Das große Einmaleins der NoSQL Datenbanken

Felix Gessert
felix.gessert@baqend.com
@baqendcom
Slides: slideshare.net/felixgessert
Article: medium.com/baqend-blog
About me

NoSQL research for PhD dissertation

CEO of startup for high-performance serverless development based on caching and NoSQL
Why NoSQL?
Architecture

Typical Data Architecture:

- Analytics
- Reporting
- Data Mining

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

**riak**
Key-Value Store
Large-scale session storage

**Neo4j**
The graph database
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

Firebase
Realtime BaaS
Communication and collaboration

Azure Tables
Wide-Column Store
Very large tables

bonsai
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

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

ElastiCache
Managed Cache
Caching and transient storage

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 requirements database 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
  - Impedance Mismatch

User-generated data, Request load

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

Orders
Payment
Customers

Line Items
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:

- \texttt{SELECT Name, Age FROM Customers}

- Explicit schema

NoSQL DB:

- Item[Price] - Item[Discount]

- Implicit schema
Open Source & Commodity Hardware

- Commercial DBMS
- Specialized DB hardware (Oracle Exadata, etc.)
- Highly available network (Infiniband, Fabric Path, etc.)
- Highly Available Storage (SAN, RAID, etc.)

- Open-Source DBMS
- Commodity hardware
- Commodity network (Ethernet, etc.)
- Commodity drives (standard HDDs, JBOD)
NoSQL System Classification

- Two common criteria:

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

  - **Consistency/Availability Trade-Off**
    - **AP**: Available & Partition Tolerant
    - **CP**: Consistent & Partition Tolerant
    - **CA**: Not Partition Tolerant
Key-Value Stores

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

Examples:
- **users:2:friends**: `{23, 76, 233, 11}
- **users:2:inbox**: `[234, 3466, 86,55]
- **users:2:settings**: Theme → "dark", cookies → "false"

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, algorithms, querys, transactions
- **Examples:** Neo4j (CA), InfiniteGraph (CA), OrientDB (CA)

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

- **Properties**
  - name: John Doe

- **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)
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*
CAP-Theorem: simplified proof

- **Problem**: when a network partition occurs, either consistency or availability have to be given up.

![Diagram showing network partition and replication issues]

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

Value = $V_1$

Value = $V_0$
NoSQL Triangle

Every client can always read and write

CA
Oracle, MySQL, ...

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

All clients share the same view on the data

A

AP
Dynamo, Redis, Riak, Voldemort
Cassandra
SimpleDB,

All nodes continue working under network partitions

Data models
Relational
Key-Value
Wide-Column
Document-Oriented

Nathan Hurst: Visual Guide to NoSQL Systems
http://blog.nahurst.com/visual-guide-to-nosql-systems
Idea: Classify systems according to their behavior during network partitions

PACELC – an alternative CAP formulation

- Availability
- Consistency
- Partition yes
- No no

No consequence of the CAP theorem

AL - Dynamo-Style Cassandra, Riak, etc.
AC - MongoDB
CC – Always Consistent HBase, BigTable and ACID systems

Abadi, Daniel. "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story."
Serializability
Not Highly Available Either

Global serializability and availability are incompatible:

- Write A = 1
  - Read B

- Write B = 1
  - Read A

\[ w_1(a = 1) r_1(b = \perp) \quad \text{and} \quad w_2(b = 1) r_2(a = \perp) \]

- 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

BASE

- Basically Available
- Soft State
- Eventually Consistent

„Gold standard“ for RDBMSs

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

- Scan Queries
- ACID Transactions
- Conditional or Atomic Writes
- Joins
- Sorting
- Filter Queries
- Full-text Search
- Aggregation and Analytics
- Sharding
- Replication
- Logging
- Update-in-Place
- Caching
- Append-Only Storage
- Storage Management
- Query Processing
- Elasticity
- Consistency
- Read Latency
- Write Throughput
- Read Availability
- Write Availability
- Durability

Non-Functional

- Data Scalability
- Write Scalability
- Read Scalability
- Write Latency
- Read Scalability
- Data Scalability
- Global Secondary Indexing
- Local Secondary Indexing
- Query Planning
- Analytics Framework
- Materialized Views
- Commit/Consensus Protocol
- Synchronous
- Asynchronous
- Primary Copy
- Update Anywhere
- Range-Sharding
- Hash-Sharding
- Entity-Group Sharding
- Consistent Hashing
- Shared-Disk

Central Techniques

- NoSQL databases employ
- Functional Requirements from the application
- require-ments
- require-ments
- require-ments
- require-ments
- require-ments
- require-ments

Enable

Enable

Enable
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. 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
- Scan Queries
- ACID Transactions
- Conditional or Atomic Writes
- Joins
- Sorting

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

Non-Functional
- Data Scalability
- Write Scalability
- Read Scalability
- Elasticity
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 (shard)
- **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

Implemented in
- 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

Causal Consistency

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

Linearity

A client reads in an increase monotonically.

Monotonic Consistency

One session are ordered on all versions.

Version-based or

Writes in one session are strictly ordered on all replicas.

Reads

Clients directly see their own writes.

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

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

Bounded Staleness

Achievable with high availability


Functional Techniques Non-Functional

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
- SW: Sequential Writes

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

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

Low Performance
High Performance

Update-In-Place
Append-Only I/O
Logging

In-Memory/Caching

In-Memory/Caching

Data

Log

Persistent Storage

Promotes durability of write operations.

Increases write throughput.

Is good for read latency.

Improve latency.

Increases write throughput.

Is good for read latency.

Promotes durability of write operations.

In-Memory/Caching

In-Memory/Caching

Data

Log

Persistent Storage

Promotes durability of write operations.

In-Memory/Caching

In-Memory/Caching

Data

Log

Persistent Storage

Promotes durability of write operations.

In-Memory/Caching

In-Memory/Caching

Data

Log

Persistent Storage

Promotes durability of write operations.
Functional Techniques

Non-Functional

Joins

Sorting

Filter Queries

Full-text Search

Aggregation and Analytics

Query Processing

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

Read Latency
Local Secondary Indexing
Partitioning By Document

Partition I

<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>

Index

<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></td>
</tr>
</tbody>
</table>

Partition II

<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>

Index

<table>
<thead>
<tr>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>[104]</td>
</tr>
<tr>
<td>[188,192]</td>
</tr>
</tbody>
</table>

Implemented in

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

WHERE color=blue

Scatter-gather query pattern.

Global Secondary Indexing
Partitioning By Term

Partition I

<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>

Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>[56, 188, 192]</td>
</tr>
<tr>
<td>Blue</td>
<td></td>
</tr>
</tbody>
</table>

Consistent maintenance requires distributed transaction.

Implemented in

- DynamoDB
- Oracle Datawarehouse
- Riak (Search)
- Cassandra (Search)

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

---


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)

\[ \text{partition} = \text{hash(key)} \mod \text{server_count} \]
Solution: **Consistent Hashing** – mapping of data to nodes is stable under topology changes
Consistent Hashing

- Extension: **Virtual Nodes** for Load Balancing

The diagram shows a circular range with points labeled A, B, and C. The text explains that B takes over two thirds of A, and C takes over one third of A. The diagram also includes a range "Range übernehmen."
Reading

- 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

\[ N = 3 \]
\[ R = 2 \]
\[ W = 1 \]
Quorums

- **N** (Replicas), **W** (Write Acks), **R** (Read Acks)
  - $R + W \leq N \implies$ No guarantee
  - $R + W > N \implies$ newest version included

![Diagram showing quorums](image-url)

<table>
<thead>
<tr>
<th>N = 12, R = 3, W = 10</th>
<th>N = 12, R = 7, W = 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D</td>
<td>A B C D</td>
</tr>
<tr>
<td>E F G H</td>
<td>E F G H</td>
</tr>
<tr>
<td>I J K L</td>
<td>I J K L</td>
</tr>
</tbody>
</table>

Read-Quorum

Write-Quorum
Writing

- **W** Servers have to ack

![Diagram with servers A, B, C, D, E, F connected and server with N=3, R=2, W=1]
Hinted Handoff

- Next node in the ring may take over, until original node is available again:

N=3
R=2
W=1
Vector clocks

- Dynamo uses **Vector Clocks** for versioning
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
Conflict Resolution

- The application merges data when writing (*Semantic Reconciliation*)
Merkle Trees: Anti-Entropy

- Every Second: Contact random server and compare
Quorum

Typical Configurations:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Quorum Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance (Cassandra Default)</td>
<td>N=3, R=1, W=1</td>
</tr>
<tr>
<td>Quorum, fast Writing:</td>
<td>N=3, R=3, W=1</td>
</tr>
<tr>
<td>Quorum, fast Reading</td>
<td>N=3, R=1, W=3</td>
</tr>
<tr>
<td>Trade-off (Riak Default)</td>
<td>N=3, R=2, W=2</td>
</tr>
</tbody>
</table>

LinkedIn (SSDs): \[P(\text{consistent}) \geq 99.9\% \text{ nach } 1.85 \text{ ms}\]

$R + W > N$ does not imply linearizability

Consider the following execution:

- **Writer**
  - set $x = 1$

- **Replica 1**
  - ok

- **Replica 2**
  - ok
  - 0

- **Replica 3**
  - 0
  - 0
  - 0

- **Reader A**
  - get $x \rightarrow 1$

- **Reader B**
  - get $x \rightarrow 0$

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)

```
Node 1

\[ S_1 = \{\} \]
\[ S_1 = \{x\} \]
\[ S_1 = \text{merge}(\{x\}, \{y\}) \]
\[ = \{x, y\} \]

Node 2

\[ S_2 = \{\} \]
\[ S_2 = \{y\} \]
\[ S_2 = \text{merge}(\{y\}, \{x\}) \]
\[ = \{x, y\} \]
```

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**

- Model: Key-Value
- License: Apache 2
- Written in: Erlang und C

Consistency Level: N, R, W, DW
Storage Backend: Bit-Cask, Memory, LevelDB

Data: KV-Pairs   Bucket
Riak Data Types

- Implemented as *state-based CRDTs*:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Convergence rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>enable wins over disable</td>
</tr>
<tr>
<td>Registers</td>
<td>The most chronologically recent value wins, based on timestamps</td>
</tr>
<tr>
<td>Counters</td>
<td>Implemented as a PN-Counter, so all increments and decrements are eventually applied.</td>
</tr>
<tr>
<td>Sets</td>
<td>If an element is concurrently added and removed, the add will win</td>
</tr>
<tr>
<td>Maps</td>
<td>If a field is concurrently added or updated and removed, the add/update will win</td>
</tr>
</tbody>
</table>

[http://docs.basho.com/riak/kv/2.1.4/learn/concepts/crdts/](http://docs.basho.com/riak/kv/2.1.4/learn/concepts/crdts/)
Hooks & Search

- **Hooks:**
  - Update/Delete/Create

- **Riak Search:**
  - /solr/mybucket/select?q=user:emil
  - Term | Dokument
  - database | 3,4,1
  - rabbit | 2
Riak Map-Reduce

POST /mapred

http://docs.basho.com/riak/latest/tutorials/querying/MapReduce/
Riak Map-Reduce

- JavaScript/Erlang, stored/ad-hoc
- Pattern: Chainable Reducers
- **Key-Filter**: Narrow down input
- **Link Phase**: Resolves links

```javascript
"key-filter" : [
    ["string_to_int"],
    ["less_than", 100]
]

"link" : {
    "bucket" : "nosql_dbs"
}
```

Same Data Format
Riak Cloud Storage

Amazon S3 API

1MB Chunks

Stanchion: Request Serializer

Files
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)
  - \( N=3, R=W=1 \) → fast, potentially inconsistent
  - \( N=3, R=3, W=1 \) → 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><strong>Sharding</strong></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>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>
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
Redis Architecture

- Redis Codebase $\cong$ 20K LOC
Persistence

- Default: „Eventually Persistent“
- **AOF**: Append Only File (~Commitlog)
- **RDB**: Redis Database Snapshot

```sh
config set appendonly everysec
```

```sh
fsync() every second
```

```
Snapshot every 60s, if > 1000 keys changed
```

```sh
config set save 60 1000
```
Persistence

1. Resistance to client crashes
2. Resistance to DB process crashes
3. Resistance to hardware crashes with Write-Through
4. Resistance to hardware crashes with Write-Back
Persistence: Redis vs an RDBMS

- **PostgreSQL:**
  - `synchronous_commit on`
  - `fsync false`
  - `pg_dump`

  Latency > Disk Latency, Group Commits, Slow

- **Redis:**
  - `synchronous_commit off`
  - `appendfsync always`
  - `appendfsync everysec`
  - `appendfysnc no`
  - `save oder bgsave`

  periodic fsync(), data loss limited

  Data loss possible, corruption prevented
Master-Slave Replication

> SLAVEOF 192.168.1.1 6379
< +OK

- Writes
- Asynchronous Replication

- Master
  - Slave1
  - Slave2
    - Slave2.1
    - Slave2.2

- Slave Offsets
- Memory Backlog
- Stream

Asynchronous Replication

Diagram showing the flow of writes from the Master to Slave1 and Slave2, with further replication to Slave2.1 and Slave2.2. SLAVEOF command configuration is also shown.
# Data structures

- **String**
  - `web:index` → "<html><head>..."

- **Set**
  - `users:2:friends` → \{23, 76, 233, 11\}

- **List**
  - `users:2:inbox` → \[234, 3466, 86,55\]

- **Hash**
  - `users:2:settings` → Theme → "dark", cookies → "false"

- **Sorted Set**
  - `top-posters` → 466 → "2", 344 → "16"

- **Pub/Sub**
  - `users:2:notifs` → "{event: 'comment posted', time : ...}"
Data Structures

- (Linked) Lists:
  - `LPUSHX`: Only if list exists
  - `LPUSH`
  - `LPUSHX`
  - `LPOP`
  - `LLEN`
  - `LRANGE inbox 1 2`
  - `LREM inbox 0 3466`
  - `LRANGE inbox 1 2`
  - `LINDEX inbox 2`
  - `RPUSH`
  - `RPOP`
  - Blocks until element arrives
  - `BLPOP`
Data Structures

Sets:

- `SADD`:
  - `user:2:friends`
  - `user:5:friends`

- `SREM`:
  - `user:2:friends`
  - `user:5:friends`

- `SCARD`:
  - `4`

- `SRANDMEMBER`:
  - `X` (23)

- `SINTER`:
  - `23 10 2 28 325 64 70`

- `SMEMBERS`:
  - `23 10 2 28 325 64 70`

- `SINTERSTORE`:
  - `common_friends`
  - `user:2:friends`
  - `user:5:friends`

- `SISMEMBER`:
  - `false` (23)

- `SRANDMEMBER`:
  - `23`

- `common_friends`
Data Structures

- Pub/Sub: 

```
PUBLISH user:2:notifs
"{
  event: 'comment posted',
  time : ...
}"
```

```
SUBSCRIBE user:2:notifs
{
  event: 'comment posted',
  time : ...
}
```
Example: Bloom filters
Compact Probabilistic Sets

- Bit array of length \( m \) and \( k \) independent hash functions
- \texttt{insert(obj)}: add to set
- \texttt{contains(obj)}: might give a false positive

Insert \( y \)

Query \( x \)

https://github.com/Baqend/Orestes-Bloomfilter
Bloomfilters in Redis

Bitvectors in Redis: String + SETBIT, GETBIT, BITOP

```java
public void add(byte[] value) {
    for (int position : hash(value)) {
        jedis.setbit(name, position, true);
    }
}

does not resize automatically

public void contains(byte[] value) {
    for (int position : hash(value))
        if (!jedis.getbit(name, position))
            return false;
    return true;
}
```

Jedis: Redis Client für Java

SETBIT creates and resizes automatically
If the Bloom filter uses 7 hashes: 7 roundtrips

**Solution:** Redis Pipelining
Our Bloom filters
Open Source Implementation

Library of different Bloom filters in Java with optional Redis-backing, counting and many hashing options. — Edit

Latest commit ab479b7 22 hours ago

- gradle/wrapper
  cleanup build
  2 years ago
- src
  - Added clear for ExpiringBloomFilter Memory
  22 hours ago
- .gitignore
  gitignore.iml
  4 months ago
- CHANGELOG.md
  Update CHANGELOG.md
  a year ago
- LICENSE
  Added Tutorial steps
  3 years ago
- README.md
  Update README.md
  a year ago
- bloom-filter.iml
  - Updated GSON
  7 months ago
- build.gradle
  - Updated GSON
  7 months ago
- gradle.properties
  [ci skip] new version commit: '1.1.9-SNAPSHOT'.
  22 hours ago
Redis for distributed systems

- Common Pattern: distributed system with **shared state** in Redis
- Example - Improve performance for legacy systems:

```
Slow Legacy System

On Hit

App Server

Bloomfilter lookup:
GETBIT, GETBIT...

Hash  →  MD5
k     →  7
m     →  80000
Bits  →  0100101011

Get Data From Legacy System
```
Why is Redis so fast?

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

Harizopoulos, Stavros, Madden, Stonebraker "OLTP through the looking glass, and what we found there."
Optimistic Transactions

- MULTI: Atomic Batch Execution
- WATCH: Condition for MULTI Block

```
WATCH users:2:followers, users:3:followers
MULTI
  SMEMBERS users:2:followers ➔ Queued
  SMEMBERS users:3:followers ➔ Queued
  INCR transactions ➔ Queued
EXEC ➔ Bulk reply with 3 results
```

Only executed if bother keys are unchanged
Lua Scripting

SCRIPT LOAD

Script Hash

EVALSHA $hash 1 "mylock" "10"

Redis Server

--lockscript, parameters: lock_key, lock_timeout
local lock = redis.call('get', KEYS[1])
if not lock then
    return redis.call('setex', KEYS[1], ARGV[1], "locked")
end
return false

Script Cache

1

Performance

- Equivalent to Memcache

> redis-benchmark -n 100000 -c 50
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>Category</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>
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: contents
  - content: "<html>...

- Column-Family: anchor
  - cnnsi.com: "CNN"
  - my.look.ca: "CNN.com"
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>C-F</td>
<td>F-I</td>
</tr>
<tr>
<td>C-F</td>
<td></td>
<td>M-T</td>
<td>T-Z</td>
</tr>
<tr>
<td>F-I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I-M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M-T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-Z</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Controls Ranges, Splits, Rebalancing

Master
Architecture

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

- Writes: In-Memory in **Memtable**
- SSTable disk access optimized by Bloom filters
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:

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

<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>
Example: Facebook Insights

Log

Extraction every 30 min

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

Row Key

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6PM Total</td>
<td>6PM Male</td>
<td>...</td>
<td>01.01 Total</td>
<td>01.01 Male</td>
<td>...</td>
<td>Total Male</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td></td>
<td>100</td>
<td>65</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”
Schema Design

- Tall vs Wide Rows:
  - **Tall**: good for Scans
  - **Wide**: good for Gets

- Hotspots: Sequential Keys (z.B. Timestamp) dangerous

# Schema: Messages

<table>
<thead>
<tr>
<th>User ID</th>
<th>CF</th>
<th>Column</th>
<th>Timestamp</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>data</td>
<td>5fc38314-e290-ae5da5fc375d</td>
<td>1307097848</td>
<td>&quot;Hi Lars, ...&quot;</td>
</tr>
<tr>
<td>12345</td>
<td>data</td>
<td>725aae5f-d72e-f90f3f070419</td>
<td>1307099848</td>
<td>&quot;Welcome, and ...&quot;</td>
</tr>
<tr>
<td>12345</td>
<td>data</td>
<td>cc6775b3-f249-c6dd2b1a7467</td>
<td>1307101848</td>
<td>&quot;To Whom It ...&quot;</td>
</tr>
<tr>
<td>12345</td>
<td>data</td>
<td>dcbee495-6d5e-6ed48124632c</td>
<td>1307103848</td>
<td>&quot;Hi, how are ...&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID:User+Message</th>
<th>CF</th>
<th>Column</th>
<th>Timestamp</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345-5fc38314-e290-ae5da5fc375d</td>
<td>data</td>
<td></td>
<td>1307097848</td>
<td>&quot;Hi Lars, ...&quot;</td>
</tr>
<tr>
<td>12345-725aae5f-d72e-f90f3f070419</td>
<td>data</td>
<td></td>
<td>1307099848</td>
<td>&quot;Welcome, and ...&quot;</td>
</tr>
<tr>
<td>12345-cc6775b3-f249-c6dd2b1a7467</td>
<td>data</td>
<td></td>
<td>1307101848</td>
<td>&quot;To Whom It ...&quot;</td>
</tr>
<tr>
<td>12345-dcbee495-6d5e-6ed48124632c</td>
<td>data</td>
<td></td>
<td>1307103848</td>
<td>&quot;Hi, how are ...&quot;</td>
</tr>
</tbody>
</table>

**Wide:**
- Atomicity
- Scan over Inbox: **Get**

**Tall:**
- Fast Message Access
- Scan over Inbox: **Partial Key Scan**

API: CRUD + Scan

Setup Cloud Cluster:

```java
> elastic-mapreduce --create --
  hbase --num-instances 2 --instance-type m1.large

HTable table = ...
Get get = new Get("my-row");
get.addColumn(Bytes.toBytes("my-cf"), Bytes.toBytes("my-col"));
Result result = table.get(get);

table.delete(new Delete("my-row"));

Scan scan = new Scan();
scan.setStartRow(Bytes.toBytes("my-row-0"));
scan.setStopRow(Bytes.toBytes("my-row-101"));
ResultScanner scanner = table.getScanner(scan)
for(Result result : scanner) { }
```

> whirr launch-cluster --config
  hbase.properties
  Login, cluster size, etc.
API: Features

- **Row Locks (MVCC):** `table.lockRow()`, `unlockRow()`
  - Problem: Timeouts, Deadlocks, Resources

- **Conditional Updates:** `checkAndPut()`, `checkAndDelete()`

- **CoProcessors - registered Java-Classes for:**
  - Observers (`prePut`, `postGet`, etc.)
  - Endpoints (Stored Procedures)

- **HBase can be a Hadoop Source:**

```java
TableMapReduceUtil.initTableMapperJob(
    tableName,  //Table
    scan,       //Data input as a Scan
    MyMapper.class, ... //usually a TableMapper<Text,Text> );
```
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>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>Sync. Replication</td>
<td>Async. Replication</td>
<td>Primary Copy</td>
<td>Update Anywhere</td>
<td></td>
</tr>
<tr>
<td>Storage Management</td>
<td>Logging</td>
<td>Update-in-Place</td>
<td>In-Memory</td>
<td>Append-Only Storage</td>
<td></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>
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
Architecture

Cassandra Node
Thrift
Session
Thrift Session
Thrift RPC
or CQL
set_keyscape()
get_slice()

TCP Cluster
Messages

Column
Family Store
Row Cache
MemTable
Local
Filesystem
Key Cache

Stores SSTables
and Commit Log

Replication,
Gossip, etc.

Stateful
Communication

Stores Rows

Stores Primary Key Index
(Seek Position)

Hashing:

Random Partitioner
Order Preserving Partitioner

MD5(key)

Snitch: Rack, Datacenter, EC2 Region Information
Consistency

- No Vector Clocks but **Last-Write-Wins**
  - Clock synchronisation required
- No Versionierung that keeps old cells

<table>
<thead>
<tr>
<th>Write</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>-</td>
</tr>
<tr>
<td>One</td>
<td>One</td>
</tr>
<tr>
<td>Two</td>
<td>Two</td>
</tr>
<tr>
<td>Quorum</td>
<td>Quorum</td>
</tr>
<tr>
<td>Local_Quorum / Each_Quorum</td>
<td>Local_Quorum / Each_Quorum</td>
</tr>
<tr>
<td>All</td>
<td>All</td>
</tr>
</tbody>
</table>
Consistency

- Coordinator chooses newest version and triggers **Read Repair**
- **Downside**: upon conflicts, changes are lost
Storage Layer

- Uses BigTables Column Family Format

**KeySpace**: music

**Column Family**: songs

- **Row Key**: Mapping to Server
- **Comparator** determines order
- **Sparse**

- **f82831...**
  - **title**: Andante
  - **album**: New World Symphony
  - **artist**: Antonin Dvorak

- **144052...**
  - **title**: Jailhouse Rock
  - **artist**: Elvis Presley

Type validated by **Validation Class** **UTFType**

More details: [http://www.datastax.com/dev/blog/cql3-for-cassandra-experts](http://www.datastax.com/dev/blog/cql3-for-cassandra-experts)
CQL Example: Compound keys

- Enables Scans despite Random Partitioner

CREATE TABLE playlists (
  id uuid,
  song_order int,
  song_id uuid, ...
PRIMARY KEY (id, song_order)
);

<table>
<thead>
<tr>
<th>id</th>
<th>song_order</th>
<th>song_id</th>
<th>artist</th>
</tr>
</thead>
<tbody>
<tr>
<td>23423</td>
<td>1</td>
<td>64563</td>
<td>Elvis</td>
</tr>
<tr>
<td>23423</td>
<td>2</td>
<td>f9291</td>
<td>Elvis</td>
</tr>
</tbody>
</table>

SELECT * FROM playlists
WHERE id = 23423
ORDER BY song_order DESC
LIMIT 50;
Other Features

- **Distributed Counters** – prevent update anomalies
- **Full-text Search** (Solr) in Commercial Version
- **Column TTL** – automatic garbage collection
- **Secondary indices**: hidden table with mapping
  → queries with simple equality condition
- **Lightweight Transactions**: linearizable updates through a Paxos-like protocol

```sql
INSERT INTO USERS (login, email, name, login_count)
values ('jbellis', 'jbellis@datastax.com', 'Jonathan Ellis', 1)
IF NOT EXISTS
```
## Classification: Cassandra

### 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>
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)
Basics

> mongod &
> mongo imdb
MongoDB shell version: 2.4.3
connecting to: imdb
> show collections
movies
tweets
> db.movies.findOne({title : "Iron Man 3"})
{
  title : "Iron Man 3",
  year : 2013,
  genre : [
    "Action",
    "Adventure",
    "Sci-Fi"],
  actors : [
    "Downey Jr., Robert",
    "Paltrow , Gwyneth",]
}
Data Modelling

Denormalisation instead of joins

Nesting replaces 1:n and 1:1 relations

Schemafreeness: Attributes per document

Unit of atomicity: document

Principles

```json
{
  "_id": ObjectId("51a5d316d70befef74ecc940"),
  "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")
    }
  ]
}
```
Sharding and Replication

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

Mounts:
- Receives all writes
- Replicates asynchronously

Client
config
mongos
MongoDB
Replica Set
Slave
Master
Slave
Slave
MongoDB Example App

Twitter Firehose

@Johnny: Watching Game of Thrones
@Jim: Star Trek rocks.

MovieService

MongoDB

REST API (Jetty)

GET
MongoDB
Tweets
Streaming
GridFS
Tweet Map
Searching
JSON
Queries

Browser

1. GET

Server

Browser

Client

HTTP

Movies

Tweets

saveTweet()
getTaggedTweets()
getByGenre()
searchByPrefix()
Unveiling the geographic patterns underlying tweets about movies.

```javascript
DBObject query = new BasicDBObject("tweets.coordinates",
  new BasicDBObject("$exists", true));

db.getCollection("movies").find(query);

Or in JavaScript:

db.movies.find({tweets.coordinates : { "$exists" : 1}})
```

Overhead caused by large results -> projection
The Movie mApp
Unveiling the geographic patterns underlying tweets about movies.

```
db.tweets.find({coordinates : {"$exists" : 1}},
    {text:1, movie:1, "user.name":1, coordinates:1})
.sort({id:-1})
```

Projected attributes, ordered by insertion date
db.movies.ensureIndex({title : 1})
db.movies.find({title : /^Incep/}).limit(10)

Index usage:
db.movies.find({title : /^Incep/}).explain().millis = 0
db.movies.find({title : /^Incep/i}).explain().millis = 340
db.movies.update({_id: id}, {"$set": {"comment": c}})
or:

db.movies.save(changed_movie);

One of the best movies, that
```javascript
fs = new GridFs(db);
fs.createFile(inputStream).save();
```
Geospatial Queries:
• Distance
• Intersection
• Inclusion
db.tweets.runCommand( "text", { search: "StAr trek" } )

Full-text Search:
- Tokenization, Stop Words
- Stemming
- Scoring
Analytic Capabilities

- Aggregation Pipeline Framework:
  - **Match:** Selection by query
  - **Projection**
  - **Unwind:** elimination of nesting
  - **Skip and Limit**
  - **Grouping**, e.g.:
    ```javascript
    {_id : "$author",
     docsPerAuthor : { $sum : 1 },
     viewsPerAuthor : { $sum : "$views" }}
    ```

- Alternative: JavaScript MapReduce
Sharding

- **Range-based:**
  - In optimal case only one shard asked per query, else: Scatter-and-gather

- **Hash-based:**
  - Even distribution, no locality

---

[docs.mongodb.org/manual/core/sharding-introduction/](docs.mongodb.org/manual/core/sharding-introduction/)
Sharding

- Splitting:

- Migration:
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

- Decision Tree
- Classification Summary
- Literature Recommendations

NoSQL Foundations and Motivation

The NoSQL Toolbox: Common Techniques

NoSQL Systems

Decision Guidance: NoSQL Decision Tree
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Redis</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Riak</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x x</td>
</tr>
<tr>
<td>Cassandra</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</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>Non-functional Requirements</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></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>Redis</td>
<td></td>
<td>x</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>
</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>x</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>x</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>x</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>
<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>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Redis</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>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>HBase</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>
<td>x</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>x</td>
<td></td>
</tr>
<tr>
<td>Cassandra</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>x</td>
<td></td>
<td></td>
<td>x</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</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

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

  - Functional Requirements

  - Non-functional Requirements

- Decision Tree
Our NoSQL research at the University of Hamburg
Presentation is loading
Why performance matters

1s

Loading...

Average: 9.3s

-$

Conversions

% Traffic

% Visitors

Revenue
If perceived speed is such an important factor...

...what causes slow page load times?
The Problem

Three Bottlenecks: Latency, Backend & Frontend

High Latency

Frontend

Backend
Network Latency: Impact

- **Page Load Time as bandwidth increases**
  - 1 Mbps: 3500 ms
  - 2 Mbps: 2300 ms
  - 3 Mbps: 1800 ms
  - 4 Mbps: 1300 ms
  - 5 Mbps: 1000 ms
  - 6 Mbps: 800 ms
  - 7 Mbps: 600 ms
  - 8 Mbps: 400 ms
  - 9 Mbps: 300 ms
  - 10 Mbps: 200 ms

- **Page Load Time as latency decreases**
  - 200 ms: 3500 ms
  - 180 ms: 3000 ms
  - 160 ms: 2500 ms
  - 140 ms: 2000 ms
  - 120 ms: 1500 ms
  - 100 ms: 1000 ms
  - 80 ms: 700 ms
  - 60 ms: 400 ms
  - 40 ms: 200 ms
  - 20 ms: 100 ms

Network Latency: Impact

\[ 2 \times \text{Bandwidth} = \text{Same Load Time} \]

\[ \frac{1}{2} \text{ Latency} \approx \frac{1}{2} \text{ Load Time} \]

Goal: Low-Latency for Dynamic Content
By Serving Data from Ubiquitous Web Caches
In a nutshell

Problem: changes cause stale data
In a nutshell
Solution: Proactively Revalidate Data

Cache Sketch (Bloom filter)

Is still fresh?

update
Innovation
Solution: Proactively Revalidate Data


Bloom filters for Caching
End-to-End Example

False-Positive Rate:
\[ f \approx \left(1 - e^{-\frac{k}{m}}\right)^k \]
Hash-Functions:
\[ k = \left\lfloor \ln(2) \cdot \left(\frac{n}{m}\right) \right\rfloor \]

With 20.000 distinct updates and 5% error rate: **11 Kbyte**

**Consistency Guarantees:** Δ-Atomicity, Read-Your-Writes, Monotonic Reads, Monotonic Writes, Causal Consistency

Gets **Time-to-Live Estimation** by the server
Orestes Architecture

Infrastructure

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

Unified REST API

Standard HTTP

Caching

Orestes Servers

Orestes

Desktop

Mobile

Tablet

Content-Delivery-Network

Internet

Node.js

User-defined Business Logic

Node.js

User-defined Business Logic

MongoDB

Elasticsearch

Redis

Reverse-Proxy Caches

Expiring Bloom Filter

Stale Data

TTL Estimator

Cache Lifetime Prediction

InvalidationDB

Streaming Queries

...
Page-Load Times
What impact does caching have in practice?

Politik
11. November 2014 12:42 Uhr
Deutsche Rentenversicherung
Renten könnten 2015 um zwei Prozent steigen
Die Deutsche Rentenversicherung geht von einem Anstieg über der Inflationsrate aus. Abschlagsfreie Rente ab 53 Jahren stößt auf großes Interesse.

11. November 2014 10:06 Uhr
Europäischer Gerichtshof
Deutschland darf EU-Ausländern Hartz IV verweigern

11. November 2014 06:48 Uhr
APEC-GIPFELTREFFEN
Obama besänftigt China
Die USA wollen China nicht klein halten, sagt Präsident Obama vor dem Treffen mit Chinas Staatschef Xi. Der plädiert für mehr wirtschaftliche Verflechtung.

10. November 2014 19:17 Uhr
ISRAEL
Keiner will von Intifada sprechen
Messerattacken auf Israelis, Kravalle auf dem Tempelberg, Schummelzüge im Gassengewirr

11. November 2014 07:15 Uhr
HONORARBERATUNG
Guter Rat zur Geldanlage ist selten
Honorarberatung ist in Deutschland endlich gesetzlich geregelt. Doch gibt es kaum Honorarberater. Und gut qualifiziertes noch viel weniger.

11. November 2014 21:32 Uhr
CHINA
Der berühmteste Wohlträger Chinas – nach eigenen Angaben
Der chinesische Unternehmer Chen Guangbiao wurde ausgezeichnet mit Bauschutt sehr reich. Jetzt baut er Wände aus Geldbündeln und zentrifugiert öffentlich Luxusautos.

11. November 2014 19:29 Uhr
KONJUNKTUR
China steckt in der Wachstumsfalle

19. November 2014 13:45 Uhr
WÄHRUNG
Russlands Zentralbank lässt Rubel frei handeln

Kultur
11. November 2014 10:14 Uhr
NICOLAUS HARMONCOURT
Mozarts Triptychon

11. November 2014 06:39 Uhr
HANS MAGNUS EINZENSBERGER
Der Unerschütterliche
Hans Magnus Einzensberger wird 85. Ein Besuch bei dem herzlich eigenwilligen Intellektuellen. Mit Tumult hat er gerade ein erstanständisches Buch veröffentlicht.

10. November 2014 um 18:25 Uhr
DDR-DESIGN
Sandmännchen und Stasi-Mikrofone

10. November 2014 um 15:25 Uhr
AZEALIA BANKS
Klare Ansage aus Harlem
Erst gab Azelia Banks als großes Raptalent, dann als streitsüchtig und selbstverliebt. Ihr seit Jahren erwartetes Debüt zeigt jetzt, wie gut das eine zum anderen passt.
The World's Fastest Backend

Build websites and apps that load instantly.

Sky-rocket your Development

Start building now. Baqend Cloud is free and easy to get started with.
Want to try Baqend?

Free Baqend Cloud Instance at bagend.com

Download Community Edition
WebLabs.Hamburg


Wir entwickeln Webseiten und Apps.

Wir realisieren für Unternehmen Web und App-Projekte. Unsere besondere Expertise liegt in schnellen Ladezeiten und hoher Skalierbarkeit. Das Team besteht sowohl aus Spezialisten für Datenbanken, Backend- und Big Data

Was wir machen  Was wir bieten  Wer wird sind  Kontakt
Literature Recommendations
Recommended Literature

1. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence

2. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems
Recommended Literature

- Lena Wiese, *Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases*
- Dan Sullivan, *NoSQL: For Mere Mortals*
Recommended Literature: Cloud-DBs

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

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

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

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

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

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

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

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

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

- S. Gilbert, et al., Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, SIGACT News, 2002
- 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
4th Workshop on Scalable Cloud Data Management

Co-located with the IEEE BigData Conference.
Washington D.C., December 5th 2016.

Submit Paper

June 6, 2016

SCDM 2016 announced

The fourth Scalable Cloud Data Management Workshop (SCDM 2016) will again be held in conjunction with the IEEE BigData 2016 - this year in Washington D.C.
Thank you

felix.gessert@baqend.com
baqend.com