Enterprise Multi-Database Natural Language Query Engine

Version 1.0

Prepared for

myOnsite Healthcare, LLC.

Document Control

Rev. No.	Description of Change	Effective Date
1.0	Initial Release	11st Aug 2025

Authored By

Name	Role	Signature	Date
Het	Team Lead		11st Aug 2025

Reviewed and approved By

Name	Role	Signature	Date

Enterprise Multi-Database Natural Language Query Engine

Advanced Text-to-SQL System with Dynamic Schema Adaptation

Project Overview

Build a production-grade, multi-database natural language query engine that can understand complex business questions, generate optimized SQL across different database systems, handle real-time schema evolution, implement advanced security controls, and provide enterprise-level governance and observability.

Time Allocation: 5 hours
Complexity Level: Principal Engineer Challenge
Focus Areas: Advanced NL-to-SQL, multi-database orchestration, real-time schema adaptation, enterprise security

---

System Architecture Requirements

You're building an enterprise query engine that must:

- Support **multiple database systems** (PostgreSQL, MySQL, SQLite, MongoDB) with dialect-specific optimizations
- Handle **complex natural language queries** including temporal reasoning, multi-table joins, and business logic inference

- Implement **real-time schema introspection** with automatic adaptation to structural changes
- Provide **advanced security controls** including SQL injection prevention, role-based access, and query governance
- Support **concurrent query execution** with intelligent caching and performance optimization
- Include **comprehensive audit logging** and explainable AI for query reasoning
- Handle **ambiguous queries** with clarification mechanisms and confidence scoring

---

Database Environment

Multi-Database Setup

You'll work with **4 interconnected databases**:

1. **PostgreSQL** (Primary OLTP)
    - **Tables:** customers, orders, products, inventory, suppliers, categories
    - **Complex features:** Foreign keys, indexes, triggers, stored procedures, views
    - **Data volume:** ~100K customer records, ~500K order records
    - **Advanced structures:** JSON columns, arrays, custom types, partitioned tables
2. **MySQL** (Analytics/Reporting)
    - **Tables:** sales_analytics, customer_segments, product_performance, regional_data
    - **Features:** Window functions, CTEs, materialized views
    - **Data volume:** ~1M aggregated records with time series data

- **Complex queries:** Multi-level aggregations, rolling averages, cohort analysis

3. **SQLite** (Configuration/Metadata)
    - **Tables:** system_config, user_preferences, query_history, schema_versions
    - **Features:** Full-text search, JSON support, temporary tables
    - **Dynamic schema:** Tables and columns change frequently
4. **MongoDB** (Document Store)
    - **Collections:** user_profiles, product_catalogs, activity_logs, recommendations
    - **Features:** Complex nested documents, aggregation pipelines, geospatial queries
    - **Challenge:** Convert natural language to MongoDB aggregation syntax

Advanced Query Dataset

25 progressively complex natural language queries:

Tier 1 - Basic (5 queries):

- "Show me all customers from California"
- "What's the total revenue this month?"

Tier 2 - Multi-table Joins (8 queries):

- "Which customers have placed orders worth more than $1000 in the last 6 months but haven't ordered in the past 30 days?"
- "Show me the top 5 product categories by revenue growth compared to last quarter"

Tier 3 - Temporal & Analytics (7 queries):

- "Calculate the 3-month rolling average of monthly recurring revenue by customer segment"
- "Find customers who exhibited churn behavior patterns similar to our top 10% revenue customers"

Tier 4 - Cross-Database Complex (5 queries):

- "Compare PostgreSQL order patterns with MongoDB user activity to identify customers likely to upgrade their subscription tier within 60 days"
- "Generate a cohort analysis of customer lifetime value using data from all databases, segmented by acquisition channel and geographic region"

Advanced Technical Requirements

Multi-Database Query Orchestration

- **Database abstraction layer** supporting PostgreSQL, MySQL, SQLite, and MongoDB
- **Dialect-specific SQL generation** with database-specific optimizations
- **Cross-database join capabilities** using temporary staging and federated queries
- **Connection pooling and failover** with automatic load balancing
- **Transaction management** across multiple database connections
- **Query result federation** and cross-database data correlation

Advanced Natural Language Processing

- **Multi-intent query parsing** handling compound questions and sub-queries

- **Business context awareness** understanding domain-specific terminology and relationships
- **Temporal reasoning** interpreting relative dates, time ranges, and business cycles
- **Ambiguity resolution** with confidence scoring and clarification requests
- **Query expansion** automatically adding relevant filters and joins based on business rules
- **Conversational context** maintaining query history and follow-up question handling

Dynamic Schema Intelligence

- **Real-time schema discovery** across all database systems with caching
- **Schema relationship mapping** automatically detecting foreign key relationships and business logic
- **Change detection and adaptation** handling schema modifications without service interruption
- **Version management** tracking schema changes and maintaining query compatibility
- **Intelligent column mapping** handling renamed/moved columns with semantic similarity matching
- **Constraint awareness** understanding business rules encoded in database constraints

Enterprise Security Framework

- **Multi-layered SQL injection prevention** with parameterized queries and input validation
- **Role-based access control** with fine-grained table and column permissions
- **Query governance** with approval workflows for potentially expensive operations

- **Data masking and anonymization** for sensitive information in query results
- **Audit trail generation** logging all queries, results, and user interactions
- **Rate limiting and quota management** preventing resource abuse

Advanced Query Generation

- **Multi-step reasoning** breaking complex questions into executable query sequences
- **Optimization hint generation** suggesting indexes and query plan improvements
- **Cost estimation** predicting query execution time and resource usage
- **Alternative query suggestions** providing multiple approaches for ambiguous requests
- **Business logic integration** incorporating calculated fields and derived metrics
- **Error recovery and self-healing** automatically fixing common SQL syntax issues

Real-Time Schema Evolution Handling

During execution, your system will face **multiple schema changes**:

Phase 1 (Hour 2): Basic structural changes

- Rename `customers.email` to `customers.email_address`
- Add new table `customer_segments` with foreign key relationships
- Modify `orders` table to add `discount_applied` column

Phase 2 (Hour 3): Complex relationship changes

- Split `products` table into `products` and `product_variants`
- Add multi-table view `customer_order_summary`
- Introduce partitioning on `orders` table by date range

Phase 3 (Hour 4): Advanced schema evolution

- Add MongoDB integration with `user_behavior` collection
- Introduce PostgreSQL array columns for tags and categories
- Add stored procedures for complex business logic calculations
- Implement row-level security policies

Your system must:

- **Detect changes automatically** without manual intervention
- **Adapt queries in real-time** maintaining correctness and performance
- **Provide graceful degradation** when schemas are temporarily incompatible
- **Maintain query history compatibility** ensuring previous queries still work
- **Update cached schema information** without service interruption

---

Extreme Implementation Challenges

Multi-Database Query Federation

- **Cross-database joins:** Query data from PostgreSQL customers table and MongoDB user behavior collection simultaneously
- **Data type harmonization:** Convert between SQL and NoSQL data structures seamlessly
- **Transaction coordination:** Maintain ACID properties across multiple database systems
- **Performance optimization:** Minimize cross-database data transfer and optimize execution plans

Advanced Natural Language Understanding

- **Multi-intent parsing:** "Show me revenue trends AND identify underperforming products AND suggest optimization strategies"
- **Contextual ambiguity resolution:** Handle pronouns, temporal references, and business domain terminology
- **Conversational query chaining:** Support follow-up questions that reference previous query results
- **Voice input processing:** Accept and process natural language queries via voice API integration

Real-Time Schema Evolution

- **Live migration handling:** Continue serving queries while schema changes are applied
- **Backward compatibility:** Ensure existing queries continue to work during and after schema changes
- **Semantic column matching:** Automatically map renamed columns based on data content and usage patterns
- **Relationship inference:** Detect new foreign key relationships and business logic automatically

Enterprise Security & Governance

- **Dynamic data masking:** Apply different masking rules based on user roles and query context
- **Query approval workflows:** Route potentially expensive or sensitive queries through approval processes
- **Compliance reporting:** Generate audit reports for SOX, GDPR, and industry-specific regulations
- **Zero-trust architecture:** Verify permissions for every query component and data access

Advanced Performance Optimization

- **Intelligent query caching:** Cache results at multiple levels with invalidation based on data changes

- **Predictive prefetching:** Anticipate follow-up queries and pre-execute common patterns
- **Cost-based routing:** Route queries to optimal database replicas based on current load and costs
- **Real-time performance tuning:** Automatically adjust query execution strategies based on performance metrics

---

Schema Evolution Stress Test

Your system will face **continuous schema changes** throughout development:

Hour 1: Initial setup with stable schemas across all databases **Hour 2: First Evolution Wave**

- PostgreSQL: Rename 3 columns, add 2 new tables with complex foreign key relationships
- MySQL: Partition existing tables, add materialized views
- SQLite: Add full-text search indices, modify constraint definitions
- MongoDB: Restructure document schemas, add new nested field structures

Hour 3: Complex Relationship Changes

- Cross-database foreign key simulation through application logic
- Introduction of database-specific advanced features (PostgreSQL arrays, MySQL JSON functions)
- Schema versioning with multiple concurrent versions active

Hour 4: Enterprise Feature Integration

- Row-level security policy implementation
- Database-specific stored procedures and functions
- Real-time replication setup with read/write splitting

- Advanced indexing strategies (partial indexes, functional indexes)

Hour 5: Stress Testing & Recovery

- Simulate database connectivity issues and failover scenarios
- Handle concurrent schema changes from multiple sources
- Test query adaptation under high concurrent load
- Validate data consistency across all database systems

Success Metrics & Expectations

Functional Requirements (Must Have)

- **Query Accuracy:** >90% correct SQL generation for Tier 1-3 queries, >75% for Tier 4
- **Schema Adaptation:** Automatically handle all schema changes within 30 seconds
- **Multi-Database Support:** Successfully execute queries across all 4 database systems
- **Security Compliance:** Pass all SQL injection tests and access control validations
- **Performance Standards:** <3 seconds for simple queries, <10 seconds for complex cross-database operations

Advanced Capabilities (Differentiation Factors)

- **Natural Language Sophistication:** Handle ambiguous queries with confidence scoring >0.8
- **Business Intelligence:** Provide actionable insights and alternative query suggestions
- **Explainable AI:** Generate clear explanations for query reasoning and decision-making

- **Enterprise Readiness:** Comprehensive audit logging, compliance reporting, and governance workflows
- **Innovation Factor:** Creative solutions to unique challenges like voice input or predictive querying

Production-Grade Excellence

- **Monitoring & Observability:** Real-time dashboards with predictive alerting
- **Scalability:** Handle 1000+ concurrent queries without performance degradation
- **Reliability:** 99.9% uptime with graceful degradation during failures
- **Cost Optimization:** Intelligent resource usage with cost tracking and budgeting
- **Security Excellence:** Zero vulnerabilities in penetration testing

---

Deliverables

1. Production System

- **Multi-interface access:** REST API, GraphQL endpoint, WebSocket streaming, and CLI
- **Enterprise dashboard:** Real-time monitoring, query analytics, and administrative controls
- **Mobile-responsive UI:** Web interface for natural language query input and result visualization

2. Comprehensive Test Suite

- **Unit tests:** >85% code coverage with edge case handling
- **Integration tests:** End-to-end workflow validation across all database systems
- **Performance benchmarks:** Load testing with scalability analysis

---

- **Security validation:** Penetration testing and vulnerability assessment
- **Schema evolution tests:** Automated testing of all schema change scenarios

3. Enterprise Documentation

- **System architecture:** Detailed technical design with decision rationale
- **API documentation:** Interactive documentation with live examples
- **Deployment guide:** Production deployment with high-availability configuration
- **Security documentation:** Security controls, threat model, and compliance mapping
- **Performance tuning:** Optimization strategies and troubleshooting guide

4. Business Intelligence Layer

- **Query analytics:** Usage patterns, performance trends, and optimization opportunities
- **Business insights:** Automated detection of data anomalies and business opportunities
- **Cost analysis:** Query cost tracking with optimization recommendations
- **Compliance reporting:** Automated audit reports and regulatory compliance validation

Parul® University

Nosql Data Management

Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER-2

NoSQL Data Management

Introduction to NoSQL

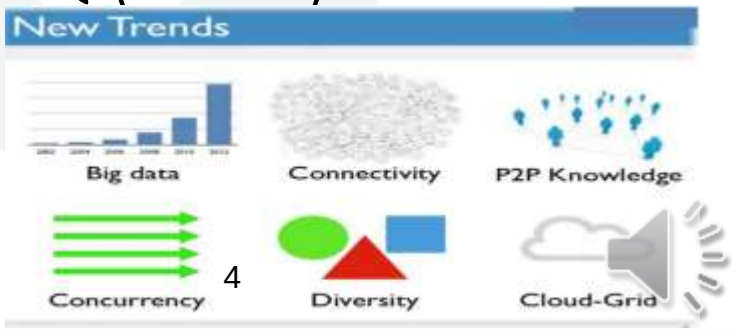
What is NoSQL?

• NoSQL is non-relational database management system.
• A NoSQL database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.
•It is a type of Non-Relational Database system.
•It follows CAP theorem rather than ACID properties of RDBMS.

Why and when NoSql should be used ?

- The Traditional database is more predictable and preferable for Structured data.
- But, NoSQL is preferable for handling unstructured type of data . Unstructured data is type of data which has no proper format this type of data is generated by machine and humans in the form of satellite image, social media, mobile data, web-content etc
- NoSQL has high performance with high availability, and offers rich query language and easy scalability compared to SQL(RDBMS)

Why NoSQL is needed today most ???



Need of NoSQL

- Explosion of Social media sites (FaceBook, Twitter etc) resulting in generation of large data of different formats and size.
- Rise of Cloud based technologies like Amazon Web Services (AWS), Google cloud and Microsoft Azure
- Expansion of Open source communities.

Characteristics of NoSQL

- Does not follow ACID properties
- Less complexity as of SQL queries
- Schema less
- Fast development
- Large data volumes
- Easy and frequent DB changes

When to use NoSQL

- When Traditional RDBMS model provides restriction
- When ACID properties is not supported
- Temporary data(Like Wishlist, shopping cart, session data etc)
- When joins are expensive in RDBMS(product cost, hardware, maintenance)

When not to use NoSQL

- Financial Data, Data requiring strict ACID properties, Buisness critical data

CAP Theorem

•**Consistency**-This means that the data in the database remains consistent after the execution of an operation. For example after an update operation all clients see the same data.

•**Availability**-This means that the system is always on (service guarantee availability), no downtime.

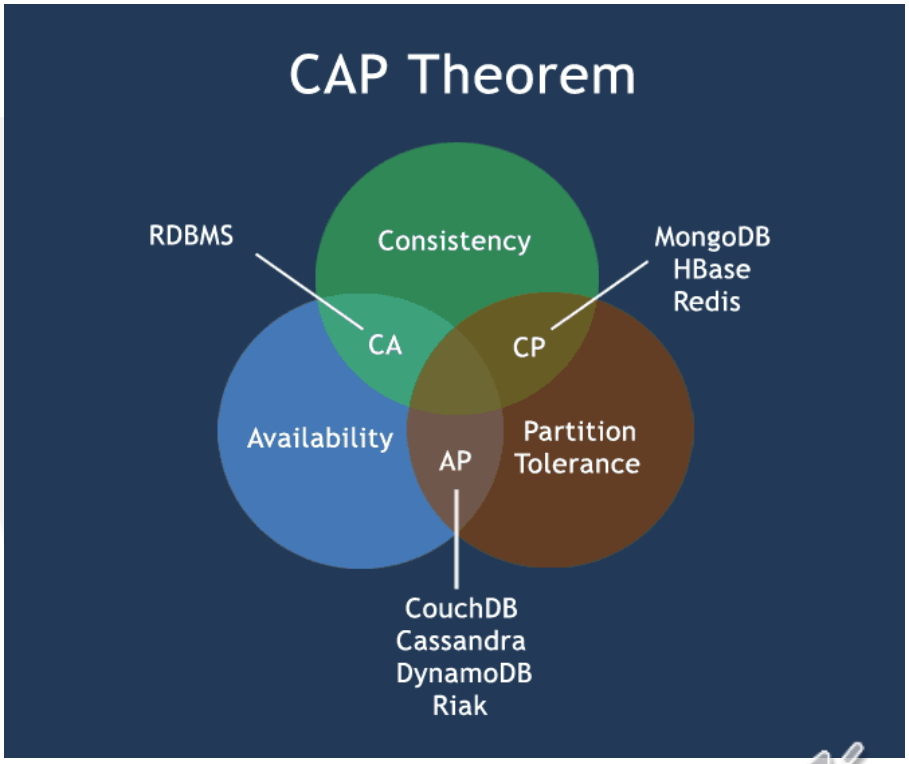
•**Partition Tolerance**-This means that the system continues to function even the communication among the servers is unreliable, i.e. the servers may be partitioned into multiple groups that cannot communicate with one another. Here, if part of the database is unavailable, other parts are always unaffected.

CAP Theorem

- **CA** -Single site cluster, therefore all nodes are always in contact. When a partition occurs, the system blocks.

- **CP**-Some data may not be accessible, but the rest is still consistent/accurate.

- **AP**-System is still available under partitioning, but some of the data returned may be inaccurate.



The BASE

- The CAP theorem states that a distributed computer system cannot guarantee all of the following three properties at the same time:
  - Consistency
  - Availability
  - Partition tolerance
- A BASE system gives up on consistency.

The BASE

- **Basically Available** indicates that the system *does guarantee availability, in terms of the CAP theorem. i.e. DB is available all the time.*

- **Soft state** indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.

- **Eventual consistency** indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

Eventual Consistency

- The term "eventual consistency" means to have copies of data on multiple machines to get high availability and scalability. Thus, changes made to any data item on one machine has to be propagated to other replicas.

- Data replication may not be instantaneous as some copies will be updated immediately while others in due course of time. These copies may be mutually, but in due course of time, they become consistent. Hence, the name eventual consistency.

Types of NoSQL

- There are 4 main types of NoSQL databases:
  - -Key value
  - -Column Family
  - -Document
  - -Graph

Image source : Google

Key Value

- Data is stored in Key/value pairs. It is designed in such a way that it can handle lots of data and heavy load.
- Key-value pair storage databases store data as a hash table where each key is unique and the value can be JSON, BLOB(Binary Large Object), String etc can be accessed by string called keys
- For EX: A key value pair may contain key like "University" associated with value like "Parul".

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

Key Value

- It is one of the most basic types of NoSQL databases
- This kind of database is used as a collection, dictionaries, associative arrays etc. key value stores help the developer to store schemaless data
- They work best for Shopping cart contents

Basic Operations are

Insert(Key,value), Fetch(key), Update(key), delete(Key).

Column- based

- Column oriented databases work on columns and are based on Big table paper by Google.
- The column is the lowest/ smallest instance of data.
- It is a tuple that contains a name, value and a timestamp
- Every column is treated separately.
- Values of single column databases are stored contiguously.

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

Column based

•They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

•Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs

•HBase, Cassandra, HBase, Hypertableare examples of column based database

Some statistics about Facebook search(Using Cassandra).

MySQL>50gb data	Rewritten in Cassandra<50gb data
Writes average – 300 ms	Writes average: 0.12 ms
Reads average _ 350 ms	reads average: 15ms

Document based

- Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document.
- The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.
- Pair each key with complex data structure known as data structure.
- Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.

Col1	Col2	Col3	Col4
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data	Data	Data

Document 1

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Document 2

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Document 3

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```


Document based

•The document type is mostly used for CMS(content management system) systems, blogging platforms, real-time analytics & e-commerce applications.

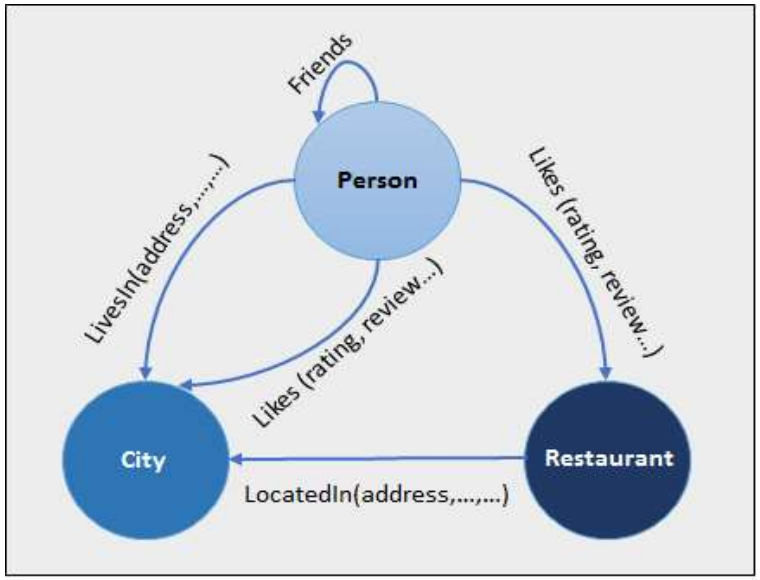
•It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.

•Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes,are popular Document originated DBMS systems.

Graph based

•A graph type database stores entities as well the relations amongst those entities.

•The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Graph based

•Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

•Graph base database mostly used for social networks, logistics, spatial data.

•Neo4J, Infinite Graph, OrientDB, FlockDBare some popular graph-based databases.

Comparison of SQL and NoSQL

1.Relational Database Management System

2. SQL is also know as Structured query language.

3.They have predefined Schema

1.Non relational management system

2.NoSQL is also known as Not only Structured query language

3.They have dynamic schema for unstructured data

Comparison of SQL and NoSQL

4.Vertically scalable	4. Horizontally scalable
5. RDBMS are Table based	5. NoSQL databases are document, key-value, graph or wide-column based
6.SQL are better for multi-row transactions.	6. NoSQL is better for unstructured data like documents or JSON
7. Example: Oracle, MySQL	7. Example: MongoDB, Cassandra, CouchDB etc.

What is NewSQL

•The term NewSQL is not exactly as wide as NoSQL.

•NewSQL systems all begin with the relational data model and the SQL query language and they all attempt to cover portion of similar types of scalability, flexibility or lack-of-focus that has driven the NoSQL development.

Why NewSQL ?

•The early counter-measures to the above approaches were a powerful single node machine that is able to handle all the transaction, a custom built middleware system to distribute queries over traditional DBMS nodes.

•But both of them are prohibitively expensive to be carried out.

Why NewSQL ?

•As a result, there was a need of an intermediate database system which combines the distributed architectures of NoSQL systems with multiple node concurrency and a whole new storage mechanism.

•Thus Newsql can be defined as a class of modern relational DBMSs that seek to provide the same scalable performance of NoSQL for OLTP workloads and simultaneously guaranteeing ACID compliance for transactions as in RDBMS.

•these systems want to achieve the scalability of NoSQL without having to discard the relational model with SQL and transaction support of the legacy DBMS.

Concepts of NewSQL

•Main Memory storage of OLTP(Online transition process) databases enables in-memory computations of databases.

•Scaling out by splitting a database into disjoint subsets called either partitions or shards, leading to the execution of a query into multiple partitions and then combine the result into a single result.

•NewSQL systems preserve the ACID properties of databases.

•Enhanced concurrency control system benefits upon traditional ones.

Concepts of NewSQL

•Presence of secondary index allowing NewSQL to support faster query processing times.

•High availability and Strong data durability is made possible with the use of replication mechanisms.

•NewSQL systems can be configured to provide synchronous updates of data over the WAN.

•Minimizes Downtime, provides fault tolerance with its crash recovery mechanism.

Comparison of NoSQL and NewSQL

1. It doesn't follow a relational model, it was designed to be entirely different from that.
2. Supports CAP Theorem
3. Supports old SQL

1. Since the relational model is equally essential for real-time analytics.
2. Supports ACID property
3. Supports old SQL and even enhanced functionality o f old SQL

Comparison of NoSQL and NewSQL

4. Supports OLTP database but it is not best suited.	4. Supports OLTP database and highly efficient.
5. Vertical Scaling	5. Vertical and horizontal scaling
6. Better than SQL for processing complex queries	6. Highly efficient to process complex queries and smaller queries
7. Supports distributed system	7. Supports distributed system

Map reduce

•Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results.

•MongoDB uses mapReduce command for map-reduce operations.

•MapReduce is generally used for processing large data sets.

MapReduce Command

```
>db.collection.mapReduce(
   function() {emit(key,value);},  //map function
   function(key,values) {return reduceFunction}, {   //reduce function
      out: collection,
      query: document,
      sort: document,
      limit: number
   }
)
```


MapReduce Command

The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs, which is then reduced based on the keys that have multiple values.

In the above syntax −
map is a javascript function that maps a value with a key and emits a key-value pair
reduce is a javascript function that reduces or groups all the documents having the same key
out specifies the location of the map-reduce query result
query specifies the optional selection criteria for selecting documents
sort specifies the optional sort criteria
limit specifies the optional maximum number of documents to be returned

Using MapReduce

Consider the following document structure storing user posts. The document stores user_name of the user and the status of post.

```
{
   "post_text": "tutorialspoint is an awesome website for tutorials",
   "user_name": "mark",
   "status":"active"
}
```


Using MapReduce

Now, we will use a mapReduce function on our posts collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code –

```
>db.posts.mapReduce(
   function() { emit(this.user_id,1); },

   function(key, values) {return Array.sum(values)}, {
      query:{status:"active"},
      out:"post_total"
   }
)
```

Using MapReduce

The above mapReduce query outputs the following result −

```
{
   "result" : "post_total",
   "timeMillis" : 9,
   "counts" : {
      "input" : 4,
      "emit" : 4,
      "reduce" : 2,
      "output" : 2
   },
   "ok" : 1,
}
```


Using MapReduce

The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.

Using MapReduce

To see the result of this mapReduce query, use the find operator −

```
>db.posts.mapReduce(
   function() { emit(this.user_id,1); },
   function(key, values) {return Array.sum(values)}, {
      query:{status:"active"},
      out:"post_total"
   }

).find()
```

Using MapReduce

The above query gives the following result which indicates that both users tom and mark have two posts in active states −

```
{ "_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

In a similar manner, MapReduce queries can be used to construct large complex aggregation queries. The use of custom Javascript functions make use of MapReduce which is very flexible and powerful.

Aggregations

- Aggregations operations process data records and return computed results.
- Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.
- In SQL count(*) and with group by is an equivalent of MongoDB aggregation.

Aggregations

- For the aggregation in MongoDB, you should use aggregate() method.

- Syntax
- Basic syntax of aggregate() method is as follows −

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```


Aggregations

- In the collection you have the following data −

```
{
    _id: ObjectId(7df78ad8902c)
    title: 'MongoDB Overview',
    description: 'MongoDB is no sql database',
    by_user: 'tutorials point',
    url: 'http://www.tutorialspoint.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 100
},
{
    _id: ObjectId(7df78ad8902d)
    title: 'NoSQL Overview',
    description: 'No sql database is very fast',
    by_user: 'tutorials point',
    url: 'http://www.tutorialspoint.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 10
},
{
    _id: ObjectId(7df78ad8902e)
    title: 'Neo4j Overview',
    description: 'Neo4j is no sql database',
    by_user: 'Neo4j',
    url: 'http://www.neo4j.com',
    tags: ['neo4j', 'database', 'NoSQL'],
    likes: 750
},
```

Aggregations

- Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following aggregate() method −

```
> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}])
{ "_id" : "tutorials point", "num_tutorial" : 2 }
{ "_id" : "Neo4j", "num_tutorial" : 1 }
>
```


Aggregations

- Sql equivalent query for the above use case will be select by_user, count(*) from mycol group by by_user.

- In the above example, we have grouped documents by field by_user and on each occurrence of by user previous value of sum is incremented.

Aggregations

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])
$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])

Aggregations

Expression	Description	Example
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])
$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])

Aggregations

Expression	Description	Example
$first	GGets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])
$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

Aggregations

- In UNIX command, shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on.

- MongoDB also supports same concept in aggregation framework. There is a set of possible stages and each of those is taken as a set of documents as an input and produces a resulting set of documents (or the final resulting JSON document at the end of the pipeline). This can then in turn be used for the next stage and so on.

Aggregations

- The MongoDB aggregation pipeline consists of stages. Each stage transforms the documents as they pass through the pipeline.

- Pipeline stages do not need to produce one output document for every input document; e.g., some stages may generate new documents or filter out documents.

Aggregations

- Following are the possible stages in aggregation framework –
- $project – Used to select some specific fields from a collection.
- $match – This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
- $group – This does the actual aggregation as discussed above.
- $sort – Sorts the documents.
- $skip – With this, it is possible to skip forward in the list of documents for a given amount of documents.

Partitioning/Sharding

- Breaking large datasets into smaller ones and distributing datasets and query loads on those datasets are requisites to high scalability.

- If a dataset becomes very large for a single node or when high throughput is required, a single node cannot suffice.

- We need to partition/shard such datasets into smaller chunks and then each partition can act as a database on its own.

Partitioning

- Thus, a large dataset can be spread across many smaller partitions/shards and each can independently execute queries or run some programs.

- This way large executions can be parallelized across nodes(Partitions/Shards)

Why Partitioning ?

- The purpose behind partitioning is to spread data so that execution can be distributed across nodes.
- Along with partitioning, if we can ensure that every partition node takes a fair share, then at least in theory, 5 nodes should be able to handle 5 times as much data and 5 times as much read and write throughput of a single partition.
- If sharding is unfair, then a single node might be taking all the load and other nodes might sit idle. This defeats the purpose of sharding/partitioning. A good partition strategy should avoid Hot spots.

Partition by key-range

- Partition by key-range divides partitions based on certain ranges.
- For example, dividing an Organization based on the first letter of their name.
- So A-C, D-F, G-K, L-P, Q-Z is one of the ways by which whole Organization data can be partitioned in 5 parts.
- Now we know the boundaries of such partition so we can directly query a particular partition if we know the name of an employee.

Partition by key-range

- Ranges are not necessarily evenly spaced.
- As in the above example, partition Q-Z has 10 letters of the alphabet but partition A-C has only three.
- The reason behind such division is to allocate the same amount of data to different ranges.
- Due to the fact that most people have the name starting from letters between A-C, as compared to Q-Z, so this strategy will result in near equal distribution of data across the partition.

Partition by key-range

- One of the biggest benefits of such partitioning is the range queries.

- Suppose I need to find all people whose name starts from letters between R-S, then I only need to send the query to partition R-S.

Partition by key hash

- Key range partition strategy is quite prone to hot spots hence many distributed datastore employs another partitioning strategy.

- The hash value of the data's key is used to find out the partition. A good hash function can distribute data uniformly across multiple partitions.

- Cassandra, MongoDB, and Voldemort are databases employing a key hash-based strategy.

Partition by key hash

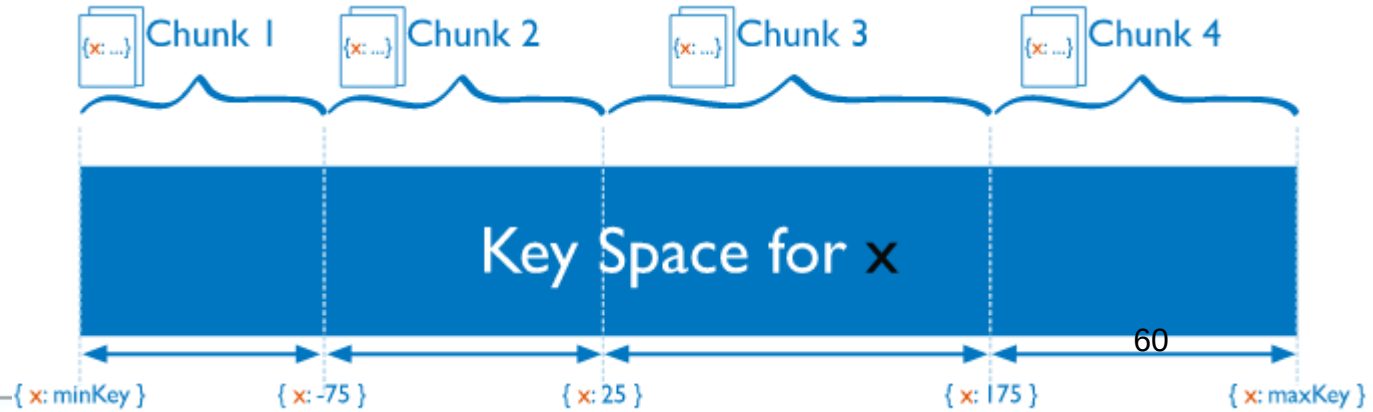
- Hash functions exercised for the purpose of partitioning should be cryptographically strong.
- Java's Object.hashCode() is usually avoided for this purpose as such implementation can lead to a large number of hash collisions.
- Each partition is then assigned a range of key hashes (Rather than range of keys) and all keys which fall within the parameter of partitions range will be stored on that partition. Partition ranges can be chosen to be evenly spaced or can be chosen from a strategy like consistent hashing.

Partition by key hash

- Sadly with the hash of the key, we lose a nice property of key-range partitioning: efficient querying data with some ranges.
- As keys are now scattered to different partitions instead of being adjacent to a single partition.
- Hashing on key indeed reduces hot spots, however, it doesn't eliminate it completely. In case read and writes are for the same key, all requests still end up on the same partition.

Partitioning with Chunks

- MongoDB uses the shard key associated to the collection to partition the data into chunks.
- A chunk consists of a subset of sharded data.
- Each chunk has a inclusive lower and exclusive upper range based on the shard key.



Partitioning with Chunks

- MongoDB splits chunks when they grow beyond the configured chunk size. Both inserts and updates can trigger a chunk split.

- The smallest range a chunk can represent is a single unique shard key value. A chunk that only contains documents with a single shard key value cannot be split.

Partitioning with Chunks

- Initial Chunks

- The sharding operation creates the initial chunk(s) to cover the entire range of the shard key values. The number of chunks created depends on the configured chunk size.

- After the initial chunk creation, the balancer migrates these initial chunks across the shards as appropriate as well as manages the chunk distribution going forward.

Partitioning with Chunks

- Chunk Size

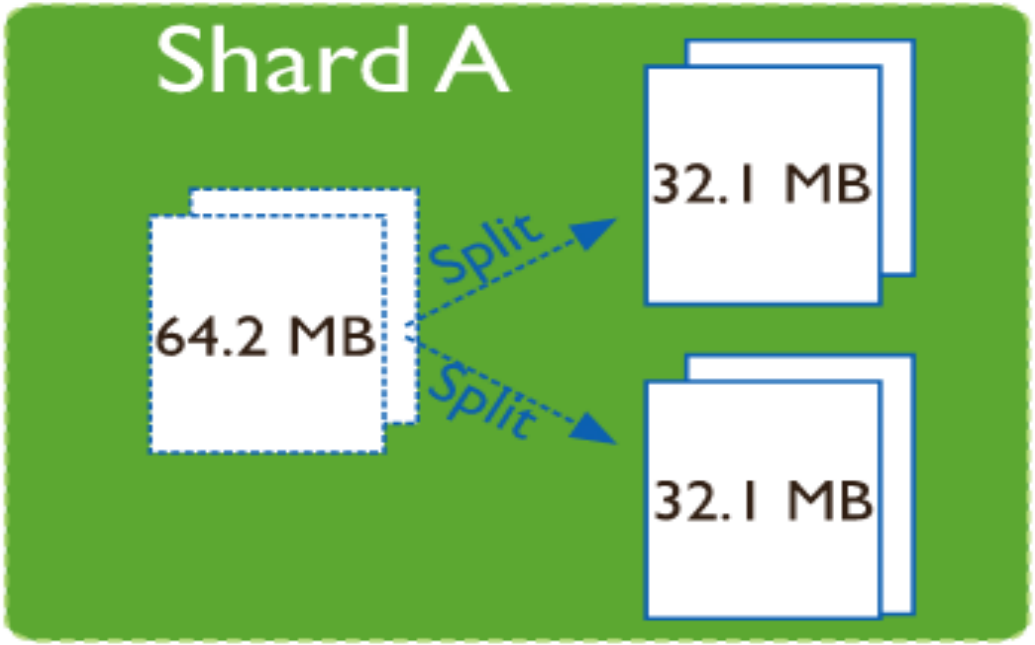
- The default chunk size in MongoDB is 64 megabytes. You can increase or reduce the chunk size. Consider the implications of changing the default chunk size:

Partitioning with Chunks

- Small chunks lead to a more even distribution of data at the expense of more frequent migrations. This creates expense at the query routing (mongos) layer.

- Large chunks lead to fewer migrations. This is more efficient both from the networking perspective and in terms of internal overhead at the query routing layer. But, these efficiencies come at the expense of a potentially uneven distribution of data.

Partitioning with Chunks



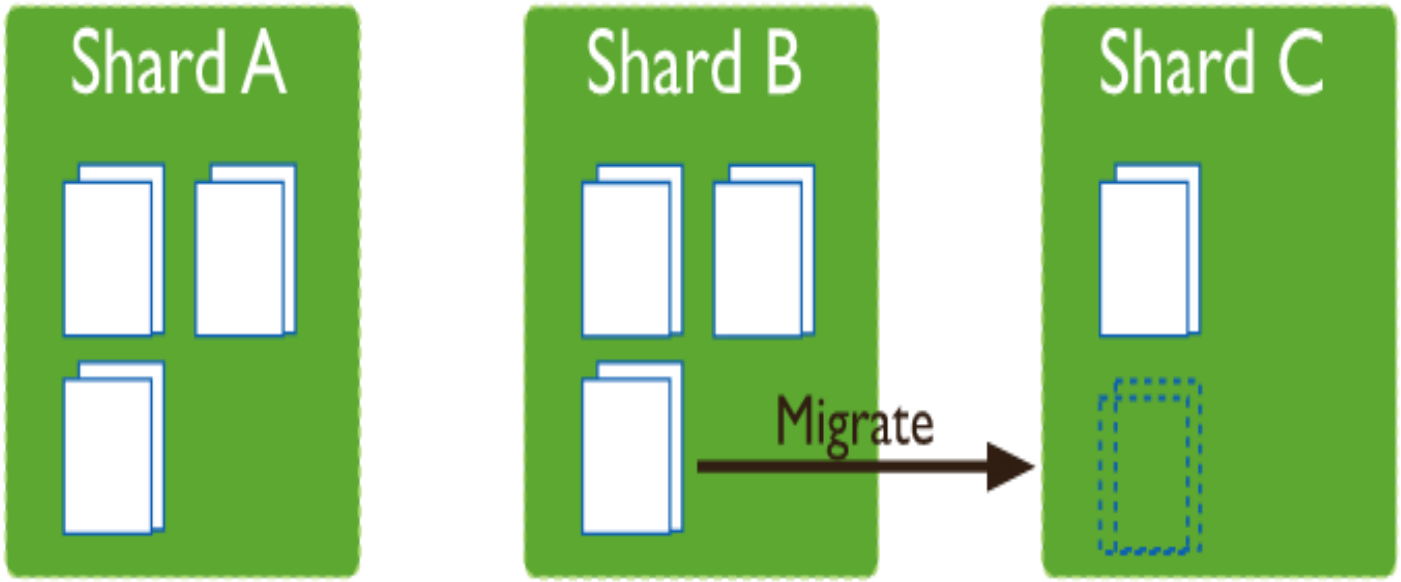
Partitioning with Chunks

- Chunk Migration
- MongoDB migrates chunks in a sharded cluster to distribute the chunks of a sharded collection evenly among shards. Migrations may be either:
- Manual. Only use manual migration in limited cases, such as to distribute data during bulk inserts.
- Automatic. The balancer process automatically migrates chunks when there is an uneven distribution of a sharded collection's chunks across the shards.

Partitioning with Chunks

- Balancing
- The balancer is a background process that manages chunk migrations.
- If the difference in number of chunks between the largest and smallest shard exceed the migration thresholds, the balancer begins migrating chunks across the cluster to ensure an even distribution of data.

Partitioning with Chunks



DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in