Introduction to MongoDB (NoSQL DB)

Why NoSql ?

- Relational databases are not designed to scale
- schema, joins

C and Latency Tradeoff

- Amazon claims that just an extra one tenth of a second on their response times will cost them 1% in sales.
- Google said they noticed that just a half a second increase in latency caused traffic to drop by a fifth.

4 Key Words on NoSQL

- Scale
- Speed
- Cloud
- New Data

What is NoSQL?

- non-relational
- simple API
- schema-free
- open-source
- horizontally scalable (sharding)
- replication support
- eventually consistent /BASE

Different types of NoSQL Databases

- NoSQL database are classified according to their data storage models:
  - Column (Cassandra)
  - Document (MongoDB)
  - Key – value Pair(Dynamo – Amazon)
  - Graph

MongoDB

- Name derived from Hu(**MONGO)**us word
- Document Oriented Database
- Built for High – Performance and scalability
- Document based queries for **Easy Readability**
- Replication and failover for **High Availability**
- Auto Sharding for **Easy Scalability**

Comparison between RDBMS and NoSQL DB

- Example: Class
- Location
- Presenter
  - Presenting at a location
- People
  - Potential attendees in context of a class
- Class
  - Presenter in location with people as actual attendees

Relational Database: Example

- Class schema in a relational database
- Presentation { id, name, location}
- People {id, name}
- Address {id, city, state, zip}

Schema for this class in a relational database model

Presentation

id	name	*location*
1	Chris	SJSU

Address

id	city	state
SJSU	San Jose	CA

People

id	name
10	Simon
11	Chris

Class

id	*person*	*presentation*
20	10	1
20	11	1

Relational database: Example

```
CREATE TABLE Presentation (
    id Integer primary key, name String, location string,
    FOREIGN KEY (location) REFERENCES Address(id));
CREATE TABLE Address (
    id String primary key, city String, state String);
CREATE TABLE People (
    id Integer primary key, name String);
CREATE TABLE Class (
    id Integer, person Integer, presentation Integer,
    PRIMARY KEY (id, person, presentation),
    FOREIGN KEY (person) REFERENCES People(id),
    FOREIGN KEY (presentation) REFERENCES Presentation(id));
```

Relational database: Example

```
select Presentation.name, Presentation.location,
  Address.city, Address.state, People.name
from Presentation, Address, People, Class
where Class.person = People.id
        and Class.presentation = Presentation.id
        and Presentation.location = Address.id;
```

```
| name | location | city | state | name |
+--------+--------+-----------+-------+---------+
| Chris | SJSU | San Jose | CA   | Simon|
| Chris | SJSU | San Jose | CA   | Chris  |
```

Relational Database: Recap

1. Schema design
   Primary key (underlined) and foreign key (cursive) constraints
2. Table creation
   DDL
3. Data insertion for each table
   DML
4. Query: join
   DML
5. Data structure creation within application system
   JDBC resultset to e.g. Java objects

NoSQL Database: Use Case Example

```
use course /* database will be created if not present */
db.presentation.insert(
{"id": 1,
  "name": “Simon",
  "location": {"id": "SJSU",
               "city": "San Jose",
               "state": "CA"
              },
"people": [{"id": 10, "name": "Simon"},
            {"id": 11, "name": "Chris"}
          ]
})
```

NoSQL Database: Use Case Example

- db.presentation.find()
- db.presentation.find({"id": 1})

NoSQL Database: Recap

1. ~~Schema design~~
    ~~Primary key (underlined) and foreign key (cursive)~~
constraints
2. Table creation
    DDL
3. Data insertion for each table
    DML
4. Query: ~~join~~
    DML
5. ~~Data structure creation within application system~~
    ~~JDBC resultset to e.g. Java objects~~

NoSQL Database: Major Players

- Too many document NoSQL databases to name a few distinct ones

29 systems in ranking, July 2014

Rank	Last Month	DBMS	Database Model	Score	Changes
1.	1.	MongoDB	Document store	238.78	+7.33
2.	2.	CouchDB	Document store	23.07	+0.28
3.	3.	Couchbase	Document store	16.58	+0.79
4.	4.	MarkLogic	Multi-model	8.20	-0.02
5.	5.	RavenDB	Document store	5.09	-0.42
6.	6.	GemFire	Document store	2.16	-0.06
7.	7.	OrientDB	Multi-model	1.71	-0.02
8.	8.	Cloudant	Document store	1.70	+0.07
9.	9.	Datameer	Document store	0.88	+0.08
10.	10.	Mnesia	Document store	0.72	+0.01

s

Key Benefit of NoSQL: O(1) Lookup

- Fast lookup
  - No joining required
  - All data about one domain concept in one document
- Direct programming language representation
  - No mapping or ‘ORM’ layer required
- JSON library
  - Direct result representation and manipulation
  - JavaScript: representation in language data types directly
  - E.g., check out MongoDB node.js driver

Key Problem of NoSQL: No Join Operator

- Many NoSQL databases do not implement a join query operator
  - If you need to join data, then you have to do it in the application system layer
- But, wait a moment ...
  - Is it ever necessary to join data in NoSQL databases?
  - Some claim: not necessary due to support of
    - Sub-documents
    - Arrays (lists)
- Let's look at an example
  - Supplier - Parts

Key Problem of NoSQL: No Join Operator

- Example
  – Supplier - Parts relationship (N:M)
  – Each supplier supplies many parts
  – Each part supplied by many suppliers
- Relational DBMS
  – “Supplier” table
  – “Part” table
  – “Supplies” relationship in table

Key Problem of NoSQL: No Join Operator

Supplier - Part - Supplies

Supplier

id	name
10	Supp1
11	Supp2

Part

id	name
20	Part1
21	Part2

Supplies

supplier_id	*part_id*
10	20
10	21
11	20

Key Problem of NoSQL: No Join Operator

Supplier - Supplies – Part

```
{ "id": 10,
"name": "Supp1",
"supplies": [{"id": 20,"name":"Part1"},
             {"id": 21, "name": "Part2"}]}

{ "id": 11,
"name": "Supp2",
"supplies": [{"id": 20,"name":"Part1"}]}
```

Supplier - Supplies – Part

```
{ "id": 10,
"name": "Supp1",
"supplies": [20, 21]}
{ "id": 10,
"name": "Supp1",
"supplies": [20, 21]}

{"id": 20, "name": "Part1"}
{"id": 21, "name": "Part2"}
```

Why use MongoDB?

- MongoDB stores data in Objects
- Uses BSON (Binary JSON)
- No Joins
- No Complex Queries
- Embedded Documents and arrays reduce the need for joins
- No multi-document transactions

22

Where to use MongoDB ?

- Ideal for Web Applications
- Applications containing semi-structured data and needing flexible schema management
- Caching and High Scalability
- Scenarios where **data availability** and **size of data** are priorities over the **transactions** of data

23

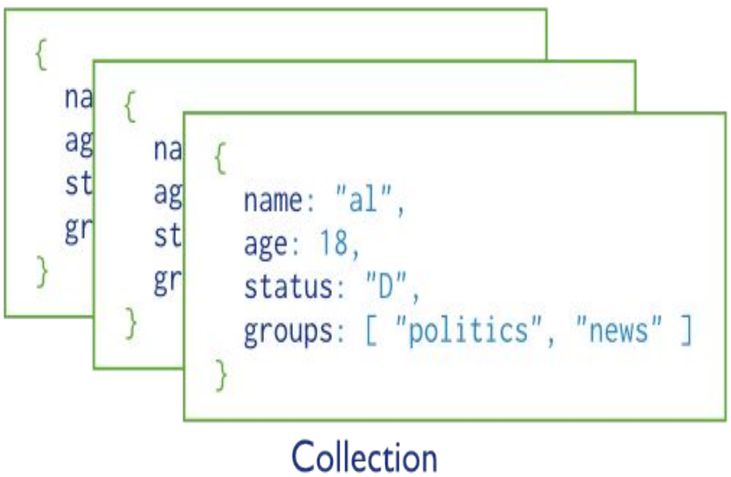
Terminology

- Mysql
- Table
- Row
- Column
- Joins
- Group By

- MongoDB
- Collection
- Document
- Field
- Not Recommended ($lookup)
- Aggregation

Collections in MongoDB

- MongoDB stores all data in Collections
- It is schema – less and contains a group of related documents
- Created on-the-fly when referenced for the first time



25

Document in MongoDB

- Stored in Collections
- Has **_id** field – works like Primary keys in Relational databases
- Sample document containing name, age, status and groups

26

Queries in MongoDB

- MongoDB provides **db.collection.find()** method
- This method accepts both query criteria and projections

27

Projections - Queries in MongoDB

- If you include 1 –it returns the value
- If you include 0 –it eliminates it from the result

```
db.records.find( { "user_id": { $lt: 42} }, { "_id": 0, "name": 1 , "email": 1 } )
```

- _id – always included in results. Specify “**_id : 0**” to exclude it from results

28

Insert Operation

- In MongoDB, **db.collection.insert()** method adds new documents to collections

29

Update Operation

- In MongoDB, **db.collection.update()** method modifies existing documents in a collection

```
db.users.update(                          ← collection
   { age: { $gt: 18 } },                  ← update criteria
   { $set: { status: "A" } },             ← update action
   { multi: true }                        ← update option
)
```


30

Remove Operation

- In MongoDB, **db.collection.remove()** method deletes document from the collection

```
db.users.remove(          ← collection
   { status: "D" }        ← remove criteria
)
```


31

References

- SQL vs NoSQL - **https://www.mongodb.com/nosql-explained**
- MongoDB Introduction - **http://docs.mongodb.org/manual/core/crud-introduction/**
- Installing MongoDB (Mac) - **https://www.youtube.com/watch?v=_WJ8m5QHvwc**
- Installing MongoDB (Windows) - **https://www.youtube.com/watch?t=1&v=sBdaRlgb4N8**

32

When to not use MongoDB?

- ACID properties are important for storage
- Highly Transactional Applications (Banking domain, Security)
- Problems and applications requiring Joins and complex queries

33

Key Problem of NoSQL: No Database-Enforced Consistency

- Not enforced
  - Primary key
  - Foreign key
  - Enumeration
  - Cascading delete
  - etc.
- Enforcement can be accomplished
  - When
    - reading or writing
  - In application system code
  - In self-implemented database access layer
  - In separate consistency check process
  - Not at all

How does MongoDB Store data?

- Stores data in form of Documents
- JSON like field – value pair
- Documents analogous to structures in programming languages with key – value pair
- Documents stored in **BSON (Binary JSON)** format
- BSON is JSON with additional type information

35

NoSQL: Key Insights

- Specialized data models
  - Not universal, but optimized towards special cases
- Specialized query access
  - Not universal, but optimized towards special cases
- Different / absent consistency supervision
  - Relaxed constraints

- Trade-off
  - Gain through specialization
  - Implementation of missing functionality outside of database

Mongoose

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It simplifies interactions with MongoDB by providing a structured way to define schemas, models, and perform database operations.

Key Features of Mongoose:

- Schema-Based Models – You define the structure of documents in a collection using a schema.
- Validation – Mongoose offers built-in validators (e.g., required fields, data types) to ensure data integrity.
- Middleware (Hooks) – You can define pre/post actions before/after database operations.
- Query Building – Mongoose allows you to perform complex queries easily.
- Relationships (Population) – Mongoose can reference other documents, enabling relational-like data handling.
    a. No Foreign Key Constraints: You must manually ensure that referenced documents exist before inserting or updating data.
    b. No Multi-Document Transactions (by Default): MongoDB operates without full ACID transactions across multiple documents unless using multi-document transactions (introduced in MongoDB 4.0), which are only supported in replica sets.
    c. No Native Joins (Slower Queries with .populate())
    d. MongoDB with References = Potential Performance Issues
    e. Since MongoDB is document-based, it is optimized for denormalized data (embedding related documents instead of referencing them).

Multi-Document Transactions in MongoDB

A transaction is a sequence of database operations that either: Fully complete (all changes are applied) or Fully rollback (if something fails, no changes are applied)

Key Features

- Start a transaction using `session.startTransaction()`
- Commit the transaction using `session.commitTransaction()`
- Abort the transaction if any operation fails using `session.abortTransaction()`

When Should You Use Multi-Document Transactions?

Best Use Cases

- Banking & Payments – Ensure money is deducted from one account only if it is credited to another.
- E-Commerce Orders – Reserve an item only if payment is successful.

Limitations

- Not supported for sharded clusters (before MongoDB 4.2)
- Transactions are limited to 16MB of data
- May **cause performance overhead** (use only when necessary)

Building Long-Term Memory in LLM Apps

with FastAPI + LangChain
Focus: conversation storage & user

memories Notes:

- Deck emphasizes long-term memory patterns and production guidance.
- Examples use Python, FastAPI, and LangChain.
- We'll connect the concepts to how ChatGPT memory works in practice.

What do we mean by “memory”?

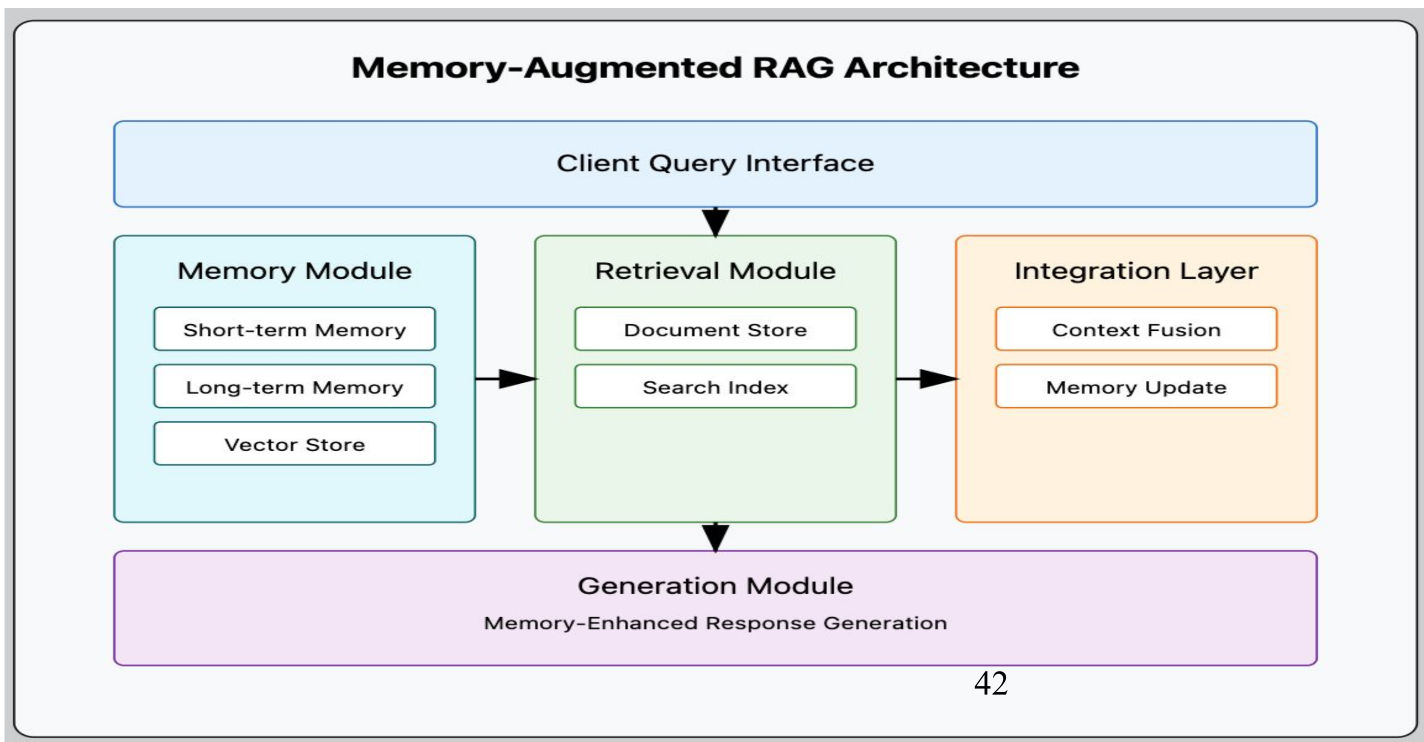
- **Short-term memory**: messages stuffed into the model’s prompt (context window).
- **Long-term memory**: information persisted outside the model (DB or vector store) and **retrieved** when relevant.
- Goals: personalization across sessions, grounding the model in **user-specific facts**, and reducing repetition.

Short-term vs Long-term

Dimension	Short-term	Long-term
Lifetime	Only this prompt	Across chats/sessions
Storage	In prompt tokens	DB / vector DB
Cost	Token-bound	Storage + retrieval
Personalization	Limited	Strong (preferences, history)
Failure modes	Context overflow	Stale/irrelevant memories

Why memory?

- Personalization: “Remember I’m vegetarian” or “I prefer TypeScript”
- Productivity: skip re-explaining projects and preferences
- Accuracy: retrieve **facts** about the user/workspace to ground answers



Memory taxonomy

- **Episodic**: specific events ("met with Sam on 2025-09-01")
- **Semantic**: distilled facts ("user prefers Python ≥3.10")
- **Entity**-centric: per-entity summaries (people, companies, repos)
- **Reflections**: higher-level syntheses generated from many memories

Concept: Generative Agents loop (inspiration)

- Observe : **Store** experiences
- Retrieve: **Reflect** synthesize higher-level memories
- Plan then Act
- this is a useful pattern for evolving user profiles while avoiding bloat.

Designing a conversation memory store (1/2)

Data model (example):

```
 message_id, user_id, role, content, ts
 memory_id, user_id, text, type
(episodic|semantic|entity|reflection), score, ts

 Embedding vectors: stored in a vector DB (Chroma / Pinecone /
Weaviate / FAISS) with metadata
```

Ingestion heuristics:

Save **explicit** user requests (“remember…”)

Auto-promote candidate facts (scored by *recency* × *importance* × *novelty*)

Designing a conversation memory store (2/2)

Retrieval:

- KNN over embeddings of the current query + chat summary
- Filter by `user_id`, freshness, and type
- Diversify results (MMR) and cap token budget

Refresh & expiry:

- Periodic summarization of stale memories
- Decay or archive low-value items

Architecture (high-level)

```
Client ──▶ FastAPI ──▶ Memory Service
                    ├─▶ Vector DB (embeddings)
                    ├─▶ SQL/NoSQL (messages & metadata)
                    └─▶ LLM (generation)
↰────────────────────────────────────────────── Retrieval (top-K) ─▶
```

Memory-augmented Chat Loop

1. Chunk & embed conversation facts and notes
2. Store in a **vector DB** with metadata
3. At query time, retrieve salient snippets
4. Compose the prompt: **system** + **instructions** + **retrieved memories** + **recent chat**
5. Generate → optionally **write back** new/updated memories

How will you plan your application?

- **FastAPI** app for `/chat` and `/remember` endpoints
- **Embeddings**: any provider (e.g., `text-embedding-3-*`)
- **Vector store**: Chroma (local), Pinecone/Weaviate (managed)
- **Retriever**: similarity search + filters (+MMR)
- **LLM**: your chosen chat model

FastAPI: skeleton

```python
from fastapi import FastAPI
from pydantic import BaseModel
from typing import Optional, List
from datetime import datetime

app = FastAPI()

class ChatRequest(BaseModel):
    user_id: str
    message: str
    project: Optional[str] = None

class RememberRequest(BaseModel):
    user_id: str
    memory: str
    kind: str = "semantic"  # episodic|semantic|entity|reflection
```

```python
@app.post("/remember")
def remember(req: RememberRequest):
    # 1) embed req.memory
    # 2) upsert into vector DB w/ metadata {user_id, kind, ts}
    # 3) persist raw memory row in SQL/NoSQL
    return {"ok": True, "stored": req.memory}

@app.post("/chat")
def chat(req: ChatRequest):
    # 1) build retriever for this user
    # 2) retrieve top-K memories for req.message (+recent summary)
    # 3) compose prompt and call LLM
    # 4) optionally store episodic memory for this message
    return {"reply": "Hello! (retrieved personalized context…)"}
```

LangChain: retrieval over your memory store

```
from langchain_community.vectorstores
import Chroma from langchain_openai
import OpenAIEmbeddings
from langchain_core.documents import Document

emb = OpenAIEmbeddings(model="text-embedding-3-small")
db = Chroma(collection_name="memories", embedding_function=emb,
persist_directory=".chroma")

def upsert_memory(user_id: str, text: str, kind:
    str, ts: str): metadata = {"user_id":
    user_id, "kind": kind, "ts": ts}
    db.add_texts([text], metadatas=[metadata])

def retrieve_memories(user_id: str, query: str, k: int = 5):
    results = db.similarity_search(query, k=k, filter={"user_id": user_id})
    return [r.page_content for r in results]
```

Notes:

- Use `.persist()` to flush Chroma to disk and survive restarts.
- Use **filters** to scope by user/project; add MMR if desired.

Summarization for long-term memory hygiene

- Periodically summarize long conversations into **compact facts**
- Keep the **semantic essence**, drop transient chatter
- Store summaries as separate “reflection” memories

Example:

```
"User prefers Django + Postgres; default region is us-east-1."
```

Entity-centric memory

- Track facts keyed by entities (people, orgs, repos)
- Maintain a short card (traceable facts) per entity and update incrementally
- Use **NLP entity extraction** to route new facts to the right card

  - Entities: `Jane Doe` (person), `Acme` (org), `acme/payments` (repo)
  - Proposed facts:
    - person/Jane: `title=Senior Engineer`
    - org/Acme: `contact=Jane Doe`
    - repo/acme/payments: `language=Python`

Prompt template (with retrieved memories)

```
System:
You are a helpful assistant. Personalize answers using the user's saved
memories when relevant. If a memory conflicts with an explicit new
instruction, prefer the new instruction.

Context (retrieved memories, trimmed to N tokens):
- {memory_1}
- {memory_2}
- ...

Recent chat:
{recent_turns}

User:
{message}

Assistant:
```

Storage choices (trade-offs)

Store	Pros	Cons
Chroma (local)	Simple, open-source, persists to disk	Single-node by default
FAISS	Very fast, in-process	You build the metadata layer
Pinecone	Managed, scalable, hybrid dense+sparse	$$$ ongoing cost
Weaviate	Open-source/managed, rich filters & modules	Operational complexity

How all of this maps to ChatGPT Memory

- Two key ideas:
  - **Saved memories** (what users explicitly tell it to remember)
  - **Referencing chat history** (implicit, from prior conversations)
- User controls: turn memory **on/off**, **ask what it remembers**, and **delete** items
- **Temporary chats** bypass memory

Example: end-to-end flow

1. User: “Remember that I prefer FastAPI” → `/remember` (store semant
2. Later: User asks about an API → retriever pulls preference
3. Prompt composes guidance (“prefer FastAPI examples”)
4. Assistant replies in preferred style; store an **episodic** memory of this

Common pitfalls

- Storing *everything* → noisy retrievals
- Not scoping by user/project → cross-talk between users
- Never summarizing → ballooning storage & cost

Metrics & tests

- Memory precision/recall (did we pull the *right* memory?)
- Personalization impact (A/B on task success / CSAT)
- Staleness rate (facts that became wrong)
    -