IOSR Journal of Engineering (IOSRJEN) **www.iosrjen.org**
ISSN (e): 2250-3021, ISSN (p): 2278-8719
PP 49-57

Efficient Object Database Management for IoT applications: Study of pattern matching under NoSQL databases

[1]Prof.S.N.Sawalkar, [2]Prof.N.D.Shelokar
[1].*Assistant Professor, Department of Computer Science & Engineering, Sipna College of Engineering and Technology,Amravati,*
[2].*Assistant Professor, Department of Computer Science & Engineering, Sipna College of Engineering and Technology,Amravati,*

Abstract: As connected objects creating Internet of Things (IoT), a large extents of data are generated every moment.These data are difficult to handle using traditional structure of database leads to provide object databases(NoSQL approach) due to their high performance, flexibility in scaling, and high availability.
In pattern matching of object databases, embeded librabrary viz. objectDatabase++ provide effective way large number of in build index designes which gives faster access to variety of datatypes from spatial B+ Tree and pattern matching to IoT biometric applications.We compare five of the most popular NoSQL databases namely, Redis, Cassandra, MongoDb, Couchbase and Neo4jwith IoT data management requirements, in NoSQL database for IoT applications.
Keywords: NoSQL,Redis, Cassandra, MongoDb, Couchbase and Neo4j,objectDatabase++

I. Introduction

We live in the world of Internet of Things (IoT) where billions of devices are interconnected generating petabytes of data,information from that data and generating value out of it give competitive advantage to businesses, organizations, governments, and even individual

Nosql Object databases ,Graph Databases and their models are need of hour to store big amount od data that is optimized for IoT, provides high scalability, high performance, and ensures high reliability.

Optimized for IoT

Key Container data model Emphasizes on extension of the typical NoSQL Key-Value store. Key Container model represents data in the form of collections that are referenced by keys. The key and the container are equivalents of the table name and table data in the Relational Database (RDB).

Eg. GridDB,NoSQL Redis

High Performance

New database structures overcomes old structure of performance with the 'Memory first, Storage second' structure where the 'primary' data that is frequently accessed from memory and rest is passed on to disks (SSD and HDD).It localizes the data accessneeded by applications by placing as much 'primary' data in the same block as possible. Based on the application's access pattern and frequency .Allocationof exclusive memory and DB file to each CPU core / thread results in shortened execution time and improves performance .

High Scalability

Traditional RDBMS are built on Scale-Up architecture, On the other hand NoSQL databases focus on Scale-Out architecture.I.e adding smaller nodes to form a large cluster.

High Reliability / Availability

Network partitions, node failures and maintaining consistency are some of the major problems that arise when data is distributed across nodes. Typically, distributed systems adopt 'Master-Slave' or 'Peer-to-Peer' architecture.NoSQL data models works on control cluster architecture integrates and overcomes the limitations of Master-Slave and Peer-to-Peer styles

Comparision between Nosql and RDBMS data models

With the increased volume of data and the expensive cost associated with scaling the relational RDBMSs.It seems that the ACID properties (Atomicity, Consistency, Isolation and Durability) implemented by RDBMSs might set unnecessary constraints to the particular applications and use cases which in turn prohibits

them .From an implementation point of view, theleverage on cloud infrastructures that are adaptive to the scaling, availability and performance requirements.Most of the NoSQL data stores generally do not provide strict ACID properties: updates are eventually propagated, but there are limited guarantees on the consistency of reads. The idea is that by giving up strict ACID constraints high performance and scalability can be achieved.

Feature	**RDBMS**	**NoSQL**
Data Validity	Lower guarantees	Higher guarantees
Query Language	Structured Query Language (SQL)	No declarative query language
Data type	Supports relational data and its relationships are stored in separate tables	Supports unstructured and unpredictable data
Data Storage	Stored in a relational model, with rows and columns. Rows contain all of the information about one specific entry/entity, and columns are all the separate data points	The term "NoSQL" encompasses a host of databases, each with different data storage models. The main ones are: document-based-store, graph-based, key-value-store, column-based-store
Schemas and Flexibility	Each record conforms to fixed schema	Schemas are dynamic. Each 'row' doesn't have to contain data for each 'column'
Scalability	Vertically	Horizontally, meaning across servers
ACID Compliancy	The vast majority of relational databases are ACID compliant.	Sacrifice ACID compliancy for performance and scalability Based on BASE principle

There are various number of classification of NoSQL data models and query models widely popular such as key-value store, column store,document store, graph store

No SQL Databases:Challenges and perspectives

NoSQL what does it mean?

What does NoSQL mean and how do you categorize these databases? NoSQL means Not Only SQL, implying that when designing a software solution or product, there are more than one storage mechanism that could be used based on the needs. NoSQL was a hashtag (#nosql) choosen for a meetup to discuss these new databases.

To explore data modeling techniques, we have to start with a more or less systematic view of NoSQL data models that preferably reveals trends and interconnection

NoSQL databases are often compared by various non-functional criteria, such as scalability, performance, and consistency. This aspect of NoSQL is well-studied both in practice and theory because specific non-functional properties are often the main justification for NoSQL usage and fundamental results on distributed systems

NoSQL Data and Query Models

As already mentioned, there are different types of NoSQL databases. A high-level taxonomy of the NoSQL data stores based on the data model can classify them into four major categories: key-value stores, column-family stores, document stores, and graph databases. This categorization is rather common and we follow the definitions provided in[2]. Concerning querying in NoSQL databases, there is no universal query language which works for all the types of data model. Each database has its own query language which is data-model specific. So far there have been efforts for the creation of a standard query language UnQL[3] (Unstructured Query Language) for NoSQL databases. The syntax for the querying is SQL-like and is used across various document-oriented databases. Also, most of the NoSQL databases allow RESTful interfaces to the data and query APIs.

3.1. Key-value stores

The use of this model implies that the stored values correspond to specific keys. They follow a similar rationale as with maps or dictionaries where data is addressed by a unique key. The key can be synthetic or auto-generated while the value can be any object type, including String, JSON, BLOB etc. The key value model uses a hash table which is comprised of a unique key and a pointer to a particular item of data. The values are stored as uninterpreted byte arrays, therefore the keys are the only way to retrieve stored data. The grouping of key value pairs into collections is the only option to add some kind of structure to the data model. Also, key value stores supports simple operations, which are applied on key attributes only. Most key value stores hold the whole data in memory (in memory key value stores-IMKVS), which make them good candidates for caching and high throughput system requirements. The key value stores considered in this paper are Redis[3], MemCached[4], and Dynamo[5].

The query operations for stored objects are associated with a key. The type of operations that are offered through APIs include functions like get, put and delete. These interfaces enable simple queries to be

performed (as complex queries are not supported) rendering the query language unnecessary. For instance, Dynamo exposes two operations:
a) get (key) which returns a list of objects and a context and b) put (key, context, object) which performs a write operation.

Column family stores

The column family stores data model is described by Google in BigTable[6] as "sparse, distributed, persistent multidimensional sorted map". In this map, an arbitrary number of key value pairs can be stored within rows. To support versioning and achieve performance, multiple versions of the values are sorted in chronological order. The columns can be grouped to column families to support organization and partitioning. The column family stores data model is more suitable for applications dealing with huge amounts of data stored on very large clusters, because the data model can be partitioned very efficiently. They can be visualized the same way as RDMS do, due to their table format. The main difference lies in their handling of null values, i.e. what constitutes the substantial difference is that RDBMS would store a null value in each column but column family stores only store a key value pair in a row only if a dataset needs it. Column family stores considered in this paper are PNUTS[7] and Cassandra[8]. Cassandra's data model differs from the typical column family stores, in that it maintains one more dimension called super column. A super column family can be visualized as a column family within a column family[2].

Column family stores support range queries and the use of regular expression on the indexed values or on the row keys for querying. This has an impact on the extent to which the databases are affected, i.e. operations are only affecting the keys that are related to the query rather –in an extreme case for the sake of the example- the entire row from disk. Apart from the typical accessors and mutators, the column-oriented databases frequently support operations like OR, IN, AND. A relevant example is the Cassandra API which provides three main operations: get (table, key, columnName), insert (table, key, rowMutation), delete (table, key, columnName).

Document stores

In Document stores the data are again a collection of key value pairs, but now they are compressed as a document, i.e. entities that provide some structure and encoding of the managed data. All keys in the document have to be unique, and every document contains a special key "ID", which is also unique within a collection of documents and therefore identifies a document explicitly. Document stores do not have any schema restrictions and support multi attribute lookups on records which may have different kinds of key value pairs. Document stores considered in this paper are MongoDB[9] and CouchDB[10].

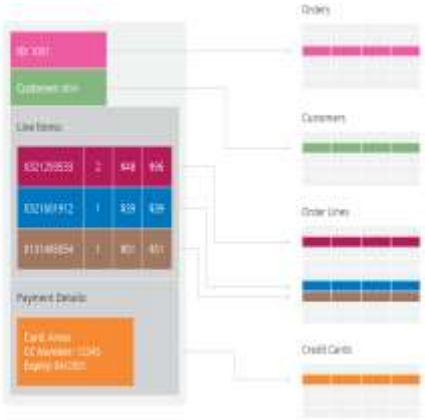
Document stores embeds attribute metadata associated with stored content, which essentially - in contrast to key value stores, provides a way to query the data based on the contents. Range queries, indexing and nested document querying are supported. Also, allows the use of operations like OR, AND and BETWEEN and queries can be implemented as Map Reduce jobs. MongoDB for example supports operators like: create, insert, read, update, remove. It also supports manual indexing, indexing on embedded documents and index location-based data.

Graph stores

In graph stores a flexible graphical representation is used which is perfect to address scalability concerns. Graph structures are used with edges, nodes and properties which provides index-free adjacency. Nodes and edges consist of objects with embedded key value pairs. The range of keys and values can be defined in a schema, whereby the expression of more complex constraints can be described easily. Data can be easily transformed from one model to the other using a Graph Base NoSQL database. Graph databases are specialized on efficient management of heavily linked data and are optimized for highly connected data[11]. Use cases for graph databases are location based services, knowledge representation and path finding problems raised in navigation systems, recommendation systems and all other use cases which involve complex relationships[2]. Graph stores considering this paper is Neo4j[11] and HyperGraphDB[12].

Applications based on data with many relationships are more suited for graph databases, since cost intensive operations like recursive joins can be replaced by efficient traversals such as graph traversal and graph pattern matching techniques. In graph traversal, the query processing starts from one node and then the other nodes are traversed as per the description of the query. The graph pattern matching technique tries to locate in the original graph, the defined pattern which is to be searched for. For example in Neo4j each vertex and edge in the graph store a “mini-index” of the objects connected to it. This means that the size of the graph has no performance impact upon a traversal and the cost of a local step (hop) remains the same. There is also global adjacency index but it is only used when trying to find the starting point of a traversal. Indexes are required in

order to quickly retrieve vertices based on their values. They provide a starting point at which to begin a traversal.



Why NoSQLDatabases?

Application developers have been frustrated with the impedance mismatch between the relational data structures and the in-memory data structures of the application. Using NoSQL databases allows developers to develop without having to convert in-memory structures to relational structures.

There is also movement away from using databases as integration points in favor of encapsulating databases with applications and integrating using services. The rise of the web as a platform also created a vital factor change in data storage as the need to support large volumes of data by running on clusters. Relational databases were not designed to run efficiently on clusters.The data storage needs of an ERP application are lot more different than the data storage needs of a Facebook or an Etsy, for example.

Aggregate Data Models:

Relational database modelling is vastly different than the types of data structures that application developers use. Using the data structures as modelled by the developers to solve different problem domains has given rise to movement away from relational modelling and towards aggregate models, most of this is driven by Domain Driven Design, a book by Eric Evans. An aggregate is a collection of data that we interact with as a unit. These units of data or aggregates form the boundaries for ACID operations with the database, Key-value, Document, and Column-family databases can all be seen as forms of aggregate-oriented database.

Aggregates make it easier for the database to manage data storage over clusters, since the unit of data now could reside on any machine and when retrieved from the database gets all the related data along with it. Aggregate-oriented databases work best when most data interaction is done with the same aggregate, for example when there is need to get an order and all its details, it better to store order as an aggregate object but dealing with these aggregates to get item details on all the orders is not elegant.Aggregate-oriented databases make inter-aggregate relationships more difficult to handle than intra-aggregate relationships. Aggregate-ignorant databases are better when interactions use data organized in many different formations. Aggregate-oriented databases often compute materialized views to provide data organized differently from their primary aggregates. This is often done with map-reduce computations, such as a map-reduce job to get items sold per day.

Distribution Models: Aggregate oriented databases make distribution of data easier, since the distribution mechanism has to move the aggregate and not have to worry about related data, as all the related data is contained in the aggregate.

There are two styles of distributing data:A)Sharding: Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.B)**Replication:** Replication copies data across multiple servers, so each bit of data can be found in multiple places. Replication comes in two forms,Master-slave replication makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads.Peer-to-peer replication allows writes to any node; the nodes coordinate to synchronize their copies of the data.

Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single server creating a single point of failure. A system may use either or both techniques. Like Riak database shards the data and also replicates it based on the replication factor.

CAP theorem: In a distributed system, managing consistency(C), availability(A) and partition toleration(P) is important, Eric Brewer put forth the CAP theorem which states that in any distributed system we can choose only two of consistency, availability or partition tolerance. Many NoSQL databases try to provide options where the developer has choices where they can tune the database as per their needs. For example if you consider Riak a distributed key-value database. There are essentially three variables r, w, n where

- r=number of nodes that should respond to a read request before its considered successful.
- w=number of nodes that should respond to a write request before its considered successful.
- n=number of nodes where the data is replicated aka replication factor.

In a Riak cluster with 5 nodes, we can tweak the r,w,n values to make the system very consistent by setting r=5 and w=5 but now we have made the cluster susceptible to network partitions since any write will not be considered successful when any node is not responding. We can make the same cluster highly available for writes or reads by setting r=1 and w=1 but now consistency can be compromised since some nodes may not have the latest copy of the data. The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency of data. Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.NoSQL databases provide developers lot of options to choose from and fine tune the system to their specific requirements. Understanding the requirements of how the data is going to be consumed by the system, questions such as is it read heavy vs write heavy, is there a need to query data with random query parameters, will the system be able handle inconsistent data.

Understanding these requirements becomes much more important, for long we have been used to the default of RDBMS which comes with a standard set of features no matter which product is chosen and there is no possibility of choosing some features over other. The availability of choice in NoSQL databases, is both good and bad at the same time. Good because now we have choice to design the system according to the requirements. Bad because now you have a choice and we have to make a good choice based on requirements and there is a chance where the same database product may be used properly or not used properly.

An example of feature provided by default in RDBMS is transactions, our development methods are so used to this feature that we have stopped thinking about what would happen when the database does not provide transactions. Most NoSQL databases do not provide transaction support by default, which means the developers have to think how to implement transactions, does every write have to have the safety of transactions or can the write be segregated into "critical that they succeed" and "its okay if I lose this write" categories. Sometimes deploying external transaction managers like ZooKeeper can also be a possibility.

Types of NoSQL Databases: NoSQL databases can broadly be categorized in four types.

Key-Value databases

Key-value stores are the simplest NoSQL data stores to use from an API perspective. The client can either get the value for the key, put a value for a key, or delete a key from the data store. The value is a blob that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled.

Some of the popular key-value databases are Riak, Redis (often referred to as Data Structure server), Memcached and its flavors, Berkeley DB, upscaledb (especially suited for embedded use), Amazon DynamoDB (not open-source), Project Voldemort and Couchbase.

All key-value databases are not the same, there are major differences between these products, for example: Memcached data is not persistent while in Riak it is, these features are important when implementing certain solutions. Lets consider we need to implement caching of user preferences, implementing them in memcached means when the node goes down all the data is lost and needs to be refreshed from source system, if we store the same data in Riak we may not need to worry about losing data but we must also consider how to update stale data. Its important to not only choose a key-value database based on your requirements, it's also important to choose which key-value database.

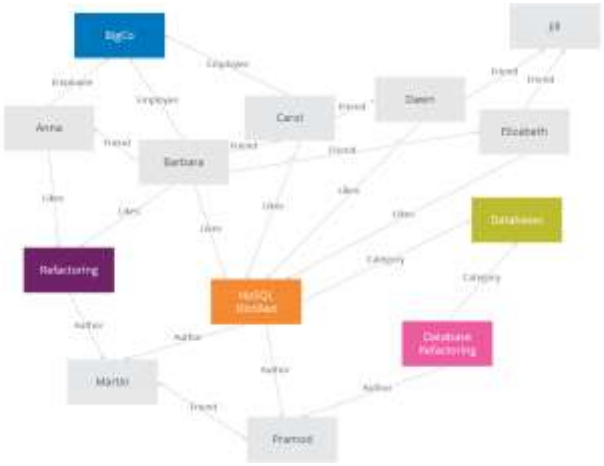
Document databases

Documents are the main concept in document databases. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. The documents stored are similar to each other but do not have to be exactly the same. Document databases store documents in the value part of the key-value store; think about document databases as key-value stores where the value is examinable. Document databases such as MongoDB provide a rich query language and constructs such as database, indexes etc allowing for easier transition from relational databases.Some of the popular document databases we have seen are MongoDB, CouchDB , Terrastore, OrientDB, RavenDB, and of course the well-known and often reviled Lotus Notes that uses document storage.

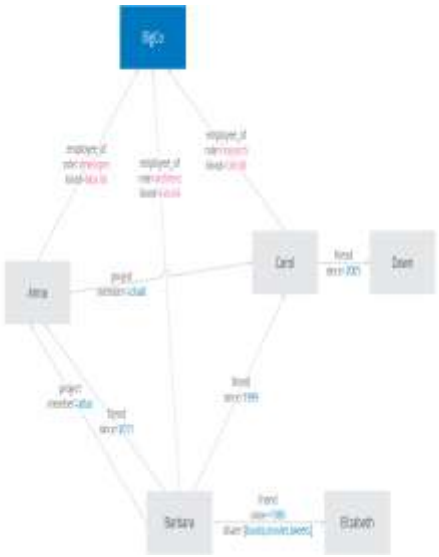
Column family stores

Column-family databases store data in column families as rows that have many columns associated with a row key (Figure 10.1). Column families are groups of related data that is often accessed together. For a Customer, we would often access their Profile information at the same time, but not their Orders.Each column family can be compared to a container of rows in an RDBMS table where the key identifies the row and the row consists of multiple columns. The difference is that various rows do not have to have the same columns, and columns can be added to any row at any time without having to add it to other rows.When a column consists of a map of columns, then we have a super column. A super column consists of a name and a value which is a map of columns. Think of a super column as a container of columns.Cassandra is one of the popular column-family databases; there are others, such as HBase, Hypertable, and Amazon DynamoDB. Cassandra can be described as fast and easily scalable with write operations spread across the cluster. The cluster does not have a master node, so any read and write can be handled by any node in the cluster.

Graph Databases



Graph databases allow you to store entities and relationships between these entities. Entities are also known as nodes, which have properties. Think of a node as an instance of an object in the application. Relations are known as edges that can have properties. Edges have directional significance; nodes are organized by relationships which allow you to find interesting patterns between the nodes. The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships.Usually, when we store a graph-like structure in RDBMS, it's for a single type of relationship ("who is my manager" is a common example). Adding another relationship to the mix usually means a lot of schema changes and data movement, which is not the case when we are using graph databases. Similarly, in relational databases we model the graph beforehand based on the Traversal we want; if the Traversal changes, the data will have to change.In graph databases, traversing the joins or relationships is very fast. The relationship between nodes is not calculated at query time but is actually persisted as a relationship. Traversing persisted relationships is faster than calculating them for every query.



Nodes can have different types of relationships between them, allowing you to both represent relationships between the domain entities and to have secondary relationships for things like category, path, time-trees, quad-trees for spatial indexing, or linked lists for sorted access. Since there is no limit to the number and kind of relationships a node can have, they all can be represented in the same graph database.

Relationships are first-class citizens in graph databases; most of the value of graph databases is derived from the relationships. Relationships don't only have a type, a start node, and an end node, but can have properties of their own. Using these properties on the relationships, we can add intelligence to the relationship—for example, since when did they become friends, what is the distance between the nodes, or what aspects are shared between the nodes. These properties on the relationships can be used to query the graph.

Since most of the power from the graph databases comes from the relationships and their properties, a lot of thought and design work is needed to model the relationships in the domain that we are trying to work with. Adding new relationship types is easy; changing existing nodes and their relationships is similar to data migration, because these changes will have to be done on each node and each relationship in the existing data.

There are many graph databases available, such as Neo4J, Infinite Graph, OrientDB, or FlockDB (which is a special case: a graph database that only supports single-depth relationships or adjacency lists, where you cannot traverse more than one level deep for relationships).

Choosing NoSQL database

Given so much choice, how do we choose which NoSQL database? As described much depends on the system requirements, here are some general guidelines:

 Key-value databases are generally useful for storing session information, user profiles, preferences, shopping cart data. We would avoid using Key-value databases when we need to query by data, have relationships between the data being stored or we need to operate on multiple keys at the same time.

 Document databases are generally useful for content management systems, blogging platforms, web analytics, real-time analytics, ecommerce-applications. We would avoid using document databases for systems that need complex transactions spanning multiple operations or queries against varying aggregate structures.

 Column family databases are generally useful for content management systems, blogging platforms, maintaining counters, expiring usage, heavy write volume such as log aggregation. We would avoid using column family databases for systems that are in early development, changing query patterns.

 Graph databases are very well suited to problem spaces where we have connected data, such as social networks, spatial data, **routing information for goods and money, recommendation engines**

Schema-less ramifications

All NoSQL databases claim to be schema-less, which means there is no schema enforced by the database themselves. Databases with strong schemas, such as relational databases, can be migrated by saving each schema change, plus its data migration, in a version-controlled sequence. Schema-less databases still need careful migration due to the implicit schema in any code that accesses the data.

Schema-less databases can use the same migration techniques as databases with strong schemas, in schema-less databases we can also read data in a way that's tolerant to changes in the data's implicit schema and

use incremental migration to update data, thus allowing for zero downtime deployments, making them more popular with 24*7 systems.

II. Conclusion

All the choice provided by the rise of NoSQL databases does not mean the demise of RDBMS databases. We are entering an era of polyglot persistence, a technique that uses different data storage technologies to handle varying data storage needs. Polyglot persistence can apply across an enterprise or within a single application.

For more details, read NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence by Pramod Sadalage and Martin Fowler.

References

[1]. Grolinger K, Higashino WA, Tiwari A, Capretz MA. Data management in cloud environments: NoSQL and NewSQL data stores. Journal of Cloud Computing: Advances, Systems and Applications. 2013;2(1):1.
[2]. Hecht R, Jablonski S, editors. Nosql evaluation. International conference on cloud and service computing; 2011: IEEE.
[3]. UnQL http://unql.sqlite.org/index.html/timeline.
[4]. Carra D, Michiardi P, editors. Memory partitioning in memcached: An experimental performance analysis. 2014