Can the Elephants Handle the NoSQL Onslaught?

Avrilia Floratou, Nikhil Teletia
David J. DeWitt, Jignesh M. Patel, Donghui Zhang

University of Wisconsin-Madison
Microsoft Jim Gray Systems Lab

Presented by Limor Lazar
Introduction - NoSQL

Not Only SQL

Leverage the NoSQL boom
Introduction overview

- Why now?
- What is it?
- Pros and Cons of No-SQL
- Short overview of two types of database
Why now?

- Big data
- Concurrency
- Connectivity
- Diversity
As we know from RDBMS...

**ACID**

**ATOMICITY:** All or nothing

**CONSISTENCY:** Any transaction will take the db from one consistent state to another, with no broken constraints (referential integrity)

**ISOLATION:** Other operations cannot access data that has been modified during a transaction that has not yet completed

**DURABILITY:** Ability to recover the committed transaction updates against any kind of system failure (transaction log)
CAP Theorem

Eric A. Brewer:

“It is impossible for a distributed computer system to simultaneously provide all three of the guarantees”

- Consistency
- Availability
- Partition Tolerance

Relational Databases emphasise Consistency, so either Availability or Partition Tolerance will suffer
What is it – No SQL

Non-relational database management systems

Designed for distribute data stores where very large scale of data storing needs (e.g. Facebook, Google which collect terabits of data every day for their users)

Not required fixed schema
Ability to dynamically define new attributes

Does not use SQL as its query language
What is it – No SQL

- Data is partitioned among different machines (no JOIN operation) - Typically scale horizontal
- Not guaranty full ACID (Atomicity, Consistency, Isolation, Durability)
- Guaranty only three of them allow the DB to be faster and distribute the data on several server
- Eventual Consistency is guaranteed
Pros & Cons

- Scale easily
  Distributed, ability to horizontal scale (by adding more servers)
- with the data held in a redundant manner on several servers (a failure of a server can be tolerant)
- Don’t required (an expensive and complex) queries optimization engine
- Can analyze the data outside the NO-SQL engine (for example by using map-reduce) from several servers in parallel.

- No indexing support (Some solutions like MongoDB have indexing but it’s not as powerful as in SQL solutions).
- No ACID
- Absence of standardization. No standard APIs or query language
NoSQL – distribute & replication
No SQL Categories

- Key - Value
- Big Table
- Documents
- Graph
Scalable, high-performance, open source, schema-free, document oriented database.

Support indexing and replications

Documents are stored in BSON (binary JSON)

Javascript-based query syntax
## Documents oriented: MongoDB

<table>
<thead>
<tr>
<th>RDBMS (mysql, postgres)</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables</td>
<td>Collections</td>
</tr>
<tr>
<td>Records/rows</td>
<td>Documents/objects</td>
</tr>
<tr>
<td>Queries return record(s)</td>
<td>Queries return a cursor</td>
</tr>
</tbody>
</table>
use mydb

j = {name:"mongo"}
k = {x:3}
db.things.insert(j)
db.things.insert(k)

db.things.find
Map-Reduce based systems (Hive on Hadoop)

- Originally built by Facebook, used to analyze and query ~15TB of log data each day
- Provides a SQL-like query language for writing concise queries on data in hive tables
- Hive engine compiles the queries into efficiently chained map-reduce jobs
- Tables in Hive can be partitioned (range partition e.g. by date) and also bucketed (hash partitions - may improve queries)
CREATE TABLE user_info_bucketed(user_id BIGINT, firstname STRING, lastname STRING) COMMENT 'A bucketed copy of user_info'
PARTITIONED BY(ds STRING)
CLUSTERED BY(user_id) INTO 32 BUCKETS;

Example to location of HDFS directory:
/wh/user_info_bucketed/ds=20090801/part-00000
The **Map** function: processes a key-value pair to generate a set of intermediate key/value pairs.

**Input:** a pair  
**Output:** set of intermediate pair

**MapReduce:** groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function.
Map Reduce

The *Reduce* function: merges all intermediate values associated with the same intermediate key.

Input: an intermediate key and a set of values for that key
Output: set of values

*MapReduce*: merges together this values to form a smaller set of values
The Map invocation are distributed across multiple machines.

Reduce invocation are distributed by partitioning the intermediate key space.
Map Reduce

Input: a collection of keys and their values

User-written Map function

Each input (k,v) mapped to set of intermediate key-value pairs

User-written Reduce function

One list of intermediate values for each key: (k, [v_1,...,v_n])

sort all key-value pairs by key

Shuffle
Problem:
counting number of occurrences of each word in a large collection of documents.

Solution:
Map (key = url, value = contents):
   For each word w in content, emits (w, "1")
reduce (key = word, values = counts):
   Sum all "1" in values list
   Emit results "(word, sum)"
Table of Content

- Motivation to the experiment
- Some more definitions
- The experiment
- Results
- Conclusions
SQL v/s NoSQL

NoSQL
- Document Stores
- Key-Value Stores
- MapReduce

RDBMS
- OLTP
- OLAP

MongoDB

Interactive data-serving applications

Hive

Data analytics
Motivation

How does the performance of RDBMSs compare to that of NoSQL systems for:

- interactive data serving applications which are characterized by the “new” OLTP workloads?
- data analysis applications on massive amounts of data
Methodology

Take one leading and representative system from each class and benchmark them against each other.

DSS
- HIVE
- SQL Server-PDW

OLTP
- MongoDB
- SQL Server
The BIG question

How does the performance of RDBMSs solutions compare to the NoSQL systems for the two classes of workloads interactive data-serving environments and decision support system (DSS)?
We are going to compare:

- SQL Server
- MongoDB
- PDW
- Hive

Comparison metrics:
- YCSB Benchmark
- TPC-H DSS Benchmark
Background

MongoDB

YSCB

TPC-H

Hive

PDW
Parallel Data Warehouse:

- Classic shared-nothing parallel database system
- Build on top of SQL Server
- Consist of multiple compute notes and a single control node
- Each compute node is a separate server running SQL Server
- The control node is responsible for handling the user query and generating an optimized plan of parallel operations
TPC-H

- a decision support benchmark
- It execute queries with a high degree of complexity
- Evaluates the performance of various support systems by execution of sets of queries against standard database

- Used for testing Hive vs. SQL
- To keep it “fair fight” we use same hardware (and configuration) for both executions
Yahoo Cloud Serving Benchmark:
- A framework for evaluating the performance of different “key-value” serving store
- It can compare between several workloads (example: 40% reads and 60% writes)
- Used for testing MongoDB vs. SQL
Background

- MongoDB
- YSCB
- TPC-H
- Hive
- PDW
Methodology

Take one leading and representative system from each class and benchmark them against each other.
We used TPC-H at four scale factors: 250 GB, 1000 GB, 4000 GB, 16000 GB.
Each scale factor represents cases where different portions of the TPC-H tables fit in main memory.
Hive on Hadoop – Data Loading

Node 1: HDFS
Node 2: HDFS
Node 3: HDFS
Node 4: HDFS

Data File

TPC-H

PDW  PDW  PDW  PDW
Hive Tables

- Hive table can contain both partitions and buckets:
  - In this case the table consists of a set of directories (one for each partition)
  - Each directory contains a set of files, each one corresponding to one bucket

- only partitions:
  - Can improve selections queries but not join

- only buckets:
  - Can help improve the performance of joins but cannot improve selections queries.
PDW tables

- Can be horizontally partitioned:
  - The records are distributed to the partitions using a hash function on a partition column
  - Can be replicated across all the nodes
## The Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>HIVE</th>
<th>PDW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partition Column</strong></td>
<td><strong>Buckets</strong></td>
<td><strong>Partition Column</strong></td>
</tr>
<tr>
<td>Customer</td>
<td>c_nationkey</td>
<td>8 buckets per partition on c_custkey</td>
</tr>
<tr>
<td>Lineitem</td>
<td>--</td>
<td>512 buckets on l_orderkey</td>
</tr>
<tr>
<td>Nation</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Orders</td>
<td>--</td>
<td>512 buckets on o_orderkey</td>
</tr>
<tr>
<td>Part</td>
<td>--</td>
<td>8 buckets on p_partkey</td>
</tr>
<tr>
<td>Partsupp</td>
<td>--</td>
<td>8 buckets on ps_partkey</td>
</tr>
<tr>
<td>Region</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Supplier</td>
<td>s_nationkey</td>
<td>8 buckets per partition on s_suppkey</td>
</tr>
</tbody>
</table>
SQL v/s NoSQL on DSS

Normalized Arithmetic Mean

<table>
<thead>
<tr>
<th>TPC-H Scale Factor</th>
<th>HIVE</th>
<th>PDW</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>48</td>
<td>4</td>
</tr>
<tr>
<td>4000</td>
<td>148</td>
<td>17</td>
</tr>
<tr>
<td>16000</td>
<td>500</td>
<td>72</td>
</tr>
</tbody>
</table>

Normalized Geometric Mean

<table>
<thead>
<tr>
<th>TPC-H Scale Factor</th>
<th>HIVE</th>
<th>PDW</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>52</td>
<td>5</td>
</tr>
<tr>
<td>4000</td>
<td>144</td>
<td>18</td>
</tr>
<tr>
<td>16000</td>
<td>474</td>
<td>72</td>
</tr>
<tr>
<td>Query</td>
<td>SF - 250 GB Time (sec)</td>
<td>SF - 250 GB Speedup</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Q1</td>
<td>207</td>
<td>54</td>
</tr>
<tr>
<td>Q2</td>
<td>411</td>
<td>7</td>
</tr>
<tr>
<td>Q3</td>
<td>508</td>
<td>32</td>
</tr>
<tr>
<td>Q4</td>
<td>367</td>
<td>8</td>
</tr>
<tr>
<td>Q5</td>
<td>536</td>
<td>33</td>
</tr>
<tr>
<td>Q6</td>
<td>79</td>
<td>5</td>
</tr>
<tr>
<td>Q7</td>
<td>1007</td>
<td>19</td>
</tr>
<tr>
<td>Q8</td>
<td>967</td>
<td>9</td>
</tr>
<tr>
<td>Q9</td>
<td>2033</td>
<td>207</td>
</tr>
<tr>
<td>Q10</td>
<td>489</td>
<td>14</td>
</tr>
<tr>
<td>Q11</td>
<td>242</td>
<td>3</td>
</tr>
<tr>
<td>Q12</td>
<td>253</td>
<td>5</td>
</tr>
<tr>
<td>Q13</td>
<td>392</td>
<td>51</td>
</tr>
<tr>
<td>Q14</td>
<td>154</td>
<td>7</td>
</tr>
<tr>
<td>Q15</td>
<td>444</td>
<td>21</td>
</tr>
<tr>
<td>Q16</td>
<td>460</td>
<td>36</td>
</tr>
<tr>
<td>Q17</td>
<td>654</td>
<td>93</td>
</tr>
<tr>
<td>Q18</td>
<td>786</td>
<td>20</td>
</tr>
<tr>
<td>Q19</td>
<td>376</td>
<td>16</td>
</tr>
<tr>
<td>Q20</td>
<td>606</td>
<td>20</td>
</tr>
<tr>
<td>Q21</td>
<td>1431</td>
<td>31</td>
</tr>
<tr>
<td>Q22</td>
<td>908</td>
<td>19</td>
</tr>
<tr>
<td>AM</td>
<td>605</td>
<td>32</td>
</tr>
<tr>
<td>GM</td>
<td>474</td>
<td>19</td>
</tr>
<tr>
<td>AM-9</td>
<td>537</td>
<td>24</td>
</tr>
<tr>
<td>GM-9</td>
<td>442</td>
<td>17</td>
</tr>
</tbody>
</table>
Query 1:
scans the lineitem table and performs an aggregation followed by an order-by clause.

Query 5:
join six tables and then performs aggregation.

Query 19:
joins two tables (lineitem, part) and performs an aggregation on the output
SQL v/s NoSQL on DSS

TPC-H on Hive and SQL-PDW

SQL-PDW outperforms Hive by a factor of 6.6X at the 16TB scale factor when using a cluster of 16 compute nodes even when indices are not used in SQL-PDW.

- Hive doesn’t exploit partitioning attributes as well as PDW does.
- Cost-based optimization in PDW results in better join ordering.
- Hive’s most efficient storage format (RCFile) is CPU-bound during scans.
SQL v/s NoSQL on DSS

Scale-up results at 4 TPC-H scale factors

Speedup based on GM Of SQL-PDW over HIVE

<table>
<thead>
<tr>
<th>TPC-H Scale Factor</th>
<th>250</th>
<th>1000</th>
<th>4000</th>
<th>16000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>26.3</td>
<td>11</td>
<td>7.8</td>
<td>6.6</td>
</tr>
</tbody>
</table>
Methodology

Take one leading and representative system from each class and benchmark them against each other.

DSS
- HIVE
- SQL Server-PDW

OLTP
- MongoDB
- SQL Server
MongoDB (Mongo-AS)

Shard 1
- mongod process

Shard 2
- mongod process

Shard 3
- mongod process

Shard 4
- mongod process

Config Server
- config process

Router Process
- mongos process

Client
- Key > 8
- Value = "abc", "ldk"
- Shards: {2, 3}

Client
- Key > 8

Client
- Key > 8
MongoDB (Mongo-AS) - Data Loading

Shard 1
- mongod process
  - Keys: 1-4
- Config Server
  - config process

Shard 2
- mongod process

Shard 3
- mongod process

Shard 4
- mongod process

Router Process
- mongos process

Client
- Key: 1, Value: “abc”
- Key: 2, Value: “def”
- Key: 3, Value: “ghi”
- Key: 4, Value: “jkl”
The data is spread across the servers by hash-partitioning the key of each record.
The sharding library determines the partition by hash-partitioning the key.
## System Comparison

<table>
<thead>
<tr>
<th>Uses range partitioning</th>
<th>Use hash partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses three types of processes (mongod, mongos, config)</td>
<td>Use only one type of process (SQL Server)</td>
</tr>
<tr>
<td>Supports auto-sharding</td>
<td>Do not support auto-sharding</td>
</tr>
<tr>
<td>Supports read/write atomic operations on single data entities</td>
<td>Each SQL-CS node supports full ACID and multiple isolation levels</td>
</tr>
</tbody>
</table>
Experimental Setup

Server Nodes

S1  S2  S3  S4  S5  S6  S7  S8

Client Nodes

C1  C2  C3  C4  C5  C6  C7  C8

Mongo-AS

Config Server
## Software Configuration

<table>
<thead>
<tr>
<th>MongoDB</th>
<th>SQL Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB version 1.8.2</td>
<td>SQL Server 2008</td>
</tr>
<tr>
<td>16 mongod instances per server node (higher concurrency)</td>
<td>1 SQL Server instance per server node</td>
</tr>
<tr>
<td>No Durability</td>
<td>Full ACID (Read Committed)</td>
</tr>
<tr>
<td>8 mongos processes one per server node and 1 config process (Mongo-AS)</td>
<td></td>
</tr>
</tbody>
</table>
Workload Description

- **Database Size**: 640 million records (~80 GB per server node)
- **Record Size**: 1024 bytes
  - 24-byte key, 10 extra fields of 100 bytes each
- **Index** on record key on MongoDB and SQL Server.
- The record key is the **shard** key for Mongo-AS.

Used the Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al.]
The YCSB benchmark aims to describe the tradeoffs between throughput and latency.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload A (Update heavy)</td>
<td>Read:50%, Update:50%</td>
</tr>
<tr>
<td>Workload B (Read heavy)</td>
<td>Read:95%, Update:5%</td>
</tr>
<tr>
<td>Workload C (Read only)</td>
<td>Read:100%</td>
</tr>
<tr>
<td>Workload D (Read latest)</td>
<td>Read:95%, Append:5%</td>
</tr>
<tr>
<td>Workload E (Short ranges)</td>
<td>Scan:95%, Append:5%</td>
</tr>
</tbody>
</table>
Transactions

- Each read transaction returns the whole record.
- Each update transaction updates one field.
- Each scan transaction reads at most 1000 records.
- Each append transaction inserts a new record in the database whose key has the next greater value than that of the last inserted key by the same client node.
Workload Description

800 client threads

Server Nodes

S1

S2

S3

S4

S5

S6

S7

S8

100 client threads

100 client threads

100 client threads

100 client threads

100 client threads

100 client threads

100 client threads

100 client threads

Client Nodes

C1

C2

C3

C4

C5

C6

C7

C8
Workload Description

- The workloads are run *sequentially* and before every run the system is restarted.

- Each experiment is run for **30 minutes** and the values of *latency* and *throughput* reported are the *averages* over the last 10-minute interval.
SQL-CS has lower update latency than MongoDB at all intermediate target throughput values.
Workload C - Read Performance

Read Only
100% Reads
(Zipfian Distribution)

Throughput (Kops/sec)

Read Latency (ms)

Mongo-AS

SQL-CS

59
Workload C - Read Performance

![Graph showing read latency and throughput for different workloads and databases. The graph compares Mongo-AS and SQL-CS.]
Workload C- Read Performance

Read Latency (ms) vs. Throughput (Kops/sec) for MongoDB with Adaptive Sampling (Mongo-AS) and SQL Content Store (SQL-CS).
Workload C - Read Performance

![Graph showing read latency (ms) vs throughput (Kops/sec) for MongoDB-AS and SQL-CS. The graph illustrates the performance comparison between the two systems across varying throughput levels.](image-url)
SQL-CS reads 8K per read request whereas MongoDB reads 32K per read request. Mongo-AS and waste bandwidth by reading data that is not needed.

SQL-CS can achieve the highest throughput (125 KOps/sec) and at the same time is 2X faster than MongoDB.
Mongo-AS has lower scan latency because it uses range partitioning whereas SQL-CS use hash partitioning.

Mongo-AS can achieve the highest throughput (6000 Ops/sec) and is 5X faster than SQL-CS.

Short Ranges
95% Scans, 5% Appends
(Zipfian/Uniform Distributions)
Workload E – Append Performance

Short Ranges
95% Scans, 5% Appends
(Zipfian/Uniform Distributions)

Mongo-AS has very high append latency.
Conclusions

Our results find that the relational systems continue to provide a significant performance advantage over their NoSQL counterparts, but the NoSQL alternatives can incorporate the techniques used by RDBMSs to improve their performance.

SQL systems and NoSQL systems have different focuses on non-performance related features (e.g. support for automatic load balancing, different data models).

It is likely that in the future these systems will start to compete on the functionality aspects.
Facebook used Cassandra to power Inbox Search, with over 200 nodes deployed. This was abandoned in late 2010 when they built Facebook Messaging platform on HBase.

IBM has done research in building a scalable email system based on Cassandra.

Twitter announced it is planning to use Cassandra because it can be run on large server clusters and is capable of taking in very large amounts of data at a time. Twitter continues to use it but not for Tweets themselves.