A Document Database is a non-relational database that stores data as structured documents, usually in JSON. They are designed to be simple, flexible, and scalable

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write.It is easy for machines to parse and generate.
JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is operationalized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is operationalized as an array, vector, list, or sequence.

These are two universal data structures supported by virtually all modern programming languages Thus, JSON makes a great data interchange format.

BSON is Binary JSON which is a binary-encoded serialization of a JSON-like document structure. It supports extended types not part of basic JSON (e.g. Date, BinaryData, etc)
Features of BSON:

- Lightweight - keep space overhead to a minimum
- Traversable - designed to be easily traversed, which is vitally important to a document DB
- Efficient - encoding and decoding must be efficient
- Supported by many modern programming languages

XML (eXtensible Markup Language)

- Precursor to JSON as data exchange format
- XML + CSS → web pages that separated content and formatting
- Structurally similar to HTML, but tag set is extensible
- Xpath - a syntax for retrieving specific elements from an XML doc
- Xquery - a query language for interrogating XML documents; the SQL of XML
- DTD - Document Type Definition - a language for describing the allowed structure of an XML document
- XSLT - eXtensible Stylesheet Language Transformation - tool to transform XML into other formats, including non-XML formats such as HTML.

Document databases address the impedance mismatch problem between object persistence in OO systems and how relational DBs structure data.The structure of a document is self-describing. They are well-aligned with apps that use JSON/XML as a transport layer

- OO Programming includes Inheritance and Composition of types.

How do we save a complex object to a relational database? We basically have to deconstruct it.

MongoDB: Started in 2007 after Doubleclick was acquired by Google, and 3 of its veterans realized the limitations of relational databases for serving > 400,000 ads per second
MongoDB was short for Humongous Database
MongoDB Atlas released in 2016 → documentdb as a service

Mongo DB Documents: No predefined schema for documents is needed
Every document in a collection could have different data/schema

RDBMS	MongoDB
Database	Database
Table/View	Collection
Row	Document
Column	Field
Index	Index
Join	Embedded Document
Foreign Key	Reference

MongoDB features

- Rich Query Support - robust support for all CRUD ops
- Indexing - supports primary and secondary indices on document fields
- Replication - supports replica sets with automatic failover
- Load balancing built in

MongoDB version:

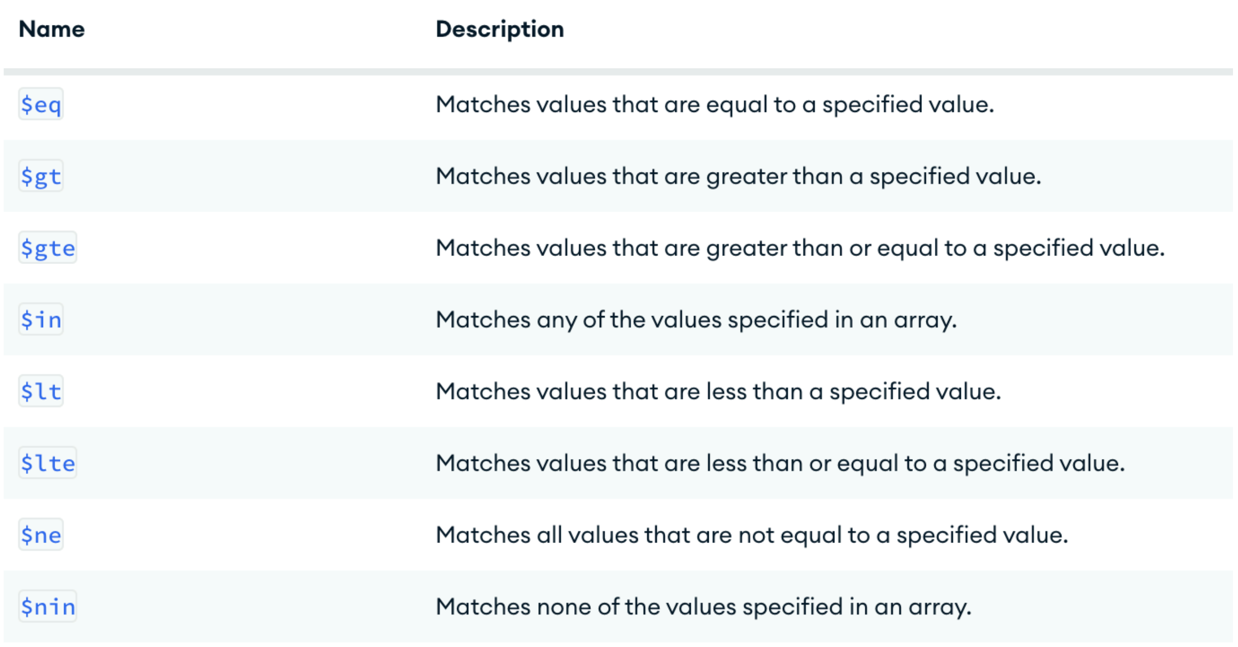
- MongoDB Atlas: Fully managed MongoDB service in the cloud (DBaaS)
- MongoDB Enterprise: Subscription-based, self-managed version of MongoDB
- MongoDB Community: source-available, free-to-use, self-managed

Interacting with MongoDB:

- mongosh → MongoDB Shell: CLI tool for interacting with a MongoDB instance
- MongoDB Compass: free, open-source GUI to work with a MongoDB database
- DataGrip and other 3rd Party Tools
- Every major language has a library to interface with MongoDB: PyMongo (Python), Mongoose (JavaScript/node), …

find(...) is like SELECT

- SELECT * FROM users WHERE name = “Davos Seaworth”; is same as db.users.find({"name": "Davos Seaworth"})
- SELECT * FROM movies WHERE rated in ("PG", "PG-13") is same as db.movies.find({rated: {$in:["PG", "PG-13"]}})
- Return movies which were released in Mexico and have an IMDB rating of at least 7: db.movies.find({ "countries": "Mexico", "imdb.rating": { $gte: 7 } })
- Return movies from the movies collection which were released in 2010 and either won at least 5 awards or have a genre of Drama: db.movies.find({ “year”: 2010, $or: [{ "awards.wins": { $gte: 5 } }, { “genres”: "Drama" }] })

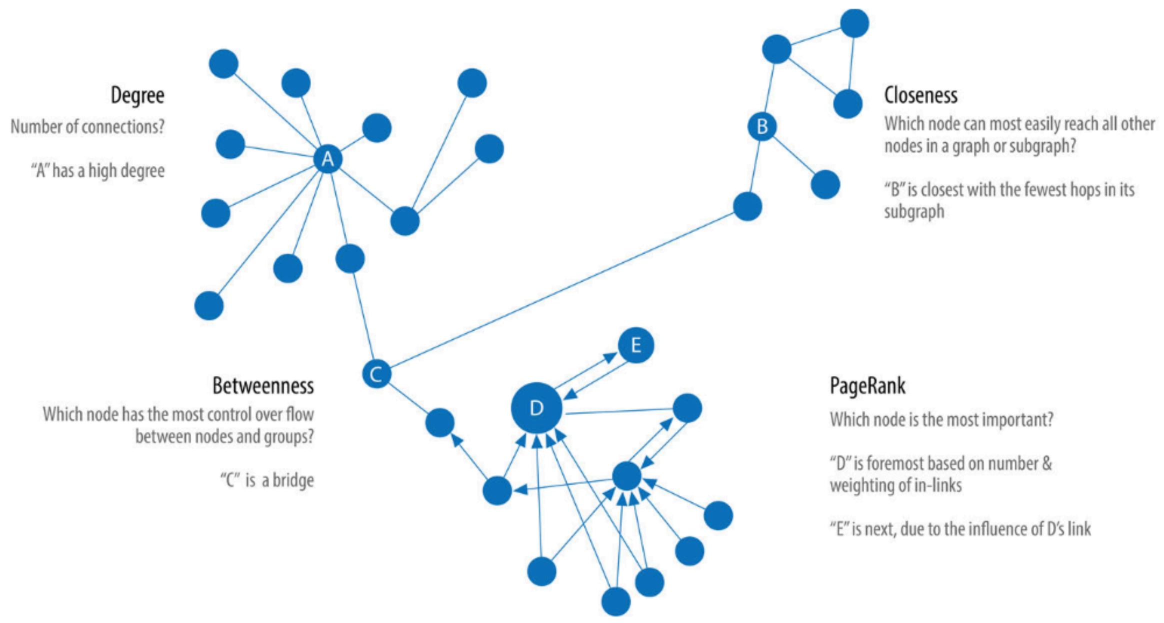


Name	Description
`$eq`	Matches values that are equal to a specified value.
`$gt`	Matches values that are greater than a specified value.
`$gte`	Matches values that are greater than or equal to a specified value.
`$in`	Matches any of the values specified in an array.
`$lt`	Matches values that are less than a specified value.
`$lte`	Matches values that are less than or equal to a specified value.
`$ne`	Matches all values that are not equal to a specified value.
`$nin`	Matches none of the values specified in an array.

-
- How many movies from the movies collection were released in 2010 and either won at least 5 awards or have a genre of Drama: db.movies.countDocuments({“year”: 2010, $or: [{ "awards.wins": { $gte: 5 } }, { “genres”: "Drama" }] })
- Return the names of all movies from the movies collection that were released in 2010 and either won at least 5 awards or have a genre of Drama: db.movies.countDocuments({“year”: 2010,$or: [{ "awards.wins": { $gte: 5 } }, { “genres”: "Drama" }] }, {“name”: 1, “_id”: 0})
- PyMongo is a Python library for interfacing with MongoDB instances from pymongo import MongoClient
- client = MongoClient(‘mongodb://user_name:pw@localhost:27017’)
- db = client[‘ds4300’]
- collection = db[‘myCollection’]
- post = {
-         “author”: “Mark”,
-         “text”: “MongoDB is Cool!”,
-         “tags”: [“mongodb”, “python”]
- }
-
- post_id = collection.insert_one(post).inserted_id
- print(post_id)
- Count Documents in Collection: demodb.collection.count_documents({})
- Find all movies from 2000: from bson.json_util import dumps
-
- # Find all movies released in 2000
- movies_2000 = db.movies.find({"year": 2000})
-
- # Print results
- print(dumps(movies_2000, indent = 2))
-
- What is a Graph Database: Data model based on the graph data structure. Composed of nodes and edges. edges connect nodes. each is uniquely identified. each can contain

properties (e.g. name, occupation, etc). supports queries based on graph-oriented operations. Traversals. shortest path, etc.

- Where do Graphs Show up?
- Social Networks yes… things like Instagram, but also… modeling social interactions in fields like psychology and sociology
- The Web: it is just a big graph of "pages" (nodes) connected by hyperlinks (edges)
- Chemical and biological data: systems biology, genetics, etc. interaction relationships in chemistry
- Labeled Property Graph: Composed of a set of node (vertex) objects and relationship (edge) objects. Labels are used to mark a node as part of a group. Properties are attributes (think KV pairs) and can exist on nodes and relationships. Nodes with no associated relationships are OK. Edges not connected to nodes are not permitted.
- A path is an ordered sequence of nodes connected by edges in which no nodes or edges are repeated. Ex: 1 → 2 → 6 → 5 , Not a path: 1 → 2 → 6 → 2 → 3
- Connected (vs. Disconnected) – there is a path between any two nodes in the graph
- Weighted (vs. Unweighted) – edge has a weight property (important for some algorithms)
- Directed (vs. Undirected) – relationships (edges) define a start and end node
- Acyclic (vs. Cyclic) – Graph contains no cycles
- Trees:
- Rooted tree- root node and no cycles
- Binary tree- up to 2 child nodes and no cycles
- spanning tree: subgraph of all nodes but not all relationships and no cycles
- Types of graph algorithms:
- Pathfinding: finding the shortest path between two nodes, if one exists, is probably the most common operation
    - "shortest" means fewest edges or lowest weight
    - Average Shortest Path can be used to monitor efficiency and resiliency of networks.
    - Minimum spanning tree, cycle detection, max/min flow… are other types of pathfinding
- Breadth first search: visits nearest neighbors first
- Depth first search: walks down each branch first
- Centrality
    - determining which nodes are "more important" in a network compared to other nodes
    - EX: Social Network Influencers?
- Community Detection
    - evaluate clustering or partitioning of nodes of a graph and tendency to strengthen or break apart


Degree
Number of connections?
"A" has a high degree
Closeness
Which node can most easily reach all other nodes in a graph or subgraph?
"B" is closest with the fewest hops in its subgraph
Betweenness
Which node has the most control over flow between nodes and groups?
"C" is a bridge
PageRank
Which node is the most important?
"D" is foremost based on number & weighting of in-links
"E" is next, due to the influence of D's link
A
B
C
D
E

-
-
- Dijkstra's Algorithm - single-source shortest path algo for positively weighted graphs
- A* Algorithm - Similar to Dijkstra's with added feature of using a heuristic to guide traversal
- PageRank - measures the importance of each node within a graph based on the number of incoming relationships and the importance of the nodes from those incoming relationships
- Neo4j: A Graph Database System that supports both transactional and analytical processing of graph-based data
    - Relatively new class of no-sql DBs
    - Considered schema optional (one can be imposed)
    - Supports various types of indexing
    - ACID compliant
    - Supports distributed computing
    - Similar: Microsoft CosmoDB, Amazon Neptune
- Neo4j - Query Language and Plugins
- Cypher
    - Neo4j's graph query language created in 2011
    - Goal: SQL-equivalent language for graph databases
    - Provides a visual way of matching patterns and relationships (nodes)-[:CONNECT_TO]->(otherNodes)
- APOC Plugin
    - Awesome Procedures on Cypher
    - Add-on library that provides hundreds of procedures and functions

Graph Data Science Plugin

- provides efficient implementations of common graph algorithms (like the ones we talked about yesterday)
- Neo4j in Docker Compose: Supports multi-container management.
    - Set-up is declarative - using YAML docker-compose.yaml file:
    - services
    - Volumes
    - networks, etc.

- 1 command can be used to start, stop, or scale a number of services at one time.
- Provides a consistent method for producing an identical environment (no more “well… it works on my machine!)
- Interaction is mostly via command line
- .env files - stores a collection of environment variables, good way to keep environment variables for different platforms separate
- Inserting Data by Creating Nodes:
- CREATE (:User {name: "Alice", birthPlace: "Paris"})
- CREATE (:User {name: "Bob", birthPlace: "London"})
- CREATE (:User {name: "Carol", birthPlace: "London"})
- CREATE (:User {name: "Dave", birthPlace: "London"})
- CREATE (:User {name: "Eve", birthPlace: "Rome"})
-
- Adding an Edge with No Variable Names
-
- MATCH (alice:User {name:”Alice”})
- MATCH (bob:User {name: “Bob”})
- CREATE (alice)-[:KNOWS {since: “2022-12-01”}]->(bob)
- Note: Relationships are directed in neo4j.
- Which users were born in London?
-
- MATCH (usr:User {birthPlace: “London”})
- RETURN usr.name, usr.birthPlace
-
- Loading CSVs - General Syntax
- LOAD CSV
- [WITH HEADERS]
- FROM 'file:///file_in_import_folder.csv'
- AS line
- [FIELDTERMINATOR ',']
- // do stuffs with 'line'
-
- Importing with Directors this Time
- LOAD CSV WITH HEADERS
- FROM 'file:///netflix_titles.csv' AS line
- WITH split(line.director, ",") as directors_list
- UNWIND directors_list AS director_name
- CREATE (:Person {name: trim(director_name)})
- But this generates duplicate Person nodes (a director can direct
- more than 1 movie)
- Importing with Directors Merged
- MATCH (p:Person) DELETE p
-
- LOAD CSV WITH HEADERS
- FROM 'file:///netflix_titles.csv' AS line
- WITH split(line.director, ",") as directors_list
- UNWIND directors_list AS director_name
- MERGE (:Person {name: director_name})
- Adding Edges: LOAD CSV WITH HEADERS
- FROM 'file:///netflix_titles.csv' AS line
- MATCH (m:Movie {id: line.show_id})

- WITH m, split(line.director, ",") as directors_list
- UNWIND directors_list AS director_name
- MATCH (p:Person {name: director_name})
- CREATE (p)-[:DIRECTED]->(m)

Let’s check the movie titled Ray:
MATCH (m:Movie {title: "Ray"})<-[:DIRECTED]-(p:Person)
RETURN m, p

Chroma DB features

- Simple and powerful:
  - Install with a simple command: `pip install chromadb`.
  - Quick start with Python SDK, allowing for seamless integration and fast setup.
- Full-featured:
  - Comprehensive retrieval features: Includes vector search, full-text search, document storage, metadata filtering, and multi-modal retrieval.
  - Highly scalable: Supports different storage backends like DuckDB for local use or ClickHouse for scaling larger applications.
- Multi-language support:
  - Offers SDKs for popular programming languages, including Python, JavaScript/TypeScript, Ruby, PHP, and Java.
- Integrated:
  - Native integration with embedding models from HuggingFace, OpenAI, Google, and more.
  - Compatible with Langchain and LlamaIndex, with more tool integrations coming soon.
- Open source:
  - Licensed under Apache 2.0.
- Speed and simplicity:
  - Focuses on simplicity and speed, designed to make analysis and retrieval efficient while being intuitive to use.

Chroma DB offers a self-hosted server option.