Semester : ____VII_____ Subject : _____BDA_____ Academic Year: 2024 - 2025

NOSQL SYSTEMS TO HANDLE BIG DATA PROBLEMS

OR

FOUR WAYS THAT NOSQL SYTEMS HANDLE BIG DATA PROBLEMS

1. **Moving queries to the data, not data to the queries**

With the exception of large graph databases, most NoSQL systems use commodity processors that each hold a subset of the data on their local shared-nothing drives. When a client wants to send a general query to all nodes that hold data, it's more efficient to send the query to each node than it is to transfer large datasets to a central processor. This simple rule helps you understand how NoSQL databases can have dramatic performance advantages over systems that weren't designed to distribute queries to the data nodes. Keeping all the data within each data node in the form of logical documents means that only the query itself and the final result need to be moved over a network. This keeps your big data queries fast.

2. **Using hash rings to evenly distribute data on a cluster:**

One of the most challenging problems with distributed databases is figuring out a consistent way of assigning a document to a processing node. Using a hash ring technique to evenly distribute big data loads over many servers with a randomly generated 40-character key is a good way to evenly distribute a network load.

Hash rings are common in big data solutions because they consistently determine how to assign a piece of data to a specific processor. Hash rings take the leading bits of a document's hash value and use this to determine which node the document should be assigned. This allows any node in a cluster to know what node the data lives on and how to adapt to new assignment methods as your data grows. Partitioning keys into ranges and assigning different key ranges to specific nodes is known as keyspace management.

3. **Using replication to scale reads:**

Databases use replication to make backup copies of data in real time. Using replication allows you to horizontally scale read requests. Figure shows this structure.

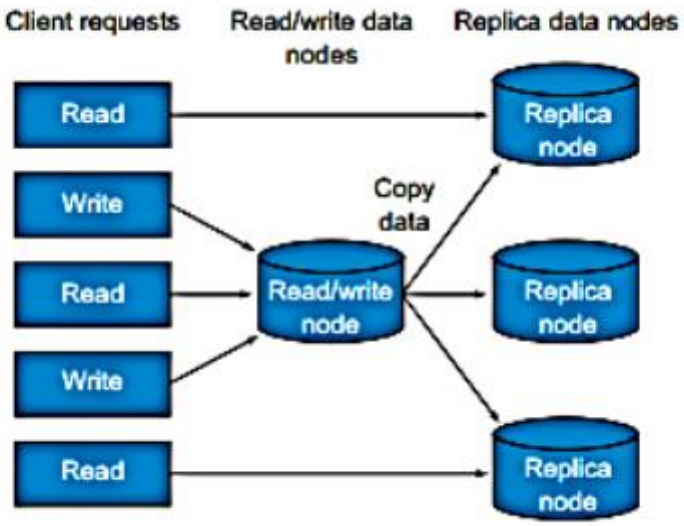

Figure: How you can replicate data to speed read performance in NoSQL systems.

This replication strategy works well in most cases. There are only a few times when you must be concerned about the lag time between a write to the read/write node and a client reading that same record from a replica. One of the most common operations after a write is a read of that same record. If a client does a write and then an immediate read from that same node, there's no problem. The problem occurs if a read occurs from a replica node before the update happens. This is an example of an inconsistent read. The best way to avoid this type of problem is to only allow reads to the same write node after a write has been done. This logic can be added to a session or state management system at the application layer. Almost all distributed databases relax database consistency rules when a large number of nodes permit writes. If your application needs fast read/write consistency, you must deal with it at the application layer.

4. **Letting the database distribute queries evenly to data nodes:**

In order to get high performance from queries that span multiple nodes, it's important to separate the concerns of query evaluation from query execution. Figure shows this structure:

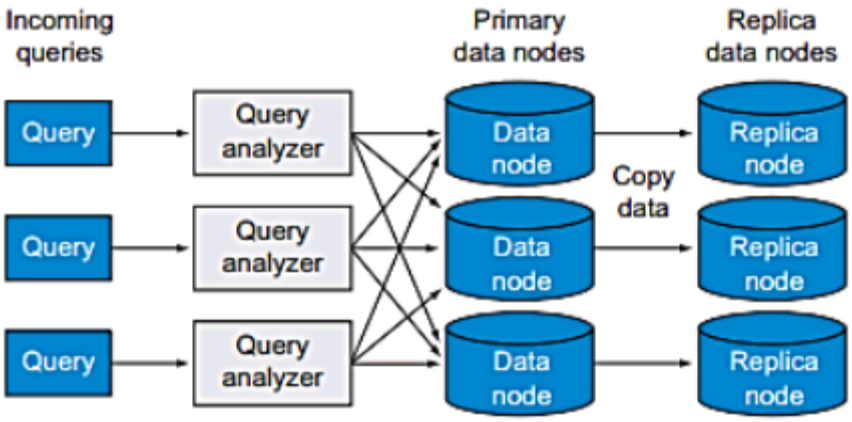

Figure NoSQL systems move the query to a data node, but don't move data to a query node. In this example, all incoming queries arrive at query analyzer nodes. These nodes then forward the queries to each data node. If they have matches, the documents are returned to the query node. The query won't return until all data nodes (or a response from a replica) have responded to the original query request. If the data node is down, a query can be redirected to a replica of the data node.

This approach is somewhat similar to the concept of federated search. Federated search takes a single query and distributes it to distinct servers and then combines the results together to give the user the impression they're searching a single system.

In this approach, the database separates query evaluation from query execution. Queries are sent to a query analyzer node, which then forwards them to each relevant data node. The results are returned to the query node only when all data nodes (or their replicas) have responded.

Real-Life Example: Think of a search engine like Google. When you search for something on Google, your query is sent to multiple servers that hold different parts of the index. Each server checks its own database and returns relevant results back to Google's central server. The final result set is then compiled and returned to you as the user. This process happens in real-time, without requiring all data to be transferred to a single point for processing.

In this example, each data node (the servers holding different parts of the index) receives the query from the query analyzer node (Google's central server), processes it locally, and returns relevant results back to Google. The final result set is then compiled and returned to the user. This approach allows Google to scale its search capabilities without having to worry about data transfer bottlenecks or single points of failure.