NoSQL & Real-Time Data Management
In Research & Practice – Part 1

Wolfram Wingerath, Felix Gessert, Norbert Ritter
{wingerath, gessert, ritter}@informatik.uni-hamburg.de
March 5, BTW 2019, Rostock
Who We Are

Research:
• NoSQL & Cloud Databases
• Polyglot Persistence
• Database Benchmarking
• …

Practice:
 Backend-as-a-Service
 Web Caching
 Real-Time Database
 …

Norbert Ritter
Professor

Felix Gessert
CEO

Wolfram Wingerath
Developer
Slides: slides.baqend.com
Articles: blog.baqend.com
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:

- Analytics
- Reporting
- Data Mining

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

→ **Specialized** data systems
The Database Explosion

Sweetspots

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

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

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

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

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

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

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

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

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

- **Firebase**
  - Realtime BaaS
  - Communication and collaboration

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

- **bonsai**
  - Managed NoSQL
  - Full-Text Search

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

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

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

- **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 225 NoSQL systems
NoSQL Databases

- Two main motivations:

Scalability

User-generated data, Request load

Impedance Mismatch

<table>
<thead>
<tr>
<th>ID</th>
<th>Line Item 1: ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>Line Item2: ...</td>
</tr>
<tr>
<td>Payment: Credit Card, ...</td>
<td></td>
</tr>
</tbody>
</table>

Orders

Payment

Customers

Line Items
Scale-up vs Scale-out

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

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

RDBMS:

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

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

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

Examples: Cassandra (AP), Google BigTable (CP), HBase (CP)
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)
Search Platforms

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

![Diagram](image)

**POST /lectures/dis**

```json
{  "topic": "databases",
  "lecturer": "ritter",
  ...
}
```

**Search Server**

<table>
<thead>
<tr>
<th>Term</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>3,4,1</td>
</tr>
<tr>
<td>ritter</td>
<td>1</td>
</tr>
</tbody>
</table>

**Inverted Index**

**REST API**

**Doc. 1**

- Key: Value
- Key: Value
- Key: Value

**Doc. 3**

- Key: Value
- Key: Value
- Key: Value

**Doc. 4**

- Key: Value
- Key: Value
- Key: Value
Object-oriented Databases

- **Data model**: Classes, objects, relations (references)
- **Interface**: CRUD, queries, 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, querys (XPath, XQuery, 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
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 Data → Stream Processing → Notifications, Statistics, Aggregates, Recommendations, Models, Warnings

- Sensor Data & IOT
- Log Streams
- DB Change Streams
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

---

Eric Brewer, ACM-PODC Keynote, Juli 2000

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

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

Network partition

Value = \(V_1\)

Value = \(V_0\)

Replication

\[\text{Value} = V_0 \quad N_2\]

\[\text{Value} = V_1 \quad N_1\]
NoSQL Triangle

Every client can always read and write

CA
Oracle, MySQL, ...

CA
Oracle, MySQL, ...

All clients share the same view on the data

AP
Dynamo, Redis, Riak, Voldemort
Cassandra
SimpleDB

AP
Dynamo, Redis, Riak, Voldemort
Cassandra
SimpleDB

All nodes continue working under network partitions

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

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

Data models

Relational
Key-Value
Wide-Column
Document-Oriented

Nathan Hurst: Visual Guide to NoSQL Systems
http://blog.nahurst.com/visual-guide-to-nosql-systems
**PACELC** – an alternative CAP formulation

- **Idea**: Classify systems according to their behavior during network partitions

- Partition yes → **Availability** (AL - Dynamo-Style) → Cassandra, Riak, etc.
- Partition no → **Consistency** (AC - MongoDB) → HBase, BigTable and ACID systems
- **Consistency** (CC – Always Consistent)

No consequence of the CAP theorem

---

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:

\[ \begin{align*}
\text{Write A}=1 & \quad \text{Write B}=1 \\
\text{Read B} & \quad \text{Read A}
\end{align*} \]

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

Some weaker isolation levels allow high availability:


Impossibility Results
Consensus Algorithms

- 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

„Gold standard“ for RDBMSs

BASE
- Basically Available
- Soft State
- Eventually Consistent

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></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>Depends</td>
<td>RR</td>
</tr>
<tr>
<td>IBM Informix 11.50</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>MySQL 5.6</td>
<td></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>RC</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 Requirements from the application

Central techniques NoSQL databases employ

Operational Requirements
NoSQL Database Systems:
A Survey and Decision Guidance

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

Universität Hamburg, Germany
{gessert, wingerath, friedrich, ritter}@informatik.uni-hamburg.de

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

1 Introduction

Traditional relational database management systems (RDBMSs) provide powerful mechanisms to store and query structured data under strong consistency and transaction guarantees and have reached an unmatched level of reliability, stability and support through decades of development. In recent years, however, the amount of useful data in some application areas has become so vast that it cannot be stored or processed by traditional database solutions. User-generated content in social networks or data retrieved from large sensor networks are only two examples of this phenomenon commonly referred to as Big Data. 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
NoSQL Databases: a Survey and Decision Guidance

Felix Gessert
Aug 15, 2016 - 26 min read

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.

As a blog post: blog.baqend.com
Functional Techniques Non-Functional

Scan Queries
ACID Transactions
Conditional or Atomic Writes
Join
Sorting

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

Data Scalability
Write Scalability
Read Scalability
Elasticity
Sharding (aka Partitioning, Fragmentation)
Scaling Storage and Throughput

- Horizontal distribution of data over nodes

**Partitioning strategies**: Hash-based vs. Range-based

**Difficulty**: Multi-Shard-Operations (join, aggregation)
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 changeable

Implemented in
- MongoDB, Riak, Redis, Cassandra, Azure Table, Dynamo
- BigTable, HBase, DocumentDB, Hypertable, MongoDB, RethinkDB, Espresso
- G-Store, MegaStore, Relational 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
- **Pro**: Fast writes, no coordination needed
- **Contra**: Replica data potentially stale (inconsistent)

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

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

Replication: Where

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

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

Synchronous Replication

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

Achievable with high availability

Causal Consistency

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

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

Linearity

Reads a client reads in steps monotonically.

Monotonic Reads

One session are ordered on all.

Monotonic Writes

Version-based or

Bounded Staleness

Writes in one session are strictly ordered on all replicas.

Versions a client reads in a session increase monotonically.

Clients directly see their own writes.

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

Read Your Writes

Writes Follow Reads


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.

Reference:
- [https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/](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.

https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/

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.

---

https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/

**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/
PRAM and Causal Consistency

- Combinations of previous session consistency guarantees:
  - PRAM = MR + MW + RYW
  - Causal Consistency = PRAM + WFR

- All consistency levels 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

---

Functional Techniques

Non-Functional

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

Storage Management

Read Latency
Write Throughput
Durability
NoSQL Storage Management
In a Nutshell

<table>
<thead>
<tr>
<th>Speed, Cost</th>
<th>Size</th>
<th>Volatile</th>
<th>Durable</th>
<th>Typical Uses in DBMSs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td></td>
<td></td>
<td></td>
<td>• Caching</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Primary Storage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Data Structures</td>
</tr>
</tbody>
</table>

| HDD         |      |          |         | • Logging               |
|             |      |          |         | • Primary Storage       |

| SSD         |      |          |         | • Caching               |
|             |      |          |         | • Logging               |
|             |      |          |         | • Primary Storage       |

- **RAM**: High Performance
- **SSD**: High Performance
- **HDD**: Low Performance

**Typical Uses**:
- **RR**: Random Reads
- **RW**: Random Writes
- **SR**: Sequential Reads
- **SW**: Sequential Writes

- **Caching**
- **Primary Storage**
- **Data Structures**

**Logging**

**Append-Only I/O**: Promotes durability of write operations.

**Update-In-Place**: Improves latency.

**In-Memory/Caching**: Increases write throughput.

**Data In-Memory/Caching**: Is good for read latency.
Local Secondary Indexing
Partitioning By Document

<table>
<thead>
<tr>
<th>Data</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>Color</td>
</tr>
<tr>
<td>12</td>
<td>Red</td>
</tr>
<tr>
<td>56</td>
<td>Blue</td>
</tr>
<tr>
<td>77</td>
<td>Red</td>
</tr>
<tr>
<td>Term</td>
<td>Match</td>
</tr>
<tr>
<td>Red</td>
<td>[ ]</td>
</tr>
<tr>
<td>Blue</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

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

WHERE color=blue

Scatter-gather query pattern.
Global Secondary Indexing
Partitioning By Term

### Partition I

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

**Data Index**

#### Index

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

**Consistent index maintenance requires distributed transaction.**

### 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>

**Match**

[12, 77]

### Implemented in

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

**Targeted Query**

WHERE color=blue

Query Processing Techniques

Summary

- **Local Secondary Indexing**: Fast writes, scatter-gather queries
- **Global Secondary Indexing**: Slow or inconsistent writes, fast queries
- **(Distributed) Query Planning**: scarce in NoSQL systems but increasing (e.g. left-outer equi-joins in MongoDB and \( \theta \)-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
NoSQL Landscape

Document
- MongoDB
- Amazon DynamoDB
- CouchDB

Wide Column
- HBase
- Google Datastore
- Cassandra

Key-Value
- Redis
- Riak

Graph
- Neo4j
- RavenDB
- InfiniteGraph

Project Voldemort
- Couchbase
## Popularity (Feb 2019)

<table>
<thead>
<tr>
<th>#</th>
<th>System</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Oracle</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>2.</td>
<td>MySQL</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>3.</td>
<td>MS SQL Server</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>4.</td>
<td>PostgreSQL</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>5.</td>
<td>MongoDB</td>
<td>Document store</td>
</tr>
<tr>
<td>6.</td>
<td>DB2</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>7.</td>
<td>Microsoft Access</td>
<td>Relational</td>
</tr>
<tr>
<td>8.</td>
<td>Redis</td>
<td>Key-value store</td>
</tr>
<tr>
<td>9.</td>
<td>ElasticSearch</td>
<td>Search engine</td>
</tr>
<tr>
<td>10.</td>
<td>SQLite</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>11.</td>
<td>Cassandra</td>
<td>Wide column store</td>
</tr>
<tr>
<td>12.</td>
<td>MariaDB</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>13.</td>
<td>Splunk</td>
<td>Search engine</td>
</tr>
<tr>
<td>14.</td>
<td>Teradata</td>
<td>Search engine</td>
</tr>
<tr>
<td>15.</td>
<td>Hive</td>
<td>Relational</td>
</tr>
<tr>
<td>16.</td>
<td>Solr</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>17.</td>
<td>HBase</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>18.</td>
<td>FileMaker</td>
<td>Relational</td>
</tr>
<tr>
<td>19.</td>
<td>SAP Adaptive Server</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>20.</td>
<td>SAP HANA</td>
<td>Relational DBMS</td>
</tr>
<tr>
<td>21.</td>
<td>Amazon DynamoDB</td>
<td>Multi-model</td>
</tr>
<tr>
<td>22.</td>
<td>Neo4j</td>
<td>Graph DB</td>
</tr>
<tr>
<td>23.</td>
<td>Couchbase</td>
<td>Document store</td>
</tr>
<tr>
<td>24.</td>
<td>Memcached</td>
<td>Key-value store</td>
</tr>
<tr>
<td>25.</td>
<td>SQL Azure</td>
<td>Multi-model</td>
</tr>
</tbody>
</table>

**Scoring:** Google/Bing results, Google Trends, Stackoverflow, job offers, LinkedIn

[http://db-engines.com/de/ranking](http://db-engines.com/de/ranking)
NoSQL: Still a Thing in 2019
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
Consistent Hashing

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

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

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

![Diagram showing consistent hashing with nodes A, B, C, D, E, F connected in a circle, with hash(key) and position = hash(ip) annotations.]
Consistent Hashing

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

![Diagram showing consistent hashing with virtual nodes]

- B takes over two thirds of A
- C takes over one third of A
- Range transferred
Reading
Parameters R, W, N

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

N=3
R=2
W=1
Quorums

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

Quorum examples:

- $N = 12, R = 3, W = 10$
- $N = 12, R = 7, W = 6$
Writing

- **W** Servers have to acknowledge
Hinted Handoff

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

N=3
R=2
W=1
Vector clocks

- Dynamo uses **Vector Clocks** for versioning

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
Conflict Resolution

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

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

### Typical Configurations:

<table>
<thead>
<tr>
<th>Performance (Cassandra Default)</th>
<th>N=3, R=1, W=1</th>
</tr>
</thead>
<tbody>
<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): $P(\text{consistent}) \geq 99.9\%$ nach 1.85 ms

$R + W > N$ does not imply linearizability

Consider the following execution:

**Diagram:**
- **Writer** sets $x = 1$.
- **Replica 1** receives the set operation.
- **Replica 2** and **Replica 3** also receive the set operation.
- **Reader A** reads $x$ and gets $1$.
- **Reader B** reads $x$ and gets $0$.

CRDTs
Convergent/Commutative Replicated Data Types

- **Goal**: avoid manual conflict-resolution

- **Approach**:
  - **State-based** – commutative, idempotent merge function
  - **Operation-based** – broadcasts of commutative updates

- **Example**: State-based Grow-only-Set (G-Set)

Node 1

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

Node 2

- $S_2 = \{\}$
- $S_2 = \{y\}$
- $S_2 = \text{merge}(\{y\}, \{x\})$
  - $S_2 = \{x, y\}$
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

**Riak**

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

**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/]
Hooks & Search

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

- **Riak Search:**
  - /solr/mybucket/select?q=user:emil
  - Search Index
    - Term | Dokument
    - database | 3,4,1
    - rabbit | 2
```javascript
function(v) {
  var json = v.values[0].data;
  return [{count: json.stackoverflow_questions}];
}

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

- 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

Amazon S3 API

1MB Chunks

F

E

D

C

B

A

Stanchion: Request Serializer

Files
Summary: Dynamo and Riak

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

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

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

- Redis Codebase $\approx$ 20K LOC

![Redis Diagram]

- Client
- Plain Text Protocol
- TCP Port 6379
- Local Filesystem
- SET mykey hello
- +OK
- Event Loop
- One Process/Thread
- Periodic
- After X Writes
- SAVE
- AOF
- RDB
- Log Dump
- RAM
Persistance

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

```bash
config set appendonly everysec
```

```c
fsync() every second
```

Snapshot every 60s, if > 1000 keys changed

```bash
config set save 60 1000
```
Persistence

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

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

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

Latency > Disk Latency, Group Commits, Slow

Periodic `fsync()`, data loss limited

Data corruption and loss possible

Data loss possible, corruption prevented
Master-Slave Replication

> SLAVEOF 192.168.1.1 6379
< +OK
Data structures

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

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

- **(Linked) Lists:**
  - LPUSH
  - LPUSHX
    - Only if list exists
  - LPOP
  - LREM
  - LPUSHX
    - Only if list exists
  - LRANGE
  - BLPOP
  - RPOP
  - LINDEX
  - LLLEN
  - LRANGE
    - inbox 1 2
    - inbox 0 3466
  - LINDEX
    - inbox 2
  - RPOP
Data Structures

Sets:

- `SADD`: Add elements to a set.
- `SREM`: Remove elements from a set.
- `SCARD`: Get the size of a set.
- `SINTER`: Intersect sets.
- `SINTERSTORE`: Intersect sets and store the result.
- `SMEMBERS`: Get all members of a set.
- `SISMEMBER`: Check if an element is a member of a set.
- `SRANDMEMBER`: Get a random member from a set.

Example:
- `user:2:friends`:
  - `SADD 23 76 233`
  - `SCARD` results in `4`

- `user:5:friends`:
  - `SADD 23 10 2 28 325 64 70`

- `SINTER` operation between `user:2:friends` and `user:5:friends` results in `23`.

- `SINTERSTORE` operation stores the intersection in `common_friends` set.

- `SMEMBERS` of `common_friends` returns `23`.
Data Structures

- Pub/Sub:

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

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

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

![diagram](https://github.com/Baqend/Orestes-Bloomfilter)
Bloomfilters in Redis

- Bitvectors in Redis: String + SETBIT, GETBIT, BITOP

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

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

**Jedis**: Redis Client for Java

**SETBIT** creates and resizes automatically.
Pipelining

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

![Diagram of Redis Pipelining]
Redis for distributed systems

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

- Data in RAM
- Single-threading (operations are lock-free)
- AOF
- No query parsing
- Hand-coded optimizations

Pessimistic transactions are expensive

- Useful work: 6.8%
- Buffer manager: 34.6%
- Loading: 14.2%
- Locking: 11.9%

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

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

SCRIPT LOAD

Script Hash

EVALSHA $hash 1 "mylock" "10"

1

Redis Server

Script Cache

Script Hash

EVALSHA $hash 1 "mylock" "10"

1

Redis Cluster
Native Sharding in Redis

- **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`
Performance

- 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

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

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

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

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

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

Wide-Column Data Modelling

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

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

  **Column-Family: contents**
  - content: "<html>..."
  - com.cnn.www
  - Sorted

  **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></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 Lock, Root Metadata Tablet**
- **Tablet Server**
- **Chubby**
- **GFS**

**Tablet Server**
- Stores Ranges, Answers client requests
- Stores data and commit log

**Master**
- ACLs, Garbage Collection, Rebalancing

**Commit Log**

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

- Writes: In-Memory in Memtable
- SSTable disk access optimized by Bloom filters
Apache HBase (CP)

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

- **Logical to physical mapping:**

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

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

Example: Facebook Insights

Log extraction every 30 min to HBase

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

<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>

Atomic HBase Counter

CF:Daily

CF:Monthly

CF:All

TTL – automatic deletion of old rows

Lars George: “Advanced HBase Schema Design”
Schema Design

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

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

---

## Schema: Messages

<table>
<thead>
<tr>
<th>User ID</th>
<th>CF</th>
<th>Column</th>
<th>Timestamp</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>data</td>
<td>5fc38314-e290-ae5da5fc375d</td>
<td>1307097848</td>
<td>&quot;Hi Lars, ...&quot;</td>
</tr>
<tr>
<td>12345</td>
<td>data</td>
<td>725aae5f-d72e-f90f3f070419</td>
<td>1307099848</td>
<td>&quot;Welcome, and ...&quot;</td>
</tr>
<tr>
<td>12345</td>
<td>data</td>
<td>cc6775b3-f249-c6dd2b1a7467</td>
<td>1307101848</td>
<td>&quot;To Whom It ...&quot;</td>
</tr>
<tr>
<td>12345</td>
<td>data</td>
<td>dcbree495-6d5e-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-dcbree495-6d5e-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.

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

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

TCP Cluster Messages

Stores Rows

Stores SSTables and Commit Log

Stateful Communication

Replication, Gossip, etc.

Local Filesystem

Key Cache

MemTable

Row Cache

Column Family Store

Storage Proxy

Stateful Communication

Stores Primary Key Index (Seek Position)

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

Diagram:

- **C₁:** writes B
  - Write(One)
  - Version C

- **C₂:** writes C
  - Write(One)
  - Version C

- **C₃:** reads C
  - Read(All)
  - Version C

Version C

Version C

Version C
Storage Layer

- Uses BigTables Column Family Format

**KeySpace**: music

**Column Family**: songs

- **Row Key**: Mapping to Server
- **f82831…**: title: Andante
- **144052…**: title: Jailhouse Rock

**Comparator** determines order

- **Sparse**

**Type validated by Validation Class UTFType**

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

http://www.datastax.com/dev/blog/cql3-for-cassandra-experts
CQL Example: Compound keys

- Enables Scans despite Random Partitioner

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

```
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>

**Partition Key**

**Clustering Columns:** sorted per node
Other Features

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

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

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

- 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), ...

<table>
<thead>
<tr>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model:</td>
</tr>
<tr>
<td>Document</td>
</tr>
<tr>
<td>License:</td>
</tr>
<tr>
<td>GNU AGPL 3.0</td>
</tr>
<tr>
<td>Written in:</td>
</tr>
<tr>
<td>C++</td>
</tr>
</tbody>
</table>
Basics

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

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

**Principles**

- Denormalisation instead of joins
- Nesting replaces 1:n and 1:1 relations
- Schemafreeness: Attributes per document
- Unit of atomicity: document
**Sharding und Replication**

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

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

- Receives all **writes**
- **Replicates** asynchronously

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

HTTP

Server

Client

Movies
Tweets

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

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

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

One of the best movies, that
fs = new GridFs(db);
fs.createFile(inputStream).save();
Geospatial Queries:
- Distance
- Intersection
- Inclusion
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:**
  - **Skip and Limit:**
  - **Grouping,** e.g.
    ```
    {_id: "$author",
    docsPerAuthor: { $sum: 1 },
    viewsPerAuthor: { $sum: "$views" }}
    ```

- Alternative: JavaScript MapReduce
Sharding

- Range-based:
  - 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:

- Migration:
# Classification: MongoDB Techniques

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

- Schema-free JSON store
- Allows complex queries, full-text search, aggregation, facets,...
- Local indexing
- **Hash-based sharding**, but custom routing available
- Synchronous replication
- Storage Management:
  - Write-ahead logging
  - **Lucene** for local data storage
Classification: Elasticsearch

Techniques

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

- **Accumolo** (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)
- **Google Cloud Spanner** (Improved F1 as a Service)
- **Microsoft Cloud SQL Server** (distributed CP, MSSQL-comp.)
- **MySQL Cluster, Galera Cluster, Percona XtraDB Cluster** (distributed storage engine for MySQL)
Summary

- **HDFS and Hadoop**: Map-Reduce platform for batch analytics
- **Spark, Kafka, Storm**: In-Memory & Real-Time Analytics
- **Dynamo and Riak**: KV-store with consistent hashing
- **Redis**: replicated, in-memory KV-store
- **BigTable, HBase, Cassandra**: wide-column stores
- **MongoDB**: sharded and replicated document store
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?
**Distributed Transactions**

**ACID and Serializability**

**Definition:** A transaction is a sequence of operations transforming the database from one consistent state to another.

- Atomicity
- Consistency
- Isolation
- Durability

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

General Processing

Commit Protocol

Commit Protocol is not available

Needs to ensure globally correct isolation

Strong Consistency – needed by Concurrency Control

Concurrency Control

Replication

Replicas

Shard

Replication

Replicas

Shard

Replication

Replicas

Shard
## 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 Wide-Area Replication

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

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

Paxos Instance

Replicas

App-Server (Coordinator)

Record-Master (v)

Record-Master (u)

Replicas
Distributed Transactions
RAMP – Read Atomic Multi Partition Transactions

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

Fractured Read:

\[
\begin{align*}
&\text{read objects} \\
&\text{validate} \\
&\text{load other version}
\end{align*}
\]
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
Our line of work for improving scalable transaction processing.
Problem of Optimistic Transactions
Abort Rates Depend on Latency

Transaction Abort Rates Increase Exponentially with Latency
Distributed Cache-Aware Transaction
Scalable ACID Transactions

1. Cache Sketch: **staleness barrier** at transaction begin
2. Shorter duration through **cached reads**
3. Optimistic commit on top of **NoSQL systems**
Distributed Cache-Aware Transaction
Speed Evaluation

- 15× Faster Transactions
- 7× More Objects Before Exceeding 2 Seconds
Selected Research Challenges

Encrypted Databases

- Example: CryptDB
- Idea: Only decrypt as much

SQL-Proxy
Encrypts and decrypts,

Relational Cloud

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

Early approach
- Not adopted in practice, yet

Dream solution: Full Homomorphic Encryption

## Research Challenges

### Transactions and Scalable Consistency

<table>
<thead>
<tr>
<th>System</th>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamo</td>
<td>Eventual</td>
</tr>
<tr>
<td>Yahoo PNuts</td>
<td>Timeline per key</td>
</tr>
<tr>
<td>COPS</td>
<td>Causality</td>
</tr>
<tr>
<td>MySQL (async)</td>
<td>Serializable</td>
</tr>
<tr>
<td>Megastore</td>
<td>Serializable</td>
</tr>
<tr>
<td>Spanner/F1</td>
<td>Snapshot isolation</td>
</tr>
<tr>
<td>MDCC</td>
<td>Read-Committed</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

Currently very few NoSQL DBs implement consistent Multi-DC replication.
Selected Research Challenges
NoSQL Benchmarking

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

<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
Selected Research Challenges

NoSQL Benchmarking

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


**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

Fast Lookups
- RAM
  - Access
    - Volume
      - Unbounded
        - CAP
          - AP
            - Redis, Memcache
          - CP
            - Cassandra, Riak, Voldemort, Aerospike
    - Complex Queries
      - HDD-Size
        - Consistency
          - ACID
            - RDBMS, Neo4j, RavenDB, MarkLogic
          - Availability
            - CouchDB, MongoDB, SimpleDB
      - Unbounded
        - Query Pattern
          - Ad-hoc
            - MongoDB, RethinkDB
          - Analytics
            - Hadoop, Spark, Parallel DWH, Cassandra, HBase, Riak, MongoDB

Example Applications
- Shopping-Basket
- Order History
- Social Network

Purpose:
Application Architects: narrowing down the potential system candidates based on requirements
Database Vendors/Researchers: clear communication and design of system trade-offs
System Properties
According to the NoSQL Toolbox

- For fine-grained system selection:

<table>
<thead>
<tr>
<th></th>
<th>Scan Queries</th>
<th>ACID Transactions</th>
<th>Conditional Writes</th>
<th>Joins</th>
<th>Sorting</th>
<th>Filter Query</th>
<th>Full-Text Search</th>
<th>Analytics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mongo</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redis</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Riak</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></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>MySQL</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</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>Non-functional Requirements</th>
<th>Data Scalability</th>
<th>Write Scalability</th>
<th>Read Scalability</th>
<th>Elasticity</th>
<th>Consistency</th>
<th>Write Latency</th>
<th>Read Latency</th>
<th>Write Throughput</th>
<th>Read Availability</th>
<th>Write Availability</th>
<th>Durability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mongo</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redis</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Riak</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
System Properties
According to the NoSQL Toolbox

For fine-grained system selection:

<table>
<thead>
<tr>
<th></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>x</td>
<td>x</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></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Redis</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>HBase</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Riak</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
Literature
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
High-Level NoSQL Categories:
- Key-Value, Wide-Column, Document, Graph
- Two out of \{Consistent, Available, Partition Tolerant\}

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

- **Decision Tree**
Summary

- Current NoSQL systems very good at scaling:
  - Data storage
  - Simple retrieval
- But how to handle real-time queries?
NoSQL & Real-Time Data Management
In Research & Practice – Part 2

Wolfram Wingerath, Felix Gessert, Norbert Ritter
{wingerath, gessert, ritter}@informatik.uni-hamburg.de
March 5, BTW 2019, Rostock
Outline

Introduction
Where From? Where To?

Stream Processing
Big Data + Low Latency

Real-Time Databases
Push-Based Collections

Future Directions
Current Research & Outlook

- A Short History of Data Management
- Database Management:
  - Triggers, ECA rules
  - Materialized Views, Change Notifications
- Data Stream Management:
  - General Architecture
  - Stream Operators
  - Approximation & Sampling
  - CEP
A Short History of Data Management
Hot Topics Through The Ages

Relational Databases
- Entity-Relationship Model
- Triggers
- Ingres
- SQL Standard
- HiPAC
- System R
- PostgreSQL

CEP & Streams
- Starburst
- Telegraph
- MapReduce
- STREAM
- Spark
- Bigtable
- Aurora & Borealis
- GFS
- Dynamo
- Storm
- Flink
- Samza
- RethinkDB

Stream Processing
- Baqend

Active Databases
- Relational Model

Big Data & NoSQL
- PostgreSQL
- HiPAC
- System R
- Relational Model

Real-Time Databases
- Baqend

Hot Topics Through The Ages
- 1970
- 1980
- 1990
- 2000
- 2010
- Today
TRIGGERS & MORE

Active Database Features
Databases are **Passive**

Challenge: How to Build **Reactive** Applications?

What’s the current state?

Periodic Polling for query result maintenance:
→ inefficient
→ slow
Active Database Features
Modeling Behavioral Domain Aspects

**Triggers**: simple action-mechanisms
- Use cases:
  - (Referential) integrity
  - Change data capture

**ECA rules**: Event-Condition-Action
- Captures **composite events**
- More expressive than triggers
  - **rule languages**
- Advanced use cases:
  - Materialized view maintenance
  - Pattern recognition
  - (complex) event processing
View Maintenance
Keeping Track of Query Results

**Materialized Views:** precomputed query results
- Used to speed up pull-based queries, e.g., in data warehouses
- Implementation aspects:
  - Eager vs. lazy
  - Incremental vs. recomputation-based
  - Partial maintenance vs. full maintenance
  - Self-maintainability vs. expressiveness

**Change Notification Mechanisms:** inform subscribers of possibly invalidated query results
- Used to invalidate caches in the middle tier (cf. 3-tier stack)
View Maintenance By Example
Matching Every Query Against Every Update

→ Potential *bottlenecks*:
  • *Number of queries*
  • *Write throughput*
  • *Query complexity*

Similar processing for:
  • Triggers
  • ECA rules
EVOLVING DOMAINS

Data Stream Management
Data Stream Management Systems
High-Level Architecture
Typical Stream Operators

Examples

Filter & Transform

Filter
Map

Group
GroupByKey

Aggregates
SUM()
COUNT()

Windows
Tumbling
Sliding

https://www.infoq.com/presentations/stream-processors-databases
https://www.infoq.com/presentations/stream-processing-apache-flink
Complex Event Processing
Detecting Patterns

- **Abstraction** from raw event streams
- Detection of **relationships** between events
- Often modeled in abstraction **hierarchies**

**Techniques:**
- Transformation, filtering
- Correlation, aggregation, ...
- Pattern detection → **low-level events**
- → **composite events**

Notions of Time

Arrival Time vs. Event Time

- **Arrival time**: When was the event received?
- **Event time**: When did the event occur?
- **Clock Skew**: difference between arrival and event time

Approximation & Load Shedding
Provide the „Best“ Answer While Avoiding to Fall Behind

- **Sampling**: can be optimized for different things, e.g.
  - Position stream (e.g. „select every 10th item“)
  - Value (e.g. hash partitioning)
  - Semantic criteria
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Database</th>
<th>Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Update rate</strong></td>
<td>Low</td>
<td>High, bursty</td>
</tr>
<tr>
<td><strong>Primitive</strong></td>
<td>Persistent collections</td>
<td>Transient streams</td>
</tr>
<tr>
<td><strong>Temporal scope</strong></td>
<td>Historical</td>
<td>Windowed</td>
</tr>
<tr>
<td><strong>Access</strong></td>
<td>random</td>
<td>sequential</td>
</tr>
<tr>
<td><strong>Queries</strong></td>
<td>One-time</td>
<td>Continuous</td>
</tr>
<tr>
<td><strong>Query Plans</strong></td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>Accurate</td>
<td>Approximate</td>
</tr>
</tbody>
</table>
Outline

Introduction
Where From? Where To?

Stream Processing
Big Data + Low Latency

Real-Time Databases
Push-Based Collections

Future Directions
Current Research & Outlook

• **Big Picture:**
  - Processing Pipelines
  - Stream vs. Batch
  - Lambda vs. Kappa Architecture

• **System Survey:**
  - Storm/Trident
  - Samza
  - Spark Streaming
  - Flink

• **Discussion:**
  - Comparison Matrix
  - Other Systems
OVERVIEW

Scalable Data Processing
A Data Processing Pipeline

We are here!
Data Processing Frameworks

Scale-Out Made Feasible

Data processing frameworks hide complexities of scaling, e.g.:

- **Deployment** - code distribution, starting/stopping work
- **Monitoring** - health checks, application stats
- **Scheduling** - assigning work, rebalancing
- **Fault-tolerance** - restarting workers, rescheduling failed work

Running in cluster

Running on single node

Scaling out
Big Data Processing Frameworks
What are your options?

What to use when?

Spark Streaming
Google Dataflow
Heron
Spark
STORM
STORM Trident
IBM InfoSphere Streams
Amazon Elastic MapReduce
Flink
APEX
coral
samza
hadoop
kafka streams
concord
CONCEPTS

Batch vs. Stream Processing
Batch Processing

„Volume“

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

**But:**
- **High latency**: periodic jobs (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)
  → „**eventually accurate**“ merged views of real-time & batch
  Typical setups: Hadoop + Storm (→ Summingbird), Spark, Flink
- **High complexity** 2 code bases & 2 deployments

Nathan Marz, *How to beat the CAP theorem* (2011)
http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
Kappa Architecture

$\text{Stream}(D_{all}) = \text{Batch}(D_{all})$

- **Simpler** than Lambda Architecture
- **Data retention** for history
- Reasons against Kappa:
  - Existing **legacy batch system**
  - **Special tools** only for a particular batch processor
  - Only **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**

**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
SURVEY

Popular Stream Processing Systems
Processing Models
Batch vs. Micro-Batch vs. Stream

- **stream**
  - Flink
  - STORM Trident
  - samza

- **micro-batch**
  - Hadoop
  - Spark Streaming

- **batch**
  - Amazon Elastic MapReduce

**low latency**

**high throughput**
Storm
„Hadoop of real-time“

Overview
- First production-ready, well-adopted stream processor
- Compatible: native Java API, Thrift, distributed RPC
- Low-level: no primitives for joins or aggregations
- Native stream processor: latency < 50 ms feasible
- Big users: Twitter, Yahoo!, Spotify, Baidu, Alibaba, ...

History
- 2010: developed at BackType (acquired by Twitter)
- 2011: open-sourced
- 2014: Apache top-level project
Dataflow

Directed Acyclic Graphs (DAG):
- **Spouts**: pull data into topology
- **Bolts**: do processing, emit data
- Asynchronous
- Lineage can be tracked for each tuple
  → At-least-once has 2x messaging overhead
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
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:
  - High-level API: aggregations & joins
  - Strong ordering
  - Stateful exactly-once processing
    → Performance penalty
Trident
Partitioned Micro-Batching

Illustration taken from: “Storm applied”, Sean T. Allen et al.
Samza
Real-Time on Top of Kafka

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)
Dataflow
Simple By Design

- **Job**: processing step (≈ Storm bolt)
  - Robust
  - But: often several jobs
- **Task**: job instance (parallelism)
- **Message**: single data item
- **Output persisted** in Kafka
  - Easy data sharing
  - 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
  - Simple recovery
- Fast lookups

Illustrations taken from: Jay Kreps, *Why local state is a fundamental primitive in stream processing* (2014)
State Management
Straightforward Recovery

Spark
„MapReduce successor“

Overview

- **High-level API**: immutable collections (RDDs)
- **Community**: 1000+ contributors in 2015
- **Big users**: Amazon, eBay, Yahoo!, IBM, Baidu, ...

History

- **2009**: developed at UC Berkeley
- **2010**: open-sourced
- **2014**: Apache top-level project
Spark Streaming

Overview

- **High-level API**: DStreams (~Java 8 Streams)
- **Micro-Batching**: seconds of latency
- **Rich features**: stateful, exactly-once, elastic

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 reduce recovery time

**DStream**: Discretized RDD
- RDDs are processed in **order**: no ordering within RDD
- RDD scheduling ~50 ms → latency >100ms

Illustration taken from:
http://spark.apache.org/docs/latest/streaming-programming-guide.html#overview (2017-02-26)
Example
Counting Page Views

```javascript
pageViews = readStream("http://...", "1s")
ones = pageViews.map(event => (event.url, 1))
counts = ones.runningReduce((a, b) => a + b)
```
Flink

Overview

- **Native stream processor**: Latency <100ms feasible
- **Abstract API** for stream and batch processing, stateful, exactly-once delivery
- **Many libraries**: Table and SQL, CEP, Machine Learning, Gelly...
- **Users**: Alibaba, Ericsson, Otto Group, ResearchGate, Zalando...

History

- **2010**: start as **Stratosphere** at TU Berlin, HU Berlin, and HPI Potsdam
- **2014**: Apache Incubator, project renamed to Flink
- **2015**: Apache top-level project
Architecture
Streaming + Batch

DataStream (Java / Scala)

DataStream (Java / Scala)

Dataset (Java / Scala)

DataStream (Java / Scala)

Streaming dataflow runtime

YARN
Cluster
Local

https://www.infoq.com/presentation/s/stream-processing-apache-flink
Managed State
Streaming + Batch

- Automatic **Backups** of local state
- Stored in **RocksDB**, Savepoints written to **HDFS**

---

https://www.infoq.com/presentation/s/stream-processing-apache-flink
Highlight: Fault Tolerance
Distributed Snapshots

- **Ordering** within stream partitions
- **Periodic checkpoints**
- **Recovery:**
  1. *reset state* to checkpoint
  2. *replay data* from there

Illustration taken from:
WRAP UP

Side-by-side comparison
## Comparison

<table>
<thead>
<tr>
<th></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>$&lt;&lt;100$ 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>no (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>
Performance
Yahoo! Benchmark

- Based on **real use case**:
  - Filter and count ad impressions
  - 10 minute windows

“**Storm [...] and Flink [...] show sub-second latencies** at relatively high throughputs with **Storm** having the **lowest 99th percentile** latency. **Spark** streaming [...] supports high throughputs, but at a relatively **higher latency.**”

Other Systems

Heron

Apex

Dataflow

Beam

Kafka Streams

IBM InfoSphere Streams

And even more: Kinesis, Gearpump, MillWheel, Muppet, S4, Photon, ...
Summary

- Stream Processors:
  - Storm
  - Flink
  - Samza
  - Spark Streaming

- Many Dimensions of Interest: consistency guarantees, state management, backpressure, ordering, elasticity, ...
Outline

• **Introduction**: Where From? Where To?

• **Stream Processing**: Big Data + Low Latency

• **Real-Time Databases**: Push-Based Collections

• **Future Directions**: Current Research & Outlook

• **Big Picture**:
  - Why Push-Based Database Queries?
  - Where Do Real-Time Databases Fit in?

• **System Survey**:
  - Meteor
  - RethinkDB
  - Parse
  - Firebase

• **Discussion**:
  - Comparison Matrix
  - Other Systems
REAL-TIME DBS

Making Databases Push-Based
Traditional Database Access
No Request? No Data!

What’s the current state?

Periodic Polling
→ inefficient
→ slow
Real-time Databases
Always In-Sync With Database State

Real-Time Queries for query result maintenance:
→ efficient
→ fast
Quick Comparison

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

Database Management

- static collections
- pull-based

Real-Time Database Management

- evolving collections
- push-based

Data Stream Management

- persistent/ephemeral streams

Stream Processing

- ephemeral streams
REAL-TIME DBS

System Survey
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

*MongoDB cluster (3 shards)*

![Diagram showing MongoDB cluster with primary and secondary servers connected through oplog]

- **Primary A**
- **Primary B**
- **Primary C**
- **Secondary C1**
- **Secondary C2**
- **Secondary C3**
Oplog Tailing
Tapping into the Oplog

- Every Meteor server receives all DB writes through oplogs

MongoDB cluster (3 shards)

Primary A  Primary B  Primary C

Oplog broadcast

query (when in doubt)

monitor oplog

push relevant events

CRUD

App server

App server

{ }

{ }
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 }
```
Oplog Tailing
Tapping into the Oplog

- *Every* Meteor server receives all DB writes through oplogs
  → *does not scale*
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](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
Pour
LiveQuery Architecture

- **LiveQuery Server**: no data, real-time query matching
- **Every LiveQuery Server receives** *all* database writes

→ **does not scale**

**Bottleneck!**

Illustration taken from:
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
- 2017, October: Firestore is released
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!**


“Scale to around 100,000 concurrent connections and 1,000 writes/second in a single database. Scaling beyond that requires sharding your data across multiple databases.”

Bottleneck!
Firebase
Firestore: New Model

Firebase
Firestore: New Model

finer access granulates

tree-like structure

Firebase

Firestore: Summary

• More specific data selection
• Logical AND for some filter combinations

... But:
• Still **Limited Expressiveness**
  • No logical OR
  • No logical AND for many filter combinations
  • No content-based search (regex, full-text search)
• Still **Limited Write Throughput:**
  • 500 writes/s per collection
  • 1 writes/s per document

Firebase, Firestore: Quotas and Limits (2018)
https://firebase.google.com/docs/firestore/quotas (2018-03-10)
Honorable Mentions
Other Systems With Real-Time Features

GRAPHCOOL
rapid.io
CouchDB
OrientDB
elasticsearch
mongoDB
realm
REAL-TIME DBS

Summary & Discussion
## Wrap-Up

<table>
<thead>
<tr>
<th></th>
<th>METEOR</th>
<th>RethinkDB</th>
<th>Parse</th>
<th>Firebase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Poll-and-Diff</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Change Log Tailing</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Unknown</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

| **Write Scalability**  | ✓      | ✓         | ✓     | ✓        |
| **Read Scalability**   | ×      | ✓         | ✓     | ✓        |

| **Composite Filters (AND/OR)** | ✓      | ✓         | ✓     | ✓        |
| **Sorted Queries**           | ✓      | ✓         | ✓     | ×        |
| **Limit**                    | ✓      | ✓         | ✓     | ×        |
| **Offset**                   | ✓      | ✓         | ×     | ×        |
| **Self-Maintaining Queries** | ✓      | ✓         | ×     | ×        |
| **Event Stream Queries**     | ✓      | ✓         | ✓     | ✓        |

- **(100k connections)**
- **(AND In Firestore)**
- **(single attribute)**
- **(value-based)**
- **(100k connections)**

Summary

Real-Time Databases: Major challenges

- **Scalability:**
  - Handle increasing throughput
  - Handle additional queries

- **Expressiveness:**
  - Content-based search? Composite filters?
  - Ordering? Limit? Offset?

- **Legacy Support:**
  - Real-time queries for *existing databases*?
  - *Decouple* OLTP from real-time workloads?
Outline

- **Introduction**
  - Where From? Where To?

- **Stream Processing**
  - Big Data + Low Latency

- **Real-Time Databases**
  - Push-Based Collections

- **Future Directions**
  - Current Research & Outlook

- **Caching Dynamic Data**:
  - Why is the Web Slow?
  - Caching to the Rescue!
  - Query Caching

- **Real-Time Queries**:
  - Scalability
  - Expressiveness
  - Legacy Compatibility
  - Use Cases

- **Open Challenges**:
  - TTLs & Transactions
  - Polyglot Persistence

- **Summary**
OUTLOOK

Our Research at the University of Hamburg
Problem: Slow Websites
Two Bottlenecks: Latency and Processing
Solution: Global Caching
Fresh Data From Distributed Web Caches
New Caching Algorithms
Solve Consistency Problem
Consistent Web Caching

The Cache Sketch

Flat(Counting Bloomfilter)
RESEARCH

How to **Invalidate DB Query** Results?
How to detect changes to query results: „Give me the most popular products that are in stock.”
InvaliDB
Invalidating DB Queries

Real-Time Queries (Websockets)

Create
Update
Delete

Server

Pub-Sub

Fresh Caches

Pub-Sub
Baqend Real-Time Queries
Realtime Decoupled

Keeps data up-to-date!
InvaliDB: A Scalable Real-Time Database Design
Two-Dimensional Workload Partitioning

Read & Write Scalability:
- many concurrent users
- high write throughput
- no single-server bottleneck

Pluggable Query Engine:
- legacy-compatibility
- multi-tenancy across databases

write scalability

write partition 1
write partition 2
write partition 3

query scalability

query partition 1
query partition 2
query partition 3

InvaliDB
Baqend Real-Time Queries
Staged Real-Time Query Processing

Change notifications go through different 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
Bagend Real-Time Queries
Low Latency + Linear Scalability

Linear Scalability

Stable Latency Distribution

Quaestor: Query Web Caching for Database-as-a-Service Providers
VLDB '17
Programming Real-Time Queries
JavaScript API

```javascript
var query = DB.Tweet.find()
    .matches('text', /my filter/)
    .descending('createdAt')
    .offset(20)
    .limit(10);

query.resultList(result => ...);

query.resultStream(result => ...);
```
Live Demo!
- Wednesday, 15:30
- Zuse 210
Baqend
Try It Out!

Platform

– Platform for building (Progressive) **Web Apps**
– **15x** Performance Edge
– Faster **Development**

Speed Kit

– Turns Existing Sites into **PWAs**
– **50-300% Faster** Loads
– **Offline** Mode
Speed Kit
Baqend Caching for Legacy Websites

Website with Snippet

Requests

Speed Kit Service Worker

Fast Requests

Baqend Service

Push

Pull

3rd Party Services

Existing Backend
Speed Kit
Measure Your Page Speed!

https://www.baur.de/

You are using Speed Kit 1.12.1

Without Speed Kit
1283 ms

1.9x Faster

Your Website (Speed Kit 1.12.1)
662 ms

Average

Without Speed Kit
1.3s

Your Website
0.7s

Fast
For a large e-commerce company like Baur, supreme performance and a snappy user experience are vital. **Speed Kit** helps Baur.de stay ahead of the competition by accelerating page loads through **cutting-edge technology**. Finally, there is a German player in the web performance market that does not only pioneer a **superior approach**, but also shines through competent onboarding and immediate support.

Revenue: 1 000 000 000 € for 2018
Traffic: 70 000 000 PIs per month

A member of the **otto group**
FUTURE DIRECTIONS

Open Challenges
**TTL Estimation**
Quantifying Cacheability of Dynamic Content

- **Setting:** server assigns a caching time-to-live (TTL) to each record and query result

- **Problem:**
  - TTLs too short: Bad cache-hit rate
  - TTLs too large: Bloom filter’s false positive rate degrades

- **Approach:** Collect access metrics and estimate

  - **Objects:** calculate the expected value of the time to next write (assuming a poisson process)

  - **Queries:**
    - **Initial estimate:** estimated time until first object in result is updated
    - **Refinement:** upon invalidation TTL is adapted towards observed TTL using an EWMA
TTL Estimation
Learning Representations

**Setting:** query results can either be represented as references (id-list) or full results (object-lists)

<table>
<thead>
<tr>
<th>Id-Lists</th>
<th>Object-Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>{id₁, id₂, id₃}</td>
<td>{ {id: 1, val: 'a'}, {id: 2, val: 'b'}, {id: 3, val: 'c'} }</td>
</tr>
</tbody>
</table>

Less Invalidations                  Less Round-Trips

**Current Approach:** Cost-based decision model that weighs expected round-trips vs expected invalidations

**Desired:** Adaptive agent that actively explores decisions
TTL Estimation
Open Challenge: Learning Workloads

**Backwards-oriented** *(current approach):*
- Measure & use moving average or newest measurement
- Cannot react to spikes/fluctuation nor detect patterns

**Predictive online-learning**:  
- Extrapolate using regression (e.g. linear or polynomial) or time-series models (Exponential Smoothing, AR, ARIMA, Gaussian Processes, ...)
- Resource intensive, very difficult to select & evaluate model

**Reactive**:  
- Use Reinforcement learning to automatically explore decisions
- Rewards usually noisy, delayed or hidden (e.g. staleness)
Polyglot Persistence Mediator
Schemas can be annotated with requirements/SLAs

- Write Throughput > 10,000 RPS
- Read Availability > 99.9999%
- Scans = true
- Full-Text-Search = true
- Monotonic Read = true
Polyglot Persistence Mediator
Routing to the „optimal“ database system

Application

Data and Operations

Polyglot Persistence Mediator

Routing Model

Recursive Ranking Algorithm for schemaElemnt → DB mapping

Database Metrics

Latency < 30ms
Polyglot Persistence
Open Challenges

- **Meta-DBaaS**: Mediate over DBaaS-systems unify SLAs
- **Live Migration**: adapt to changing requirements
- **Database Selection**: Actively minimize SLA violations
- **Utility Functions/SLAs**: Capture trade-offs comprehensively
- **Workload Management**: Adaptive Runtime Scheduling
Distributed Transactions

Transaction Abort Rates: How to mitigate high abort rates caused by long running transactions?

Automatic Transaction Protocol Selection: Can the optimal protocol (2PL, BOCC+, RAMP, ...) be learned and chosen at runtime?

Transactional Visibility For Real-Time Queries: How to include transactions without introducing bottlenecks?
CLOSING TIME

Summary
Summary
Real-Time Data Management

pull-based Database Management
- static collections

Real-Time Databases
- evolving collections

Data Stream Management
- structured streams

push-based Stream Processing
- unstructured streams

{wingerath, gessert, ritter}@informatik.uni-hamburg.de
Our Related Publications

Scientific Papers:

- Quaestor: Query Web Caching for Database-as-a-Service Providers
  VLDB ‘17
- NoSQL Database Systems: A Survey and Decision Guidance
  SummerSOC ‘16
- Real-time stream processing for Big Data
  it - Information Technology 58 (2016)
- The Case For Change Notifications in Pull-Based Databases
  BTW ‘17

Blog Posts:

- Real-Time Databases Explained: Why Meteor, RethinkDB, Parse and Firebase Don’t Scale

Learn more at blog.baqend.com!
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.

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.

Read them on blog.baqend.com!
For videos & book, visit slides.baqend.com!
Thank you

{wingerath, gessert, ritter}@informatik.uni-hamburg.de

Blog: blog.baqend.com
Slides: slides.baqend.com

@baqendcom

Remember: Live Demo on Wednesday, 15:30, Zuse 210!