MOVING TOWARDS NON-RELATIONAL DATABASES

*Uma Bhat, #Shraddha Jadhav,
Usha Mittal Institute of Technology, SNDT Women’s University, Santacruz (W), Mumbai, 400049
*uma23bhat@gmail.com, #22.shraddha@gmail.com

This work was carried out under the edges of Patni Computer Systems Ltd. as part of the Summer Training Project of BTech Course.

ABSTRACT:
RDBMS has been around since long. It is the founderstone of many application stacks. It provides the users with the best mix of simplicity, robustness, flexibility, performance, scalability, and compatibility.

However the emergence of the cloud centric application poses a set of challenges to the existing RDBMS Vendors. The RDBMS are not so suitable to cater to some of the critical requirement of these new generation applications such as handling large set of unstructured data or providing elastic scalability.

1. INTRODUCTION
Database model is a theory or specification describing how a database is structured and used. Several such models have been suggested such as hierarchical, network, relational and non-relational. Now-a-days, relational database models are the dominant persistent storage technology. Inspite of this fact, it has many shortcomings which can hinder performance levels. As more and more applications are launched in environments that have massive workloads such as cloud and web services, their scalability requirements change very quickly and also grow very large. It is difficult to manage with a relational database sitting on a single in-house server. To overcome all these shortcomings, vendors can opt for non relational database models.
Non-Relational databases enjoy schema-free architecture and possess the power to manage highly unstructured data. They can be easily deployed to multi-core or multi-server clusters serving modularization, scalability and incremental replication. Non relational databases being extremely scalable, offer high availability and reliability, even while running on hardware that is typically prone to failure, thereby challenging relational database, where consistency, data integrity, uptime and performance are of prime importance.

In this paper with the help of a case study, we have attempted to demonstrate how non-relational databases such as, Apache CouchDB, “Cluster Of Unreliable Commodity Hardware”, Google's Big Table and Amazon SimpleDB would prove to be tried, tested and trusted solutions overcoming drawbacks of relational databases.
2. CHALLENGES WITH RELATIONAL DATABASES

1. **Performance issues are difficult to predict**

While working with a shared database, the performance characteristics of database is hard to predict because each application accesses the database in its own unique way.

2. **Data integrity is difficult to ensure with shared databases**

Because no single application has control over the data it is very difficult to be sure that all applications are operating under the same business principles.

3. **Operational databases require different design strategies than reporting databases**

The schemas of operational databases reflect the operational needs of the applications that access them, often resulting in a reasonably normalized schema with some portions of it denormalized for performance reasons. Reporting databases are highly denormalized with significant data redundancy within them to support a wide range of reporting needs.

3. PROMISING NON RELATIONAL DATABASES

Web 2.0 approaches and design patterns are becoming more established in online applications and enterprise architecture. Social architectures, crowdsourcing, and open supply chains are becoming the norm in the latest software systems faster than expected in many cases. New advances in processors, virtualization technology, disk storage, broadband Internet access and fast, inexpensive servers have all combined to make cloud computing a compelling paradigm. Unfortunately, as a result, the architectural expertise needed to effectively leverage these ideas is often far from abundant. Non relational databases like Apache CouchDB, Amazon’s SimpleDB and Google’s BigTable prove to be effective methods in overcoming these shortcomings.

3.1 Apache CouchDB: Overview

CouchDB is an open source document-oriented database-management system, accessible using a RESTful JavaScript Object Notation (JSON) API. The term "Couch" is an acronym for "Cluster Of Unreliable Commodity Hardware," reflecting the goal of CouchDB being extremely scalable, offering high availability and reliability, even while running on hardware that is typically prone to failure. CouchDB was originally written in C++ but moved to the Erlang OTP platform for its emphasis on fault tolerance. It is a database built for the future. CouchDB has been developed from the ground up with Web applications as the primary focus and has its sights on becoming the de-facto database for Web application development.
**Features of COUCHDB**

1. **Schema Free**

CouchDB has a hand full of advantages over relational database model and also aims at storing data in documents, which have no schema. One document can have a field that another one does not have. The documents in CouchDB are actual representations of the data objects. CouchDB stores data in the JSON format. It is language-agnostic. Data is serialized and de-serialized to and from that format. JSON can include all the native data types in a programming language.

2. **Document Oriented Structure**

Document oriented structure forms building blocks of CouchDB database. CouchDB databases store uniquely named documents with document ID and revision number. All data in a CouchDB database is stored in a document, and each document can be made up of an undefined number of fields which are not bound by size and have unique names. Documents can have attachments which can be of both text as well as digital format.

When changes are made to a CouchDB document, a new version of the entire document called a revision is created. The document-revision system works in much the same way as a wiki or Web-based document-management system manages revision control. CouchDB does not feature locking mechanisms; two clients can load and edit the same document at the same time. CouchDB maintains data consistency by ensuring that document updates are all or nothing — it either works or it fails.

3. **Concurrent**

CouchDB is written in Erlang. Erlang is a functional programming language, a virtual machine, and a set of standard libraries. CouchDB uses Erlang because the problems that it solved for Telecom applications are the same for the Web today. CouchDB has an inbuilt ACID compliant datastore which is referred to as Non-locking Multi Version Concurrency Control. Concurrent requests can continue to be served by using versioning and it never overwrites data. Compaction can be used to get rid of previous revisions and to reclaim disk space thus making it crash-resistant.

4. **RESTful HTTP API**

CouchDB treats all stored items in the database as resources. It offers an API as a means to retrieve data from the database. This API is accessible via HTTP GET and POST requests, and returns data in the form of JavaScript objects using JSON. HTTP is understood, interoperable, scalable and proven technology and can be used with software and hardware for caching, proxying and load balancing.

Four basic operations on document using CouchDB via HTTP REST API:

**Create:** HTTP PUT /db/docid  
**Read:** HTTP GET /db/docid  
**Update:** HTTP POST /db/docid  
**Delete:** HTTP DELETE /db/docid
5. Map/Reduce

The Map and Reduce functions of Map/Reduce are both defined with respect to data structured in (key, value) pairs. Map() is a user defined function which transform each document into zero, one or multiple intermediate objects, reduce() is user defined function to consolidate the intermediate objects into the final result.

6. Replication

CouchDB achieves eventual consistency between databases by using incremental replication, a process where document changes are periodically copied between servers. Share nothing clusters of databases can be built where each node is independent and self-sufficient, leaving no single point of contention across the system. Replication allows synchronizing two or more CouchDB instances and includes automatic conflict detection and resolution. When trying to replicate, CouchDB detects the conflict and each node independently chooses one conflicting revision to be the winning revision. All losing revisions get stored as previous revisions.

Data Model for COUCHDB

```json
{
   "_id": "44e55a9a3ebe5204a747343b91430434",
   "_rev": "11-4290911434",
   "clinical history": {"Kidney stone": "yes", "TB": "no"},
   "name": "Mr. Kiran",
   "symptoms": {"1": "Vomiting", "2": "Fever", "3": "Stomach_ache"},
   "personal_data": {"Age": 36, "Height": 179, "Weight": 76},
   "test": {"a": "Culture_test", "b": "Blood_test", "c": "HC/CBC"}
}
```
Advantages

1. Scalability

CouchDB embraces RESTful architectural constraints to promote resolvability, efficiency, performance, and reliability thus enhancing scalability.

2. Schema free:

There is no need to predefine a schema before creating documents. All documents can be independent which reduces their inter dependency and there is no need to update database when a field is added.

3. HTTP request-response mechanism:

It makes use of simple HTTP create, request, update and delete operations on documents. It is accessible just by giving HTTP requests through the browser.

4. Support for attachments:

It stores file attachments in the form of images, music, and flash. Thus, it supports digital attachments which are not seen in traditional relational databases.

5. Zero configuration Replication:

Now, there is no need to carry storage devices. Data can be replicated in the network across nodes both in local and remote fashion with absolutely no need of internet connection.

6. Provision of Web administration interface-Futon:

Futon JavaScript interface can be used for displaying and editing data, deleting, inserting, creating, updating documents and triggering replication.

7. Low Memory requirement:

Takes 150MB compared to 8Gig taken by MySQL for a similar database setup.

Comparison between RDBMS and COUCHDB

1. Unstructured with no table format

Relational databases are structured and have a predefined schema. But CouchDB is a document oriented database where data is stored in the document itself, not in a related table as it would be in a relational database. There are no tables, rows, columns or relationships in a document-oriented database at all.

2. Schema Free:

No strict schema needs to be defined in advance before using the database. If a document needs to add a new field, it can simply include that field, without adversely affecting other documents in the database. These documents do not have to store empty data values for fields they do not have a value for.

3. Concept of identifiers

Relational databases use concept of primary keys, generated by an auto-increment feature or by a sequence generator. These identifiers are unique for the table or database they are used
on, hence they can be reused by other tables and databases. If an update operation is made at the same time on two databases on separate networks, they cannot both accurately retrieve the next unique identifier. CouchDB does not come with an auto-increment or sequence feature. Instead, it assigns a Universally Unique Identifier (UUID) to each and every document, making it almost impossible for another database to accidentally select the same unique identifier.

4. Concept of views: Document-oriented databases do not support joins. This is a consequence of there being no primary and foreign keys in CouchDB. Instead it provides a feature called view which allows creating an arbitrary relation between documents that is not actually defined in the database itself. This means all benefits of typical SQL join queries can be achieved without the burden of predefining their relationships in the database layer.

5. Replication

CouchDB uses replication to propagate application changes across participating nodes. This is a fundamentally different approach from relational databases, which operate at different intersections of consistency, availability, and partition tolerance.

3.2 Amazon SIMPLEDB: Overview

Amazon SimpleDB is a web service providing the core database functions of data indexing and querying. This service works in close conjunction with Amazon Simple Storage Service (Amazon S3) and Amazon Elastic Compute Cloud (Amazon EC2), collectively providing the ability to store, process and query data sets in the cloud, making web-scale computing easier and more cost effective for developers. A traditional, clustered relational database requires a sizable upfront capital outlay, is complex to design, and often requires a DBA to maintain and administer. Amazon SimpleDB is simpler, requiring no schema, automatically indexing and providing a simple API for storage and access. This approach eliminates the administrative burden of data modeling, index

Consistency

All database clients see the same data, even with concurrent updates.

Availability

All database clients are able to access some version of the data.

Partition tolerance

The database can be split over multiple servers.

Fig 3.1 Difference between CouchDB and RDBMS
Developers gain access to this functionality within Amazon’s proven computing environment, are able to scale instantly, and pay only for what they use.

**Features of SimpleDB**

1. **Simple to use**

   Amazon SimpleDB provides streamlined access to the lookup and query functions that traditionally are achieved using a relational database cluster while leaving out other complex, often-unused database operations. The service allows you to quickly add data and easily retrieve or edit that data through a simple set of API calls.

2. **Flexible**

   With Amazon SimpleDB, it is not necessary to pre-define all of the data formats to be stored; simply new attributes can be added to the Amazon SimpleDB data set when needed, and the system will automatically index the data accordingly. The ability to store structured data without first defining a schema provides developers with greater flexibility when building applications.

3. **Scalable**

   Amazon SimpleDB allows to easily scale the application. New domains can be quickly created as data grows or request throughput increases. Currently, store up to 10 GB per domain can be stored and up to 100 domains can be created.

4. **Fast**

   Amazon SimpleDB provides quick, efficient storage and retrieval of data to support high performance web applications.

5. **Reliable**

   The service runs within Amazon's high-availability data centers to provide strong and consistent performance. To prevent data from being lost or becoming unavailable, fully indexed data is stored redundantly across multiple servers and data centers.

6. **Low touch**

   Accessing capabilities through the AWS cloud eliminates the complexity of maintaining and scaling operations in-house. The service allows focusing on value-added application development, rather than arduous and time-consuming database administration.

7. **Designed for use with other Amazon Web Services**

   Amazon SimpleDB is designed to integrate easily with other web-scale services such as Amazon EC2 and Amazon S3.

8. **Inexpensive**

   Amazon SimpleDB passes on the financial benefits of Amazon's scale. Vendors have to pay only for resources they consume.
Data Model for SimpleDB

![Data Model for SimpleDB](image)

Fig 3.2.1 A SimpleDB model

**Advantages**

1. **No big infrastructure investment**
   Freedom from the shackles of big infrastructure investment opens up great opportunities for innovation. Vendors can now focus on their business ideas instead of fretting over the number of servers they have, worrying about running out of disk space, etc. Amazon worries about the mundane details of the hardware and infrastructure and how to make it highly available while they can concentrate on bringing their ideas to life. Low setup costs and pay-as-you-go expansion make it perfect for startups.

2. **Maintenance is simpler**
   Setting up and maintaining a highly available clustered database that is constantly growing is extremely difficult. However, maintenance in SimpleDB is much simpler than a typical database because there is nothing to set up or configure. Amazon takes care of all the administrative tasks. Data is automatically indexed by Amazon and is available anytime from anywhere.

3. **High Scalability**
   SimpleDB is designed from the ground up to address the use case of scaling massive amounts of data utilizing a cloud approach. All data stored in SimpleDB is replicated multiple times in geographically disbursed data centers, so customer databases need not be backed and will automatically fail over to another replica if one is not available. Requests can be done via https for encryption.

4. **Parallelism**
   It is based on Erlang which supports both distributed and concurrent operations. Hence, data are stored across multiple nodes which support parallel query execution.

5. **Schema free**
   The ability to store structured data without first defining a schema provides developers with greater flexibility when building applications and eliminates the need to re-factor an entire database as those applications evolve.

6. **Better storage**
   SimpleDB can be used as both data storage and an indexing service. For example, one can use SimpleDB as a flat-file store, where each record maps onto a line. Another use case is to use each record in SimpleDB as Meta data for a media file that is stored on Amazon
S3. This solution enables quick search of media files via SimpleDB and then retrieval via Amazon S3. Individual item names, attribute names, and attribute values can be up to 1,024 bytes in length. Amazon SimpleDB allows 10GB of storage for each domain with 100 domains per customer account, which provides you with 1 TB of total storage.

**Differences between RDBMS and SIMPLEDB**

1. **Items are stored in hierarchical structure, not a table:**

   Items stored in SimpleDB domain can contain multiple attributes, each of which may have multiple values. Relationship between these resources is best visualized as a hierarchical tree structure rather than as a rigid, predefined table structure. Because there are no predefined database or table schemas, an item can have a different set of attributes from the other items in a domain. There is freedom to rearrange attribute and value portions of the tree as and when new data elements are added or needs of application are changed.

2. **Better querying Capabilities**

   Amazon SimpleDB does not storing raw data. It takes data as input and expands it to create indices across multiple dimensions to quickly query that data. Amazon SimpleDB stores smaller bits of data and uses dense drives that are optimized for data access speed.

3. **Provision of technical benefits not offered by RDBMS:**

   Amazon SimpleDB requires no schema, automatically indexes your data and provides a simple API for storage and access. This eliminates the administrative burden of data modeling, index maintenance, and performance tuning. Developers gain access to this functionality within Amazon’s proven computing environment, are able to scale instantly, and pay only for what they use. Traditionally, this type of functionality has been
accomplished with a clustered relational database that requires a sizable upfront investment, brings more complexity than is typically needed, and often requires a DBA to maintain and administer. In contrast, Amazon SimpleDB is easy to use and provides the core functionality of a database, real-time lookup and simple querying of structured data without the operational complexity.

3.3 Google’s Big Table: Overview

BigTable is a distributed storage system for managing structured data that is designed to scale to a very large size: terabytes of data across thousands of commodity servers. Many projects at Google store data in BigTable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on BigTable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving).

Despite these varied demands, BigTable has successfully provided a flexible, high-performance solution for all of these Google products. Google’s Appengine is built on top of DataStore, which is built on top of BigTable.

Google App Engine datastore

The App Engine datastore is a schemaless object datastore, with a query engine and atomic transactions. The Python interface includes a rich data modeling API and a SQL-like query language called GQL.

The Google App Engine datastore provides robust scalable data storage for your web application. The datastore is designed with web applications in mind, with an emphasis on read and query performance. It stores data entities with properties, organized by application-defined kinds. It can perform queries over entities of the same kind, with filters and sort orders on property values and keys. All queries are pre-indexed for fast results over very large data sets. The datastore supports transactional updates, using entity groupings defined by the application as the unit of transactionality in the distributed data network.

Features
1. Schema Free

The App Engine datastore is a schemaless object datastore, with a query engine and atomic transactions. The Python interface includes a rich data modeling API and a SQL-like query language called GQL.

2. Scalable

The datastore uses a distributed architecture to manage scaling to very large data sets.

3. Queries and Indexes

An App Engine datastore query operates on every entity of a given kind (a data class). App Engine has defined Query and GqlQuery class to query the datastore. The Query class is a datastore query interface that uses objects and methods to prepare queries. The GqlQuery class is a datastore query interface that uses the App Engine query language GQL. GQL is a SQL-like query language suitable for querying the App Engine datastore. An entity is returned as a result for a query if the entity has at least one value (possibly null) for every property mentioned in the query's filters and
sort orders, and all of the filter criteria are met by the property values.

4. Consistency and Reliability Of Data
With the App Engine datastore, every attempt to create, update or delete an entity happens in a transaction. A transaction ensures that every change made to the entity is saved to the datastore, or, in the case of failure, none of the changes are made. This ensures consistency of data within an entity. The datastore can execute multiple operations in a single transaction, and roll back the entire transaction if any of the operations fail. This is especially useful for distributed web applications, where multiple users may be accessing or manipulating the same data object at the same time.

5. Quotas and Limits
Each call to the datastore API counts toward the Datastore API Calls quota. Note that some library calls result in multiple calls to the underlying datastore API.
Data sent to the datastore by the app counts toward the Data Sent to (Datastore) API quota. Data received by the app from the datastore counts toward the Data Received from (Datastore) API quota.
The total amount of data currently stored in the datastore for the app cannot exceed the Stored Data (adjustable) quota. This includes entity properties and keys, but does not include indexes.

Data Model for Google’s DataStore

class Patient(db.Model):

    id = db.IntegerProperty()
    _rev= db.StringProperty()
    clinicalHistory= db.StringProperty()
    name = db.StringProperty()
    symptoms= db.ReferenceProperty(Symptom)
    personalData= db.ReferenceProperty(PersonalData)
    test= db.ReferenceProperty(Test)

where Symptom ,PersonalData and Test are other entities.

class Symptom(db.Model):

    id = db. IntegerProperty()
    symptom = db.StringProperty()

class PersonalData(db.Model):

    age= db.Integer Property()
    symptom = db.StringProperty()
class Test(db.Model):

    id= db.Integer Property()
    test_name = db.StringProperty()

**Advantages**

1. **Better Scaling**

   Unlike traditional databases, the datastore uses a distributed architecture to manage scaling to very large data sets. An App Engine application can optimize how data is distributed by describing relationships between data objects, and by defining indexes for queries.

2. **Better Design**

   The App Engine datastore is strongly consistent, but it's not a relational database.

**Differences between RDBMS and Google's BigTable**

1. **Scaling**

   Unlike traditional databases, the datastore uses a distributed architecture to manage scaling to very large data sets. An App Engine application can optimize how data is distributed by describing relationships between data objects, and by defining indexes for queries.

2. **Uniqueness Of Data**

   The concept of Primary Key in Relational Database ensures uniqueness of data. There is no such concept in Google’s DataStore. Datastore does not provide any implicit method to maintain uniqueness. The only thing that is unique is key/key_name across all your entities. However, there have been a couple of methods (individual projects) that provide this feature. If you don’t use above framework, or create something of your own to maintain uniqueness, AppEngine has no problem with duplicate entries, since the entries are not duplicate. Every entity has a unique key/key name.

3. **Relationship between entities**

   The concept of foreign key, helps us in establishing one-one, one-many, many-many relationship between various entities. However, there is no such concept in Datastore. For implementing something similar to foreign key, you need what is called Reference Property. Again, reference property is nothing like foreign key. So, don’t expect an error when an entity is deleted and the other entity still points to the non-existing entity.
4. COMPARISON BETWEEN NON-RELATIONAL DATABASES

Scorecard for non-relational databases

<table>
<thead>
<tr>
<th>Database</th>
<th>Capability</th>
<th>Manageability</th>
<th>Availability</th>
<th>Scalability</th>
<th>Value</th>
<th>Overall Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache CouchDB</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>7.4</td>
</tr>
<tr>
<td></td>
<td>25%</td>
<td>25%</td>
<td>20%</td>
<td>20%</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>Amazon SimpleDB</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>8.2</td>
</tr>
<tr>
<td></td>
<td>25%</td>
<td>25%</td>
<td>20%</td>
<td>20%</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>Google BigTable</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>8.2</td>
</tr>
</tbody>
</table>

Table 4.1 Performance Matrix

5. CONCLUSION

Thus, through various illustrations given in this paper, we have tried to showcase features of non relational databases which ensure that better integrity, scalability, robustness are achieved. Enhanced data modeling and representation, faster computations, load balancing are the major targets of using these database concepts. They can bring about a revolution in this modern era of supercomputer power through their resource centric and support for versatile data approaches.

6. REFERENCES


