WHITE PAPER

Big Data with NoSQL

aspire SYSTEMS
attention. always.

CONTENTS

This paper discusses on the challenges associated in using traditional RDBMS for Big Data scenarios and selecting the right NoSQL Database ideally suited for specific Big Data use cases.

Boundaries of Traditional Databases for Big Data

Most of the traditional databases are RDBMS solutions. They have a rigid schema design and data is entered into tables based on that schema. Typically, a SQL-based query is used to read/write data to such databases.

So, what are the limiting factors with these databases for Big Data scenarios? There are many!

A rigid schema means storing data into tables is not flexible. This results in difficulty in scaling the databases for larger sizes. Let's consider an example:

Assume an order management system for a large scale shopping website, where you need to store orders and the items within the orders. In a typical RDBMS database, this will be stored in a table with schema similar to this:

In order to display the names of customers, a JOIN would be required typically with a Customers table. This results in 2 joins – one with OrderItem and another with Customers.

When data is stored in this format, once a physical storage limit is reached, traditional RDBMS databases cannot horizontally scale. One has to shard the data manually by writing specifically into relevant table. For example, in the above scenario,orders for user ids 1 – 500000 could be written into Feeds_A table and 500001 to 1000000 could be written into Feeds_B table. This is called manual sharding.

Manual sharding is always cumbersome. With manual sharding, things like how various JOINs are handled, how data that is present in other related tables are managed and so on should also be handled manually. This makes an RDBMS database not a suitable candidate for the above scenario.

Why NoSQL?

NoSQL databases take a different approach to solve Big Data problems. Many inherent architecture and design considerations – like schema-less design, compromise on ACID properties, predominantly RAM-based processing for large set of data, key-value/document based storage – have been taken into account in most of the NoSQL databases.

For the example discussed in section 2, a typical NoSQL document based storage (e.g. MongoDB) would store data in this manner:

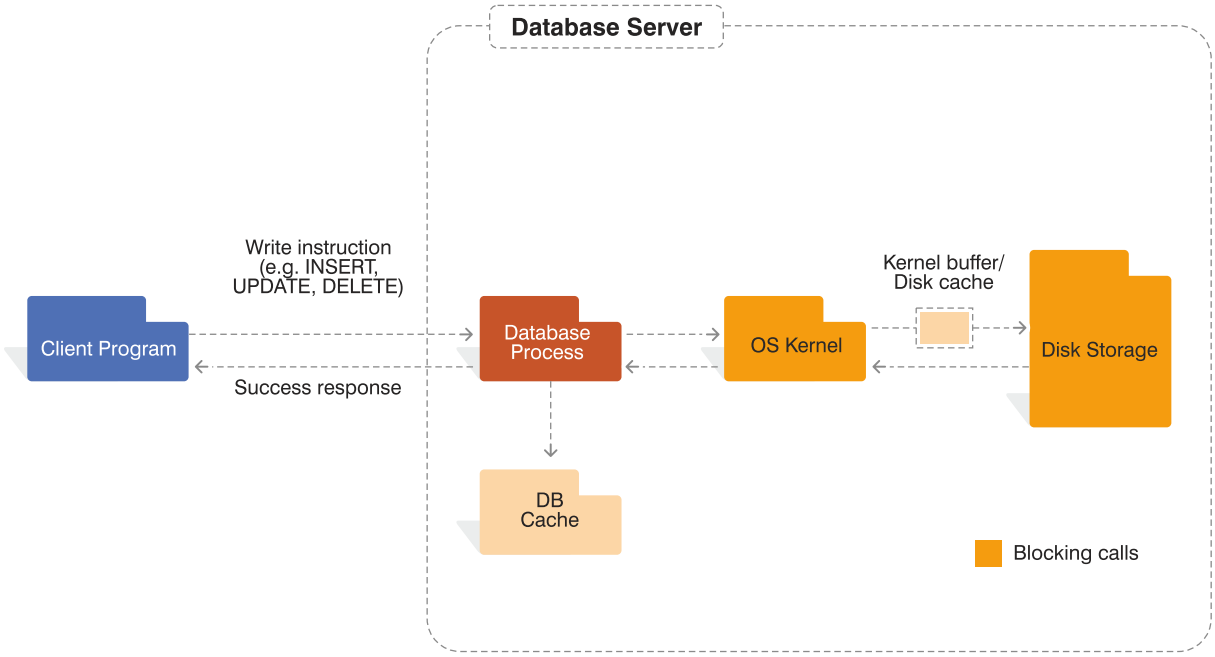
```
{
 "Id": 100001,
 "CustId": 14,
 "FirstName": "John",
 "LastName": "Doe",
 "Items": [
  {
  "Id": 1,
  "ItemId": 43,
  "ItemName": "Logitech Mouse",
  "Quantity": 2
  },
  {
  "Id": 2,
  "ItemId": 52,
  "ItemName": "iPad 4",
  "Quantity": 1
  }
 ]
}
```

This results in a single hit to database. But more importantly, this can scale well horizontally for Big Data scenarios. When the size of the database runs into a few terabytes, a NoSQL solution like MongoDB can handle this better.

Understanding the Core Architectural difference

Following diagrams depict the most important difference between RDBMS databases and NoSQL databases that has a significant impact while solving a Big Data requirement.

RDBMS



ACID Compliant

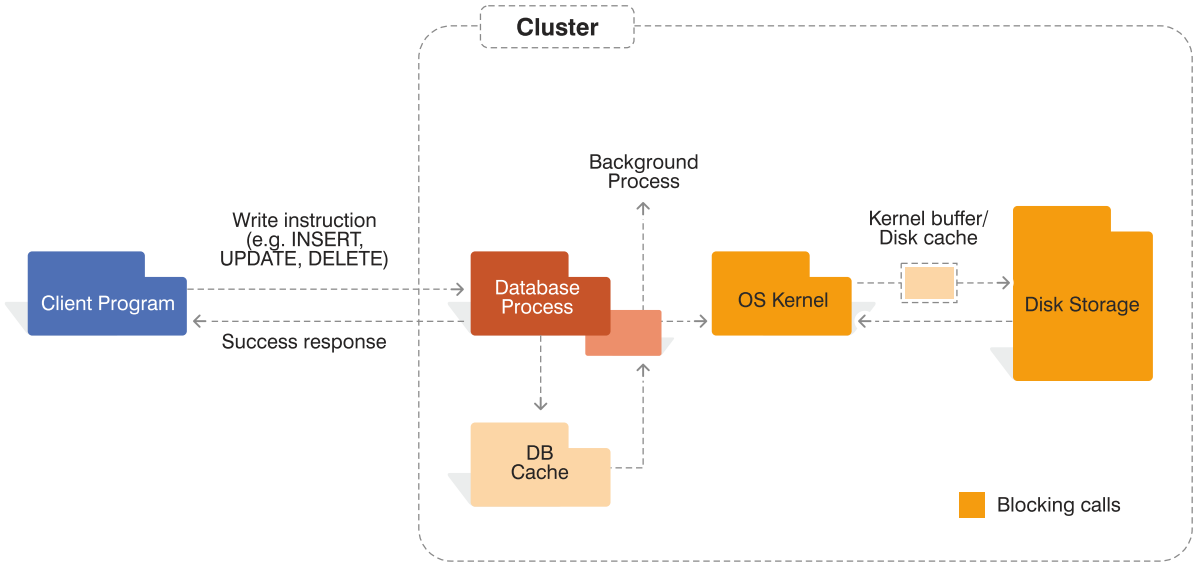
RDBMS databases always guarantee durability in one of these two ways:

1. By making sure the response to writes are ALWAYS received after a full fsync() is performed by the OS.

2. By writing into an append-only log that has all required data that could be written into main database tables even if there is a crash.

With this approach, an RDMBS solution should always make sure that it persists data in an ACID compliant manner. Particularly, all replicas / saves should be required to guarantee that it is written, so that consistency is achieved.

RDBMS



Fire & Forget

Generally, NoSQL databases make some compromises on certain aspects of ACID-ness.

Redis

Redis is predominantly RAM-based provider, and it sacrifices durability (though it can be explicitly enabled through configuration). Once data is sent to database process, the response returns immediately irrespective of whether it is written into the disk, but it is available still in RAM.

Note that this doesn't mean the data is NOT persisted. This just means during rare circumstances when things go wrong (e.g. server crash), some items might not have been written into the disk. Even this can be overcome in most cases by enabling the AOF (Append Only File) that keeps track of entries before writing into final data files.

MongoDB

MongoDB follows an eventual consistency model like many other NoSQL databases.

Once data reaches the master in a cluster, MongoDB writes it into the journal (similar to Write Ahead Log in RDBMS) and returns response immediately. This makes sure data is ultimately persisted, but reads are not consistent. Replication to slaves/replicas doesn't happen immediately, but could be delayed by a few milliseconds or seconds. That is, the data becomes eventually consistent across all nodes / replicas in a cluster.

NoSQL Approach for Big Data Scenarios

While planning a NoSQL based solution for your Big Data project, it is important to consider the following evaluation criterias:

1) Suitability of a solution that fits best to the requirement.

a) Document based application

b) Publish / Subscribe

c) Real-time updates

d) Analytics Processing

2) The technical features that cannot be handled (or poorly handled) by RDBMS solution.

a) Scalability

b) Persistence

c) ACID compliance

Based on above parameters, one must select the apt NoSQL solution for the Big Data scenarios carefully.

A. Suitability of a Solution

Not all NoSQL databases are created equal. Each could cater to a different kind of Big Data application. For example, Redis is a key value pair that is a specialized for publish/subscribe solution. We will now look at some common application requirements that deal with Big Data and analyze a possible NoSQL based solution for them.

A. i) Document based applications

Some RDBMS applications are mainly document stores. The document here not only refers to a typical file like Word, Excel, PDF and so on, but also the unit that stores information in a free flowing format rather than on a strict schema basis.

Consider an order management system for a large online shopping platform.

- Number of orders per year ~= millions.
- Throughput per second ~= few hundreds.

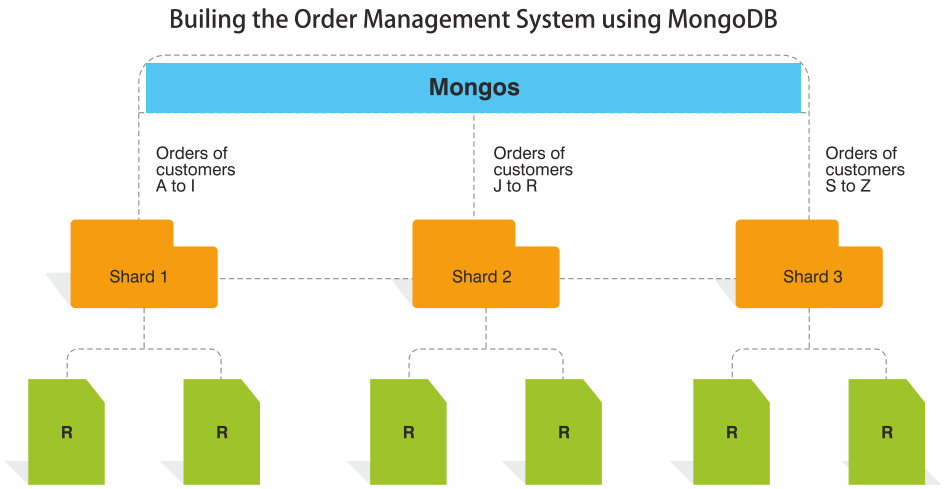
At such scales, both the real time transactions and analytics processes will require databases that are highly scalable.Such a system could be built using MongoDB.

MongoDB

MongoDB is a typical document based database that is ideal for such applications.

Each MongoDB document contains key-value pairs. It can also store sub-documents (also called embedded documents). The storage format BSON (Binary JSON) is similar to JSON format, so it is easy to integrate it with most of the applications that also provide a REST API.

By default, a MongoDB document can also store binary objects like PDF documents, images, videos up to 16 MB. If the size exceeds that limit, MongoDB provides a feature called GridFS which splits the binary objects into chunks and stores them in different files. This also enables advanced features like skipping to a particular moment in a video / audio, visiting a specific page in a PDF document etc.



MongoDB has configurable sharding that helps in distributing writes across shards.

A server instance can be marked as a shard based on a shard key. MongoDB uses an internal service called Mongos that handles the redirection of data into relevant shards. In the above example, if the [CustFirstName] is marked as the shard key, then Mongos sends the order that belongs to a particular customer to its relevant server instance.

MongoDB supports two kinds of sharding – Range based and Hash based. Range based sharding take a direct range of values in a shard key and distributes data accordingly. Hash based sharding computes a unique hash value based on the shard key and sends it to the correct instance. The above is an example of range based sharding.

A. ii) Publish / Subscribe

In large scale enterprise applications where publish / subscribe feature is required without the need to go for complex messaging platforms, a simpler NoSQL based solution could be a better fit.

Redis

Redis is a key-value store that stores data mainly in RAM with option to store on disk. It uses an algorithm similar to LRU (Least Recently Used) to page data on to disk when it is not actively being used. This helps in gaining better performance. Also, this means it requires more RAM to run.

Building a Notification Service with Redis

Consider an automobile company with multiple product divisions wants to build a notification service for different teams based on different product lines or brands (e.g. Chevrolet, Cadillac, Buick). The goal is to publish companywide messages and get them notified to users properly. Redis can be used to accomplish this efficiently.

A client can just subscribe to a channel like this:

SUBSCRIBE Cadillac

This will enable the client to receive notifications published into [Cadillac] channel. Even wild-carding is possible. For example, SUBSCRIBE Cadillac.Europe.* will subscribe to only Europe geography notifications of the Cadillac division, but not others.

A. iii) Real-time Updates

Applications that receive huge amount of data real-time (e.g. at least a few thousand records per second) and needs to update a dashboard / leader board based on the latest data are good fit to be implemented as a NoSQL solution. There are quite a number of NoSQL databases that meet this need.

DynamoDB

Assume an application that needs to update latest geo-location data. Not only it sends huge number of such records every second, but also the database should be able to handle that traffic pretty fast.

DynamoDB is a hosted database solution from Amazon that solves this problem handily. In DynamoDB, one can provision a "Throughput" for both reads and writes, which tells it how many items could be received by the database in a unit of time. If you require 1000 items (each item up to 1 KB) to be stored into a DynamoDB table per second, you configure that number as a throughput and DynamoDB will be able to handle that load seamlessly.

This throughput configuration can be dynamic too. It can be increased and decreased multiple times in a day with some limitations. So, an application can increase throughput during peak traffic load and decrease it during non-peak times thereby resulting in better savings.

Redis

Consider an onlinegame that keeps receiving scores from various users across the globe. The numbers of users could be in millions. The number of active games at a moment in time could also be in millions. Assume each game runs for a few minutes before a high score is published.

This could result in a few thousand scores being sent to the game serverin a second.

Such applications would have a leaderboard / dashboard that represent the top scores at the moment, and also the top scores of individuals too. With few thousand entries per second, the data is too big to be handled reasonably well by an RDBMS database.

Redis would be helpful in creating such a leaderboard / dashboard.

Redis has built-in data structures like list, set, sorted set and so on. For the above use cases, scores could be sent to a sorted set using a command similar to this:

ZADD "Scores Dashboard" 900 "John Doe"

ZADD adds data into a sorted set based on a key (Dashboard Scores) sorted by a score (900) and the value (John Doe).

With this, data can be readily stored with relevant sorting in place, and hence the score updates on the screen would be real-time.

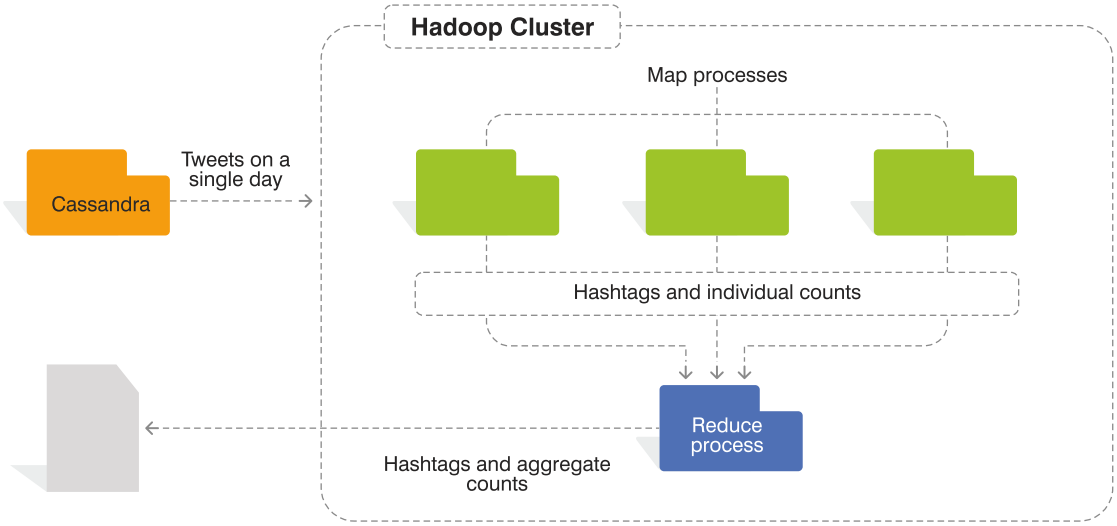
A. iv) Analytics Processing

Applications that require analytics can be implemented using a variety of Big Data solutions.

A Near Real-time Tweets Analytics

Consider an organization that wants to analyze tweets of its followers based on certain keywords or hashtags which could help it in creating targeted campaigns and advertisements. With so many users filling the timeline at a rapid pace, it is difficult to gain insights from such plethora of information using traditional databases.

For this,Cassandra could be used to persist the data that comes in, and then in real-time it can be analyzed using a Map-Reduce framework like Hadoop.



MapReduce is an algorithm where large scale data can be processed by splitting it into chunks and mapping them to various processes or server instances which could further process the data as required. The output of the "map" process is then collected and aggregated (i.e. reduced) to get the final output.

As observed in the diagram above, the tweets of a single day – which could run into millions – could be fed into Hadoop MapReduce from Cassandra. Hadoop then performs the map and reduce steps and arrives at the final count of tweets for hash tags, which could be written to a target file or displayed on screen.

Cassandra has built-in support for Hadoop, so performing map-reduce is a breeze.

B. Technical Considerations

Apart from suitability of a solution for requirements, one must also look into the technical strengths / weaknesses of a NoSQL flavor and adopt a suitable approach.

B. i) Scalability

Scalability can be viewed from three perspectives – read, write and persistence. (Write and Persistence can be combined together too and viewed as just "Write" scaling.)

While scaling the reads is more or less the same for both RDBMS and NoSQL solutions, scaling the writes is different and that is where most of the scalability issues crop up.

RDBMS databases are designed to scale up. If you want to handle more writes or need more storage, you need to upgrade the resources in the existing machine. But NoSQL databases are designed to scale out. To handle larger number of writes or storage, one just needs to add additional boxes in an existing cluster.

Refer section 2 which discusses sharding – which is a write-scaling technique.

B. ii) Persistence

Most of the NoSQL databases are predominantly RAM based. They serve data mostly from RAM, and go to disk only if required. This is one of the reasons NoSQL databases perform better than those of RDBMS variety. This also implies that they require servers with higher RAM capacity.

B. iii) ACID Compliance

RDBMS databases are mostly ACID compliant. You get Atomicity, Consistency, Integrity and Durability at all times. But most NoSQL solutions are not fully ACID compliant. They sacrifice some aspects to achieve better performance. Let us discuss this with an example from a MongoDB cluster.

Consider a MongoDB cluster with master/slave replication and also setup with sharding. Data will always be sharded across different servers, and will also be replicated to different slaves. But the replication could be lagging a bit, whereas the whole cluster doesn't give a guarantee that the slaves have the latest data as master. i.e. it could be lagging anywhere between a few milliseconds to a few seconds.

This model is called Eventual Consistency. It is a fine model for many types of applications like social networking sites, blogs, community forums and even analytics. But real-time financial applications (like banking and insurance) may not be suitable candidates for such solutions.

Close to ACID Compliance

Cassandra has an architecture that is close enough to ACID compliance.

In general, Cassandra too is eventually consistent like other NoSQL databases. But it also has an option called "Tunable Consistency", which lets you choose consistency for a particular transaction. Tunable consistency allows the transaction to be fully isolated and provides a check for other clients / connections to see if the updated data is available.

Consider a large scale publishing platform where a user sends an invite to a friend whose email is johndoe@gmail.com. Meanwhile another user sends an invite to John Doe too. If you want to

prevent multiple users from creating invite in this case, Cassandra provides an option while creating the record:

```
INSERT INTO invitations (user_id, user_email)
VALUES (“John Doe”, “johndoe@gmail.com”)
IF NOT EXISTS;
```

With IF NOT EXISTS, Cassandra looks across the nodes to make sure this transaction is fully consistent. Obviously, this should be used in cases where this is absolutely required.

Conclusion

The strength of RDBMS lies in its ACID compliance which reflects well in its transactional nature of processing, while the major drawback is scaling to larger volumes of data with decent performance.

NoSQL databases provide optimal solutions for most Big Data requirements where both volume of data and velocity of requests are huge. There are many Big Data problems that do not require ACID compliance, and NoSQL databases come in handy in such cases.

For your Big Data requirements, it is best to evaluate the right NoSQL databases according to your use cases and choose the one that fits your goals applying the evaluation guidelines.