Supply cloud-level data scalability with NoSQL databases

Explore cloud and NoSQL database design basics

Sherif Sakr
Senior Research Scientist
National ICT Australia

25 March 2013

Relatively low-cost, high-performance NoSQL (Not Only SQL) databases contain some features that are useful when it comes to data scalability: Horizontal scalability, support for weaker consistency models, more flexible schemas and data models, and support for simple low-level query interfaces. This article explores NoSQL databases, including an overview of the capabilities and features of NoSQL systems HBase, MongoDB, SimpleDB. It also covers cloud and NoSQL database design basics.

The rapidly expanding generation of Internet-based services — such as email, blogging, social networking, search, and e-commerce — have substantially redefined the behavior and trends of web users when it comes to creating, communicating, accessing content, sharing information, and purchasing products. IT professionals are witnessing a proliferation in the scale of the data generated and consumed because of the growth in the number of these systems; this ever increasing need for scalability and new application requirements has created new challenges for traditional relational database management systems (RDBMS).

Enter the low-cost, high-performance NoSQL database software. The main features of NoSQL database software include:

- The ability to horizontally scale data.
- Support for weaker consistency models (one of the Atomicity, Consistency, Isolation, Durability (ACID) properties that guarantee that database transactions are processed reliably).
- The ability to use flexible schemas and data models.
- Support for simple low-level query interfaces.

This article explores the recent advancements in database systems for supporting web-scale data management. It presents an overview of the capabilities and features of main representatives for
different alternatives of NoSQL systems — HBase, MongoDB, and SimpleDB — and the suitability of each to support different types of web applications.

Cloud database design basics

So how has cloud computing changed the way people interact with data?

Easy data transactions for all

Recent advances in web technology has made it easy for any user to provide and consume content of any form. For example:

- Building a personal web page (such as Google Sites).
- Starting a blog (with WordPress, Blogger, LiveJournal).
- Interacting in online communities (with Facebook, Twitter, LinkedIn, etc.).

These avenues have become commodities, tools which have made it easy for many more people to create, consume, and transfer (also known as transactions) more data in more diverse forms (such as blog posts, tweets, social network interactions, video, audio, and photos ... data that is both structured and unstructured).

Apps become distributed, scalable services

The apparent next goal of the system and tool manufacturers is to facilitate the job of implementing every application as a distributed, scalable, and widely-accessible service on the web (for examples, look at such services from Facebook, Flickr, YouTube, Zoho, and LinkedIn).

The applications that meet this criteria are both data-intensive and very interactive. For example, at the time this article was written, Facebook announced it has 800 million monthly active users (it may be more like a billion now). Each user has an average friendship relation count of 130. Moreover, there are about 900 million objects that registered users interact with such as pages, groups, events, and community pages.

Other smaller scale social networks such as LinkedIn, which is mainly used for professionals, has recently reached 200 million registered users. Twitter claims to have more than 100 million active monthly users. It is a given fact that the ultimate goal is to make it easy for anyone who wants to achieve this high level of scalability and availability; how to do that with minimum efforts and resources is the challenge.

Cloud models facilitate services deployment

Cloud computing technology is a relatively new model for hosting software applications (although cloud is so integrated that by now it is possibly indistinguishable from the rest of the entire data-transaction system of the Internet. The cloud model simplifies the time-consuming processes of hardware provisioning, hardware purchasing, and software deployment, therefore it revolutionizes the way computational resources and services are commercialized and delivered to customers. In particular, it shifts the location of this infrastructure to the network in order to reduce the costs associated with the management of hardware and software resources.
This means that the cloud represents the long-held dream of envisioning computing as a utility, a dream in which the economy of scale principles help to effectively drive down the cost of the computing infrastructure.

Cloud computing promises a number of advantages for the deployment of software applications such as pay-per-use cost model, short time to market, and the perception of (virtually) unlimited resources and infinite scalability.

**New distro model means more new data and datatypes**

In practice, the advantages of the cloud computing model opens up new avenues for deploying novel applications that were not economically feasible in a traditional enterprise infrastructure setting. Therefore, the cloud has become an increasingly popular platform for hosting software applications in a variety of domains such as e-retail, finance, news, and social networking. The proliferation in the number of applications also delivers a tremendous increase in the scale of the data generated and consumed by these applications. This is why a cloud-hosted database system powering these applications forms a critical component in the software stack of these applications.

**Cloud models lead to cloud database model**

To meet the challenges posed by hosting databases on cloud computing environments there are a plethora of systems and approaches. In practice, there are three main technologies that are commonly used for deploying the database tier of software applications in cloud platforms:

- Virtualized database servers
- Database as a service platforms
- NoSQL storage systems

**The role of virtualization**

Virtualization is a key component of cloud computing technology; it allows you to abstract away the details of physical hardware and provides virtualized resources for high-level applications.

A virtualized server is commonly called a virtual machine (VM). VMs allow the isolation of both applications from the underlying hardware and other VMs. Ideally each VM is both unaware and unaffected by other VMs that could be operating on the same physical machine.

In principle, resource virtualization technologies add a flexible and programmable layer of software between applications and the resources used by these applications. The concept that defines a virtualized database server makes use of these advantages, especially where an existing database tier of a software application that has been designed to be used in a conventional data center can be directly ported to virtual machines in the public cloud.

Such a migration process usually requires minimal changes in the architecture or the code of the deployed application. In the virtualized database approach, database servers, like any other software components, are migrated to run in virtual machines. While the provisioning of a VM for each database replica imposes a performance overhead, this overhead is estimated to be of
less than 10 percent. In practice, one of the major advantages of the virtualized database server approach is that the application can have full control in dynamically allocating and configuring the physical resources of the database tier (database servers) as needed.

Therefore, software applications can fully utilize the elasticity feature of the cloud environment to achieve their defined and customized scalability or cost reduction goals; however, achieving these goals requires the existence of an admission control component that is responsible for monitoring the system state and taking the corresponding actions (for example, allocating more/less computing resources) according to the defined application requirements and strategies. One of the main responsibilities of this admission control component is to decide when to trigger an increase or decrease in the number of the virtualized database servers which are allocated to the software application.

The role of multi-tenancy in data centers

Data centers are often underutilized due to over-provisioning, as well as the time-varying resource demands of typical enterprise applications. Multi-tenancy is an optimization mechanism for hosted services in which multiple customers are consolidated onto the same operational system (a single instance of the software runs on a server, serving multiple clients) and thus the economy of scale principles help to effectively drive down the cost of computing infrastructure.

In particular, multi-tenancy allows pooling of resources which improves utilization by eliminating the need to provision each tenant for their maximum load. This makes multi-tenancy an attractive mechanism for both:

- Cloud providers (who are able to serve more customers with a smaller set of machines) and
- Customers of cloud services (who do not need to pay the price of renting the full capacity of a server).

Database as a service is a concept in which a third-party service provider hosts a relational database as a service. Such services alleviate the need for users to purchase expensive hardware and software, deal with software upgrades, and hire professionals for administrative and maintenance tasks.

In practice, the migration of the database tier of any software application to a relational database service is expected to require minimal effort if the underlying RDBMS of the existing software application is compatible with the offered service. However, some limitations or restrictions might be introduced by the service provider for different reasons (for example, the maximum size of the hosted database, the maximum number of possible concurrent connections). Moreover, software applications do not have sufficient flexibility in being able to control the allocated resources of their applications (like dynamically allocating more resources for dealing with increasing workload or dynamically reducing the allocated resources in order to reduce the operational cost). The whole resource management and allocation process is controlled at the provider side which requires an accurate planning for the allocated computing resources for the database tier and limits the ability of the consumer applications to maximize their benefits by leveraging the elasticity and scalability features of the cloud environment.
Why relational databases may not be optimal

In general, relational database management systems have been considered as a "one-size-fits-all solution for data persistence and retrieval" for decades. They have matured after extensive research and development efforts and very successfully created a large market and solutions in different business domains.

The ever-increasing need for scalability and new application requirements have created new challenges for traditional RDBMS, including some dissatisfaction with this one-size-fits-all approach in some web-scale applications. The answer to this has been a new generation of low-cost, high-performance database software designed to challenge dominance of relational database management systems. A big reason for the NoSQL movement is that different implementations of web, enterprise, and cloud computing applications have different requirements of their databases — not every application requires rigid data consistency, for example.

Another example: For high-volume websites like eBay, Amazon, Twitter, or Facebook, scalability and high availability are essential requirements that cannot be compromised. For these applications, even the slightest outage can have significant financial consequences and impacts customer trust.

Next, let's explore the basic design principle of a NoSQL database.

Design principles of NoSQL databases

Looking at CAP from a different angle

According to one of the creators of the CAP theorem Eric Brewer, the 2 out of 3 view can be misleading because:

- Partitions are rare; you usually don't need to forfeit consistency or availability when the system is not partitioned.
- Choosing between consistency and availability can occur many times in the same system at very fine granularity; subsystems can make different choices and the choice can change according to the operation, data, or user involved.
- The three properties exist in a more continuous state than in a binary one. In reality, the necessary levels of each property is more important to the outcome than whether the system is exhibits 0 or 100 percent of the property.

In other words, the CAP theorem works best when you take into account the nuances of each property and focus on what level of each property is necessary to achieve your specific outcome based on existing parameters. Brewer proposes a three-step strategy to detect and deal with partitions when they are present:

1. Detect the partition.
2. Enter an explicit partition mode that limits some operations.
3. Initiate a recovery process designed to "restore consistency and compensate for mistakes made during a partition."


In practice, the CAP theorem — consisting of the three properties of Consistency, Availability, and tolerance to Partitions — has shown that a distributed database system can only choose to fulfill
at most two out of the three properties. Most of these systems decide to compromise the strict consistency requirement (for more on the evolution of the CAP theorem, see sidebar). In particular, they apply a relaxed consistency policy called eventual consistency which guarantees that if no new updates are made to a replicated object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme.

These new NoSQL systems have a number of design features in common:

- The ability to horizontally scale out throughput over many servers.
- A simple call level interface or protocol (in contrast to a SQL binding).
- Support for weaker consistency models than the ACID transactions in most traditional RDBMS.
- Efficient use of distributed indexes and RAM for data storage.
- The ability to dynamically define new attributes or data schema.

These design features of these systems are mainly targeting to achieve the following system goals:

- **Availability**: The system should be accessible even in the situation of having a network failure or an entire data center going offline.
- **Scalability**: The system should be able to support very large databases with very high request rates at very low latency.
- **Elasticity**: The system should be able to satisfy changing application requirements in both directions (scaling up or scaling down). Moreover, the system must be able to gracefully respond to these changing requirements and quickly recover its steady state.
- **Load balancing**: The system should be able to automatically move load between servers so that most of the hardware resources are effectively utilized and to avoid any resource overloading situations.
- **Fault tolerance**: The system should be able to deal with the reality that even the rarest hardware problems at some point will go from being freak events to eventualities. While hardware failure is still a serious concern, this concern needs to be addressed at the architectural level of the database rather than requiring developers, administrators, and operations staff to build their own redundant solutions.

In fact, Google’s BigTable and Amazon’s Dynamo have provided a proof of concept that inspired and triggered the development all of the NoSQL systems:

- BigTable has demonstrated that persistent record storage could be scaled to thousands of nodes.
- Dynamo has pioneered the idea of eventual consistency as a way to achieve higher availability and scalability.

The open source community has responded to that with many systems such as HBase, Cassandra, Voldemort, Riak, Redis, Hypertable, MongoDB, CouchDB, Neo4j, and countless other
projects. These projects mainly differ in their decisions regarding alternative design choices such as:

- **Data model**: Key/value, row store, graph-oriented, or document-oriented.
- **Access-path optimization**: Read-intensive versus write-intensive or single-key versus multi-key.
- **Data partitioning**: Row-oriented, column-oriented, multi-column oriented.
- **Concurrency management**: Strong consistency, eventual consistency, weak consistency.
- **CAP theorem**: CP, CA, or AP.

Next is an overview of the capabilities and features for the main representatives for different alternatives of NoSQL systems: SimpleDB, HBase, and MongoDB.

**SimpleDB**

### Did you know? Consistency models

The important models for this article are eventual and strong consistency.

The eventual consistency model states that given a sufficiently long period of time over which no changes are sent, all updates can be expected to propagate eventually through the system and the replicas will be consistent. (There is an slightly different, stronger definition: Eventually for a given accepted update and a given replica, either the update reaches the replica or the replica retires from service.)

Strong consistency means that all accesses are seen by all parallel processes (or nodes) in the same sequential order.

Weak consistency means all accesses to synchronization variables are seen by all processes sequentially; other accesses may be seen in different order on different processes.

Immediate consistency means that when an update operation returns to the user with a successful result, the result of that update is immediately visible to all observers. It defines only the state of the system with regard to a single update after the update has been reported as completed.

SimpleDB is a commercial Amazon Web Service (AWS) which is designed for providing structured data storage in the cloud and backed by clusters of Amazon-managed database servers. It is a highly available and flexible non-relational data store that offloads the work of database administration. In practice, SimpleDB represents a cloud-hosted storage service that provides different application interfaces (like REST, APIs) so it can be conveniently used and accessed by any software application which is hosted on IBM® SmartCloud® Enterprise.

Storing data in SimpleDB does not require any pre-defined schema information. Developers simply store and query data items via web services requests and SimpleDB does the rest. There is no rule that forces every data item (data record) to have the same fields; however, the lack of schema means also that there are no data types since all data values are treated as variable length character data.
The drawbacks of a schema-less data storage also include the lack of automatic integrity checking in the database (no foreign keys) and an increased burden on the application to handle formatting and type conversions.

SimpleDB has a pricing structure that includes charges for data storage, data transfer, and processor usage; there are no base fees nor minimums. Similar to most AWS services, SimpleDB provides a simple API interface that follows the rules and the principles for both of REST and SOAP protocols where the user sends a message with a request to carry out a specific operation. The SimpleDB server completes the operations unless there is an error and responds with a success code and response data. The response data is an HTTP response packet in XML format which has headers, storage metadata, and some payload.

The top level abstract element of data storage in SimpleDB is the domain. A domain is roughly analogous to a database table where you can create and delete the domains as needed. There are no design or configuration options to create a domain. The only parameter you can set is the domain name.

All the data stored in a SimpleDB domain takes the form of key-value attribute pairs. Each attribute pair is associated with an item which plays the role of a table row. The attribute name is similar to a database column name, but different items (rows) can contain different attribute names which give you the freedom to store different attributes in some items without changing the layout of other items that do not have the same attributes. This flexibility allows you to painlessly add new data fields in the common situation of schema changing or schema evolution.

In addition, it is possible for each attribute to have not just one value (multi-valued attributes) but an array of values. In this case, all you do is add another attribute to an item and use the same attribute name but with a different value. Each value is automatically indexed as you add it, but there are no explicit indexes to maintain so you have no index maintenance work of any kind to do. On the other side, you do not have any direct control over the created indices.

The following snippet of Java™ code is an example of creating SimpleDB domain:

```java
SimpleDB simpleDB = new SimpleDB(accessKeyID, secretAccessKey);
try {
    simpleDB.createDomain("employee");
    System.out.println("create domain succeeded");
} catch (SDBException e) {
    System.err.printf("create domain failed");
}
```

In this example, the variables accessKeyID and secretAccessKey represent the AWS credentials of the connecting user and employee represents the created domain name. Other domain operation supported by SimpleDB include:

- The DeleteDomain operation which permanently deletes all the data associated with the existing named domain.
- ListDomains which returns a listing of all the domains associated with the user account.
• The DomainMetadata operation which returns detailed information about one specific domain such as:
  • ItemCount that returns the count of all items in the domain,
  • AttributeNameCount that returns the count of all unique attribute names in the domain, and
  • AttributeValueCount that returns the count of all name/value pairs in the domain.

The following code snippet shows an example of storing data with the PutAttribute operation:

```
SimpleDB simpleDB = new SimpleDB(accessKeyID, secretAccessKey);
ItemAttribute[] employeeData = new empAttribute[] {
    new ItemAttribute("empID", "JohnS"),
    new ItemAttribute("first_name", "John"),
    new ItemAttribute("last_name", "Smith"),
    new ItemAttribute("city", "Sydney"),
};
try {
    Domain domain = simpleDB.getDomain("employee");
    Item newItem = domain.getItem("E12345");
    newItem.putAttributes(Arrays.asList(employeeData));
    System.out.println("put attributes succeeded");
} catch (SDBException e) {
    System.err.printf("put attributes failed");
}
```

Getting back the data you previously stored can be achieved using the following snippet of Java code:

```
SimpleDB simpleDB = new SimpleDB(accessKeyID, secretAccessKey);
try {
    Domain domain = simpleDB.getDomain("users");
    Item item = domain.getItem("E12345");
    String fname = item.get("first_name ");
    String lname = item.get("last_name ");
    String city = item.get("city ");
    System.out.printf("%s %s %s", fname, lname, city);
} catch (SDBException e) {
    System.err.printf("get attributes failed");
}
```

Other data manipulation operations which are supported by SimpleDB include:

• The BatchPutAttributes operation which enables storing the data for multiple items in a single call.
• The DeleteAttributes operation which removes the data for the items that has previously been stored in SimpleDB.

The SimpleDB Select API uses a query language that is similar to the SQL Select statement. This query language makes SimpleDB Select operations very familiar, with a gentle learning curve for its user. Watch out for the issue that the language supports issuing queries only over the scope of a single domain (with no joins, multiple domains, or sub-select queries). For example, this SimpleDB select statement retrieves the top-rated movies in the underlying domain:

```
SELECT * FROM movies WHERE rating > '04' ORDER BY rating LIMIT 10
```
In this example, movies represents the domain name and rating is a filtering attribute for the items of the domain. Since there is an index in SimpleDB for each attribute, an index will be available for optimizing the evaluation each query predicate. SimpleDB attempts to automatically choose the best index for each query as part of the query execution plan.

SimpleDB is implemented with complex replication and failover mechanisms behind the scenes, so when you use SimpleDB, you get a high availability guarantee with your data replicated to different locations automatically. A user does not need to do any extra effort or become an expert on high availability or the details of replication techniques to achieve the high availability goal.

SimpleDB supports two options for each user read request — eventual consistency or strong consistency. In general, using the option of a consistent read eliminates the consistency window for the request. The results of a consistent read are guaranteed to return the most up-to-date values. In most cases, a consistent read is no slower than an eventually consistent read, but it is possible for consistent read requests to show higher latency and lower bandwidth on some occasions (for example, with heavy workloads). SimpleDB does not offer any guarantees about the eventual consistency window but it is frequently less than one second.

There are quite a few limitations a user needs to consider while using the SimpleDB service:

- The maximum storage size per domain is 10GB.
- The maximum attribute values per domain is 1 billion.
- The maximum attribute values per item is 256.
- The maximum length of item name, attribute name, or value is 1024 bytes.
- For queries:
  - The maximum query execution time is 5 seconds.
  - The max query results are 2500.
  - The maximum query response size is 1MB.

**HBase**

Apache HBase, a Hadoop database, is a distributed storage system for managing structured data that is designed to scale to a very large size across horizontally scalable commodity machines. It is an Apache project which implements a clone of the BigTable storage architecture faithfully (other clones for the ideas of BigTable include the Apache Cassandra project and Hypertable project). In practice, HBase clusters can be installed and run on any IaaS cloud service providers such as IBM SmartCloud Enterprise; IBM InfoSphere® BigInsights™ builds atop Hadoop to provide the necessary enterprise-oriented capabilities.

Released in 2007, HBase is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp. Each value in the map is an uninterpreted array of bytes, so clients usually need to serialize various forms of structured and semi-structured data into these strings. A concrete example that reflects some of the main design decisions of HBase is found in a scenario in which you want to store a copy of a large collection of web pages into a single table.
Figure 1 illustrates an example of this table in which URLs are used as row keys and various aspects of web pages as column names. The contents of the web pages are stored in a single column which stores multiple versions of the page under the timestamps when they were fetched.

**Figure 1. Sample record in HBase table**

![Sample record in HBase table]

The row keys in a table are arbitrary strings where every read or write of data under a single row key is atomic. HBase maintains the data in lexicographic order by row key where the row range for a table is dynamically partitioned. Having the row keys always sorted can give you something like the primary key index found in relational databases.

While the original BigTable implementation only considers a single index, HBase adds support for secondary indices. Each row range is in each table called a *tablet* that represents the unit of distribution and load balancing. Reads of short row ranges are efficient and typically require communication with only a small number of machines.

**Did you know? Sparse data sets**

Since I've just mentioned sparse data sets again in this article, I thought I'd define what I mean. A *sparse data matrix* is one that has a low density of significant data or connections. Like I say over in the text, the majority of the values are null. (The other side of the spectrum, in a dense data matrix, the majority of values will be not-null. The implications for storage are this:

- sparse matrices are easily compressed, resulting in much less storage space usage.
- Algorithms used to manipulate dense matrices are by their nature slow with quite a bit of computational overhead; that costly effort is wasted on sparse matrices.

HBase can have an unbounded number of columns that are grouped into sets called *column families*. These column families represent the basic unit of access control. The data is stored by column and it doesn't need to store values when they are null, so HBase is suitable for sparse data sets. Every column value (cell) either is timestamped implicitly by the system or can be set explicitly by the user. This means that each cell in a BigTable can contain multiple versions of the same data which are indexed by their timestamps. The user can flexibly specify how many versions of a value should be kept. These versions are stored in decreasing timestamp order so that the most recent versions can be always read first. Putting this together, you'll find you can express the access to data as follows:

\[(\text{Table, RowKey, Family, Column, Timestamp}) \rightarrow \text{Value}\]

The fact that reads of short row ranges require low communication can affect the development of queries, so that they are suitable to the available network resources. On the physical level, HBase uses the Hadoop distributed file system in place of the Google file system. It puts updates into memory and periodically writes them out to files on the disk.
The basic unit of scalability and load balancing in HBase is called a **region**. Regions are essentially contiguous ranges of rows stored together. They are dynamically split by the system when they become too large; alternatively they may also be merged to reduce their number and required storage files.

Initially there is only one region for a table; as you start adding data to it, the system is monitoring it to ensure that you do not exceed a configured maximum size. If you exceed the limit, the region is split into two at the middle key of the region creating two roughly equal halves. Each region is served by exactly one region server and each of these servers can serve many regions at any time.

Splitting and serving regions can be thought of as auto-**sharding**, a technique offered by other systems. The region servers can be added or removed while the system is up and running. The master is responsible for assigning regions to region servers and uses Apache ZooKeeper, a reliable, highly available, persistent, and distributed coordination service, to facilitate that task. In addition, it handles schema changes such as table and column family creations.

In HBase, all operations that mutate data are guaranteed to be atomic on a per-row basis. It uses optimistic concurrency control which aborts any operations if there is a conflict with other updates. This affects all other concurrent readers and writers of that same row as they either read a consistent last mutation or may have to wait before being able to apply their change. For data storage and access, HBase provides a Java API, a Thrift API, REST API, and JDBC/ODBC connection.

The following code snippet shows an example of storing data in HBase:

```java
Configuration conf = HBaseConfiguration.create();
HTable table = new HTable(conf, "employee");
Put put = new Put(Bytes.toBytes("E12345"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("first_name"), Bytes.toBytes("John"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("last_name"), Bytes.toBytes("Smith"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("city"), Bytes.toBytes("Sydney"));
table.put(put);
```

This example creates a row in the **employee** table, adds a column family with three columns (**first_name**, **last_name**, and **city**), stores the data values in the created columns, and adds the row into the HBase table. The following code snippet shows an example of retrieving data from HBase:

```java
Configuration conf = HBaseConfiguration.create();
HTable table = new HTable(conf, "employee");
Get get = new Get(Bytes.toBytes("E12345"));
get.addColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("first_name"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes("first_name"));
System.out.println("Employee's first name is: " + Bytes.toString(val));
```
In addition, HBase provide a rich set of API functions such as the `Delete` function that can remove the attribute value of any key, the `DeleteColumn` function which deletes an entire column, and the `DeleteFamily` function which deletes an entire column family, including all contained columns. In addition there are a set of batch `Put`, `Get`, and `Delete` functions and the resulting scanning operations.

**MongoDB**

MongoDB is a distributed, schema-free, document-oriented database which was created at 10gen. It stores binary JSON documents (BSON) that are binary-encoded JSON like objects. BSON supports nested object structures with embedded objects and arrays. Similar to HBase, MongoDB and CouchDB clusters can be installed and run on any IaaS provider platform such as IBM SmartCloud Enterprise.

At the heart of MongoDB is the concept of a *document* which is represented as an ordered set of keys with associated values; a *collection* is a group of documents. If a document is the MongoDB analog of a row in a relational database, then a collection can be thought of as the analog to a table.

Collections are schema-free. This means that the documents within a single collection can have any number of different "shapes." For example, both of the following documents could be stored in a single collection:

```json
{"postID" : "P1", "postText" : "This is my blog post"}
```

Note that the previous documents not only have different structures and types for their values, they also have entirely different keys.

MongoDB groups collections into databases. A single instance of MongoDB can host several databases, each of which can be thought of as virtually independent. The `insert` function adds a document to a collection. The following code snippet shows an example of inserting a post document into a blog collection:

```javascript
post = {"title" : "MongoDB Example", "content" : "Insertion Example ", "author" : :
:"John Smith"}
db.blog.insert(post)
```

After saving the blog post to the database, you can retrieve it by calling the `find` method on the collection: `db.blog.find("title" : "MongoDB Example")`.

The `find` method returns all of the documents in a collection satisfying the filtering conditions. If you just want to see one document from a collection, you can use `findOne` method: `db.blog.findOne()`.

If you want to modify the post, you can use the `update` method, which takes (at least) two parameters:
The following code snippet shows an example of updating the content of a blog post:

```javascript
post = {"content" : "Update Example"
blog.update({{title : " MongoDB Example "}}, post }
```

**Did you know? Sharding**

A database shard is a horizontal partition in a database or search engine. Each individual partition is referred to as a shard. The rows of a database table are held separately, rather than being split into columns (normalization), and each partition (shard) can be located on a separate database service or even in a different physical location.

The advantages to sharding are:

- A reduction in the number of rows in each table in each database.
- The reduced index size usually improves search performance.
- The database can be distributed over a large number of machines (which can also increase performance).
- Although not as common, the shard segmentation may correlate with real-world data access patterns. (For example, European customer data accessed by European sales, Asian customer data by Asian sales, etc.; each would only need to query the relevant shard making it more efficient.)

In practice, sharding is fairly complex. Author, developer Andy Glover provides a strong overview of sharding in this developerWorks article.

Note that MongoDB provides only eventual consistency guarantees, so a process could read an old version of a document even if another process has already performed an update operation on it. In addition, it provides no transaction management so that if a process reads a document and writes a modified version back to the database, it may happen that another process writes a new version of the same document between the read and write operation of the first process.

The **remove** method deletes documents permanently from the database. If it is called with no parameters, then it removes all documents from a collection. If parameters are specified, then it removes only the documents specifying criteria for removal. The following code snippet shows an example of removing the sample blog post: `db.blog.remove({title : " MongoDB Example "}).

You can create indices using the **ensureIndex** method. The following code snippet shows an example of indexing the blog posts using their title information: `db.people.ensureIndex({"title " : 1}).

MongoDB also supports indexing the documents on multiple files. The API interface is rich and supports different batch operations and aggregation functions. MongoDB is Implemented in C++; it provides drivers for a number of programming languages including C, C#, C++, Erlang. Haskell, Java, JavaScript, Perl, PHP, Python, Ruby, and Scala. It provides a JavaScript command-line interface.

The project of Apache CouchDB is another example of document-oriented databases which is similar to MongoDB in its data model and data-management mechanism. It is written in Erlang,
but is currently not a distributed database, though it can be used for applications which are written in JavaScript. The documents are stored and accessed as JSON objects. CouchDB has no support for document types or equivalent to tables, so developers have to build the document type distinction by themselves which they can do by putting a `type` attribute in each document.

An interesting feature in CouchDB is that it supports defining a validation functions so that if the document is updated or created, the validation functions are triggered to approve or validate the operations. CouchDB can be also tuned to provide either strong consistency or eventual consistency guarantees.

**DB2 support for NoSQL**

In a quest for efficient data management in a cloud environment, using the most appropriate data model for each data management task is the end goal, so I thought it might be worth a mention at this point that even traditional relational databases are supporting various NoSQL versions to reach that goal (I'll talk more about this in the conclusion).

For example, IBM DB2 supports a NoSQL graph store through an API that supports multiple calls and a NoSQL software solution stack. The graph store stores data in only three columns (called a triple) that represents data as a noun, a verb, and a noun. A simple query can pull any data triples together that have matching nouns. DB2 provides performance-enhancing features for the graph store such as the ability to map graph triples to relational tables (with unlimited columns), data compression, parallel execution, and load balancing.

Through DB2 pureXML, DB2 also offers a type of NoSQL-like database, the XML data store, which makes it easier to manage web-based data in its native hierarchical format. DB2 users can use query languages such XPath and XQuery to process XML documents. This data model comes in handy in use cases where there is ongoing change in the information being managed.

**In conclusion**

For more than a quarter of a century, relational database management systems (RDBMS) have been the dominant model for database management. They provide an extremely attractive interface for managing and accessing data and have proven to be wildly successful in many financial, business, and Internet applications. Along with the trend of web-scale data management, they have started to suffer from some limitations.

As an alternative model for database management, NoSQL systems' main advantages are:

- **Elastic scaling:** For years, database administrators have relied on the scale up approach rather than the scale out approach; with the current increase in the transaction rates and high availability requirements, the economic advantages of scaling out (especially on commodity hardware) become very attractive. NoSQL systems are designed with the ability to expand transparently in order to take advantage of the addition of any new nodes.
- **Less administration:** NoSQL databases are generally designed to support features like automatic repair and a simpler data model; this should lead to lower administration and tuning requirements.
• **Better economics:** NoSQL databases typically use clusters of inexpensive commodity servers to manage the exploding data and transaction volumes.

• **Flexible data models:** NoSQL databases have more relaxed (if any) data model restrictions, so application and database schema alterations can be changed more softly.

There are still many obstacles NoSQL databases need to overcome before these systems can appeal to mainstream enterprises:

• **Programming models:** NoSQL databases offer few facilities for ad-hoc query and analysis. Even a simple query requires significant programming expertise. Missing the support of declaratively expressing the important join operation has been always considered one of the main limitations of these systems.

• **Transaction support:** Transaction management is one of the powerful features of RDBMS. The current limited-to-non-existent support of the transaction notion on NoSQL database systems is considered an obstacle to them being accepted to implement mission-critical systems.

• **Maturity:** RDBMS systems are well-known for their high stability and rich functionalities. In comparison, most NoSQL alternatives are in pre-production versions with many key features either not stable enough or yet to be implemented. This means that enterprises are still approaching this new wave of data management with extreme caution.

• **Support:** Enterprises look for the assurance that if the system fails, they will be able to get timely and competent support. RDBMS vendors go to great lengths to provide that high level of support. In contrast, many NoSQL systems are open source projects and the support levels are not there yet.

• **Expertise:** Almost every NoSQL developer is in learning mode; granted, this is a situation that will change over time. But currently, it is far easier to find experienced RDBMS programmers or administrators than a NoSQL expert.

The debate between the NoSQL and RDBMS factions will probably never generate a definitive answer. Most likely, the best possible answer will be that different data management solutions will coexist in a single application. For example, I can imagine an application which uses different data stores for different purposes:

• An SQL RDBMS for low-volume, high-value data such as user profiles and billing information.
• A key/value store for high-volume, low-value data such as hit counts and logs.
• A file storage service for user-uploaded assets such as photos, sound files, and big binary files.
• A document database for storing the application documents such as bills.
Resources

Learn

- There are lots of resources available, in a simple web search, for resources pertaining to the topics in this article. Here are a select few to get you started:
  - Using NoSQL and analyzing big data
  - Discover MongoDB
  - "An introduction to the Hadoop Distributed File System"
  - More on the Hadoop Distributed File System
  - Understanding HBase
  - List of NoSQL databases
  - Werner Vogels on Eventually Consistent (from his blog All Things Distributed)
  - DB2 pureXML for XML data management
- Learn more about cloud computing technologies at cloud at developerWorks.
- For technology sites pertaining to the topics in this article:
  - InfoSphere BigInsights
  - Amazon SimpleDB
  - Amazon Simple Storage Service (Amazon S3)
  - Apache HBase
  - Apache Hadoop
  - Apache ZooKeeper
  - Apache Cassandra
  - Hypertable
  - MongoDB
  - Apache CouchDB
  - Neo4j
  - Apache Thrift
  - Amazon Relational Database Service (Amazon RDS)
  - Windows Azure
  - Google Cloud SQL
  - Heroku Postgres
  - Relational Cloud Project
- Follow developerWorks on Twitter.
- Watch developerWorks demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.

Get products and technologies

- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.
About the author

Sherif Sakr

Dr. Sherif Sakr is a senior research scientist in the Software Systems Group at National ICT Australia (NICTA) in Sydney. He is also a conjoint lecturer in The School of Computer Science and Engineering (CSE) at University of New South Wales (UNSW). He received his PhD degree in Computer Science from Konstanz University in Germany in 2007. He received his BSc and MSc degrees in Computer Science from the Information Systems department at the Faculty of Computers and Information in Cairo University in Egypt in 2000 and 2003 respectively. In 2011, Sakr held a visiting research scientist position in the eXtreme Computing Group (XCG) at Microsoft Research. In 2012, he held a research MTS position in Alcatel-Lucent Bell Labs. Sherif is a Cloudera certified developer for Apache Hadoop and Cloudera certified specialist for HBase.

© Copyright IBM Corporation 2013
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)