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

Universität Hamburg

www.baqend.com

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
- …

Universität Hamburg

BaQend
www.baqend.com

Outline

Introduction
Where From? Where To?

- 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

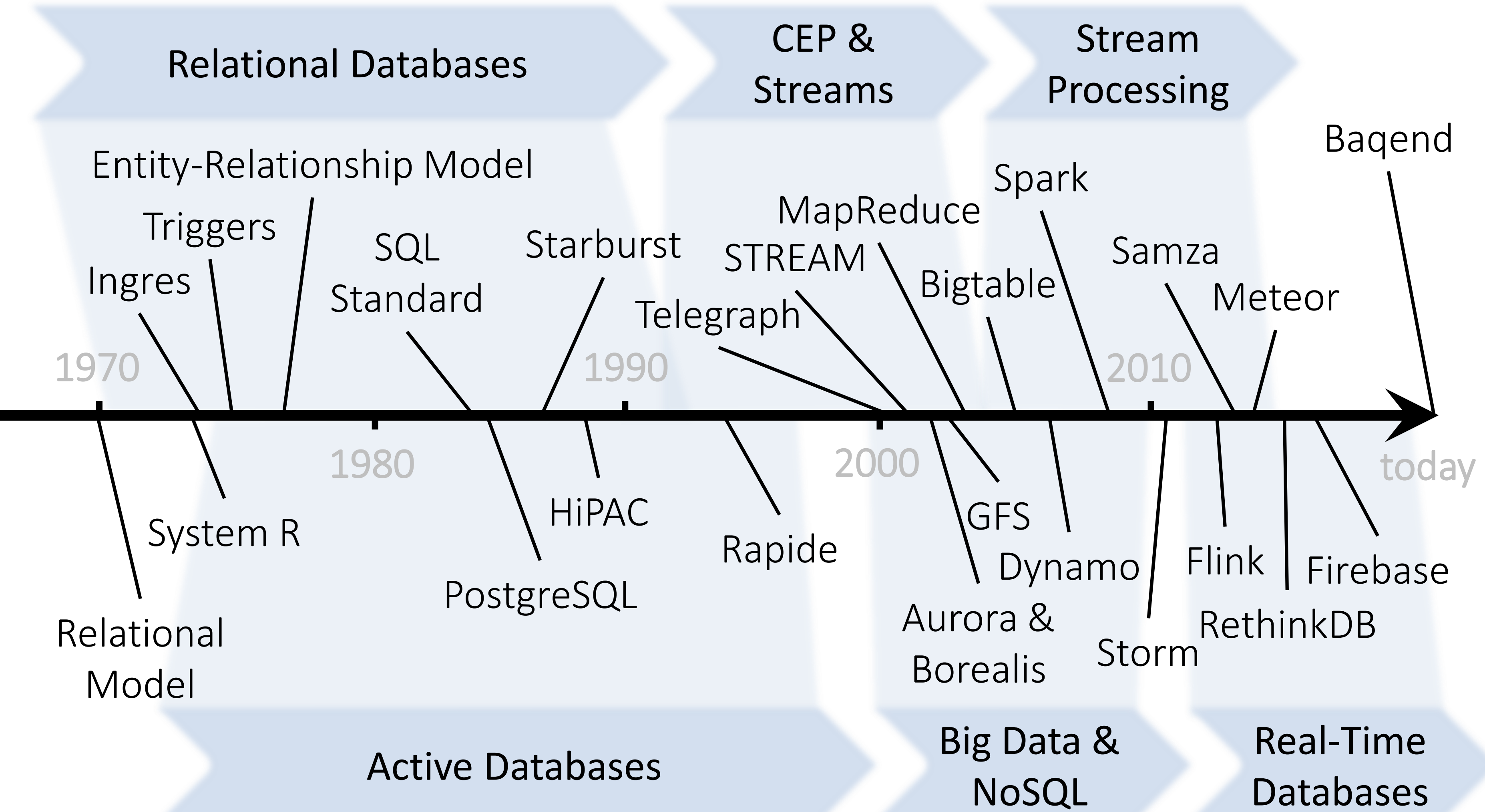
Stream Processing
Big Data + Low Latency

Real-Time Databases
Push-Based Collections

Future Directions
Current Research & Outlook

A Short History of Data Management

Hot Topics Through The Ages



CONCEPTS & REQUIREMENTS

The NoSQL Toolbox

NoSQL Database Systems: A Survey and Decision Guidance

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

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

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

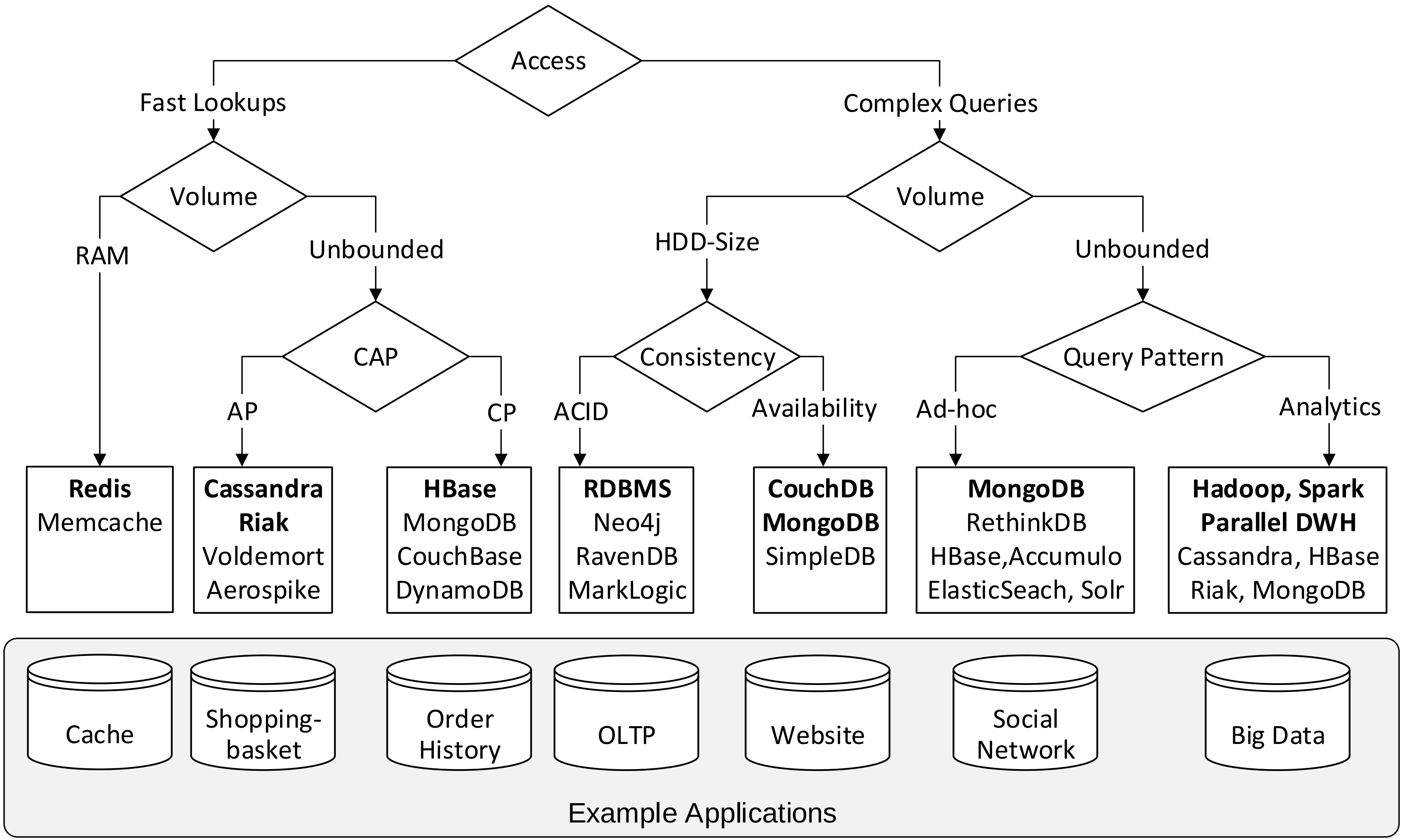
1 Introduction

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

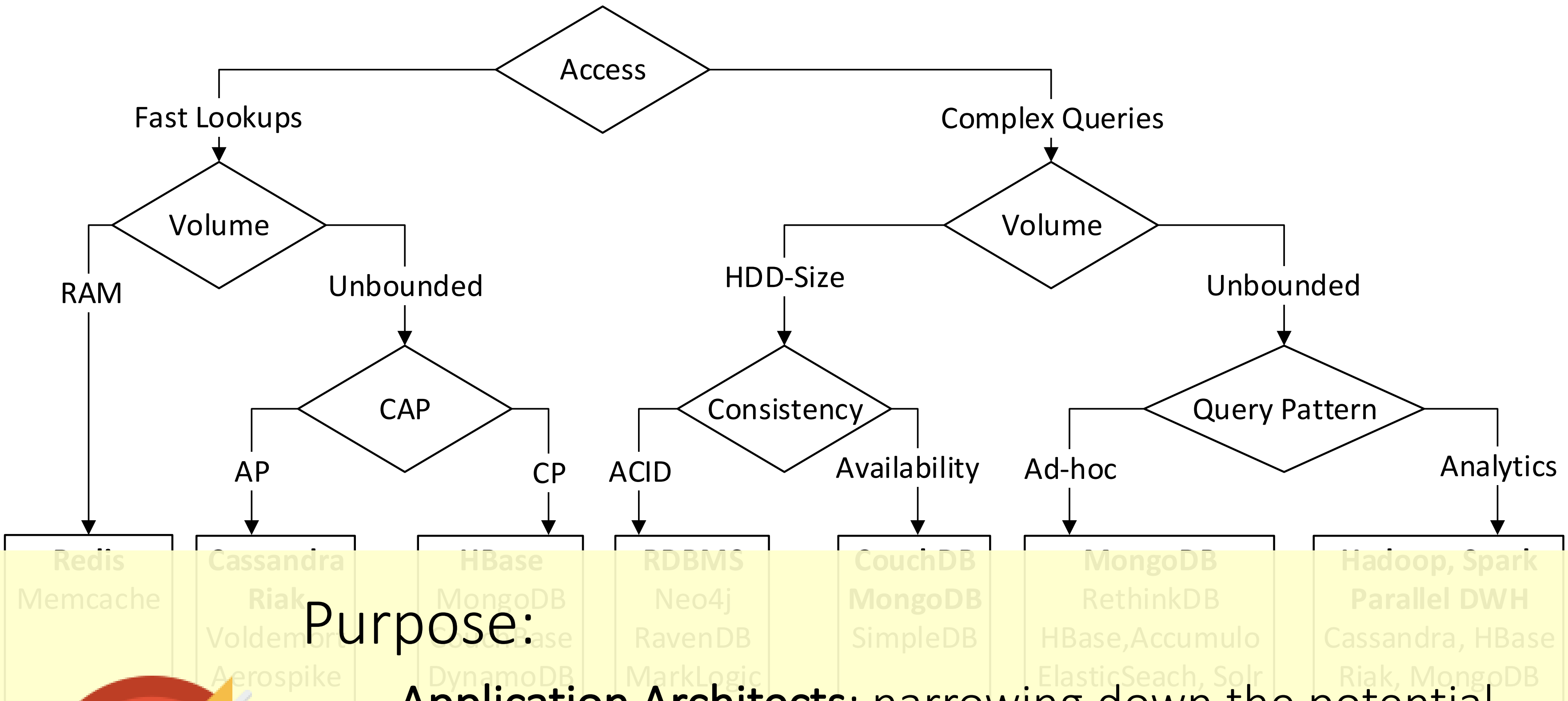
There are dozens of NoSQL database systems and it is hard to keep track of where they excel, where they fail or even where they differ, as implementation details change quickly and feature sets evolve over time. In this article, we therefore aim to provide an overview of the NoSQL landscape by discussing employed concepts rather than system specificities and explore the requirements typically posed to NoSQL database systems, the techniques used to fulfil 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

http://www.baqend.com
/files/nosql-survey.pdf

(No)SQL Decision Tree



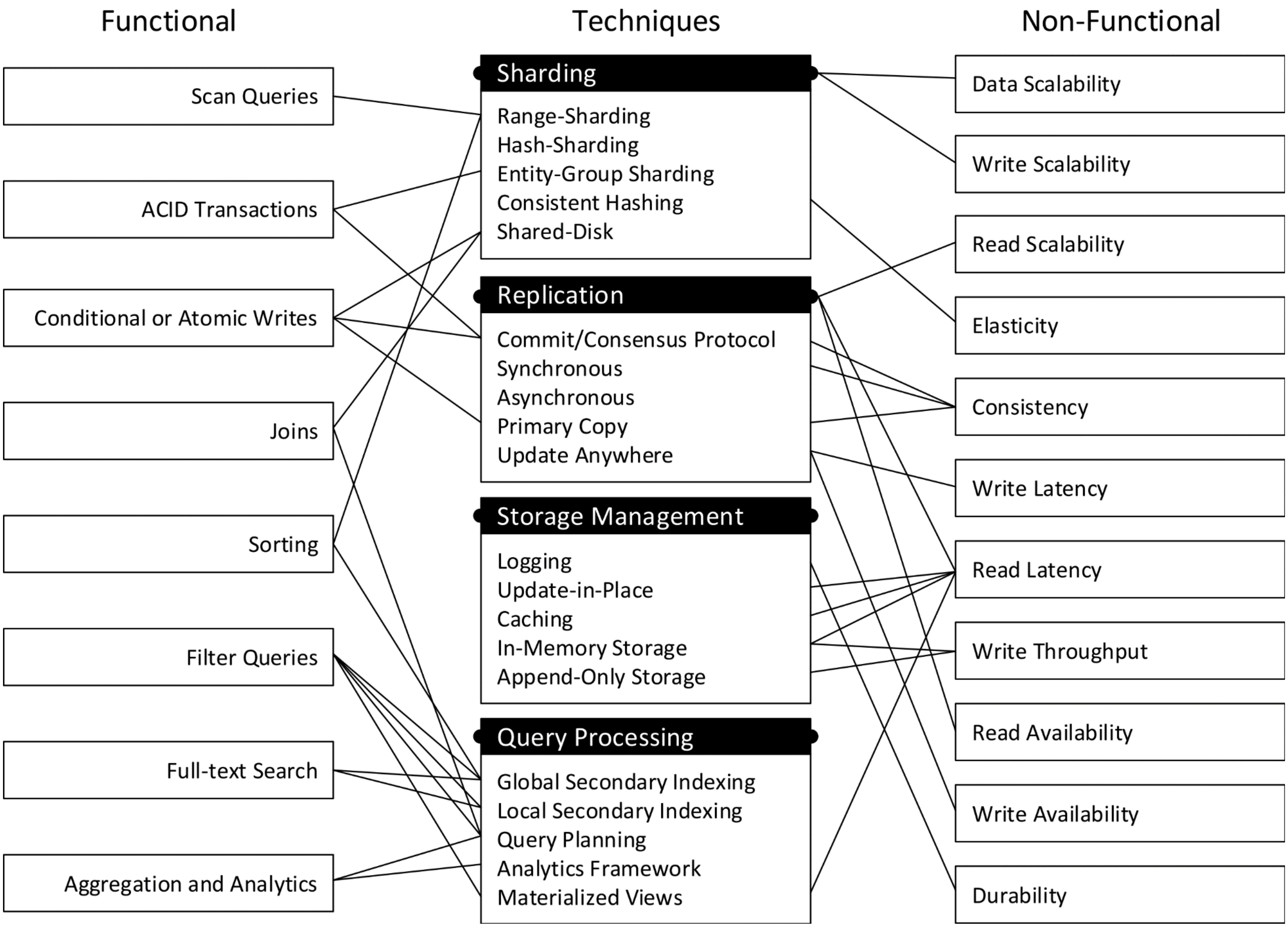
(No)SQL Decision Tree

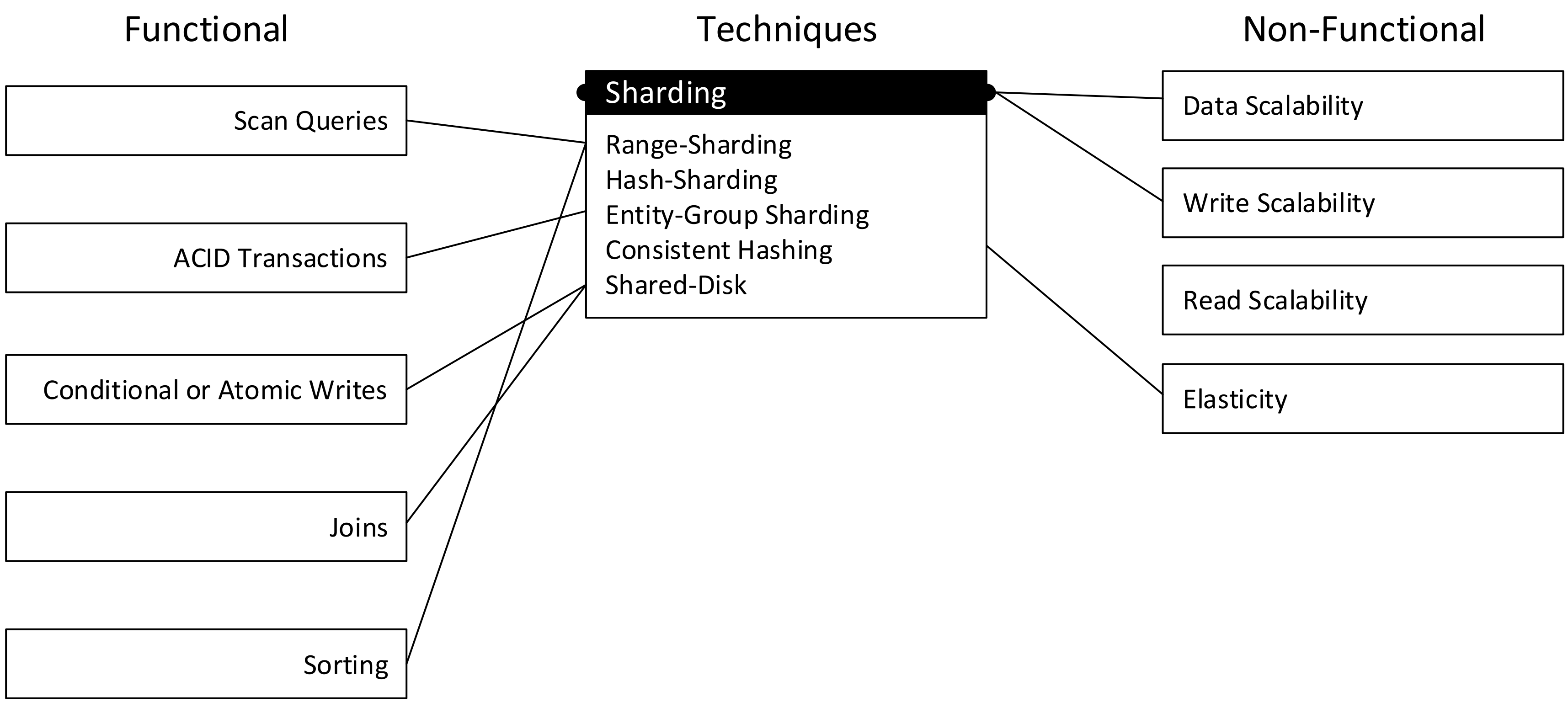


Purpose:

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

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


Functional
Techniques
Non-Functional
Scan Queries
ACID Transactions
Conditional or Atomic Writes
Joins
Sorting
Filter Queries
Full-text Search
Aggregation and Analytics
Sharding
Range-Sharding
Hash-Sharding
Entity-Group Sharding
Consistent Hashing
Shared-Disk
Replication
Commit/Consensus Protocol
Synchronous
Asynchronous
Primary Copy
Update Anywhere
Storage Management
Logging
Update-in-Place
Caching
In-Memory Storage
Append-Only Storage
Query Processing
Global Secondary Indexing
Local Secondary Indexing
Query Planning
Analytics Framework
Materialized Views
Data Scalability
Write Scalability
Read Scalability
Elasticity
Consistency
Write Latency
Read Latency
Write Throughput
Read Availability
Write Availability
Durability

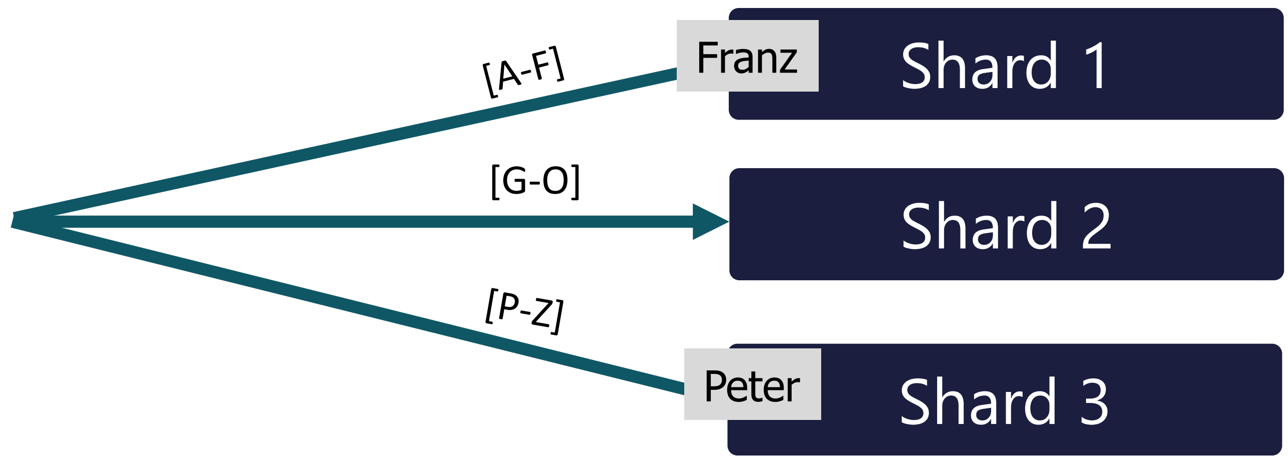


Functional
Techniques
Non-Functional
Scan Queries
ACID Transactions
Conditional or Atomic Writes
Joins
Sorting
Sharding
Range-Sharding
Hash-Sharding
Entity-Group Sharding
Consistent Hashing
Shared-Disk
Data Scalability
Write Scalability
Read Scalability
Elasticity

Sharding (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

David J DeWitt and Jim N Gray: "Parallel database systems: The future of high performance database systems," Communications of the ACM, volume 35, number 6, pages 85–98, June 1992.

Sharding

Approaches

Hash-based Sharding

- Hash of data values (e.g. key) d
- **Pro**: Even distribution
- **Contra**: No data locality

Implemented in

MongoDB, Riak, Redis, Cassandra, Azure Table, Dynamo

Range-based Sharding

- Assigns ranges defined over fie
- **Pro**: Enables *Range Scans* and *S*
- **Contra**: Repartitioning/balancin

Implemented in

BigTable, HBase, DocumentDB Hypertable, MongoDB, RethinkDB, Espresso

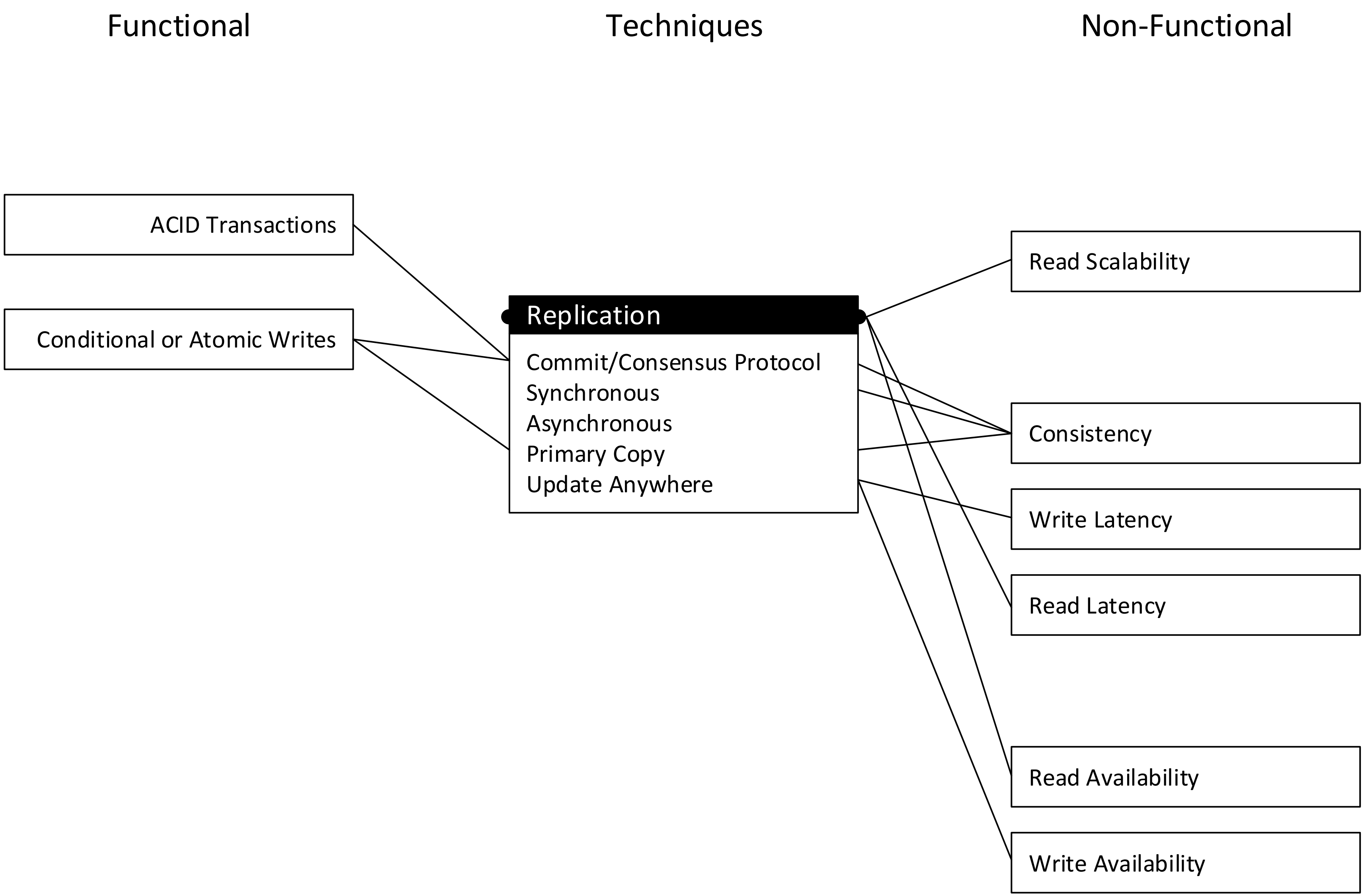
Entity-Group Sharding

- Explicit data co-location for sin
- **Pro**: Enables *ACID Transactions*
- **Contra**: Partitioning not easily c

Implemented in

G-Store, MegaStore, Relational Cloud, Cloud SQL Server

David J DeWitt and Jim N Gray: “Parallel database systems: The future of high performance database systems,” Communications of the ACM, volume 35, number 6, pages 85–98, June 1992.

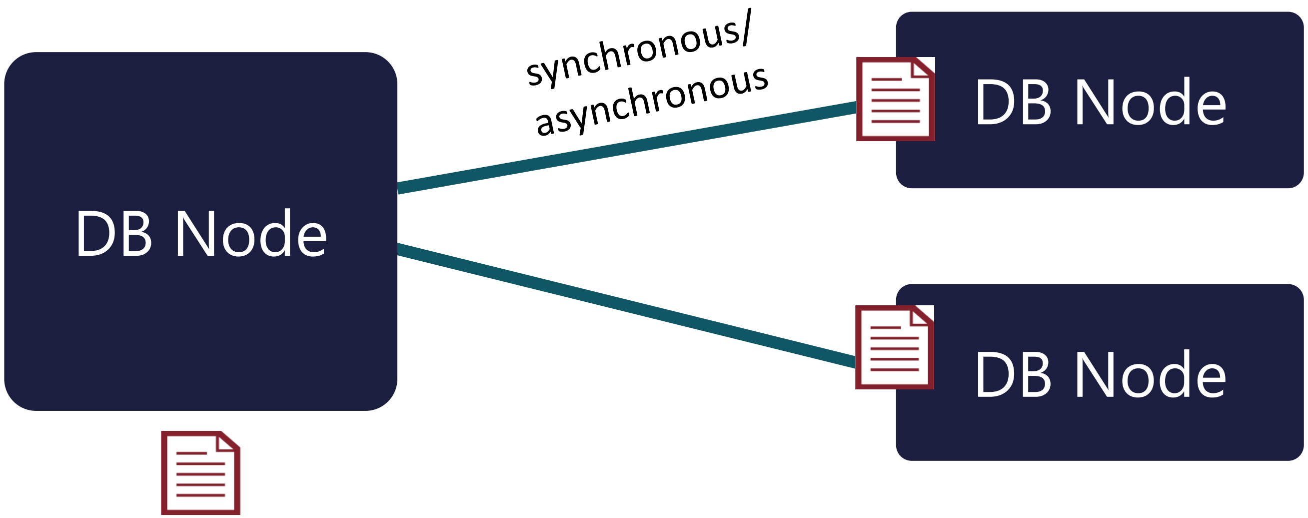


Functional
Techniques
Non-Functional
ACID Transactions
Conditional or Atomic Writes
Replication
Commit/Consensus Protocol
Synchronous
Asynchronous
Primary Copy
Update Anywhere
Read Scalability
Consistency
Write Latency
Read Latency
Read Availability
Write Availability

Replication

Read Scalability + Failure Tolerance

- Stores *N* copies of each data item



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

Özsu, M.T., Valduriez, P.: Principles of distributed database systems. Springer Science & Business Media (2011)

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), unavaialable under certain network partitions

Charron-Bost, B., Pedone, F., Schiper, A. (eds.): Replication: Theory and Practice, Lecture Notes in Computer Science, vol. 5959. Springer (2010)

Replication: When

Asynchronous (lazy)

- Writes are acknowledged imm
- Performed through *log shippi*
- **Pro**: Fast writes, no coordinati
- **Contra**: Replica data potential

Synchronous (eager)

- The node accepting writes syn tes updates/transactions before a
- **Pro**: Consistent
- **Contra**: needs a commit proto unavaialable under certain network partitions

Implemented in

Dynamo , Riak, CouchDB, Redis, Cassandra, Voldemort, MongoDB, RethinkDB

Implemented in

BigTable, HBase, Accumulo, CouchBase, MongoDB, RethinkDB

Charron-Bost, B., Pedone, F., Schiper, A. (eds.): Replication: Theory and Practice, Lecture Notes in Computer Science, vol. 5959. Springer (2010)

Replication: Where

Master-Slave (*Primary Copy*)

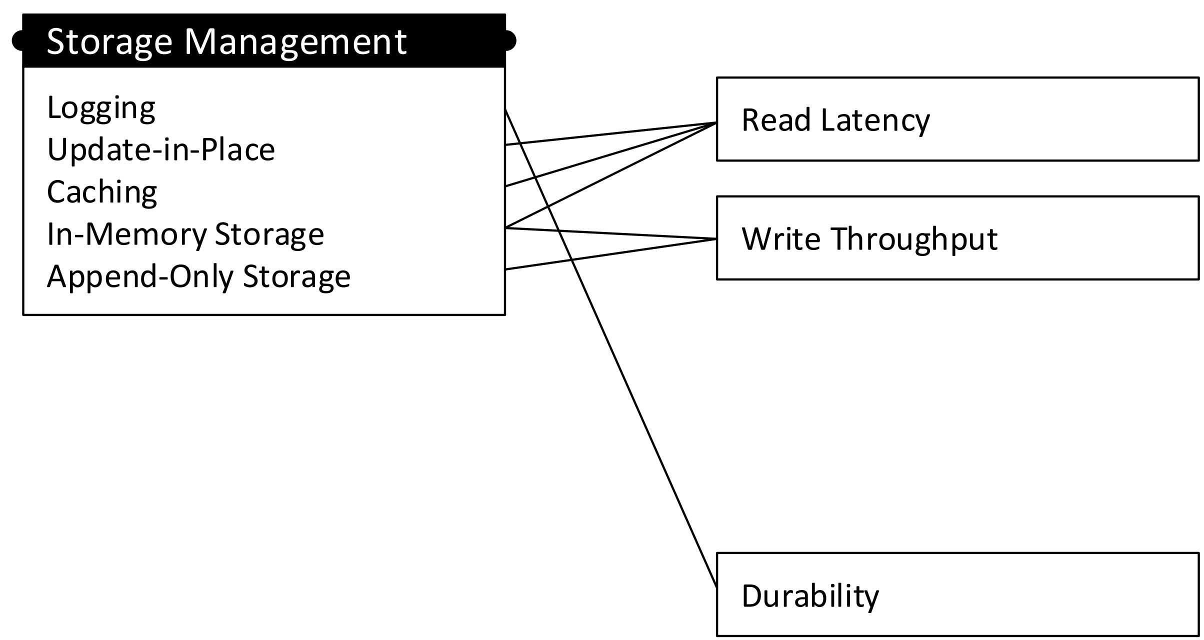
- Only a dedicated master is allowed to accept writes, slaves are read-replicas
- **Pro**: reads from the master are consistent
- **Contra**: master is a bottleneck and SPOF

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

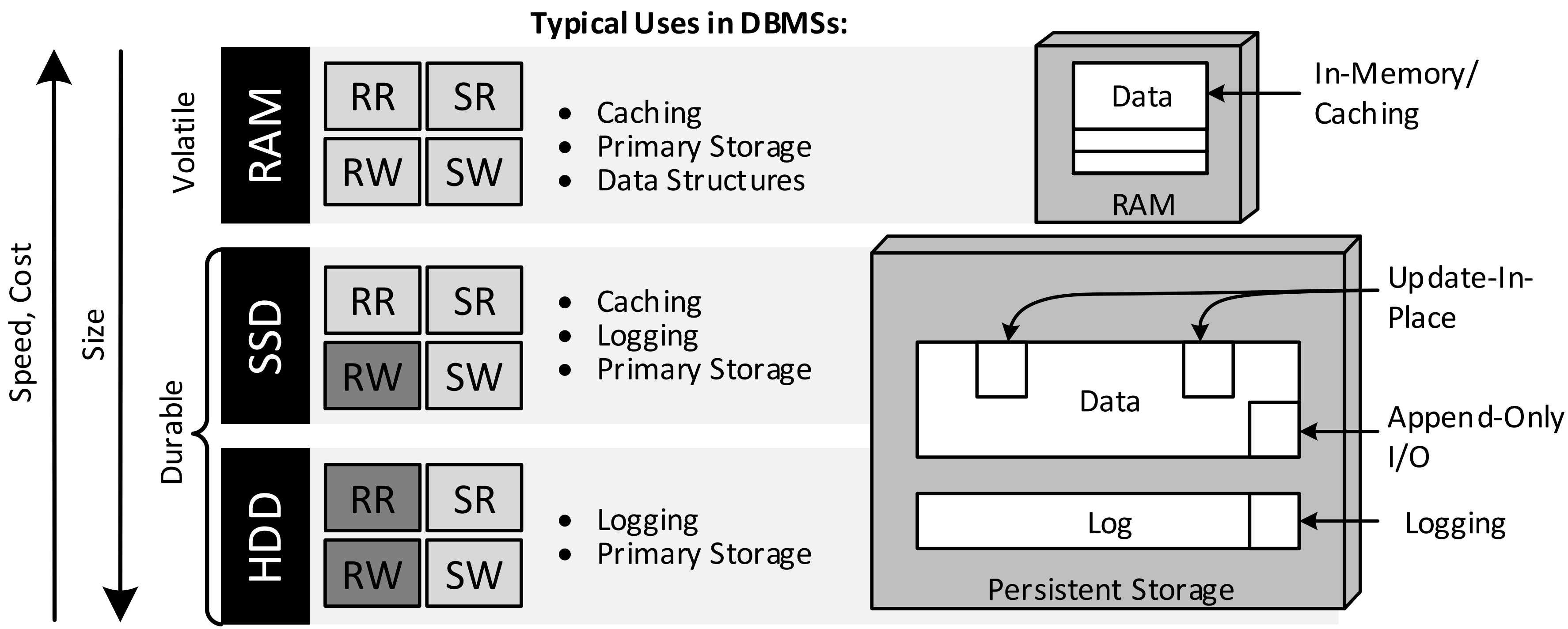
Charron-Bost, B., Pedone, F., Schiper, A. (eds.): Replication: Theory and Practice, Lecture Notes in Computer Science, vol. 5959. Springer (2010)

Functional Techniques Non-Functional



NoSQL Storage Management

In a Nutshell



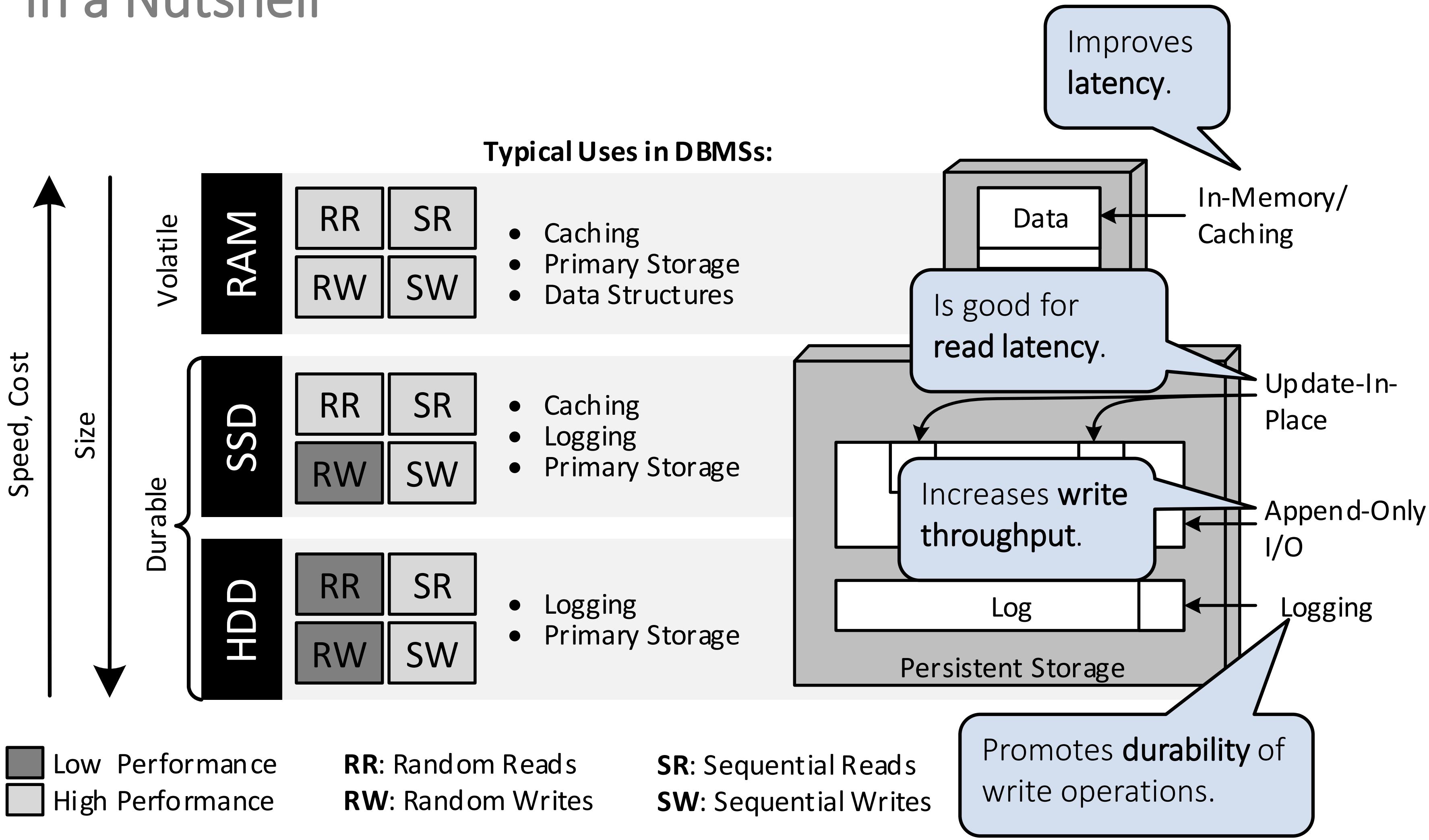
Low Performance
High Performance

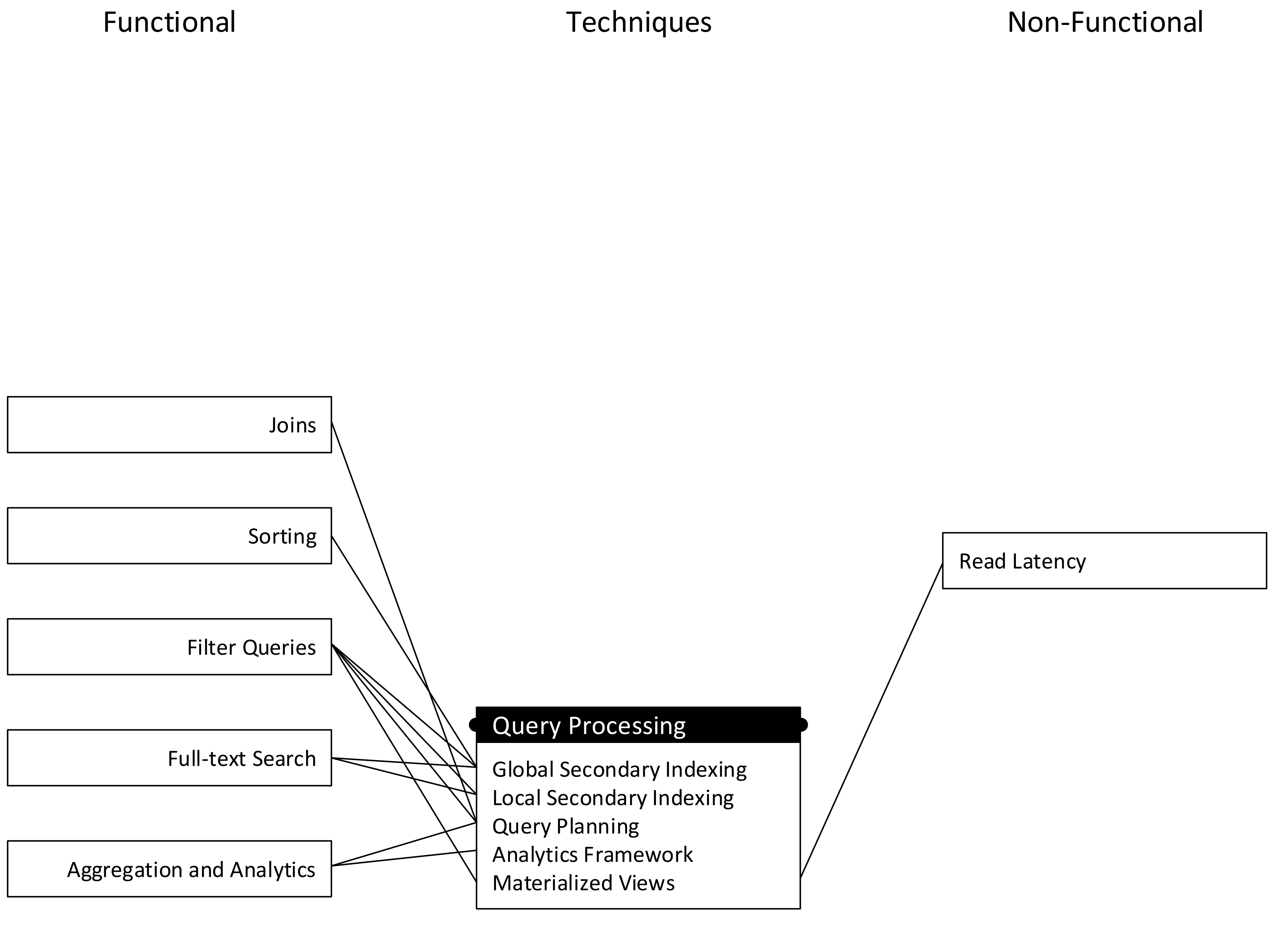
RR: Random Reads
RW: Random Writes

SR: Sequential Reads
SW: Sequential Writes

NoSQL Storage Management

In a Nutshell



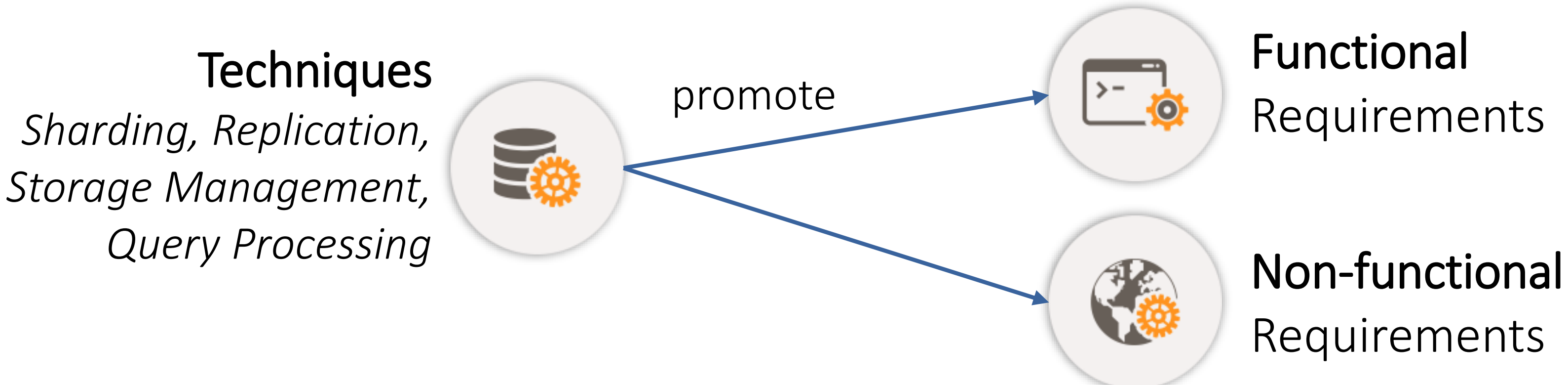

Functional
Techniques
Non-Functional
Joins
Sorting
Filter Queries
Full-text Search
Aggregation and Analytics
Query Processing
Global Secondary Indexing
Local Secondary Indexing
Query Planning
Analytics Framework
Materialized Views
Read Latency

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?

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

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

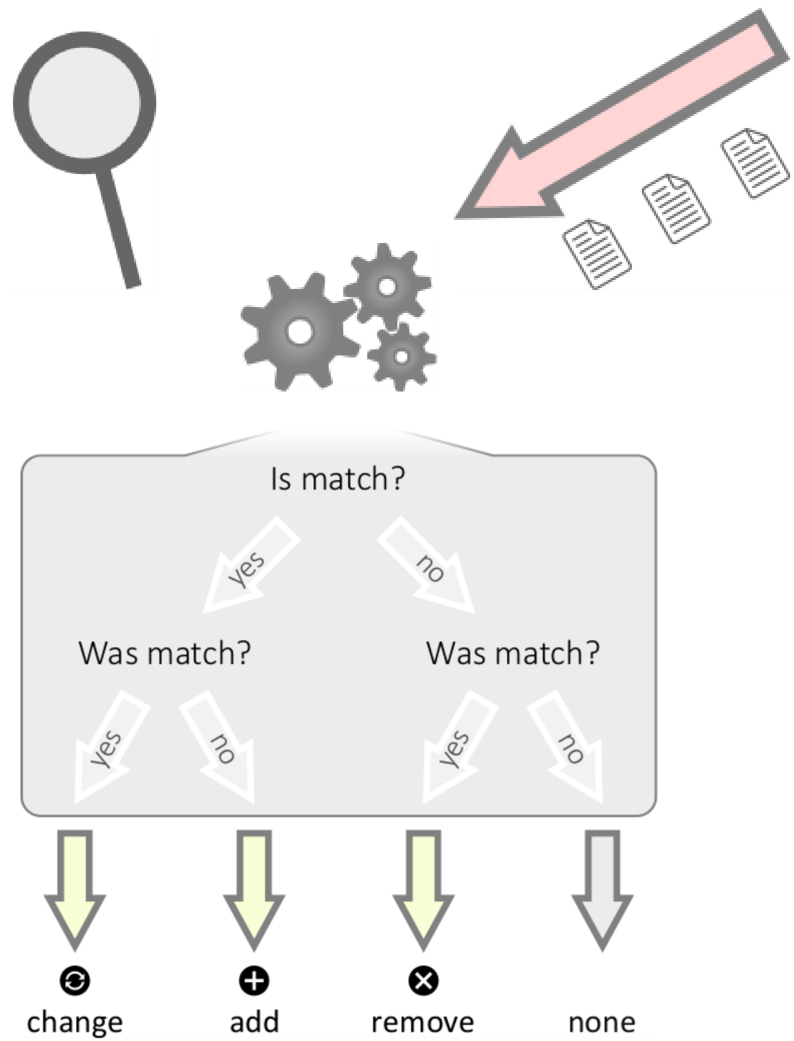
Matching Every Query Against Every Update

Similar processing for:

- Triggers
- ECA rules

→ Potential ***bottlenecks***:

- *Number of queries/triggers/rules*
- *Write throughput*
- *Complexity*



EVOLVING DOMAINS

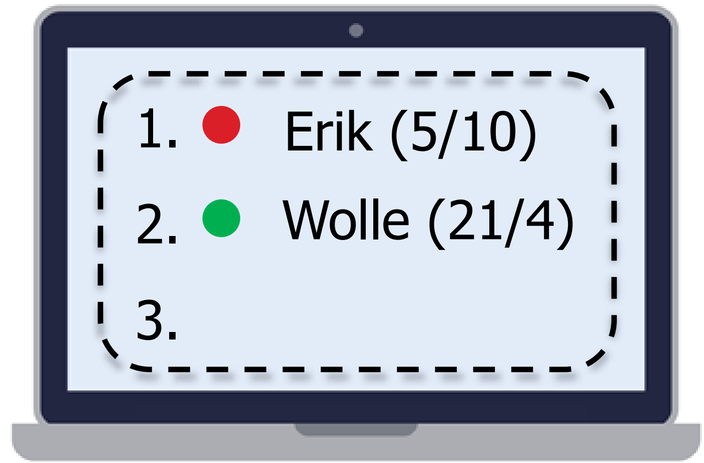
Data Stream Management

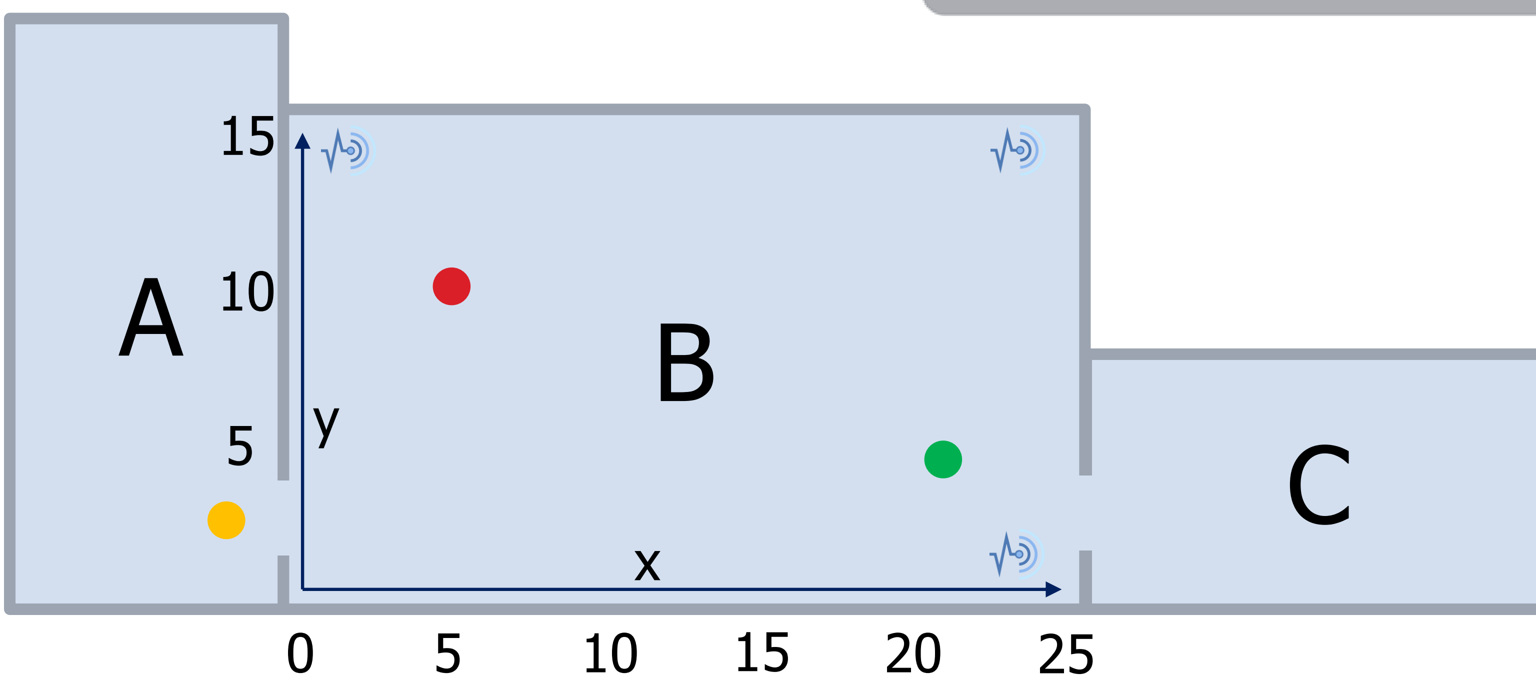
Push-Based Access For Evolving Domains

Continuous Queries Over Data Streams

Find people in Room B:

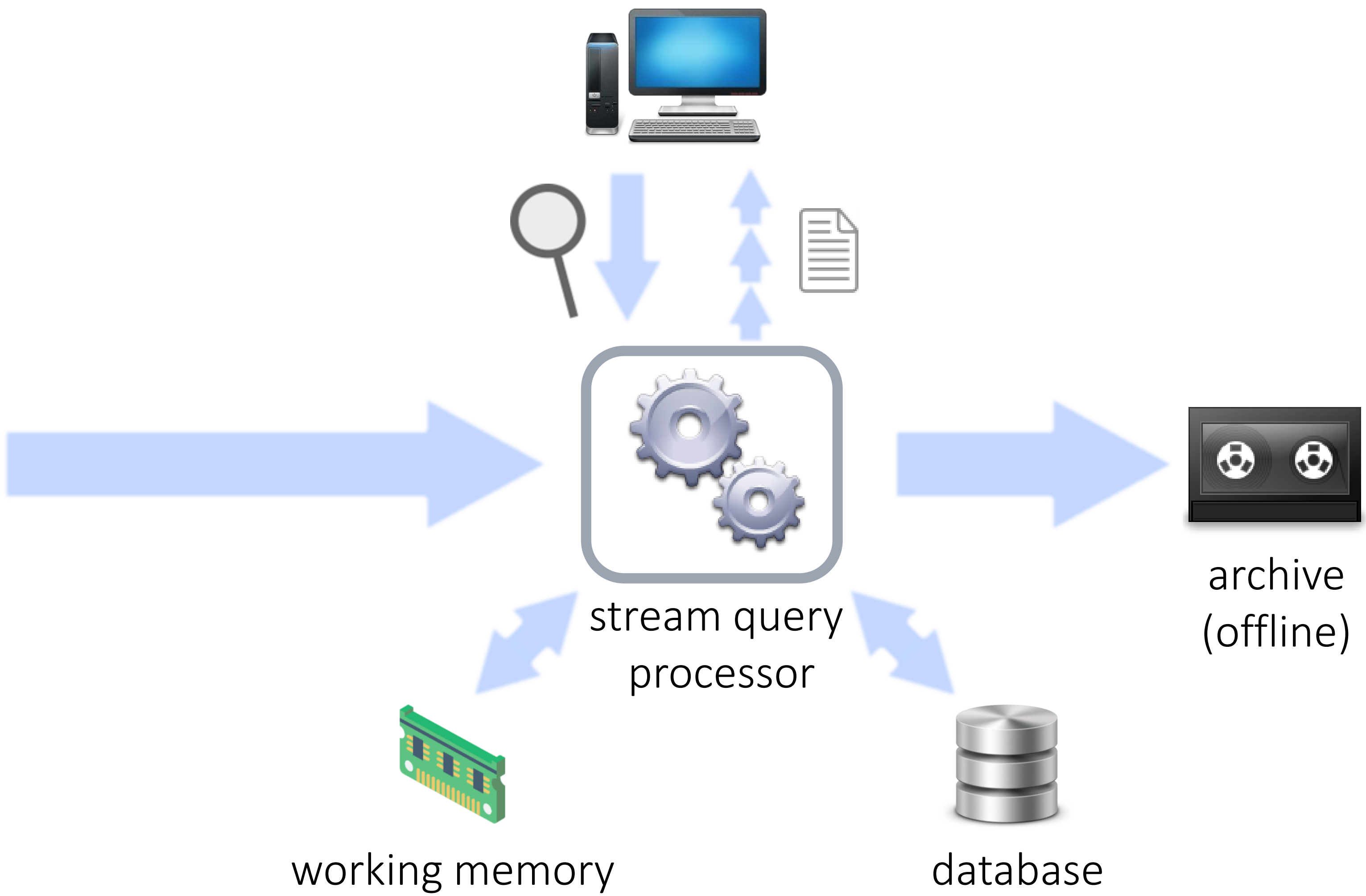
```
SELECT name, x, y
  FROM People
  WHERE x BETWEEN 0 AND 25
    AND y BETWEEN 0 AND 15
  ORDER BY name ASC
```





Data Stream Management Systems

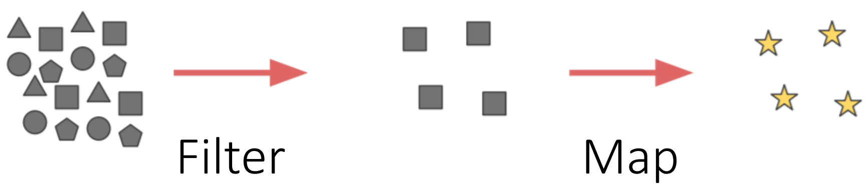
High-Level Architecture



Typical Stream Operators

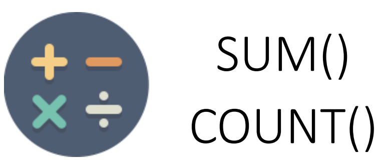
Examples

Filter & Transform

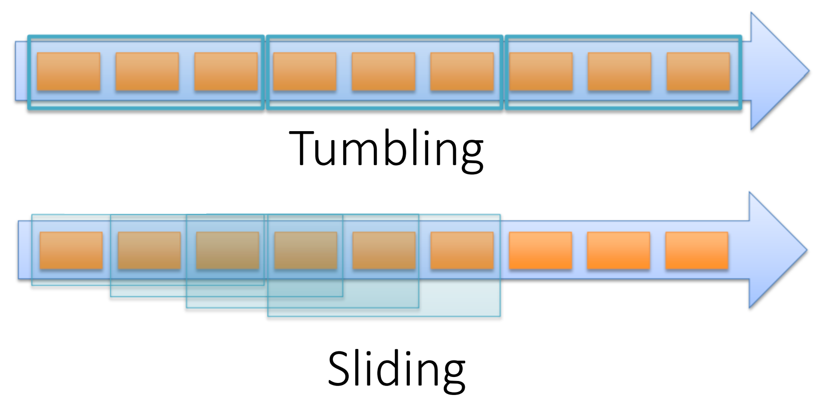


Group

Aggregates



Windows



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**

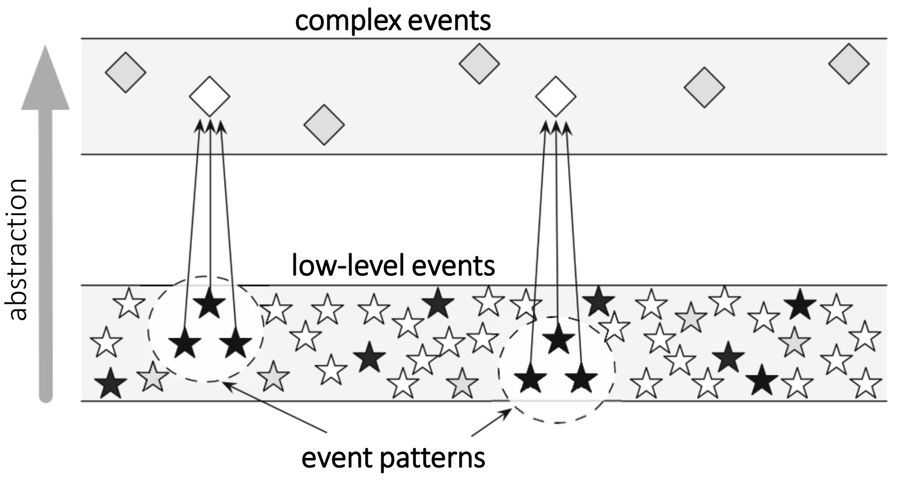


Illustration taken from: Bruns, R. & Dunkel, J, *Complex Event Processing: Komplexe Analyse von massiven Datenströmen mit CEP (2015). Springer Vieweg, 2015*

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

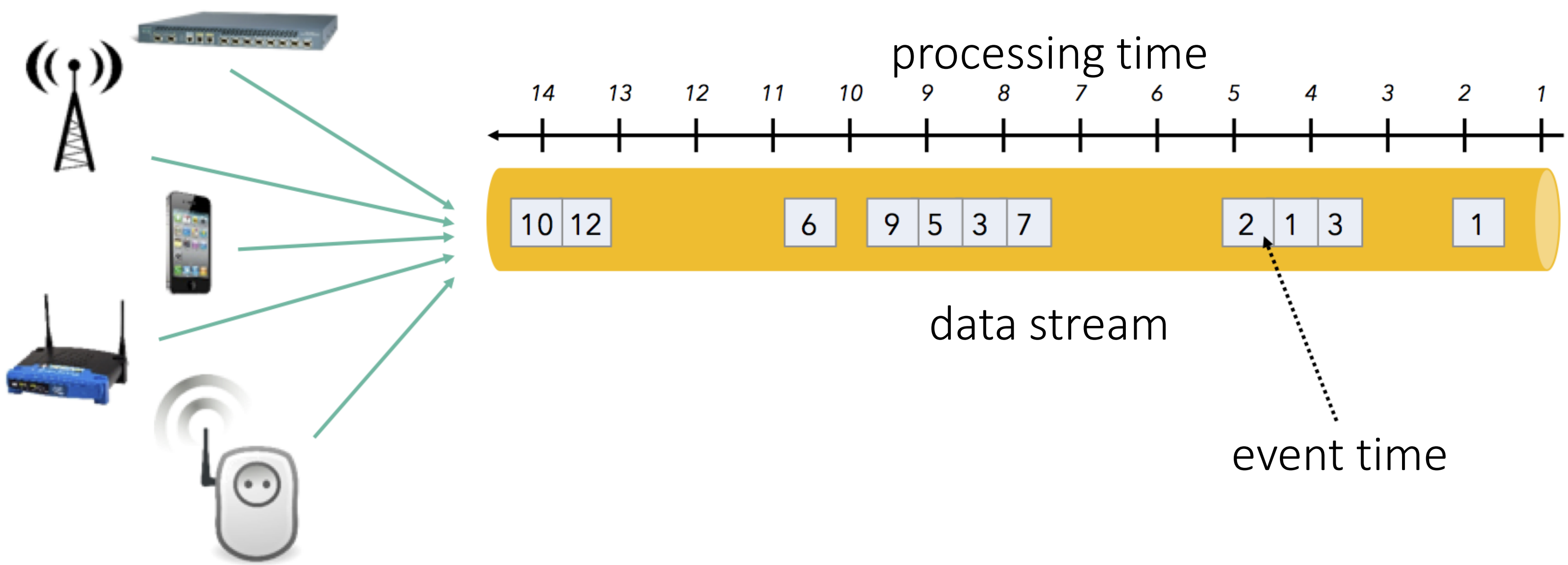
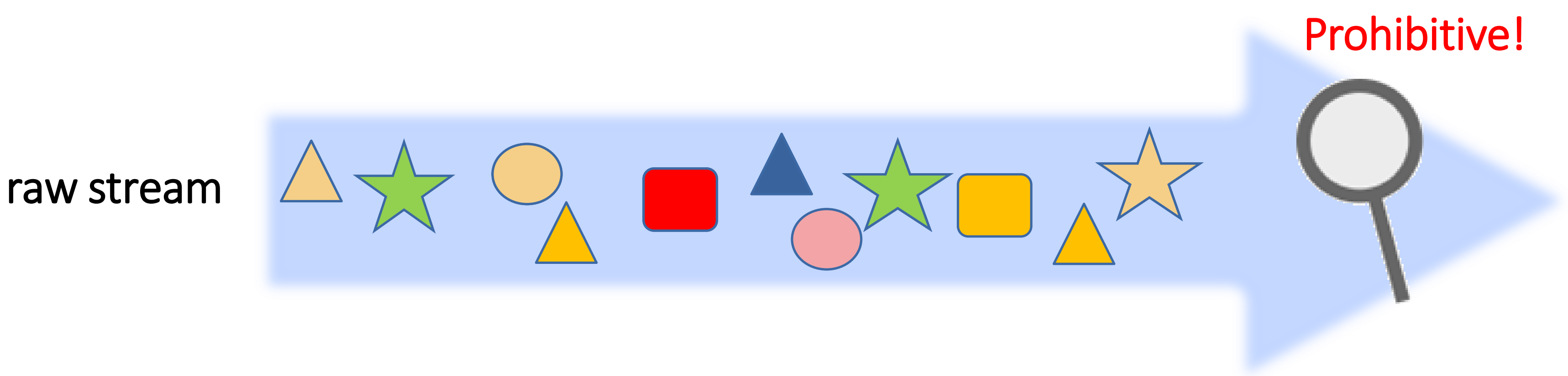


Illustration take from: Stephan Ewen, *How Apache Flink™ Enables New Streaming Applications, Part 1* (2015) https://data-artisans.com/blog/how-apache-flink-enables-new-streaming-applications-part-1 (2018-03-16)

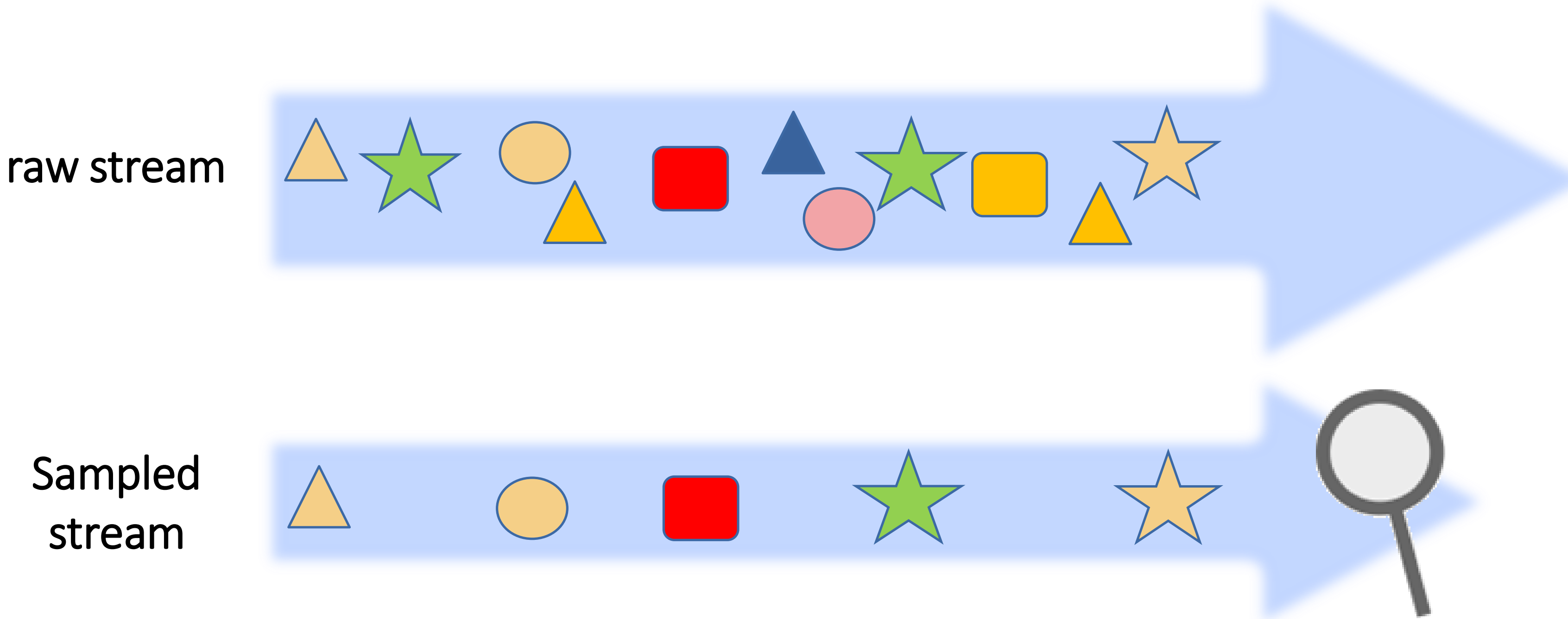
Approximation & Load Shedding

Provide the „Best“ Answer While Avoiding to Fall Behind



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

	Database	Stream
Update rate	Low	High, bursty
Primitive	Persistent collections	Transient streams
Temporal scope	Historical	Windowed
Access	random	sequential
Queries	One-time	Continuous
Query Plans	Static	Dynamic
Precision	Accurate	Approximate

Outline

Introduction
Where From? Where To?

Stream Processing
Big Data + Low Latency

Real-Time Databases
Push-Based Collections

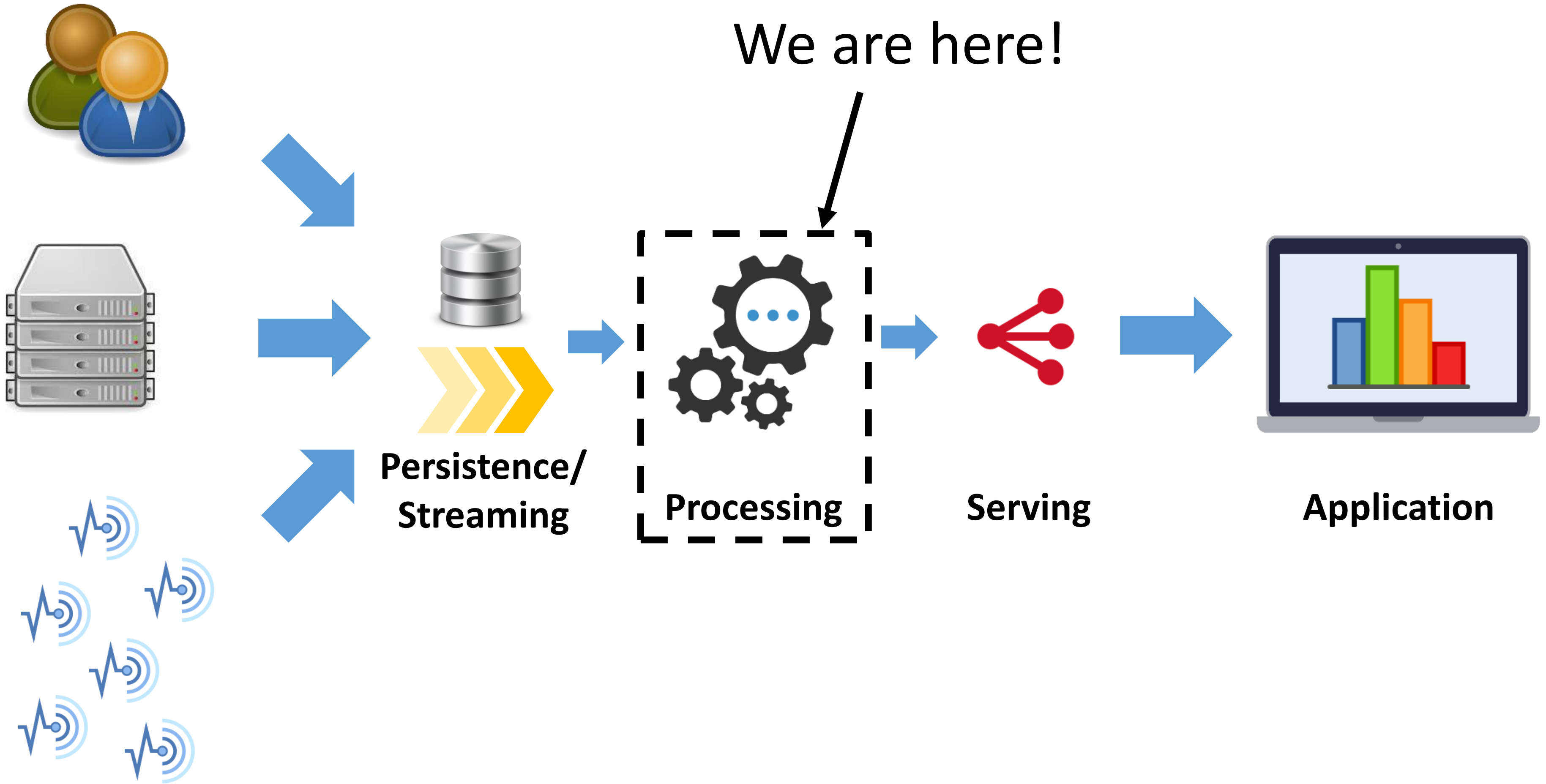
Future Directions
Current Research & Outlook

- **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

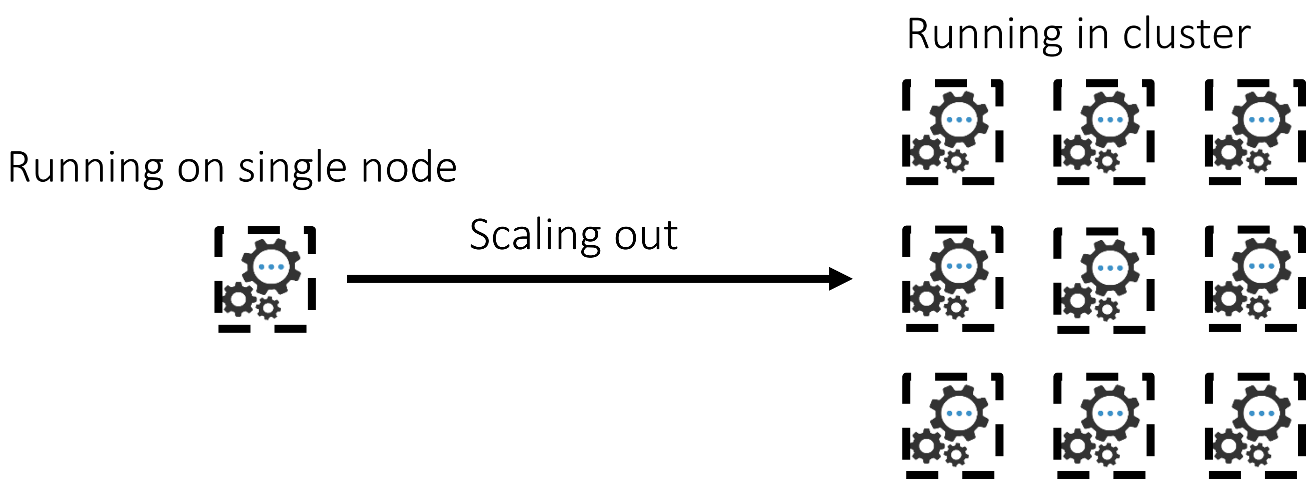


Data Processing Frameworks

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



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

kafka streams

samza

hadoop

concord

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

kafka streams

samza

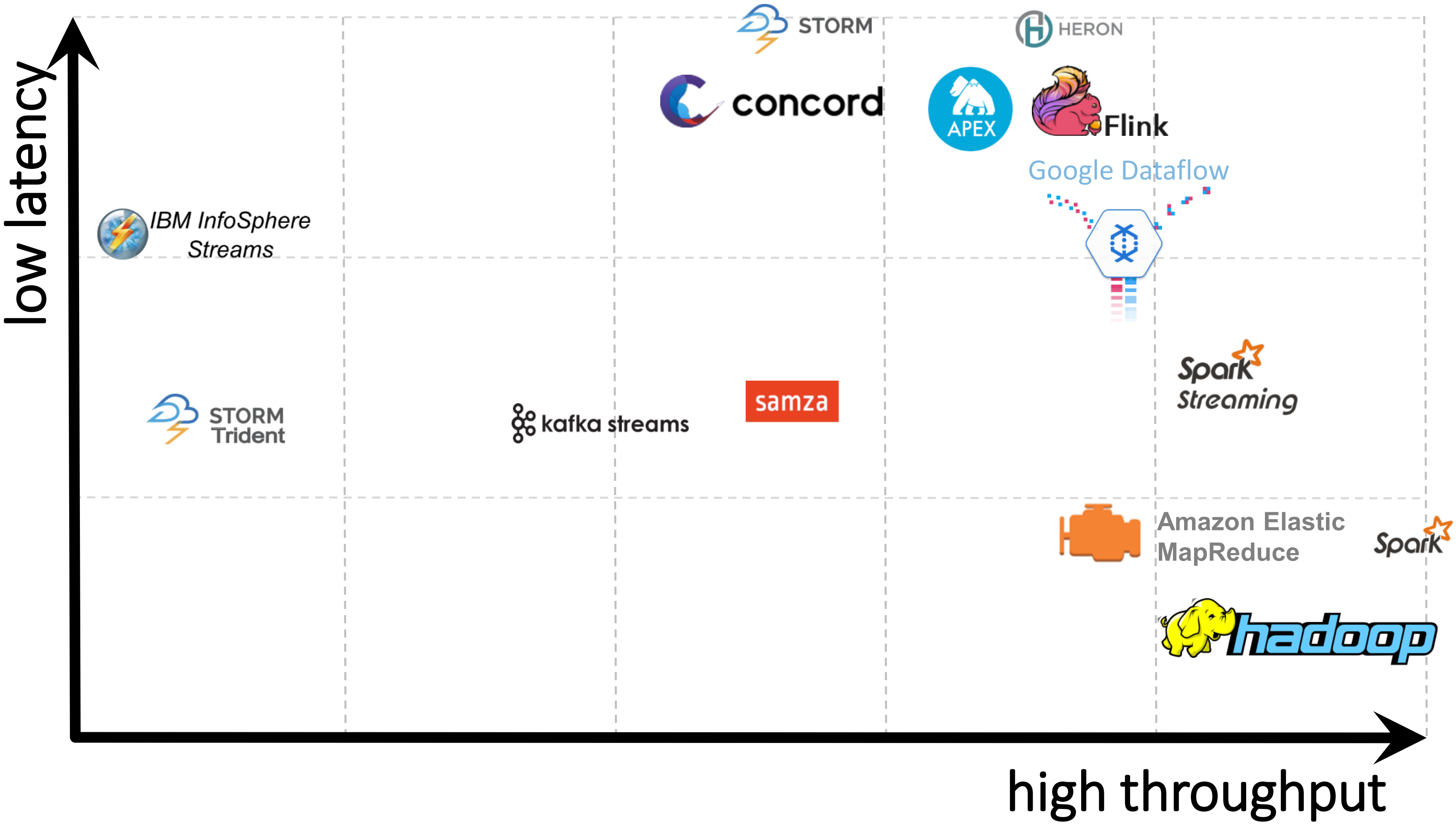
hadoop

concord

What to use when?

Big Data Processing Frameworks

What are your options?



CONCEPTS

Batch vs. Stream Processing

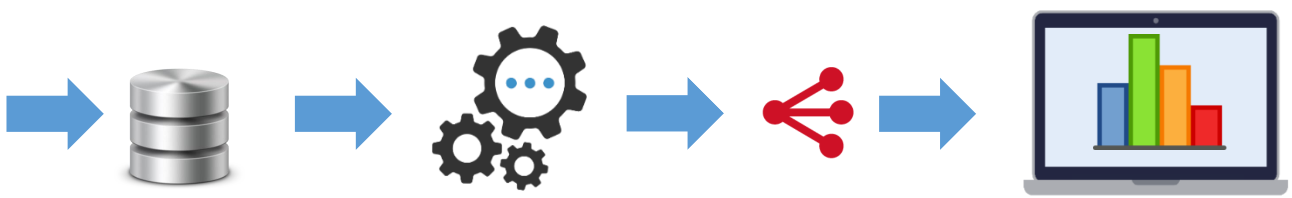
Batch Processing

„Volume“

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

But:

- High latency: periodic jobs (e.g. during night times)



Persistence (e.g. HDFS)

Batch (e.g. MapReduce)

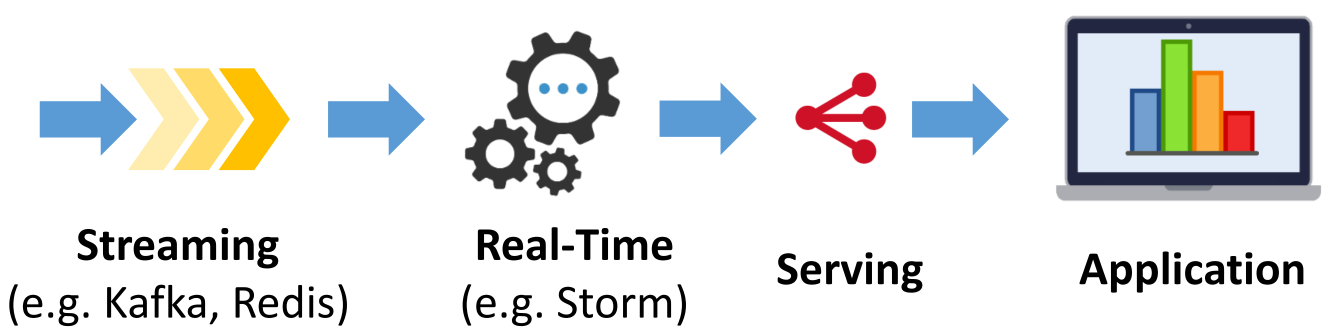
Serving (e.g. HBase)

Application

Stream Processing

„Velocity“

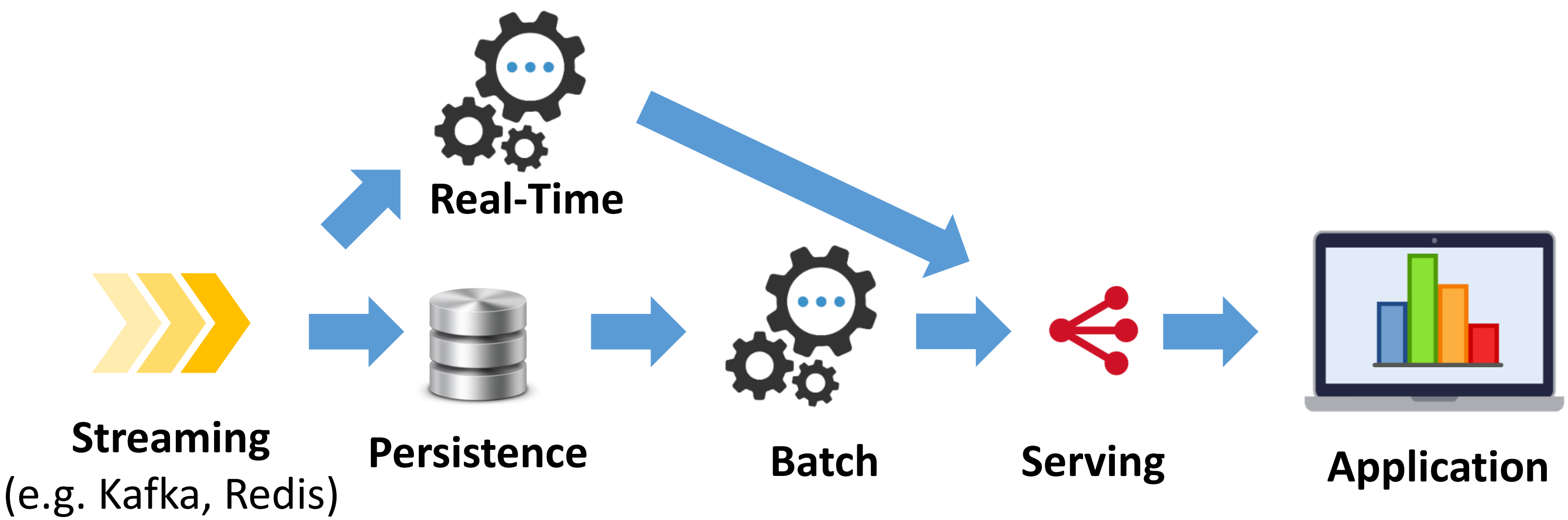
- **Low end-to-end latency**
- Challenges:
  - **Long-running jobs** - no downtime allowed
  - **Asynchronism** - data may arrive delayed or out-of-order
  - **Incomplete input** - algorithms operate on partial data
  - More: fault-tolerance, state management, guarantees, ...



Lambda Architecture

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

- **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

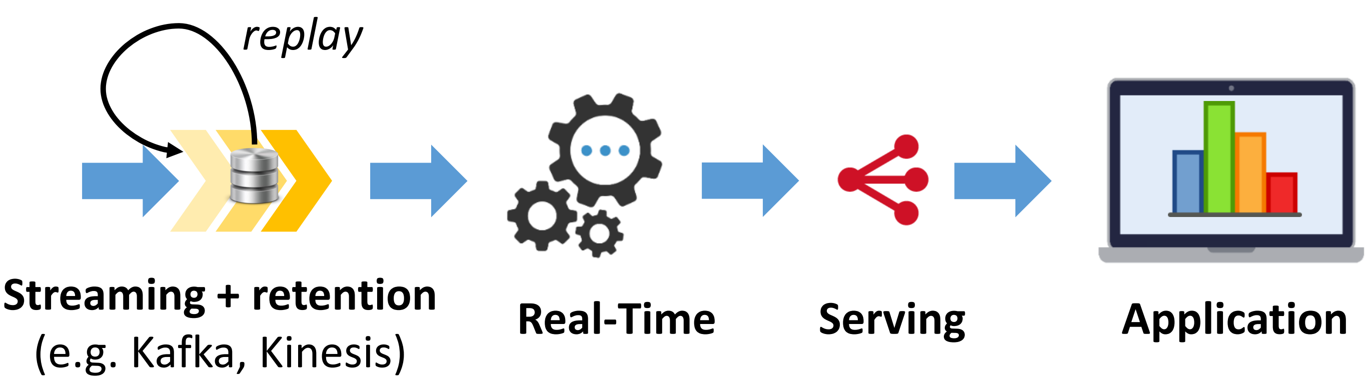


Nathan Marz, *How to beat the CAP theorem* (2011)
http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

Kappa Architecture

Stream(D_{all}) = 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

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, samza

micro-batch — STORM Trident, Spark Streaming

batch — hadoop, Spark, 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, …

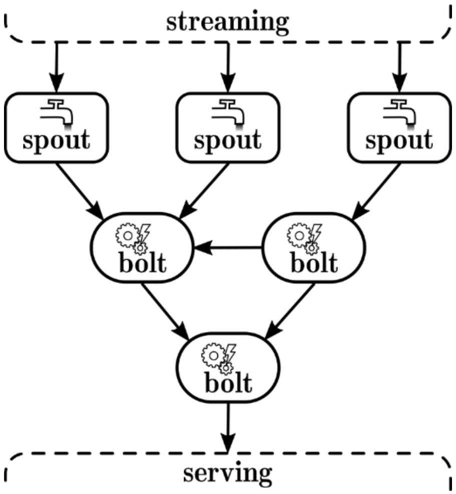
History

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

Dataflow

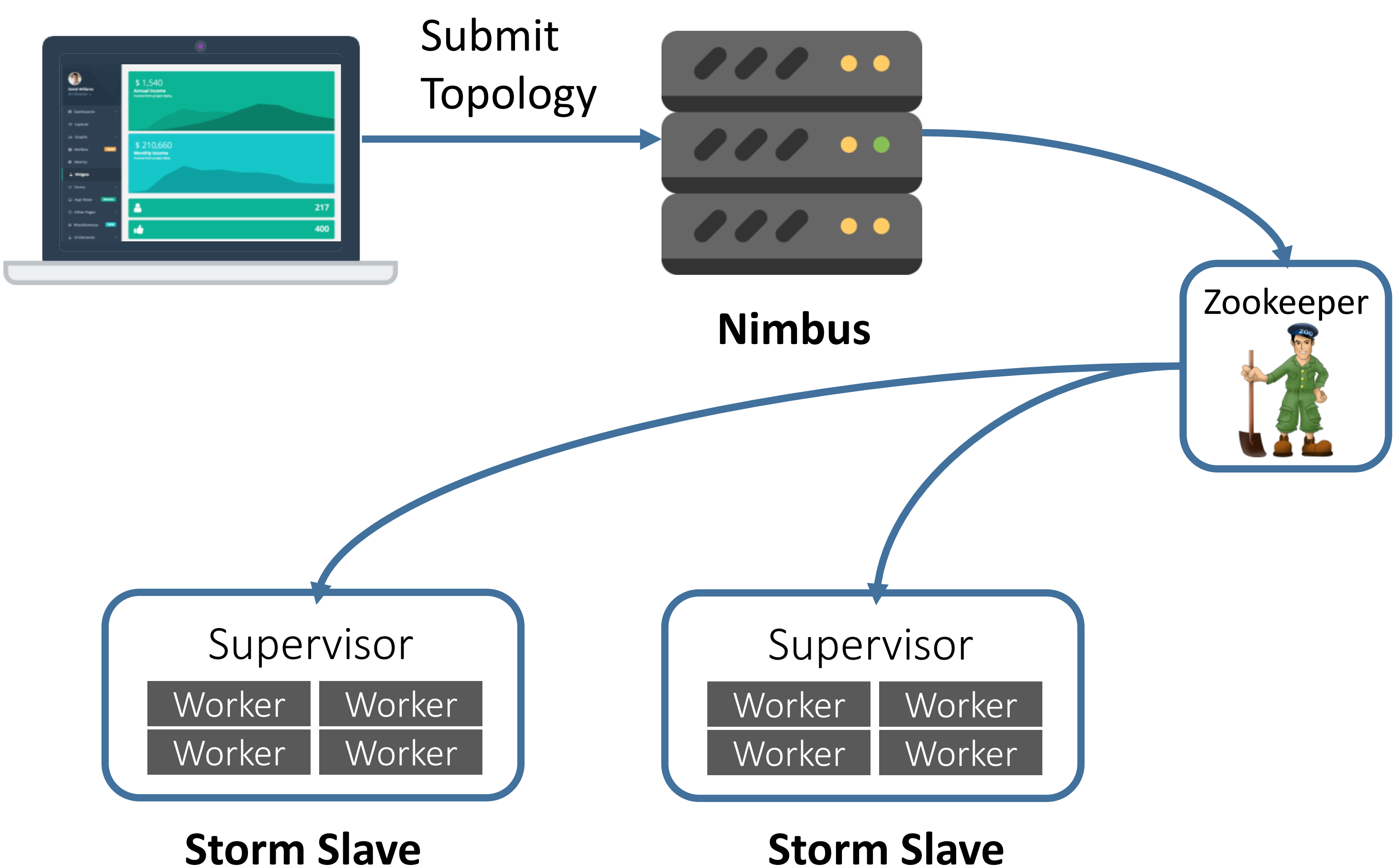
Directed Acyclic Graphs (DAG):

- **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



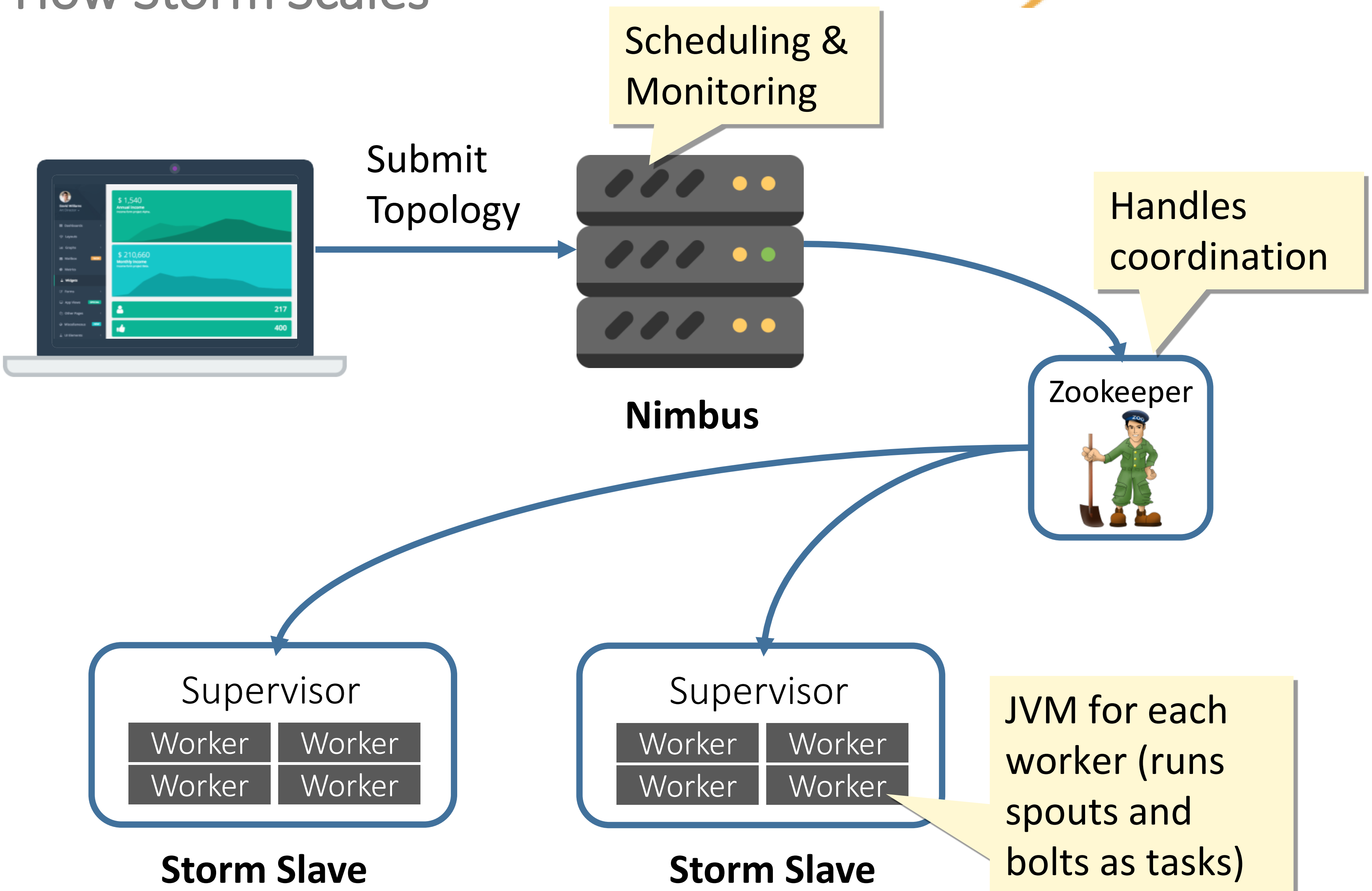
Cluster Architecture

How Storm Scales



Cluster Architecture

How Storm Scales



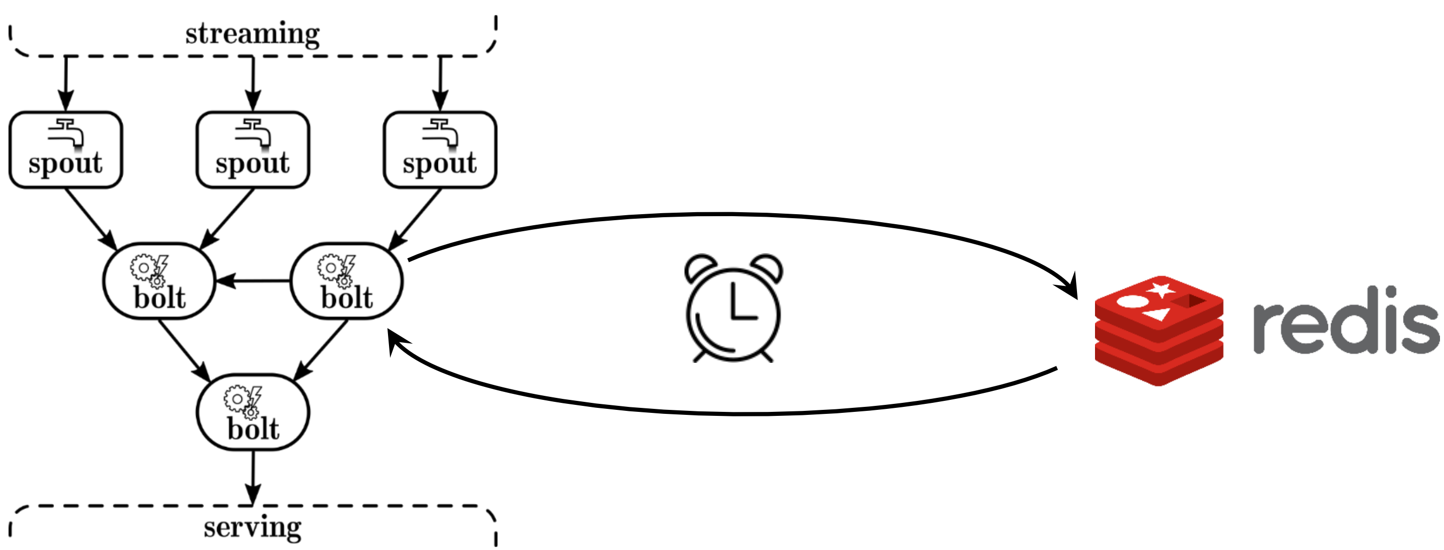
State Management

Recover State on Failure

STORM

- **In-memory** or **Redis**-backed reliable state
- *Synchronous state communication* on the critical path

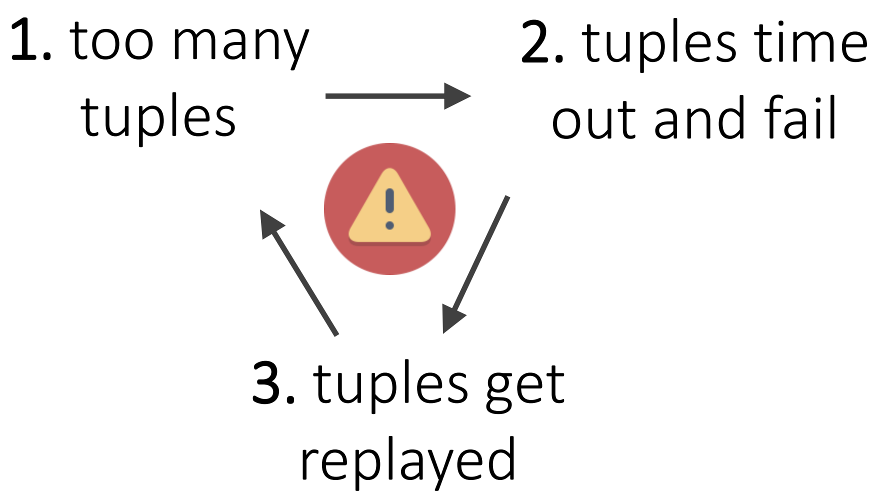
  → infeasible for large state



Back Pressure

Throttling Ingestion on Overload

STORM



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

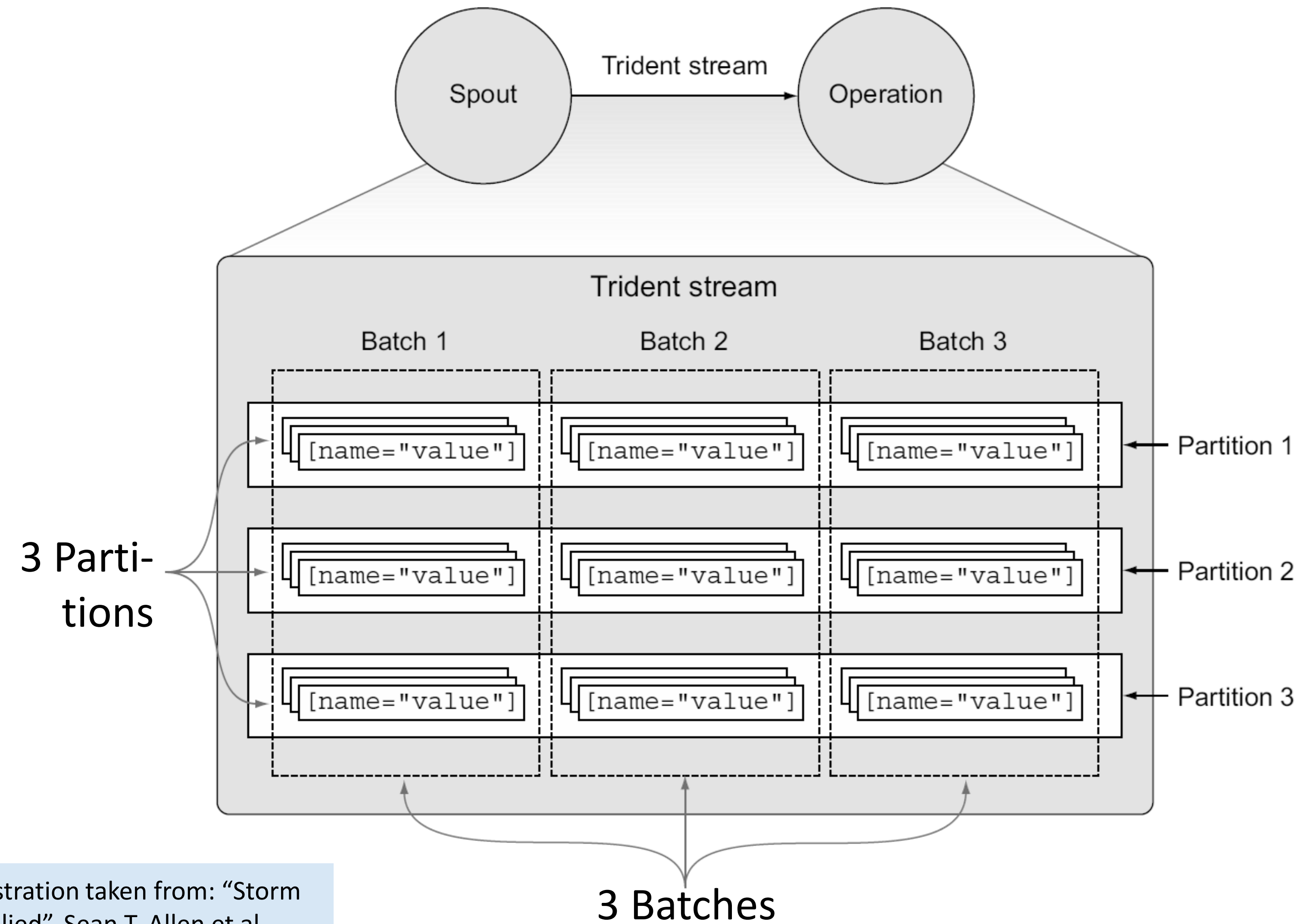
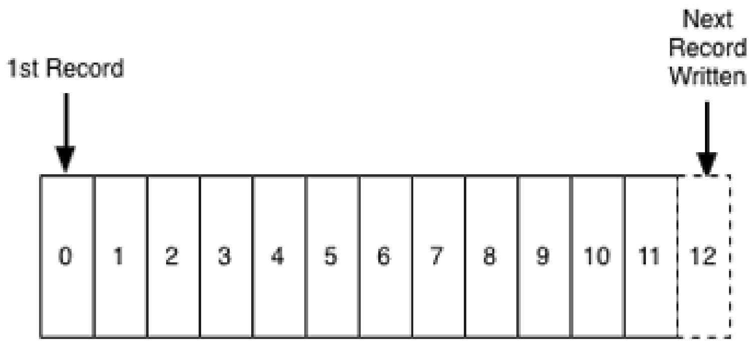


Illustration taken from: “Storm applied”, Sean T. Allen et al.

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

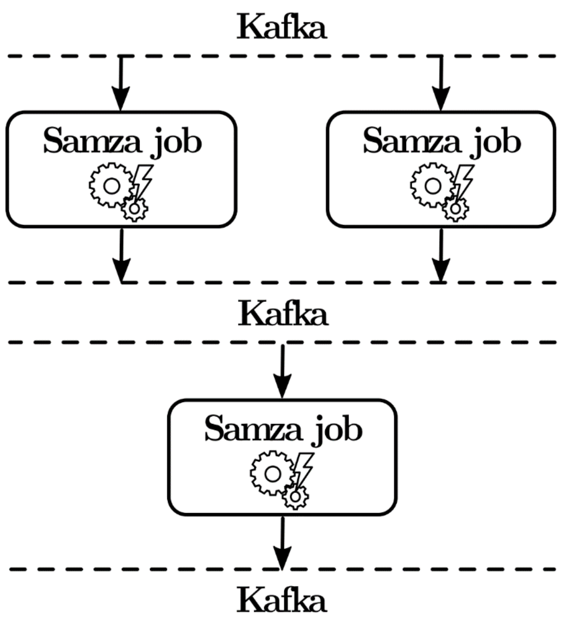
- Developed at **LinkedIn**
- **2013**: open-source (Apache Incubator)
- **2015**: Apache top-level project

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

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

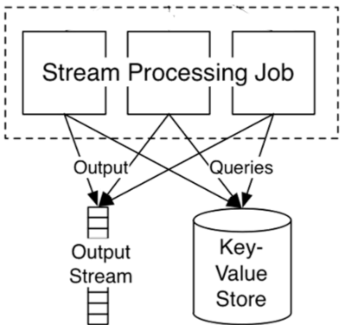


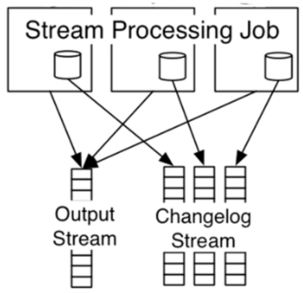
Martin Kleppmann, *Turning the database inside-out with Apache Samza* (2015)
https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/ (2017-02-23)

Samza
Local State

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)
https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing (2017-02-26)

Dataflow

Example: Enriching a Clickstream

Example: the *enriched clickstream* is available to every team within the organization

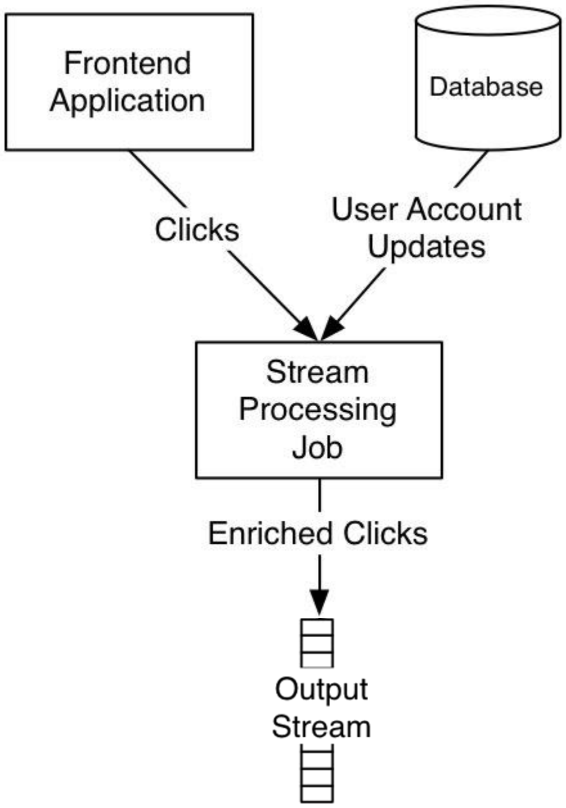


Illustration taken from: Jay Kreps, *Why local state is a fundamental primitive in stream processing* (2014)
https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing (2017-02-26)

State Management

Straightforward Recovery

samza

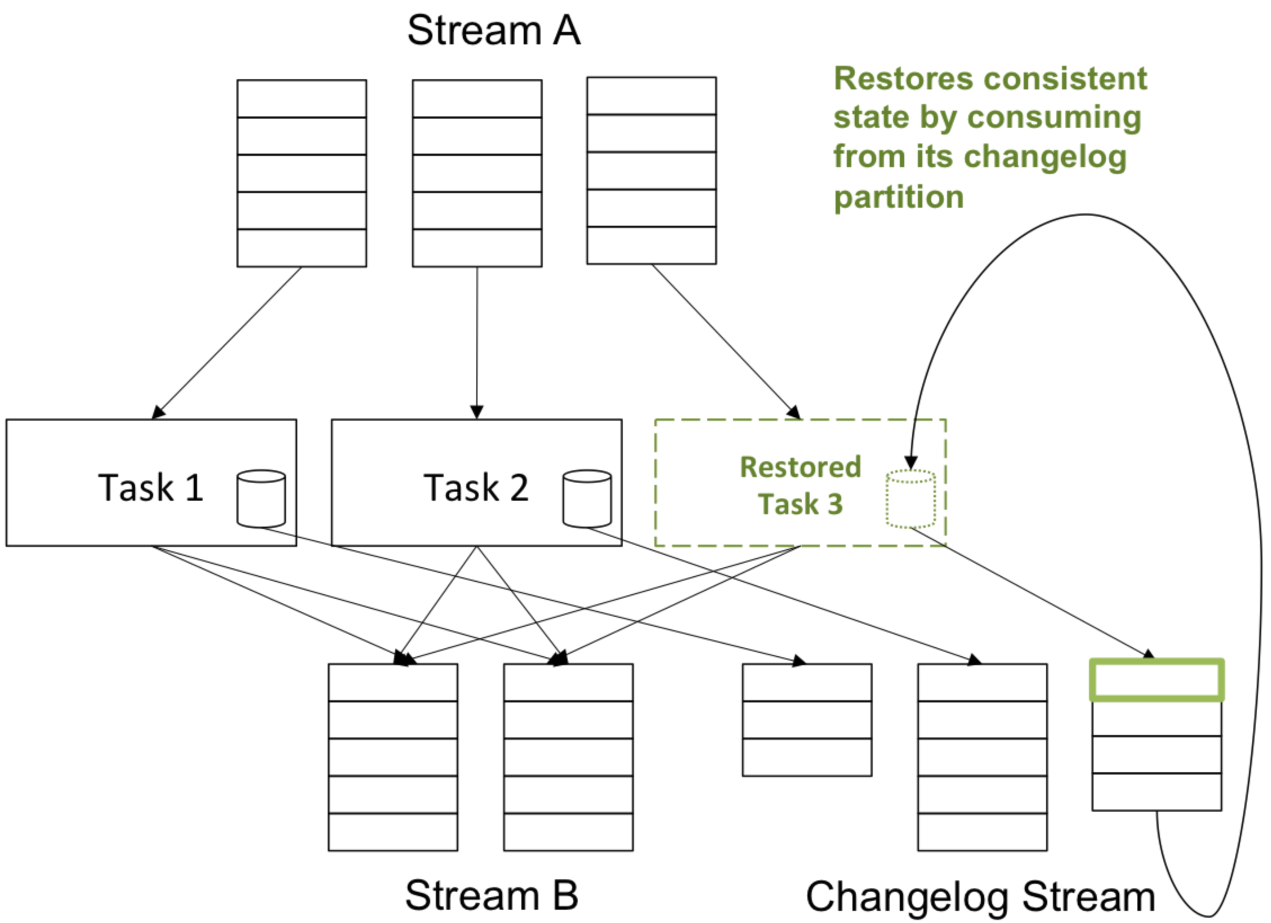


Illustration taken from: Navina Ramesh, *Apache Samza, LinkedIn's Framework for Stream Processing* (2015)
https://thenewstack.io/apache-samza-linkedins-framework-for-stream-processing (2017-02-26)

Spark

„MapReduce successor“

Overview

- **High-level API**: immutable collections (RDDs)

Core	SQL	MLlib	GraphX	Spark Streaming

- **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

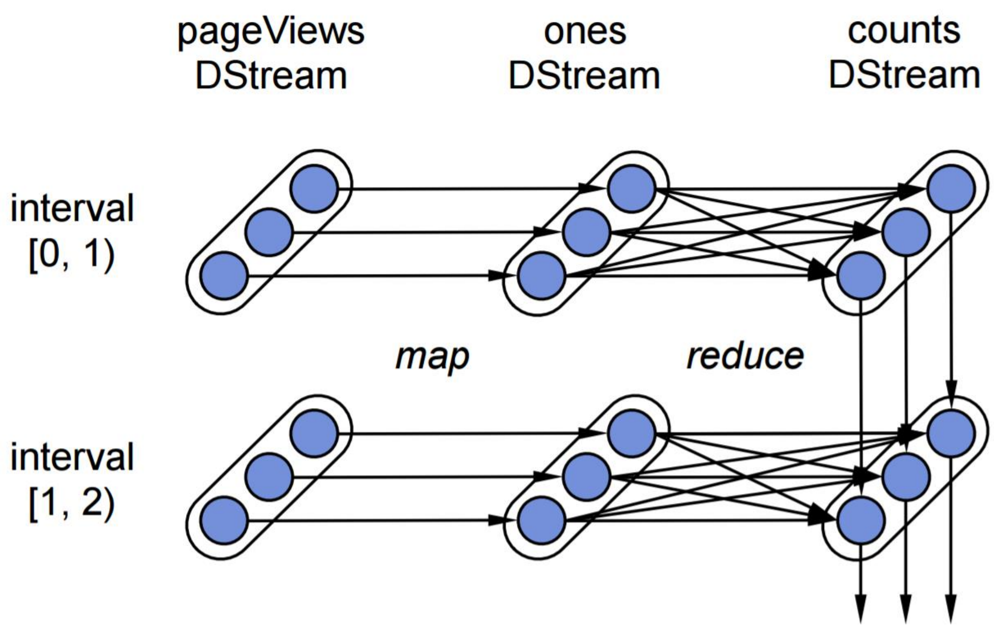
Illustration taken from:
http://spark.apache.org/docs/latest/streaming-programming-guide.html#overview (2017-02-26)

Example

Counting Page Views

Spark Streaming

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



Zaharia, Matei, et al. "Discretized streams: Fault-tolerant streaming computation at scale." *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013.

Flink

Overview

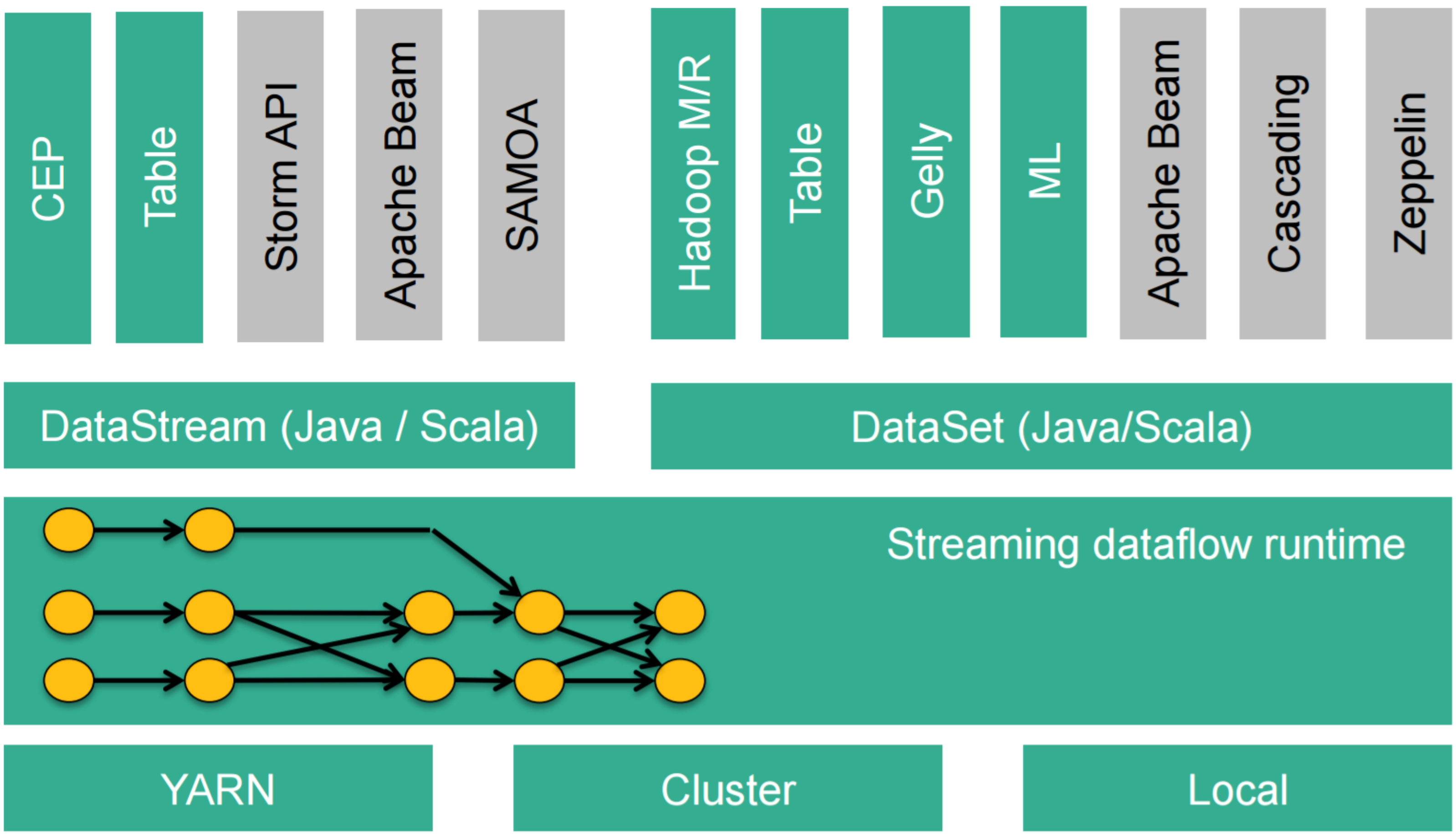
- **Native stream processor:** Latency <100ms feasible
- **Abstract API** for stream and batch processing, stateful, exactly-once delivery
- **Many libraries**: Table and SQL, CEP, Machine Learning , Gelly…
- **Users**: Alibaba, Ericsson, Otto Group, ResearchGate, Zalando…

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

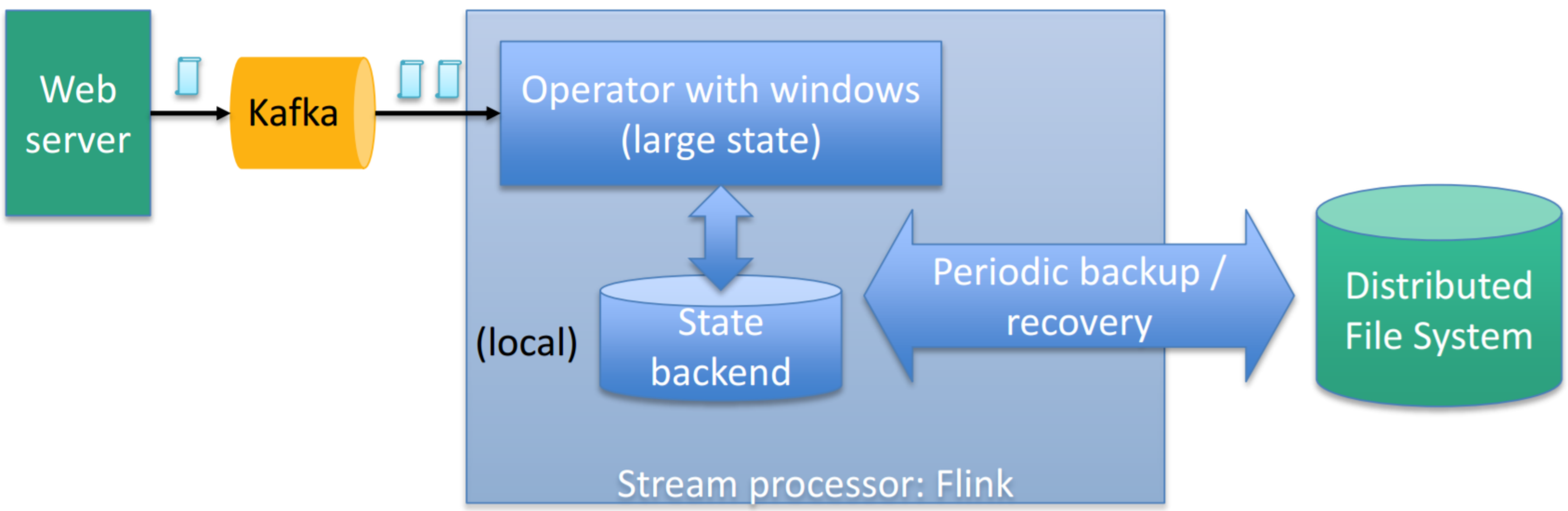


https://www.infoq.com/presentations/stream-processing-apache-flink

Managed State

Streaming + Batch

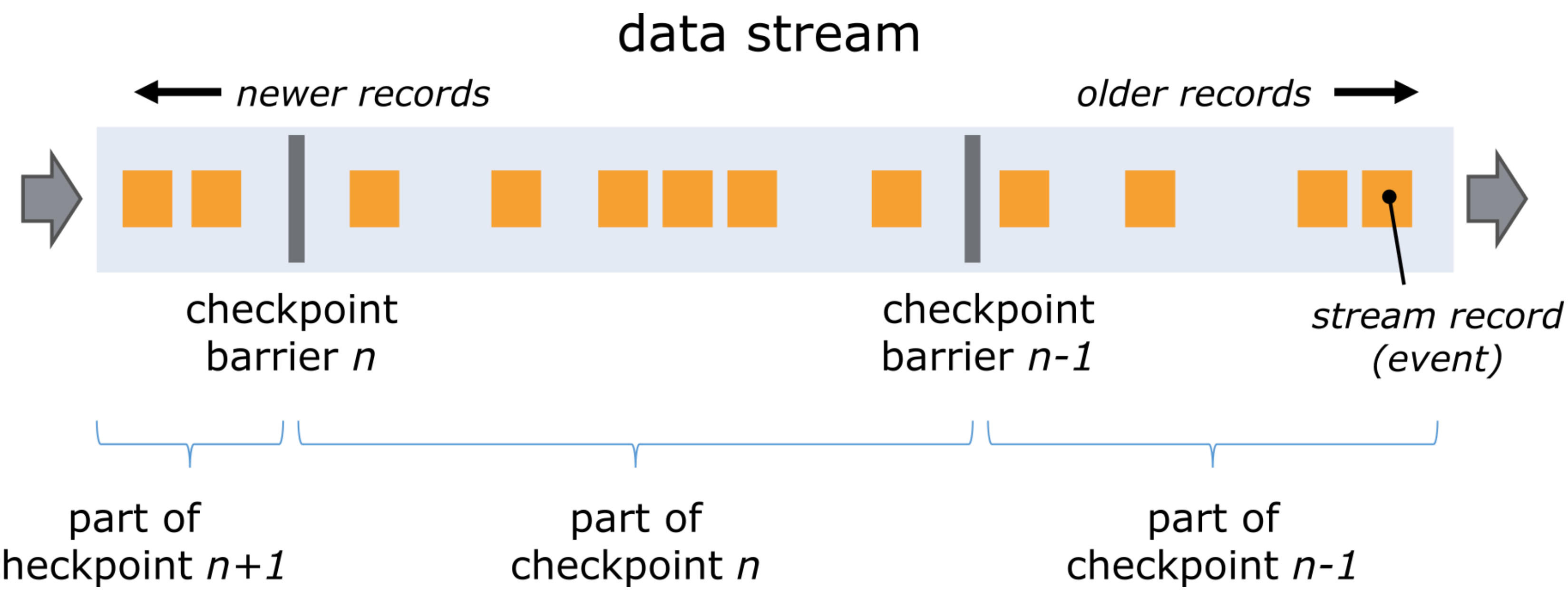
- Automatic **Backups** of local state
- Stored in **RocksDB,** Savepoints written to **HDFS**



https://www.infoq.com/presentations/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

→ **Exactly-once**

Illustration taken from: https://ci.apache.org/projects/flink/flink-docs-release-1.2/internals/stream_checkpointing.html (2017-02-26)

WRAP UP

Side-by-side comparison

Comparison

	Storm	Trident	Samza	Spark Streaming	Flink (streaming)
Strictest Guarantee	at-least-once	exactly-once	at-least-once	exactly-once	exactly-once
Achievable Latency	≪100 ms	<100 ms	<100 ms	<1 second	<100 ms
State Management	○ (small state)	○ (small state)	✓	✓	✓
Processing Model	one-at-a-time	micro-batch	one-at-a-time	micro-batch	one-at-a-time
Backpressure	✓	✓	no (buffering)	✓	✓
Ordering	✗	between batches	within partitions	between batches	within partitions
Elasticity	✓	✓	✗	✓	✗

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**."

From https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Other Systems

Heron

Apex

Dataflow

Beam

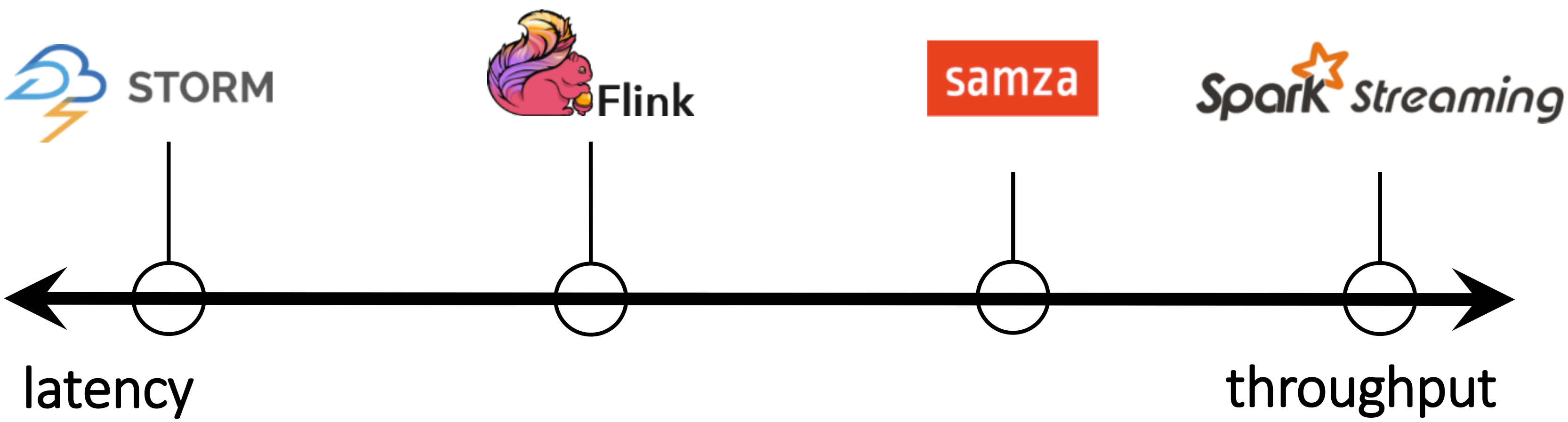
Kafka Streams

IBM InfoSphere Streams

And even more: Kinesis, Gearpump, MillWheel, Muppet, S4, Photon, …

Summary

- Stream Processors:



- **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

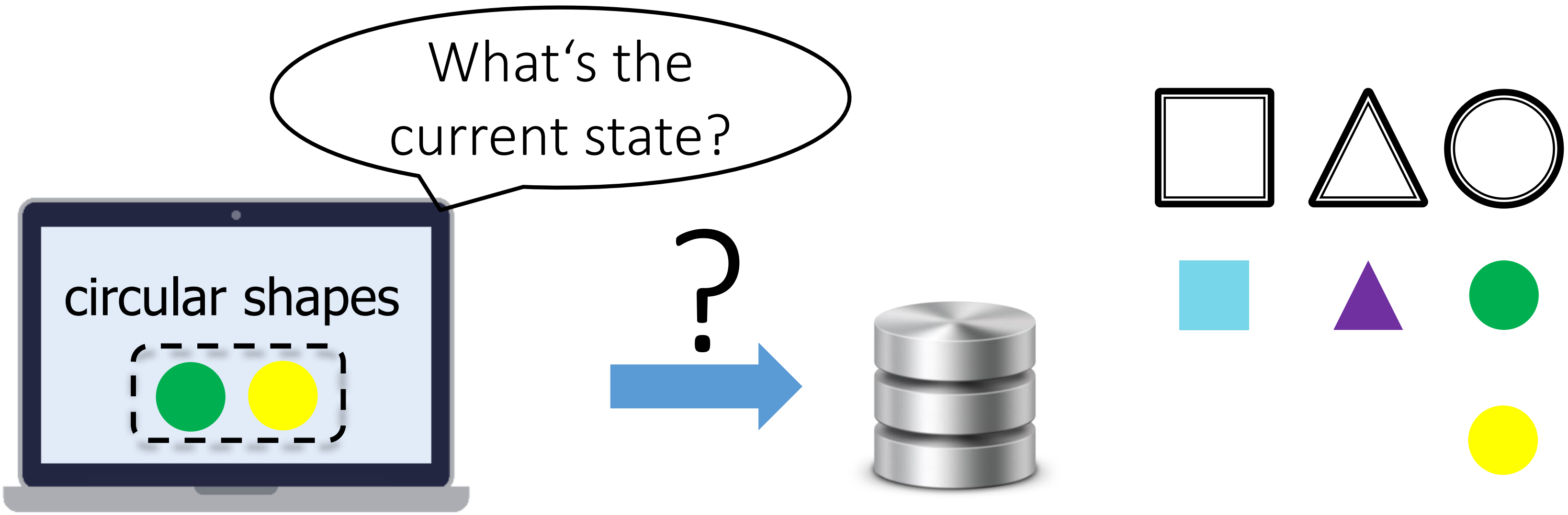
- **Big Picture:**
  - **Why** Push-Based Database Queries?
  - **Where** Do Real-Time Databases Fit in?
- **System Survey:**
  - Meteor
  - RethinkDB
  - Parse
  - Firebase
- **Discussion:**
  - Comparison Matrix
  - Other Systems

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	Real-Time Databases	Data Stream Management	Stream Processing
static collections	evolving collections	persistent/ ephemeral streams	ephemeral streams

pull-based

push-based

REAL-TIME DBS

System Survey

Meteor

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)

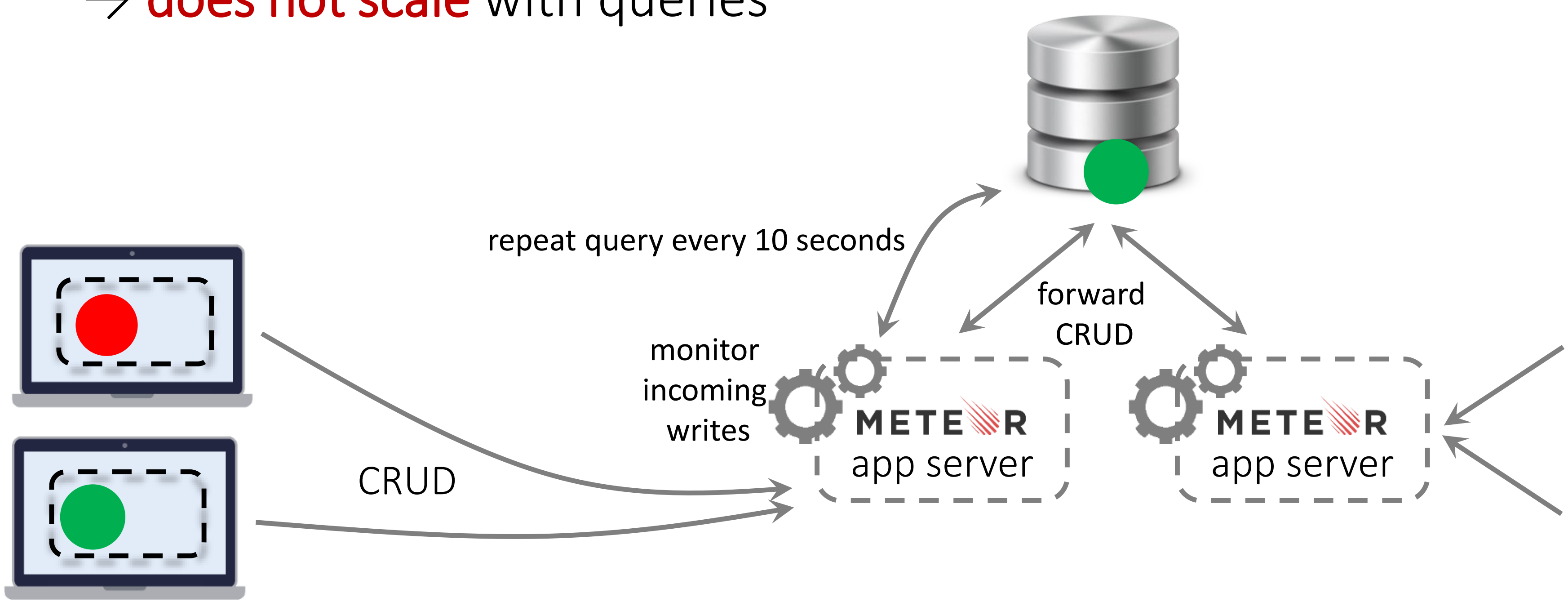
History:

- 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

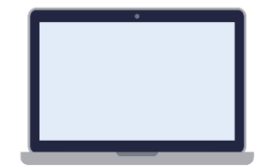


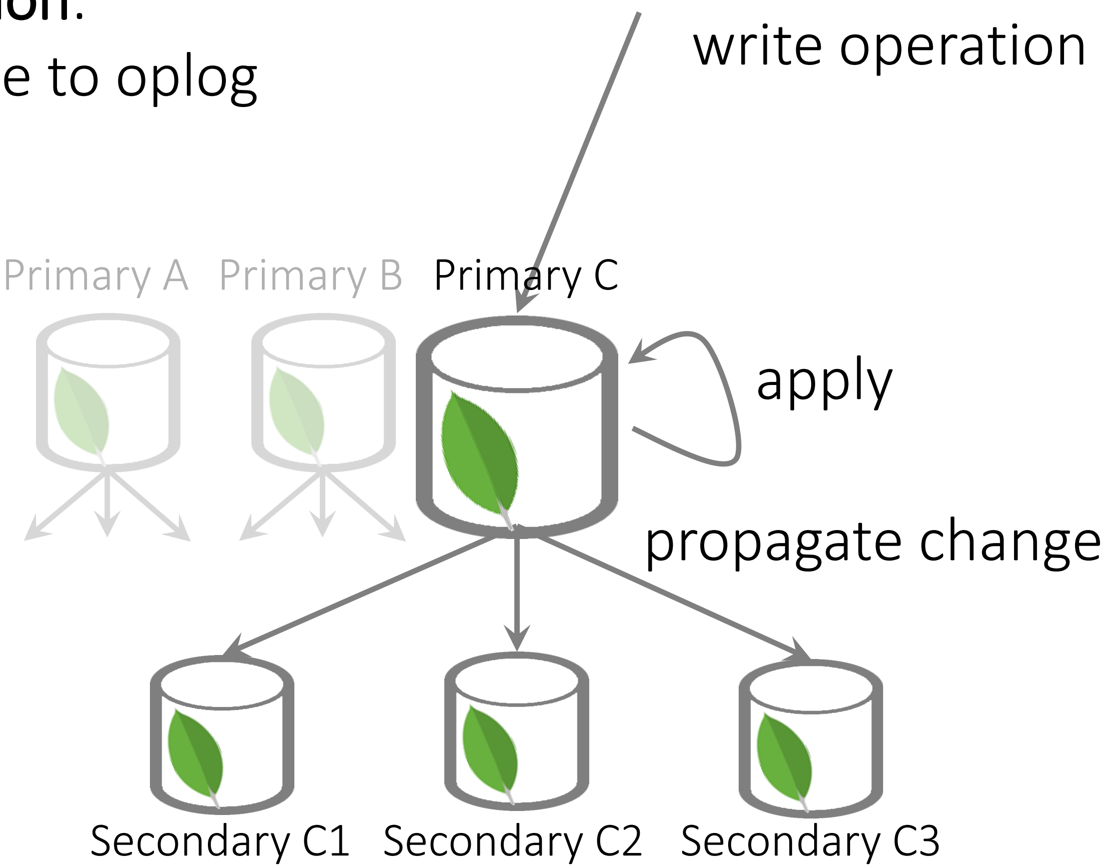
Oplog Tailing

Basics: MongoDB Replication

METEOR

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



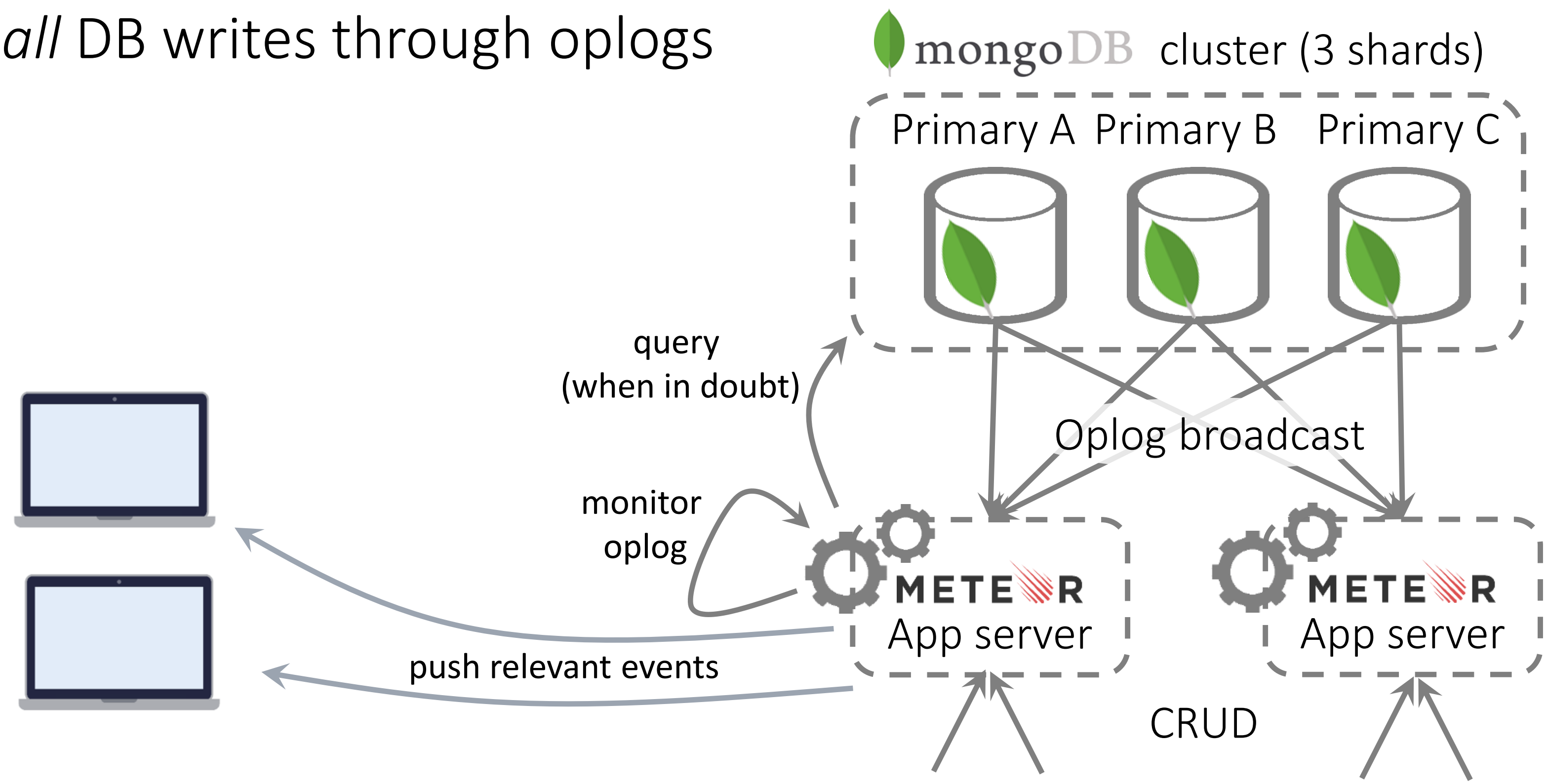


mongoDB cluster
(3 shards)

Oplog Tailing

Tapping into the Oplog

- *Every* Meteor server receives *all* DB writes through oplogs



Oplog Tailing

Oplog Info is Incomplete

METEOR

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

METEOR

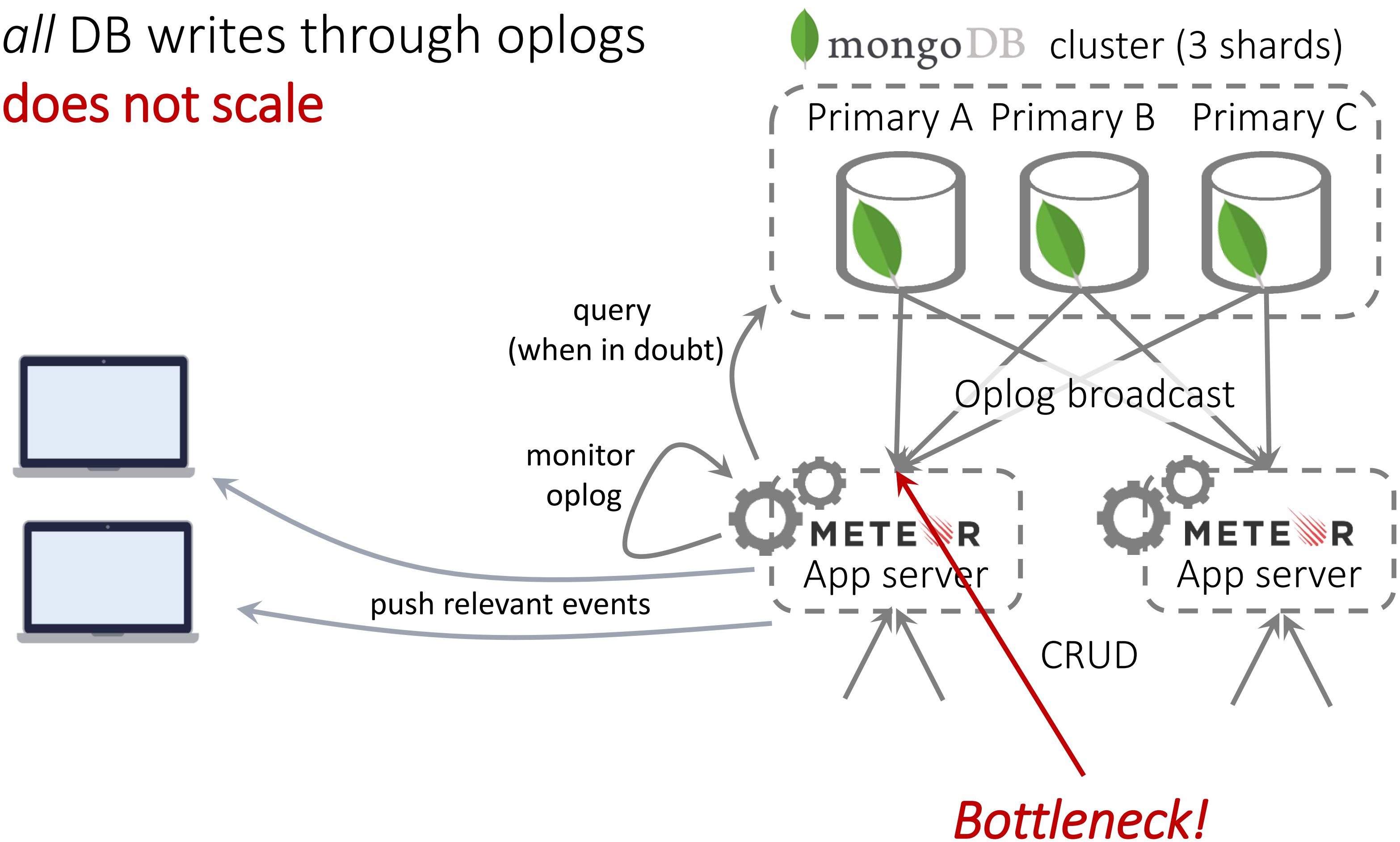
```
1. { name: „Joy“, game: „baccarat“, score: 100 }
2. { name: „Tim“, game: „baccarat“, score: 90 }
3. { name: „Lee“, game: „baccarat“, score: 80 }
```

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:

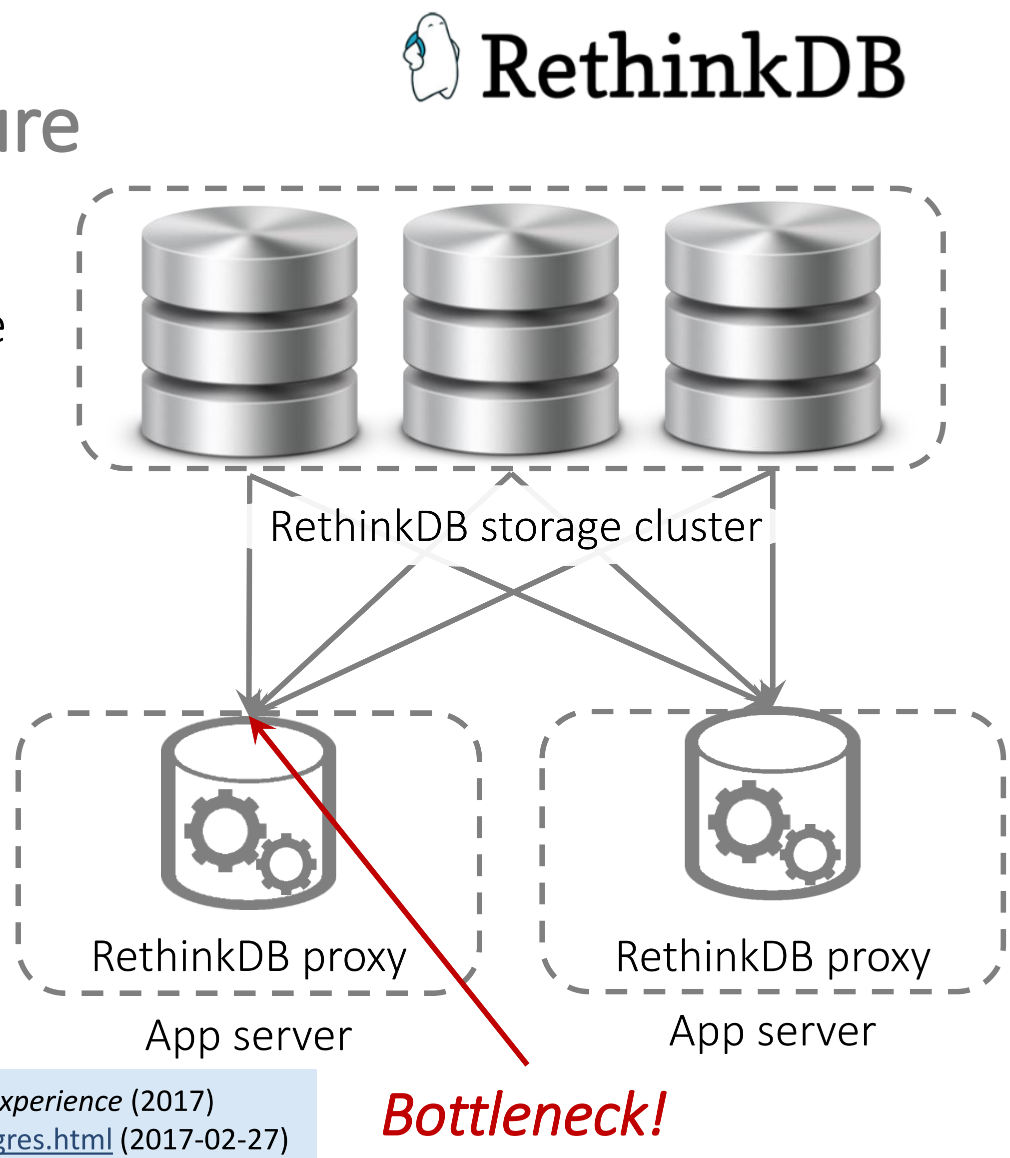
- 2009: RethinkDB is founded
- 2012: RethinkDB is open-sourced under AGPL
- 2016, May: first official release of Horizon (JavaScript SDK)
- 2016, October: RethinkDB announces shutdown
- 2017: RethinkDB is relicensed under Apache 2.0

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



William Stein, *RethinkDB versus PostgreSQL: my personal experience* (2017)
http://blog.sagemath.com/2017/02/09/rethinkdb-vs-postgres.html (2017-02-27)

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

Parse

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:

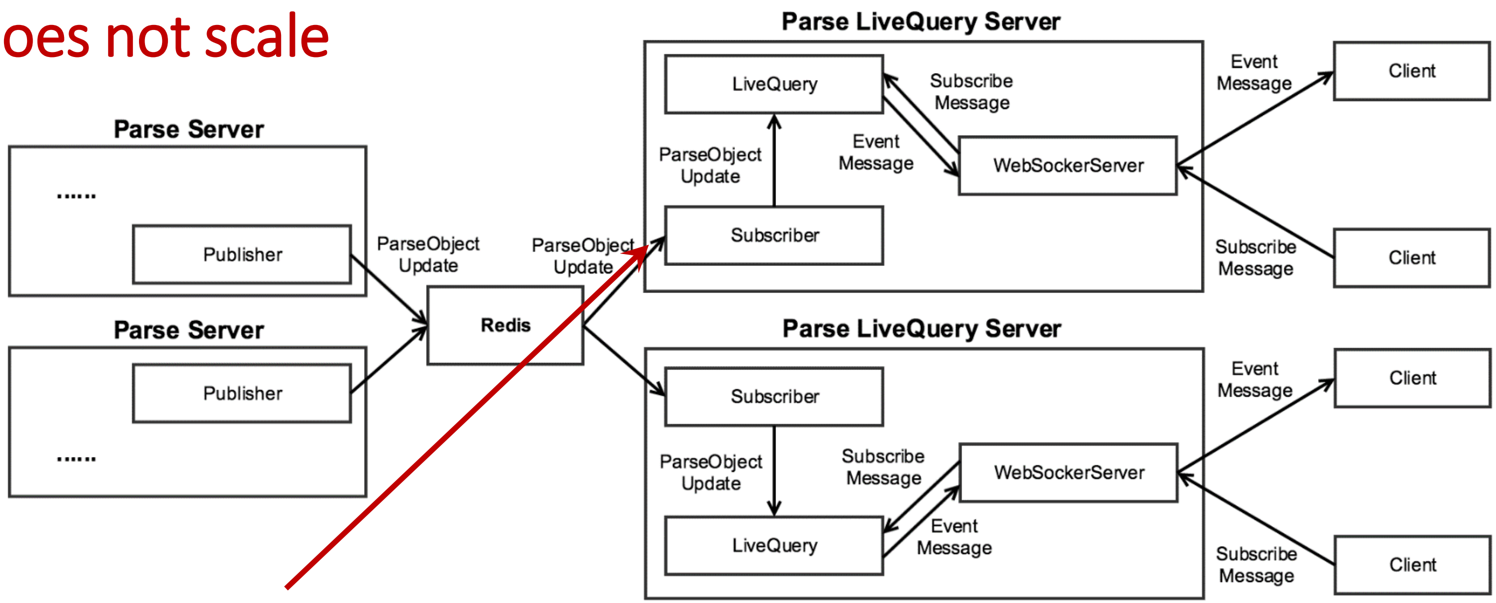
- 2011: Parse is founded
- 2013: Parse is acquired by Facebook
- 2015: more than 500,000 mobile apps reported on Parse
- 2016, January: Parse shutdown is announced
- 2016, March: **Live Queries** are announced
- 2017: Parse shutdown is finalized

Parse

LiveQuery Architecture

Parse

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



Bottleneck!

Illustration taken from:
http://parseplatform.github.io/docs/parse-server/guide/#live-queries (2017-02-22)

Firebase

Firebase

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!

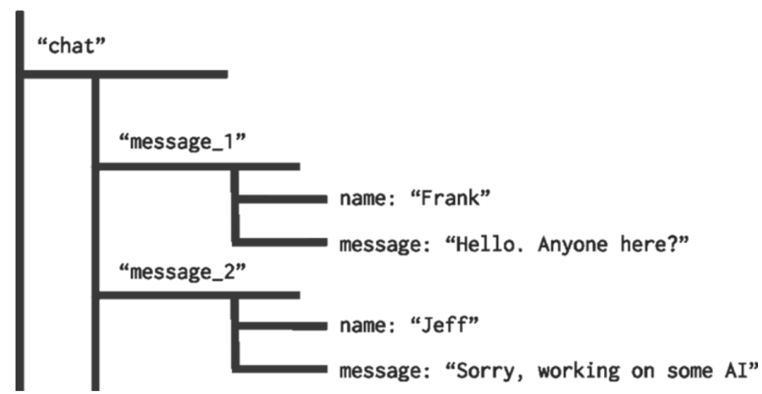
Illustration taken from: Frank van Puffelen, *Have you met the Realtime Database? (2016)*
https://firebase.googleblog.com/2016/07/have-you-met-realtime-database.html (2017-02-27)

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)*
https://firebase.googleblog.com/2016/07/have-you-met-realtime-database.html (2017-02-27)

Firebase

Hard Scaling Limits

Firebase

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

Firebase, *Choose a Database: Cloud Firestore or Realtime Database (2018)*
https://firebase.google.com/docs/database/rtdb-vs-firestore (2018-03-10)

Firebase

Firestore: New Model

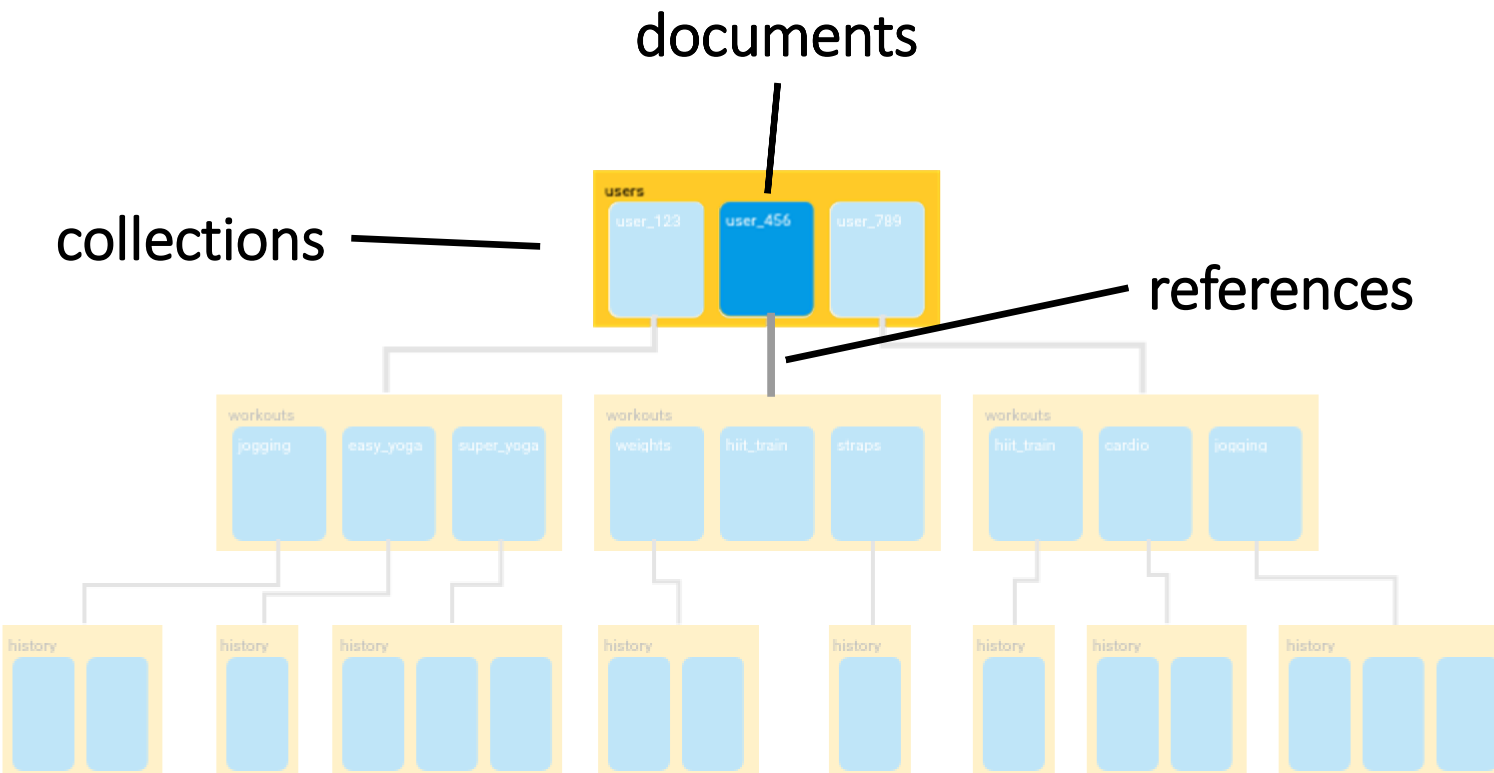


Illustration taken from: Todd Kerpelman, *Cloud Firestore for Realtime Database Developers (2017)*
https://firebase.googleblog.com/2017/10/cloud-firestore-for-rtdb-developers.html (2018-03-10)

Firebase

Firestore: New Model

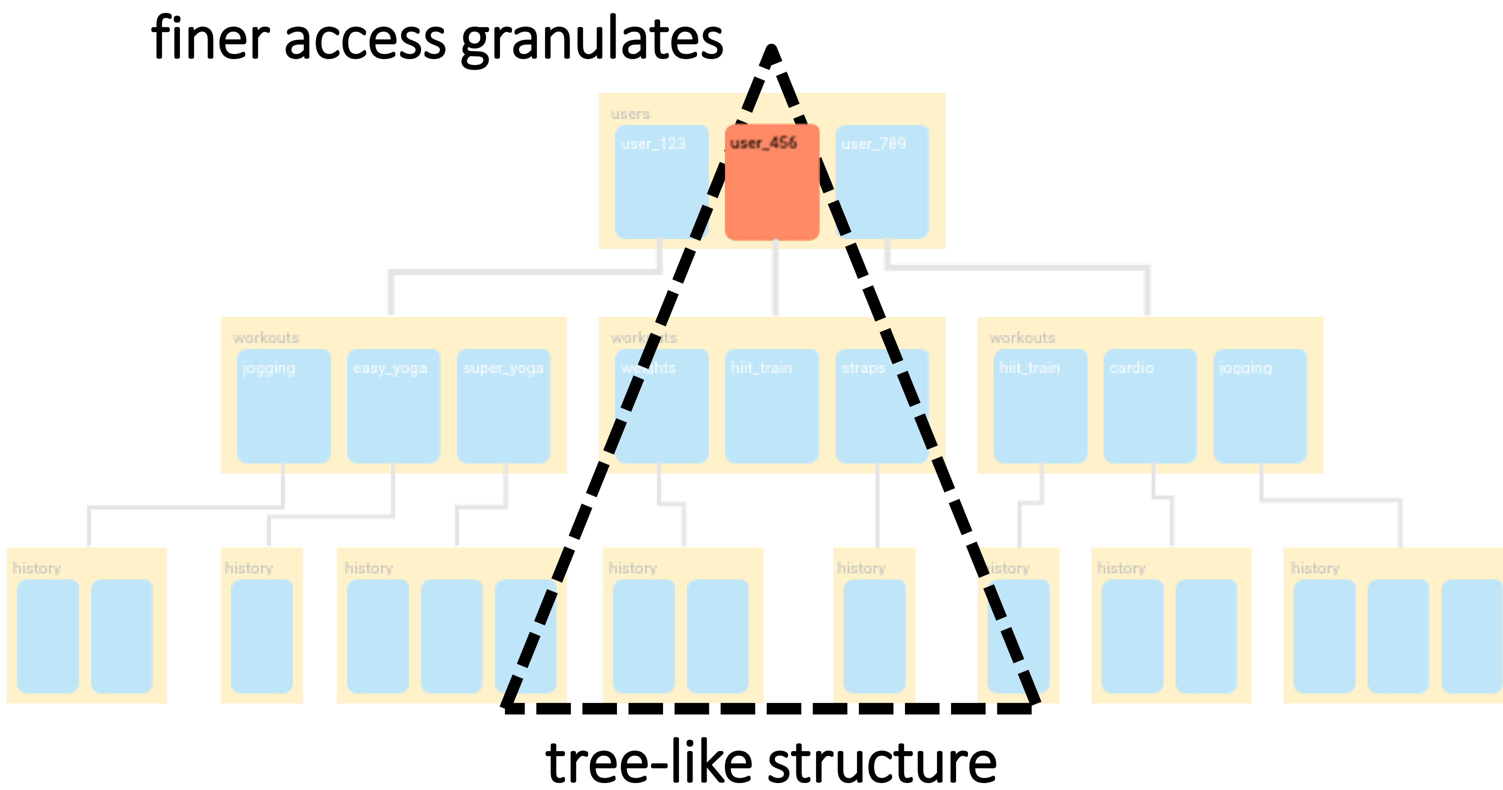


Illustration taken from: Todd Kerpelman, *Cloud Firestore for Realtime Database Developers (2017)*
https://firebase.googleblog.com/2017/10/cloud-firestore-for-rtdb-developers.html (2018-03-10)

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

Apache CouchDB relax

OrientDB®

elasticsearch

mongoDB

RxDB

realm

REAL-TIME DBS

Summary & Discussion

Wrap-Up

Direct Comparison

	Meteor Poll-and-Diff	 Oplog Tailing	RethinkDB	Parse	Firebase	Baqend
Scales with write TP	✓	✗	✗	✗	✗	✓
Scales with no. of queries	✗	✓	✓	✓	? (100k connections)	✓
Composite queries (AND/OR)	✓	✓	✓	✓	○ (AND In Firestore)	✓
Sorted queries	✓	✓	✓	✗	○ (single attribute)	✓
Limit	✓	✓	✓	✗	✓	✓
Offset	✓	✓	✗	✗	○ (value-based)	✓

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

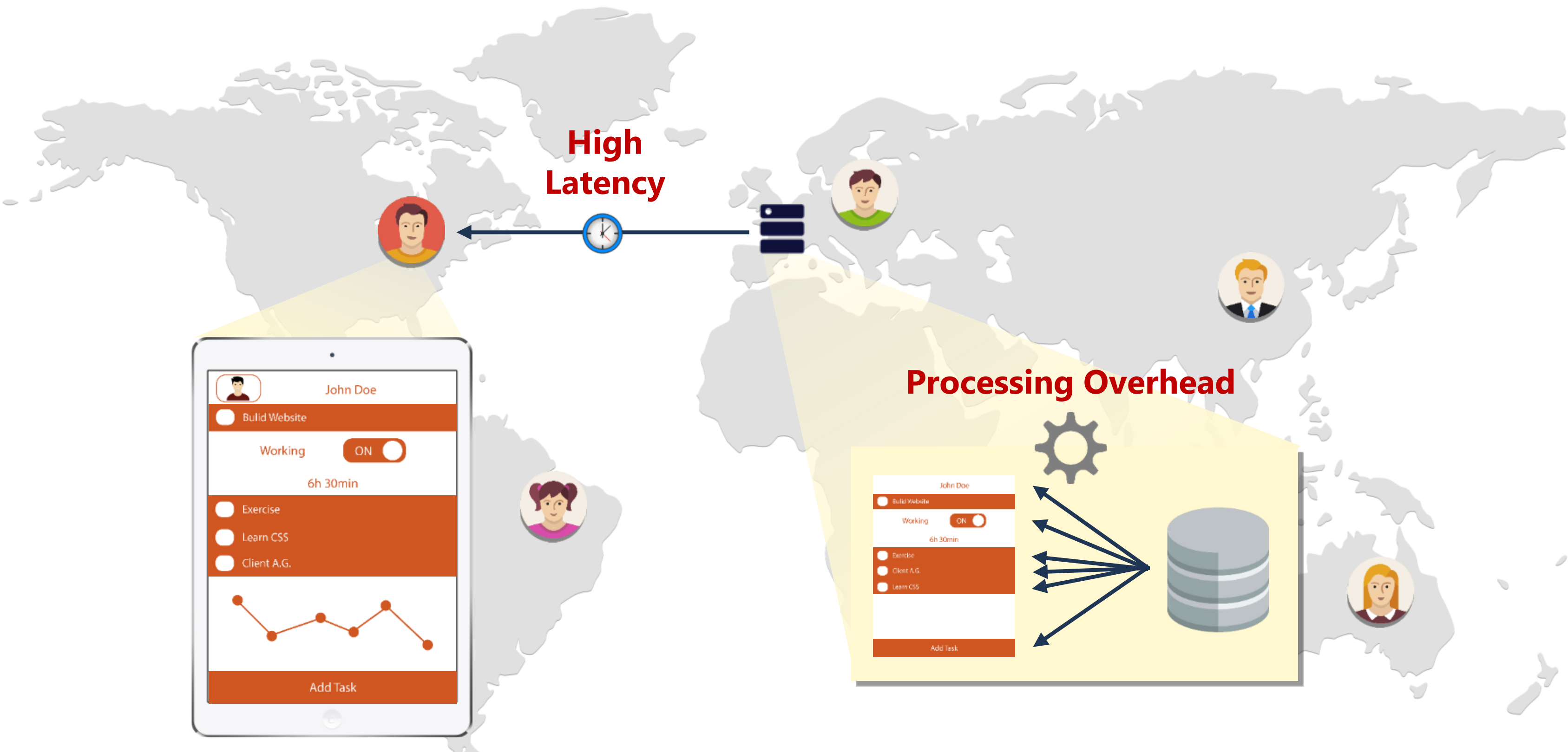
- **Caching Dynamic Data:**
  - Why is the Web Slow?
  - Caching to the Rescue!
  - Query Caching
- **Real-Time Queries:**
  - Scalability
  - Expressiveness
  - Legacy Compatibility
  - Use Cases
- **Open Challenges:**
  - TTLs & Transactions
  - Polyglot Persistence
- **Summary**

OUTLOOK

Our Research at the University of Hamburg

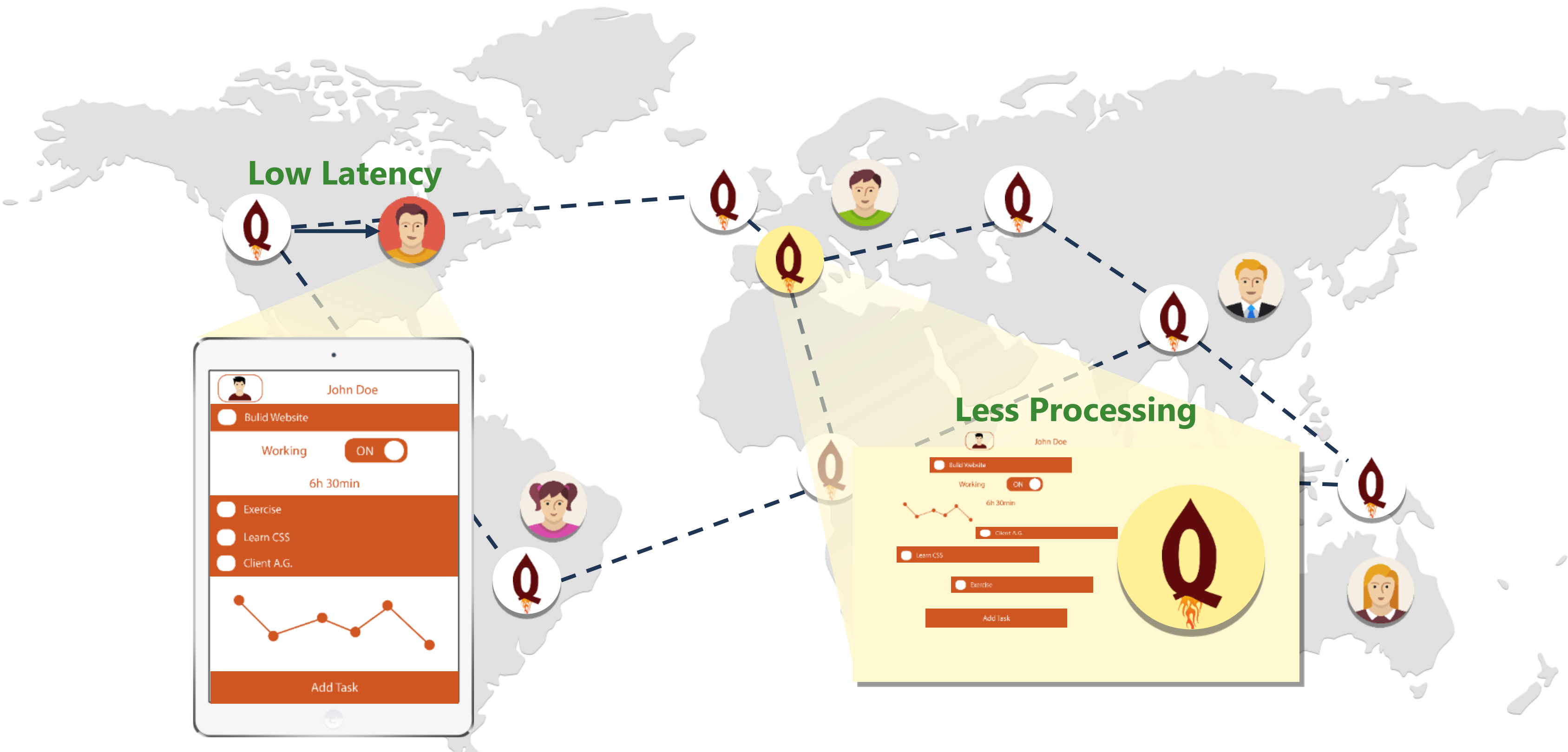
Problem: Slow Websites

Two Bottlenecks: Latency and Processing



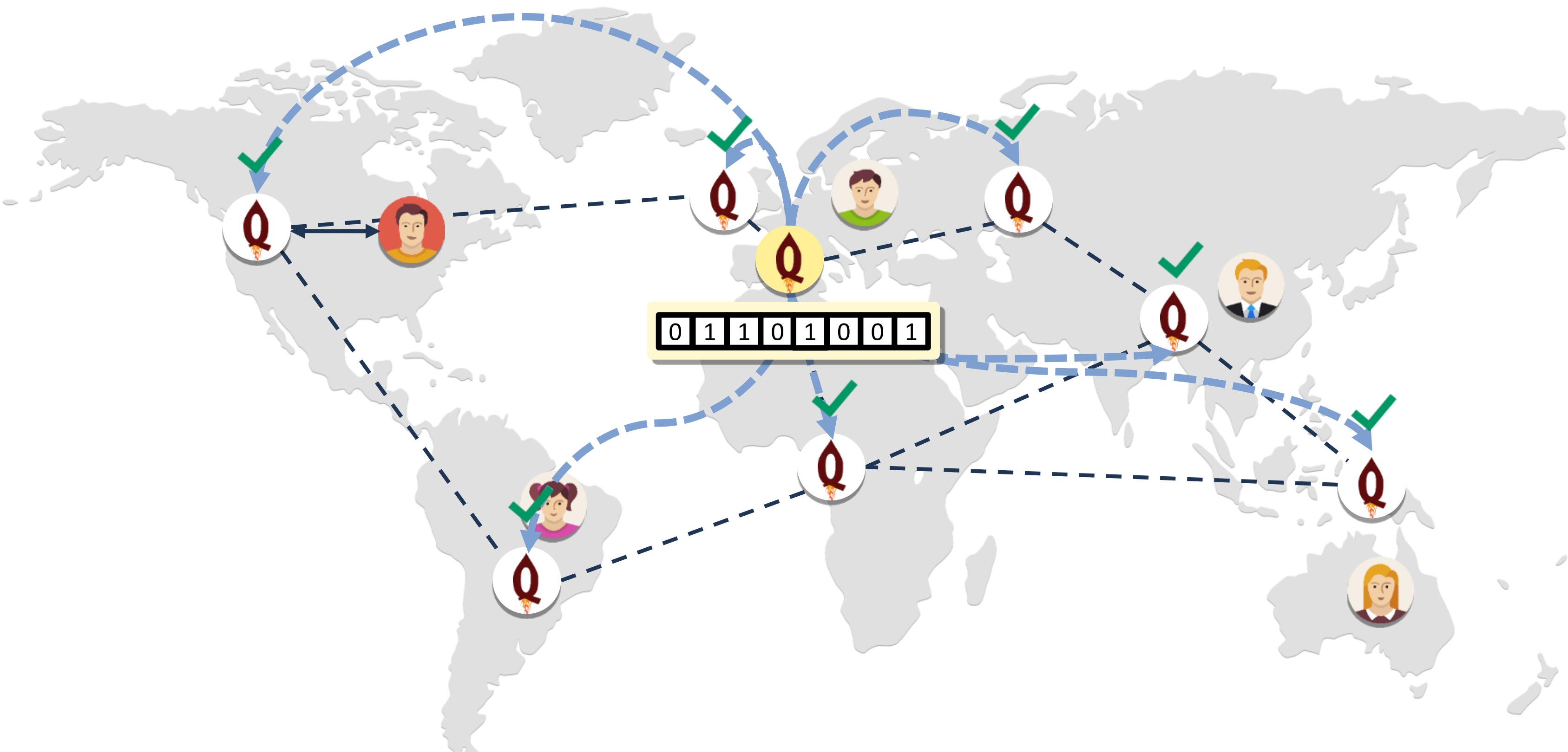
Solution: Global Caching

Fresh Data From Distributed Web Caches



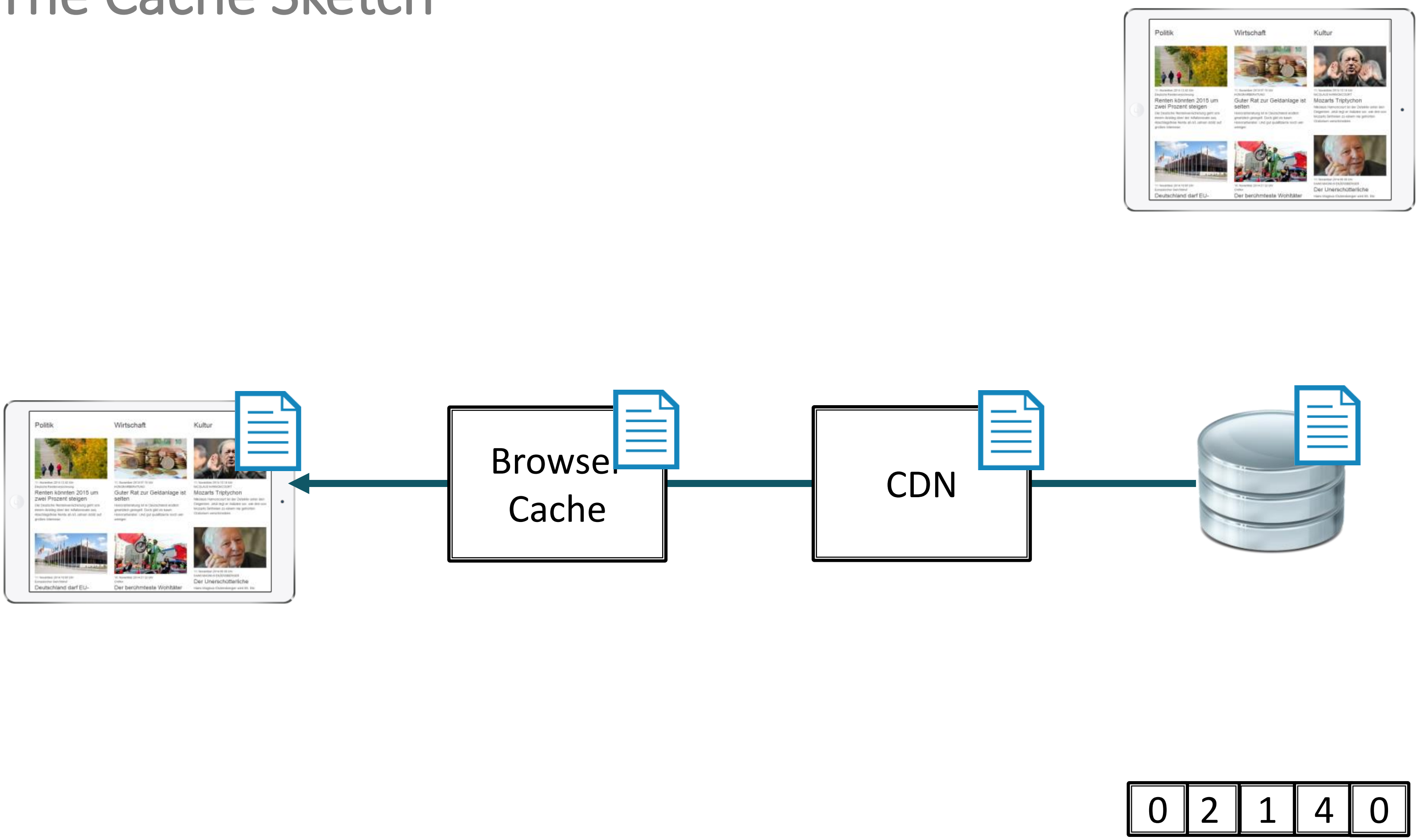
New Caching Algorithms

Solve Consistency Problem



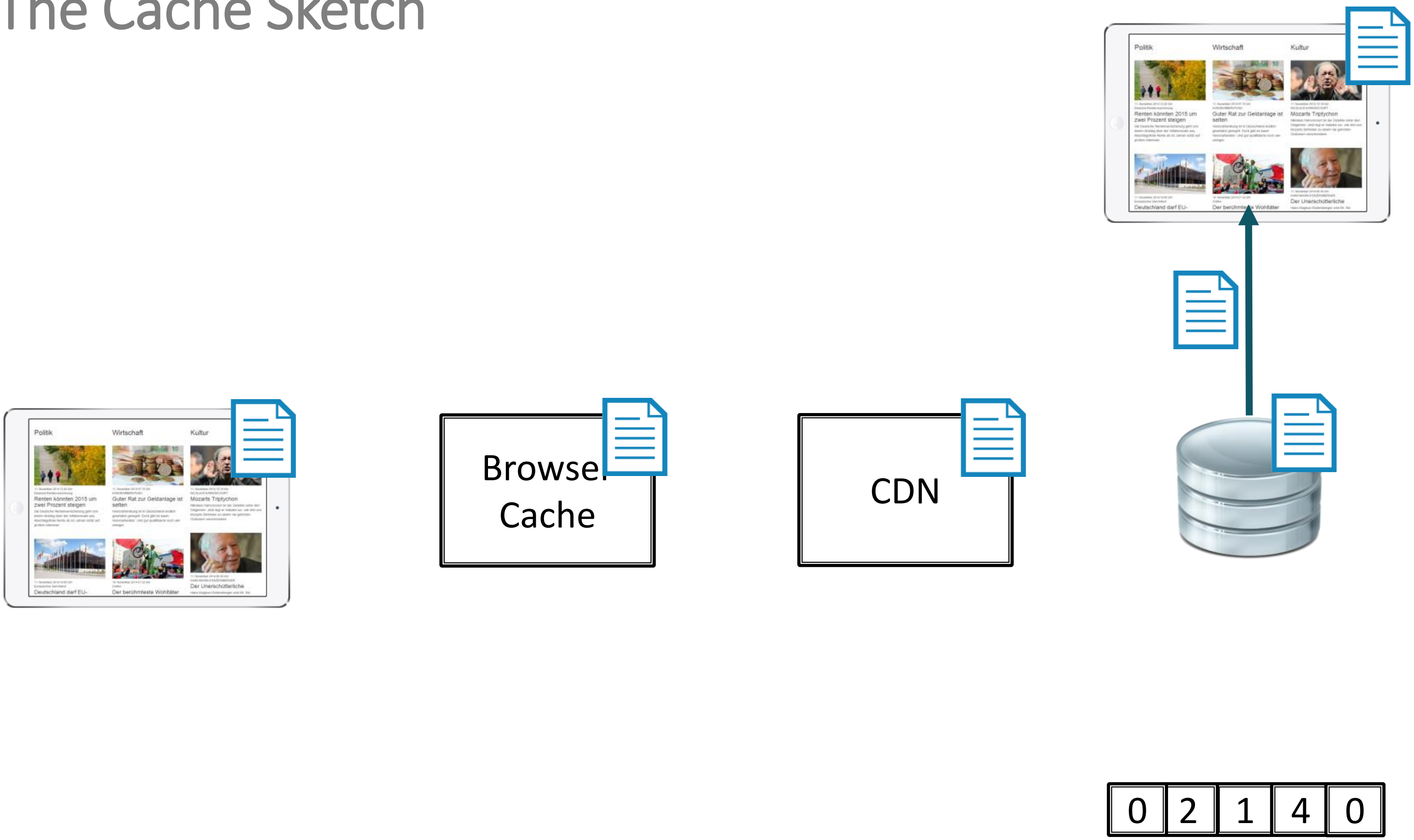
Consistent Web Caching

The Cache Sketch



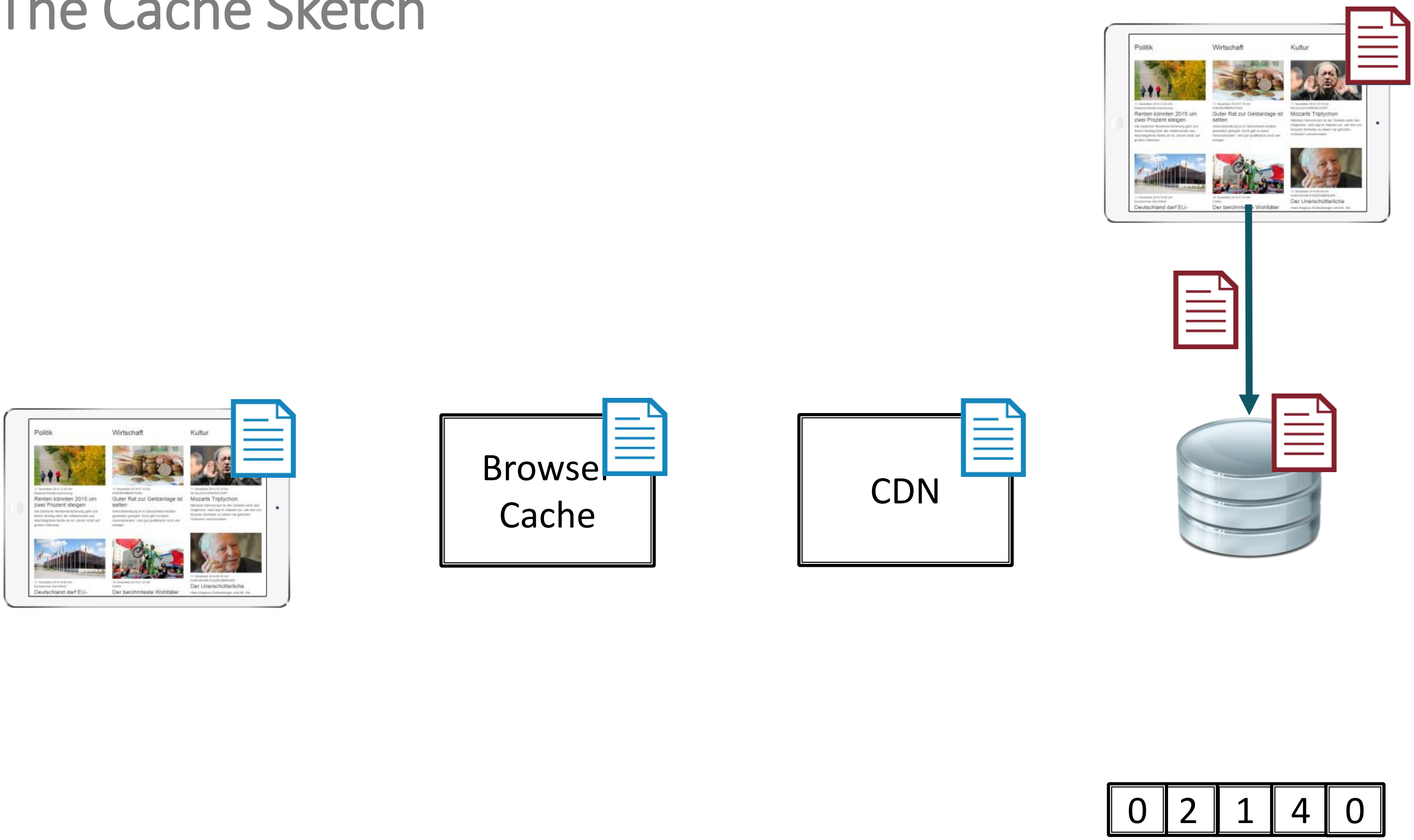
Consistent Web Caching

The Cache Sketch



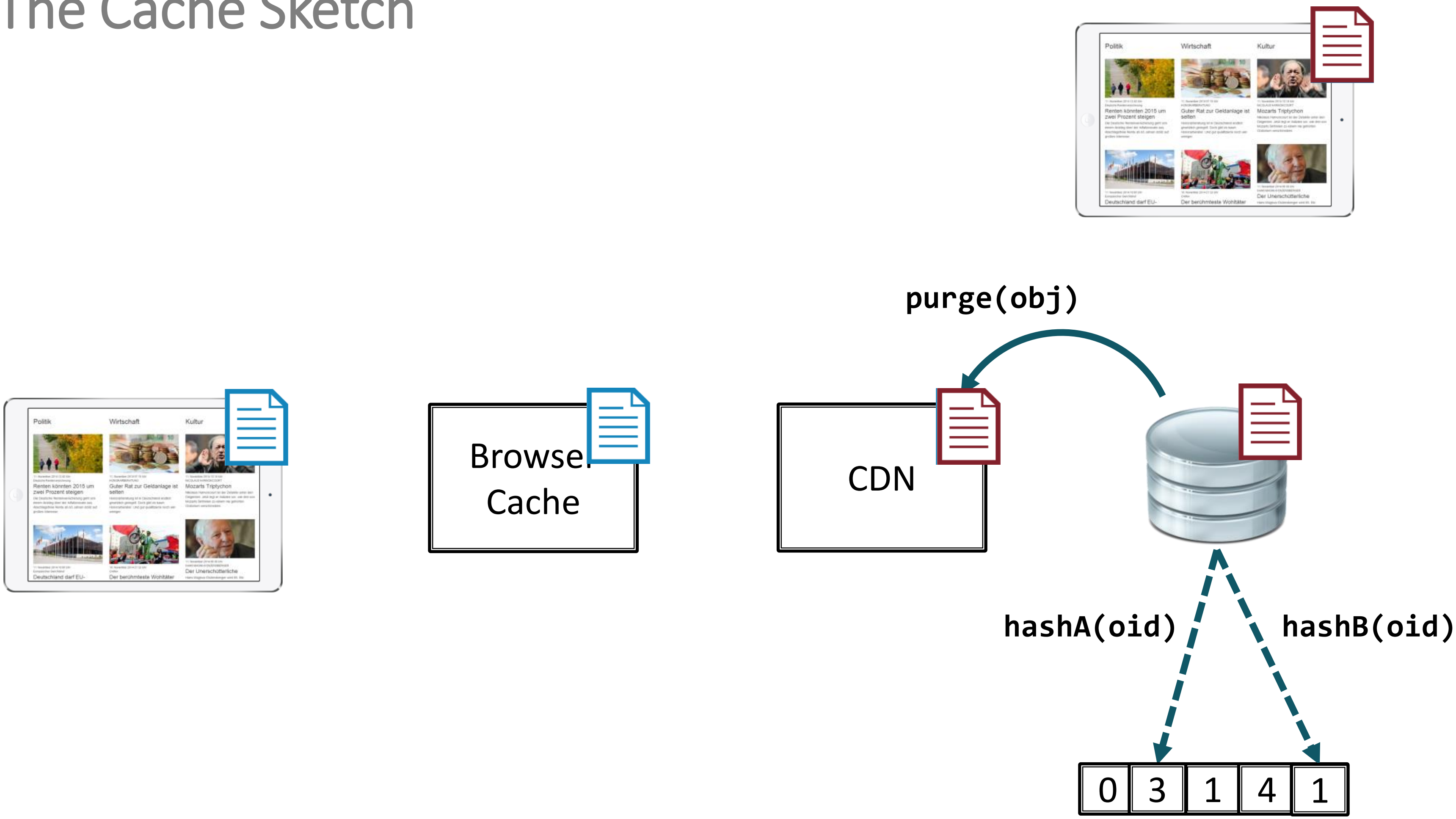
Consistent Web Caching

The Cache Sketch



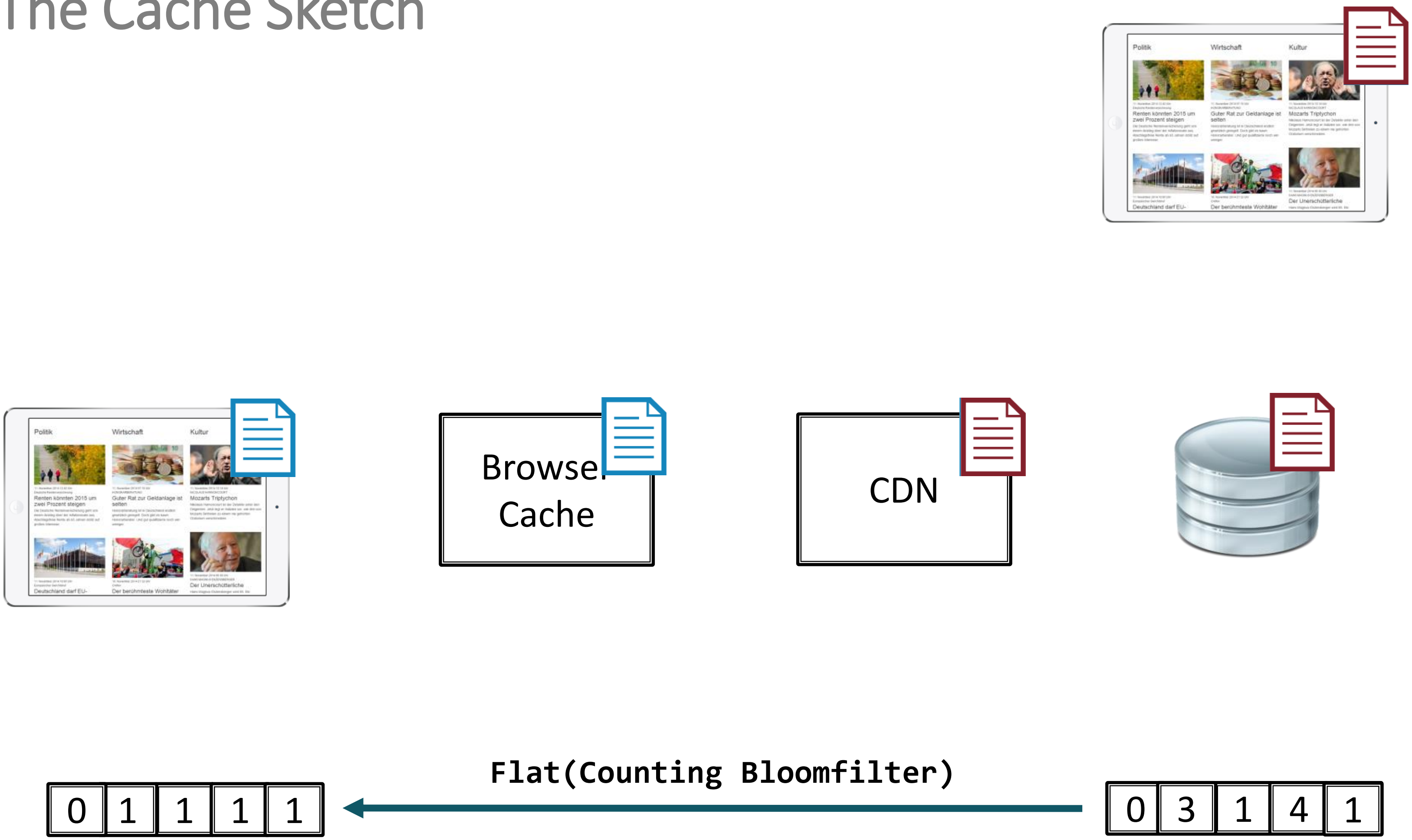
Consistent Web Caching

The Cache Sketch



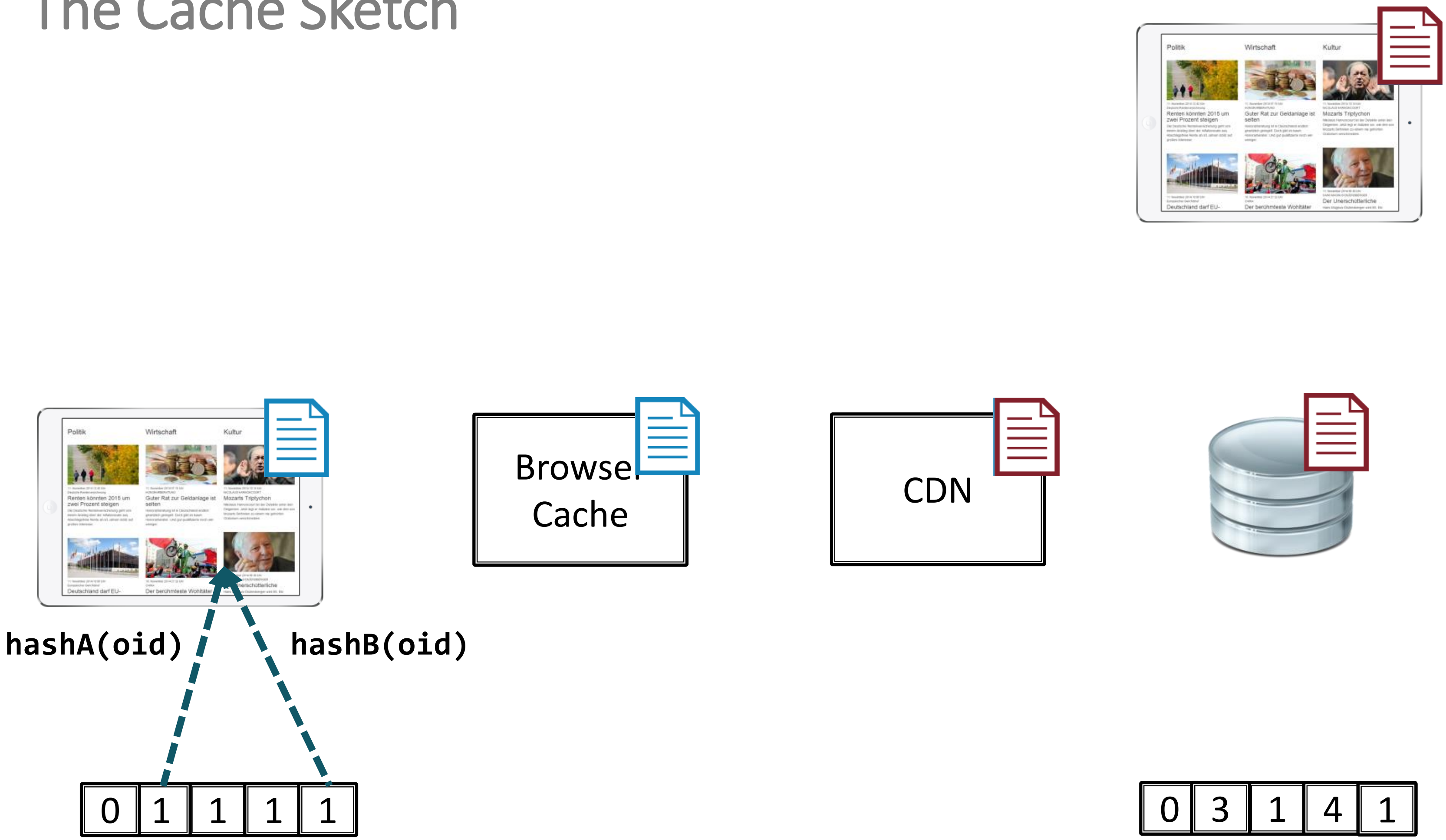
Consistent Web Caching

The Cache Sketch



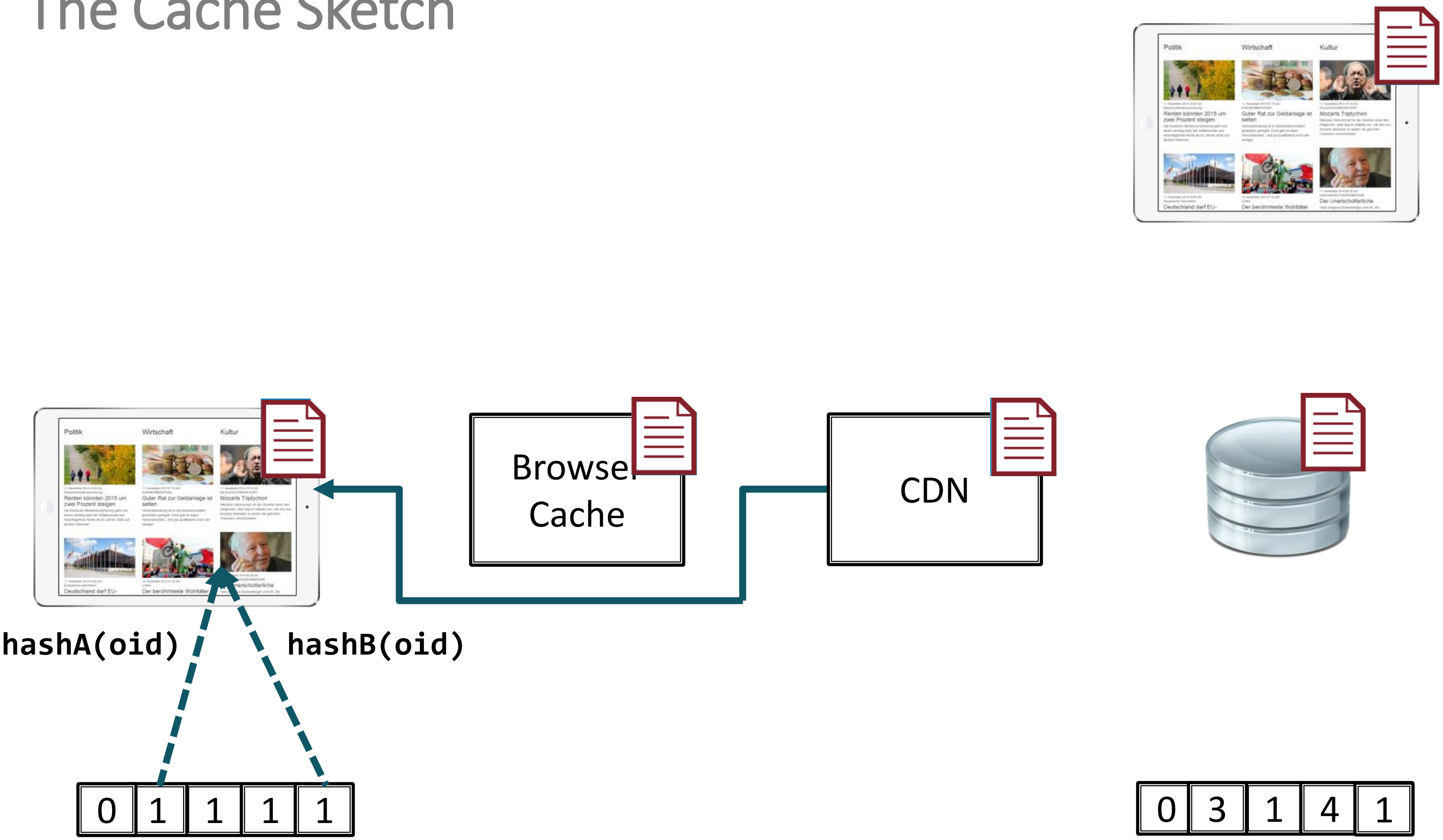
Consistent Web Caching

The Cache Sketch



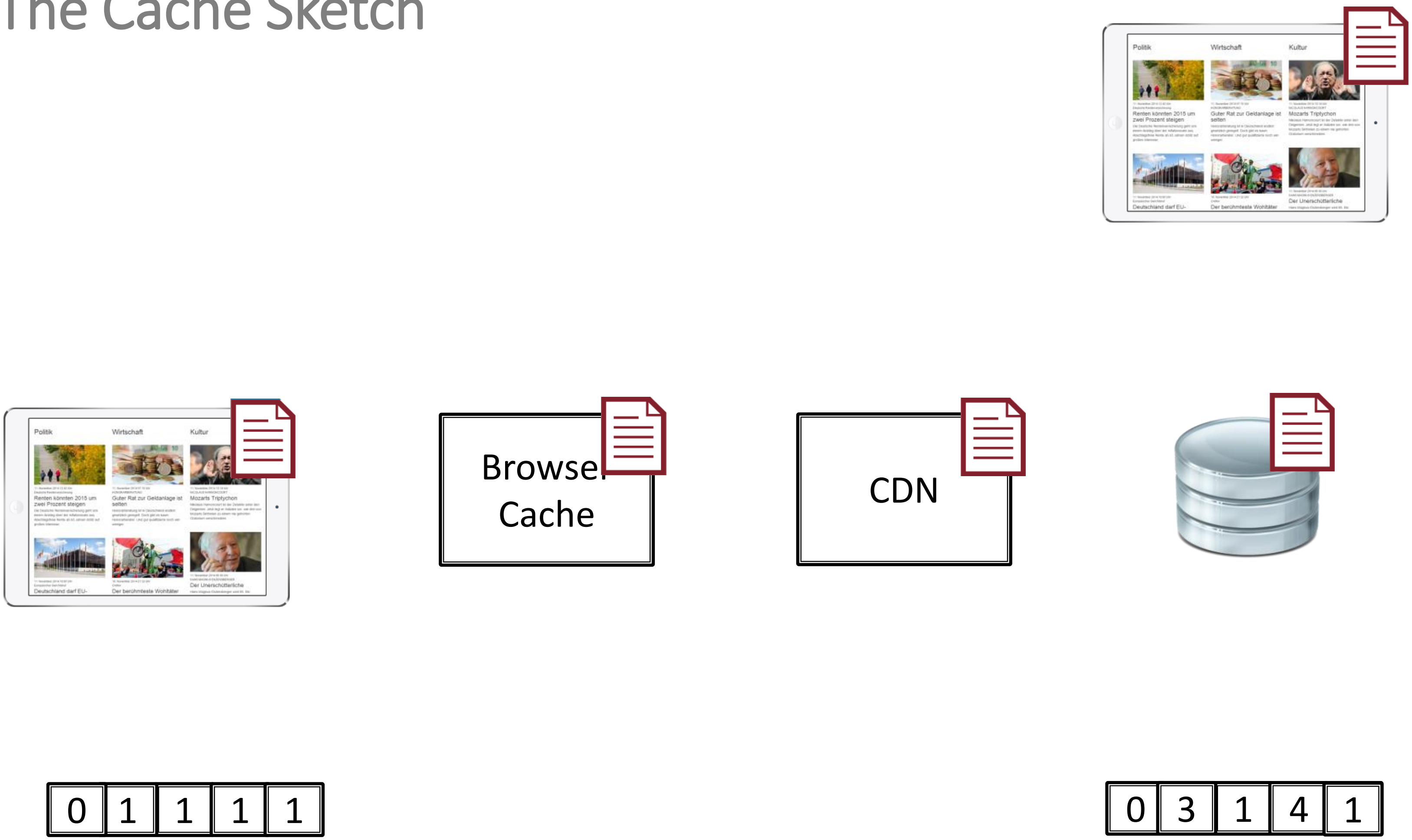
Consistent Web Caching

The Cache Sketch



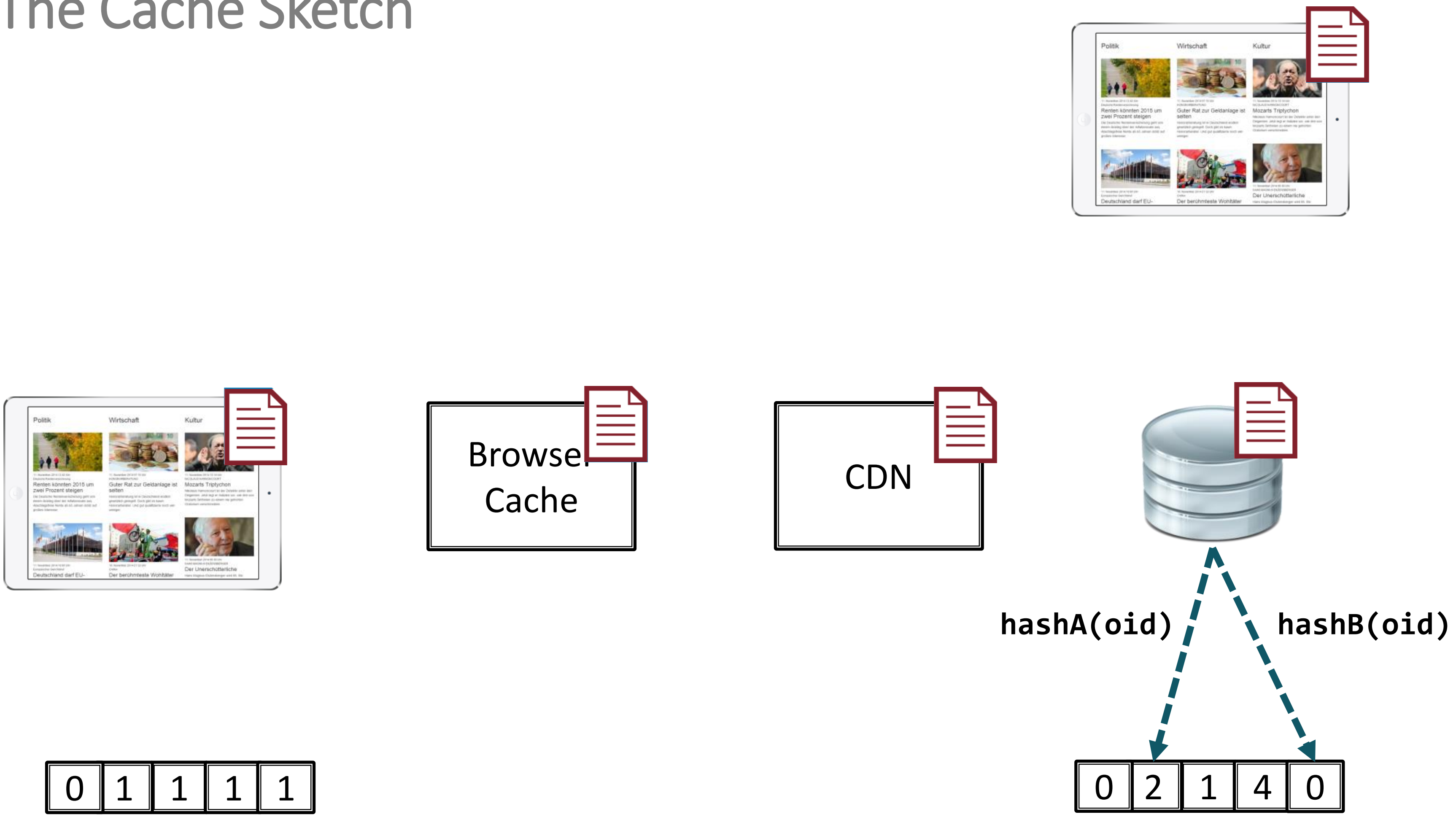
Consistent Web Caching

The Cache Sketch



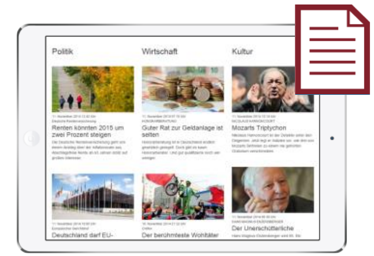
Consistent Web Caching

The Cache Sketch

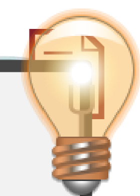


Consistent Web Caching

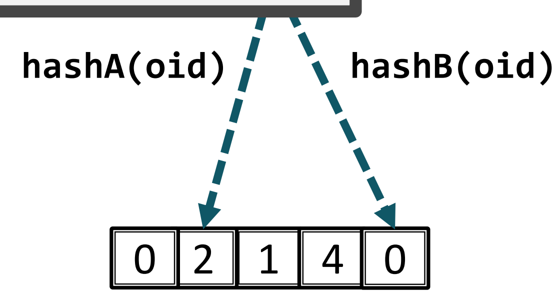
The Cache Sketch



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



0	1	1	1	1



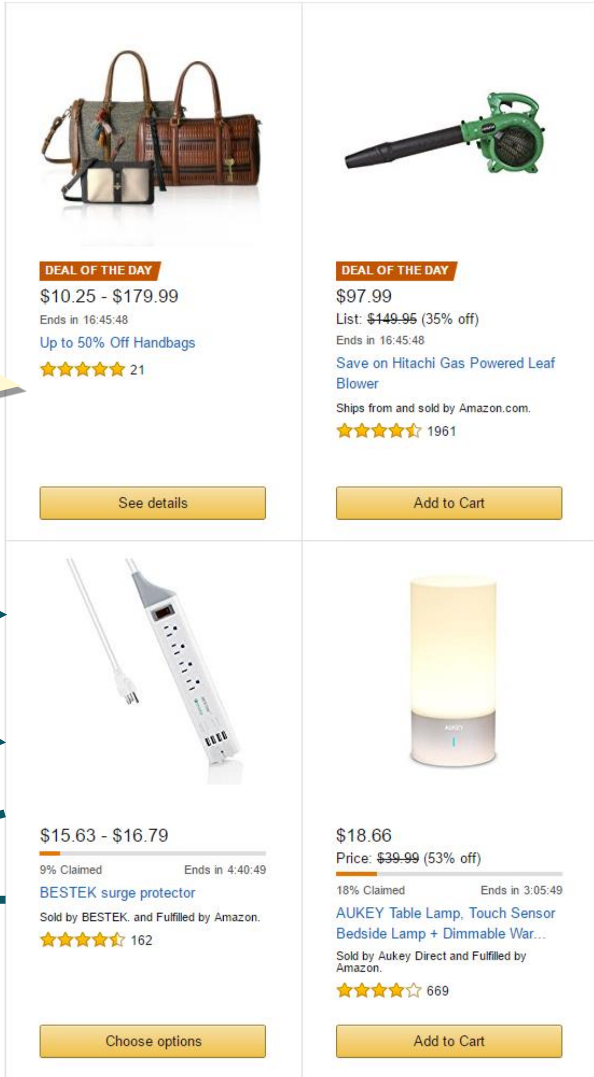
RESEARCH

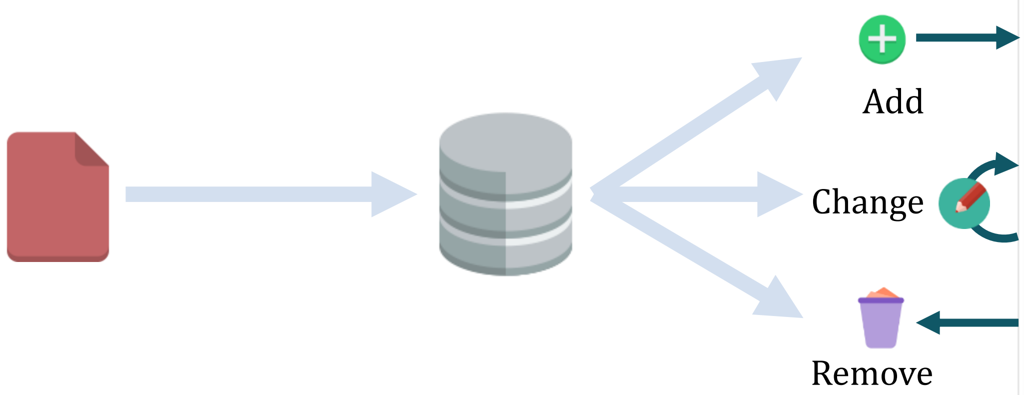
How to Invalidate DB Query Results?

InvaliDB

Invalidating DB Queries

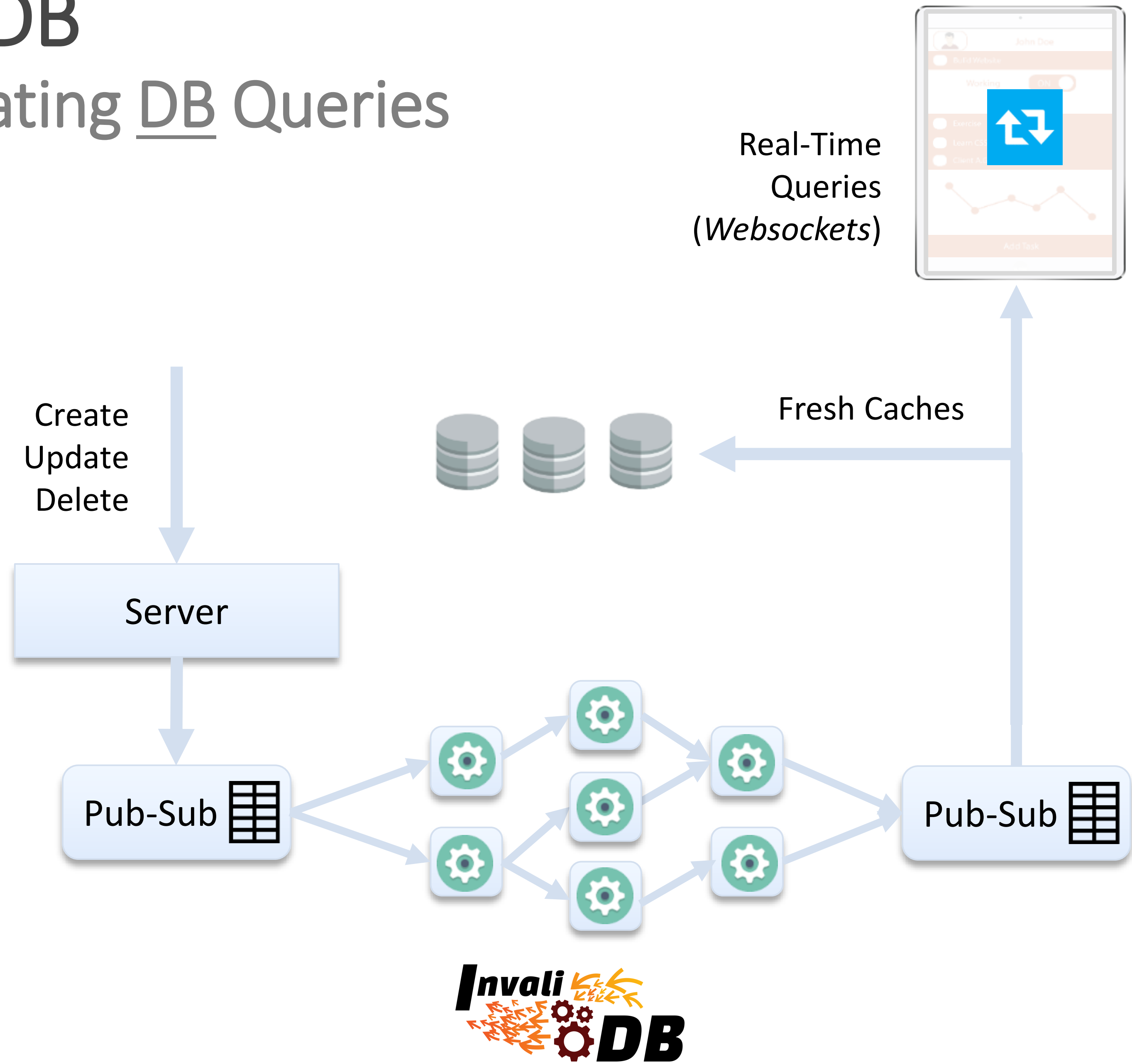
How to **detect changes to query results**:
„Give me the most popular products that are in stock.“





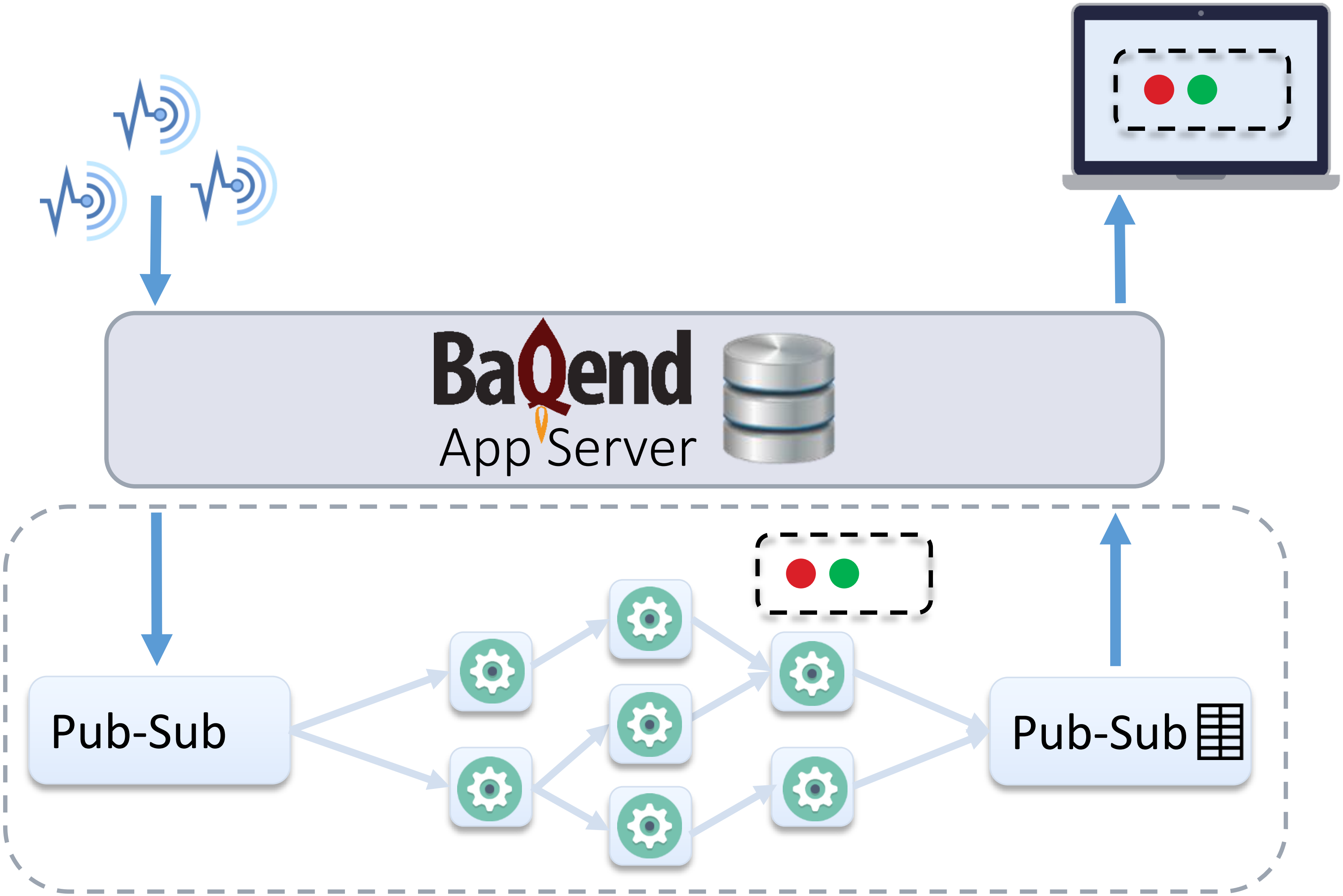
InvaliDB

Invalidating DB Queries



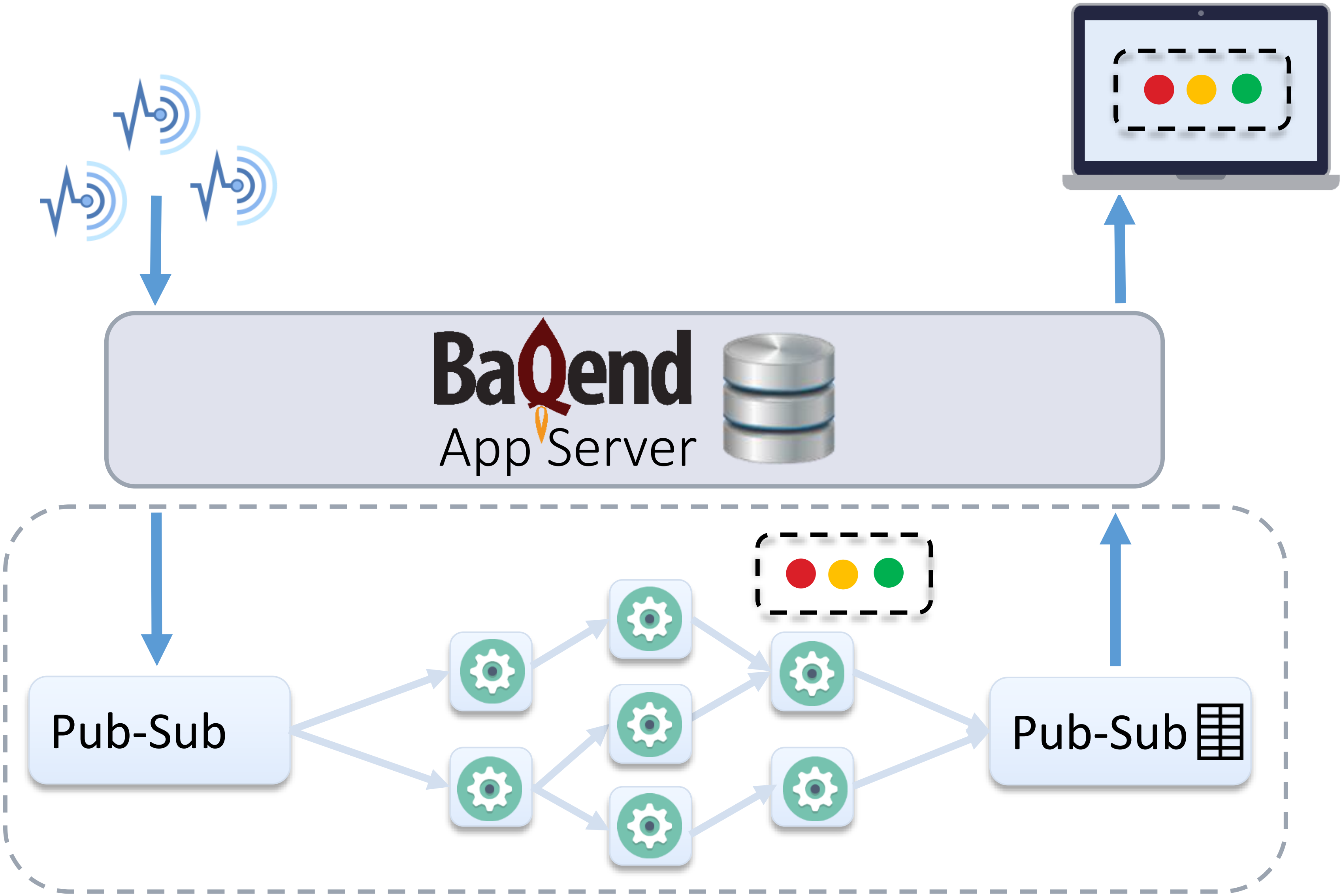
Baqend Real-Time Queries

Realtime Decoupled



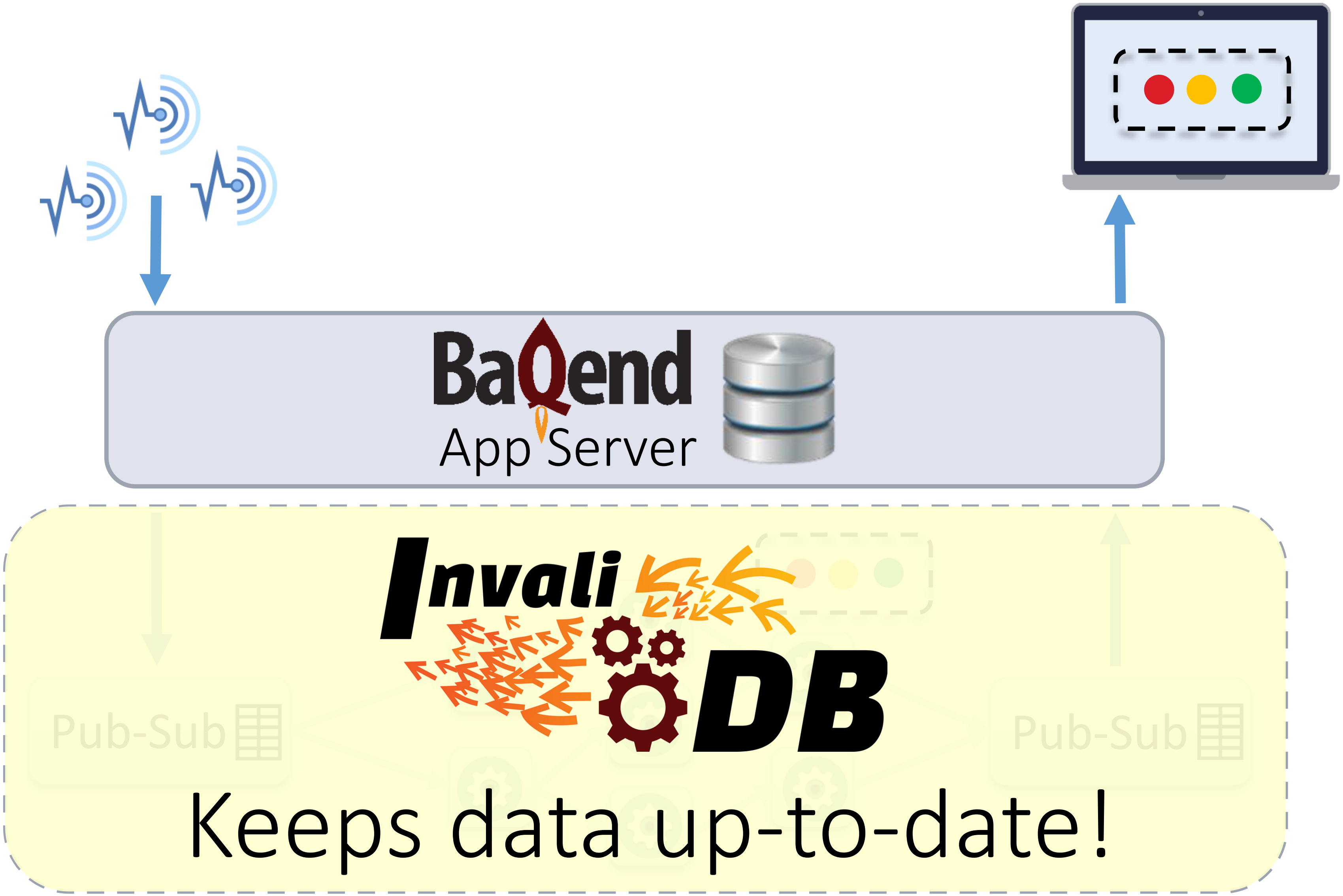
Baqend Real-Time Queries

Realtime Decoupled



Baqend Real-Time Queries

Realtime Decoupled



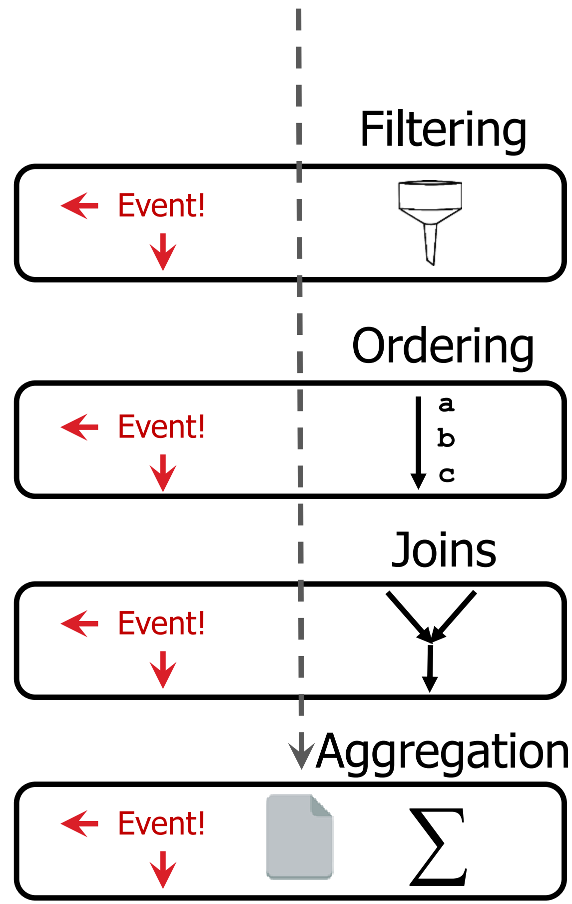
Keeps data up-to-date!

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



Baqend Real-Time Queries
Filter Queries: Distributed Query Matching

Two-dimensional partitioning:

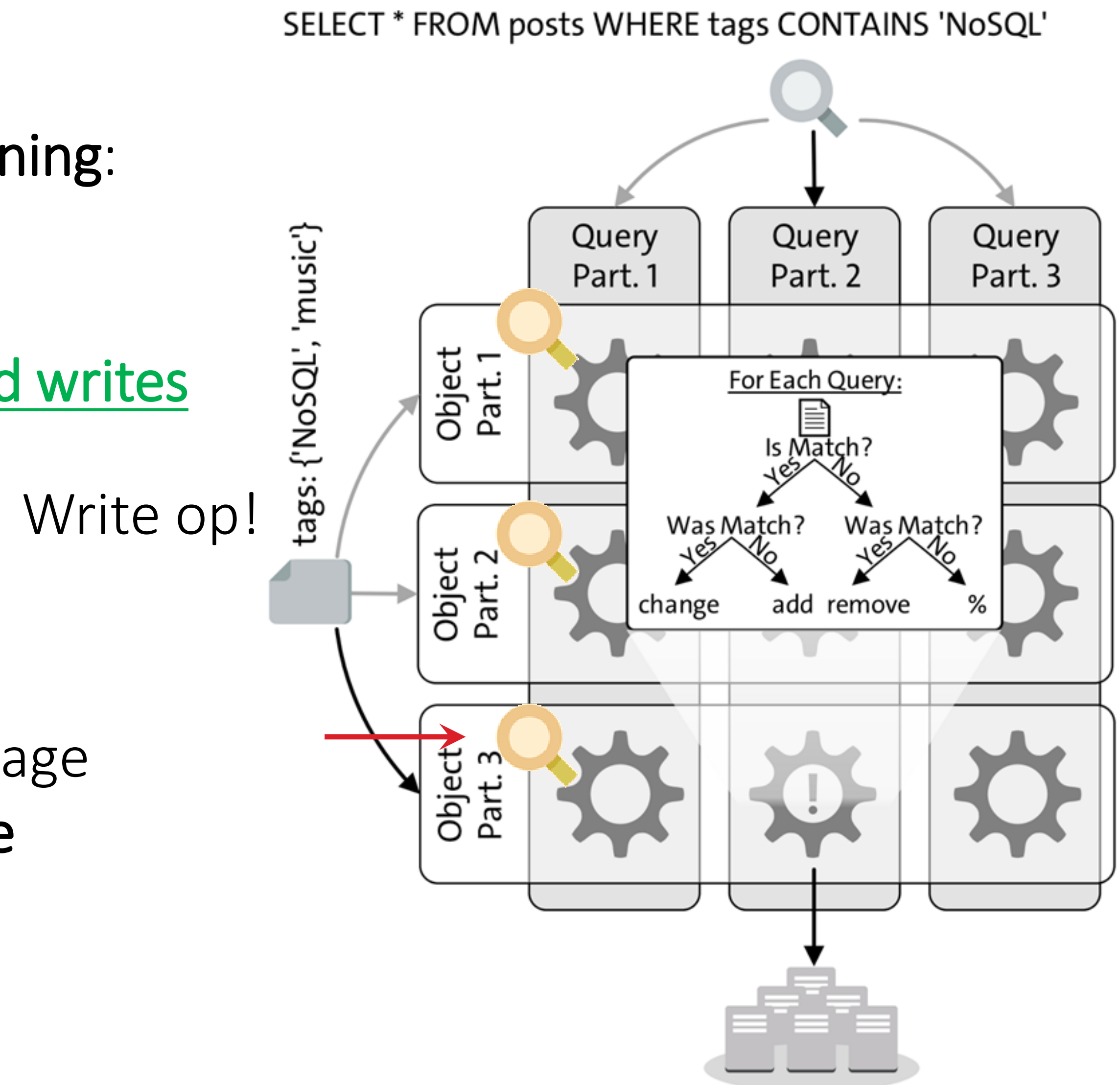
- *by Query*
- *by Object*

→ scales with queries <u>and writes</u>

Implementation:

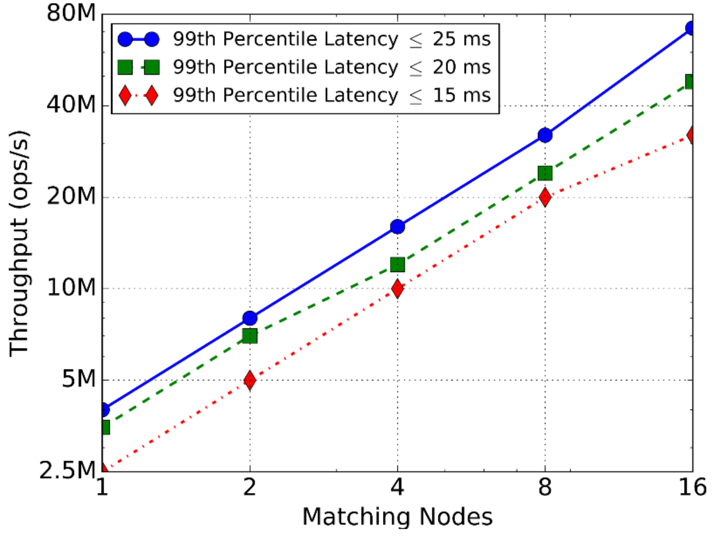
- Apache Storm
- Topology in Java
- MongoDB query language
- **Pluggable query engine**

→ legacy-compatible

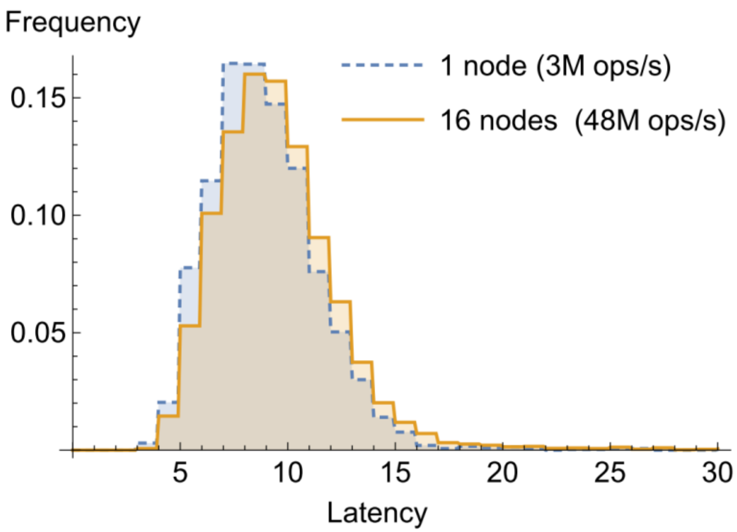


Baqend Real-Time Queries
Low Latency + Linear Scalability

Linear Scalability



Stable Latency Distribution



Quaestor: Query Web Caching for Database-as-a-Service Providers
VLDB '17

Programming Real-Time Queries

JavaScript API

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

Static Query

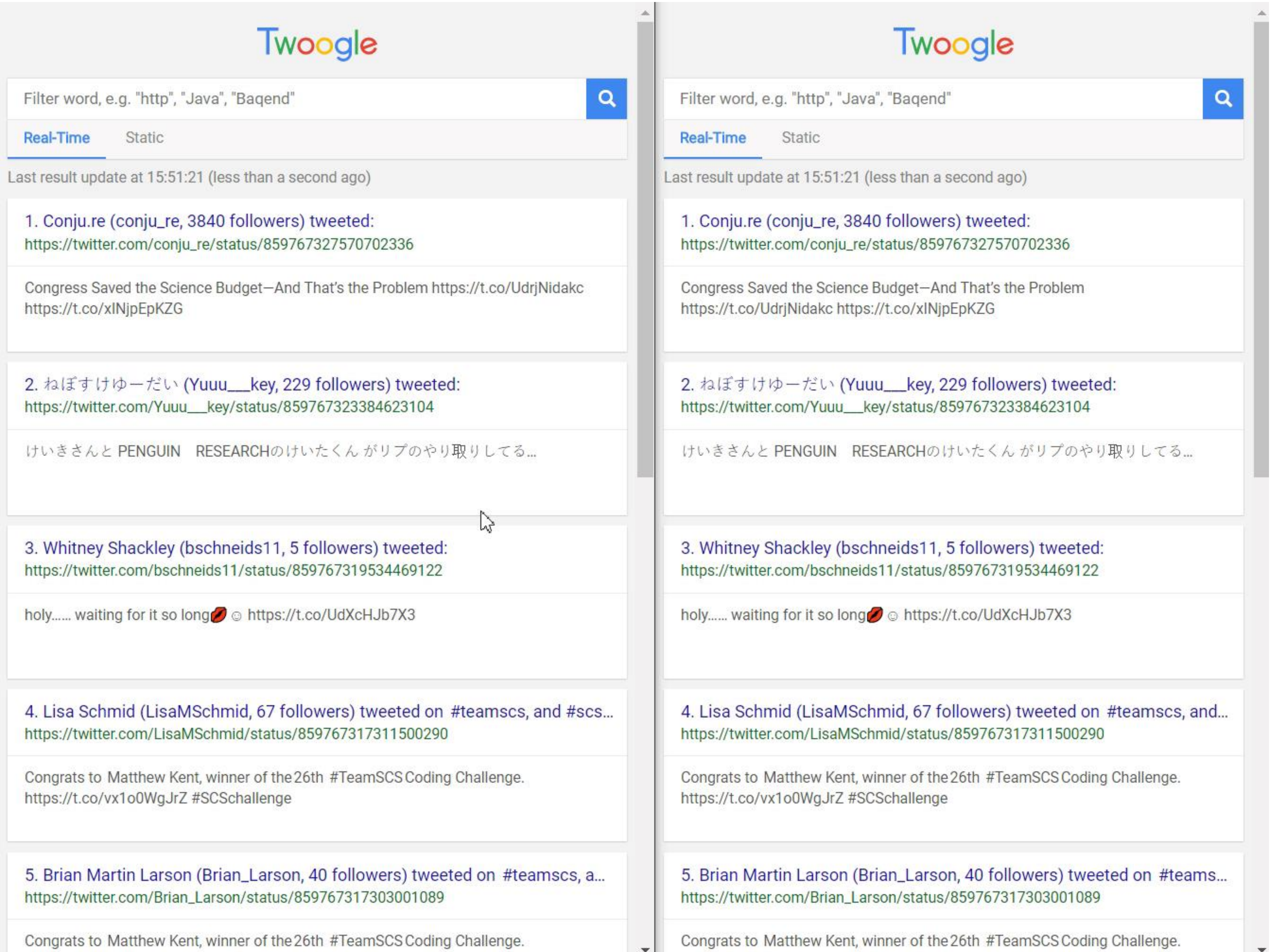
```
query.resultList(result => ...);
```

Google

Real-Time Query

```
query.resultStream(result => ...);
```

Twoogle



Twoogle
Filter word, e.g. "http", "Java", "Baqend"
Real-Time
Static
Last result update at 15:51:21 (less than a second ago)
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/xlNjpEpKZG
2. ねぼすけゆーだい (Yuuu__key, 229 followers) tweeted:
https://twitter.com/Yuuu__key/status/859767323384623104
けいきさんと PENGUIN　RESEARCHのけいたくん がリプのやり取りしてる...
3. Whitney Shackley (bschneids11, 5 followers) tweeted:
https://twitter.com/bschneids11/status/859767319534469122
holy...... waiting for it so long💋☺ 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 #TeamSCS Coding Challenge. https://t.co/vx1o0WgJrZ #SCSchallenge
5. Brian Martin Larson (Brian_Larson, 40 followers) tweeted on #teamscs, a...
https://twitter.com/Brian_Larson/status/859767317303001089
Congrats to Matthew Kent, winner of the 26th #TeamSCS Coding Challenge.
Twoogle
Filter word, e.g. "http", "Java", "Baqend"
Real-Time
Static
Last result update at 15:51:21 (less than a second ago)
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/xlNjpEpKZG
2. ねぼすけゆーだい (Yuuu__key, 229 followers) tweeted:
https://twitter.com/Yuuu__key/status/859767323384623104
けいきさんと PENGUIN　RESEARCHのけいたくん がリプのやり取りしてる...
3. Whitney Shackley (bschneids11, 5 followers) tweeted:
https://twitter.com/bschneids11/status/859767319534469122
holy...... waiting for it so long💋☺ https://t.co/UdXcHJb7X3
4. Lisa Schmid (LisaMSchmid, 67 followers) tweeted on #teamscs, and...
https://twitter.com/LisaMSchmid/status/859767317311500290
Congrats to Matthew Kent, winner of the 26th #TeamSCS Coding Challenge. https://t.co/vx1o0WgJrZ #SCSchallenge
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 #TeamSCS Coding Challenge.

Baqend
Try It Out!

Platform

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

Speed Kit

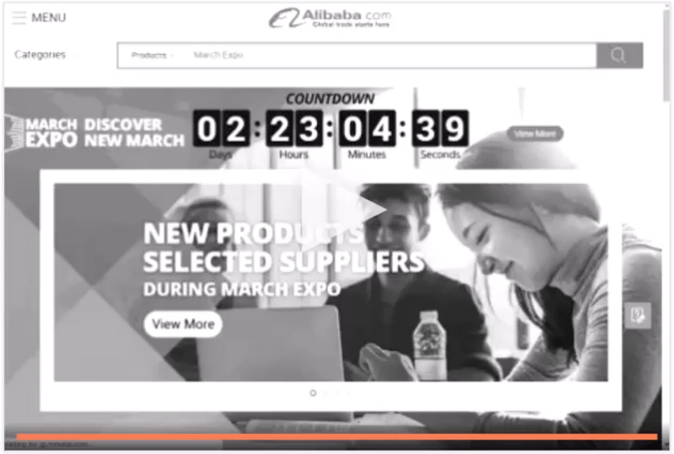
- Turns Existing Sites into **PWAs**
- **50-300% Faster** Loads
- **Offline** Mode

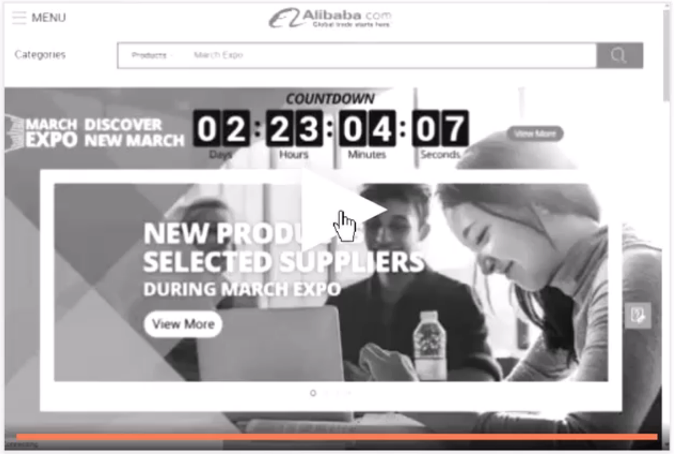
Speed Kit
Accelerate Your Website!

https://test.speed-kit.com/

https://www.alibaba.com/

Your Website		With Speed Kit
3514ms	2.3x Faster	1543ms





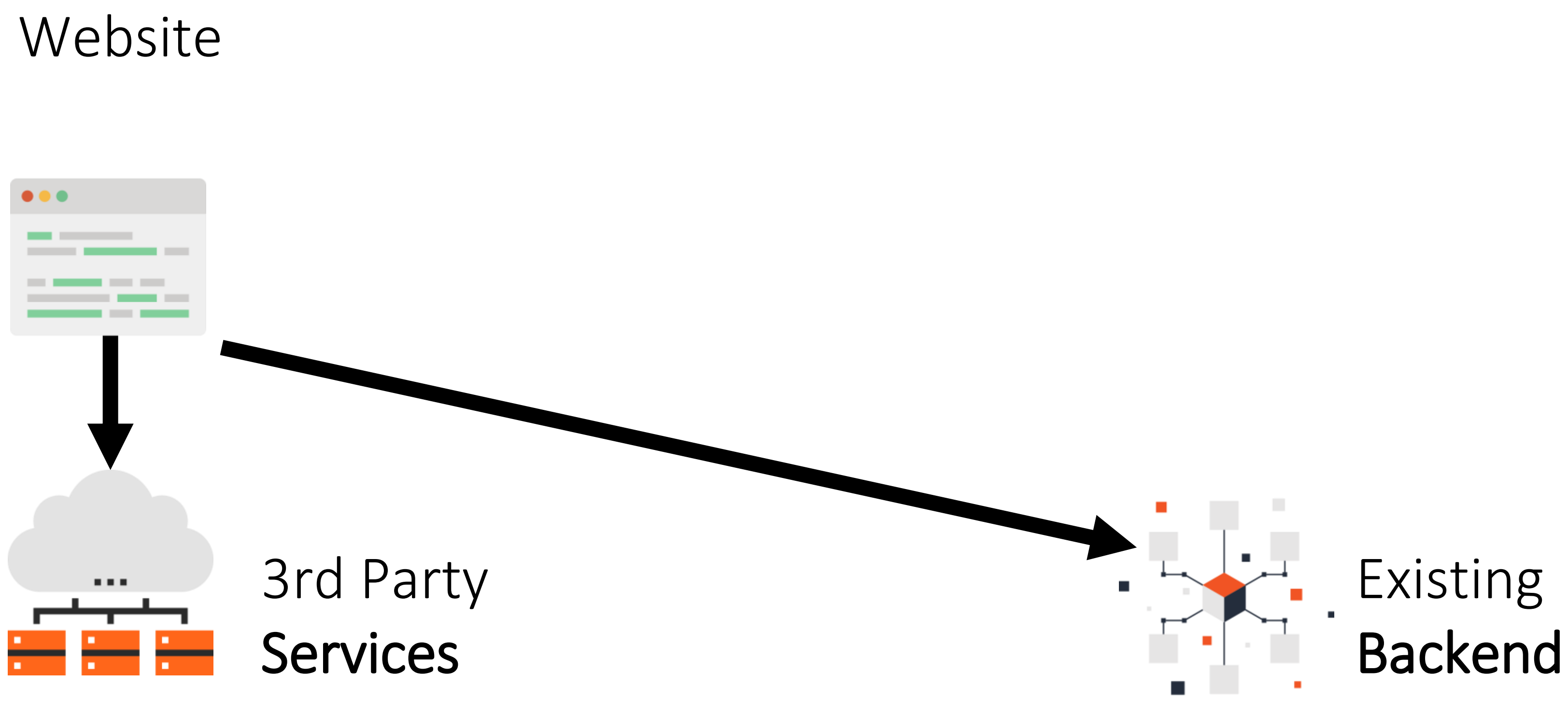
With Speed Kit 1.5s

Your Website 3.5s

Excellent — Poor

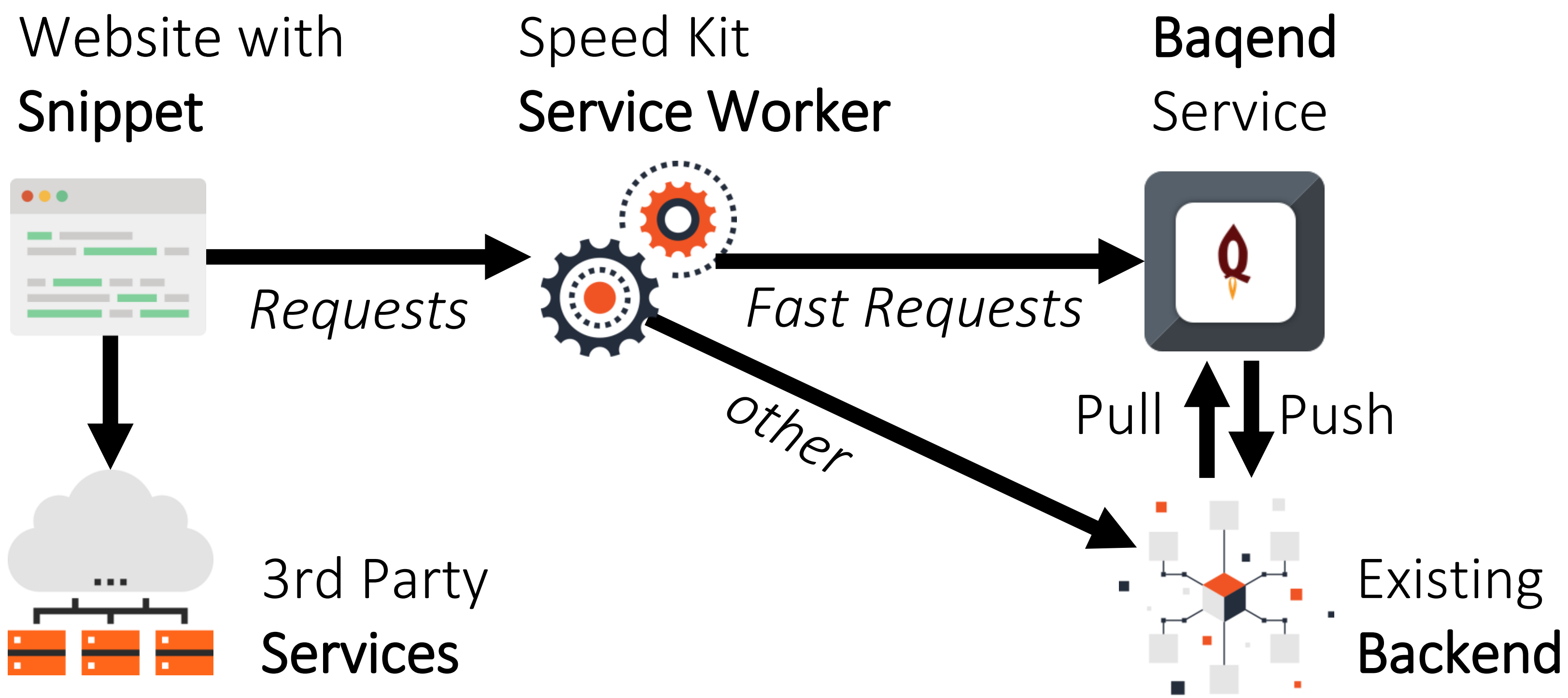
Show Details

Speed Kit
Baqend Caching for Legacy Websites



Speed Kit

Baqend Caching for Legacy Websites



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)

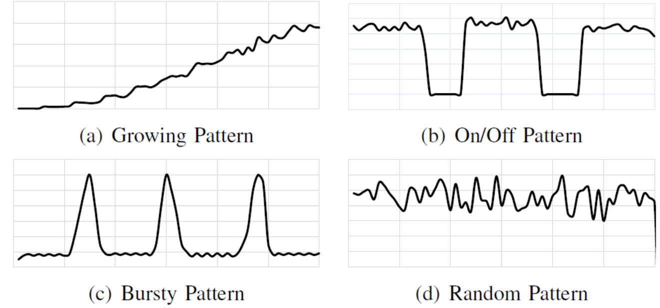
Id-Lists	Object-Lists
$\{id_1, id_2, id_3\}$	$\{\,\{id\!: 1, val\!: 'a'\}, \{id\!: 2, val\!: 'b'\}, \{id\!: 3, val\!: 'c'\}\}$
Less Invalidations	Less Round-Trips

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*):

- Mesure & 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 & evalute model

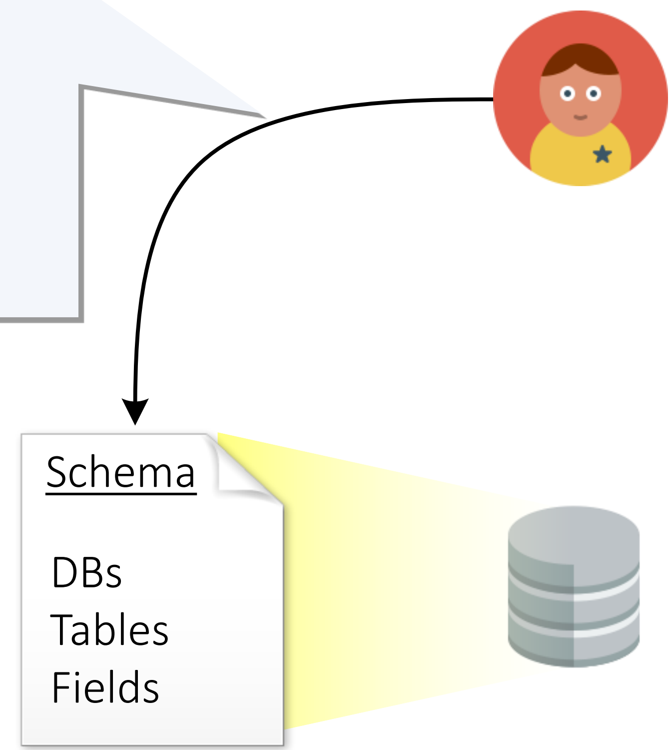
„**Reactive**“:

- Use **Reinforcement learning** to automatically explore decisions
- Rewards usually noisy, delayed or hidden (e.g. staleness)

Polyglot Persistence Mediator

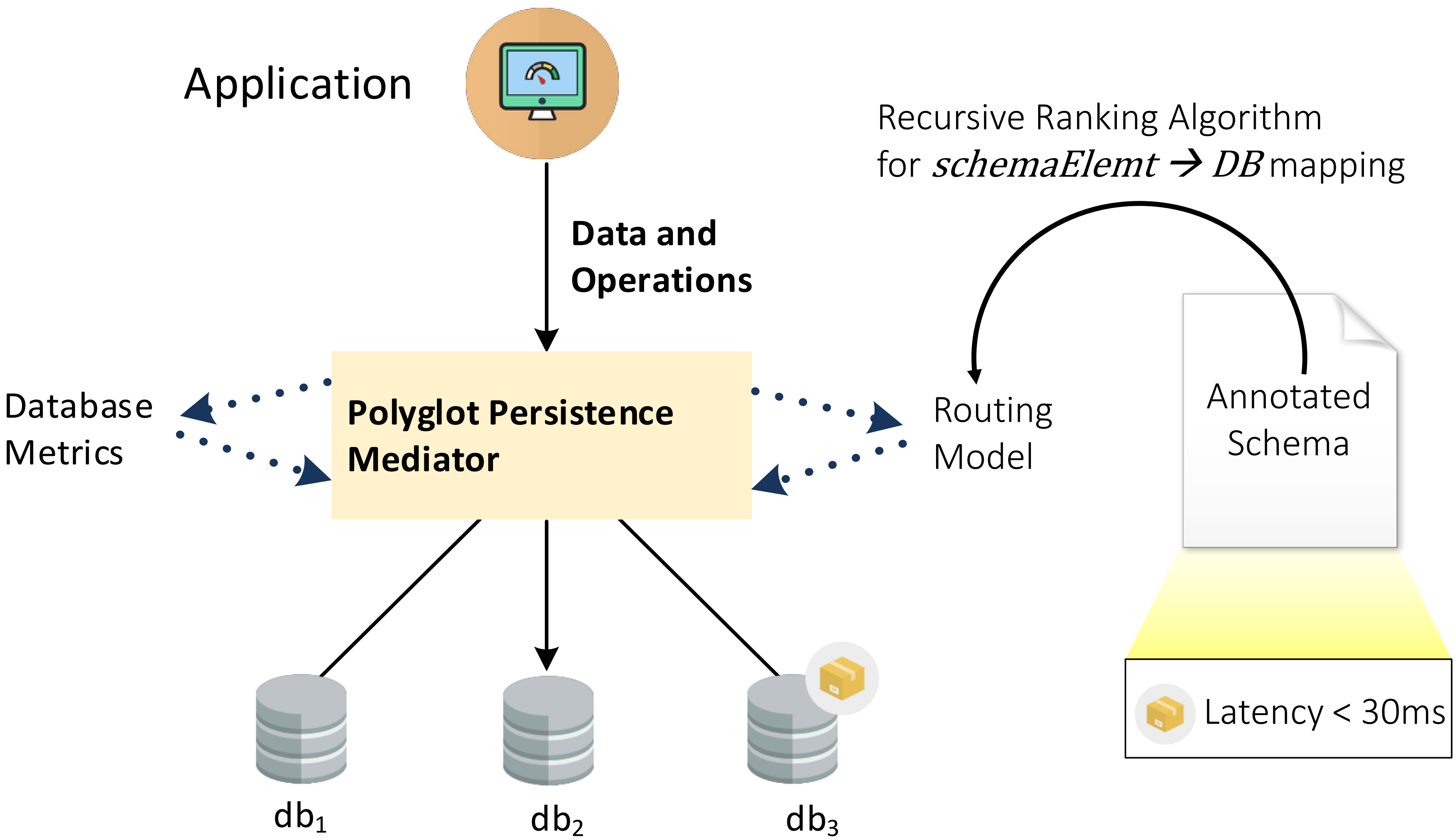
Schemas can be annotated with requirements/SLAs

- 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“ datbase 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

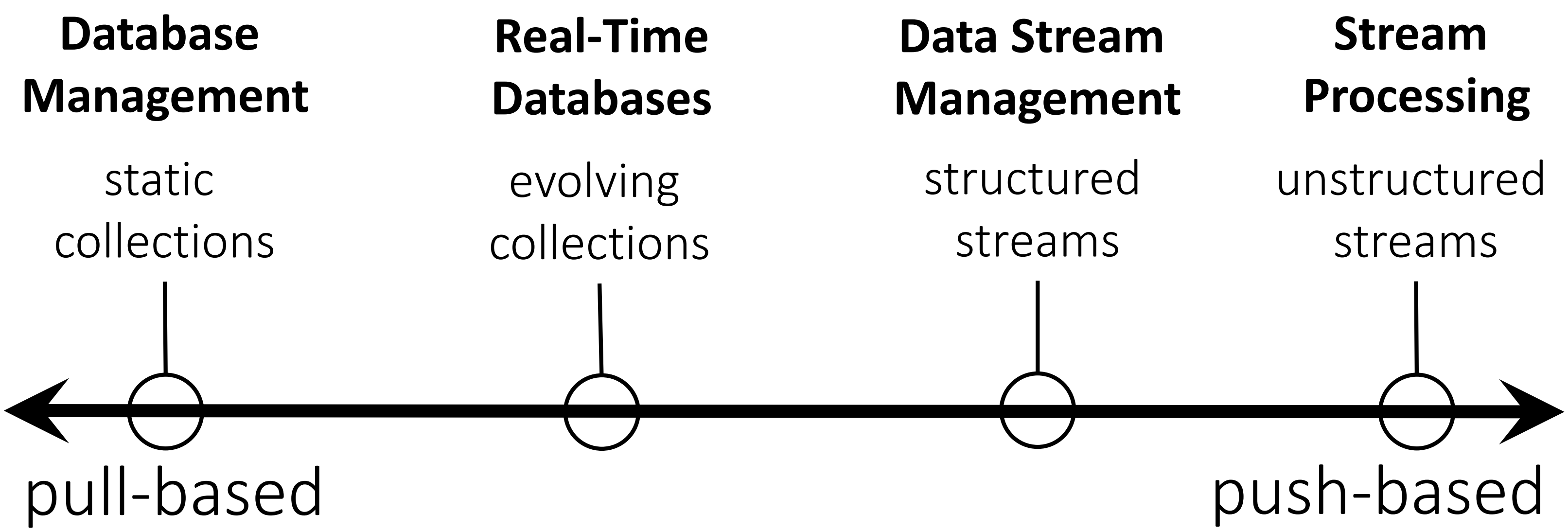
Real-Time Data Management

pull-based push-based

<table>
<tr><th></th><th>Database Management</th><th>Real-Time Databases</th><th>Data Stream Management</th><th>Stream Processing</th></tr>
<tr><td>Data</td><td colspan="2">persistent collections</td><td colspan="2">persistent/ephemeral streams</td></tr>
<tr><td>Processing</td><td>one-time</td><td>one-time + continuous</td><td colspan="2">continuous</td></tr>
<tr><td>Access</td><td>random</td><td>random + sequential</td><td colspan="2">sequential</td></tr>
<tr><td>Schema</td><td colspan="3">structured</td><td>structured, unstructured</td></tr>
</table>

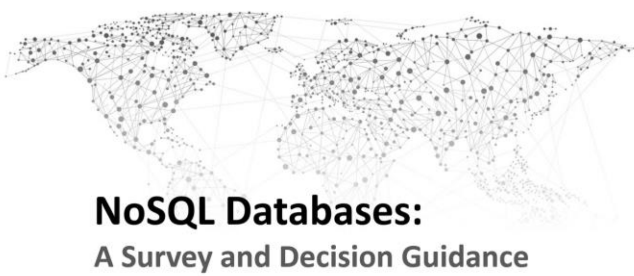
Summary

Real-Time Data Management



NoSQL Databases: a Survey and Decision Guidance

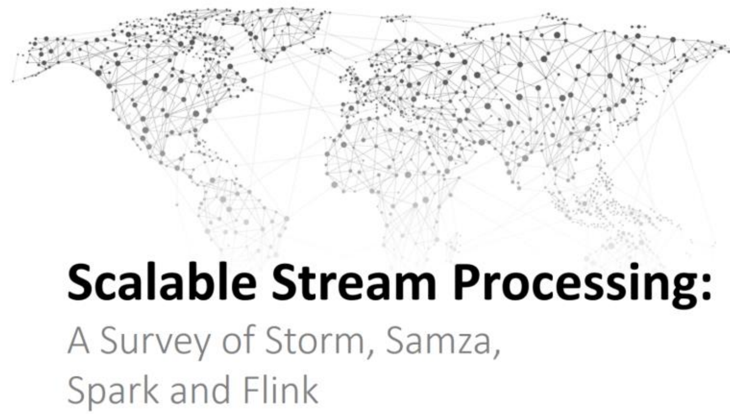
Together with our colleagues at the University of Hamburg, we—that is Felix Gessert, Wolfram Wingerath, Steffen Friedrich and Norbert Ritter—presented an overview over the NoSQL landscape at SummerSOC'16 last month. Here is the written gist. We give our best to convey the condensed NoSQL knowledge we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

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

Read them at blog.baqend.com!

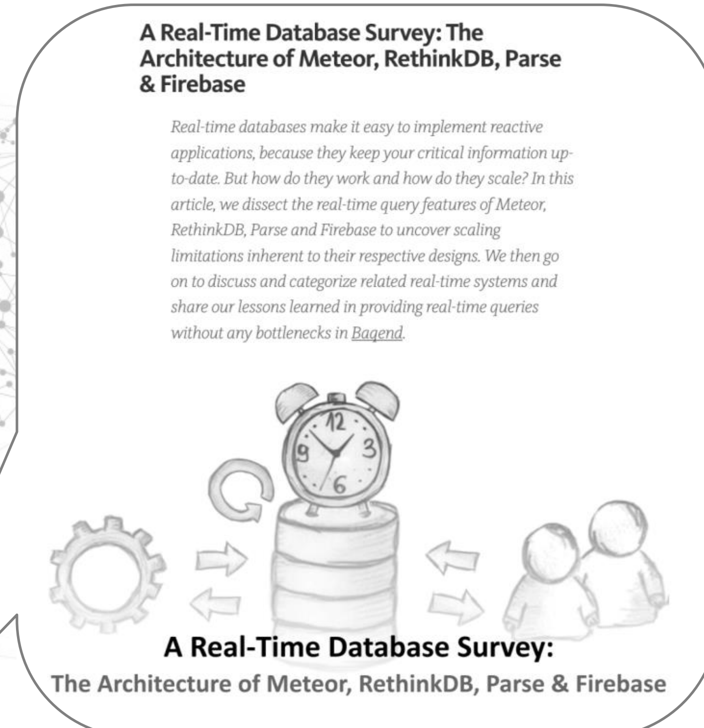
Our Related Publications

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*
  Baqend Tech Blog (2017): https://medium.com/p/822ff87d2f87


A Real-Time Database Survey: The Architecture of Meteor, RethinkDB, Parse & Firebase

Real-time databases make it easy to implement reactive applications, because they keep your critical information up-to-date. But how do they work and how do they scale? In this article, we dissect the real-time query features of Meteor, RethinkDB, Parse and Firebase to uncover scaling limitations inherent to their respective designs. We then go on to discuss and categorize related real-time systems and share our lessons learned in providing real-time queries without any bottlenecks in Baqend.

Learn more at blog.baqend.com!

Thank you

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

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

@baqendcom