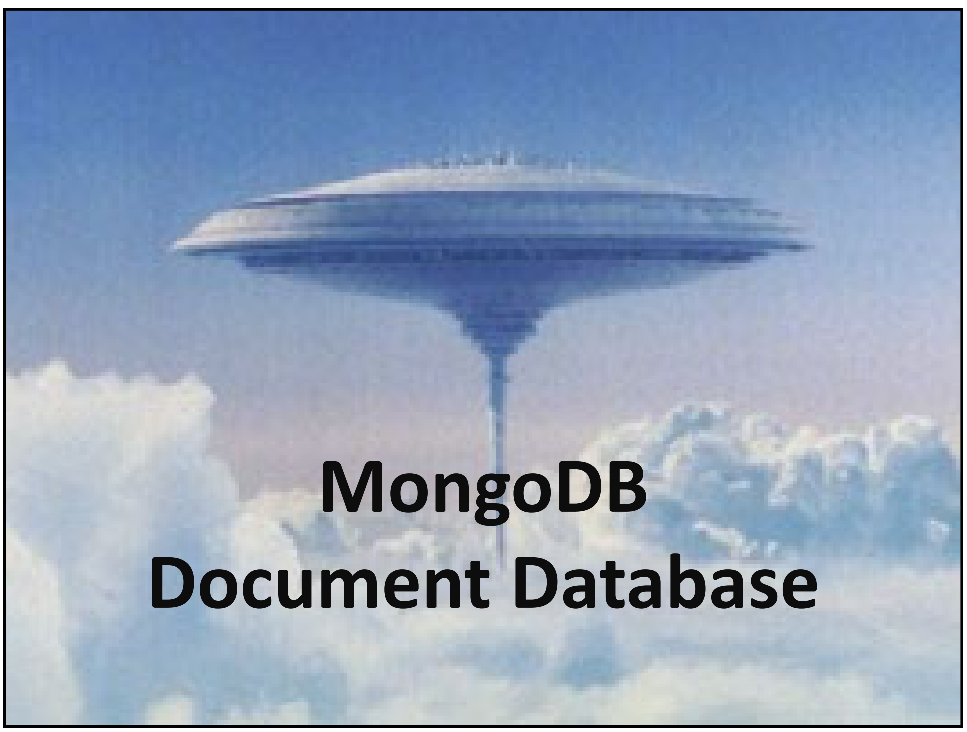
Manage Big Data in the Cloud -- Different NoSQL DBs



MongoDB

- NoSql systems
  - Most of those in the wide column family and key-value store categories have similar designs
  - Document based NoSql has some new data model concept
    - Similar to old time "object-oriented" database model
- Applications
  - Content management systems
  - Product management systems
  - …

MongoDB – Data Model

- Semantics
  - MongoDB = a set of databases
  - Database = a set of collections
  - Collection = a set of documents
    - With similar data fields, but still highly flexible
  - Documents
    - A set of attributes and their values
    - Use BSON format (Binary JSON)
    - Support nesting: Documents can include documents
    - Document example (nested):
      - { "name" : "Alice Anderson", "SSN" : "123456789",
        "member" : "UTD", "member" : "IEEE",
        "address" : { "street" : "800 W. Campbell",
        "city" : "Richardson", "state" : "TX" } }

Can have any #values for the same field

Can have nested documents
⇒ Offer better semantics

MongoDB – Data Model

- Semantics
  - Relationship
    - Embedded relationship
      - { “name” : “Alice Anderson”, “SSN” : “123456789”,
        “address” : [{ “street” : “a”, “city” : “b”, “state” : “c” }
        { “street” : “d”, “city” : “e”, “state” : “f” }] }
      - Each entry in [ ] is a document, [ ] embeds the entries
    - Reference relationship
      - Similar to RDB foreign key
      - { “Id” : ObjectId(“12345”), “name” : “Alice Anderson”, “SSN” : “…”,
        “address” : [ObjecId(“13579”), ObjectId(“24680”)]
    - Objects within the relationship definition are first class objects

MongoDB – Data Model

- Query
  - Can lookup by any keyword defined in the document
    - E.g., db.users.find ({“name” : “Alice”, “age” : {“$gte”, 18}})
  - Can specify what fields to return
    - E.g., db.users.find ({…}, {“email”:1, “birth-date”:1})
      - 0 means not to return the corresponding field
  - Can specify the subfields for lookup
    - E.g., db.users.find ({“address” : {“state” : “Texas”}})

users

name	email	phone	address	age	birth	…

MongoDB – Data Model

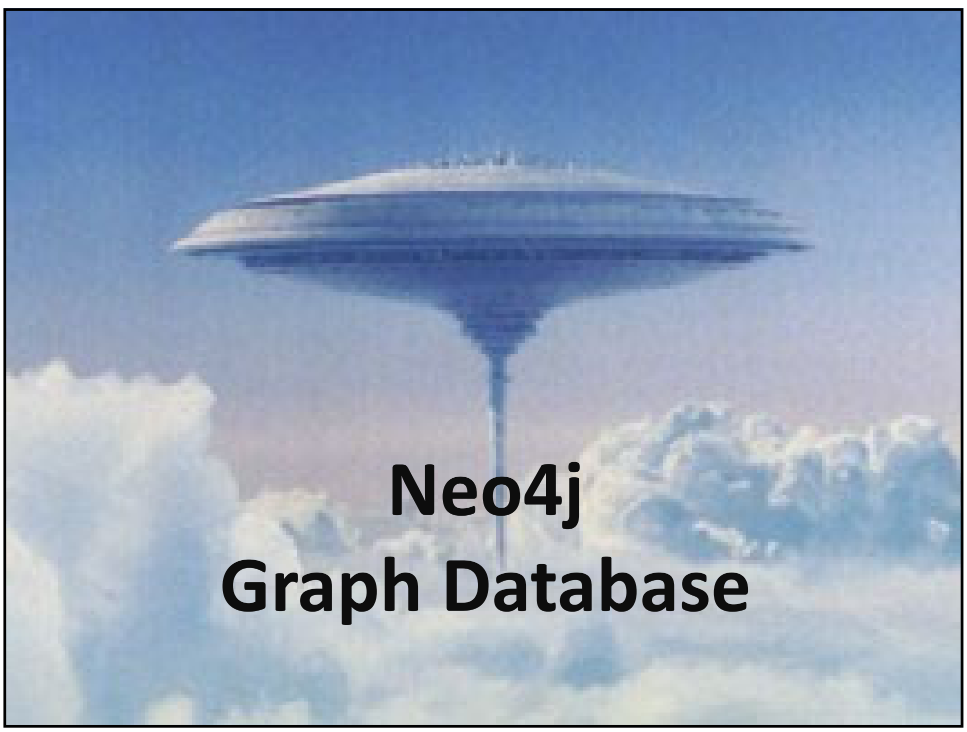
- Indexing
  - Index is created for efficient search
    - Data structure: B-tree
  - User can specify the field to be indexed
    - E.g., db.users.ensureIndex ({"name":1, "age":1})
  - Can index the subfields also
    - E.g., db.users.ensureIndex ({"address.state":1})
- Indexes
  - Cached in memory and stored on disk
  - Stored in a shard unit

MongoDB – Consistency

- Support multiple consistency models
    - But not as many as Cassandra
  - Passive replication with primary and backup
  - Strong consistency for writes
    - Can have a set of primary nodes (primary set)
    - Update is sent to all nodes in the primary set
    - If a primary fails, a secondary can be elected to be the primary
    - Atomicity is guaranteed only for an individual document, not cross documents
  - Reads
    - If strong consistency is desired, then read from the primary
    - If only require eventual consistency, then read from any

MongoDB – Sharding

- Range based sharding
  - Similar to GBT
  - Each chunk contains a key range
- Centralized shard distribution
  - Mongo is the query router
  - Config server stores the centralized metadata and configuration settings for the system

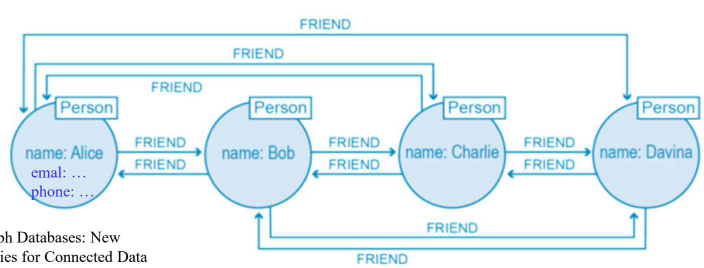


Graph Databases

- Why graph database
  - Applications
    - The web, the Linked Data, social networks, etc.
    - The RDF database
    - Many data are highly linked: bio data, chemical data, …
  - Why are relational and nosql databases not sufficient?
    - They can capture all the relations
    - But query processing can be very slow
    - E.g., in a social network database
      - Find all Alice's friends: can be done efficiently in conventional DBs
      - Final all people having Alice as a friend: slow query processing

Graph Database Models

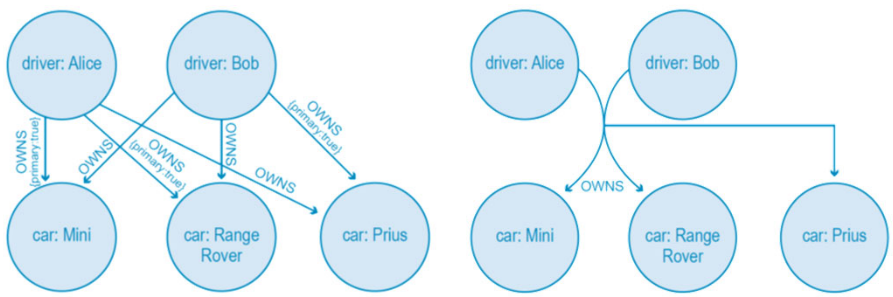
- Neo4j https://neo4j.com/docs/getting-started/current/ The community version is free
  - Model: (Nodes, relationships) ← (labels and properties)
  - Example
    - Node with properties: (name:Alice, email:a@utd.edu, …)
    - label for nodes: person A node can have multiple labels
      - Labels can support grouping, etc.
    - Relationships can also have properties



From: Graph Databases: New Opportunities for Connected Data

Graph Database Models

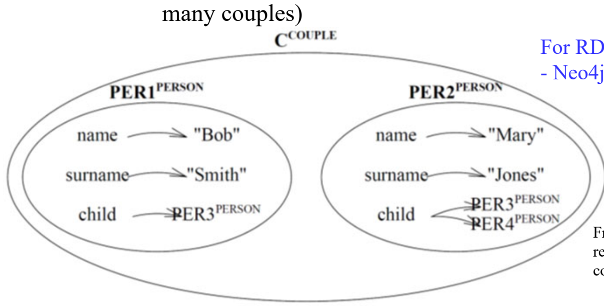
- General graph database models
  - Simple graph model: (starting node, relation, ending node)
    - E.g., Neo4j
  - Hypergraph model: M to N relation
    - A couple owns three cars ⇒ 6 relations ⇒ 1 hyper-relation



From: Graph Databases: New Opportunities for Connected Data

Graph Database Models

- General graph database models
  - Nested graph model
    - Each node may also be a graph
    - Example 1
      - Neo4j: use properties for each person (better)
        - » Add couple partner as a property
      - Neo4j: use labels for couples (may not be ideal when there are too many couples)



For RDF model ⇒ nesting helps!
- Neo4j does criticize RDF model

From: A nested-graph model for the representation and manipulation of complex objects

Graph Database Models

- General graph database models
  - Nested graph model

    For this example, nesting is better!

    - Example 2: Activities in a company

      If no grouping, how many edges will there be?

      - Inter-division, company wide
        - » E.g., benefit selection to everyone, company wide gatherings
        - » E.g., hiring request from engineering to HR and back
      - Inter-group in a division
        - » E.g., in the software engineering group: system group with database group (relation: design meeting, software delivery), database group with QA group (relation: delivery for testing)
      - Intra group
        - » E.g., in the software engineering group: group meetings, manager to group member evaluations, member to member discussions for design, code integration, etc.
      - Other
        - » Individuals cross group/division activities

Graph Database Queries

- Neo4j

  - https://neo4j.com/docs/cypher-manual/current/clauses/match/
  - Graph Databases: New Opportunities for Connected Data

  - Match and Return clauses
  - Node finding
    - MATCH (movie:Movie) RETURN movie.title
      - Return movie.title field from all nodes with label “Movie”
    - MATCH (director {name: ‘xxx'})--(movie) RETURN movie.title
      - Return movie.title from all nodes with property “director.name=xxx”
    - Node finding can be handled similarly with relational database
  - Relation finding
    - MATCH (:Person {name: ‘xxx'})-[r]->(movie) RETURN type(r)
      - Returns the types of the outgoing relations from “person.name=xxx”
      - E.g., acted, directed

Graph Database Queries

- Neo4j
  - Path based match
    - MATCH (a:Person)-[:Knows]->(b)-[:Friend]->(c), WHERE a.name = ‘xxx’ RETURN b, c
      - Returns all b and c where b knows xxx and c is a friend of b
    - MATCH (Movie {title: ‘xxx'})-[*0..1]-(x) RETURN x
      - Returns all movies with title xxx (path length=0) and all the nodes that are 1 hop away with the matching nodes (path length=1)
    - MATCH p = (p1:Person)-[*]-(p2:Person) WHERE p1.name = ‘xxx’ AND p2.name = ‘yyy’ RETURN p
      - Returns the entire path between nodes p1 and p2 with the specific name property (p1.name=xxx, p2.name=yyy)

Graph Database Queries

- General graph database queries
  - Subgraph query
    - Search for a specific pattern in a graph
    - Some part of the pattern may be uncertain
    - Return the graphs that contains the pattern and has the match
    - Path based match in Neo4j has some similarities to this
  - Supergraph query
    - Search in a graph database {g1, g2, …} and find all graphs that are contained in the input graph given by the query
  - Similarity queries
    - Instead of exact match, consider similarity based matching
    - Similar in node names/properties, relationship names/properties
    - Similar in the graph pattern, not exactly isomorphic graphs

Graph Database Storage

- Graph database storage
  - Most existing works still use relational DB tables
    - E.g., RDF databases
- Neo4j graph storage
  - Use native graph storage(?), but still different from graph itself
  - Storages for nodes, relationships, properties, labels, …
  - Nodes
    - In use: if not in use (e.g., deleted), can be reclaimed
    - Pointers to properties, labels, relationships (4 bytes)

In-use	Relations ↓	Properties ↓	Labels ↓	flags

  - Relationships
    - Relationship type pointer: to relationship type table
    - Starting/Ending node pointers: to node table
    - Doubly linked list (previous/next relation for the node)

Relation type ↓	Start node ↓	End node ↓	pre ↓	next ↓

Graph Database Storage

- Graph database storage
- Neo4j graph storage

Type ↓	Key	Value (↓)	next ↓
Type ↓	Key	Value (↓)	
Type ↓	Key	Value (↓)	
Type ↓	Key	Value (↓)	

  - Properties
    - Each record can have 4 properties
      - » Each is a key-value entry
      - » Each have inline property value or a pointer to property value store (if the value is large)
    - Singly linked list to next properties of the same node (> 4 properties)
  - Labels: no discussion, can be similar to property
  - Node, relationship, property stores: have fixed sized records
    - Pointers are like the array indexes to the records in the table (ids)
  - Match: n: {property} – r: {property} –> m
    - Property table: find the property id specified for n and r (starting id)
    - Node table: find the node ids of nodes with the property id
    - Relation table: find matching starting node ids and relationship ids

Graph Database Indexing Techniques

- Neo4j
  - Some tables are like indexes
- Other, general
  - Tables for nodes
    - With detailed properties and labels
    - With incoming and outgoing edges
    - Each edge can have corresponding node pointers (other nodes)
  - Indexing: Keyword based inverted indexing
    - A table with all keywords in the database
    - Points to nodes (in any field)
  - Pattern matching
    - Match all keywords for nodes and find matching nodes
    - From these nodes, find matching relationships give in the pattern

Distributed Graph Databases

- Neo4j https://neo4j.com/docs/operations-manual/current/clustering/introduction/
    - Only available in the enterprise version
  - Mainly consider replication for availability
    - A cluster may host multiple graphs, which can be hosted by different nodes, but a single graph is not partitioned
      - Still investigating partitioning techniques for a single graph
    - Primary and secondary hosts
  - Consistency
    - Consider causal consistency
      - A client should see the latest updates it knows about
    - Client: issues an update → gets back a bookmark → issues subsequent r/w requests with the bookmark
    - Server: processes the r/w request only if the bookmarked update has been completed

Distributed Graph Databases

- Pregel
  - MapReduce on graph databases, by Google
  - Partitioning
    - Graph DB is a collection of vertices, each with a unique ID
    - Each vertex is hashed to a node by hash(ID) % N
      - N is the number of storage nodes
  - Computation
    - Implement supports for functions that are similar to MapReduce
      - Pass messages (shuffling)
      - Aggregation (reduce)
      - …

Distributed Graph Databases

- Sedge
  - Mainly focuses on graph database partitioning
  - Computation steps are based on Pregel
  - Partitioning
    - Principle: minimize communication cost during query processing
    - Partitioned by graph minimal cut
    - Repartitioning
      - When there are many cross-partition accesses ⇒ Repartition
      - Partitioning is evolving continuously
  - Two level partition management
    - Primary partitioning: partitions are disjoint
      - One vertex can only belong to one partition
    - Secondary partitioning: balance workloads

Distributed Graph Databases

- Sedge
  - On demand partitioning
      - Manage partitions to achieve the secondary partitioning principle
    - Partition replication
      - When a partition becomes a hotspot, replicate it
        - Should be referring to query processing without updates (write to all, so no benefit to replicate)
    - Dynamic partitioning
      - Decide new optimal partitioning based on historical queries
        - Without considering original partitioning
      - Apply the new partitioning result
        - ??? Too much data migration

Distributed Graph Databases

- Sedge
  - Dynamic partitioning
    - Need to track query processing traces (vertices and edges visited) ⇒ Too much information to track
    - Group vertices in k hops (k=1 has the best effect)
      - Partition the graph into disjoint groups, radius of each group is k
      - Choose one vertex, put all its neighbors within k hops in a group
      - Give the group a color (or just an ID)
    - Query history
      - Track the colors that each query traversed
      - All the colors of a query form an envelop
        - E.g., query Q1, Q2 has colors {c1, c3, c5}, {c1, c2, c8}
      - Recording visited groups instead of vertices ⇒ space/time efficient

Distributed Graph Databases

- Sedge
  - Dynamic partitioning
    - Cluster the envelops (queries) based on their similarity
      - Jaccard similarity = $|Q1 \cap Q2| / |Q1 \cup Q2|$
      - E.g., similarity of {c1, c2, c3, c5} and {c1, c3, c5, c7} is 3/5
      - ⇒ Each cluster includes a set of colors, which is the union of all the colors of all the queries in the cluster
    - Rank the clusters by ρ = (#queries) / (#colors)
      - Access frequency for each color
      - Sort the clusters by ρ
    - Greedy algorithm for partitioning

      Pack similar clusters ⇒ More groups, less vertices
      ⇒ Because similar groups has more overlaps
      - Each partition is of size S (predetermined)
      - Pack as many clusters (largest ρ first) to the current partition
        - » I.e., add the colors (groups of vertices) in the cluster to the partition
      - Switch to the next partition when the current partition is filled

References

- MongoDB
  - Kristina Chodorow, Michael Dirolf. “MongoDB: The Definitive Guide”, O’Reilly Media, Sep. 2010
- Graph DBs
  - Graph Databases: New Opportunities for Connected Data (Neo4j)
  - Pregel: A system for large-scale graph processing (Pregel)
  - Towards effective partition management for large graphs (Sedge)
  - An introduction to graph data management (models & queries)
  - Graph indexing and querying: A review (indexing techniques)