Scalable Data Management
An In-Depth Tutorial on NoSQL Data Stores

Felix Gessert, Wolfram Wingerath, Norbert Ritter
fg@baqend.com
March 7th, 2017, Stuttgart
@baqendcom
Slides: slideshare.net/felixgessert

Article: medium.com/baqend-blog
Outline

- NoSQL Foundations and Motivation
- The NoSQL Toolbox: Common Techniques
- NoSQL Systems & Decision Guidance
- Scalable Real-Time Databases and Processing
- 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:

Data Analytics
- Analytics
- Data Warehouse

Data Management
- Operative Database
- Applications

Reporting
Data Mining

Diagram showing the relationship between data management and data analytics components.
Architecture

Typical Data Architecture:

- Analytics
- Data Warehouse
- Operative Database
- Applications
- Reporting
- Data Mining
- NoSQL
Architecture

Typical Data Architecture:

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

→ **Specialized** data systems
## The Database Explosion

### Sweetspots

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>General-purpose</th>
<th>ACID transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM DB2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wide-Column Store</th>
<th>Long scans over structured data</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBase</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph Database</th>
<th>Graph algorithms &amp; queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neo4j</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel DWH</th>
<th>Aggregations/OLAP for massive data amounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greenplum</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In-Memory KV-Store</th>
<th>Counting &amp; statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redis</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key-Value Store</th>
<th>Large-scale session storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Riak</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NewSQL</th>
<th>High throughput relational OLTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>VoltDB</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wide-Column Store</th>
<th>Massive user-generated content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td></td>
</tr>
</tbody>
</table>

---
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

**Amazon 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

Payment: Credit Card, ...

Line Item 1: ...
Line Item2: ...

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
Big Data
The Analytic side of NoSQL

- **Idea**: make existing massive, unstructured data amounts usable

  - Sources
    - Structured data (DBs)
    - Log files
    - Documents, Texts, Tables
    - Images, Videos
    - Sensor data
    - Social Media, Data Services

  - Analyst, Data Scientist, Software Developer
    - Statistics, Cubes, Reports
    - Recommender
    - Classificators, Clustering
    - Knowledge
NoSQL Paradigm Shift

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 Paradigm Shift
Shared Nothing Architectures

Shift towards higher distribution & less coordination:

- **Shared Memory** e.g. "Oracle 11g"
- **Shared Disk** e.g. "Oracle RAC"
- **Shared Nothing** e.g. "NoSQL"
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: 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), RethinkDB (CP), MongoDB (CP)
Graph Databases

- **Data model:** $G = (V, E)$: Graph-Property Modell
- **Interface:** Traversal algorithms, queries, transactions

- **Examples:** Neo4j (CA), InfiniteGraph (CA), OrientDB (CA)
Graph Databases

- **Data model**: \( G = (V, E) \): Graph-Property Modell
- **Interface**: Traversal algorithms, queries, transactions
- **Examples**: Neo4j (CA), InfiniteGraph (CA), OrientDB (CA)

Nodes

- Company: Apple
  - Value: 300Mrd

Properties

- Name: John Doe

usually unscalable (optimal partitioning is NP-complete)

WORKS FOR since: 1999

Salary: 140K
Search Platforms

- **Data model:** vectorspace model, docs + metadata
- **Examples:** Solr, ElasticSearch

REST API

```
POST /lectures/dis
{  "topic": "databases",
    "lecturer": "ritter",
    ...
}
```
Object-oriented Databases

- **Data model**: Classes, objects, relations (references)
- **Interface**: CRUD, querys, transactions

Examples: Versant (CA), db4o (CA), Objectivity (CA)
Object-oriented Databases

- **Data model:** Classes, objects, relations (references)
- **Interface:** CRUD, querys, transactions

- Examples: Versant (CA), db4o (CA), Objectivity (CA)

- not scalable
- strong coupling between programming language and database
XML databases, RDF Stores

- **Data model**: XML, RDF
- **Interface**: CRUD, queries (XPath, XQueries, SPARQL), transactions (some)
- **Examples**: MarkLogic (CA), AllegroGraph (CA)
XML databases, RDF Stores

- **Data model:** XML, RDF
- **Interface:** CRUD, querys (XPath, XQuerys, SPARQL), transactions (some)
- **Examples:** MarkLogic (CA), AllegroGraph (CA)

- not scalable
- not widely used
- specialized data model
Distributed File System

- **Data model**: files + folders

**Network FS**
- Client
- RPC
- Server
  - Stub
  - NFS, AFS

**Cluster FS**
- I/O Nodes
  - RPC
  - SAN
  - GPFS, Lustre

**Distributed FS**
- RPC
  - HDFS
Big Data Batch Processing

- **Data model**: arbitrary (frequently unstructured)
- **Examples**: Hadoop, Spark, Flink, DryadLink, Pregel
Big Data Stream Processing
Covered in Depth in the Last Part

- **Data model**: arbitrary
- Examples: Storm, Samza, Flink, Spark Streaming
Real-Time Databases
Covered in Depth in the Last Part

- **Data model**: several data models possible
- **Interface**: CRUD, Querys + Continuous Queries

- Examples: Firebase (CP), Parse (CP), Meteor (CP), Lambda/Kappa Architecture
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
CAP-Theorem: simplified proof

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

![Diagram showing network partition, replication, and values V0 and V1]

- **Value**: $V_0$
- **Value**: $V_1$
- **Response** before successful replication
- **Consistency**
- **Availability**

Network partition
NoSQL Triangle

Every client can always read and write

All clients share the same view on the data

All nodes continue working under network partitions

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

CA
Oracle, MySQL, ...

AP
Dynamo, Redis, Riak, Voldemort
Cassandra
SimpleDB

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

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

[Diagram showing the classification of systems based on availability (AL), consistency (AC), and latency (CC).]

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

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:

1. Write $A = 1$
   Read $B$

2. Write $B = 1$
   Read $A$

\[ \begin{align*}
  w_1(\mathit{a} = 1) & \quad r_1(\mathit{b} = \perp) \\
  w_2(\mathit{b} = 1) & \quad r_2(\mathit{a} = \perp)
\end{align*} \]

- Some weaker isolation levels allow high availability:

Consensus:
- **Agreement**: No two processes can commit different decisions
- **Validity (Non-triviality)**: If all initial values are same, nodes must commit that value
- **Termination**: Nodes commit eventually

No algorithm *guarantees* termination (FLP)

Algorithms:
- **Paxos** (e.g. Google Chubby, Spanner, Megastore, Aerospike, Cassandra Lightweight Transactions)
- **Raft** (e.g. RethinkDB, etcd service)
- Zookeeper Atomic Broadcast (**ZAB**)

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
Weaker guarantees in a database?!

Default Isolation Levels in RDBMSs

<table>
<thead>
<tr>
<th>Database</th>
<th>Default Isolation</th>
<th>Maximum Isolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres 10.0/10S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Aerospike</td>
<td>RC</td>
<td>RC</td>
</tr>
<tr>
<td>Clustrix CLX 4100</td>
<td>RR</td>
<td>?</td>
</tr>
<tr>
<td>Greenplum 4.1</td>
<td>RC</td>
<td>S</td>
</tr>
<tr>
<td>IBM DB2 10 for z/OS</td>
<td>CS</td>
<td>S</td>
</tr>
<tr>
<td>IBM Informix 11.50</td>
<td>Depends</td>
<td>RR</td>
</tr>
<tr>
<td>MySQL 5.6</td>
<td>RR</td>
<td>S</td>
</tr>
<tr>
<td>MemSQL 1b</td>
<td>RC</td>
<td>RC</td>
</tr>
<tr>
<td>MS SQL Server 2012</td>
<td>RC</td>
<td>S</td>
</tr>
<tr>
<td>NuoDB</td>
<td>CR</td>
<td>CR</td>
</tr>
<tr>
<td>Oracle 11g</td>
<td>RC</td>
<td>SI</td>
</tr>
<tr>
<td>Oracle Berkeley DB</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Postgres 9.2.2</td>
<td>RC</td>
<td>S</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>RC</td>
<td>SI</td>
</tr>
<tr>
<td>ScaleDB 1.02</td>
<td>RC</td>
<td>RC</td>
</tr>
<tr>
<td>VoltDB</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>


Weaker guarantees in a database?!

Default Isolation Levels in RDBMSs

<table>
<thead>
<tr>
<th>Database</th>
<th>Default Isolation</th>
<th>Maximum Isolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres 10.0/10S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Aerospike</td>
<td></td>
<td>RC</td>
</tr>
<tr>
<td>Clustrix CLX 4100</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>Greenplum 4.1</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>IBM DB2 10 for z/OS</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>IBM Informix 11.50</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>MySQL 5.6</td>
<td></td>
<td>Depends</td>
</tr>
<tr>
<td>MemSQL 1b</td>
<td>RC</td>
<td>S</td>
</tr>
<tr>
<td>MS SQL Server 2012</td>
<td>RC</td>
<td>S</td>
</tr>
<tr>
<td>NuoDB</td>
<td>CR</td>
<td>CR</td>
</tr>
<tr>
<td>Oracle 11g</td>
<td>RC</td>
<td>SI</td>
</tr>
<tr>
<td>Oracle Berkeley DB</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Postgres 9.2.2</td>
<td>RC</td>
<td>S</td>
</tr>
<tr>
<td>SAP HANA</td>
<td>RC</td>
<td>SI</td>
</tr>
<tr>
<td>ScaleDB 1.02</td>
<td>RC</td>
<td>RC</td>
</tr>
<tr>
<td>VoltDB</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

**Theorem:**
Trade-offs are central to database systems.

---


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

Scalable Real-Time Databases and Processing

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

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

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

- Replication
  - Commit/Consensus Protocol
  - Synchronous
  - Asynchronous
  - Primary Copy
  - Update Anywhere

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

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

- Data Scalability
- Write Scalability
- Read Scalability
- Elasticity
- Consistency
- Write Latency
- Read Latency
- Write Throughput
- Read Availability
- Write Availability
- Durability
Functional Techniques

- Scan Queries
- ACID Transactions
- Conditional or Atomic Writes
- Joins
- Sorting
- Filter Queries
- Full-text Search

Non-Functional

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

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

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

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

Central techniques NoSQL databases employ

Operational Requirements

- Data Scalability
- Write Scalability
- Read Scalability
- Write Scalability
- Read Scalability

Enable

Functional Requirements from the application

Operational Requirements

Enable

Central techniques NoSQL databases employ

Operational Requirements

Enable

Central techniques NoSQL databases employ

Operational Requirements

Enable

Central techniques NoSQL databases employ

Operational Requirements

Enable
NoSQL Database Systems:
A Survey and Decision Guidance

Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Rütter

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 [3]. 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 Non-Functional

- Scan Queries
- ACID Transactions
- Conditional or Atomic Writes
- Joins
- Sorting
- Sharding
  - Range-Sharding
  - Hash-Sharding
  - Entity-Group Sharding
  - Consistent Hashing
  - Shared-Disk

- Data Scalability
- Write Scalability
- Read Scalability
- Elasticity
Sharding

Approaches

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) to partitions
- **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 changable

---

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 field(s) to partitions
- **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

Read Scalability + Failure Tolerance

- 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 or update propagation
- **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
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

Synchronous Replication
Example: Two-Phase Commit is not partition-tolerant
Synchronous Replication

Example: Two-Phase Commit is not partition-tolerant
Synchronous Replication

Example: Two-Phase Commit is not partition-tolerant
Synchronous Replication
Example: Two-Phase Commit is not partition-tolerant

Diagram showing the process of committing transactions in a synchronous replication setup. The diagram illustrates the preparation and commitment phases, with nodes indicating the status of the transaction (prepared, committed, etc.). A lightning bolt indicates a possible failure point.
Synchronous Replication

Example: Two-Phase Commit is not partition-tolerant
Synchronous Replication

Example: Two-Phase Commit is not partition-tolerant
Synchronous Replication

Example: Two-Phase Commit is not partition-tolerant
Consistency Levels

- Linearity
- Causal Consistency
- PRAM
  - Writes Follow Reads
  - Read Your Writes
  - Monotonic Reads
  - Monotonic Writes
  - Bounded Staleness

References:
Consistency Levels

- **Linearity**
  - Causal Consistency
  - PRAM
    - Writes Follow Reads
    - Read Your Writes
    - Monotonic Reads
    - Monotonic Writes
    - Bounded Staleness

Either *version-based* or *time-based*. Both not highly available.

---


Consistency Levels

- **Linearity**
  - Causal Consistency
    - Writes in one session are strictly ordered on all replicas.
  - PRAM Causal Consistency

- **Consistency**
  - Writes Follow Reads
  - Read Your Writes
  - Monotonic Reads
  - Monotonic Writes
  - Bounded Staleness

---


Consistency Levels

Linearity

Causal Consistency

Versions a client reads in a session increase monotonically.

 Writes Follow Reads

Read Your Writes

Monotonic Reads

Monotonic Writes

Bounded Staleness


Consistency Levels

- Clients directly see their own writes.
- Writes Follow Reads
- Read Your Writes
- Monotonic Reads
- Monotonic Writes
- Bounded Staleness

- Causal Consistency
- Linearity
- PRAM
- Bounded Staleness

References:
Consistency Levels

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


Consistency Levels

Achievable with high availability

Causal Consistency

Linearity

PRAM

 Writes Follow Reads

Read Your Writes

Monotonic Reads

Monotonic Writes

Bounded Staleness


Consistency Levels

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


Problem: Terminology


**Definition:** Once the user has written a value, subsequent reads will return this value (or newer versions if other writes occurred in between); the user will never see versions older than his last write.


https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/
**Monotonic Reads (MR)**

**Definition:** Once a user has read a version of a data item on one replica server, it will never see an older version on any other replica server.
Montonic Writes (MW)

**Definition:** Once a user has written a new value for a data item in a session, any previous write has to be processed before the current one. I.e., the order of writes inside the session is strictly maintained.
**Definition:** When a user reads a value written in a session after that session already read some other items, the user must be able to see those *causally relevant* values too.


[https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/](https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/)
PRAM and Causal Consistency

- Combinations of previous session consistency guarantees
  - PRAM = MR + MW + RYW
  - Causal Consistency = PRAM + WFR
- All consistency level up to causal consistency can be guaranteed with **high availability**
- Example: Bolt-on causal consistency

Bounded Staleness

- Either **time-based**:
  
  **t-Visibility (Δ-atomicity)**: the inconsistency window comprises at most $t$ time units; that is, any value that is returned upon a read request was up to date $t$ time units ago.

- Or **version-based**:
  
  **k-Staleness**: the inconsistency window comprises at most $k$ versions; that is, lags at most $k$ versions behind the most recent version.

- Both are *not* achievable with high availability
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: HDD SSD RAM
SRRR SWRW SRRR SWRW SRRR SWRW

Speed, Cost: Low Performance, High Performance

RAM: In-Memory/ Caching
SSD: Update-In-Place
HDD: Append-Only I/O

Data In-Memory/ Caching
Logging
Append-Only I/O
Update-In-Place

Data
Persistent Storage

Log
Logging
NoSQL Storage Management
In a Nutshell

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

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

- Caching
- Logging
- Primary Storage

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

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

Speed, Cost
Size

Low Performance
High Performance
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

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>

<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>

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>

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>[104]</td>
</tr>
<tr>
<td>Blue</td>
<td>[188,192]</td>
</tr>
</tbody>
</table>
Local Secondary Indexing
Partitioning By Document

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

Indexing is always local to a partition.

Scatter-gather query pattern.

WHERE color=blue

Local Secondary Indexing
Partitioning By Document

WHERE color = blue

Implemented in:
- MongoDB
- Riak
- Cassandra
- Elasticsearch
- SolrCloud
- VoltDB

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>

<table>
<thead>
<tr>
<th>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td>[104]</td>
</tr>
<tr>
<td>Blue</td>
<td>[56, 188, 192]</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>

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

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>[104]</td>
</tr>
<tr>
<td>Blue</td>
<td>[56, 188, 192]</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>Term</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>[12, 77]</td>
</tr>
</tbody>
</table>

Targeted Query

WHERE color=blue

Consistent Index-maintenance requires distributed transaction.

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>

Partition II

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

Targeted Query

WHERE color=blue

Implemented in

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

Consistent index maintenance requires distributed transaction.

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

Scalable Real-Time Databases and Processing

- 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."
Dynamo (AP)

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

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

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

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

![Diagram showing the transfer of ranges between nodes A, B, and C.]

- B takes over two thirds of A
- C takes over one third of A
- Range transferred

The diagram illustrates how consistent hashing with virtual nodes can be used to balance loads among nodes. The range transferred between nodes is shown as an example of load balancing.
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

**Parameters**: \( R, W, N \)

**Example**: \( N=3 \), \( R=2 \), \( W=1 \)
Quorums

- **N** (Replicas), **W** (Write Acks), **R** (Read Acks)
  - \( R + W \leq N \Rightarrow \text{No guarantee} \\
  - \( R + W > N \Rightarrow \text{newest version included} \\

\[ N = 12, R = 3, W = 10 \]

\[ N = 12, R = 7, W = 6 \]
Writing

- **W** Servers have to acknowledge

\[ N=3 \quad R=2 \quad W=1 \]
Hinted Handoff

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

![Diagram showing network topology with nodes labeled A, B, C, D, E, and F, and connections indicating the hinted handoff process.]

N=3
R=2
W=1
Vector clocks

- Dynamo uses **Vector Clocks** for versioning

C. J. Fidge, Timestamps in message-passing systems that preserve the partial ordering (1988)
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
Versioning and Consistency

- \( R + W \leq N \Rightarrow \text{no consistency guarantee} \)
- \( R + W > N \Rightarrow \text{newest acked value included in reads} \)
- **Vector Clocks** used 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
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*)
Conflict Resolution

- The application merges data when writing (*Semantic Reconciliation*)
Conflict Resolution

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

- Every Second: Contact random server and compare
Merkle Trees: Anti-Entropy

- Every Second: Contact random server and compare
Merkle Trees: Anti-Entropy

- Every Second: Contact random server and compare
Merkle Trees: Anti-Entropy

- Every Second: Contact random server and compare
### Quorum

**Typical Configurations:**

<table>
<thead>
<tr>
<th>Type</th>
<th>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:

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**
- $S_1 = \emptyset$
- $S_1 = \{x\}$
- $S_1 = merge(\{x\}, \{y\})$
  - $= \{x, y\}$

**Node 2**
- $S_2 = \emptyset$
- $S_2 = \{y\}$
- $S_2 = merge(\{y\}, \{x\})$
  - $= \{x, y\}$

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
  - Option for **strongly consistent** buckets (experimental)
  - **Riak CS**: S3-like file storage, **Riak TS**: time-series database

<table>
<thead>
<tr>
<th>Riak</th>
<th>Model:</th>
<th>Key-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>License:</td>
<td>Apache 2</td>
</tr>
<tr>
<td></td>
<td>Written in:</td>
<td>Erlang und C</td>
</tr>
</tbody>
</table>

Consistency Level: **N, R, W, DW**

Storage Backend: Bit-Cask, Memory, LevelDB
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

<table>
<thead>
<tr>
<th>Term</th>
<th>Dokument</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>3,4,1</td>
</tr>
<tr>
<td>rabbit</td>
<td>2</td>
</tr>
</tbody>
</table>

Search Index

JS/Erlang Pre-Commit Hook

JS/Erlang Post-Commit Hook

Riak_search_kv_hook
Riak Map-Reduce

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

```
function(v) {
    var json = v.values[0].data;
    return [{count: json.stackoverflow_questions}];
}
```

POST /mapred

http://docs.basho.com/riak/latest/tutorials/querying/MapReduce/
function(mapped) {
  var sum = 0;
  for(var i in mapped) {
    sum += i.count;
  }
  return [{count : 0}];
}

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

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

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

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

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

Same Data Format
Riak Cloud Storage

Files

Amazon S3 API

1MB Chunks

Stanchion: Request Serializer
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

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

- **Replication**
  - Transaction Protocol
  - Sync. Replication
  - Async. Replication
  - Primary Copy
  - Update Anywhere

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

- **Query Processing**
  - Global Index
  - Local Index
  - Query Planning
  - Analytics
  - Materialized Views

- **Transaction Protocol**
- **Sync. Replication**
- **Async. Replication**
- **Primary Copy**
- **Update Anywhere**
- **Logging**
- **Update-in-Place**
- **Caching**
- **In-Memory**
- **Append-Only Storage**
- **Global Index**
- **Local Index**
- **Query Planning**
- **Analytics**
- **Materialized Views**
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 ≈ 20K LOC

Client → TCP Port 6379 → Event Loop → Redis Server

Plain Text Protocol

SET mykey hello

+OK

One Process/Thread

- Periodic
- After X Writes
- SAVE

Local Filesystem

RAM

Log Dump

AOF
RDB
Persistence

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

```config
set appendonly everysec
```

```fsync()
every second
```

Snapshot every 60s, if > 1000 keys changed

```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
- `synchronous_commit` off
- `fsync` false
- `pg_dump`

Redis:
- `appendfsync` always
- `appendfsync` everysec
- `appendfsync` no
- `save` oder `bgsave`
Master-Slave Replication

> SLAVEOF 192.168.1.1 6379
< +OK
# 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
  - **LLLEN**: 
  - **LPOP**: Blocks until element arrives
  - **BLPOP**: 
  - **RPUSH**: 
  - **LRANGE inbox 1 2**: 
  - **LREM inbox 0 3466**: 
  - **LINDEX inbox 2**: 
  - **RPOP**: 
  - **inbox 234 34 6 86 55**
Data Structures

- **Sets:**
  - `SADD` user:2:friends
  - `SREM` user:2:friends
  - `SCARD` user:2:friends
  - `SINTER` user:2:friends user:5:friends
  - `SINTERSTORE` common_friends
  - `SMEMBERS` 23 10 2 28 325 64 70
  - `SISMEMBER` false
  - `SREM` 23 10 2 28 325 64 70
  - `SADD` 23 10 2 28 325 64 70
  - `SRANDMEMBER` 23 10 2 28 325 64 70
  - `SMEMBERS` user:5:friends
  - `SINTER` user:2:friends user:5:friends
  - `SINTERSTORE` common_friends
  - `SMEMBERS` 23
  - `SISMEMBER` false
  - `SREM` 23 10 2 28 325 64 70
  - `SADD` 23 10 2 28 325 64 70
  - `SRANDMEMBER` 23 10 2 28 325 64 70
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
- $\text{insert}(\text{obj})$: add to set
- $\text{contains}(\text{obj})$: might give a false positive

![Diagram of Bloom filter operations: Insert $y$ and Query $x$.]
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);
    }
}

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

- If the Bloom filter uses 7 hashes: 7 roundtrips
- **Solution**: Redis Pipelining

![Diagram of clients and Redis with SETBIT commands]
Redis for distributed systems

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

![Diagram showing how Redis can be used to improve performance for legacy systems.](chart.png)

- **Slow Legacy System**
- **App Server**
- **Bloomfilter lookup:**
  - GETBIT, GETBIT...
- **Get Data From Legacy System**

---

**Diagram Elements:**
- **Hash**: MD5
- **k**: 7
- **m**: 80000
- **Bits**: 0100101011
Redis Bloom filters
Open Source

Library of different Bloom filters in Java with optional Redis-backing, counting and many hashing options.
Why is Redis so fast?

Pessimistic transactions are expensive

- 16.2% hand-coded optimizations
- 11.9% logging
- 16.3% locking
- 14.2% learning
- 34.6% buffer manager

Data in RAM

- 6.8% useful work

No Query Parsing
AOF
Operations are lock-free
Single-threading

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 Hash

```
-- 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
```

EVALSHA $hash 1 "mylock" "10"

Redis Cluster
Work-in-Progress

- **Idea:** Client-driven hash-based sharing (CRC32, „hash slots“)
- **Asynchronous** replication with failover (variant of Raft‘s leader election)
  - **Consistency:** not guaranteed, last failover wins
  - **Availability:** only on the majority partition
    → neither AP nor CP

- No multi-key operations
- Pinning via key: `{user1}.followers`
Comparable 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

RPUSHX user_id tweet

http://www.infoq.com/presentations/Real-Time-Delivery-Twitter
<table>
<thead>
<tr>
<th>Classification: Redis Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sharding</strong></td>
</tr>
<tr>
<td>Range-Sharding</td>
</tr>
<tr>
<td>Hash-Sharding</td>
</tr>
<tr>
<td>Entity-Group Sharding</td>
</tr>
<tr>
<td>Consistent Hashing</td>
</tr>
<tr>
<td>Shared Disk</td>
</tr>
<tr>
<td><strong>Replication</strong></td>
</tr>
<tr>
<td>Transaction Protocol</td>
</tr>
<tr>
<td>Sync. Replication</td>
</tr>
<tr>
<td>Async. Replication</td>
</tr>
<tr>
<td>Primary Copy</td>
</tr>
<tr>
<td>Update Anywhere</td>
</tr>
<tr>
<td><strong>Storage Management</strong></td>
</tr>
<tr>
<td>Logging</td>
</tr>
<tr>
<td>Update-in-Place</td>
</tr>
<tr>
<td>Caching</td>
</tr>
<tr>
<td>In-Memory</td>
</tr>
<tr>
<td>Append-Only Storage</td>
</tr>
<tr>
<td><strong>Query Processing</strong></td>
</tr>
<tr>
<td>Global Index</td>
</tr>
<tr>
<td>Local Index</td>
</tr>
<tr>
<td>Query Planning</td>
</tr>
<tr>
<td>Analytics</td>
</tr>
<tr>
<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"): 

Column-Family: **contents**

- `com.cnn.www`
- `content : "<html>..."`
- `cnnsi.com : "CNN"`
- `t3`, `t5`, `t6`

Column-Family: **anchor**

- `com.cnn.www`
- `my.look.ca : "CNN.com"`
Wide-Column Data Modelling

- Storage of crawled web-sites ("Webtable"): 

1. Dimension: Row Key
- `com.cnn.www`

2. Dimension: CF:Column
- `content`: "<html>...

3. Dimension: Timestamp
- `t3`
- `t5`
- `t6`

Column-Family: `contents`

Column-Family: `anchor`
- `cnnsi.com`: "CNN"
- `my.look.ca`: "CNN.com"

Sparse
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>I-M</td>
<td>M-T</td>
<td></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>

Master

Controls Ranges, Splits, Rebalancing
Architecture

Master

Tablet Server

Chubby

GFS

SSTables

Commit
Log
Architecture

ACLs, Garbage Collection, Rebalancing

Master Lock, Root Metadata Tablet

Master

Chubby

Tablet Server

Stores Ranges, Answers client requests

Stores data and commit log

GFS

SSTables

Commit Log

Stores data and commit log
**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: Sorted-String Tables**

- *Key/Value* Sorted String Table
- *Variable Length* Key/Value
- **Row-Key**

![Diagram of Sorted-String Tables](image)
Storage: Optimization

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

![Diagram of storage optimization]

1. **Client** performs a **Write(x)** operation to update the **Memtable**.
2. **Read(x)** operations are processed through **Bloom filters** in **Main Memory**.
3. **Hit** cases are served from **Main Memory**.
4. **Miss** cases are directed to **Disk** for **SSTables**.
5. **Periodic Compaction** maintains efficient disk access.
6. **Periodic Flush** updates **SSTables** to **Disk**.
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>
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>

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: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></th>
<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></td>
<td>10</td>
<td>7</td>
<td></td>
<td>100</td>
<td>65</td>
<td></td>
<td></td>
<td>567</td>
<td></td>
</tr>
</tbody>
</table>

CF:Daily

CF:Monthly

CF:All

TTL – automatic deletion of old rows

Atomic HBase Counter

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>dcbее495-6d5e-6ed48124632c</td>
<td>1307103848</td>
<td>&quot;Hi, how are ...&quot;</td>
</tr>
</tbody>
</table>

### VS

<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-dcbее495-6d5e-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:

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

```
> whirr launch-cluster --config
hbase.properties
```

Login, cluster size, etc.

```java
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) { }```

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) → 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 → implicit schema (key design) needs to be carefully planned
- **HBase**: very literal open-source BigTable implementation
Classification: HBase Techniques

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

- **Replication**
  - Transaction Protocol
  - Sync. Replication
  - Async. Replication
  - Primary Copy
  - Update Anywhere

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

- **Query Processing**
  - Global Index
  - Local Index
  - Query Planning
  - Analytics
  - Materialized Views
**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_keyspace()

get_slice()

TCP Cluster Messages

Column Family Store

Row Cache

Local Filesystem

Key Cache

MemTable

Storage Proxy

Hashing:

MD5(key)

Random Partitioner

Order Preservering Partitioner

Snitch: Rack, Datacenter, EC2 Region Information

Hashing:

MD5(key)
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

Stateful Communication

Replication, Gossip, etc.

TCP Cluster Messages

Stores Rows

Stores Primary Key Index (Seek Position)

Stores SSTables and Commit Log

Hashing:

MD5(key)

Random Partitioner

Order Preservering Partitioner

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

- **f82831...**: *title*: Andante  
- **144052...**: *title*: Jailhouse Rock

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

- **album**: New World Symphony
- **artist**: Antonin Dvorak
- **artist**: Elvis Presley  

**Type validated by Validation Class UTFType**

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

- Enables Scans despite *Random Partitioner*

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

```cql
SELECT * FROM playlists  
WHERE id = 23423  
ORDER BY song_order DESC  
LIMIT 50;
```

<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>
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

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

## Replication
- Transaction Protocol
- Sync. Replication
- Async. Replication
- Primary Copy
- Update Anywhere

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

## Query Processing
- Global Index
- Local Index
- Query Planning
- Analytics
- Materialized Views
From humongous ≈ gigantic
Schema-free document database with tunable consistency
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

```bash
> 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" ]
}
```

Properties

Arrays, Nesting allowed
Data Modelling

Movie
  title
  year
  rating
  director

Genre

Actor

Tweet
text coordinates
retweets

User
  name
  location

1
n
n
1
1
Data Modelling

```json
{
    "_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")
    }]
}
```

Movie Document
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")
            } ]
}
```

**Movie Document**

**Denormalisation** instead of joins

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

**Schemafreeness**: Attributes per document

**Unit of atomicity**: document

**Principles**
Sharding und 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*

- Receives all **writes**
- Replicates asynchronously

**Client**
- Connects to mongos

**MongoDB Cluster**
- Master
- Slave
- Replica Set
MongoDB Example App

Twitter Firehose

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

REST API (Jetty)

GET
MongoDB
Tweets
Streaming
GridFS
Tweet Map
Searching
JSON
Queries

Browser

Movies
Tweets
Browser
HTTP

saveTweet()
getTaggedTweets()
getByGenre()
searchByPrefix()
The Movie mApp

Unveiling the geographic patterns underlying tweets about movies.

Show Mongo at http://127.0.0.1:28017/

Geotagged Tweets about Movies

Note: Click on markers to show tweets and click on the map to show coordinates and its 1000km radius.

Movie: Family Guy
User: david hourigan
Tweet: That 70's show, American dad, family guy, courage the cowardly dog, how I met your mother.. This is too good
DBObject query = new BasicDBObject("tweets.coordinates",
    new BasicDBObject("$exists", true));

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

Or in JavaScript:

db.movies.find({tweets.coordinates : { "$exists" : 1}})
DBObject query = new BasicDBObject("tweets.coordinates",
    new BasicDBObject("$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.

Show Mongo at http://127.0.0.1:28017/

```javascript
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.\texttt{ensureIndex}({\texttt{title} : 1})

\begin{verbatim}
db.movies.find({title : /\texttt{^Incep}\texttt{/}}).limit(10)
\end{verbatim}

\textit{Index usage:}
\begin{verbatim}
db.movies.find({title : /\texttt{^Incep}\texttt{/}}).explain().millis = 0
db.movies.find({title : /\texttt{^Incep}\texttt{i}/}).explain().millis = 340
\end{verbatim}
db.movies.update({}_id: id), {
  "$set": {
    "comment": c
  }
})
or:
db.movies.save(changed_movie);

@TRIXIA: #nowwatching.Inception

@青峰 大輝。: So, I finally finished Vampire Knight, this beautiful manga I followed since its inception. It ends beautifully and oddly I like Kaname.

One of the best movies, that
```
fs = new GridFs(db);
fs.createFile(inputStream).save();
```
Query Tweets

Query: Get Tweets Near: lat,lng, radius-in-km
Parameter: 51.54155217692421, 10.406249463558197, 1000
Result Limit: 10

db.tweets.ensureIndex({coordinates: "2dsphere"})
db.tweets.find({"$near": {"$geometry": ...}})

Geospatial Queries:
• Distance
• Intersection
• Inclusion
Query Tweets

Query: Indexed Fulltext Search on Tweets
Parameter: StAr trek
Result Limit: 100

Show: 25

Filter search results:

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

db.tweets.runCommand( "text", { search: "StAr trek" } )
Analytic Capabilities

- Aggregation Pipeline Framework:

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

- Alternative: JavaScript MapReduce
Sharding

- Range-based:
  - Key Space for x
  - In the 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/
Sharding

- Splitting:
  - Split chunks that are too large

- Migration:
  - Mongos Load Balancer triggers rebalancing

[Diagram showing split chunks and migration between shards]

docs.mongodb.org/manual/core/sharding-introduction/
<table>
<thead>
<tr>
<th>Classification: MongoDB Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sharding</strong></td>
</tr>
<tr>
<td>Range-Sharding</td>
</tr>
<tr>
<td>Hash-Sharding</td>
</tr>
<tr>
<td>Entity-Group Sharding</td>
</tr>
<tr>
<td>Consistent Hashing</td>
</tr>
<tr>
<td>Shared Disk</td>
</tr>
<tr>
<td><strong>Replication</strong></td>
</tr>
<tr>
<td>Transaction Protocol</td>
</tr>
<tr>
<td>Sync. Replication</td>
</tr>
<tr>
<td>Async. Replication</td>
</tr>
<tr>
<td>Primary Copy</td>
</tr>
<tr>
<td>Update Anywhere</td>
</tr>
<tr>
<td><strong>Storage Management</strong></td>
</tr>
<tr>
<td>Logging</td>
</tr>
<tr>
<td>Update-in-Place</td>
</tr>
<tr>
<td>Caching</td>
</tr>
<tr>
<td>In-Memory</td>
</tr>
<tr>
<td>Append-Only Storage</td>
</tr>
<tr>
<td><strong>Query Processing</strong></td>
</tr>
<tr>
<td>Global Index</td>
</tr>
<tr>
<td>Local Index</td>
</tr>
<tr>
<td>Query Planning</td>
</tr>
<tr>
<td>Analytics</td>
</tr>
<tr>
<td>Materialized Views</td>
</tr>
</tbody>
</table>
Other Systems
Graph databases

- **Neo4j** (ACID, replicated, Query-language)
- **HypergraphDB** (directed Hypergraph, BerkleyDB-based)
- **Titan** (distributed, Cassandra-based)
- **ArangoDB, OrientDB** („multi-model“)
- **SparkleDB** (RDF-Store, SPARQL)
- **InfinityDB** (embeddable)
- **InfiniteGraph** (distributed, low-level API, Objectivity-based)
Other Systems

Key-Value Stores

- **Aerospike** (SSD-optimized)
- **Voldemort** (Dynamo-style)
- **Memcache** (in-memory cache)
- **LevelDB** (embeddable, LSM-based)
- **RocksDB** (LevelDB-Fork with Transactions and Column Families)
- **HyperDex** (Searchable, Hyperspace-Hashing, Transactions)
- **Oracle NoSQL database** (distributed frontend for BerkleyDB)
- **HazelCast** (in-memory data-grid based on Java Collections)
- **FoundationDB** (ACID through Paxos)
Other Systems

Document Stores

- **CouchDB** (Multi-Master, lazy synchronization)
- **CouchBase** (distributed Memcache, N1QL~SQL, MR-Views)
- **RavenDB** (single node, SI transactions)
- **RethinkDB** (distributed CP, MVCC, joins, aggregates, real-time)
- **MarkLogic** (XML, distributed 2PC-ACID)
- **ElasticSearch** (full-text search, scalable, unclear consistency)
- **Solr** (full-text search)
- **Azure DocumentDB** (cloud-only, ACID, WAS-based)
Other Systems
Wide-Column Stores

- **Accumulo** (BigTable-style, cell-level security)
- **HyperTable** (BigTable-style, written in C++)
Other Systems

NewSQL Systems

- **CockroachDB** (Spanner-like, SQL, no joins, transactions)
- **Crate** (ElasticSearch-based, SQL, no transaction guarantees)
- **VoltDB** (HStore, ACID, in-memory, uses stored procedures)
- **Calvin** (log- & Paxos-based ACID transactions)
- **FaunaDB** (based on Calvin design, by Twitter engineers)
- **Google F1** (based on Spanner, SQL)
- **Microsoft Cloud SQL Server** (distributed CP, MSSQL-comp.)
- **MySQL Cluster, Galera Cluster, Percona XtraDB Cluster** (distributed storage engine for MySQL)
Open Research Questions
For Scalable Data Management

- **Service-Level Agreements**
  - How can SLAs be guaranteed in a virtualized, multi-tenant cloud environment?

- **Consistency**
  - Which consistency guarantees can be provided in a geo-replicated system without sacrificing availability?

- **Performance & Latency**
  - How can a database deliver low latency in face of distributed storage and application tiers?

- **Transactions**
  - Can ACID transactions be aligned with NoSQL and scalability?
**Definition:** A transaction is a sequence of operations transforming the database from one consistent state to another.

**Isolation Levels:**
1. Serializability
2. Snapshot Isolation
3. Read-Committed
4. Read-Atomic
5. ...
Distributed Transactions

General Processing

Commit Protocol

Concurrency Control

Replication

Replicas

Shard

Concurrency Control

Replication

Replicas

Shard

Concurrency Control

Replication

Replicas

Shard
Distributed Transactions
General Processing

Commit Protocol is not available

Concurrency Control

Replication

Shard

Needs to ensure globally correct isolation

Strong Consistency – needed by Concurrency Control
### Distributed Transactions

In NoSQL Systems – An Overview

<table>
<thead>
<tr>
<th>System</th>
<th>Concurrency Control</th>
<th>Isolation</th>
<th>Granularity</th>
<th>Commit Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Megastore</td>
<td>OCC</td>
<td>SR</td>
<td>Entity Group</td>
<td>Local</td>
</tr>
<tr>
<td>G-Store</td>
<td>OCC</td>
<td>SR</td>
<td>Entity Group</td>
<td>Local</td>
</tr>
<tr>
<td>ElasTras</td>
<td>PCC</td>
<td>SR</td>
<td>Entity Group</td>
<td>Local</td>
</tr>
<tr>
<td>Cloud SQL Server</td>
<td>PCC</td>
<td>SR</td>
<td>Entity Group</td>
<td>Local</td>
</tr>
<tr>
<td>Spanner / F1</td>
<td>PCC / OCC</td>
<td>SR / SI</td>
<td>Multi-Shard</td>
<td>2PC</td>
</tr>
<tr>
<td>Percolator</td>
<td>OCC</td>
<td>SI</td>
<td>Multi-Shard</td>
<td>2PC</td>
</tr>
<tr>
<td>MDCC</td>
<td>OCC</td>
<td>RC</td>
<td>Multi-Shard</td>
<td>Custom – 2PC like</td>
</tr>
<tr>
<td>CloudTPS</td>
<td>TO</td>
<td>SR</td>
<td>Multi-Shard</td>
<td>2PC</td>
</tr>
<tr>
<td>Cherry Garcia</td>
<td>OCC</td>
<td>SI</td>
<td>Multi-Shard</td>
<td>Client Coordinated</td>
</tr>
<tr>
<td>Omid</td>
<td>MVCC</td>
<td>SI</td>
<td>Multi-Shard</td>
<td>Local</td>
</tr>
<tr>
<td>FaRMville</td>
<td>OCC</td>
<td>SR</td>
<td>Multi-Shard</td>
<td>Local</td>
</tr>
<tr>
<td>H-Store/VoltDB</td>
<td>Deterministic CC</td>
<td>SR</td>
<td>Multi-Shard</td>
<td>2PC</td>
</tr>
<tr>
<td>Calvin</td>
<td>Deterministic CC</td>
<td>SR</td>
<td>Multi-Shard</td>
<td>Custom</td>
</tr>
<tr>
<td>RAMP</td>
<td>Custom</td>
<td>Read-Atomic</td>
<td>Multi-Shard</td>
<td>Custom</td>
</tr>
</tbody>
</table>
Distributed Transactions

Megastore

- Synchronous Paxos-based replication
- Fine-grained partitions (entity groups)
- Based on BigTable
- Local commit protocol, optimistic concurrency control

EG: User + n Photos
- Unit of ACID transactions/consistency
- Local commit protocol, optimistic concurrency control
Distributed Transactions

Megastore

Spanner

Idea:
- Auto-sharded Entity Groups
- Paxos-replication per shard

Transactions:
- Multi-shard transactions
- SI using TrueTime API (GPA and atomic clocks)
- SR based on 2PL and 2PC
- Core of F1 powering ad business

Percolator

Idea:
- Indexing and transactions based on BigTable

Implementation:
- Metadata columns to coordinate transactions
- Client-coordinated 2PC
- Used for search index (not OLTP)


Distributed Transactions
MDCC – Multi Datacenter Concurrency Control

Properties:
- Read Committed Isolation
- Geo Replication
- Optimistic Commit

Paxos Instance

Record-Master (v)

Replicas

T1 = \{v \rightarrow v',
     u \rightarrow u'\}

App-Server (Coordinator)

Record-Master (u)

Replicas
Distributed Transactions

RAMP – Read Atomic Multi Partition Transactions

Properties:
- Read Atomic Isolation
- Synchronization Independence
- Partition Independence
- Guaranteed Commit

Fractured Read:

1. read objects
2. validate
3. load other version
Distributed Transactions in the Cloud

The Latency Problem

Interactive Transactions:

Optimistic Concurrency Control
Optimistic Concurrency Control

The Abort Rate Problem

- 10,000 objects
- 20 writes per second
- 95% reads
Optimistic Concurrency Control
The Abort Rate Problem

- 10,000 objects
- 20 writes per second
- 95% reads
Distributed Cache-Aware Transaction
Scalable ACID Transactions

Solution: **Conflict-Avoidant Optimistic Transactions**
- Cached reads → Shorter transaction duration → less aborts
- Bloom Filter to identify outdated cache entries
Distributed Cache-Aware Transaction
Speed Evaluation

- 10,000 objects
- 20 writes per second
- 95% reads

➤ 16 times speedup
Distributed Cache-Aware Transaction Abort Rate Evaluation

- 10.000 objects
- 20 writes per second
- 95% reads
  - 16 times speedup
  - Significantly less aborts
  - Highly reduced runtime of retried transactions
Distributed Cache-Aware Transaction
Combined with RAMP Transactions

1. read objects
2. validate
3. load other version
Selected Research Challenges

Encrypted Databases

- Example: **CryptDB**
- **Idea:** Only decrypt as much as necessary

**SQL-Proxy**

Encrypts and decrypts, rewrites queries

**RDBMS**

- **RND:** no functionality
- **DET:** equality selection
- **JOIN:** equality join
- **Search**
  - text value
- **Onion Search**
- **OPE:** order
- **OPE-JOIN:** range join
- **HOM:** add
- **Onion Ord**
- **Onion Add**
- **Onion Eq**

*Note: Onion Eq represents a specific functionality for equality comparisons.*
Selected Research Challenges

Encrypted Databases

- **Example:** CryptDB
- **Idea:** Only decrypt as much as necessary

**SQL-Proxy**

Encrypts and decrypts, rewrites queries

**RDBMS**

**RND:** no functionality

**DET:** equality selection

**JOIN:** equality join

**any value**

**Onion Eq**

**RND:** no functionality

**OPE:** order

**OPE-JOIN:** range join

**any value**

**Onion Ord**

**SEARCH**

- **text value**

- **Onion Search**

- **HOM:** add

- **int value**

- **Onion Add**

**Relational Cloud**

**DBaaS Architecture:**
- Encrypted with CryptDB
- **Multi-Tenancy** through live migration
- Workload-aware **partitioning** (graph-based)

Selected Research Challenges

Encrypted Databases

- Example: CryptDB
- Idea: Only decrypt as much as necessary

SQL-Proxy

Encrypts and decrypts

RDBMS

- Early approach
- Not adopted in practice, yet

DBaaS Architecture:
- Encrypted with CryptDB
- Multi-Tenancy through live migration
- Workload-aware partitioning (graph-based)

Dream solution: Full Homomorphic Encryption

## Research Challenges

### Transactions and Scalable Consistency

<table>
<thead>
<tr>
<th></th>
<th>Consistency</th>
<th>Transactional Unit</th>
<th>Commit Latency</th>
<th>Data Loss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamo</td>
<td>Eventual</td>
<td>None</td>
<td>1 RT</td>
<td>-</td>
</tr>
<tr>
<td>Yahoo PNuts</td>
<td>Timeline per key</td>
<td>Single Key</td>
<td>1 RT</td>
<td>possible</td>
</tr>
<tr>
<td>COPS</td>
<td>Causality</td>
<td>Multi-Record</td>
<td>1 RT</td>
<td>possible</td>
</tr>
<tr>
<td>MySQL (async)</td>
<td>Serializable</td>
<td>Static Partition</td>
<td>1 RT</td>
<td>possible</td>
</tr>
<tr>
<td>Megastore</td>
<td>Serializable</td>
<td>Static Partition</td>
<td>2 RT</td>
<td>-</td>
</tr>
<tr>
<td>Spanner/F1</td>
<td>Snapshot Isolation</td>
<td>Partition</td>
<td>2 RT</td>
<td>-</td>
</tr>
<tr>
<td>MDCC</td>
<td>Read-Commited</td>
<td>Multi-Record</td>
<td>1 RT</td>
<td>-</td>
</tr>
</tbody>
</table>
### Research Challenges

Transactions and Scalable Consistency

<table>
<thead>
<tr>
<th>System</th>
<th>Consistency</th>
<th>Transactions</th>
<th>Latency</th>
<th>Data Loss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamo</td>
<td>Eventual</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yahoo PNuts</td>
<td>Timeline per Key</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPS</td>
<td>Causality</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL (async)</td>
<td>Serializable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Megastore</td>
<td>Serializable</td>
<td>Static Partition</td>
<td>2 RT</td>
<td></td>
</tr>
<tr>
<td>Spanner/F1</td>
<td>Snapshot Isolation</td>
<td>Partition</td>
<td>2 RT</td>
<td>-</td>
</tr>
<tr>
<td>MDCC</td>
<td>Read-Committed</td>
<td>Multi-Record</td>
<td>1 RT</td>
<td>-</td>
</tr>
</tbody>
</table>

**Google’s F1**

**Idea:**
- Consistent multi-data center replication with SQL and ACID transaction

**Implementation:**
- Hierarchical schema (Protobuf)
- Spanner + Indexing + Lazy Schema Updates
- Optimistic and Pessimistic Transactions

---

## Research Challenges

### Transactions and Scalable Consistency

<table>
<thead>
<tr>
<th>Database</th>
<th>Consistency Model</th>
<th>Implementation Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamo</td>
<td>Eventual</td>
<td>Consistent multi-data center replication with SQL and ACID transaction</td>
</tr>
<tr>
<td>Yahoo PNuts</td>
<td>Timeline per key</td>
<td>Hierarchical schema (Protobuf)</td>
</tr>
<tr>
<td>COPS</td>
<td>Causality</td>
<td>Spanner + Indexing + Lazy Schema Updates</td>
</tr>
<tr>
<td>MySQL (async)</td>
<td>Serializable</td>
<td>Optimistic and Pessimistic Transactions</td>
</tr>
<tr>
<td>Megastore</td>
<td>Serializable</td>
<td>Serial Partition 2 RT</td>
</tr>
<tr>
<td>Spanner/F1</td>
<td>Snapshot isolation</td>
<td>Partition 2 RT</td>
</tr>
<tr>
<td>MDCC</td>
<td>Read-Commited</td>
<td>Consistent Multi-DC replication</td>
</tr>
</tbody>
</table>

### Google’s F1

**Idea:**
- Consistent multi-data center replication with SQL and ACID transaction

**Implementation:**
- Hierarchical schema (Protobuf)
- Spanner + Indexing + Lazy Schema Updates
- Optimistic and Pessimistic Transactions

---

Research Challenges
NoSQL Benchmarking

- **YCSB** *(Yahoo Cloud Serving Benchmark)*

### Workload Generator
- **Runtime Parameters:**
  - DB host name, threads, etc.

- **Workload:**
  1. Operation Mix
  2. Record Size
  3. Popularity Distribution

### Client
- Threads
- Stats

### Pluggable DB interface

### Data Store
- DB protocol
- Read()
- Insert()
- Update()
- Delete()
- Scan()
Research Challenges
NoSQL Benchmarking

- **YCSB** (**Y**ahoo **C**loud **S**erving **B**enchmark)

<table>
<thead>
<tr>
<th>Workload</th>
<th>Operation Mix</th>
<th>Distribution</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A – Update Heavy</td>
<td>Read: 50% Update: 50%</td>
<td>Zipfian</td>
<td>Session Store</td>
</tr>
<tr>
<td>B – Read Heavy</td>
<td>Read: 95% Update: 5%</td>
<td>Zipfian</td>
<td>Photo Tagging</td>
</tr>
<tr>
<td>C – Read Only</td>
<td>Read: 100%</td>
<td>Zipfian</td>
<td>User Profile Cache</td>
</tr>
<tr>
<td>D – Read Latest</td>
<td>Read: 95% Insert: 5%</td>
<td>Latest</td>
<td>User Status Updates</td>
</tr>
<tr>
<td>E – Short Ranges</td>
<td>Scan: 95% Insert: 5%</td>
<td>Zipfian/Uniform</td>
<td>Threaded Conversations</td>
</tr>
</tbody>
</table>

3. Popularity Distribution
Research Challenges
NoSQL Benchmarking

Example Result
(Read Heavy):

- Cassandra
- HBase
- PNUTS
- MySQL

![Graph showing read latency vs throughput for different NoSQL databases](image-url)
Research Challanges
NoSQL Benchmarking

- Example Result
  (Read Heavy):

  ![Graph showing latency and throughput]

  **Weaknesses:**
  - Single client can be a bottleneck
  - No consistency & availability measurement
Research Challenges

NoSQL Benchmarking

YCSB++

- Clients coordinate through Zookeeper
- Simple Read-After-Write Checks
- Evaluation: Hbase & Accumulo


Weaknesses:
- Single client can be a bottleneck
- No consistency & availability measurement
Research Challenges
NoSQL Benchmarking

**YCSB++**
- Clients coordinate through Zookeeper
- Simple Read-After-Write Checks
- Evaluation: Hbase & Accumulo

*S. Patil, M. Polte, et al., "Ycsb++: benchmarking and performance debugging advanced features in scalable table stores", SOCC 2011*

**YCSB+T**
- **New workload:** Transactional Bank Account
- Simple anomaly detection for Lost Updates
- No comparison of systems

*A. Dey et al. “YCSB+T: Benchmarking Web-Scale Transactional Databases”, CloudDB 2014*

**Weaknesses:**
- Single client can be a bottleneck
- No consistency & availability measurement
- No Transaction Support

No specific application ➔ CloudStone, CARE, TPC extensions?
How can the choices for an appropriate system be narrowed down?
NoSQL Decision Tree

Access

Fast Lookups

Volume

RAM

Unbounded

CAP

AP

CP

Redis

Memcache

Cassandra

Riak

Voldemort

Aerospike

HBase

MongoDB

CouchBase

DynamoDB

RDBMS

Neo4j

RavenDB

MarkLogic

RDBMS

Neo4j

RavenDB

MarkLogic

OLTP

Website

Social Network

Big Data

Complex Queries

Volume

HDD-Size

Consistency

ACID

Availability

Ad-hoc

Query Pattern

Analytics

Example Applications

Cache

Shopping-basket

Order History

OLTP

Website

Social Network

Big Data

NoSQL Decision Tree

Access

Fast Lookups

Volume

RAM

Unbounded

CAP

AP CP

Redis

Memcache

Cassandra

Riak

Voldemort

Aerospike

HBase

MongoDB

CouchBase

DynamoDB

RDBMS

Neo4j

RavenDB

MarkLogic

RDBMS

Neo4j

RavenDB

MarkLogic

OLTP

Website

Social Network

Big Data
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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redis</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>x</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>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>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</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>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>HBase</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>Riak</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>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</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>
</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>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</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>
</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>
</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>
</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></td>
<td></td>
<td></td>
<td>x</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></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
Future Work
Online Collaborative Decision Support

- Select **Requirements** in Web GUI:
  - **Read Scalability**
  - **Conditional Writes**
  - **Consistent**

- System makes **suggestions** based on data from **practitioners, vendors and automated benchmarks**:
  - **redis**: 4/5, 4/5, 3/5
  - **mongoDB**: 4/5, 5/5, 5/5
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

  - **Decision Tree**

  - **Functional Requirements**
  - **Non-functional Requirements**


Summary

- Current NoSQL systems very good at scaling:
  - Data storage
  - Simple retrieval
- But how to handle real-time queries?
Real-Time Data Management in Research and Industry

Wolfram Wingerath
wingerath@informatik.uni-hamburg.de
March 7th, 2017, Stuttgart
About me
Wolfram Wingerath

- PhD student at the University of Hamburg, Information Systems group
- Researching distributed data management:
  - NoSQL database systems
  - Scalable stream processing
  - Scalable real-time queries
  - NoSQL benchmarking
# Outline

<table>
<thead>
<tr>
<th>Scalable Data Processing: Big Data in Motion</th>
<th>• Data Processing Pipelines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream Processors: Side-by-Side Comparison</td>
<td>• Why Data Processing Frameworks?</td>
</tr>
<tr>
<td>Real-Time Databases: Push-Based Data Access</td>
<td>• Overview: Processing Landscape</td>
</tr>
<tr>
<td>Current Research: Opt-In Push-Based Access</td>
<td>• Batch Processing</td>
</tr>
<tr>
<td></td>
<td>• Stream Processing</td>
</tr>
<tr>
<td></td>
<td>• Lambda Architecture</td>
</tr>
<tr>
<td></td>
<td>• Kappa Architecture</td>
</tr>
<tr>
<td></td>
<td>• Wrap-Up</td>
</tr>
</tbody>
</table>
Scalable Data Processing
Today’s topic!
Data Processing Frameworks
Scale-Out Made Feasible

Data processing frameworks **hide some complexities of scaling**, e.g.:
- **Deployment**: code distribution, starting/stopping work
- **Monitoring**: health checks, application stats
- **Scheduling**: assigning work to machines, rebalancing
- **Fault-tolerance**: restarting failed workers, rescheduling failed work
Big Data Processing Frameworks

What are your options?

- IBM InfoSphere Streams
- STORM Trident
- kafka streams
- Samza
- Apache Storm
- Google Dataflow
- Apache Apex
- Apache Hadoop
- Apache Spark
- Amazon Elastic MapReduce

low latency

high throughput
Batch Processing

„Volume“

- Cost-effective
- Efficient
- Easy to reason about: operating on complete data

But:
- **High latency**: jobs periodically (e.g. during night times)
Stream Processing „Velocity“

- Low end-to-end latency
- Challenges:
  - **Long-running jobs**: no downtime allowed
  - **Asynchronism**: data may arrive delayed or out-of-order
  - **Incomplete input**: algorithms operate on partial data
  - More: fault-tolerance, state management, guarantees, ...

---

*Streaming (e.g. Kafka, Redis)*  →  *Real-Time (e.g. Storm)*  →  *Serving*  →  *Application*
Lambda Architecture

\[ \text{Batch}(D_{\text{old}}) + \text{Stream}(D_{\Delta \text{now}}) \approx \text{Batch}(D_{\text{all}}) \]

- **Fast** output (real-time)
- **Data retention + reprocessing** (batch)
  \[ \rightarrow \text{“eventually accurate“} \] merged views of real-time and batch layer
- Typical setups: Hadoop + Storm (→ Summingbird), Spark, Flink
- **High complexity**: synchronizing 2 code bases, managing 2 deployments

---

Nathan Marz, *How to beat the CAP theorem* (2011)
Kappa Architecture

\[ \text{Stream}(D_{all}) = \text{Batch}(D_{all}) \]

**Simpler** than Lambda Architecture

- Data retention for relevant portion of history
- Reasons to forgo Kappa:
  - Legacy batch system that is not easily migrated
  - Special tools only available for a particular batch processor
  - Purely incremental algorithms

Jay Kreps, *Questioning the Lambda Architecture* (2014)

[https://www.oreilly.com/ideas/questioning-the-lambda-architecture](https://www.oreilly.com/ideas/questioning-the-lambda-architecture)
Wrap-up: Data Processing

- Processing frameworks abstract from scaling issues
- Two paradigms:
  - **Batch processing:**
    - easy to reason about
    - extremely efficient
    - Huge input-output latency
  - **Stream processing:**
    - Quick results
    - purely incremental
    - potentially complex to handle
- **Lambda Architecture:** batch + stream processing
- **Kappa Architecture:** stream-only processing
Outline

- Scalable Data Processing: Big Data in Motion
- Stream Processors: Side-by-Side Comparison
- Real-Time Databases: Push-Based Data Access
- Current Research: Opt-In Push-Based Access
- Processing Models: Stream ↔ Batch
- Stream Processing Frameworks:
  - Storm
  - Trident
  - Samza
  - Flink
  - Other Systems
- Side-By-Side Comparison
- Discussion
Stream Processors
Processing Models
Batch vs. Micro-Batch vs. Stream

- **stream**
  - Flink
  - Storm
  - Samza

- **micro-batch**
  - Storm Trident
  - Apache Spark Streaming

- **batch**
  - Apache Hadoop
  - Amazon Elastic MapReduce

**Axes:**
- **low latency**
- **high throughput**
Overview:

- „Hadoop of real-time“: abstract programming model (cf. MapReduce)
- First production-ready, well-adopted stream processing framework
- Compatible: native Java API, Thrift-compatible, distributed RPC
- Low-level interface: no primitives for joins or aggregations
- Native stream processor: end-to-end latency < 50 ms feasible
- Many big users: Twitter, Yahoo!, Spotify, Baidu, Alibaba, ...

History:

- 2010: start of development at BackType (acquired by twitter)
- 2011: open-sourced
- 2014: Apache top-level project
Dataflow

Directed Acyclic Graphs (DAG):

- **Spouts**: pull data into the topology
- **Bolts**: do the processing, emit data
- Asynchronous
- Lineage can be tracked for each tuple
  \[\rightarrow\] At-least-once delivery roughly doubles messaging overhead
Parallelism

Illustration taken from:
State Management
Recover State on Failure

• **In-memory or Redis**-backed reliable state
• *Synchronous state communication* on the critical path
→ infeasible for large state
Back Pressure
Flow Control Through Watermarks

1. An executor finds recv queue full (> high watermark), so notifies backpressure thread
2. Backpressure thread adds worker1 entry under the topo1 dir on ZooKeeper
3. The watch on topo1 notifies all workers that this topology is throttled.
4. Spout slows down its tuple sending speed.

Illustration taken from: https://issues.apache.org/jira/browse/STORM-886 (2017-02-21)
**Back Pressure**

**Throttling Ingestion on Overload**

1. too many tuples
   → 2. tuples time out and fail

3. tuples get replayed

**Approach:** monitoring bolts’ inbound buffer

1. Exceeding **high watermark** → throttle!
2. Falling below **low watermark** → full power!
Trident
Stateful Stream Joining on Storm

Overview:
- Abstraction layer on top of Storm
- Released in 2012 (Storm 0.8.0)
- Micro-batching
- New features:
  - Stateful exactly-once processing
  - High-level API: aggregations & joins
  - Strong ordering
Trident
Exactly-Once Delivery Configs

Can *block* the topology when failed batch cannot be replayed

Does not scale:
• Requires before- *and* after-images
• Batches are written in order

Illustration taken from: [http://storm.apache.org/releases/1.0.2/Trident-state.html](http://storm.apache.org/releases/1.0.2/Trident-state.html) (2017-02-26)
Samza

Overview:
- Co-developed with Kafka → Kappa Architecture
- **Simple**: only single-step jobs
- Local state
- Native stream processor: low latency
- **Users**: LinkedIn, Uber, Netflix, TripAdvisor, Optimizely, ...

History:
- Developed at LinkedIn
- 2013: open-source (Apache Incubator)
- 2015: Apache top-level project

Illustration taken from: Jay Kreps, *Questioning the Lambda Architecture* (2014)
https://www.oreilly.com/ideas/questioning-the-lambda-architecture (2017-03-02)
Dataflow
Simple By Design

- **Job**: a single processing step (≈ Storm bolt)
  → Robust
  → But: complex applications require several jobs
- **Task**: a job instance (determines job parallelism)
- **Message**: a single data item

- **Output is always persisted** in Kafka
  → Jobs can easily share data
  → Buffering (no back pressure!)
  → But: Increased latency
- **Ordering** within partitions
- Task = Kafka partitions: not-elastic on purpose

Advantages of local state:

- **Buffering**
  - No back pressure
  - At-least-once delivery
  - Straightforward recovery (see next slide)
- **Fast lookups**
Example: the *enriched clickstream* is available to every team within the organization.
State Management
Straightforward Recovery

Spark

Spark
◦ „MapReduce successor“: batch, no unnecessary writes, faster scheduling
◦ High-level API: immutable collections (RDDs) as core abstraction
◦ Many libraries
  • Spark Core: batch processing
  • Spark SQL: distributed SQL
  • Spark MLlib: machine learning
  • Spark GraphX: graph processing
  • Spark Streaming: stream processing
◦ Huge community: 1000+ contributors in 2015
◦ Many big users: Amazon, eBay, Yahoo!, IBM, Baidu, ...

History:
◦ 2009: Spark is developed at UC Berkeley
◦ 2010: Spark is open-sourced
◦ 2014: Spark becomes Apache top-level project
Spark Streaming

Spark

- **High-level API**: DStreams as core abstraction (~Java 8 Streams)
- **Micro-Batching**: latency on the order of seconds
- **Rich feature set**: statefulness, exactly-once processing, elasticity

History:

- 2011: start of development
- 2013: Spark Streaming becomes part of Spark Core
Spark Streaming
Core Abstraction: DStream

Resilient Distributed Data set (RDD):
- **Immutable** collection
- **Deterministic** operations
- **Lineage** tracking:
  → state can be reproduced
  → periodic checkpoints to reduce recovery time

**DStream:** Discretized RDD
- RDDs are processed in order: no ordering for data within an RDD
- RDD Scheduling ~50 ms → latency <100ms infeasible

Illustration taken from:
http://spark.apache.org/docs/latest/streaming-programming-guide.html#overview (2017-02-26)
Spark Streaming
Fault-Tolerance: Receivers & WAL

Illustrations taken from:
Flink

Overview:
- **Native stream processor**: Latency <100ms feasible
- **Abstract API** for stream and batch processing, stateful, exactly-once delivery
- **Many libraries**:
  - Table and SQL: distributed and streaming SQL
  - CEP: complex event processing
  - Machine Learning
  - Gelly: graph processing
  - Storm Compatibility: adapter to run Storm topologies
- **Users**: Alibaba, Ericsson, Otto Group, ResearchGate, Zalando...

History:
- 2010: start of project **Stratosphere** at TU Berlin, HU Berlin, and HPI Potsdam
- 2014: Apache Incubator, project renamed to Flink
- 2015: Apache top-level project
Highlight: State Management
Distributed Snapshots

• **Ordering** within stream partitions
• Periodic **checkpointing**
• **Recovery** procedure:
  1. *reset state* to last checkpoint
  2. *replay data* from last checkpoint

Illustration taken from:
State Management

Checkpointing (1/4)

State Management
Checkpointing (2/4)

State Management
Checkpointing (3/4)

State Management
Checkpointing (4/4)

Other Systems

- **Heron**: open-source, Storm successor
- **Apex**: stream and batch process so with many libraries
- **Dataflow**: Fully managed cloud service for batch and stream processing, proprietary
- **Beam**: open-source runtime-agnostic API for Dataflow programming model; runs on Flink, Spark and others
- **KafkaStreams**: integrated with Kafka, open-source
- **IBM Infosphere Streams**: proprietary, managed, bundled with IDE
- **And even more**: Kinesis, Gearpump, MillWheel, Muppet, S4, Photon, ...
## Direct Comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>Storm</th>
<th>Trident</th>
<th>Samza</th>
<th>Spark Streaming</th>
<th>Flink (streaming)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strictest Guarantee</strong></td>
<td>at-least-once</td>
<td>exactly-once</td>
<td>at-least-once</td>
<td>exactly-once</td>
<td>exactly-once</td>
</tr>
<tr>
<td><strong>Achievable Latency</strong></td>
<td>$\ll 100$ ms</td>
<td>$&lt;100$ ms</td>
<td>$&lt;100$ ms</td>
<td>$&lt;1$ second</td>
<td>$&lt;100$ ms</td>
</tr>
<tr>
<td><strong>State Management</strong></td>
<td>(small state)</td>
<td>(small state)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Processing Model</strong></td>
<td>one-at-a-time</td>
<td>micro-batch</td>
<td>one-at-a-time</td>
<td>micro-batch</td>
<td>one-at-a-time</td>
</tr>
<tr>
<td><strong>Backpressure</strong></td>
<td>✓</td>
<td>✓</td>
<td>not required (buffering)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Ordering</strong></td>
<td>✗</td>
<td>between batches</td>
<td>within partitions</td>
<td>between batches</td>
<td>within partitions</td>
</tr>
<tr>
<td><strong>Elasticity</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
Wrap-Up
Wrap-up

- **Push-based data access**
  - Natural for many applications
  - Hard to implement on top of traditional (pull-based) databases

- **Real-time databases**
  - Natively push-based
  - Challenges: scalability, fault-tolerance, semantics, rewrite vs. upgrade, ...

- **Scalable Stream Processing**
  - Stream vs. Micro-Batch (vs. Batch)
  - Lambda & Kappa Architecture
  - Vast feature space, many frameworks

- **InvaliDB**
  - A linearly scalable design for add-on push-based queries
  - Database-independent
  - Real-time updates for powerful queries: filter, sorting, joins, aggregations
Outline

Scalable Data Processing: 
Big Data in Motion

Stream Processors: 
Side-by-Side Comparison

Real-Time Databases: 
Push-Based Data Access

Current Research: 
Opt-In Push-Based Access

- Pull-Based vs Push-Based Data Access
- DBMS vs. RT DB vs. DSMS vs. Stream Processing
- Popular Push-Based DBs:
  - Firebase
  - Meteor
  - RethinkDB
  - Parse
  - Others
- Discussion
Real-Time Databases
Traditional Databases
No Request? No Data!

What’s the current state?

circular shapes

Query maintenance: periodic polling
→ Inefficient
→ Slow
Ideal: Push-Based Data Access

Self-Maintaining Results

Find people in Room B:

```javascript
db.User.find()
  .equal('room', 'B')
  .ascending('name')
  .limit(3)
  .streamResult()
```

1. Erik (5/10)
2. Wolle (22/8)
3. Ideal: Push-Based Data Access
Self-Maintaining Results
Popular Real-Time Databases
Overview:

- **Real-time state synchronization** across devices
- **Simplistic data model**: nested hierarchy of lists and objects
- **Simplistic queries**: mostly navigation/filtering
- **Fully managed**, proprietary
- **App SDK** for App development, mobile-first
- **Google services integration**: analytics, hosting, authorization, ...

History:

- 2011: chat service startup Envolve is founded
  → was often used for cross-device state synchronization
  → state synchronization is separated (Firebase)
- 2012: Firebase is founded
- 2013: Firebase is acquired by Google
Firebase
Real-Time State Synchronization

- **Tree data model**: application state ~ JSON object
- **Subtree synching**: push notifications for specific keys only
  → Flat structure for fine granularity

→ **Limited expressiveness**!
Firebase
Query Processing in the Client

• Push notifications for **specific keys** only
  • Order by a **single attribute**
  • Apply a **single filter** on that attribute

• Non-trivial query processing in client
  → **does not scale!**

Illustration taken from: Frank van Puffelen, *Have you met the Realtime Database? (2016)*

Jacob Wenger, on the Firebase Google Group (2015)
https://groups.google.com/forum/#!topic/firebase-talk/d-XjaBVL2Ko (2017-02-27)
Overview:
- **JavaScript Framework** for interactive apps and websites
  - **MongoDB** under the hood
  - **Real-time** result updates, full MongoDB expressiveness
- **Open-source**: MIT license
- **Managed service**: Galaxy (Platform-as-a-Service)

History:
- 2011: *Skybreak* is announced
- 2012: Skybreak is renamed to Meteor
- 2015: Managed hosting service Galaxy is announced
Live Queries
Poll-and-Diff

- **Change monitoring**: app servers detect relevant changes → *incomplete* in multi-server deployment
- **Poll-and-diff**: queries are re-executed periodically → *staleness window* → *does not scale* with queries
Oplog Tailing
Basics: MongoDB Replication

- **Oplog**: rolling record of data modifications
- **Master-slave replication**: Secondaries subscribe to oplog

![Diagram showing MongoDB cluster with primary and secondary nodes]
Oplog Tailing
Tapping into the Oplog

• *Every* Meteor server receives *all* DB writes through oplogs → does not scale
What game does Bobby play?
→ if baccarat, he takes first place!
→ if something else, nothing changes!

Partial update from oplog:
{ name: „Bobby“ , score: 500 } // game: ???

Baccarat players sorted by high-score:
1. { name: „Joy“ , game: „baccarat“ , score: 100 }
2. { name: „Tim“ , game: „baccarat“ , score: 90 }
3. { name: „Lee“ , game: „baccarat“ , score: 80 }
Overview:

- „MongoDB done right“: comparable queries and data model, but also:
  - Push-based queries (filters only)
  - Joins (non-streaming)
  - Strong consistency: linearizability
- JavaScript SDK (*Horizon*): open-source, as managed service
- Open-source: Apache 2.0 license

History:

- 2009: RethinkDB is founded
- 2012: RethinkDB is open-sourced under AGPL
- 2016, May: first official release of Horizon (JavaScript SDK)
- 2016, October: RethinkDB announces shutdown
- 2017: RethinkDB is relicensed under Apache 2.0
RethinkDB

Changefeed Architecture

• Range-sharded data
• **RethinkDB proxy**: support node without data
  • Client communication
  • Request routing
  • Real-time query matching

• *Every* proxy receives *all* database writes
  → **does not scale**

---


Daniel Mewes, *Comment on GitHub issue #962: Consider adding more docs on RethinkDB Proxy* (2016)
https://github.com/rethinkdb/docs/issues/962 (2017-02-27)
Overview:

- **Backend-as-a-Service** for mobile apps
  - **MongoDB**: largest deployment world-wide
  - **Easy development**: great docs, push notifications, authentication, ...
  - **Real-time** updates for most MongoDB queries
- **Open-source**: BSD license
- **Managed service**: discontinued

History:

- 2011: Parse is founded
- 2013: Parse is acquired by Facebook
- 2015: more than 500,000 mobile apps reported on Parse
- 2016, January: Parse shutdown is announced
- 2016, March: **Live Queries** are announced
- 2017: Parse shutdown is finalized
LiveQuery Architecture

- **LiveQuery Server**: no data, real-time query matching
- *Every* LiveQuery Server receives *all* database writes
  \[\rightarrow \text{does not scale}\]

**Bottleneck!**

Illustration taken from:
### Comparison by Real-Time Query

#### Why Complexity Matters

<table>
<thead>
<tr>
<th></th>
<th>matching conditions</th>
<th>ordering</th>
<th>Firebase</th>
<th>Meteor</th>
<th>RethinkDB</th>
<th>Parse</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Todos</strong></td>
<td>created by „Bob“</td>
<td>ordered by deadline</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Todos</strong></td>
<td>created by „Bob“ AND with status equal to „active“</td>
<td></td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Todos</strong></td>
<td>with „work“ in the name</td>
<td></td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>ordered by deadline</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Todos</strong></td>
<td>with „work“ in the name AND status of „active“</td>
<td>ordered by deadline AND then by the creator’s name</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
# Quick Comparison

## DBMS vs. RT DB vs. DSMS vs. Stream Processing

<table>
<thead>
<tr>
<th></th>
<th>Database Management</th>
<th>Real-Time Databases</th>
<th>Data Stream Management</th>
<th>Stream Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td>persistent collections</td>
<td>persistent/ephemeral streams</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Processing</strong></td>
<td>one-time</td>
<td>one-time + continuous</td>
<td>continuous</td>
<td></td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>random</td>
<td>random + sequential</td>
<td>sequential</td>
<td></td>
</tr>
<tr>
<td><strong>Streams</strong></td>
<td>structured</td>
<td></td>
<td>structured, unstructured</td>
<td></td>
</tr>
</tbody>
</table>

### Examples:
- **Database Management**: ORACLE, PostgreSQL, MySQL, IBM DB2
- **Real-Time Databases**: Firebase, RethinkDB, Parse
- **Data Stream Management**: PIPELINE DB, EsperTech, sqlstream
- **Stream Processing**: STORM, samza, Flink, Spark Streaming
Discussion

Common Issues

Every database with real-time features suffers from several of these problems:

• **Expressiveness**:  
  • Queries  
  • Data model  
  • Legacy support

• **Performance**:  
  • Latency & throughput  
  • **Scalability**

• **Robustness**:  
  • Fault-tolerance, handling malicious behavior etc.  
  • Separation of concerns:  
    → **Availability**: will a crashing real-time subsystem take down primary data storage?  
    → **Consistency**: can real-time be scaled out independently from primary storage?
Outline

- InvaliDB: Opt-In Real-Time Queries
- Distributed Query Matching
- Staged Query Processing
- Performance Evaluation
- Wrap-Up

Scalable Data Processing: Big Data in Motion

Stream Processors: Side-by-Side Comparison

Real-Time Databases: Push-Based Data Access

Current Research: Opt-In Push-Based Access
Current Research
InvaliDB
External Query Maintenance
SELECT * 
FROM posts 
WHERE title LIKE "%NoSQL%" 
ORDER BY year DESC

{ title: "SQL", 
  year: 2016 }

add    changelIndex  change    remove
InvaliDB
Filter Queries: Distributed Query Matching

Two-dimensional partitioning:
- by Query
- by Object
→ scales with queries and writes

Implementation:
- Apache Storm
- Topology in Java
- MongoDB query language
- Pluggable query engine

SELECT * FROM posts WHERE tags CONTAINS 'NoSQL'

Match!
InvaliDB
Staged Real-Time Query Processing

Change notifications go through up to 4 query processing stages:
1. **Filter queries**: track matching status → *before*- and after-images
2. **Sorted queries**: maintain result order
3. **Joins**: combine maintained results
4. **Aggregations**: maintain aggregations
InvaliDB
Low Latency + Linear Scalability

![Diagram showing throughput vs. matching nodes with latency benchmarks.]

- 99th Percentile Latency \( \leq \) 25 ms
- 99th Percentile Latency \( \leq \) 20 ms
- 99th Percentile Latency \( \leq \) 15 ms
Our NoSQL research at the University of Hamburg
The Latency Problem

- Average: 9.3s
- -7% Conversions
- -20% Traffic
- -9% Visitors
- -1% Revenue

Aberdeen Group
Google
Yahoo!
Amazon.com
If perceived speed is such an important factor...

...what causes slow page load times?
State of the Art

Two bottlenecks: latency und processing
Network Latency: Impact

Page Load Time as bandwidth increases

Page Load Time as latency decreases

Network Latency: Impact

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

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

Our Low-Latency Vision

Data is served by ubiquitous web-caches
Innovation
Solution: Proactively Revalidate Data

5 Years Research & Development
New Algorithms Solve Consistency Problem
Innovation
Solution: Proactively Revalidate Data


We measured page load times for users in four geographic regions. Our caching technology achieves on average **6.8x faster** loading times compared to competitors.
Business Model

Backend-as-a-Service

Pay-per-use or on-Premise

Simplified development

Cached data with minimal latency

Baqend Cloud

Baqend Enterprise

Backend

Caching infrastructure

End user
Orestes
Components
Orestes
Components

Polyglot Persistence Mediator
Orestes
Components

Backend-as-a-Service Middleware: Caching, Transactions, Schemas, Invalidation Detection, ...
Orestes
Components

Standard HTTP Caching
Orestes
Components

Unified REST API

- Desktop
- Mobile
- Tablet

Content-Delivery-Network

Internet

Orestes Servers

Node.js User-defined Business Logic

TTL Estimator Cache Lifetime Prediction

InvalidDB Streaming Queries

Expiring Bloom Filter Stale Data

Reverse-Proxy Caches

mongoDB

elasticsearch
Bloom filters for Caching
End-to-End Example
Bloom filters for Caching
End-to-End Example

Bloom filters are used for caching purposes. The process involves:

1. Browser Cache:
   - Requests are made to the Browser Cache.
   - Time-to-Live Estimation is obtained from the server.

2. CDN (Content Delivery Network):
   - Requests are served from CDN.

3. Server:
   - Data is stored and managed.

This process optimizes content delivery and reduces load times.
Bloom filters for Caching
End-to-End Example
Bloom filters for Caching
End-to-End Example
Bloom filters for Caching
End-to-End Example

Browser Cache

CDN

purge(obj)

hashA(oid)  hashB(oid)

0 3 1 4 1

Bloom filters for Caching
End-to-End Example

Browser Cache

CDN

purge(obj)

hashA(oid)  hashB(oid)

0 3 1 4 1
Bloom filters for Caching

End-to-End Example

Flat(Counting Bloomfilter)
Bloom filters for Caching

End-to-End Example
Bloom filters for Caching
End-to-End Example
Bloom filters for Caching
End-to-End Example
Bloom filters for Caching

End-to-End Example
Bloom filters for Caching
End-to-End Example

False-Positive Rate:
\[ f \approx (1 - e^{-\frac{kn}{m}})^k \]

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:** \(\Delta\)-Atomicity, Read-Your-Writes, Monotonic Reads, Monotonic Writes, Causal Consistency
Baqend: Core Features

Automatic Scaling

Faster Development

>250%
Faster Loads

#1 Users are less annoyed and less annoying.

#2 The admin does not look as grim and angry as usual.

#3 The nerds have time to catch some fresh air.

Sources

http://de.slideshare.net/felixgessert/talk-cache-sketches-using-bloom-filters-and-web-caching-against-slow-load-times

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](http://bagend.com)

Download **Community Edition**
Literature Recommendations
Recommended Literature

**NoSQL Databases: A Survey and Decision Guidance**
Together with our colleagues at the University of Hamburg, we—that is Felix Gessert, Wolfram Wingerath, Steffen Friedrich and Norbert Ritter—presented an overview over the NoSQL landscape at SummerSOC'16 last month. Here is the written gist. We give our best to convey the condensed NoSQL knowledge we gathered building Baqend.

**Scalable Stream Processing: A Survey of Storm, Samza, Spark and Flink**

**TL;DR**
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.

With this article, we would like to share our insights on real-time data processing we gained building Baqend. This is an updated version of our most recent stream processor survey which is another cooperation with the University of Hamburg (authors: Wolfram Wingerath, Felix Gessert, Steffen Friedrich and Norbert Ritter). As you may or may not have been aware of, a lot of stream processing is going on behind the curtains at Baqend. In our quest to provide the lowest possible latency, we have built a system to enable query caching and real-time notifications (similar to changefeeds in RethinkDB/Horizon) and hence learned a lot about the competition in the field of stream processors.

Read them at blog.baqend.com!
Recommended Literature

1. Lena Wiese: *Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases*

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

- **Big Data**
  - Principles and best practices of scalable real-time data systems
  - Nathan Marz
  - With James Warren

- **NOSQL For Mere Mortals**
  - Dan Sullivan

![Big Data book cover](image1)
![NOSQL book cover](image2)
Recommended Literature: Cloud-DBs
Recommended Literature: Blogs

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

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

**InfoQ**

**Aphyr**
https://aphyr.com/

**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

- 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
Thank you – questions?

Norbert Ritter, Felix Gessert, Wolfram Wingerath
{ritter,gessert,wingerath}@informatik.uni-hamburg.de
Polyglot Persistence
Current best practice
Polyglot Persistence
Current best practice

Research Question:
Can we automate the mapping problem?
Vision

Schemas can be annotated with requirements

- Write Throughput > 10,000 RPS
- Read Availability > 99.9999%
- Scans = true
- Full-Text-Search = true
- Monotonic Read = true
Vision
The Polyglot Persistence Mediator chooses the database

Application

Data and Operations

Polyglot Persistence Mediator

Database Metrics

db1

db2

db3

Latency < 30ms

Annotated Schema
Step I - Requirements
Expressing the application’s needs

- Tenant annotates schema with his requirements

1. Define schema
2. Annotate

Annotations
- Continuous non-functional e.g. write latency < 15ms
- Binary functional e.g. Atomic updates
- Binary non-functional e.g. Read-your-writes
Step 1 - Requirements
Expressing the application’s needs

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Type</th>
<th>Annotated at</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Availability</td>
<td>Continuous</td>
<td>*</td>
</tr>
<tr>
<td>Write Availability</td>
<td>Continuous</td>
<td>*</td>
</tr>
<tr>
<td>Read Latency</td>
<td>Continuous</td>
<td>*</td>
</tr>
<tr>
<td>Write Latency</td>
<td>Continuous</td>
<td>*</td>
</tr>
<tr>
<td>Write Throughput</td>
<td>Continuous</td>
<td>*</td>
</tr>
<tr>
<td>Data Vol. Scalability</td>
<td>Non-Functional</td>
<td>Field/Class/DB</td>
</tr>
<tr>
<td>Write Scalability</td>
<td>Non-Functional</td>
<td>Field/Class/DB</td>
</tr>
<tr>
<td>Read Scalability</td>
<td>Non-Functional</td>
<td>Field/Class/DB</td>
</tr>
<tr>
<td>Elasticity</td>
<td>Non-Functional</td>
<td>Field/Class/DB</td>
</tr>
<tr>
<td>Durability</td>
<td>Non-Functional</td>
<td>Field/Class/DB</td>
</tr>
<tr>
<td>Replicated</td>
<td>Non-Functional</td>
<td>Field/Class/DB</td>
</tr>
<tr>
<td>Linearizability</td>
<td>Non-Functional</td>
<td>Field/Class</td>
</tr>
<tr>
<td>Read-your-Writes</td>
<td>Non-Functional</td>
<td>Field/Class</td>
</tr>
<tr>
<td>Causal Consistency</td>
<td>Non-Functional</td>
<td>Field/Class</td>
</tr>
<tr>
<td>Writes follow reads</td>
<td>Non-Functional</td>
<td>Field/Class</td>
</tr>
<tr>
<td>Monotonic Read</td>
<td>Non-Functional</td>
<td>Field/Class</td>
</tr>
<tr>
<td>Monotonic Write</td>
<td>Non-Functional</td>
<td>Field/Class</td>
</tr>
<tr>
<td>Scans</td>
<td>Functional</td>
<td>Field</td>
</tr>
<tr>
<td>Sorting</td>
<td>Functional</td>
<td>Field</td>
</tr>
<tr>
<td>Range Queries</td>
<td>Functional</td>
<td>Field</td>
</tr>
<tr>
<td>Point Lookups</td>
<td>Functional</td>
<td>Field</td>
</tr>
<tr>
<td>ACID Transactions</td>
<td>Functional</td>
<td>Class/DB</td>
</tr>
<tr>
<td>Conditional Updates</td>
<td>Functional</td>
<td>Field</td>
</tr>
<tr>
<td>Joins</td>
<td>Functional</td>
<td>Class/DB</td>
</tr>
<tr>
<td>Analytics Integration</td>
<td>Functional</td>
<td>Field/Class/DB</td>
</tr>
<tr>
<td>Fulltext Search</td>
<td>Functional</td>
<td>Field</td>
</tr>
<tr>
<td>Atomic Updates</td>
<td>Functional</td>
<td>Field/Class</td>
</tr>
</tbody>
</table>

Annotations:
- Continuous non-functional e.g. write latency < 15ms
- Binary functional e.g. Atomic updates
- Binary non-functional e.g. Read-your-writes

1. Define schema
2. Annotate

Inherits continuous annotations

annotated
Step II - Resolution
Finding the best database

- The Provider resolves the requirements
- **RANK**: scores available database systems
- **Routing Model**: defines the optimal mapping from schema elements to databases

RANK(schema_root, DBs) through recursive descent using annotated schema and metrics

Routing Model
Route schema_element → db
- transform db-independent to db-specific operations
Step III - Mediation
Routing data and operations

- The PPM routes data
- **Operation Rewriting:** translates from abstract to database-specific operations
- **Runtime Metrics:** Latency, availability, etc. are reported to the resolver
- **Primary Database Option:** All data periodically gets materialized to designated database
Evaluation: News Article
Prototype of Polyglot Persistence Mediator in ORESTES

Scenario: news articles with impression counts
Objectives: low-latency top-k queries, high-throughput counts, article-queries

Article

Counter

Hacker News new | threads | comments | show | ask | jobs | submissions

1. * NoSQL Databases: A Survey and Decision Guidance (medium.com)
297 points by DivineTraube 9 days ago | past | web | 73 comments | in pocket speichern

read by 53,222
Evaluation: News Article
Prototype built on ORESTES

Scenario: news articles with impression counts
Objectives: low-latency top-k queries, high-throughput counts, article-queries

Counter updates kill performance
Evaluation: News Article
Prototype built on ORESTES

Scenario: news articles with impression counts
Objectives: low-latency top-k queries, high-throughput counts, article-queries

Mediator: No powerful queries
**Evaluation: News Article**

Prototype built on ORESTES

**Scenario:** news articles with impression counts

**Objectives:** low-latency top-k queries, high-throughput counts, article-queries

---

**Found Resolution**

- **Article:** ID, Title, ...
- **Imp.:** ID, Imp.

- **mongoDB**
- **redis**
Cloud Data Management

- New field tackling the *design, implementation, evaluation* and *application implications* of *database systems* in *cloud environments*:

- Protocols, APIs, Caching
- Load distribution, Auto-Scaling, SLAs
- Workload Management, Metering
- Replication, Partitioning, Transactions, Indexing
- Application architecture, Data Models
- Multi-Tenancy, Consistency, Availability, Query Processing, Security
Cloud-Database Models

Data Model

Data Model

Deployment Model

unstructured

unstructured

schema-free

relational

structured

structured

unmanaged

cloud-deployed
(laaS/PaaS)

Managed
(cloud-hosted)

Proprietary
DB & Cloud

managed

Analytics machine image

NoSQL machine image

RDBMS machine image

Analytics-as-a-Service

Managed NoSQL

Managed RDBMS/DWH

Analytics/ML APIs

NoSQL Service

RDBMS/DWH Service

Database-as-a-Service

Deployment Model
**Cloud-Deployed Database**

Database-image provisioned in IaaS/PaaS-cloud

---

**IaaS/PaaS deployment of database system**

**Does not solve:**
Provisioning, Backups, Security, Scaling, Elasticity, Performance Tuning, Failover, Replication, ...
Managed RDBMS/DWH/NoSQL DB

Cloud-hosted database

DBaaS-Provider

RDBMS | DWH | NoSQL DB

IaaS-Cloud

RDBMS
- Amazon RDS
- SQL Azure
- Clustrix
- EDB
- Google Cloud SQL
- Heroku Postgres

NoSQL DB
- Amazon ElastiCache
- MongoDB
- mongolab
- Cloudant
- Redis
- Instaclustr
- Bonsai
- Iris Couch

DWH
- Amazon Redshift
Managed RDBMS/DWH/NoSQL DB

Cloud-hosted database

DBaaS-Provider

RDBMS
DWH
NoSQL DB

IaaS-Cloud

Provisioning, Backups, Security, Scaling, Elasticity, Performance Tuning, Failover, Replication, ...

Amazon RDS
SQL Azure
Clustrix
EDB
Roku Postgres
Amazon ElastiCache
mongoHQ
mongolab
Cloudant
Iris Couch
redis
bonsai
Instaclustr

Amazon Redshift

NoSQL DB
RDBMS
DWH
Proprietary Cloud Database
Designed for and deployed in vendor-specific cloud environment

Cloud

Black-box system

Managed by Cloud Provider

Provider’s API

Database

Amazon SimpleDB
Google Cloud Datastore
Database.com

BigTable, Megastore, Spanner, F1, Dynamo, PNuts, Relational Cloud, …

Object Store

Azure Blob Storage
Openstack Swift
Amazon S3
Google Cloud Storage

Cloud
Analytics-as-a-Service

Analytic frameworks and machine learning with service APIs

Cloud

Provisioning, Data Ingest

Analytics Cluster

Analytics

Amazon Elastic MapReduce

Azure HDInsight

ML

Google BigQuery

Google Prediction API
Backend-as-a-Service
DBaaS with embedded custom and predefined application logic

- Authentication, Users, Validation, etc.
- Maps to (different) databases

Backend API — Data API
Service-Layer
IaaS-Cloud

(mobile) BaaS

AppCelerator Cloud
BaQend
Parse
GoInstant
Meteor PREVIEW
Firebase
APIOMAT
Pricing Models
Pay-per-use and plan-based

Pay-per-use
Parameters: Network, Bandwidth, Storage, CPU, Requests, etc.
Payment: Pre-Paid, Post-Paid
Variants: On-Demand, Auction, Reserved

Account
Usage
End of month

e.g. DynamoDB
e.g. Compose
Pricing Models
Pay-per-use and plan-based

Plan-based
Parameters: Allocated Plan (e.g. 2 instances + X GB storage)

Account
Usage
End of month

e.g. Compose
e.g. DynamoDB
Database-as-a-Service

Approaches to Multi-Tenancy

Private OS
- Schema
- Database
- Database Process
- VM
- Hardware Resources
  - e.g. Amazon RDS

Private Process/DB
- Schema
- Database
- Database Process
- VM
- Hardware Resources
  - e.g. Compose

Private Schema
- Schema
- Database
- Database Process
- VM
- Hardware Resources
  - e.g. Google DataStore

Shared Schema
- Virtual Schema
- Schema
- Database
- Database Process
- VM
- Hardware Resources
  - Most SaaS Apps

T. Kiefer, W. Lehner “Private table database virtualization for dbaas”
UCC, 2011
## Multi-Tenancy: Trade-Offs

<table>
<thead>
<tr>
<th></th>
<th>App. indep.</th>
<th>Ressource Util.</th>
<th>Isolation</th>
<th>Maintenance, Provisioning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Private OS</strong></td>
<td>✔</td>
<td>★★★</td>
<td>★★★</td>
<td>★★★★</td>
</tr>
<tr>
<td><strong>Private Process/DB</strong></td>
<td>✔</td>
<td>★★★</td>
<td>★★★</td>
<td>★★★★</td>
</tr>
<tr>
<td><strong>Private Schema</strong></td>
<td>✔</td>
<td>★★★</td>
<td>★★★</td>
<td>★★★★</td>
</tr>
<tr>
<td><strong>Shared Schema</strong></td>
<td>✗</td>
<td>★★★★★</td>
<td>★★★</td>
<td>★★★★</td>
</tr>
</tbody>
</table>

Authentication & Authorization
Checking Permissions and Identity

<table>
<thead>
<tr>
<th>Internal Schemes</th>
<th>External Identity Provider</th>
<th>Federated Identity (Single Sign On)</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. Amazon IAM</td>
<td>e.g. OpenID</td>
<td>e.g. SAML</td>
</tr>
</tbody>
</table>

User-based Access Control
- e.g. Amazon S3 ACLs

Role-based Access Control
- e.g. Amazon IAM

Policies
- e.g. XACML
Service Level Agreements (SLAs)
Specification of Application/Tenant Requirements

Service Level Objectives:
- Availability
- Durability
- Consistency/Staleness
- Query Response Time

SLA

Technical Part
1. SLO
2. SLO
3. SLO

Legal Part
1. Fees
2. Penalties
Service Level Agreements
Expressing application requirements

**Functional** Service Level Objectives
- Guarantee a „feature“
- Determined by database system
- *Examples*: transactions, join

**Non-Functional** Service Level Objectives
- Guarantee a certain *quality of service* (QoS)
- Determined by database system and service provider
- *Examples*:
  - *Continuous*: response time (latency), throughput
  - *Binary*: Elasticity, Read-your-writes
Service Level Objectives
Making SLOs measurable through utilities

Utility expresses „value“ of a continuous non-functional requirement:

\[ f_{utility}(\text{metric}) \rightarrow [0,1] \]
Workload Management

Guaranteeing SLAs

Typical approach:
Workload Management

Guaranteeing SLAs

Typical approach:
Workload Management

Guaranteeing SLAs

Typical approach:
Workload Management

Guaranteeing SLAs

Typical approach:

Maximize:

utility

response time

classes

admission control & scheduling

workload classification

transaction

response time
Workload Management

Guaranteeing SLAs

Typical approach:
Resource & Capacity Planning
From a DBaaS provider’s perspective

**Goal:** minimize penalty and resource costs

Resource & Capacity Planning
From a DBaaS provider’s perspective

Goal: minimize penalty and resource costs

Provisioned Resources:
- #No of Shard- or Replica servers
- Computing, Storage, Network Capacities

Resources vs Time

Expected Load

Resource & Capacity Planning
From a DBaaS provider’s perspective

**Goal**: minimize penalty and resource costs

**Resource & Capacity Planning**

*From a DBaaS provider’s perspective*

**Goal:** minimize penalty and resource costs

- **Overprovisioning:**
  - SLAs met
  - Excess Capacities

- **Underprovisioning:**
  - SLAs violated
  - Usage maximized

---

## SLAs in the wild

Most DBaaS systems offer no SLAs, or only a simple uptime guarantee.

<table>
<thead>
<tr>
<th></th>
<th>Model</th>
<th>CAP</th>
<th>SLAs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SimpleDB</strong></td>
<td>Table-Store (NoSQL Service)</td>
<td>CP</td>
<td>[X] 99.9% uptime</td>
</tr>
<tr>
<td><strong>Dynamo-DB</strong></td>
<td>Table-Store (NoSQL Service)</td>
<td>CP</td>
<td>[X]</td>
</tr>
<tr>
<td><strong>Azure Tables</strong></td>
<td>Table-Store (NoSQL Service)</td>
<td>CP</td>
<td>99.9% uptime</td>
</tr>
<tr>
<td><strong>AE/Cloud DataStore</strong></td>
<td>Entity-Group Store (NoSQL Service)</td>
<td>CP</td>
<td>[X]</td>
</tr>
<tr>
<td><strong>S3, Az. Blob, GCS</strong></td>
<td>Object-Store (NoSQL Service)</td>
<td>AP</td>
<td>99.9% uptime (S3)</td>
</tr>
</tbody>
</table>
Open Research Questions in Cloud Data Management

- **Service-Level Agreements**
  - How can SLAs be guaranteed in a virtualized, multi-tenant cloud environment?

- **Consistency**
  - Which consistency guarantees can be provided in a geo-replicated system without sacrificing availability?

- **Performance & Latency**
  - How can a DBaaS deliver low latency in face of distributed storage and application tiers?

- **Transactions**
  - Can ACID transactions be aligned with NoSQL and scalability?
### DBaaS Example
**Amazon RDS**

- **Relational Database Service**

<table>
<thead>
<tr>
<th>Engine Selection</th>
<th>Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL Community Edition</td>
<td><img src="https://via.placeholder.com/15/ff6300/000000" alt="Select" /></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td><img src="https://via.placeholder.com/15/ff6300/000000" alt="Select" /></td>
</tr>
<tr>
<td>Oracle Database Standard Edition One</td>
<td><img src="https://via.placeholder.com/15/ff6300/000000" alt="Select" /></td>
</tr>
<tr>
<td>Oracle Database Standard Edition</td>
<td><img src="https://via.placeholder.com/15/ff6300/000000" alt="Select" /></td>
</tr>
<tr>
<td>Oracle Database Enterprise Edition</td>
<td><img src="https://via.placeholder.com/15/ff6300/000000" alt="Select" /></td>
</tr>
<tr>
<td>Microsoft SQL Server Express Edition</td>
<td><img src="https://via.placeholder.com/15/ff6300/000000" alt="Select" /></td>
</tr>
<tr>
<td>Microsoft SQL Server Standard Edition</td>
<td><img src="https://via.placeholder.com/15/ff6300/000000" alt="Select" /></td>
</tr>
<tr>
<td>Microsoft SQL Server Web Edition</td>
<td><img src="https://via.placeholder.com/15/ff6300/000000" alt="Select" /></td>
</tr>
</tbody>
</table>

**RDS**

- **Model:** Managed RDBMS
- **Pricing:** Instance + Volume + License
- **Underlying DB:** MySQL, Postgres, MSSQL, Oracle
- **API:** DB-specific
DBaaS Example
Amazon RDS

- Relational Database Service

RDS

Model:
- Managed RDBMS

Pricing:
- Instance + Volume + License

Underlying DB:
- MySQL, Postgres, MSSQL, Oracle

API:
- DB-specific
DBaaS Example
Amazon RDS

- **Relational Database Service**

  - **Synchronous** Replication
  - **Automatic Failover**

**RDS**

- **Model:**
  - Managed RDBMS

- **Pricing:**
  - Instance + Volume + License

- **Underlying DB:**
  - MySQL, Postgres, MSSQL, Oracle

- **API:**
  - DB-specific

---

**Question:** Is this database intended for production or pre-production?

- **Yes:** Use Multi-AZ Deployment and Provisioned IOPS Storage as defaults while creating this instance.
- **No:** This instance is intended for use outside of production or under the RDS Free Usage Tier.

**Billing:**

- Billing is based upon the RDS pricing table. An instance which uses these features is not eligible for the RDS Free Usage Tier.
DBaaS Example
Amazon RDS

- **Relational Database Service**
  - **Synchronous** Replication
  - **Automatic Failover**

99.95% uptime SLA

- Underlying DB: MySQL, Postgres, MSSQL, Oracle
- API: DB-specific

Pricing:
Instance + Volume + License
Relational Database Service

- **Synchronous** Replication
- **Automatic Failover**

**Provisioned IOPS**: access to EBS volumes network-optimized (up to 4000 IOPS)

**99.95% uptime SLA**

**Pricing**:
- Instance + Volume + License

**Underlying DB**:
- MySQL,
- Postgres,
- MSSQL,
- Oracle

**API**:
- DB-specific
DBaaS Example
Amazon RDS

- Relational Database Service

**RDS**

- **Model:**
- **Managed RDBMS**
- **Pricing:** Instance + Volume + License
- **Underlying DB:** MySQL, Postgres, MSSQL, Oracle
- **API:** DB-specific
DBaaS Example
Amazon RDS

- Relational Database Service

EC2 instances: Up to 32 Cores, 244 GB RAM, 10 GbE

RDS

Model:
Managed RDBMS

Pricing:
Instance + Volume + License

Underlying DB:
MySQL, Postgres, MSSQL, Oracle

API:
DB-specific
DBaaS Example
Amazon RDS

- Relational Database Service

**RDS**
- **Model:**
- **Managed RDBMS**
- **Pricing:** Instance + Volume + License
- **Underlying DB:** MySQL, Postgres, MSSQL, Oracle
- **API:** DB-specific

**RDS**

**Minor Version Upgrades** are performed without downtime.

**EC2 instances:** Up to 32 Cores, 244 GB RAM, 10 GbE
DBaaS Example
Amazon RDS

- **Relational Database Service**

<table>
<thead>
<tr>
<th>RDS</th>
<th>Model: Managed RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pricing:</td>
<td>Instance + Volume + License</td>
</tr>
<tr>
<td>Underlying DB:</td>
<td>MySQL, Postgres, MSSQL, Oracle</td>
</tr>
<tr>
<td>API:</td>
<td>DB-specific</td>
</tr>
</tbody>
</table>
DBaaS Example
Amazon RDS

- **Relational Database Service**

Backups are automated and scheduled

<table>
<thead>
<tr>
<th>RDS</th>
<th>Model:</th>
<th>Managed RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pricing:</td>
<td>Instance + Volume + License</td>
</tr>
<tr>
<td></td>
<td>Underlying DB:</td>
<td>MySQL, Postgres, MSSQL, Oracle</td>
</tr>
<tr>
<td></td>
<td>API:</td>
<td>DB-specific</td>
</tr>
</tbody>
</table>
DBaaS Example
Amazon RDS

- **Relational Database Service**

- **RDS**
  - Model: Managed RDBMS
  - Pricing: Instance + Volume + License
  - Underlying DB: MySQL, Postgres, MSSQL, Oracle
  - API: DB-specific

- **Backups** are automated and scheduled

- Support for (asynchronous) Read Replicas
- **Administration**: Web-based or SDKs
- Only RDBMSs
- “Analytic Brother” of RDS: RedShift (PDWH)
## DBaaS Example

### Azure Tables

<table>
<thead>
<tr>
<th>REST API</th>
<th>Partition Key</th>
<th>Row Key <em>(sortiert)</em></th>
<th>Timestamp <em>(autom.)</em></th>
<th>Property1</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>intro.pdf</td>
<td>v1.1</td>
<td>14/6/2013</td>
<td>...</td>
<td>...</td>
<td>Partition</td>
</tr>
<tr>
<td>intro.pdf</td>
<td>v1.2</td>
<td>15/6/2013</td>
<td>...</td>
<td>...</td>
<td>Sparse</td>
</tr>
<tr>
<td>präs.pptx</td>
<td>v0.0</td>
<td>11/6/2013</td>
<td>...</td>
<td>...</td>
<td>Partition</td>
</tr>
</tbody>
</table>

- **No Index:** Lookup only (!) by full table scan
- **Atomic "Entity-Group Batch Transaction"** possible
- **Hash-distributed to partition servers**

---

### Similar to Amazon

- **SimpleDB**
  - Indexes all attributes
  - Rich(er) queries
  - Many Limits (size, RPS, etc.)

- **DynamoDB**
  - Provisioned Throughput
  - On SSDs ("single digit latency")
  - Optional Indexes
DBaaS and PaaS Example

Heroku Addons

- Many Hosted NoSQL DbaaS Providers represented
- And Search
DBaaS and PaaS Example

Heroku Addons

Create Heroku App:

```
$ heroku create
```

Add Redis2Go Addon:

```
$ heroku addons:add redistogo
------> Adding RedisToGo to fat-unicorn-1337... done, v18 (free)
```

Use Connection URL (environment variable):

```
uri = URI.parse(ENV['REDISTOGO_URL'])
REDIS = Redis.new(:url => ENV['REDISTOGO_URL'])
```

Deploy:

```
$ git push heroku master
```
DBaaS and PaaS Example

Heroku Addons

Create Heroku App:

```
$ heroku create
```

Add Redis2Go Addon:

```
$ heroku addons:add redistogo
----> Adding RedisToGo to fat-unicorn-1337... done, v18 (free)
```

Use Connection URL (environment variable):

```ruby
uri = URI.parse(ENV['REDISTOGO_URL'])
REDIS = Redis.new(:url => ENV['REDISTOGO_URL'])
```

- Very simple
- Only suited for small to medium applications (no SLAs, limited control)
Cloud-Deployed DB
An alternative to DBaaS-Systems

- **Idea**: Run (mostly) unmodified DB on IaaS

- **Method I: DIY**
  1. **Provision** VM(s)
  2. **Install** DBMS (manual, script, Chef, Puppet)

- **Method II: Deployment Tools**
  - `whirr` launch-cluster --config hbase.properties
  - Login, cluster-size etc.

- **Method III: Marketplaces**
Google BigQuery

- **Idea:** Web-scale analysis of nested data

---

**Google BigQuery**

Model: **Analytics-aaS**

Pricing:
- Storage + GBs
- Processed

API: **REST**

---

**New Query**

```
1. SELECT TOP(title, 5), COUNT(*)
2. FROM [publicdata:samples.wikipedia]
3. WHERE title CONTAINS "data";
```

---

**Query Results**

```
<table>
<thead>
<tr>
<th>Row</th>
<th>f0_</th>
<th>f1_</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Comparison of relational database systems</td>
<td>1320</td>
</tr>
<tr>
<td>2</td>
<td>Computer data storage</td>
<td>1319</td>
</tr>
<tr>
<td>3</td>
<td>Metadata</td>
<td>1097</td>
</tr>
<tr>
<td>4</td>
<td>Array data structure</td>
<td>852</td>
</tr>
<tr>
<td>5</td>
<td>Relational database</td>
<td>795</td>
</tr>
</tbody>
</table>
```
Google BigQuery

**Idea**: Web-scale analysis of nested data

```sql
1 SELECT TOP(title, 5), COUNT(*)
2 FROM [publicdata:samples.wikipedia]
3 WHERE title CONTAINS "data";
```

Query complete (2.4s elapsed, 6.79 GB processed)
Google BigQuery

- **Idea**: Web-scale analysis of nested data

![Diagram of multi-level execution tree on nested columnar data format](image)

- **Model**: Analytics-aaS
- **Pricing**: Storage + GBs Processed
- **API**: REST

---

**Dremel**

- **Idea**: Multi-Level execution tree on nested columnar data format ($\geq 100$ nodes)

Melnik et al. “Dremel: Interactive analysis of web-scale datasets”, VLDB 2010
Google BigQuery

**Idea:** Web-scale analysis of nested data

- **SLA:** 99.9% uptime / month
- Fundamentally different from relational DWHs and MapReduce
- Design copied by Apache Drill, Impala, Shark

---

**Model:**
- **Analytics-aaS**

**Pricing:**
- Storage + GBs Processed

**API:**
- REST
## Managed NoSQL services

### Summary

<table>
<thead>
<tr>
<th>Model</th>
<th>CAP</th>
<th>Scans</th>
<th>Sec. Indices</th>
<th>Largest Cluster</th>
<th>Learning</th>
<th>Lic.</th>
<th>DBaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HBase</strong></td>
<td>Wide-Column</td>
<td>CP</td>
<td>Over Row Key</td>
<td>&lt;cross&gt;</td>
<td>~700</td>
<td>1/4</td>
<td>Apache (EMR)</td>
</tr>
<tr>
<td><strong>MongoDB</strong></td>
<td>Document</td>
<td>CP</td>
<td>yes</td>
<td>&lt;check&gt;</td>
<td>&gt;100 &lt;500</td>
<td>4/4</td>
<td>GPL</td>
</tr>
<tr>
<td><strong>Riak</strong></td>
<td>Key-Value</td>
<td>AP</td>
<td>&lt;cross&gt;</td>
<td>&lt;check&gt;</td>
<td>~60</td>
<td>3/4</td>
<td>Apache (Softlayer)</td>
</tr>
<tr>
<td><strong>Cassandra</strong></td>
<td>Wide-Column</td>
<td>AP</td>
<td>With Comp. Index</td>
<td>&lt;check&gt;</td>
<td>&gt;300 &lt;1000</td>
<td>2/4</td>
<td>Apache</td>
</tr>
<tr>
<td><strong>Redis</strong></td>
<td>Key-Value</td>
<td>CA</td>
<td>Through Lists, etc.</td>
<td>manual</td>
<td>N/A</td>
<td>4/4</td>
<td>BSD</td>
</tr>
</tbody>
</table>

**Indices**

- **CP**: CAP-fulfilling
- **<cross>**: CAP-fulfilling
- **<check>**: CAP-fulfilling
- **Manual**
### Managed NoSQL services

#### Summary

<table>
<thead>
<tr>
<th>Model</th>
<th>CAP</th>
<th>Scans</th>
<th>Sec. Indices</th>
<th>Largest Cluster</th>
<th>Learning</th>
<th>Lic.</th>
<th>DBaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBase</td>
<td>Wide-Column</td>
<td>CP</td>
<td>Over Row Key</td>
<td>~700</td>
<td>1/4</td>
<td>Apache (EMR)</td>
<td></td>
</tr>
<tr>
<td>MongoDB</td>
<td>Document</td>
<td>CP</td>
<td>yes</td>
<td>&gt;100 &lt;500</td>
<td>4/4</td>
<td>GPL</td>
<td></td>
</tr>
<tr>
<td>Riak</td>
<td>Key-Value</td>
<td>AP</td>
<td>~60</td>
<td>3/4</td>
<td></td>
<td>Apache</td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td>AP</td>
<td>With</td>
<td></td>
<td></td>
<td></td>
<td>Apache</td>
<td>instaclustr</td>
</tr>
<tr>
<td>Redis</td>
<td>CAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BSD</td>
<td>Amazon</td>
</tr>
</tbody>
</table>

And there are many more:

- CouchDB (e.g. Cloudant)
- CouchBase (e.g. KuroBase Beta)
- ElasticSearch(e.g. Bonsai)
- Solr (e.g. WebSolr)
- ...
<table>
<thead>
<tr>
<th>Model</th>
<th>CAP</th>
<th>Scans</th>
<th>Sec. Indices</th>
<th>Queries</th>
<th>API</th>
<th>Scale-out</th>
<th>SLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleDB</td>
<td>CP</td>
<td><strong>Yes</strong> (as queries)</td>
<td>Automatic</td>
<td>SQL-like (no joins, groups, ...)</td>
<td>REST + SDKs</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DynamoDB</td>
<td>CP</td>
<td>By range key / index</td>
<td>Local Sec. Global Sec.</td>
<td>Key+Cond. On Range Key(s)</td>
<td>REST + SDKs</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Azure Tables</td>
<td>CP</td>
<td>By range key</td>
<td>✗</td>
<td>Key+Cond. On Range Key</td>
<td>REST + SDKs</td>
<td>✗</td>
<td>✗ 99.9% uptime</td>
</tr>
<tr>
<td>AE/Cloud DataStore</td>
<td>CP</td>
<td>Yes (as queries)</td>
<td>Automatic</td>
<td>Conjunct. of Eq. Predicates</td>
<td>REST/SDK, JDO,JPA</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>S3, Az. Blob, GCS</td>
<td>AP</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>REST + SDKs</td>
<td>✗</td>
<td>✗ 99.9% uptime (S3)</td>
</tr>
</tbody>
</table>
Big Data Frameworks
Hadoop Distributed FS (CP)

- **Master-Slave** Replication
  - **Namenode**: Metadata (files + block locations)
  - **Datanodes**: Save file blocks (usually 64 MB)
- **Design goal**: Maximum Throughput and data locality for Map-Reduce

### HDFS

<table>
<thead>
<tr>
<th>Model:</th>
<th>File System</th>
</tr>
</thead>
<tbody>
<tr>
<td>License:</td>
<td>Apache 2</td>
</tr>
<tr>
<td>Written in:</td>
<td>Java</td>
</tr>
</tbody>
</table>

HDD Size

- **1990**:
  - Size: 1.4 GB
  - Reading: 4.8 MB/s
  - \( \rightarrow 5 \text{ min/HDD} \)
- **2013**:
  - Size: 1 TB
  - Reading: 100 MB/s
  - \( \rightarrow 2.5 \text{ h/HDD} \)

Size:

- **1990**: 1.4 GB
- **2013**: 1 TB

Reading:

- **1990**: 4.8 MB/s
- **2013**: 100 MB/s
Holds filesystem data and block locations in RAM

Sends data operations to DataNodes and metadata operations to the NameNode

DataNodes communicate to perform 3-way replication

Files are split into blocks and scattered over DataNodes

Hadoop

- For many synonymous to *Big Data Analytics*
- Large Ecosystem
- Creator: Doug Cutting (Lucene)
- Distributors: Cloudera, MapR, HortonWorks
- Gartner Prognosis: By 2015 65% of all complex analytic applications will be based on Hadoop
- Users: Facebook, Ebay, Amazon, IBM, Apple, Microsoft, NSA

**Hadoop**

<table>
<thead>
<tr>
<th>Model:</th>
<th>Batch-Analytics Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>License:</td>
<td>Apache 2</td>
</tr>
<tr>
<td>Written in:</td>
<td>Java</td>
</tr>
</tbody>
</table>

http://de.slideshare.net/cultur...
MapReduce: Example
Constructing a reverse-index

Input (HDFS)  Mappers  Intermediate Output  Reducers  Output

doc1.txt

<table>
<thead>
<tr>
<th>cat</th>
<th>sat</th>
<th>mat</th>
</tr>
</thead>
</table>

Mappers

| cat, doc1.txt | sat, doc1.txt | mat, doc1.txt |

Intermediate Output

Reducers

<table>
<thead>
<tr>
<th>part-r-0000</th>
<th>cat: doc1.txt, doc2.txt</th>
</tr>
</thead>
</table>

| part-r-0001  | sat: doc1.txt, doc2.txt, dog: doc2.txt |

| part-r-0002  | mat: doc1.txt |

Holmes, Alex. Hadoop in Practice
Cluster Architecture

The client sends job and configuration to the JobTracker

The JobTracker coordinates the cluster and assigns tasks

TaskTrackers execute Mappers and Reducers as child-processes

Arun Murthy “Apache Hadoop YARN”
Cluster Architecture

YARN – Abstracting from MR

The ResourceManager is a pure scheduler.

Only the ApplicationMaster is Framework specific (e.g. MR)

Arun Murthy “Apache Haddop YARN”
Summary: Hadoop Ecosystem

- **Hadoop**: Ecosystem for Big Data Analytics
- **Hadoop Distributed File System**: scalable, shared-nothing file system for throughput-oriented workloads
- **Map-Reduce**: Paradigm for performing scalable distributed batch analysis
- **Other Hadoop projects**:  
  - **Hive**: SQL(-dialect) compiled to YARN jobs (Facebook)  
  - **Pig**: workflow-oriented scripting language (Yahoo)  
  - **Mahout**: Machine-Learning algorithm library in Map-Reduce  
  - **Flume**: Log-Collection and processing framework  
  - **Whirr**: Hadoop provisioning for cloud environments  
  - **Giraph**: Graph processing à la Google Pregel  
  - **Drill, Presto, Impala**: SQL Engines
Spark

- "In-Memory" Hadoop that does not suck for iterative processing (e.g. k-means)
- Resilient Distributed Datasets (RDDs): partitioned, in-memory set of records

 Spark
Example RDD Evaluation

- **Transformations**: RDD → RDD
- **Actions**: Reports an operation

```python
errors = sc.textFile("log.txt").filter(lambda x: "error" in x)
warnings = inputRDD.filter(lambda x: "warning" in x)
badLines = errorsRDD.union(warningsRDD).count()
```

![RDD Lineage Diagram](diagram)

H. Karau et al. "Learning Spark"
Storm

- Distributed Stream Processing Framework
- Topology is a DAG of:
  - **Spouts**: Data Sources
  - **Bolts**: Data Processing Tasks
- Cluster:
  - **Nimbus** (Master) ↔ **Zookeeper** ↔ **Worker**
Kafka

- Scalable, Persistent Pub-Sub
- Log-Structured Storage
- **Guarantee**: At-least-once
- **Partitioning**:
  - By Topic/Partition
  - Producer-driven
    - Round-robin
    - Semantic
- **Replication**:
  - Master-Slave
  - Synchronous to majority

---

**J. Kreps, N. Narkhede, J. Rao, und others, „Kafka: A distributed messaging system for log processing“**