NoSQL Project Report

Introduction to NoSQL

NoSQL databases are a non relational database management systems. It does not mean that we are saying no to SQL. It means that apart fromm SQL, there could be other solutions too that are relevant in our applications that we use. Hence, th etern NoSQL can be interpreted as 'Not Only SQL'.

Now, let us see the architeture of NoSQL and how its been evolving over the years:

In the 1980's:

All the applications used the same database. For example, considering the database MySQL, all the applications that were being developed used MySQL but each used it individually.

In the 1990's:

Database was used as an integration hub where a group of applications all used the same database.

In the 2000's and hereafter:

Each application had its own database, and each database was different from the other.

NoSQL databases differ from the traditional NoSQL systems in many ways:

- NoSQL databases are unstructured unorganized and have unpredictable data, whereas SQL databases have structured and organized data.
- NoSQL can be classified further as key value/tuple store, column store, graph db, document store, whereas SQL databases can be classified as DDL, DML.
- NoSQL systems do not have any predefined schema, whereas SQL databases do.
- NoSQL systems relax one or more of the ACID properties, whereas SQL systems strictly follow the ACID properties

Some other **important characteristics** of NoSQL databases are:

- Next generation databases
- Open source
- Distributed
- Large data volumes
- Non relational
- Scalable replication and distribution.

CAP Theorem:

There are 3 main properties of a system:

- Consistency
- Availability
- Partitioning

Consistency means that the database remains same even after execution of the operations.

Availablility states that the system is always on, i.e., service is always guaranteed.

Partitioning requires that the system continues to function even if communicaton among servers is unreliable.

CAP Theorem states that for a shared data system, at most 2 of these 3 properties can be satisfied.
For example, regardingthe horizontal scaling aspect, partitioning property is necessary, and hence with consistency and availability to choose from, most of the times, availability is chosen over consistency.

Some reasons for the advent of NoSQL systems:
- Cloud based applications
- Applications that involve huge amounts of data, like all the social networking sites we use today.
- Open source community.

History of NoSQL
The name “NoSQL” was in fact first used by Carlo Strozzi in 1998 as the name of file-based database he was developing. It was a relational database just one without a SQL interface. As such it is not actually a part of the whole NoSQL movement we see today. The term re-surfaced in 2009 when Eric Evans used it to name the current surge in non-relational databases. It seems like the name has stuck.

Some other reasons for using NoSQL databases
- Large amounts of data involved
- No schema
- Horizontal scaling
- High availability
- Programming is easy
- Lower costs incurred
- Distributed computing
- No complicated relationships.

Classification of NoSQL databases

- Wide column store
  Examples include HBase, Cassandra, Cloudata
- Key value/ Tuple store
  Examples include Berkeley DB, Scalari, Redis
- Graph databases
  Examples include GraphBase, InfiniteGraph
- Document store
  Examples include Mongo DB, Raven DB

BERKELEY DB

Introduction
Berkeley DB is an open source, embedded database library. We can say its embedded as it directly links into our applicaiton. It runs in the same address space as the

application. As a result, no inter-process communication, either over the network or between processes on the same machine, is required for database operations. It comes under the category key value/tuple store of NoSQL databases.

Some featurs are:
- High performance
- Scalable
- Transaction oriented
- Highly reliable
- Portable

It is written in C with API bindings for
- C++
- Java
- C#
- Python
- Ruby

and so on.

The Berkeley DB library can be directly linked in to your application. It is extremely portable- Almost all Unix/Linux variants, Windows and also supported by 32 or 64 bit systems. All database operations happen inside the library. Multiple processes, or multiple threads in a single process, can all use the database at the same time as each uses the Berkeley DB library. Low-level services like locking, transaction logging, shared buffer management, memory management, and so on are all handled transparently by the library. It is scalable in many aspects- it can run either in very small sized embedded systems or very large servers and can make good use of gigabytes or terrabytes and so on. Berkeley DB generally outperforms relational and object-oriented database systems in embedded applications for a couple of reasons. First, because the library runs in the same address space, no inter-process communication is required for database operations. The cost of communicating between processes on a single machine, or among machines on a network, is much higher than the cost of making a function call. Second, because Berkeley DB uses a simple function-call interface for all operations, there is no query language to parse, and no execution plan to produce. Berkeley DB was designed to provide industrial-strength database services to application developers, without requiring them to become database experts. It is a classic C-library style *toolkit*, providing a broad base of functionality to application writers. Berkeley DB was designed by programmers, for programmers: its modular design surfaces simple, orthogonal interfaces to core services, and it provides mechanism (for example, good thread support) without imposing policy (for example, the use of threads is not required). Just as importantly, Berkeley DB allows developers to balance performance against the need for crash recovery and concurrent use. An application can use the storage structure that provides the fastest access to its data and can request only the degree of logging and locking that it needs.

What Berkeley DB is not?

- Berkeley DB is not a relational database. First, Berkeley DB does not support SQL queries. All access to data is through the Berkeley DB API. Developers must learn a new set of interfaces in order to work with Berkeley DB. Although the interfaces are fairly simple, they are non-standard.
- Object-oriented databases are designed for very tight integration with object-oriented programming languages. Berkeley DB is written entirely in the C programming language. It includes language bindings for C++, Java, and other languages, but the library has no information about the objects created in any object-oriented application. Berkeley DB never makes method calls on any application object. It has no idea what methods are defined on user objects, and cannot see the public or private members of any instance. The key and value part of all records are opaque to Berkeley DB.

Architecture of Berkely DB

The Berkeley DB product family has 3 main products:

- Berkeley DB

- Berkeley DB Java edition

- Berkeley DB XMl edition

The Berkeley DB products are available as libraries with simple API's for data access and database administration. It does not support SQL though it has been used as the storage enginefor SQL database products.

A typical Berkeley DB application makes API calls to start and end transactios, store data, retrieve data, perform administrative functions like restore and providing backup. Therefore, a Berkeley DB application can be called as "self conained" w.r.t data management activities.

Berkeley DB Java Edition is a pure java implementation. Berkeley DB was initialy developed in C. Both have similar API's similar features and functionality, but their structure dffers from one another. Portability is not an issue for the Berjkeley DB Java Edition as the JVM is portable. It also supports Btree indices whereas Berkeley DB java supports both the Btrees and hash indexing. It has APi's for administrative functions like backup and recovery and so on. For the embedded storage feature like in C, java developers use the Berkeley DB java edition as it is entirely in java. It is a single jar file that is embedded into the JVM and provdes the same transactional storge and retieval systems as Berkeley DB C product but just that the implementation is entirey in java.

Berkeley DB XML edition is useful when there is the need to manage XML documents. XML documents are stored either as whole documents or a individual nodes. It is an XML database that supportsXQuery and is designed to store XML documents for fast and scalable access. It provides fast reliable and scalable persistence for applications that need to manage XML content. Developing building XML documents can use this. It

understands XML schemas, it can parse and index XML documents and for data retrieval, it uses XPath an XQuery. It spports C++, Java.

The various Data Access Services Berkey DB provides:

It can choose the storage mechanisms that bestsuits its applications.

- Hash tables
- Btrees
- Persistent Queues
- Record number based storage

The various Data Access Management Berkey DB provides:

- Concurrency
- Transactions
- Recovery

Many users can work on the same database concurrently. It handles locking transparency, ensurig that two users working on the same record do not interfere with one another. Multiple operations can be grouped in to a single transaction and can be committed or rolled back automatically. When an application is started, it can ask the Berkeley DB to run recovery. This restores the database to a clean state and committed changes are present even after a crash.

ABOUT THE APPLICATION

Proposal: BerkelyDb for Computationally Intensive Algorithms.

Sometimes, we have to deal with an algorithm that repeatedly execute a computationally intensive operation. One example may be an application that works with factorials. In this situation, it would be useful to create a cache containing the already computed results. There are two good reasons for doing so:
1. we could avoid to re-compute results for the same input (even over different executions)
2. by adopting a fault-tolerant system for the cache implementation, in case of process crash, we would be able to start again the process and quickly go back to the point where it stopped.

To implement such a cache we have different possibilities. The simplest one would be to create an in memory map, which link the input of our computationally intensive

operation, to the result. This kind of cache is very efficient, considering that it is completely in memory. The downside are that we need a considerable amount of memory, and that there is no fault tolerance; in fact, we have to manually save the data to a file. Despite of this, if system failures are unlikely and the cache is going to be small, which happens if we have few different inputs, it may be a good solution.

DBMS might be a solution but today, the volumes of “big data” that can be handled by NoSQL systems, outstrip what can be handled by the biggest RDBMS. Normal HashMaps are inefficient in handling large volumes of data. Increasing memory footprint and use of plain java hashmaps will not always work effiiently and leads to out-of-memory errors for large volumes of data ,for example, data generated in computationally intensive applications.

In addition to implementation to demonstrate the use of BerkeleyDB in computationally intesive applications, we plan to provide experimental evidence of its performance compared to its alternatives (pain hashmaps, relational dbs) in terms of cpu utilization, memory utilization, garbage collection kickoff time and frequency, and effect of garbage collection in program completion time for following scenarios:

(i) original factorial java program (memory restricted setting) for larger input values with no additional cache support ,
(ii)factorial with berkelydb support (with same memory limitations as in (i))
(iii) factorial with relational db support ((with same memory limitations as in (i))