DATA MANAGEMENT IN CLOUD ENVIRONMENTS: NoSQL and NewSQL DATA STORES
(JOURNAL OF CLOUD COMPUTING 2013)

Presented by:
Faizan Ahmed
AUTHORS

- KATARINA GROLINGER
- WILSON A HIGASHINO
- ABHINAV TIWARI
- MIRIAM AM CAPRETZ

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING
WESTERN UNIVERSITY OF LONDON, ONTARIO
Advances in Web Technology and IoT (Internet of Things) have led to data explosion.

Massive amounts of data generated by businesses cannot be handled by traditional RDBMS tools.

The Big Data Reality:
- Volume
- Velocity
- Variety
“A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction”

- Provides benefits in the form of:
  - Lower up-front investment
  - Lower operating costs
  - Ease of Scalability (horizontal and vertical)
  - Elasticity
  - Reduced business risks due to high fault tolerance

- High extent of synergy between processing requirements of big data applications & computational resources offered by cloud computing.
Large number of options available in the form of NoSQL & NewSQL.

This is both a blessing and a problem:
- Blessing because many alternatives are available which suit application specific requirements
- Challenging because the existing number of solutions and the discrepancies within them make it difficult to select an appropriate solution for a problem at hand.

The paper aims to summarize these discrepancies with NoSQL and NewSQL solutions and provides use case scenarios which fit a particular family of data stores for the problem at hand by summarizing the characteristics and features of each of these solutions categorically.
The CAP principle states that in the event of a partition (P), the system must choose either consistency (C) or availability (A), and cannot provide both simultaneously.

**Option 1:** In the event of a partition, choose C over A (CP system). Example: distributed ACID database (NewSQL).

**Option 2:** In the event of a partition, choose A over C (AP system). Example: eventually consistent data systems like NoSQL.

**Option 3:** In the event of no partition, choose CA system. Example: SQL RDBMS.
## ACID vs BASE SYSTEMS

| ACID                                                                 | BASE                                                                 |
|----------------------------------------------------------------------|----------------------------------------------------------------------|
| Atomicity, Consistency, Isolation & Durability                      | Basically Available, Soft-state, Eventually Consistent              |
| Relational Data Model                                               | Simple and flexible non-relational data model                       |
| SQL systems more suited to vertical scalability                    | Allows better horizontal scalability.                               |
| Much more suited for mission critical strongly consistent transaction based systems. | Provide a much better environment for cloud data management in a cluster scenario. |
| Consists of a standard query language i.e. SQL                     | No standard query language                                          |
| Has standardized interfaces e.g Oracle Lite Client, MySQL Lite     | Third Party interfaces                                              |
| Huge investments already made in this area                         | Difficult to migrate data for enterprises from SQL systems.         |
METHODOLOGY

DB-Engines Ranking:
- [http://db-engines.com/en/ranking](http://db-engines.com/en/ranking)
- Ranks database systems according to their popularity by using parameters such as the number of mentions on Web sites, Google trends, number of job offers and number of professional profiles in which solutions are mentioned.
- In short measures overall popularity of database systems

IEEE Xplore database:
- Most often cited database systems in the NoSQL and NewSQL categories were chosen for this paper.
THE CONTESTANTS

| NoSQL                       | NewSQL                                                   |
|-----------------------------|----------------------------------------------------------|
| **Document based**          | Fewer selections since the NewSQL technologies are fairly recent and not very popular yet. |
| - MongoDB                   | VoltDB                                                   |
| - CouchDB                   | Google Spanner                                           |
| - Couchbase                 | Clustrix                                                  |
| **Column-Family based**     |                                                           |
| - HBase                     |                                          |
| - Cassandra                 |                                          |
| - SimpleDB                  |                                          |
| - DynamoDB                  |                                          |
| **Key-Value Data Stores**   |                                          |
| - Redis                     |                                          |
| - Memcached                 |                                          |
| - Riak                      |                                          |
| - BerkeleyDB                |                                          |
| - Voldemort                |                                          |
| **Graph Databases**         |                                          |
| - Neo4J                     |                                          |
| - HyperGraphDB              |                                          |
| - Allegro Graph             |                                          |
Comparitive Analysis : NoSQL and NewSQL

We will now perform a comparative analysis between NoSQL & NewSQL data stores based on the following dimensions:

• DATA MODELS
• QUERYING CAPABILITIES
• SCALABILITY
• PARTITIONING
• REPLICAATION
• CONSISTENCY
• CONCURRENCY CONTROL
• SECURITY
Based on key value pairs similar to an associative map data structure

- The values are *opaque* to the data store and thus can be used to store any type of data structure including strings, lists, tuples etc.

- Schema free

- Due to opaqueness of values, these data stores do not support data level querying and indexing and only performs queries based on keys.

- Not suitable for scenarios requiring relations

---

**NoSQL: KEY-VALUE STORES**

- redis
- memcached
- riak
- Oracle Berkeley DB
NoSQL: COLUMN-FAMILY STORES

- Mostly derived from Google’s Bigtable
- Dataset consists of several rows, each addressed by a unique row key
  - Each row comprises of set of column families
  - Each column family further acts as a key for the one of or more columns it holds.
  - Each column holds a name-value pair.
- Implementation
  - Hadoop HBase implements directly matches this model whereas Amazon’s SimpleDB and DynamoDB implement a small variation.
- Data belonging to the same row resides on the same server node
- Unlike SQL, a prior definition of columns is not required
- The additional indexing based on column families makes column-family stores querying model more powerful than key-value stores.
NoSQL: DOCUMENT STORES

- Using keys to locate document
- Documents represented using JSON or BSON
- Can contain complex nested data structures
- Schema-less
- Documents can be grouped into collections
- Each document can be indexed based on the primary key or contents of the document
- Hence the values are *not* opaque to the data store
- Suitable for scenarios in which data can be represented as a document

mongoDB

CouchDB

Couchbase
NoSQL: GRAPH DATABASES

- Originated from graph theory
- Efficiently stores relationships between different data nodes unlike the former data models.
  - Nodes and edges have individual properties consisting of key value pairs.
- Very efficient in traversing relationships between entities
- Not efficient in horizontal scalability as there is a performance overhead related.
- Application involving graph databases have narrow scope.
Figure 1: Different types of NoSQL data models.
NewSQL

- Offers a purely relational model of data
- Addresses limitations of scalability of SQL systems with stronger consistency.
- ACID support for transactions
- Non-locking concurrency control
- NuoDB set out to be a cluster-first SQL database with a focus on cloud-ops:
  - Run on many nodes across many datacenters and let the underlying system manage data locality and consistency for you.
- Volt DB is the most mature NewSQL data store
Querying: NoSQL

- Querying capabilities depend directly on the underlying data model.
- Opacity of data contents enables the querying or searching by value in a value store or by contents in a document.
- **Map-Reduce:**
  - A widely accepted approach for performing distributed data processing on a cluster of computers.
  - Many NoSQL data stores support the Map-Reduce paradigm.
- **SQL** is the most standard and adopted way of querying data and hence many NoSQL data stores like MongoDB offer SQL-like query language support.
NewSQL solutions like NuoDB and Clustrix offer the most SQL-like querying language with a few restrictions.

VoltDB has even higher restrictions:
- Use of `having` clause is not available
- Tables cannot perform a self join

Google Spanner has a query language which looks similar to SQL with some extensions to support extra functionality.

Both NoSQL and NewSQL now provide REST API’s to facilitate querying via simple GET and POST requests.
Traditional RDBMS’s don’t scale well horizontally.

One of the main reasons for the advent of NoSQL and NewSQL databases was the need for horizontal scalability whereby simple commodity servers could be used to increase computing power.

Three dimensions for measuring scalability:
- Scaling read requests
- Scaling write requests
- Scaling data storage

Horizontal Scalability in both NoSQL and NewSQL is heavily dependent upon the following characteristics:
- Partitioning strategy
- Consistency model
- Replication
- Concurrency Control
Partitioning strategy is heavily dependent on the data model used.

Two main partitioning strategies are Range Partitioning and Consistent Hashing.

Range Partitioning:
- Assigns data to partitions based on ranges of a partition key.
- A particular node is responsible for the read/write handling of a specific range of keys and data storage.
- An advantage is effective processing of range queries
- Possibly could lead to generation of hot spots and load-balancing issues
- A special server needs to be dedicated which acts as a routing server.
Dataset is represented in a ring format
- Ring of data is divided into number of ranges = the number of nodes
- Each node is mapped to a point in the data ring
- Each node is responsible for storing the data object in the ring region between itself and its predecessor

Figure 2 Consistent hashing.
Advantages of consistent hashing:
- No mapping service required unlike range partitioning
- Dynamic resizing

Disadvantages of consistent hashing:
- Not fit for range queries because neighboring keys are distributed across different nodes which can have a performance cost.
- Some in-memory data stores leave it to the client to devise a partitioning strategy manually.
- Partitioning of graph databases is specially challenging because graphs do not have stable keys.
  - A trade-off is achieved by keeping related graph nodes on the same server to provide good transversal performance
  - However too many nodes are not concentrated on the same node in order to avoid load-balancing issues.
| Range Partitioning | Consistent Hashing | Manual Partitioning | Other Techniques |
|--------------------|--------------------|---------------------|------------------|
| BerkeleyDB         | DynamoDB           | Redis               | Google Spanner   |
| Cassandra          | CouchDB            | Memcache            | NuoDB            |
| HBase              | VoltDB             | SimpleDB            | HyperGraphDB     |
| MongoDB            | Clustrix           | Neo4J               |                  |
|瑞典黑魔法          | Voldemort          | Allegro Graph       |                  |
| Riak               |                    |                     |                  |
| Couchbase          |                    |                     |                  |
Replication causes an increase in read/write scalability, system reliability, fault tolerance and durability.

Three main techniques:
- Master-slave model
  - Only master processes write requests
  - Only one master node
  - Changes propagated from masters to slaves
- Multi Master/Masterless model
- Master-slave model:
  - Only master processes write requests
  - Only one master node
  - Changes propagated from masters to slaves
- Multi-master model:
  - Multiple nodes can process write requests
  - Changes propagated to remaining nodes
Figure 3 Replication models.

a) Master-slave replication

b) Multi-master replication
Master-slave replication model is useful for scaling only read requests.

Multi-master and masterless models scale both read and write requests.

However, there is cost of synchronization between replicas dependent upon the type of replication model:

- Master-slave replication model uses synchronous synchronization model in which changes are propagated to the replicas before acknowledgement of write request.
- Multi-master or masterless systems use asynchronous synchronization model in which acknowledgement of a write operation is given before changes are propagated.
- Synchronous model achieves consistent replica copies whereas asynchronous model may result in inconsistent copies of data.
# Replication Categorization

| Master-slave   | Multi-master | Master-less | Others     |
|----------------|--------------|-------------|------------|
| Redis          | CouchDB      | Voldemort  | Memcached  |
| BerkeleyDB     | Couchbase    | Riak        | DynamoDB   |
| HBase          | HyperGraphDB | Cassandra   | SimpleDB   |
| MongoDB        | NuoDB        |             | VoltDB     |
| Neo4J          |              |             | Spanner    |
| Allegro Graph  |              |             | Clustrix   |
Consistency as used in CAP Theorem, not the same as consistency in ACID.

Two main consistency models: Strong & Eventual

Strong Consistency:
- Same data visible to all read requests after a successful write request
- Synchronous replication ensures strong consistency
- Not suitable for NoSQL systems since it introduces latency
- HBase the only NoSQL data store which supports strong consistency

Eventual Consistency:
- Changes eventually propagate given sufficient time
- Some nodes may contain inconsistent copies transiently
- Asynchronous replication causes eventual consistency

Most NoSQL data stores give option to select consistency models
| Strong Consistency | Eventual Consistency | Configurable   | Quorum    |
|--------------------|----------------------|----------------|-----------|
| Memcached          | Redis                | BerkeleyDB     | Voldemort|
| HBase              | CouchDB              | DynamoDB       | Riak      |
| Couchbase          | Neo4J                | SimpleDB       | Cassandra |
| VoltDB             | HyperGraphDB         | MongoDB        |           |
| Spanner            | Allegro Graph        |                |           |
| Clustrix           | NuoDB                |                |           |
A means to achieving simultaneous access to the same entity, row, or record on a single server node.

Concurrency control schemes are either **pessimistic** or **optimistic**:

- **Pessimistic locking** assumes that two or more users will try to update the same entity at the same time
  - To prevent this a lock is placed on the resource to give exclusive rights so that updates are guaranteed in a single operation
  - This strategy can lead to performance degradation specially in write-intensive scenarios.

- **Optimistic locking** assumes that conflicts are possible but happen rarely
  - Instead of locking the record, the data store check at the end of a write operation whether concurrent users have attempted to modify the same record
  - If a conflict is found then different conflict resolution strategies can be used.
MVCC is a conflict resolution model within optimistic concurrency.

In MVCC, when the data store needs to update a record, instead of overwriting the record a new version of the record is created and marked as current; the old data is marked as obsolete.

With MVCC, a read operation sees the data the way it is when it began reading.

This can usually result in many version of the same data existing in the data store.
Google Spanner implements pessimistic locking techniques for read-write transactions. Read-only transactions are lock-free.

VoltDB uses a single-threaded lock free environment for all transactions which are executed sequentially.

Clustrix and NuoDB use the MVCC approach.
## Concurrency Control Categorization

| Pessimistic  | Optimistic | Configurable | Other          |
|--------------|------------|--------------|----------------|
| Spanner      | Voldemort  | Redis        | Cassandra      |
| Neo4J        | Riak       | Memcached    | MongoDB        |
| MongoDB      | HBase      | DynamoDB     | Couchbase      |
| BerkeleyDB   | CouchDB    |              | Allegro        |
|              | HyperGraphDB |              | VoltDB         |
|              | Clustrix    |              |                |
|              | SimpleDB    |              |                |
Security is an aspect usually overlooked by NoSQL data stores. The following features of security w.r.t a data store are considered:
- Authentication
- Authorization
- Encryption
- Auditing

Usually NoSQL and NewSQL data store documentations do not cover the topic of security clearly. Some even outrightly state ‘NA’ for some features.

Generally speaking the security of these data stores is immature as compared to the counterpart traditional SQL systems.

Usually these data stores assume that the network infrastructure/admin is responsible for ensuring protection from unauthorized access.
## SECURITY CATEGORIZATION

| No-Authentication | No-Authorization | Encryption (Data) | Auditing       |
|-------------------|------------------|-------------------|----------------|
| BerkeleyDB        | Redis            | BerkeleyDB        | Cassandra      |
| Voldemort         | Memcached        | Voldemort         | DynamoDB       |
| Riak              | BerkeleyDB       | Cassandra         | Allegro Graph  |
| HyperGraphDB      | Voldemort        | MongoDB            | VoltDB         |
|                   | Riak             | NuoDB              | NuoDB          |
|                   | Couchbase        |                   |                |
|                   | Neo4J            |                   |                |
|                   | HyperGraphDB     |                   |                |
USE CASES

- **Key-Value Stores (Good for):**
  - Storing Web Session Info
  - User Profiles
  - Shopping Cart Data (Querying based on User’s ID)
  - Content provider applications (Image/Video IDs)
  - Caching layer (Storing results of processing intensive requests)

- **Limitations:**
  - Highly interconnected data
  - Operations involving multiple-key data
- **Good for:**
  - Blogging platforms
  - Content Management Systems (CMS)
  - Logging Event Streams (Business Enterprises)
  - Store sensor network data
  - Write intensive Operations (Cassandra)
  - Data Analytics with MapReduce aggregation (HBase)
  - Personalized search applications

- **Limitations:**
  - Data cannot be joined
  - Transactions involving multiple documents
GRAPH BASED

- **Good for:**
  - Location-based services
  - Recommendation Services
  - Social Networks
  - Complex queries requiring multiple joins

- **Limitations:**
  - Horizontal Scalability
  - Traversing relationships comes at a cost.
Good for:
- Financial Markets, Banking Operations
- Schema is known upfront and unlikely to change
- Traditional DBMS applications which require horizontal scalability
- Applications requiring ACID and scalability.

Limitations:
- No NewSQL systems are as general-purpose as traditional SQL systems set out to be.
- Offers only partial access to the rich tooling of traditional SQL systems.
CRITIQUE

- **Pros’s:**
  - Holistic and clear comparison of various features of NoSQL and NewSQL systems over various dimensions
  - Security, which is usually an overlooked aspect in NoSQL and NewSQL systems has been thoroughly explored
  - Detailed discussion on the suitability of various NoSQL and NewSQL solutions for different sets of applications
  - Good perspective provided in the domain of cloud computing.

- **Con’s:**
  - All features stated of various NoSQL and NewSQL data stores is taken from their respective documentation only. No experimentation was done to validate the claims of the data stores.
  - Data model for NewSQL data stores is not as clearly explained as for NoSQL.
  - No comparison regarding performance is stated regarding in-memory and disk-based NewSQL data stores.
THANK YOU!