Real-Time Data Management
For Big Data

Wolfram Wingerath, Felix Gessert, Norbert Ritter
{wingerath, gessert, ritter}@informatik.uni-hamburg.de
March 29, EDBT 2018, Vienna
Who We Are

Norbert Ritter
Professor

Felix Gessert
CEO

Wolfram Wingerath
Developer

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

Practice:
Backend-as-a-Service
Web Caching
Real-Time Database
...
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:**
  • (No)SQL Decision Tree
  • (No)SQL Toolbox
  • Active Database Features
• **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
- HiPAC
- System R
- PostgreSQL
- Active Databases

CEP & Streams
- Starburst
- Telegraph
- MapReduce
- STREAM
- Rapide
- Bigtable
- Open claims

Stream Processing
- Spark
- Samza
- Meteor
- Baqend
- GFS
- Dynamo
- Storm
- Big Data & NoSQL

Real-Time Databases
- Aurora & Borealis
- RethinkDB
- Firebase
- Stream Processing

- Baqend
CONCEPTS & REQUIREMENTS

The NoSQL Toolbox
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
(No)SQL Decision Tree

Access

Fast Lookups

Volume

RAM

Unbounded

CAP

AP

CP

Redis

Memcache

Cassandra

Riak

Voldemort

Aerospike

HBase

MongoDB

CouchBase

DynamoDB

RDBMS

Neo4j

RavenDB

MarkLogic

Hadoop

Spark

Parallel DWH

Cassandra

HBase

Riak

MongoDB

SimpleDB

RethinkDB

HBase, Accumulo

ElasticSearch, Solr

Social Network

Big Data

Complex Queries

Volume

HDD-Size

Consistency

Availability

Ad-hoc

Analytics

Query Pattern

Example Applications

Cache

Shopping-basket

Order History

OLTP

Website

Social Network

Big Data

Oltp

Example Applications

Access

Fast Lookups

RAM

Redis

Memcache

Unbounded

AP CP

Complex Queries

Volume

HDD-Size

Consistency

Availability

Ad-hoc

Analytics

Query Pattern

Example Applications

Cache

Shopping-basket

Order History

OLTP

Website

Social Network

Big Data

Oltp
(No)SQL Decision Tree

Purpose:

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

Database Vendors/Researchers: clear communication and design of system trade-offs
Functional Techniques

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

Techniques

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

Non-Functional

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

- Horizontal distribution of data over nodes

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

Approaches

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

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

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

Sharding Approaches

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

Range-based Sharding
- Assigns ranges defined over fields (shard keys)
- **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

---

ACID Transactions
Conditional or Atomic Writes
Commit/Consensus Protocol
Synchronous
Asynchronous
Primary Copy
Update Anywhere
Read Scalability
Consistency
Write Latency
Read Latency
Read Availability
Write Availability
Replication

Read Scalability + Failure Tolerance

- Stores $N$ copies of each data item

- **Consistency model**: synchronous vs asynchronous

- **Coordination**: Multi-Master, Master-Slave

Replication: When

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

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

Replication: When

Asynchronous (lazy)

- Writes are acknowledged immediately
- Performed through log shipping
- **Pro:** Fast writes, no coordination needed
- **Contra:** Replica data potentially 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
Functional Techniques

Non-Functional

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

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

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

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

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

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

Speed, Cost
Size

Low Performance
High Performance

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

Data
Log
Persistent Storage

RAM

Data

I/O

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

Data

I/O

Low Performance
High Performance
RR: Random Reads
SR: Sequential Reads
SW: Sequential Writes
RW: Random Writes
NoSQL Storage Management
In a Nutshell

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

Speed, Cost

Size

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

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

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

- Caching
- Primary Storage
- Data Structures

In-Memory/Caching

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

Persistent Storage

Data

Log

Implements durability of write operations.

Increases write throughput.

Is good for read latency.

Promotes durability of write operations.

Improves latency.
Functional Techniques

Non-Functional

Joins

Sorting

Filter Queries

Full-text Search

Aggregation and Analytics

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

Read Latency
Query Processing Techniques

Summary

- **Local Secondary Indexing**: Fast writes, scatter-gather queries
- **Global Secondary Indexing**: Slow or inconsistent writes, fast queries
- **(Distributed) Query Planning**: Scarce in NoSQL systems but increasing (e.g. left-outer equi-joins in MongoDB and θ-joins in RethinkDB)
- **Analytics Frameworks**: fallback for missing query capabilities
- **Materialized Views**: similar to global indexing
Summary

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

- **The (No)SQL Toolbox:** systems use similar techniques that promote certain capabilities

- **Decision Tree:** maps requirements to concrete systems
TRIGGERS & MORE

Active Database Features
Databases are **Passive**

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

Are there new circles?
Databases are **Passive**

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

Change discovery through periodic polling

→ **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
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

Similar processing for:
• Triggers
• ECA rules

→ Potential bottlenecks:
• Number of queries/triggers/rules
• Write throughput
• Complexity
EVOLVING DOMAINS

Data Stream Management
Push-Based Access For Evolving Domains
Continuous Queries Over Data Streams

Find people in Room B:

```
SELECT name, x, y
FROM People
WHERE x BETWEEN 0 AND 25
AND y BETWEEN 0 AND 15
ORDER BY name ASC
```

1. Erik (5/10)
2. Wolle (21/4)

Diagram:
- Room A
- Room B
- Room C
- People positioned at (x, y) coordinates:
  - Erik at (5, 10)
  - Wolle at (21, 4)
Data Stream Management Systems

High-Level Architecture

- Stream query processor
- Working memory
- Database
- Archive (offline)
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
  → **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

raw stream

Prohibitive!
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/stoping work
- **Monitoring** - health checks, application stats
- **Scheduling** - assigning work, rebalancing
- **Fault-tolerance** - restarting workers, rescheduling failed work
Big Data Processing Frameworks
What are your options?

- Spark Streaming
- Google Dataflow
- HERON
- Spark
- STORM
- STORM Trident
- IBM InfoSphere Streams
- Amazon Elastic MapReduce
- Flink
- APEX
- samza
- hadoop
- kafka streams
- concord
Big Data Processing Frameworks
What are your options?

What to use when?
Big Data Processing Frameworks
What are your options?

- IBM InfoSphere Streams
- STORM Trident
- Google Dataflow
- Flink
- APEX
- Spark Streaming
- Spark
- Amazon Elastic MapReduce
- hadoop
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)
  \( \rightarrow \) "eventually accurate" merged views of real-time & batch
- Typical setups: Hadoop + Storm (\( \rightarrow \) 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**

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

- **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
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
Processing Models
Batch vs. Micro-Batch vs. Stream

stream  micro-batch  batch

Flink  Storm  Storm Trident  Spark Streaming  Spark
samza  Hadoop  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
  $\rightarrow$ At-least-once has **2x messaging overhead**
Cluster Architecture
How Storm Scales

Submit Topology

Nimbus

Zookeeper

Supervisor
Worker
Worker
Worker
Worker

Supervisor
Worker
Worker
Worker
Worker

Storm Slave

Storm Slave
Cluster Architecture

How Storm Scales

Submit Topology

handling coordination

Zookeeper

Nimbus

Scheduling & Monitoring

JVM for each worker (runs spouts and bolts as tasks)
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.
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)
Example: the *enriched clickstream* is available to every team within the organization

Illustration 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
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)
```

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)

DataSet (Java/Scala)

Streaming dataflow runtime

YARN
Cluster
Local

https://www.infoq.com/presentations/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>≪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>🟦</td>
<td>🟦</td>
<td>🟦</td>
<td>🟦</td>
<td>🟦</td>
</tr>
<tr>
<td></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>🟦</td>
<td>🟦</td>
<td>🟦</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>no (buffering)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ordering</strong></td>
<td>✗</td>
<td></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!

Query maintenance: periodic polling
→ Inefficient
→ Slow
Quick Comparison
DBMS vs. RT DB vs. DSMS vs. Stream Processing

Database Management
- static collections
- pull-based

Real-Time Databases
- 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
Oplog Tailing
Tapping into the Oplog

• Every Meteor server receives all DB writes through oplogs
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*
RethinkDB

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
LiveQuery Architecture

- **LiveQuery Server**: no data, real-time query matching
- **Every LiveQuery Server receives all database writes**
  → does not scale
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!**

Jacob Wenger, on the Firebase Google Group (2015)
https://groups.google.com/forum/#!topic/firebase-talk/d-XjaBVL2Ko (2017-02-27)

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

*Firebase, Choose a Database: Cloud Firestore or Realtime Database (2018)*
Firebase
Firestore: New Model

documents

collections

references

Firebase
Firestore: New Model

finer access granulates

tree-like structure

Illustration taken from: Todd Kerpelman, Cloud Firestore for Realtime Database Developers (2017)
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 [BETA]
- CouchDB [relax]
- OrientDB®
- elasticsearch
- mongoDB
- realm
REAL-TIME DBS
Summary & Discussion
### Wrap-Up

#### Direct Comparison

<table>
<thead>
<tr>
<th></th>
<th>Meteor</th>
<th>RethinkDB</th>
<th>Parse</th>
<th>Firebase</th>
<th>Baqend</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scales with write TP</strong></td>
<td>![√]</td>
<td>![×]</td>
<td>![×]</td>
<td>![×]</td>
<td>![✓]</td>
</tr>
<tr>
<td><strong>Scales with no. of queries</strong></td>
<td>![×]</td>
<td>![✓]</td>
<td>![✓]</td>
<td>![✓]</td>
<td>![✓]</td>
</tr>
<tr>
<td><strong>Composite queries (AND/OR)</strong></td>
<td>![✓]</td>
<td>![✓]</td>
<td>![✓]</td>
<td>![✓]</td>
<td>![✓]</td>
</tr>
<tr>
<td><strong>Sorted queries</strong></td>
<td>![✓]</td>
<td>![✓]</td>
<td>![✓]</td>
<td>![×]</td>
<td>![✓]</td>
</tr>
<tr>
<td><strong>Limit</strong></td>
<td>![✓]</td>
<td>![✓]</td>
<td>![✓]</td>
<td>![×]</td>
<td>![✓]</td>
</tr>
<tr>
<td><strong>Offset</strong></td>
<td>![✓]</td>
<td>![✓]</td>
<td>![×]</td>
<td>![×]</td>
<td>![✓]</td>
</tr>
</tbody>
</table>

- **Meteor** supports poll-and-diff and oplog tailing.
- **RethinkDB** does not support poll-and-diff.
- **Parse** supports composite queries and sorted queries.
- **Firebase** supports limit and offset with value-based operations.
- **Baqend** supports all features listed.

*Note: The question mark indicates partial support or limitations.*
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

Low Latency

Less Processing
New Caching Algorithms
Solve Consistency Problem
Consistent Web Caching
The Cache Sketch
Consistent Web Caching
The Cache Sketch
Consistent Web Caching
The Cache Sketch
Consistent Web Caching

The Cache Sketch

p_{urge}(obj)

hashA(oid)  hashB(oid)

0 3 1 4 1
Consistent Web Caching
The Cache Sketch
Consistent Web Caching
The Cache Sketch
Consistent Web Caching
The Cache Sketch

hashA(oid) hashB(oid)

Browser Cache CDN
Consistent Web Caching
The Cache Sketch
Consistent Web Caching

The Cache Sketch
Consistent Web Caching
The Cache Sketch

With 20,000 distinct updates and 5% error rate: 11 KByte

hashA(oid) hashB(oid)

0 1 1 1 1

0 2 1 4 0
RESEARCH

How to **Invalidate DB Query Results**?
**InvaliDB**

Invalidating DB Queries

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
Baqend Real-Time Queries
Realtime Decoupled
Baqend Real-Time Queries
Realtime Decoupled

Keeps data up-to-date!
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
Two-dimensional partitioning:
- by Query
- by Object
→ scales with queries and writes

Implementation:
- Apache Storm
- Topology in Java
- MongoDB query language
- Pluggable query engine
→ legacy-compatible
Baqend 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 => ...);
```

**Static Query**

**Real-Time Query**
1. Conju.re (conju_re, 3840 followers) tweeted:
https://twitter.com/conju_re/status/859767327570702336
Congress Saved the Science Budget—And That's the Problem https://t.co/UdrjNidakc
https://t.co/xINjpEpKZG

2. ねぼすけゆーだい (Yuuum_key, 229 followers) tweeted:
https://twitter.com/Yuuum_key/status/859767323384623104
けいきさんと PENGUIN RESEARCHのけいたくんがリブのやり取りしてる...

3. Whitney Shackley (bschneids11, 5 followers) tweeted:
https://twitter.com/bschneids11/status/859767319534469122
holy...... waiting for it so long️ ○ https://t.co/UdXcHJb7X3

4. Lisa Schmid (LisaMSchmid, 67 followers) tweeted on #teamscs, and #scs...
https://twitter.com/LisaMSchmid/status/859767317311500290
Congrats to Matthew Kent, winner of the 26th #TeamSCSCoding Challenge. https://t.co/vX1o0WgJrZ #SCScoded

5. Brian Martin Larson (Brian_Larson, 40 followers) tweeted on #teams...
https://twitter.com/Brian_Larson/status/859767317303001089
Congrats to Matthew Kent, winner of the 26th #TeamSCSCoding Challenge. https://t.co/vX1o0WgJrZ #SCScoded
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
Accelerate Your Website!

https://www.alibaba.com/

Your Website
3514ms

With Speed Kit
1543ms

2.3x Faster

With Speed Kit
1.5s
Excellent

Your Website
3.5s
Poor

Show Details
Speed Kit
Baqend Caching for Legacy Websites

Website

3rd Party Services

Existing Backend
Speed Kit
Baqend Caching for Legacy Websites

Website with Snippet

Speed Kit Service Worker

Requests

Fast Requests

Baqend Service

3rd Party Services

Pull

Push

Existing Backend

other
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>{id_{1}, id_{2}, id_{3}}</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
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

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

Real-Time Data Management

Database Management
- static collections
  - pull-based

Real-Time Databases
- evolving collections

Data Stream Management
- structured streams

Stream Processing
- unstructured streams
  - push-based

{wingerath, gessert, ritter}@informatik.uni-hamburg.de
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 SummerSOC16 last month. Here is the written gist. We give our best to convey the condensed NoSQL knowledge we gathered building Bagend.

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.

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

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

Read them at blog.baqend.com!
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!
Thank you

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

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

@baqendcom