NoSQL Databases

I think we should build an SQL database.

Uh-oh.

Does he understand what he said or is it something he saw in a trade magazine ad?

What color do you want that database?

I think mauve has the most RAM.
Acknowledgements

• Material from
  – Stanford courses (CS145 and CS347)
  – Washington University
  – Illinois University
  – Cattell’s paper and website
Contents

• Intro, motivation, key definitions
• Overview
• Systems
  – Cassandra
  – HBase
• Applications
Not every data management/analysis problem is best solved using a traditional DBMS

Database Management System (DBMS) provides:

- efficient, reliable, convenient, and safe multi-user storage of and access to massive amounts of persistent data.
Traditional DBMSs

These types of DBMSs show severe limitations due to challenges posed by big data.

One architectural feature that may not respond promptly is **consistency** *(the second of the ACID properties of transactions)*
Traditional DBMSs

Consistency Types

**Strict:** The changes to the data are atomic and appear to take effect instantaneously. This is the highest form of consistency.

**Sequential:** Every client sees all changes in the same order they were applied.

**Causal:** All changes that are causally related are observed in the same order by all clients.

**Eventual:** When no updates occur for a period of time, eventually all updates will propagate through the system and all replicas will be consistent.

**Weak:** No guarantee is made that all updates will propagate and changes may appear out of order to various clients.
NoSQL Systems

Alternative to traditional relational DBMS

+ Flexible schema
+ Quicker/cheaper to set up
+ Massive scalability
+ Relaxed consistency $\rightarrow$ higher performance & availability

– No declarative query language $\rightarrow$ more programming
– Relaxed consistency $\rightarrow$ fewer guarantees
NoSQL Systems

Several incarnations

- MapReduce framework
- Key-value stores
  - Extensible record stores
- Document stores
- Graph database systems
MapReduce Framework

Schemas and declarative queries are missed

**Hive** – schemas, SQL-like query language

**Pig** – more imperative but with relational operators

- Both compile to “workflow” of Hadoop (MapReduce) jobs
Key-Value Stores

Extremely simple interface

- **Data model**: (key, value) pairs
- **Operations**: Insert(key, value), Fetch(key), Update(key), Delete(key)

Implementation: efficiency, scalability, fault-tolerance

- Records distributed to nodes based on key
- Replication
- Single-record transactions, “eventual consistency”
Key-Value Stores

Extremely simple interface

- **Data model**: (key, value) pairs
- **Operations**: Insert(key, value), Fetch(key), Update(key), Delete(key)
- Some allow (non-uniform) columns **within** value
  - Extensible record stores
- Some allow Fetch on range of keys

Example systems

- Google BigTable, Amazon Dynamo, Cassandra, Voldemort, HBase, ...
Document Stores

Like Key-Value Stores except value is document

- **Data model:** (key, document) pairs
- **Document:** JSON, XML, other semistructured formats
- **Basic operations:** Insert(key,document), Fetch(key), Update(key), Delete(key)
  - Also Fetch based on document contents

Example systems

- CouchDB, MongoDB, SimpleDB, ...
Why Key-value Store?

• (Business) Key -> Value
• (twitter.com) tweet id -> information about tweet
• (kayak.com) Flight number -> information about flight, e.g., availability
• (yourbank.com) Account number -> information about it
• (amazon.com) item number -> information about it

• Search is usually built on top of a key-value store
Isn’t that just a database?

- Yes
- Relational Databases (RDBMSs) have been around for ages
- MySQL is the most popular among them
- Data stored in tables
- Schema-based, i.e., structured tables
- Queried using SQL

SQL queries: SELECT user_id from users WHERE username = “jbellis”
Cassandra Data Model

- **Column Families:**
  - Like SQL tables
  - but may be unstructured (client-specified)
  - Can have index tables

- **Hence “column-oriented databases” / “NoSQL”**
  - No schemas
  - Some columns missing from some entries
  - “Not Only SQL”
  - Supports get(key) and put(key, value) operations
  - Often write-heavy workloads
Contents

• Intro, motivation, key definitions
• Overview
• Systems
  – Cassandra
  – HBase
• Applications
Early “Proof of Concepts”

- Memcached: demonstrated that in-memory indexes (DHT) can be highly scalable
- Dynamo: pioneered eventual consistency for higher availability and scalability
- BigTable: demonstrated that persistent record storage can be scaled to thousands of nodes
ACID v.s. BASE

- **ACID** = Atomicity, Consistency, Isolation, and Durability

- **BASE** = Basically Available, Soft state, Eventually consistent
Data Model

- **Tuple** = row in a relational db
- **Document** = nested values, extensible records (think XML or JSON)
- **Extensible record** = families of attributes have a schema, but new attributes may be added
- **Object** = like in a programming language, but without methods
1. Key-value Stores

Think “file system” more than “database”

- Persistence,
- Replication
- Versioning,
- Locking
- Transactions
- Sorting
1. Key-value Stores

- Voldemort, Riak, Redis, Scalaris, Tokyo Cabinet, Memcached/Membrain/Membase

- Consistent hashing (DHT)
- Only primary index: lookup by key
- No secondary indexes
- Transactions: single- or multi-update TXNs
  – locks, or MVCC
2. Document Stores

- A "document" = a pointerless object = e.g. JSON = nested or not = schema-less

- In addition to KV stores, may have secondary indexes
2. Document Stores

- SimpleDB, CouchDB, MongoDB, Terrastore

- Scalability:
  - Replication (e.g. SimpleDB, CouchDB – means entire db is replicated),
  - Sharding (MongoDB);
  - Both
3. Extensible Record Stores

- Typical Access: Row ID, Column ID, Timestamp

- Rows: sharding by primary key
  - BigTable: split table into *tablets* = units of distribution

- Columns: "column groups" = indication for which columns to be stored together (e.g. customer name/address group, financial info group, login info group)

- HBase, HyperTable, Cassandra, PNUT, BigTable
4. Scalable Relational Systems

• Means RDBS that are offering sharding

• Key difference: NoSQL make it difficult or impossible to perform large-scope operations and transactions (to ensure performance), while scalable RDBMS do not *preclude* these operations, but users pay a price only when they need them.

• MySQL Cluster, VoltDB, Clusterix, ScaleDB, Megastore (the new BigTable)
Contents

• Intro, motivation, key definitions
• Overview
• Systems
  – Cassandra
  – HBase
• Applications
The Dawn of NOSQL

There are several features that may be different from system to system:

- data model
- storage model
- consistency model
- physical model
- failure handling
- secondary indices
- compression
- load balancing
- atomic operations
- locking policy
Cassandra

- Originally designed at Facebook
- Open-sourced
- Some of its myriad users:
Cassandra

• “Apache Cassandra is an open-source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tunably consistent, column-oriented database that bases its distribution design on Amazon’s dynamo and its data model on Google’s Big Table.”

• Clearly, it is buzz-word compliant!!
Basic Idea: Key-Value Store

Table T:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>v1</td>
</tr>
<tr>
<td>k2</td>
<td>v2</td>
</tr>
<tr>
<td>k3</td>
<td>v3</td>
</tr>
<tr>
<td>k4</td>
<td>v4</td>
</tr>
</tbody>
</table>
Basic Idea: Key-Value Store

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>v1</td>
</tr>
<tr>
<td>k2</td>
<td>v2</td>
</tr>
<tr>
<td>k3</td>
<td>v3</td>
</tr>
<tr>
<td>k4</td>
<td>v4</td>
</tr>
</tbody>
</table>

Table T: keys are sorted

- **API:**
  - `lookup(key) → value`
  - `lookup(key range) → values`
  - `getNext → value`
  - `insert(key, value)`
  - `delete(key)`

- Each row has timestamp
- Single row actions atomic (but not persistent in some systems?)
- No multi-key transactions
- No query language!
Fragmentation (Sharding)

- use a partition vector
- “auto-sharding”: vector selected automatically
### Tablet Replication

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k7</td>
<td>v7</td>
</tr>
<tr>
<td>k8</td>
<td>v8</td>
</tr>
<tr>
<td>k9</td>
<td>v9</td>
</tr>
<tr>
<td>k10</td>
<td>v10</td>
</tr>
</tbody>
</table>

- **Cassandra:**
  - Replication Factor (# copies)
  - R/W Rule: One, Quorum, All
  - Policy (e.g., Rack Unaware, Rack Aware, ...)
  - Read all copies (return fastest reply, do repairs if necessary)

- **HBase:** Does not manage replication, relies on HDFS
Need a “directory”

• Table Name: Key → Server that stores key → Backup servers

• Can be implemented as a special table.
**Tablet Internals**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k3</td>
<td>v3</td>
</tr>
<tr>
<td>k8</td>
<td>v8</td>
</tr>
<tr>
<td>k9</td>
<td>delete</td>
</tr>
<tr>
<td>k15</td>
<td>v15</td>
</tr>
</tbody>
</table>

**memory**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k2</td>
<td>v2</td>
</tr>
<tr>
<td>k6</td>
<td>v6</td>
</tr>
<tr>
<td>k9</td>
<td>v9</td>
</tr>
<tr>
<td>k12</td>
<td>v12</td>
</tr>
</tbody>
</table>

**disk**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k4</td>
<td>v4</td>
</tr>
<tr>
<td>k5</td>
<td>delete</td>
</tr>
<tr>
<td>k10</td>
<td>v10</td>
</tr>
<tr>
<td>k20</td>
<td>v20</td>
</tr>
<tr>
<td>k22</td>
<td>v22</td>
</tr>
</tbody>
</table>

**Design Philosophy**: Primary scenario is where all data is in memory. Disk storage added as an afterthought.
Tablet Internals

tombstone

memory

flush periodically

disk

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k3</td>
<td>v3</td>
</tr>
<tr>
<td>k8</td>
<td>v8</td>
</tr>
<tr>
<td>k9</td>
<td>delete</td>
</tr>
<tr>
<td>k15</td>
<td>v15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k2</td>
<td>v2</td>
</tr>
<tr>
<td>k6</td>
<td>v6</td>
</tr>
<tr>
<td>k9</td>
<td>v9</td>
</tr>
<tr>
<td>k12</td>
<td>v12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k4</td>
<td>v4</td>
</tr>
<tr>
<td>k5</td>
<td>delete</td>
</tr>
<tr>
<td>k10</td>
<td>v10</td>
</tr>
<tr>
<td>k20</td>
<td>v20</td>
</tr>
<tr>
<td>k22</td>
<td>v22</td>
</tr>
</tbody>
</table>

- tablet is merge of all segments (files)
- disk segments immutable
- writes efficient; reads only efficient when all data in memory
- periodically reorganize into single segment
## Column Family

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
</tr>
<tr>
<td>k2</td>
<td>a2</td>
<td>null</td>
<td>c2</td>
<td>d2</td>
<td>e2</td>
</tr>
<tr>
<td>k3</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>d3</td>
<td>e3</td>
</tr>
<tr>
<td>k4</td>
<td>a4</td>
<td>b4</td>
<td>c4</td>
<td>e4</td>
<td>e4</td>
</tr>
<tr>
<td>k5</td>
<td>a5</td>
<td>b5</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
Column Family

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
</tr>
<tr>
<td>k2</td>
<td>a2</td>
<td>null</td>
<td>c2</td>
<td>d2</td>
<td>e2</td>
</tr>
<tr>
<td>k3</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>d3</td>
<td>e3</td>
</tr>
<tr>
<td>k4</td>
<td>a4</td>
<td>b4</td>
<td>c4</td>
<td>e4</td>
<td>e4</td>
</tr>
<tr>
<td>k5</td>
<td>a5</td>
<td>b5</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- for storage, treat each row as a single “super value”
- API provides access to sub-values (use family:qualifier to refer to sub-values e.g., price:euros, price:dollars )
- Cassandra allows “super-column”: two level nesting of columns (e.g., Column A can have sub-columns X & Y )
Vertical Partitions

can be manually implemented as
### Vertical Partitions

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
</tr>
<tr>
<td>k2</td>
<td>a2</td>
<td>null</td>
<td>c2</td>
<td>d2</td>
<td>e2</td>
</tr>
<tr>
<td>k3</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>d3</td>
<td>e3</td>
</tr>
<tr>
<td>k4</td>
<td>a4</td>
<td>b4</td>
<td>c4</td>
<td>e4</td>
<td>e4</td>
</tr>
<tr>
<td>k5</td>
<td>a5</td>
<td>b5</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

#### Column Family
- good for sparse data;
- good for column scans
- not so good for tuple reads
- API supports actions on full table; mapped to actions on column tables
- To decide on vertical partition, need to know access patterns
Failure Recovery (BigTable, HBase)

- Memory
- Tablet server
- Master node
- Spare tablet server
- Log
- GFS or HFS

Connections:
- Ping
- Write ahead logging
Failure recovery (Cassandra)

• No master node, all nodes in “cluster” equal

server 1

server 2

server 3
Failure recovery (Cassandra)

• No master node, all nodes in “cluster” equal

access any table in cluster at any server

server 1

server 2

server 3

that server sends requests to other servers
Cassandra Vs. SQL

- **MySQL** is the most popular (and has been for a while)
- On > 50 GB data
- **MySQL**
  - Writes 300 ms avg
  - Reads 350 ms avg
- **Cassandra**
  - Writes 0.12 ms avg
  - Reads 15 ms avg
Cassandra Summary

• While RDBMS provide ACID (Atomicity Consistency Isolation Durability)
• Cassandra provides BASE
  – Basically Available Soft-state Eventual Consistency
  – Prefers Availability over consistency
• Other NoSQL products
  – MongoDB, Riak (look them up!)
• Next: HBase
  – Prefers (strong) Consistency over Availability
Brewer's CAP Theorem

Brewer's CAP theorem states that a distributed system is not possible to guarantee all three of the following properties simultaneously:

- **Consistency**: all nodes see the same data at the same time
- **Availability**: a guarantee that every request receives a response about whether it succeeded or failed
- **Partition Tolerance** (the system continues to operate despite arbitrary message loss or failure of part of the system)
HBase

• Google’s BigTable was first “blob-based” storage system
• Yahoo! Open-sourced it → HBase
• Major Apache project today
• Facebook uses HBase internally
• API
  – Get/Put(row)
  – Scan(row range, filter) – range queries
  – MultiPut
HBase Storage hierarchy

• HBase Table
  – Split it into multiple regions: replicated across servers
    • One Store per ColumnFamily (subset of columns with similar query patterns) per region

• HFile
  – SSTable from Google’s BigTable
The master is responsible for **assigning regions** to RegionServers.

**API**

RegionServers are responsible for all read and write requests for all regions they serve, and also split regions that have exceeded the configured region size thresholds.

**HFile**

Low-level storage file to store column families.

**Memstore**

In memory storage.

**Logs**

In case of updates data are first written to a **write-ahead log**.

**Hadoop Distributed File System (HDFS)**

The Hadoop Distributed File System.

**ZooKeeper**

A reliable, highly available, persistent and distributed **coordination** service.
Contents

• Intro, motivation, key definitions
• Overview
• Systems
  – Cassandra
  – HBase
• Applications
Application 1

- Web application that needs to display lots of customer information; the users data is rarely updated, and when it is, you know when it changes because updates go through the same interface. Store this information persistently using a KV store.
Application 2

- Department of Motor Vehicle: lookup objects by multiple fields (driver's name, license number, birth date, etc); "eventual consistency" is ok, since updates are usually performed at a single location.

Document Store
Application 3

- eBay style application. Cluster customers by country; separate the rarely changed "core" customer information (address, email) from frequently-updated info (current bids).

Extensible Record Store
Application 4

- Everything else (e.g. a serious DMV application)

Scalable RDBMS
HBase
Create a table named test with a single column family named cf. Verify its creation by listing all tables and then insert some values.

```
hbase(main):003:0> create 'test', 'cf'
0 row(s) in 1.2200 seconds
hbase(main):003:0> list 'test'
...
1 row(s) in 0.0550 seconds
hbase(main):004:0> put 'test', 'row1', 'cf:a', 'value1'
0 row(s) in 0.0560 seconds
hbase(main):005:0> put 'test', 'row2', 'cf:b', 'value2'
0 row(s) in 0.0370 seconds
hbase(main):006:0> put 'test', 'row3', 'cf:c', 'value3'
0 row(s) in 0.0450 seconds
```

Above we inserted 3 values, one at a time. The first insert is at row1, column cf:a with a value of value1. Columns in HBase are case sensitive.

Verify the data insert by running a scan of the table as follows:

```
hbase(main):007:0> scan 'test'
ROW COLUMN+CELL
row1 column=cf:a, timestamp=1288380727188, value=value1
row2 column=cf:b, timestamp=1288380738440, value=value2
row3 column=cf:c, timestamp=1288380747365, value=value3
3 row(s) in 0.0590 seconds
```

Get a single row:

```
hbase(main):008:0> get 'test', 'row1'
COLUMN CELL
cf:a timestamp=1288380727188, value=value1
1 row(s) in 0.0400 seconds
```
import org.apache.hadoop.hbase.util.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.util.Bytes;
import java.io.IOException;

public class TestHBase {

    public static void main(String[] arg) throws IOException {
        Configuration config = HBaseConfiguration.create();

    }
}
//read values of cf:a
byte[] family = Bytes.toBytes("cf");
byte[] qual = Bytes.toBytes("a");

HTable testTable = new HTable(config, "test");

Scan scan = new Scan();
scan.addColumn(family, qual);
ResultScanner rs = testTable.getScanner(scan);
for (Result r = rs.next(); r != null; r = rs.next()) {
    byte[] valueObj = r.getValue(family, qual);
    String value = new String(valueObj);
    System.out.println(value);
}
//add a row with key “newtest”
// and value of cf:a “new-value”

Put put = new Put(Bytes.toBytes("newtest"));
put.add(Bytes.toBytes("cf"), Bytes.toBytes("a"), Bytes.toBytes("new-value"));
testTable.put(put);