Scalable Data Management
NoSQL Data Stores in Research and Practice

Felix Gessert, Norbert Ritter
{gessert, ritter}@informatik.uni-hamburg.de
May 17, ICDE 2016
Extended version of this tutorial: slideshare.net/felixgessert
Outline

NoSQL Foundations and Motivation

The NoSQL Toolbox: Common Techniques

NoSQL Systems

Decision Guidance: NoSQL Decision Tree

• The Database Explosion
• NoSQL: Motivation and Origins
• The 4 Classes of NoSQL Databases:
  • Key-Value Stores
  • Wide-Column Stores
  • Document Stores
  • Graph Databases
• CAP Theorem
Introduction: What are NoSQL data stores?
Architecture

Typical Data Architecture:

The era of **one-size-fits-all** database systems is over

→ **Specialized** data systems
The Database Explosion

Sweetspots

- **IBM DB2**
  - RDBMS
  - General-purpose ACID transactions

- **Greenplum**
  - Parallel DWH
  - Aggregations/OLAP for massive data amounts

- **VoltDB**
  - NewSQL
  - High throughput relational OLTP

- **HBase**
  - Wide-Column Store
  - Long scans over structured data

- **MongoDB**
  - Document Store
  - Deeply nested data models

- **Neo4j**
  - Graph Database
  - Graph algorithms & queries

- **Redis**
  - In-Memory KV-Store
  - Counting & statistics

- **Cassandra**
  - Wide-Column Store
  - Massive user-generated content
The Database Explosion
Cloud-Database Sweetspots

**Realtime BaaS**
Communication and collaboration

**Azure Tables**
Wide-Column Store
Very large tables

**Firebase**
Managed NoSQL
Full-Text Search

**Amazon RDS**
Managed RDBMS
General-purpose ACID transactions

**Amazon DynamoDB**
Wide-Column Store
Massive user-generated content

**Google Cloud Storage**
Object Store
Massive File Storage

**ElastiCache**
Managed Cache
Caching and transient storage

**Parse**
Backend-as-a-Service
Small Websites and Apps

**Amazon Elastic MapReduce**
Hadoop-as-a-Service
Big Data Analytics
How to choose a database system?

Many Potential Candidates

Question in this tutorial:

How to approach the decision problem?
NoSQL Databases

- „NoSQL“ term coined in 2009
- Interpretation: „Not Only SQL“
- Typical properties:
  - Non-relational
  - Open-Source
  - Schema-less (schema-free)
  - Optimized for distribution (clusters)
  - Tunable consistency

NoSQL-Databases.org: Current list has over 150 NoSQL systems
NoSQL Databases

Two main motivations:

Scalability

- User-generated data, Request load

Impedance Mismatch

- ID
- Customer
- Line Item 1: ...
- Line Item 2: ...
- Payment: Credit Card, ...

Orders

Line Items

Payment

Customers
Scale-up vs Scale-out

**Scale-Up** (*vertical scaling):*
- More RAM
- More CPU
- More HDD

**Scale-Out** (*horizontal scaling):*
- Commodity Hardware
- Shared-Nothing Architecture
Schemafree Data Modeling

RDBMS:

```
SELECT Name, Age
FROM   Customers
```

NoSQL DB:

```
Item[Price] - Item[Discount]
```

Explicit schema

Implicit schema
Highly Available Storage (SAN, RAID, etc.)

Highly available network (Infiniband, Fabric Path, etc.)

Specialized DB hardware (Oracle Exadata, etc.)

Commercial DBMS

Commercial drives (standard HDDs, JBOD)

Commodity network (Ethernet, etc.)

Commodity hardware

Open-Source DBMS

Open Source & Commodity Hardware
NoSQL System Classification

- Two common criteria:
  - *Consistency/Availability Trade-Off*
  - **AP**: Available & Partition Tolerant
  - **CP**: Consistent & Partition Tolerant
  - **CA**: Not Partition Tolerant

**Data Model**
- Key-Value
- Wide-Column
- Document
- Graph
Key-Value Stores

- **Data model**: (key) -> value
- **Interface**: CRUD (Create, Read, Update, Delete)

Examples: Amazon Dynamo (AP), Riak (AP), Redis (CP)
Wide-Column Stores

- **Data model:** (rowkey, column, timestamp) -> value
- **Interface:** CRUD, Scan

Examples: Cassandra (AP), Google BigTable (CP), HBase (CP)
Document Stores

- **Data model**: (collection, key) -> document
- **Interface**: CRUD, Querys, Map-Reduce

Examples: CouchDB (AP), Amazon SimpleDB (AP), MongoDB (CP)
Graph Databases

- **Data model**: $G = (V, E)$: Graph-Property Modell
- **Interface**: Traversal, edges, properties, transactions
- **Examples**: Neo4j (CA), InfiniteGraph (CA), OrientDB (CA)

**Nodes**
- company: Apple
- value: 300Mrd

**Properties**
- name: John Doe

Note: Usually unscalable (optimal partitioning is NP-complete)
Soft NoSQL Systems

Not Covered Here

Search Platforms (Full Text Search):
- No persistence and consistency guarantees for OLTP
  - Examples: ElasticSearch (AP), Solr (AP)

Object-Oriented Databases:
- Strong coupling of programming language and DB
  - Examples: Versant (CA), db4o (CA), Objectivity (CA)

XML-Databases, RDF-Stores:
- Not scalable, data models not widely used in industry
  - Examples: MarkLogic (CA), AllegroGraph (CA)
CAP-Theorem

Only 2 out of 3 properties are achievable at a time:

- **Consistency**: all clients have the same view on the data
- **Availability**: every request to a non-failed node most result in correct response
- **Partition tolerance**: the system has to continue working, even under arbitrary network partitions

---

**Eric Brewer, ACM-PODC Keynote, Juli 2000**

**Gilbert, Lynch: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services, SigAct News 2002**
**Problem**: when a network partition occurs, either consistency or availability have to be given up.

**CAP-Theorem: simplified proof**

- **Value** = \( V_0 \)  
- **Value** = \( V_1 \)

- **Response before successful replication** → **Availability**
- **Block response until ACK arrives** → **Consistency**

**Network partition**
NoSQL Triangle

Every client can always read and write

C
Oracle, MySQL, ...

All clients share the same view on the data

A

Data models

Relational
Key-Value
Wide-Column
Document-Oriented

All nodes continue working under network partitions

P

CP
Postgres, MySQL Cluster, Oracle RAC
BigTable, HBase, Accumulo, Azure Tables
MongoDB, RethinkDB

AP
Dynamo, Redis, Riak, Voldemort
Cassandra
SimpleDB,
Idea: Classify systems according to their behavior during *network partitions*

PACELC – an alternative CAP formulation

- **Avail-ability** (AL) - Dynamo-Style
- **Consistency** (AC) - MongoDB
- **Consistency** (CC) - Always Consistent

*No consequence of the CAP theorem*
Serializability
Not Highly Available Either

Global serializability and availability are incompatible:

- Write A=1, Read B
- Write B=1, Read A

\[ w_1(a = 1) \land r_1(b = \bot) \]
\[ w_2(b = 1) \land r_2(a = \bot) \]

Some weaker isolation levels allow high availability:


Where CAP fits in
Negative Results in Distributed Computing

**Asynchronous Network, Unreliable Channel**

Atomic Storage

- **Impossible**: CAP Theorem

Consensus

- **Impossible**: 2 Generals Problem

**Asynchronous Network, Reliable Channel**

Atomic Storage

- **Possible**: Attiya, Bar-Noy, Dolev (ABD) Algorithm

Consensus

- **Impossible**: Fisher Lynch Patterson (FLP) Theorem

ACID vs BASE

ACID

- Atomicity
- Consistency
- Isolation
- Durability

"Gold standard" for RDBMSs

BASE

- Basically Available
- Soft State
- Eventually Consistent

Model of many NoSQL systems

http://queue.acm.org/detail.cfm?id=1394128
Data Models and CAP provide high-level classification.

But what about fine-grained requirements, e.g. query capabilities?
Outline

NoSQL Foundations and Motivation

The NoSQL Toolbox: Common Techniques

NoSQL Systems

Decision Guidance: NoSQL Decision Tree

• Techniques for Functional and Non-functional Requirements
  • Sharding
  • Replication
  • Storage Management
  • Query Processing
Functional Techniques

- ACID Transactions
- Conditional or Atomic Writes
- Joins
- Sorting
- Filter Queries
- Full-text Search
- Aggregation and Analytics

Non-Functional

- Sharding
- Range-Sharding
- Hash-Sharding
- Entity-Group Sharding
- Consistent Hashing
- Shared-Disk

- Replication
- Consistent Replication
- Synchronous Replication
- Asynchronous Replication
- Primary Copy
- Update Anywhere

- Storage Management
- Logging
- Update-In-Place
- Caching
- In-Memory Storage
- Append-Only Storage

- Query Processing
- Elasticity
- Consistency
- Read Latency
- Write Throughput
- Read Availability
- Write Availability
- Durability

Operational Requirements

- Central techniques NoSQL databases employ

Enable

Requirements from the application

Enable

Enable

Operational Requirements
NoSQL Database Systems:
A Survey and Decision Guidance

Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Rüter
Universität Hamburg, Germany
{gessert, wingerath, friedrich, ritter}@informatik.uni-hamburg.de

Abstract. Today, data is generated and consumed at unprecedented scale. This has lead to novel approaches for scalable data management subsumed under the term “NoSQL” database systems to handle the ever-increasing data volume and request loads. However, the heterogeneity and diversity of the numerous existing systems impede the well-informed selection of a data store appropriate for a given application context. Therefore, this article gives a top-down overview of the field: Instead of contrasting the implementation specifics of individual representatives, we propose a comparative classification model that relates functional and non-functional requirements to techniques and algorithms employed in NoSQL databases. This NoSQL Toolbox allows us to derive a simple decision tree to help practitioners and researchers filter potential system candidates based on central application requirements.

1 Introduction

Traditional relational database management systems (RDBMSs) provide powerful mechanisms to store and query structured data under strong consistency and transaction guarantees and have reached an unmatched level of reliability, stability and support through decades of development. In recent years, however, the amount of useful data in some application areas has become so vast that it cannot be stored or processed by traditional database solutions. User-generated content in social networks or data retrieved from large sensor networks are only two examples of this phenomenon commonly referred to as Big Data [35]. A class of novel data storage systems able to cope with Big Data are subsumed under the term NoSQL databases, many of which offer horizontal scalability and higher availability than relational databases by sacrificing querying capabilities and consistency guarantees. These trade-offs are pivotal for service-oriented computing and as-a-service models, since any stateful service can only be as scalable and fault-tolerant as its underlying data store.

There are dozens of NoSQL database systems and it is hard to keep track of where they excel, where they fail or even where they differ, as implementation details change quickly and feature sets evolve over time. In this article, we therefore aim to provide an overview of the NoSQL landscape by discussing employed concepts rather than system specificities and explore the requirements typically posed to NoSQL database systems, the techniques used to fulfill these requirements and the trade-offs that have to be made in the process. Our focus lies on key-value, document and wide-column stores, since these NoSQL categories
Functional Techniques
- Scan Queries
- ACID Transactions
- Conditional or Atomic Writes
- Joins
- Sorting

Non-Functional
- Data Scalability
- Write Scalability
- Read Scalability
- Elasticity

Techniques
- Sharding
  - Range-Sharding
  - Hash-Sharding
  - Entity-Group Sharding
  - Consistent Hashing
  - Shared-Disk
Sharding (aka Partitioning, Fragmentation)

- Horizontal distribution of data over nodes

- **Partitioning strategies**: Hash-based vs. Range-based
- **Difficulty**: Multi-Shard-Operations (join, aggregation)
Sharding

Hash-based Sharding
- Hash of data values (e.g. key) determines partition
  - **Pro:** Even distribution
  - **Contra:** No data locality

Range-based Sharding
- Assigns ranges defined over fields (shard keys)
  - **Pro:** Enables *Range Scans* and *Sorting*
  - **Contra:** Repartitioning/balancing required

Entity-Group Sharding
- Explicit data co-location for single-node transactions
  - **Pro:** Enables *ACID Transactions*
  - **Contra:** Partitioning not easily changeable

Implemented in
- MongoDB, Riak, Redis, Cassandra, Azure Table, Dynamo
- BigTable, HBase, DocumentDB, Hypertable, MongoDB, RethinkDB, Espresso
- G-Store, MegaStore, Relation Cloud, Cloud SQL Server

---

Problems of Application-Level Sharding

Example: **Tumblr**
- Caching
- Sharding from application

Moved towards:
- Redis
- HBase
Functional Techniques Non-Functional

ACID Transactions
Conditional or Atomic Writes
Replication
  Commit/Consensus Protocol
  Synchronous
  Asynchronous
  Primary Copy
  Update Anywhere

Read Scalability
Consistency
Write Latency
Read Latency
Read Availability
Write Availability
Replication

- Stores \( N \) copies of each data item

- Consistency model: synchronous vs asynchronous
- Coordination: Multi-Master, Master-Slave

Replication: When

Asynchronous (lazy)
- Writes are acknowledged immediately
- Performed through log shipping
- **Pro**: Fast writes, no coordination needed
- **Contra**: Replica data potentially stale (inconsistent)

Synchronous (eager)
- The node accepting writes synchronously propagates updates/transactions before acknowledging
- **Pro**: Consistent
- **Contra**: Needs a commit protocol (more roundtrips), unavailable under certain network partitions

Implemented in
- Dynamo, Riak, CouchDB, Redis, Cassandra, Voldemort, MongoDB, RethinkDB
- BigTable, HBase, Accumulo, CouchBase, MongoDB, RethinkDB

Replication: Where

Master-Slave (*Primary Copy*)
- Only a dedicated master is allowed to accept writes, slaves are read-replicas
- **Pro**: reads from the master are consistent
- **Contra**: master is a bottleneck and SPOF

Multi-Master (*Update anywhere*)
- The server node accepting the writes synchronously propagates the update or transaction before acknowledging
- **Pro**: fast and highly-available
- **Contra**: either needs coordination protocols (e.g. Paxos) or is inconsistent

Consistency Levels

Achievable with high availability

Causal Consistency

If a value is read, any causally relevant data items that lead to that value are available, too.

Write Follow Reads

Read Your Writes

Monotonic Reads

Monotonic Writes

Bounded Staleness

Strategies:
- Single-mastered reads and writes
- Multi-master replication with consensus on writes

Version-based or time-based. Both not highly available.

Version-based or time-based. Both not highly available.

writes

increased

monotonically.

increased

monotonically.

Reads

Monotonic

Causal Consistency

 Writes Follow Reads

Read Your Writes

Monotonic Reads

Monotonic Writes

Bounded Staleness


Storage Management
- Logging
- Update-in-Place
- Caching
- In-Memory Storage
- Append-Only Storage

- Read Latency
- Write Throughput
- Durability
NoSQL Storage Management
In a Nutshell

Typical Uses in DBMSs:
- Caching
- Primary Storage
- Data Structures

RAM

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

SSD

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

HDD

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

Size

Volatile

Durable

Speed, Cost

High Performance

Low Performance

- Promotes durability of write operations.
- Increases write throughput.
- Is good for read latency.
- Improve latency.

Persistent Storage

Log

Data

In-Memory/Caching

Update-In-Place

Append-Only I/O

Logging

Data In-Memory/
Caching

Logging

Append-Only
I/O

Update-In-
Place

In-Memory/
Caching

Typical Uses in DBMSs:
- Caching
- Primary Storage
- Data Structures

RAM

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

SSD

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

HDD

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

Size

Volatile

Durable

Speed, Cost

High Performance

Low Performance

- Promotes durability of write operations.
- Increases write throughput.
- Is good for read latency.
- Improve latency.

Persistent Storage

Log

Data

In-Memory/Caching

Update-In-Place

Append-Only I/O

Logging

Data In-Memory/
Caching

Logging

Append-Only
I/O

Update-In-
Place

In-Memory/
Caching

Promotes durability of write operations.

Typical Uses in DBMSs:
- Caching
- Primary Storage
- Data Structures

RAM

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

SSD

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

HDD

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

Size

Volatile

Durable

Speed, Cost

High Performance

Low Performance

- Promotes durability of write operations.
- Increases write throughput.
- Is good for read latency.
- Improve latency.

Persistent Storage

Log

Data

In-Memory/Caching

Update-In-Place

Append-Only I/O

Logging

Data In-Memory/
Caching

Logging

Append-Only
I/O

Update-In-
Place

In-Memory/
Caching

Promotes durability of write operations.

Typical Uses in DBMSs:
- Caching
- Primary Storage
- Data Structures

RAM

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

SSD

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

HDD

- RR: Random Reads
- SR: Sequential Reads
- RW: Random Writes
- SW: Sequential Writes

Size

Volatile

Durable

Speed, Cost

High Performance

Low Performance

- Promotes durability of write operations.
- Increases write throughput.
- Is good for read latency.
- Improve latency.

Persistent Storage

Log

Data

In-Memory/Caching

Update-In-Place

Append-Only I/O

Logging

Data In-Memory/
Caching

Logging

Append-Only
I/O

Update-In-
Place

In-Memory/
Caching

Promotes durability of write operations.
Functional Techniques

- Joins
- Sorting
- Filter Queries
- Full-text Search
- Aggregation and Analytics

Non-Functional

- Query Processing
  - Global Secondary Indexing
  - Local Secondary Indexing
  - Query Planning
  - Analytics Framework
  - Materialized Views

- Read Latency
Local Secondary Indexing
Partitioning By Document

<table>
<thead>
<tr>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Red</td>
</tr>
<tr>
<td>56</td>
<td>Blue</td>
</tr>
<tr>
<td>77</td>
<td>Red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>[12,77]</td>
</tr>
<tr>
<td>Blue</td>
<td>[56]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>Yellow</td>
</tr>
<tr>
<td>188</td>
<td>Blue</td>
</tr>
<tr>
<td>192</td>
<td>Blue</td>
</tr>
</tbody>
</table>

Implemented in

- MongoDB
- Riak
- Cassandra
- Elasticsearch
- SolrCloud
- VoltDB

Scatter-gather query pattern.

WHERE color=blue

Global Secondary Indexing
Partitioning By Term

<table>
<thead>
<tr>
<th>Data</th>
<th>Index</th>
<th>Partition I</th>
<th>Partition II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>Color</td>
<td>12</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td></td>
<td>56</td>
<td>Yellow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>77</td>
<td>Blue</td>
</tr>
<tr>
<td>Term</td>
<td>Match</td>
<td>Yellow</td>
<td>[12, 77]</td>
</tr>
<tr>
<td>Yellow</td>
<td>[56, 188, 192]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blue</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Implemented in**
- DynamoDB
- Oracle Datawarehouse
- Riak (Search)
- Cassandra (Search)

Consistent index maintenance requires distributed transaction.

Targeted Query
WHERE color=blue

Query Processing Techniques

Summary

- **Local Secondary Indexing**: Fast writes, scatter-gather queries
- **Global Secondary Indexing**: Slow or inconsistent writes, fast queries
- **(Distributed) Query Planning**: scarce in NoSQL systems but increasing (e.g. left-outer equi-joins in MongoDB and θ-joins in RethinkDB)
- **Analytics Frameworks**: fallback for missing query capabilities
- **Materialized Views**: similar to global indexing
How are the techniques from the NoSQL toolbox used in actual data stores?
Outline

- **NoSQL Foundations and Motivation**

- **The NoSQL Toolbox: Common Techniques**

- **NoSQL Systems**

- **Decision Guidance: NoSQL Decision Tree**

- **Overview & Popularity**
- **Core Systems:**
  - Dynamo
  - BigTable
  - Riak
  - HBase
  - Cassandra
  - Redis
  - MongoDB
## Popularity

<table>
<thead>
<tr>
<th>#</th>
<th>System</th>
<th>Model</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Oracle</td>
<td>Relational DBMS</td>
<td>1462.02</td>
</tr>
<tr>
<td>2.</td>
<td>MySQL</td>
<td>Relational DBMS</td>
<td>1371.83</td>
</tr>
<tr>
<td>3.</td>
<td>MS SQL Server</td>
<td>Relational DBMS</td>
<td>1142.82</td>
</tr>
<tr>
<td>4.</td>
<td>MongoDB</td>
<td>Document store</td>
<td>320.22</td>
</tr>
<tr>
<td>5.</td>
<td>PostgreSQL</td>
<td>Relational DBMS</td>
<td>307.61</td>
</tr>
<tr>
<td>6.</td>
<td>DB2</td>
<td>Relational DBMS</td>
<td>185.96</td>
</tr>
<tr>
<td>7.</td>
<td>Cassandra</td>
<td>Wide column store</td>
<td>134.50</td>
</tr>
<tr>
<td>8.</td>
<td>Microsoft Access</td>
<td>Relational DBMS</td>
<td>131.58</td>
</tr>
<tr>
<td>9.</td>
<td>Redis</td>
<td>Key-value store</td>
<td>108.24</td>
</tr>
<tr>
<td>10.</td>
<td>SQLite</td>
<td>Relational DBMS</td>
<td>107.26</td>
</tr>
<tr>
<td>11.</td>
<td>Elasticsearch</td>
<td>Search engine</td>
<td>86.31</td>
</tr>
<tr>
<td>12.</td>
<td>Teradata</td>
<td>Relational DBMS</td>
<td>73.74</td>
</tr>
<tr>
<td>13.</td>
<td>SAP Adaptive Server</td>
<td>Relational DBMS</td>
<td>71.48</td>
</tr>
<tr>
<td>14.</td>
<td>Solr</td>
<td>Search engine</td>
<td>65.62</td>
</tr>
<tr>
<td>15.</td>
<td>HBase</td>
<td>Wide column store</td>
<td>51.84</td>
</tr>
<tr>
<td>16.</td>
<td>Hive</td>
<td>Relational DBMS</td>
<td>47.51</td>
</tr>
<tr>
<td>17.</td>
<td>FileMaker</td>
<td>Relational DBMS</td>
<td>46.71</td>
</tr>
<tr>
<td>18.</td>
<td>Splunk</td>
<td>Search engine</td>
<td>44.31</td>
</tr>
<tr>
<td>19.</td>
<td>SAP HANA</td>
<td>Relational DBMS</td>
<td>41.37</td>
</tr>
<tr>
<td>20.</td>
<td>MariaDB</td>
<td>Relational DBMS</td>
<td>33.97</td>
</tr>
<tr>
<td>21.</td>
<td>Neo4j</td>
<td>Graph DBMS</td>
<td>32.61</td>
</tr>
<tr>
<td>22.</td>
<td>Informix</td>
<td>Relational DBMS</td>
<td>30.58</td>
</tr>
<tr>
<td>23.</td>
<td>Memcached</td>
<td>Key-value store</td>
<td>27.90</td>
</tr>
<tr>
<td>24.</td>
<td>Couchbase</td>
<td>Document store</td>
<td>24.29</td>
</tr>
<tr>
<td>25.</td>
<td>Amazon DynamoDB</td>
<td>Multi-model</td>
<td>23.60</td>
</tr>
</tbody>
</table>

### Scoring: Google/Bing results, Google Trends, Stackoverflow, job offers, LinkedIn
NoSQL foundations

- **BigTable** (2006, Google)
  - Consistent, Partition Tolerant
  - Wide-Column data model
  - Master-based, fault-tolerant, large clusters (1,000+ Nodes), HBase, Cassandra, HyperTable, Accumulo

- **Dynamo** (2007, Amazon)
  - Available, Partition tolerant
  - Key-Value interface
  - Eventually Consistent, always writable, fault-tolerant
  - Riak, Cassandra, Voldemort, DynamoDB

References:

- DeCandia, Giuseppe, et al. "Dynamo: Amazon’s highly available key-value store."
Dynamo (AP)

- Developed at Amazon (2007)
- Sharding of data over a ring of nodes
- Each node holds multiple partitions
- Each partition replicated \( N \) times

DeCandia, Giuseppe, et al. "Dynamo: Amazon's highly available key-value store."
Consistent Hashing

- Naive approach: **Hash-partitioning** (e.g. in Memcache, Redis Cluster)

\[ \text{partition} = \text{hash(key)} \% \text{server\_count} \]
Consistent Hashing

Solution: **Consistent Hashing** – mapping of data to nodes is stable under topology changes

Consistent Hashing:

\[
\text{position} = \text{hash(ip)}
\]

\[
\text{hash(key)} = \text{hash}(\text{key})
\]

\[
2^{160} \quad 0
\]
Reading and Writing

- An arbitrary node acts as a coordinator
- $N$: number of replicas
- $R$: number of nodes that need to confirm a read
- $W$: number of nodes that need to confirm a write

$N=3$
$R=2$
$W=1$
Versioning and Consistency

- $R + W \leq N \Rightarrow$ no consistency guarantee
- $R + W > N \Rightarrow$ newest acked value included in reads
- **Vector Clocks** used for versioning

Read Repair

Semantic Reconciliation
$R + W > N$ does not imply linearizability

- Consider the following execution:
CRDTs
Convergent/Commutative Replicated Data Types

- **Goal**: avoid manual conflict-resolution
- **Approach**:
  - **State-based** – commutative, idempotent merge function
  - **Operation-based** – broadcasts of commutative updates
- **Example**: State-based Grow-only-Set (G-Set)

\[
\begin{align*}
S_1 &= \{\} \\
S_1 &= \{x\} \\
S_1 &= \text{merge}(\{x\}, \{y\}) \\
&= \{x, y\}
\end{align*}
\]

\[
\begin{align*}
S_2 &= \{\} \\
S_2 &= \{y\} \\
S_2 &= \text{merge}(\{y\}, \{x\}) \\
&= \{x, y\}
\end{align*}
\]

Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski "Conflict-free Replicated Data Types"
Riak (AP)

- Open-Source Dynamo-Implementation
- Extends Dynamo:
  - Keys are grouped to **Buckets**
  - KV-pairs may have **metadata** and **links**
  - Map-Reduce support
  - Secondary Indices, Update Hooks, Solr Integration
  - **Riak CS**: S3-like file storage, **Riak TS**: time-series database

### Riak Details

- **Model**: Key-Value
- **License**: Apache 2
- **Written in**: Erlang und C
- **Consistency Level**: N, R, W, DW
- **Storage Backend**: Bit-Cask, Memory, LevelDB
Summary: Dynamo and Riak

- Available and Partition-Tolerant
- **Consistent Hashing**: hash-based distribution with stability under topology changes (e.g. machine failures)
- Parameters: N (Replicas), R (Read Acks), W (Write Acks)
  - N=3, R=W=1 $\rightarrow$ fast, potentially inconsistent
  - N=3, R=3, W=1 $\rightarrow$ slower reads, most recent object version contained
- **Vector Clocks**: concurrent modification can be detected, inconsistencies are healed by the application
- **API**: Create, Read, Update, Delete (CRUD) on key-value pairs
- **Riak**: Open-Source Implementation of the Dynamo paper
## Dynamo and Riak

### Classification

<table>
<thead>
<tr>
<th>Sharding</th>
<th>Range-Sharding</th>
<th>Hash-Sharding</th>
<th>Entity-Group Sharding</th>
<th>Consistent Hashing</th>
<th>Shared Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replication</td>
<td>Transaction Protocol</td>
<td>Sync. Replication</td>
<td>Async. Replication</td>
<td>Primary Copy</td>
<td>Update Anywhere</td>
</tr>
<tr>
<td>Storage Management</td>
<td>Logging</td>
<td>Update-in-Place</td>
<td>Caching</td>
<td>In-Memory</td>
<td>Append-Only Storage</td>
</tr>
<tr>
<td>Query Processing</td>
<td>Global Index</td>
<td>Local Index</td>
<td>Query Planning</td>
<td>Analytics</td>
<td>Materialized Views</td>
</tr>
</tbody>
</table>
Redis (CA)

- **Remote Dictionary Server**
- In-Memory Key-Value Store
- Asynchronous Master-Slave Replication
- Data model: rich data structures stored under key
- **Tunable persistence**: logging and snapshots
- Single-threaded event-loop design (similar to Node.js)
- Optimistic **batch transactions** *(Multi blocks)*
- Very high performance: >100k ops/sec per node
- Redis Cluster adds sharding
# Data structures

- **String**: List, Set, Hash, Sorted Set

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>web:index</td>
<td>&quot;&lt;html&gt;&lt;head&gt;...&quot;</td>
</tr>
<tr>
<td>Set</td>
<td>users:2:friends</td>
<td>{23, 76, 233, 11}</td>
</tr>
<tr>
<td>List</td>
<td>users:2:inbox</td>
<td>[234, 3466, 86, 55]</td>
</tr>
<tr>
<td>Hash</td>
<td>users:2:settings</td>
<td>Theme → &quot;dark&quot;, cookies → &quot;false&quot;</td>
</tr>
<tr>
<td>Sorted Set</td>
<td>top-posters</td>
<td>466 → &quot;2&quot;, 344 → &quot;16&quot;</td>
</tr>
<tr>
<td>Pub/Sub</td>
<td>users:2:notifs</td>
<td>&quot;{event: 'comment posted', time : ...}&quot;</td>
</tr>
</tbody>
</table>
Example Redis Data Structure: lists

- **(Linked) Lists:**

  - **LPUSHX** (Only if list exists)
  - **LPUSH**
  - **LRANGE** inbox 1 2
  - **RPUSH**

  - **inbox**
  - 234
  - 34 6
  - 86
  - 55

  - **LLEN**
  - **LREM** inbox 0 3466

  - **LPOP**
  - **LRANGE** inbox 1 2
  - **LINDEX** inbox 2
  - **RPOP**

  - **BLPOP**

  - Blocks until element arrives

  - 4
Master-Slave Replication

> SLAVEOF 192.168.1.1 6379
< +OK

writes

Asynchronous Replication

Memory Backlog

Slave Offsets

Stream
Why is Redis so fast?

- Pessimistic transactions are expensive
- Data in RAM
- Operations are lock-free
- Single-threading
- No Query Parsing
- AOF
- Hand-coded optimizations

Harizopoulos, Stavros, Madden, Stonebraker "OLTP through the looking glass, and what we found there."
Example Redis Use-Case: Twitter

- Per User: one materialized timeline in Redis
- Timeline = List
- Key: User ID

>150 million users
~300k timeline queries/s

http://www.infoq.com/presentations/Real-Time-Delivery-Twitter
## Classification: Redis Techniques

<table>
<thead>
<tr>
<th>Classification</th>
<th>Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sharding</strong></td>
<td>Range-Sharding</td>
</tr>
<tr>
<td></td>
<td>Hash-Sharding</td>
</tr>
<tr>
<td></td>
<td>Entity-Group Shardng</td>
</tr>
<tr>
<td></td>
<td>Consistent Hashing</td>
</tr>
<tr>
<td></td>
<td>Shared Disk</td>
</tr>
<tr>
<td><strong>Replication</strong></td>
<td>Transaction Protocol</td>
</tr>
<tr>
<td></td>
<td>Sync. Replication</td>
</tr>
<tr>
<td></td>
<td>Async. Replication</td>
</tr>
<tr>
<td></td>
<td>Primary Copy</td>
</tr>
<tr>
<td></td>
<td>Update Anywhere</td>
</tr>
<tr>
<td><strong>Storage Management</strong></td>
<td>Logging</td>
</tr>
<tr>
<td></td>
<td>Update-in-Place</td>
</tr>
<tr>
<td></td>
<td>Caching</td>
</tr>
<tr>
<td></td>
<td>In-Memory</td>
</tr>
<tr>
<td></td>
<td>Append-Only Storage</td>
</tr>
<tr>
<td><strong>Query Processing</strong></td>
<td>Global Index</td>
</tr>
<tr>
<td></td>
<td>Local Index</td>
</tr>
<tr>
<td></td>
<td>Query Planning</td>
</tr>
<tr>
<td></td>
<td>Analytics</td>
</tr>
<tr>
<td></td>
<td>Materialized Views</td>
</tr>
</tbody>
</table>
Google BigTable (CP)

- Published by Google in 2006
- Original purpose: storing the Google search index

A Bigtable is a sparse, distributed, persistent multidimensional sorted map.

- Data model also used in: HBase, Cassandra, HyperTable, Accumulo

Wide-Column Data Modelling

- Storage of crawled web-sites („Webtable“):

1. Dimension: Row Key
2. Dimension: CF:Column
3. Dimension: Timestamp

- Column-Family: \textit{contents}:
  - content : "<html>..."

- Column-Family: \textit{anchor}:
  - cnnsi.com : "CNN"
  - my.look.ca : "CNN.com"

- Sorted
Range-based Sharding
BigTable Tablets

**Tablet**: Range partition of ordered records

<table>
<thead>
<tr>
<th>Rows</th>
<th>Tablet Server 1</th>
<th>Tablet Server 2</th>
<th>Tablet Server 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-C</td>
<td>A-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-F</td>
<td></td>
<td>C-F</td>
<td></td>
</tr>
<tr>
<td>F-I</td>
<td></td>
<td></td>
<td>F-I</td>
</tr>
<tr>
<td>I-M</td>
<td>I-M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M-T</td>
<td></td>
<td>M-T</td>
<td></td>
</tr>
<tr>
<td>T-Z</td>
<td></td>
<td></td>
<td>T-Z</td>
</tr>
</tbody>
</table>
Architecture

- Tablet Server
- Tablet Server
- Tablet Server
- Master
- Master Lock, Root Metadata Tablet
- Chubby
- ACLs, Garbage Collection, Rebalancing
- Stores Ranges, Answers client requests
- Stores data and commit log
- SSTables
- GFS
- Commit Log
- SSTables

Stores ranges, answers client requests

ACLs, Garbage Collection, Rebalancing

Stores data and commit log

Master Lock, Root Metadata Tablet

Governed by Master
Storage: Sorted-String Tables

- **Goal**: Append-Only IO when writing (no disk seeks)
- Achieved through: **Log-Structured Merge Trees**
- **Writes** go to an in-memory *memtable* that is periodically persisted as an *SSTable* as well as a *commit log*
- **Reads** query *memtable* and all *SSTables*
Apache HBase (CP)

- Open-Source Implementation of BigTable
- Hadoop-Integration
  - Data source for Map-Reduce
  - Uses Zookeeper and HDFS
- Data modelling challenges: key design, tall vs wide
  - **Row Key**: only access key (no indices) → key design important
  - **Tall**: good for scans
  - **Wide**: good for gets, consistent (*single-row atomicity*)
- No typing: application handles serialization
- Interface: REST, Avro, Thrift
HBase Storage

Logical to physical mapping:

<table>
<thead>
<tr>
<th>Key</th>
<th>cf1:c1</th>
<th>cf1:c2</th>
<th>cf2:c1</th>
<th>cf2:c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key Design – where to store data:
- r2:cf2:c2:t1:<value>
- r2:<value>:cf2:c2:t1:_
- r2:cf2:c2:<value>:t1:_

HFile cf2
- r1:cf2:c1:t1:<value>
- r2:cf2:c2:t1:<value>
- r3:cf2:c2:t2:<value>
- r3:cf2:c2:t1:<value>
- r5:cf2:c1:t1:<value>

HFile cf1
- r1:cf1:c1:t1:<value>
- r2:cf1:c2:t1:<value>
- r3:cf1:c2:t1:<value>
- r3:cf1:c1:t2:<value>
- r5:cf1:c1:t1:<value>

Example: Facebook Insights

Log Extraction every 30 min

MD5(Reversed Domain) + Reversed Domain + URL-ID

Row Key

<table>
<thead>
<tr>
<th>6PM Total</th>
<th>6PM Male</th>
<th>...</th>
<th>01.01 Total</th>
<th>01.01 Male</th>
<th>...</th>
<th>Total</th>
<th>Male</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7</td>
<td></td>
<td>100</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
<td>567</td>
</tr>
</tbody>
</table>

Atomic HBase Counter

CF:Daily

CF:Monthly

CF:All

TTL – automatic deletion of old rows

Lars George: “Advanced HBase Schema Design”
Summary: BigTable, HBase

- Data model: \((rowkey, cf: column, timestamp) \rightarrow value\)
- **API**: CRUD + Scan\((start-key, end-key)\)
- Uses distributed file system (GFS/HDFS)
- Storage structure: **Memtable** (in-memory data structure) + **SSTable** (persistent; append-only-IO)
- **Schema design**: only primary key access \(\rightarrow\) implicit schema (key design) needs to be carefully planned
- **HBase**: very literal open-source BigTable implementation
## Classification: HBase

### Techniques

<table>
<thead>
<tr>
<th>Classification</th>
<th>Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sharding</strong></td>
<td>Range-Sharding, Hash-Sharding, Entity-Group Sharding, Consistent Hashing, Shared Disk</td>
</tr>
<tr>
<td><strong>Replication</strong></td>
<td>Transaction Protocol, Sync. Replication, Async. Replication, Primary Copy, Update Anywhere</td>
</tr>
<tr>
<td><strong>Storage Management</strong></td>
<td>Logging, Update-in-Place, Caching, In-Memory, Append-Only Storage</td>
</tr>
<tr>
<td><strong>Query Processing</strong></td>
<td>Global Index, Local Index, Query Planning, Analytics, Materialized Views</td>
</tr>
</tbody>
</table>
Apache Cassandra (AP)

- Published 2007 by Facebook

**Idea:**
- BigTable’s wide-column data model
- Dynamo ring for replication and sharding

**Cassandra Query Language (CQL):** SQL-like query- and DDL-language

**Compound indices:** partition key (shard key) + clustering key (ordered per partition key) → Limited range queries

**Secondary indices:** hidden table with mapping → queries with simple equality condition
Architecture

Cassandra Node

Thrift

Session

Thrift

Session

Thrift RPC

or CQL

set_keyspace()

get_slice()

TCP Cluster

Messages

Column

Family Store

Row Cache

MemTable

Local

Filesystem

Key Cache

Storage

Proxy

Stores SSTables

and Commit Log

Replication,

Gossip, etc.

Stores Rows

Stores Primary Key Index

(Seek Position)

Hashing:

MD5(key)

Random Partitioner

Order Preserving

Partitioner

Snitch: Rack, Datacenter,

EC2 Region Information
## Classification: Cassandra Techniques

<table>
<thead>
<tr>
<th>Classification</th>
<th>Range-Sharding</th>
<th>Hash-Sharding</th>
<th>Entity-Group Sharding</th>
<th>Consistent Hashing</th>
<th>Shared Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sharding</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Replication</strong></td>
<td>Transaction Protocol</td>
<td>Sync. Replication</td>
<td>Async. Replication</td>
<td>Primary Copy</td>
<td>Update Anywhere</td>
</tr>
<tr>
<td><strong>Storage Management</strong></td>
<td>Logging</td>
<td>Update-In-Place</td>
<td>Caching</td>
<td>In-Memory</td>
<td>Append-Only Storage</td>
</tr>
<tr>
<td><strong>Query Processing</strong></td>
<td>Global Index</td>
<td>Local Index</td>
<td>Query Planning</td>
<td>Analytics</td>
<td>Materialized Views</td>
</tr>
</tbody>
</table>
MongoDB (CP)

- From humongous ≅ gigantic
- Tunable consistency
- Schema-free document database
- Allows complex queries and indexing
- Sharding (either range- or hash-based)
- Replication (either synchronous or asynchronous)
- Storage Management:
  - Write-ahead logging for redos (*journaling*)
  - Storage Engines: memory-mapped files, in-memory, Log-structured merge trees (WiredTiger)
Data Modelling

```
{
    "_id": ObjectId("51a5d316d70beffe74ecc940"),
    "title": "Iron Man 3",
    "year": 2013,
    "rating": 7.6,
    "director": "Shane Block",
    "genre": ["Action", "Adventure", "Sci-Fi"],
    "actors": ["Downey Jr., Robert", "Paltrow, Gwyneth"],
    "tweets": [
        {
            "user": "Franz Kafka",
            "text": "#nowwatching Iron Man 3",
            "retweet": false,
            "date": ISODate("2013-05-29T13:15:51Z")
        }
    ]
}

```

**Denormalisation** instead of joins

**Nesting** replaces 1:n and 1:1 relations

**Schemafreeness**: Attributes per document

**Unit of atomicity**: document

**Principles**
Sharding:
- Sharding attribute
- Hash vs. range sharding

Load-Balancing
- can trigger rebalancing of chunks (64MB) and splitting

Controls Write Concern:
Unacknowledged, Acknowledged, Journaled, Replica Acknowledged

- Receives all writes
- Replicates asynchronously
## Classification: MongoDB Techniques

<table>
<thead>
<tr>
<th>Sharding</th>
<th>Range-Sharding</th>
<th>Hash-Sharding</th>
<th>Entity-Group Sharding</th>
<th>Consistent Hashing</th>
<th>Shared Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replication</td>
<td>Transaction Protocol</td>
<td>Sync. Replication</td>
<td>Async. Replication</td>
<td>Primary Copy</td>
<td>Update Anywhere</td>
</tr>
<tr>
<td>Storage Management</td>
<td>Logging</td>
<td>Update-in-Place</td>
<td>Caching</td>
<td>In-Memory</td>
<td>Append-Only Storage</td>
</tr>
<tr>
<td>Query Processing</td>
<td>Global Index</td>
<td>Local Index</td>
<td>Query Planning</td>
<td>Analytics</td>
<td>Materialized Views</td>
</tr>
</tbody>
</table>
How can the choices for an appropriate system be narrowed down?
Outline

- NoSQL Foundations and Motivation
- The NoSQL Toolbox: Common Techniques
- NoSQL Systems
- Decision Guidance: NoSQL Decision Tree
- Decision Tree
- Classification Summary
- Literature Recommendations
NoSQL Decision Tree

Purpose: Application Architects: narrowing down the potential system candidates based on requirements

Database Vendors/Researchers: clear communication and design of system trade-offs
System Properties
According to the NoSQL Toolbox

- For fine-grained system selection:

<table>
<thead>
<tr>
<th></th>
<th>Scan Queries</th>
<th>ACID Transactions</th>
<th>Conditional Writes</th>
<th>Joins</th>
<th>Sorting</th>
<th>Filter Query</th>
<th>Full-Text Search</th>
<th>Analytics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mongo</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Redis</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Riak</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Cassandra</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

For fine-grained system selection:
System Properties
According to the NoSQL Toolbox

For fine-grained system selection:

<table>
<thead>
<tr>
<th></th>
<th>Data Scalability</th>
<th>Write Scalability</th>
<th>Read Scalability</th>
<th>Elasticity</th>
<th>Consistency</th>
<th>Write Latency</th>
<th>Read Latency</th>
<th>Write Throughput</th>
<th>Read Availability</th>
<th>Write Availability</th>
<th>Durability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mongo</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Redis</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Riak</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
System Properties
According to the NoSQL Toolbox

- For fine-grained system selection:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mongo</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Riak</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- **High-Level NoSQL Categories:**
  - Key-Value, Wide-Column, Document, Graph
  - Two out of {Consistent, Available, Partition Tolerant}

- **The NoSQL Toolbox:** systems use similar techniques that promote certain capabilities

- **Decision Tree**

  **Techniques**
  - Sharding, Replication, Storage Management, Query Processing

  **Promote**
  - Functional Requirements
  - Non-functional Requirements
Our NoSQL research at the University of Hamburg
Caching- and Database-as-a-Service Middleware for NoSQL databases

Cloud Startup for Orestes as a Service
Orestes
Components

Backend-as-a-Service Middleware:
- Caching, Transactions, Schemas,
  Invalidation Detection,
  …

Standard HTTP Caching
Polyglot Persistence Mediator

Unified REST API
This year's SCDM will be announced soon.
Literature Recommendations
Recommended Literature

1. NoSQL Distilled
   Pramodkumar J Sadalage & Martin Fowler

2. Designing Data-Intensive Applications
   Martin Kleppmann
Recommended Literature
Recommended Literature: Cloud-DBs

- *Web-Scale Data Management for the Cloud* by Wolfgang Lehner and Kai-Uwe Sattler
- *Cloud Data Management* by Liang Zhao, Sherif Sakr, Anna Liu, and Athman Bouguettaya
Recommended Literature: Blogs

**Martin Kleppmann**
https://martin.kleppmann.com/

**DZone**
http://www.dzone.com/mz/nosql

**InfoQ**
http://www.infoq.com/nosql/

**NoSQL Weekly**
http://www.nosqlweekly.com/

**BaQend**
http://blog.baqend.com/

**High Scalability**
http://highscalability.com/

**Metadata**
http://muratbuffalo.blogspot.de/

**DB-engines**
http://db-engines.com/en/ranking
Seminal NoSQL Papers

- G. DeCandia, et al., *Dynamo: Amazon's Highly Available Key-Value Store*, SOSP, 2007
- M. Stonebraker, et al., *The end of an architectural era: (it's time for a complete rewrite)*, VLDB, 2007
- Werner Vogels, *Eventually Consistent*, ACM Queue, 2009
- B. Cooper, et al., *Benchmarking cloud serving systems with YCSB.*, SOCC, 2010