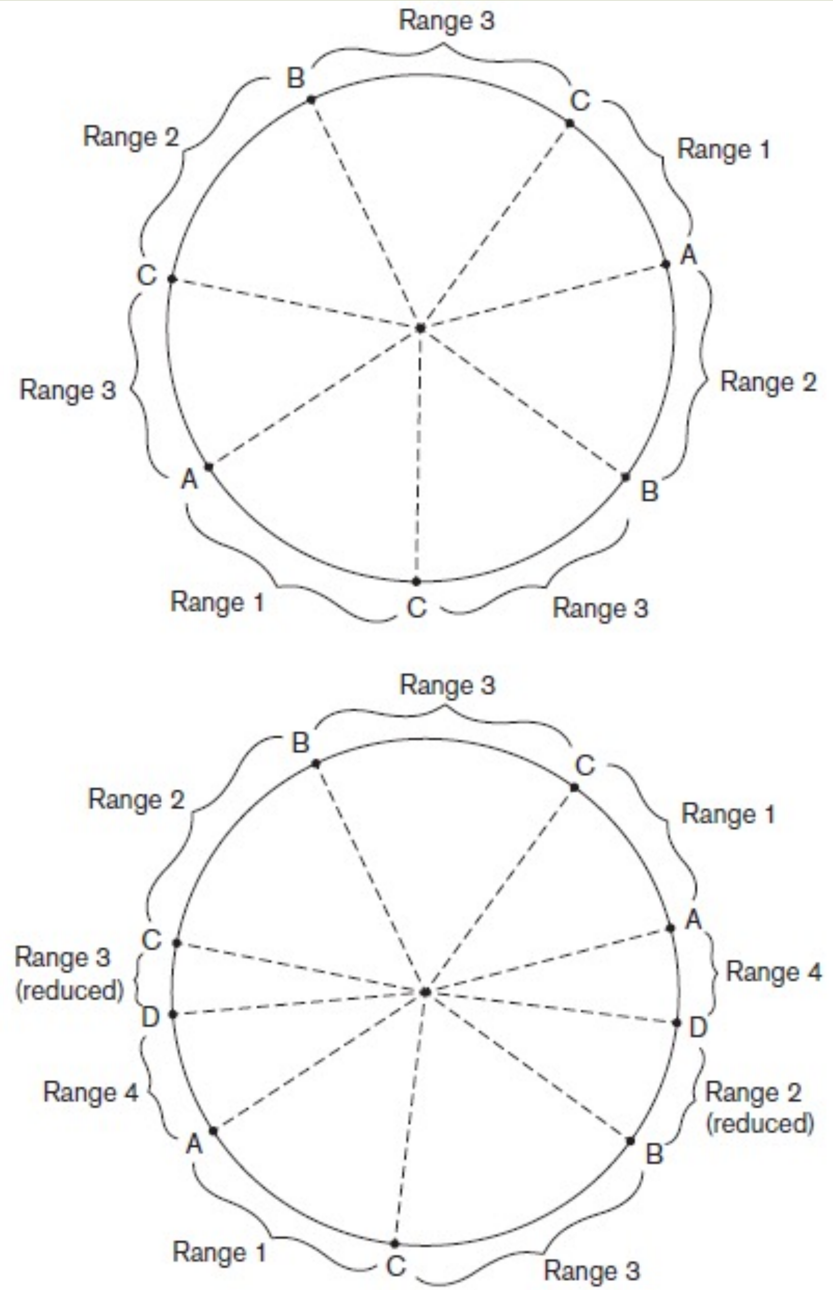
DynamoDB Overview

- DynamoDB part of Amazon's Web Services/SDK platforms
  - Proprietary
- Table holds a collection of self-describing items
- Item consists of attribute-value pairs
  - Attribute values can be single or multi-valued
- Primary key used to locate items within a table
  - Can be single attribute or pair of attributes

Voldemort Key-Value Distributed Data Store

- Voldemort: open-source key-value system similar to DynamoDB
- Voldemort features
  - Simple basic operations (get, put, and delete)
  - High-level formatted data values
  - Consistent hashing for distributing (key, value) pairs

Figure 24.2 Example of consistent hashing (a) Ring having three nodes A, B, and C, with C having greater capacity. The *h*(*K*) values that map to the circle points in *range 1* have their (k, v) items stored in node A, *range 2* in node B, *range 3* in node C (b) Adding a node D to the ring. Items in *range 4* are moved to the node D from node B (*range 2* is reduced) and node C (*range 3* is reduced)



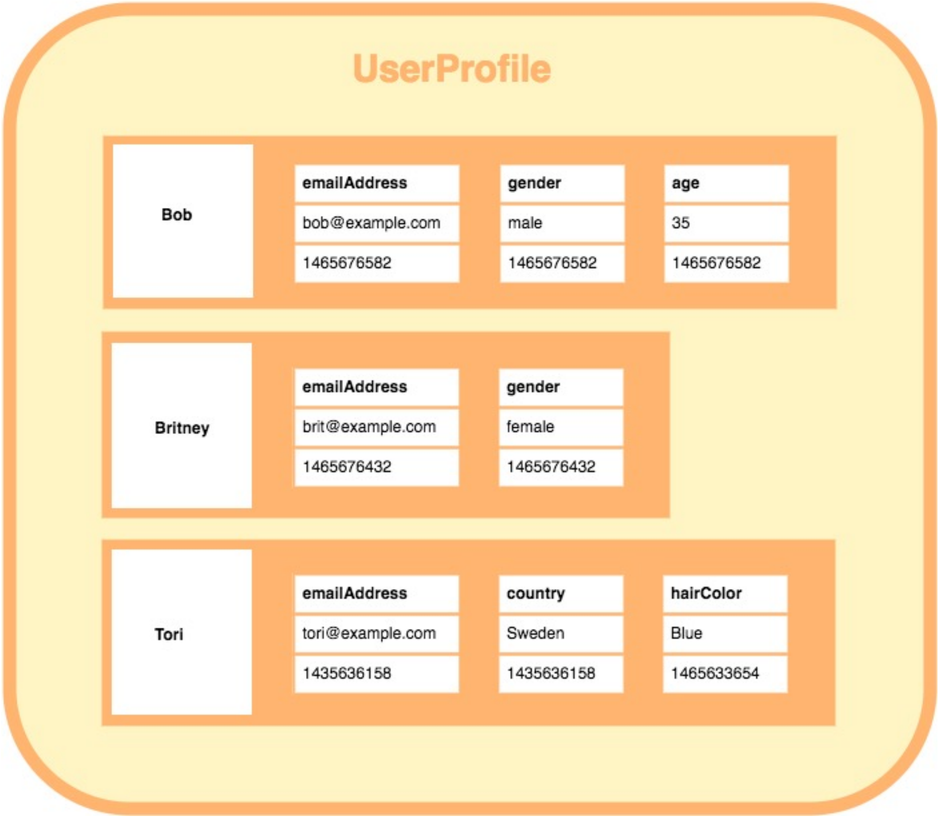
Examples of Other Key-Value Stores

- Oracle key-value store
  - Oracle NoSQL Database
- Redis key-value cache and store
  - Caches data in main memory to improve performance
  - Offers master-slave replication and high availability
  - Offers persistence by backing up cache to disk
- Apache Cassandra
  - Offers features from several NoSQL categories
  - Used by Facebook and others

24.5 Column-Based or Wide Column NoSQL Systems

- BigTable: Google's distributed storage system for big data
  - Used in Gmail
  - Uses Google File System for data storage and distribution
- Apache Hbase a similar, open-source system
  - Uses Hadoop Distributed File System (HDFS) for data storage
  - Can also use Amazon's Simple Storage System (S3)

Column-Based NoSQL Systems



https://database.guide/what-is-a-column-store-database/

Hbase Data Model and Versioning

- Data organization concepts
  - Namespaces
  - Tables
  - Column families
  - Column qualifiers
  - Columns
  - Rows
  - Data cells
- Data is self-describing

Hbase Data Model and Versioning (cont'd.)

- Namespace is collection of tables
- Table associated with one or more column families
- Column qualifiers can be dynamically specified as new table rows are created and inserted
- Each row in a table has a unique row key
- Cell holds a basic data item
- HBase stores multiple versions of data items
  - Timestamp associated with each version

(a) **creating a table:**

create 'EMPLOYEE', 'Name', 'Address', 'Details'

(b) **inserting some row data in the EMPLOYEE table:**

put 'EMPLOYEE', 'row1', 'Name:Fname', 'John'
put 'EMPLOYEE', 'row1', 'Name:Lname', 'Smith'
put 'EMPLOYEE', 'row1', 'Name:Nickname', 'Johnny'
put 'EMPLOYEE', 'row1', 'Details:Job', 'Engineer'
put 'EMPLOYEE', 'row1', 'Details:Review', 'Good'
put 'EMPLOYEE', 'row2', 'Name:Fname', 'Alicia'
put 'EMPLOYEE', 'row2', 'Name:Lname', 'Zelaya'
put 'EMPLOYEE', 'row2', 'Name:MName', 'Jennifer'
put 'EMPLOYEE', 'row2', 'Details:Job', 'DBA'
put 'EMPLOYEE', 'row2', 'Details:Supervisor', 'James Borg'
put 'EMPLOYEE', 'row3', 'Name:Fname', 'James'
put 'EMPLOYEE', 'row3', 'Name:Minit', 'E'
put 'EMPLOYEE', 'row3', 'Name:Lname', 'Borg'
put 'EMPLOYEE', 'row3', 'Name:Suffix', 'Jr.'
put 'EMPLOYEE', 'row3', 'Details:Job', 'CEO'
put 'EMPLOYEE', 'row3', 'Details:Salary', '1,000,000'

(c) **Some Hbase basic CRUD operations:**

Creating a table: create <tablename>, <column family>, <column family>, …
Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>
Reading Data (all data in a table): scan <tablename>
Retrieve Data (one item): get <tablename>,<rowid>

Figure 24.3 Examples in Hbase (a) Creating a table called EMPLOYEE with three column families: Name, Address, and Details (b) Inserting some in the EMPLOYEE table; different rows can have different self-describing column qualifiers (Fname, Lname, Nickname, Mname, Minit, Suffix, … for column family Name; Job, Review, Supervisor, Salary for column family Details). (c) Some CRUD operations of Hbase

Hbase Crud Operations

- Provides only low-level CRUD (create, read, update, delete) operations
- Application programs implement more complex operations
- Create
  - Creates a new table and specifies one or more column families associated with the table
- Put
  - Inserts new data or new versions of existing data items

Hbase Crud Operations (cont'd.)

- Get
  - Retrieves data associated with a single row
- Scan
  - Retrieves all the rows

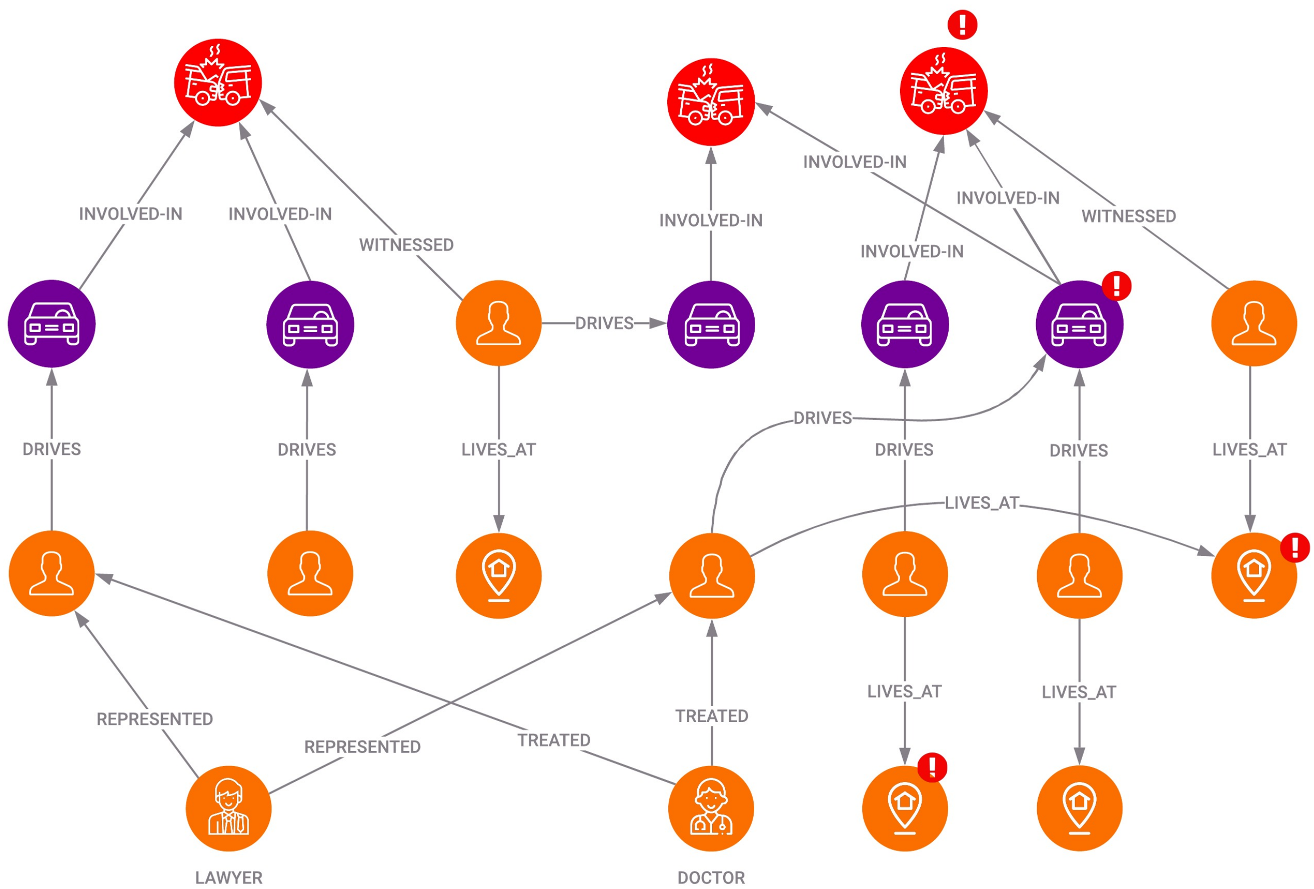
Hbase Storage and Distributed System Concepts

- Each Hbase table divided into several regions
  - Each region holds a range of the row keys in the table
  - Row keys must be lexicographically ordered
  - Each region has several stores
    - Column families are assigned to stores
- Regions assigned to region servers for storage
  - Master server responsible for monitoring the region servers
- Hbase uses Apache Zookeeper and HDFS

24.6 NoSQL Graph Databases and Neo4j

- Graph databases
  - Data represented as a graph
  - Collection of vertices (nodes) and edges
  - Possible to store data associated with both individual nodes and individual edges
- Neo4j
  - Open source system
  - Uses concepts of nodes and relationships

NoSQL Graph Databases



https://memgraph.com/blog/why-your-business-should-use-a-graph-database

Neo4j (cont'd.)

- Nodes can have labels
  - Zero, one, or several
- Both nodes and relationships can have properties
- Each relationship has a start node, end node, and a relationship type
- Properties specified using a map pattern
- Somewhat similar to ER/EER concepts

Neo4j (cont'd.)

- Creating nodes in Neo4j
  - CREATE command
  - Part of high-level declarative query language Cypher
  - Node label can be specified when node is created
  - Properties are enclosed in curly brackets

Neo4j (cont'd.)

(a) **creating some nodes for the COMPANY data (from Figure 5.6):**

```
CREATE (e1: EMPLOYEE, {Empid: '1', Lname: 'Smith', Fname: 'John', Minit: 'B'})
CREATE (e2: EMPLOYEE, {Empid: '2', Lname: 'Wong', Fname: 'Franklin'})
CREATE (e3: EMPLOYEE, {Empid: '3', Lname: 'Zelaya', Fname: 'Alicia'})
CREATE (e4: EMPLOYEE, {Empid: '4', Lname: 'Wallace', Fname: 'Jennifer', Minit: 'S'})
...
CREATE (d1: DEPARTMENT, {Dno: '5', Dname: 'Research'})
CREATE (d2: DEPARTMENT, {Dno: '4', Dname: 'Administration'})
...
CREATE (p1: PROJECT, {Pno: '1', Pname: 'ProductX'})
CREATE (p2: PROJECT, {Pno: '2', Pname: 'ProductY'})
CREATE (p3: PROJECT, {Pno: '10', Pname: 'Computerization'})
CREATE (p4: PROJECT, {Pno: '20', Pname: 'Reorganization'})
...
CREATE (loc1: LOCATION, {Lname: 'Houston'})
CREATE (loc2: LOCATION, {Lname: 'Stafford'})
CREATE (loc3: LOCATION, {Lname: 'Bellaire'})
CREATE (loc4: LOCATION, {Lname: 'Sugarland'})
...
```

Figure 24.4 Examples in Neo4j using the Cypher language (a) Creating some nodes

Neo4j (cont'd.)

(b) **creating some relationships for the COMPANY data (from Figure 5.6):**

```
CREATE (e1) – [ : WorksFor ] –> (d1)
CREATE (e3) – [ : WorksFor ] –> (d2)
...
CREATE (d1) – [ : Manager ] –> (e2)
CREATE (d2) – [ : Manager ] –> (e4)
...
CREATE (d1) – [ : LocatedIn ] –> (loc1)
CREATE (d1) – [ : LocatedIn ] –> (loc3)
CREATE (d1) – [ : LocatedIn ] –> (loc4)
CREATE (d2) – [ : LocatedIn ] –> (loc2)
...
CREATE (e1) – [ : WorksOn, {Hours: '32.5'} ] –> (p1)
CREATE (e1) – [ : WorksOn, {Hours: '7.5'} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p1)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p3)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p4)
...
```

Figure 24.4 (cont'd.) Examples in Neo4j using the Cypher language
(b) Creating some relationships

Neo4j (cont'd.)

- Path
  - Traversal of part of the graph
  - Typically used as part of a query to specify a pattern
- Schema optional in Neo4j
- Indexing and node identifiers
  - Users can create for the collection of nodes that have a particular label
  - One or more properties can be indexed

The Cypher Query Language of Neo4j

- Cypher query made up of clauses
- Result from one clause can be the input to the next clause in the query

The Cypher Query Language of Neo4j (cont'd.)

(c) **Basic simplified syntax of some common Cypher clauses:**

Finding nodes and relationships that match a pattern: MATCH <pattern>
Specifying aggregates and other query variables: WITH <specifications>
Specifying conditions on the data to be retrieved: WHERE <condition>
Specifying the data to be returned: RETURN <data>
Ordering the data to be returned: ORDER BY <data>
Limiting the number of returned data items: LIMIT <max number>
Creating nodes: CREATE <node, optional labels and properties>
Creating relationships: CREATE <relationship, relationship type and optional properties>
Deletion: DELETE <nodes or relationships>
Specifying property values and labels: SET <property values and labels>
Removing property values and labels: REMOVE <property values and labels>

Figure 24.4 (cont'd.) Examples in Neo4j using the Cypher language
(c) Basic syntax of Cypher queries

The Cypher Query Language of Neo4j (cont'd.)

Figure 24.4 (cont'd.) Examples in Neo4j using the Cypher language (d) Examples of Cypher queries

(d) Examples of simple Cypher queries:

1. MATCH (d : DEPARTMENT {Dno: '5'}) – [: LocatedIn] → (loc)
   RETURN d.Dname , loc.Lname
2. MATCH (e: EMPLOYEE {Empid: '2'}) – [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
3. MATCH (e) – [w: WorksOn] → (p: PROJECT {Pno: 2})
   RETURN p.Pname, e.Ename , w.Hours
4. MATCH (e) – [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
5. MATCH (e) – [w: WorksOn] → (p)
   RETURN e.Ename , w.Hours, p.Pname
   ORDER BY e.Ename
   LIMIT 10
6. MATCH (e) – [w: WorksOn] → (p)
   WITH e, COUNT(p) AS numOfprojs
   WHERE numOfprojs > 2
   RETURN e.Ename , numOfprojs
   ORDER BY numOfprojs
7. MATCH (e) – [w: WorksOn] → (p)
   RETURN e , w, p
   ORDER BY e.Ename
   LIMIT 10
8. MATCH (e: EMPLOYEE {Empid: '2'})
   SET e.Job = 'Engineer'

Neo4j Interfaces and Distributed System Characteristics

- Enterprise edition versus community edition
  - Enterprise edition supports caching, clustering of data, and locking
- Graph visualization interface
  - Subset of nodes and edges in a database graph can be displayed as a graph
  - Used to visualize query results
- Master-slave replication
- Caching
- Logical logs

24.7 Summary

- NoSQL systems focus on storage of “big data”
- General categories
  - Document-based
  - Key-value stores
  - Column-based
  - Graph-based
  - Some systems use techniques spanning two or more categories
- Consistency paradigms
- CAP theorem