NoSQL & Real-Time Data Management

In Research & Practice – Part 1

Wolfram Wingerath, Felix Gessert, Norbert Ritter

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

March 5, BTW 2019, Rostock

Universität Hamburg

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
- …

UHH Universität Hamburg

BaQend
www.baqend.com

Slides: slides.baqend.com

Articles: blog.baqend.com

Outline

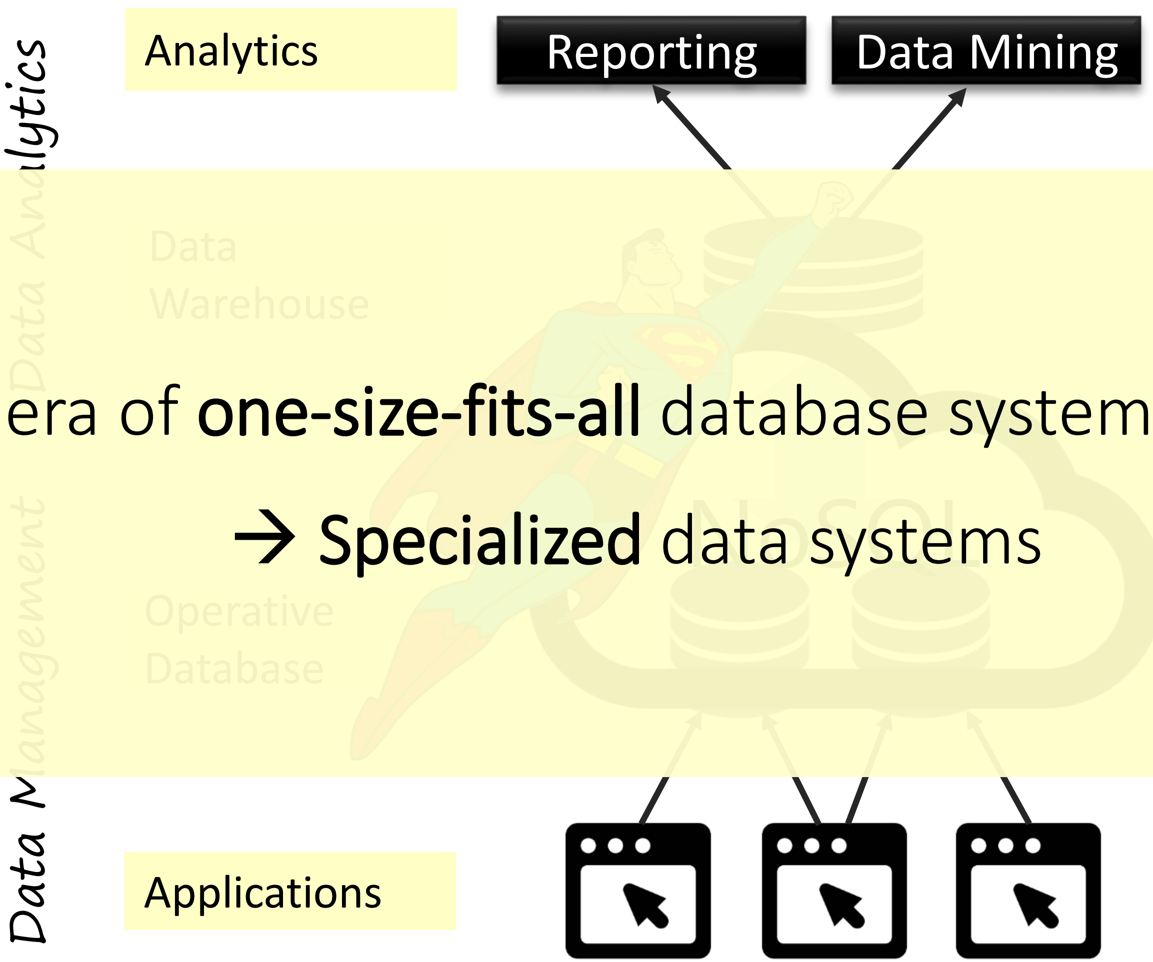
- NoSQL Foundations and Motivation
- The NoSQL Toolbox: Common Techniques
- NoSQL Systems & Decision Guidance
- Scalable Real-Time Databases and Processing

- The Database Explosion
- NoSQL: Motivation and Origins
- The 4 Classes of NoSQL Databases:
  - Key-Value Stores
  - Wide-Column Stores
  - Document Stores
  - Graph Databases
- CAP Theorem

Introduction: What are NoSQL data stores?

Architecture

Typical Data Architecture:



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

→ **Specialized** data systems

The Database Explosion

Sweetspots

IBM DB2

RDBMS

General-purpose ACID transactions

Greenplum

Parallel DWH

Aggregations/OLAP for massive data amounts

VoltDB

NewSQL

High throughput relational OLTP

HBASE

Wide-Column Store

Long scans over structured data

mongoDB

Document Store

Deeply nested data models

riak

Key-Value Store

Large-scale session storage

Neo4j the graph database

Graph Database

Graph algorithms & queries

redis

In-Memory KV-Store

Counting & statistics

cassandra

Wide-Column Store

Massive user-generated content

The Database Explosion

Cloud-Database Sweetspots

Firebase

Realtime BaaS

Communication and collaboration

Amazon RDS

Managed RDBMS

General-purpose ACID transactions

Amazon ElastiCache

Managed Cache

Caching and transient storage

Azure Tables

Wide-Column Store

Very large tables

Amazon DynamoDB

Wide-Column Store

Massive user-generated content

Parse

Backend-as-a-Service

Small Websites and Apps

bonsai

Managed NoSQL

Full-Text Search

Google Cloud Storage

Object Store

Massive File Storage

Amazon Elastic MapReduce

Hadoop-as-a-Service

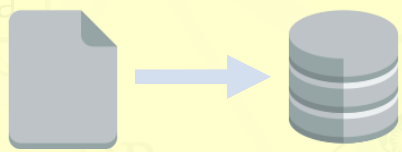
Big Data Analytics

How to choose a database system?

Many Potential Candidates

Question in this tutorial:

How to approach the requirements → database decision problem?



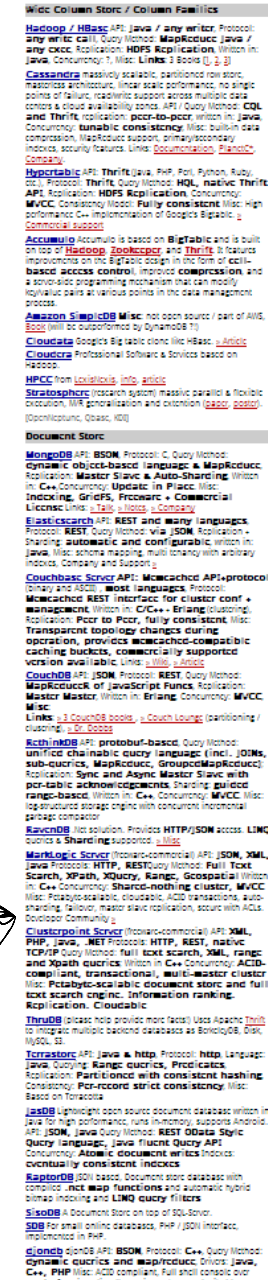
requirements

database

NoSQL Databases

- „NoSQL“ term coined in 2009
- Interpretation: „**N**ot **O**nly **SQL**“
- Typical properties:
  - Non-relational
  - Open-Source
  - Schema-less (*schema-free*)
  - Optimized for distribution (clusters)
  - Tunable consistency

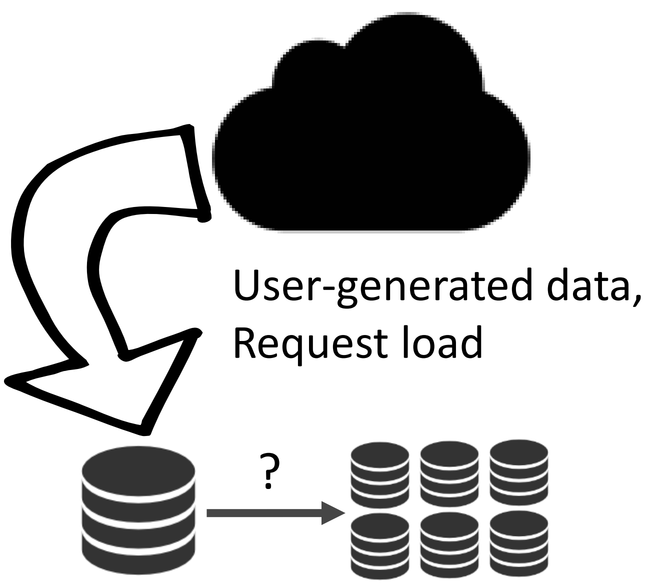
***NoSQL-Databases.org*:**
Current list has over 225
NoSQL systems

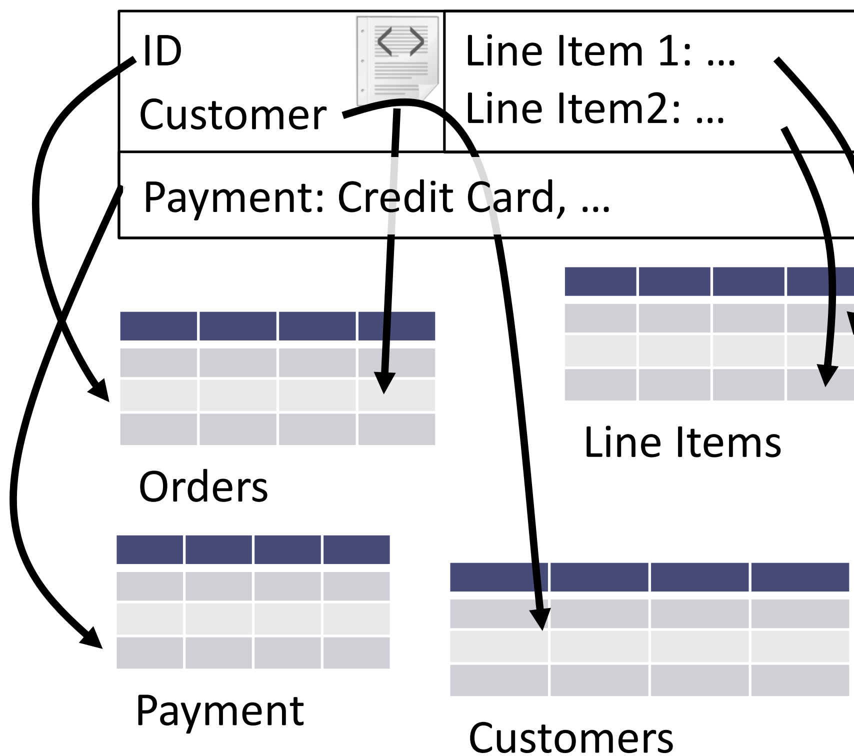


NoSQL Databases

- Two main motivations:

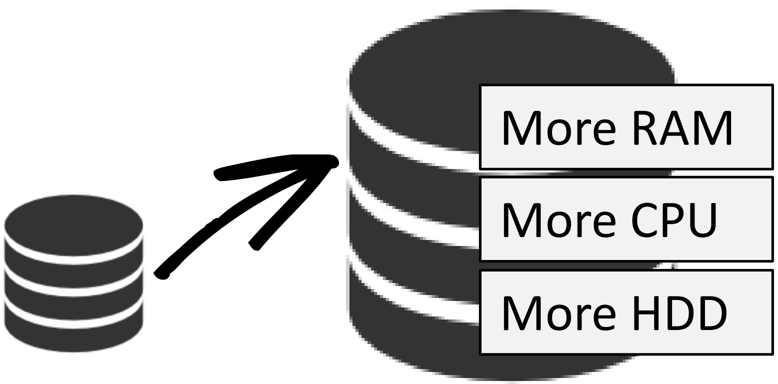
Scalability	Impedance Mismatch



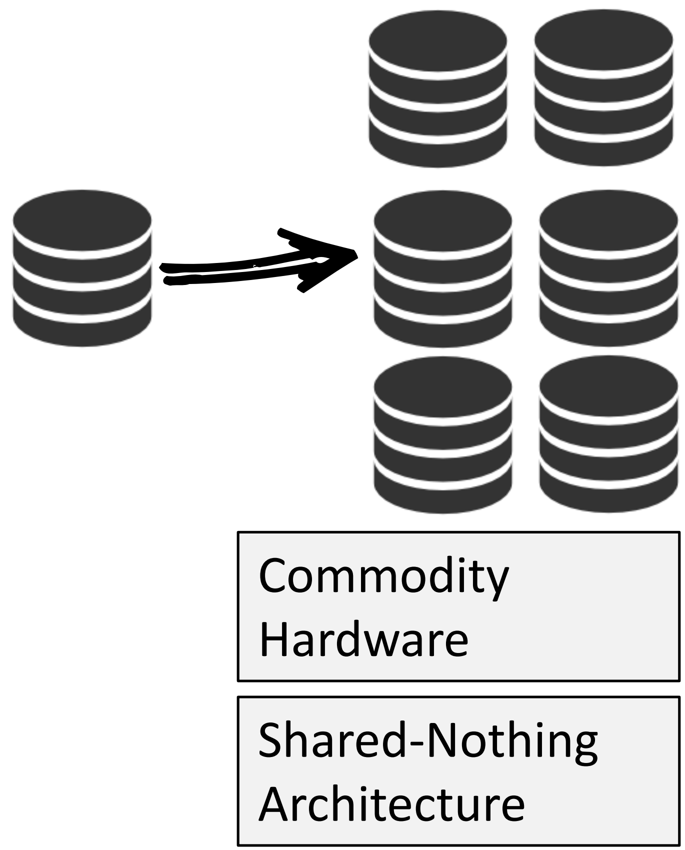


Scale-up vs Scale-out

Scale-Up (*vertical* scaling):

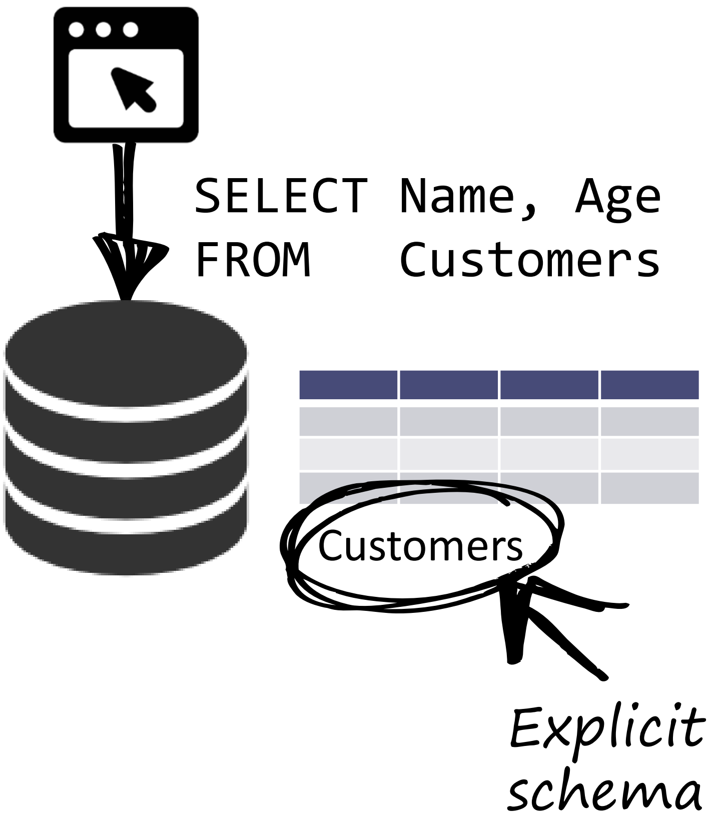


Scale-Out (*horizontal* scaling):

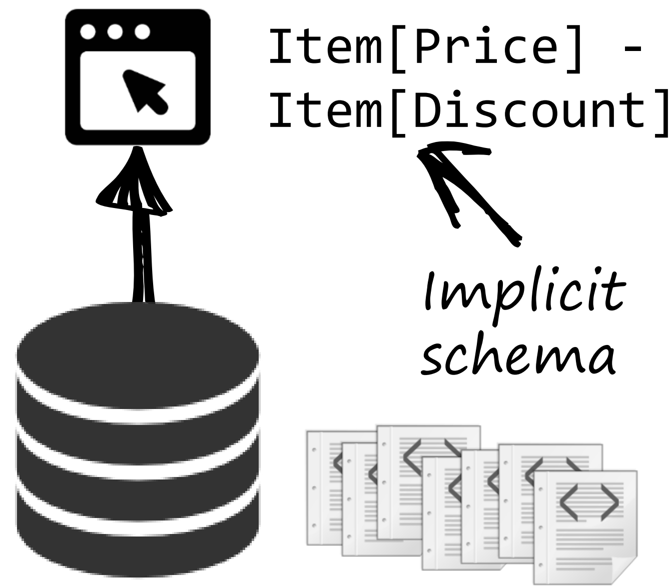


Schemafree Data Modeling

RDBMS:



NoSQL DB:



Big Data
The Analytic side of NoSQL

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

Sources

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

Analyst, Data Scientist, Software Developer

- Statistics, Cubes, Reports
- Recommender
- Classificators, Clustering
- Knowledge

NoSQL Paradigm Shift

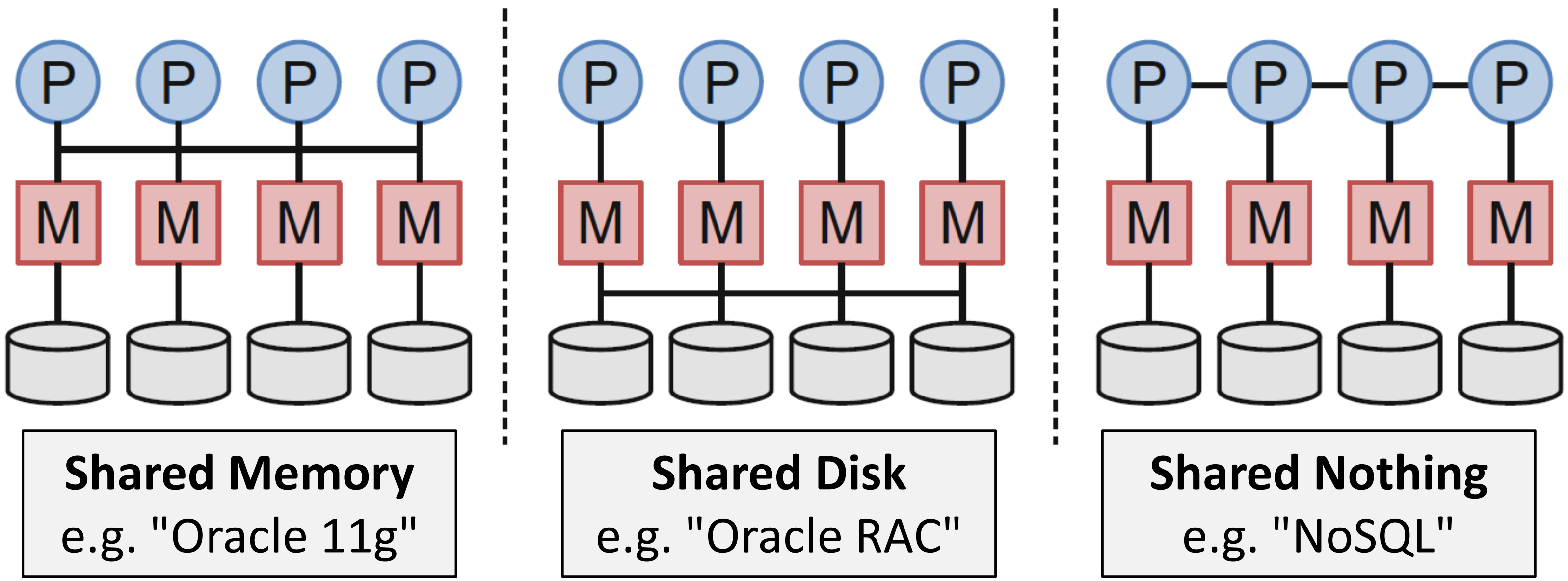
Open Source & Commodity Hardware

Commercial DBMS	Open-Source DBMS
Specialized DB hardware (Oracle Exadata, etc.)	Commodity hardware
Highly available network (Infiniband, Fabric Path, etc.)	Commodity network (Ethernet, etc.)
Highly Available Storage (SAN, RAID, etc.)	Commodity drives (standard HDDs, JBOD)

NoSQL Paradigm Shift

Shared Nothing Architectures

Shift towards higher distribution & less coordination:



NoSQL System Classification

- Two common criteria:

Data Model

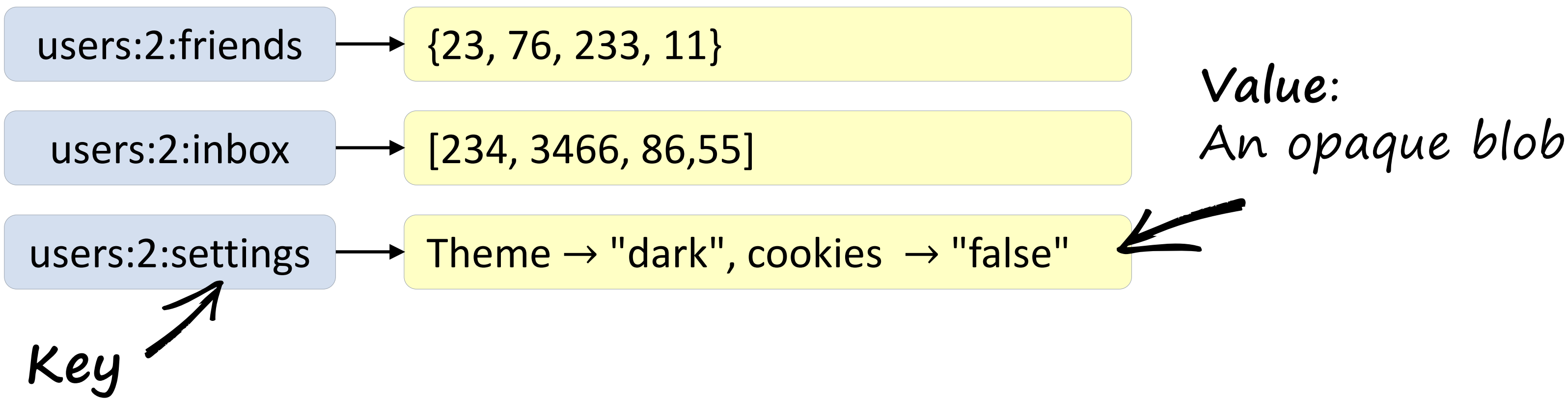
- Key-Value
- Wide-Column
- Document
- Graph

Consistency/Availability Trade-Off

- **AP**: Available & Partition Tolerant
- **CP**: Consistent & Partition Tolerant
- **CA**: Not Partition Tolerant

Key-Value Stores

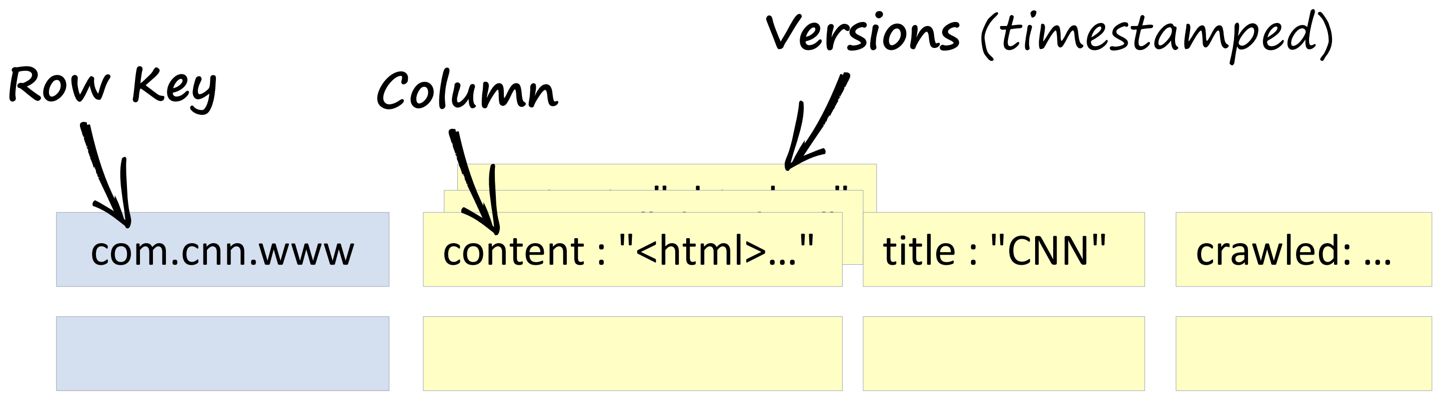
- **Data model:** (key) -> value
- **Interface**: CRUD (Create, Read, Update, Delete)



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

Wide-Column Stores

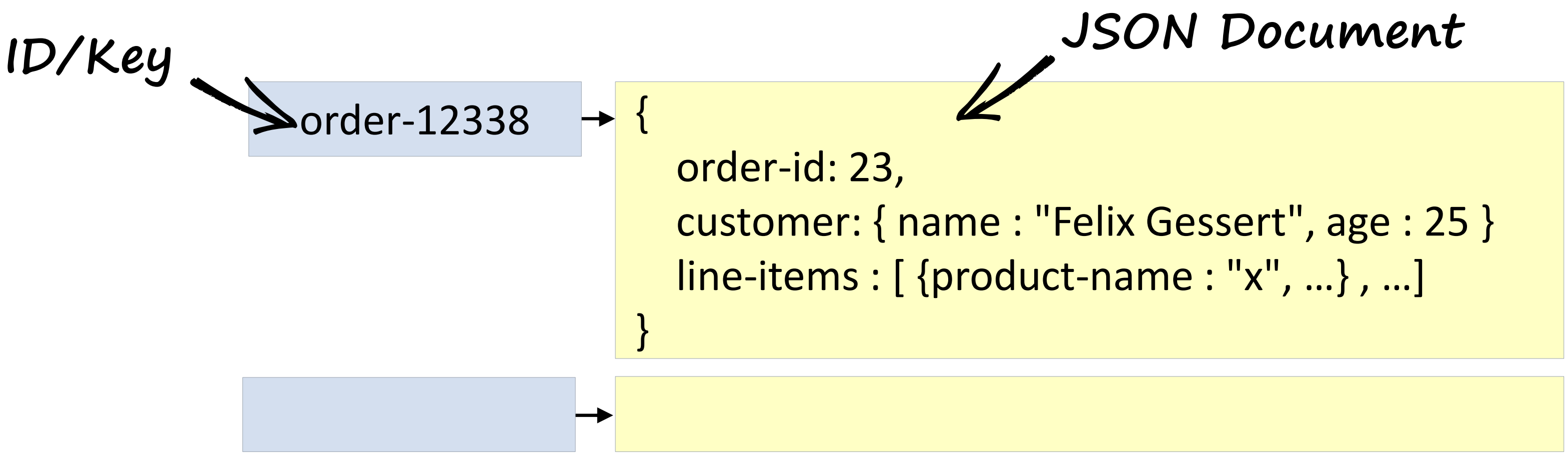
- **Data model:** (rowkey, column, timestamp) -> value
- **Interface**: CRUD, Scan



- Examples: Cassandra (AP), Google BigTable (CP), HBase (CP)

Document Stores

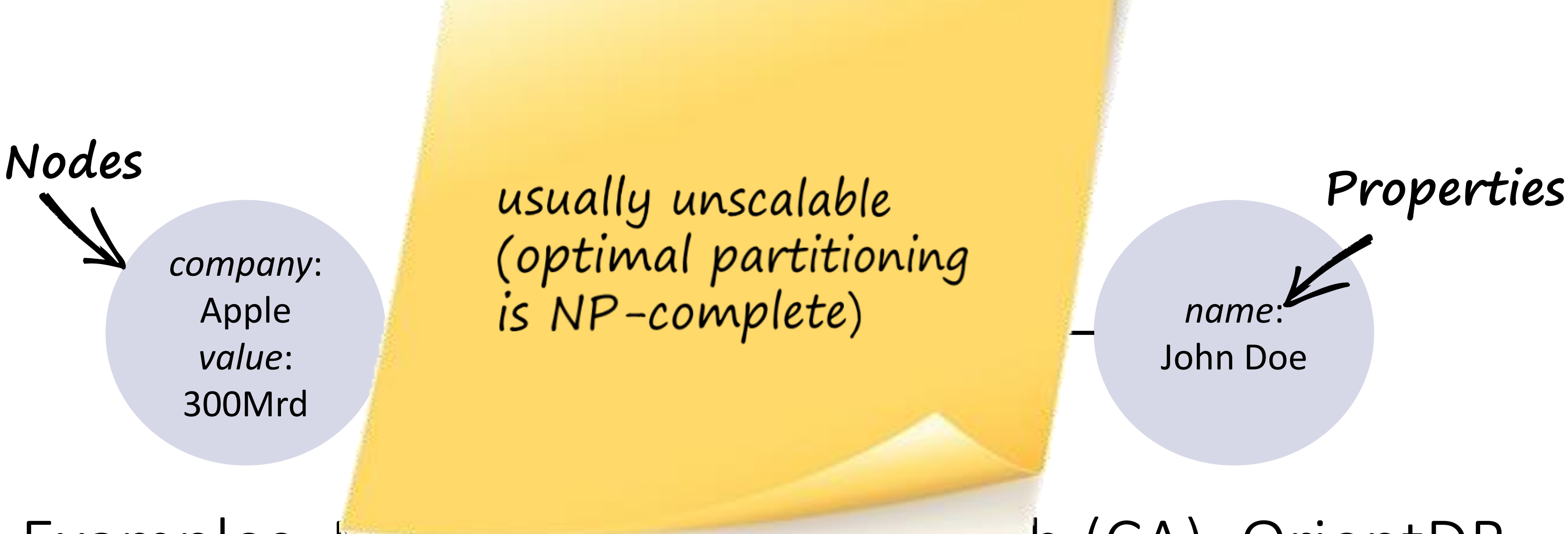
- **Data model:** (collection, key) -> document
- **Interface**: CRUD, Querys, Map-Reduce



- Examples: CouchDB (AP), RethinkDB (CP), MongoDB (CP)

Graph Databases

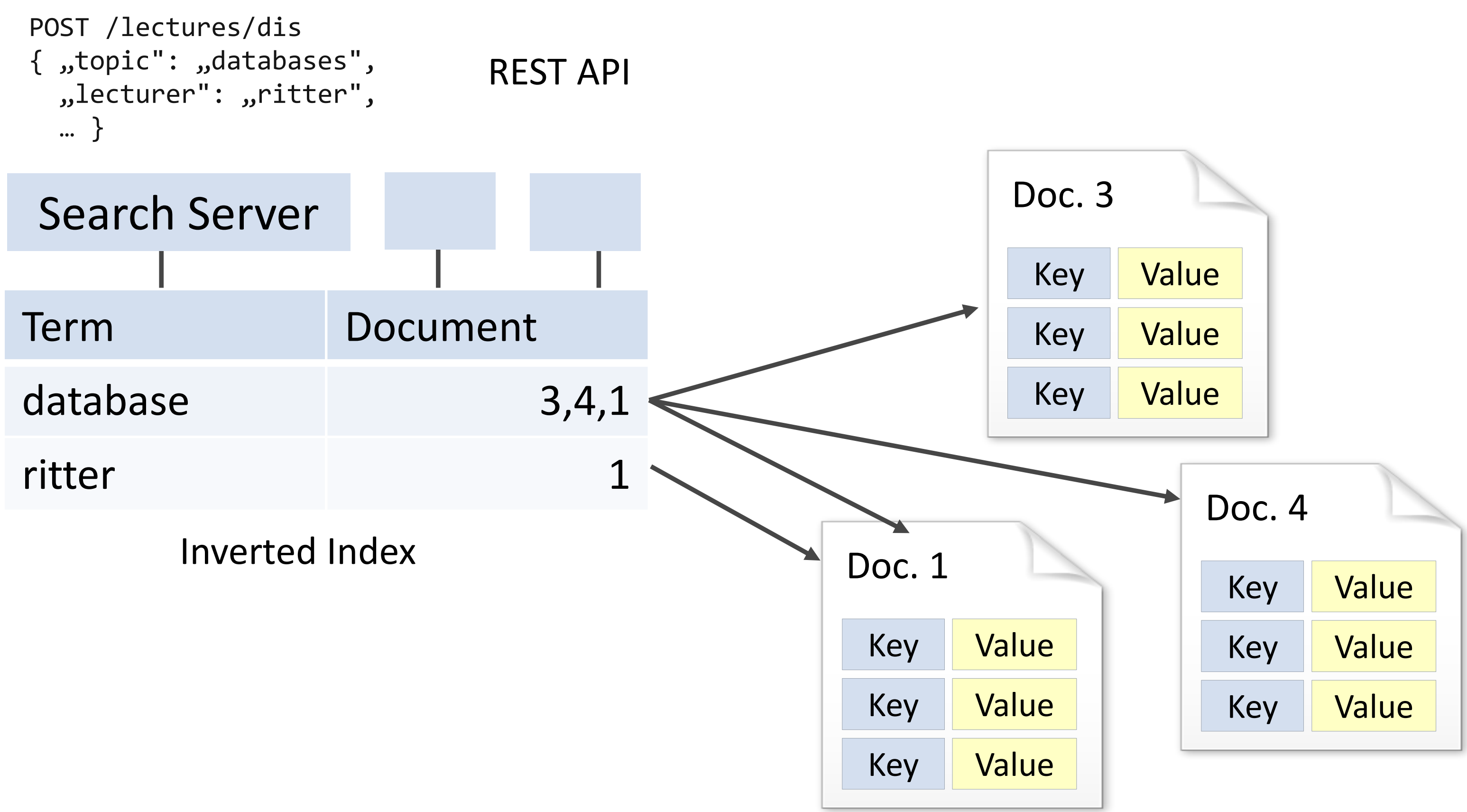
- **Data model:** G = (V, E): Graph-Property Modell
- **Interface**: Trave [illegible], transactions



- Examples: [illegible] aph (CA), OrientDB (CA)

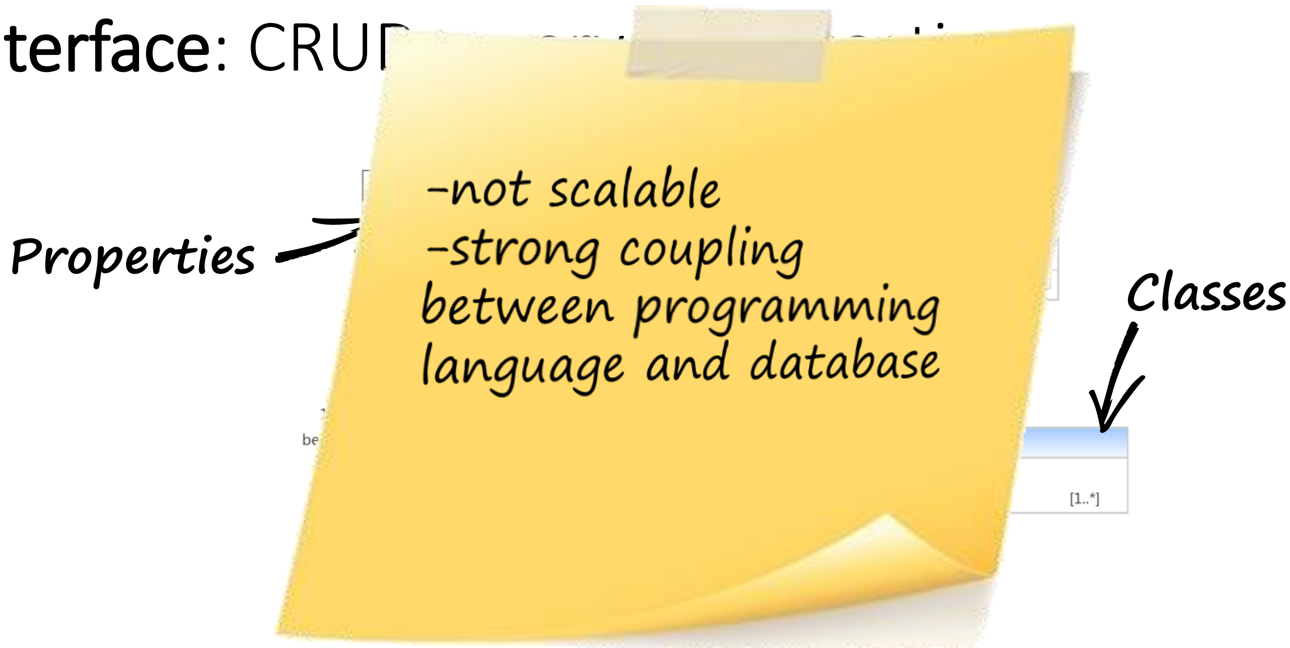
Search Platforms

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



Object-oriented Databases

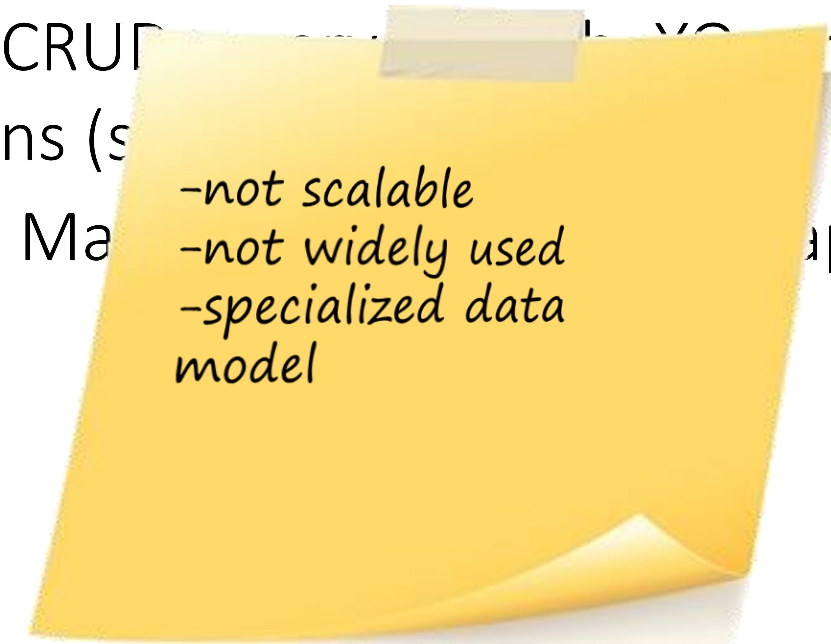
- **Data model:** Classes, objects, relations (references)
- **Interface**: CRUD



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

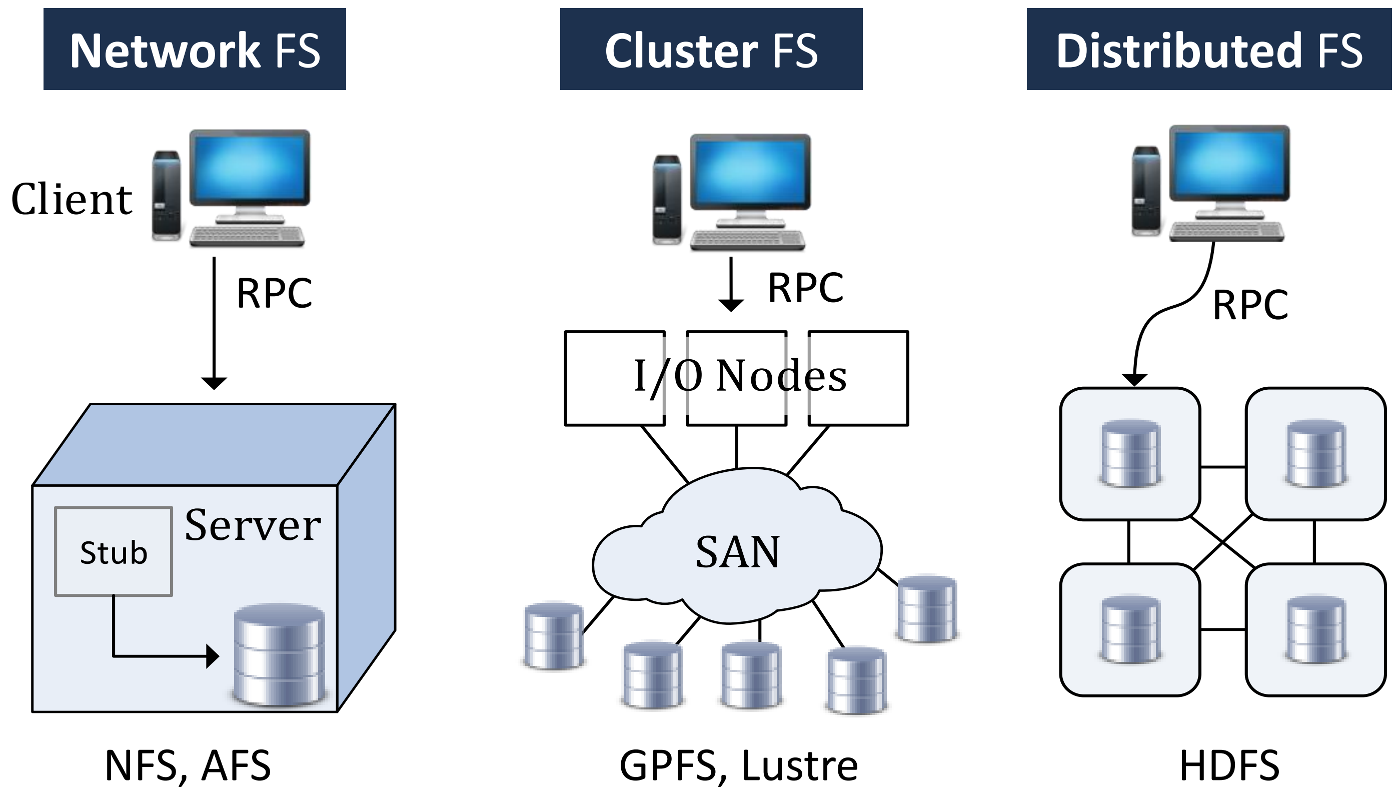
XML databases, RDF Stores

- **Data model:** XML, RDF
- **Interface**: CRUD … rys, SPARQL), transactions (s…
- Examples: Ma… aph (CA)



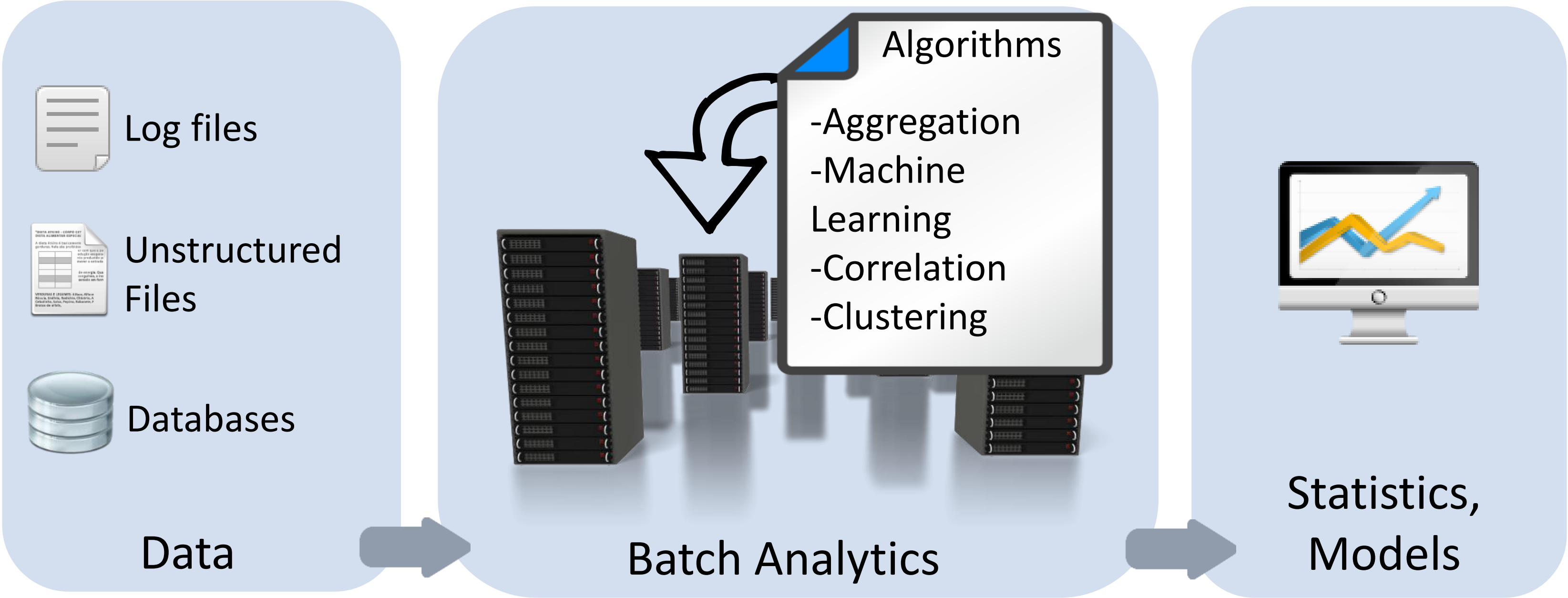
Distributed File System

- **Data model:** files + folders



Big Data Batch Processing

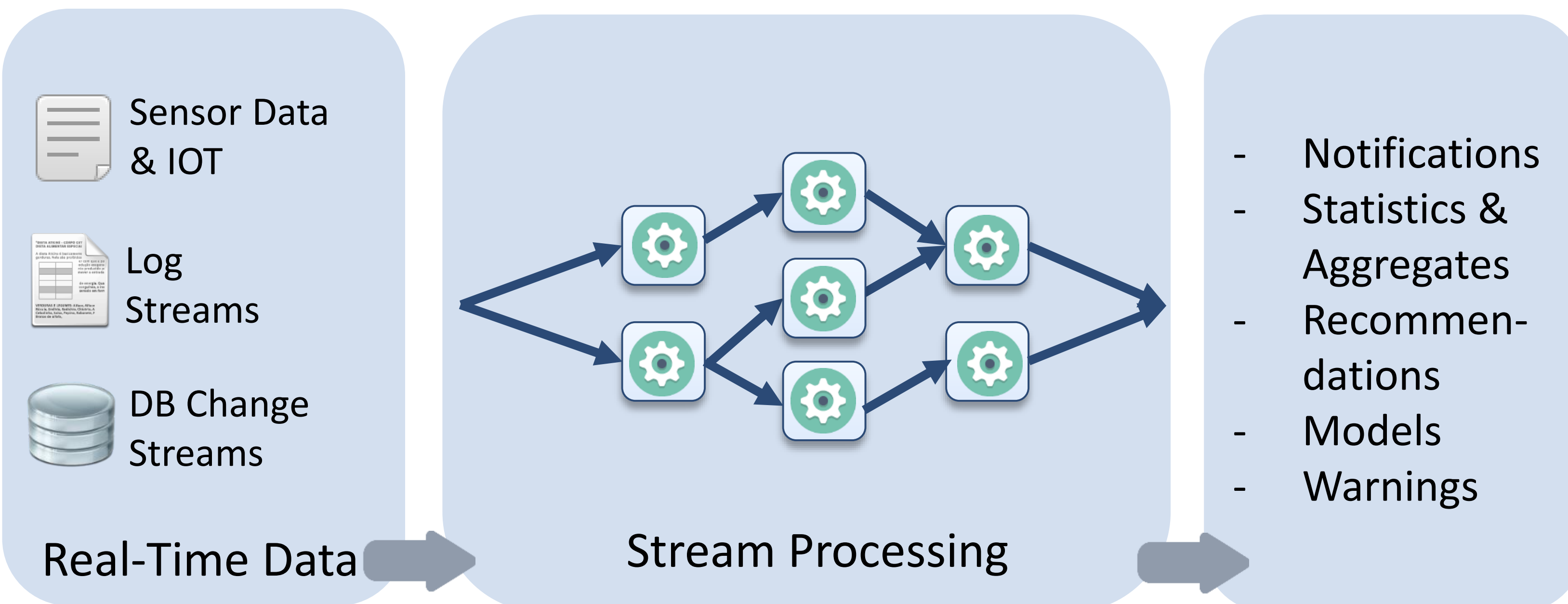
- **Data model:** arbitrary (frequently unstructured)
- Examples: Hadoop, Spark, Flink, DryadLink, Pregel



Big Data Stream Processing

Covered in Depth in the Last Part

- **Data model:** arbitrary
- Examples: Storm, Samza, Flink, Spark Streaming



Real-Time Databases

Covered in Depth in the Last Part

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

- Examples: Firebase (CP), Parse (CP), Meteor (CP), Lambda/Kappa Architecture

Soft NoSQL Systems

Not Covered Here

Search Platforms (Full Text Search):

- No persistence and consistency guarantees for OLTP
- *Examples*: ElasticSearch (AP), Solr (AP)

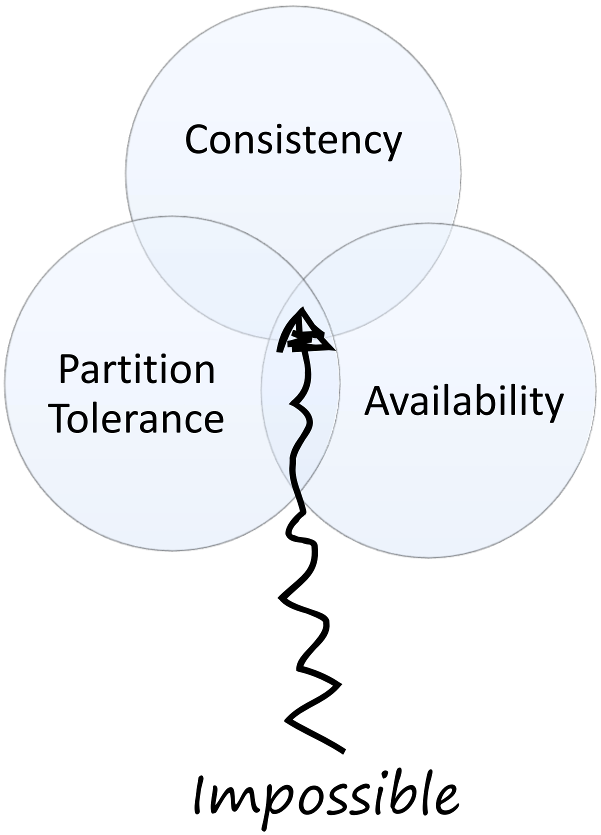
Object-Oriented Databases:

- Strong coupling of programming language and DB
- *Examples*: Versant (CA), db4o (CA), Objectivity (CA)

XML-Databases, RDF-Stores:

- Not scalable, data models not widely used in industry
- *Examples*: MarkLogic (CA), AllegroGraph (CA)

CAP-Theorem



Only 2 out of 3 properties are achievable at a time:

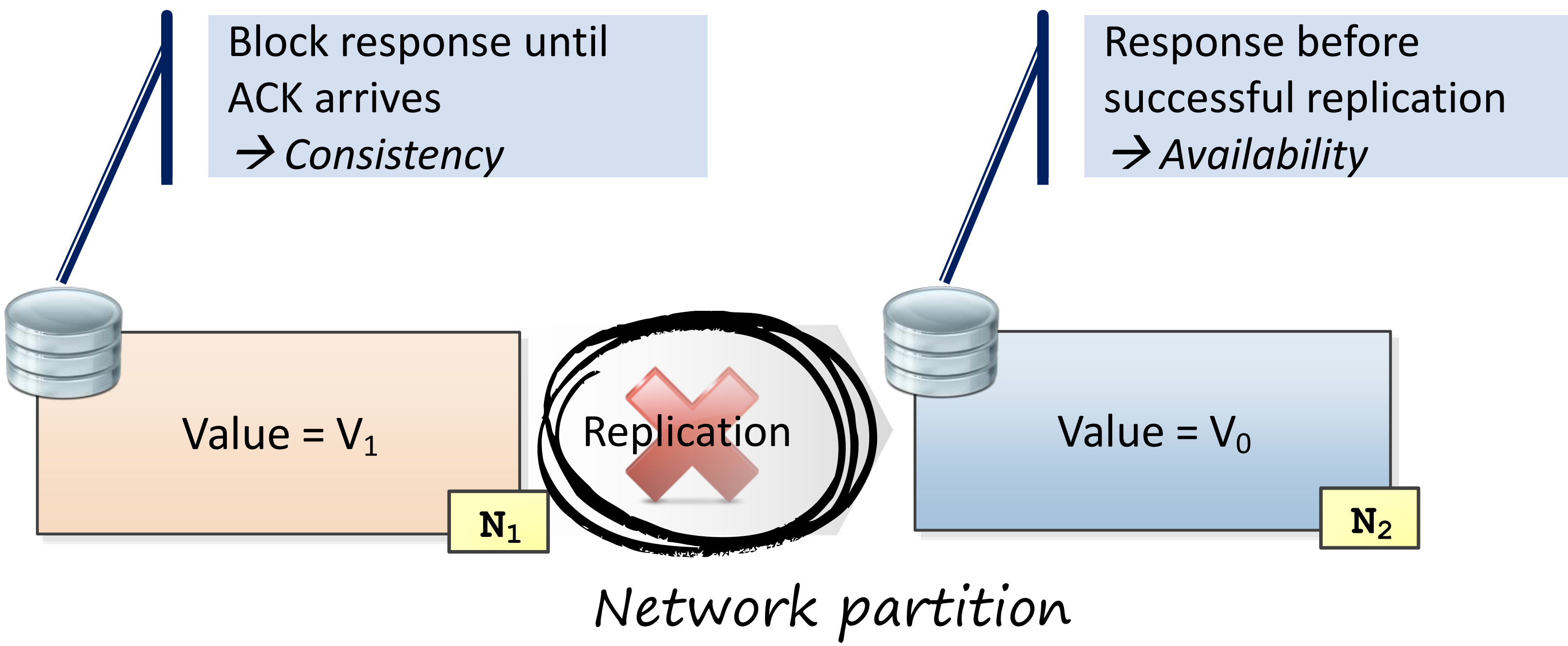
- **Consistency**: all clients have the same view on the data
- **Availability**: every request to a non-failed node most result in correct response
- **Partition tolerance**: the system has to continue working, even under arbitrary network partitions

Eric Brewer, ACM-PODC Keynote, Juli 2000

Gilbert, Lynch: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services, SigAct News 2002

CAP-Theorem: simplified proof

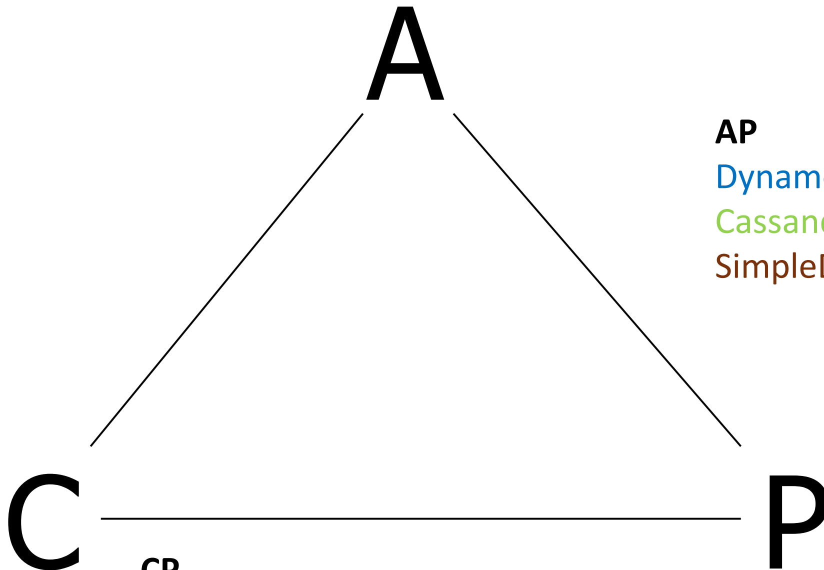
- **Problem**: when a network partition occurs, either consistency or availability have to be given up



NoSQL Triangle

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

Every client can always read and write



CA
Oracle, MySQL, …

AP
Dynamo, Redis, Riak, Voldemort
Cassandra
SimpleDB

All clients share the same view on the data

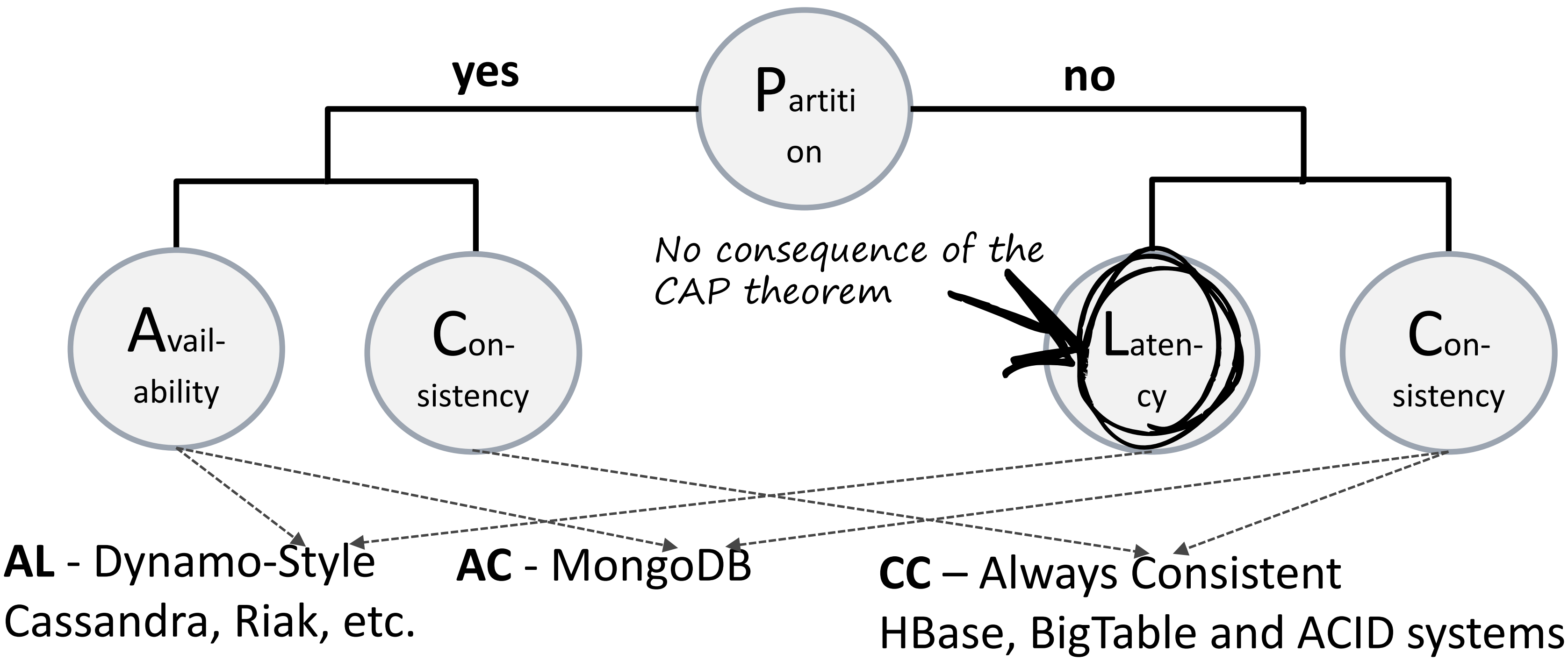
CP
Postgres, MySQL Cluster, Oracle RAC
BigTable, HBase, Accumulo, Azure Tables
MongoDB, RethinkDB, DocumentsDB

All nodes continue working under network partitions

Nathan Hurst: Visual Guide to NoSQL Systems
http://blog.nahurst.com/visual-guide-to-nosql-systems

PACELC – an alternative CAP formulation

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



Abadi, Daniel. "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story."

Serializability

Not Highly Available Either

Global serializability and availability are incompatible:

Write A=1
Read B

Write B=1
Read A

$w_1(a = 1)\; r_1(b = \perp)$

$w_2(b = 1)\; r_2(a = \perp)$

- Some weaker isolation levels allow high availability:
  - RAMP Transactions (P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, und I. Stoica, „Scalable Atomic Visibility with RAMP Transactions“, SIGMOD 2014)

S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. ACM CSUR, 17(3):341–370, 1985.

Impossibility Results

Consensus Algorithms

- Consensus:
  - *Agreement*: No two processes can commit different decisions
  - *Validity (Non-triviality)*: If all initial values are same, nodes must commit that value
  - *Termination*: Nodes commit eventually
- No algorithm *guarantees* termination (FLP)
- Algorithms:
  - **Paxos** (e.g. Google Chubby, Spanner, Megastore, Aerospike, Cassandra Lightweight Transactions)
  - **Raft** (e.g. RethinkDB, etcd service)
  - Zookeeper Atomic Broadcast (**ZAB**)

Safety Properties

Liveness Property

Lynch, Nancy A. *Distributed algorithms*. Morgan Kaufmann, 1996.

Where CAP fits in

Negative Results in Distributed Computing

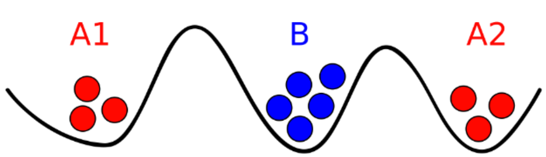
***Asynchronous* Network, *Unreliable* Channel**

Atomic Storage

Impossible:
CAP Theorem

Consensus

Impossible:
2 Generals Problem



***Asynchronous* Network, *Reliable* Channel**

Atomic Storage

Possible:
Attiya, Bar-Noy, Dolev (ABD) Algorithm

Consensus

Impossible:
Fisher **L**ynch **P**atterson (FLP) Theorem

Lynch, Nancy A. *Distributed algorithms*. Morgan Kaufmann, 1996.

ACID vs BASE

ACID

„Gold standard“ for RDBMSs

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

BASE

Model of many NoSQL systems

- **Ba**sically **A**vailable
- **S**oft State
- **E**ventually Consistent

http://queue.acm.org/detail.cfm?id=1394128

Weaker guarantees in a database?!

Default Isolation Levels in RDBMSs

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

Theorem:

Trade-offs are central to database systems.

RC: read committed, RR: repeatable read, S: serializability, SI: snapshot isolation, CS: cursor stability, CR: consistent read

Bailis, Peter, et al. "Highly available transactions: Virtues and limitations." *Proceedings of the VLDB Endowment* 7.3 (2013): 181-192.

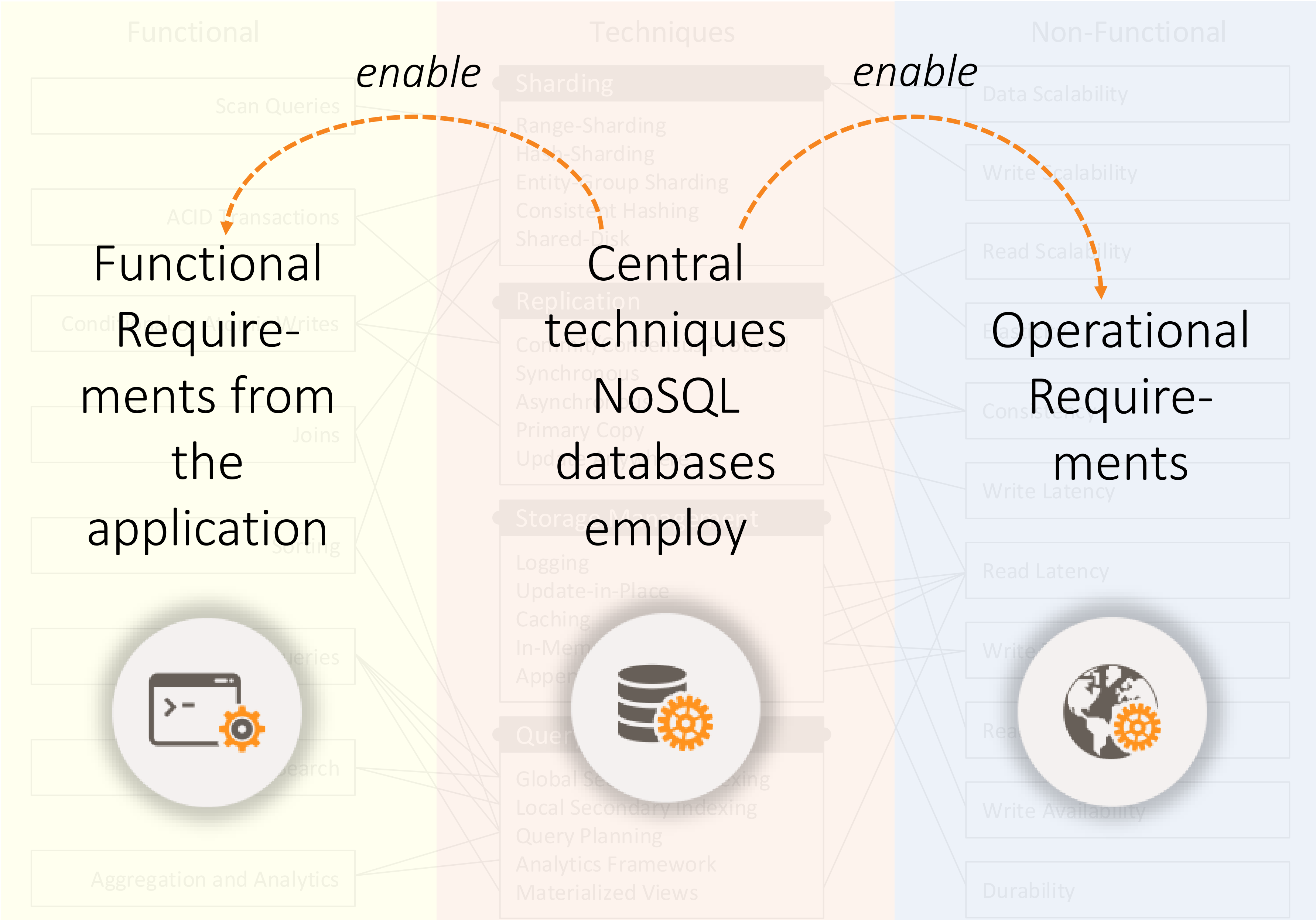
Data Models and **CAP** provide high-level classification.

But what about **fine-grained requirements**, e.g. query capabilites?

Outline

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

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


enable
enable
Functional Require-ments from the application
Central techniques NoSQL databases employ
Operational Require-ments

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

NoSQL Databases: a Survey and Decision Guidance

Felix Gessert
Aug 15, 2016 · 26 min read

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

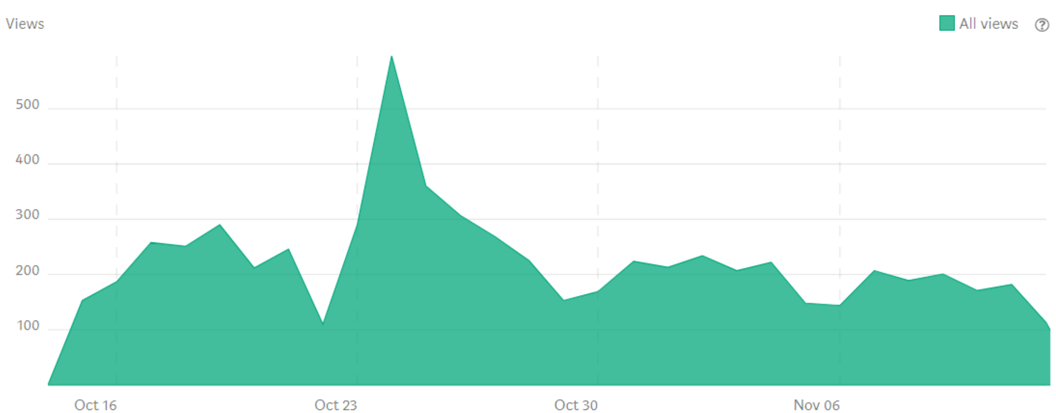
As a blog post: blog.baqend.com

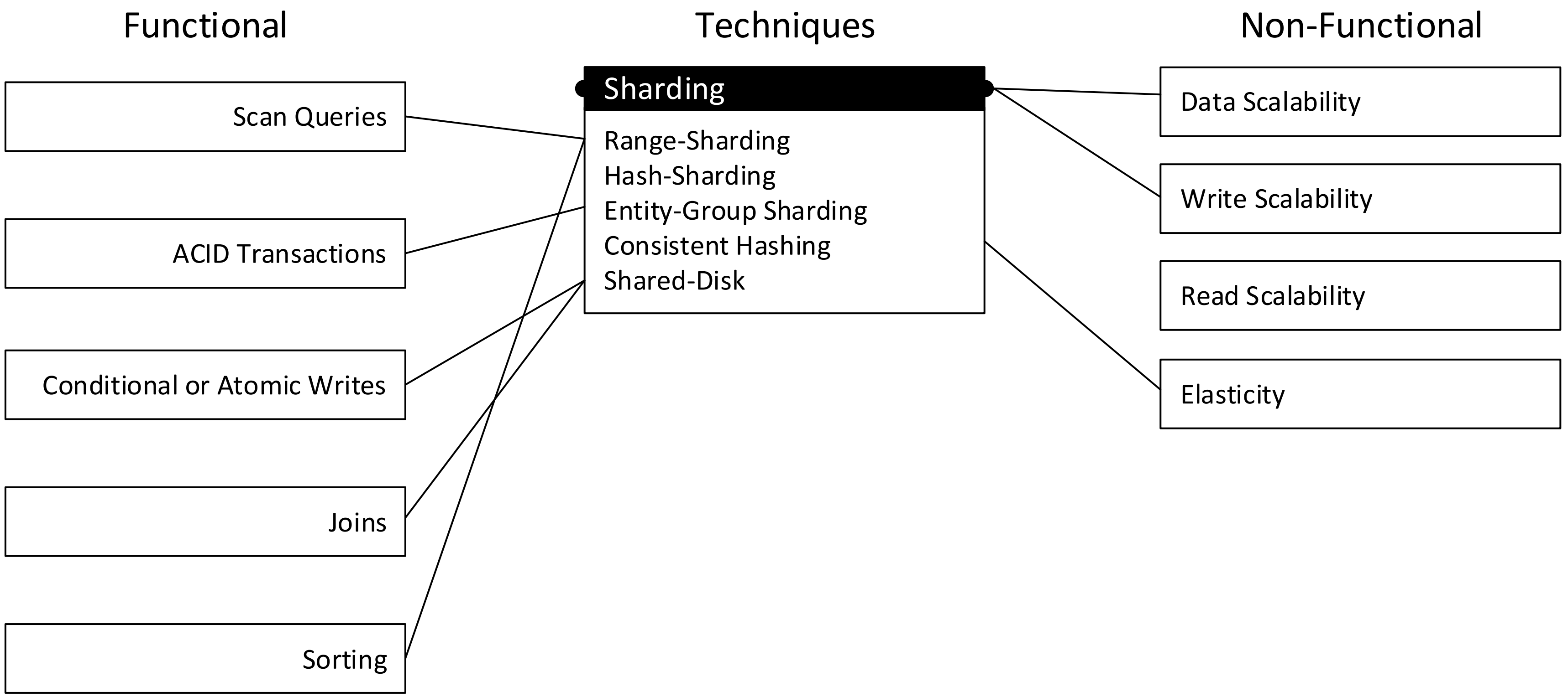
Published on August 15, 2016 in Baqend Blog

See all stats

NoSQL Databases: a Survey and Decision Guidance

TOTAL VIEWS	READ RATIO
134K	28%



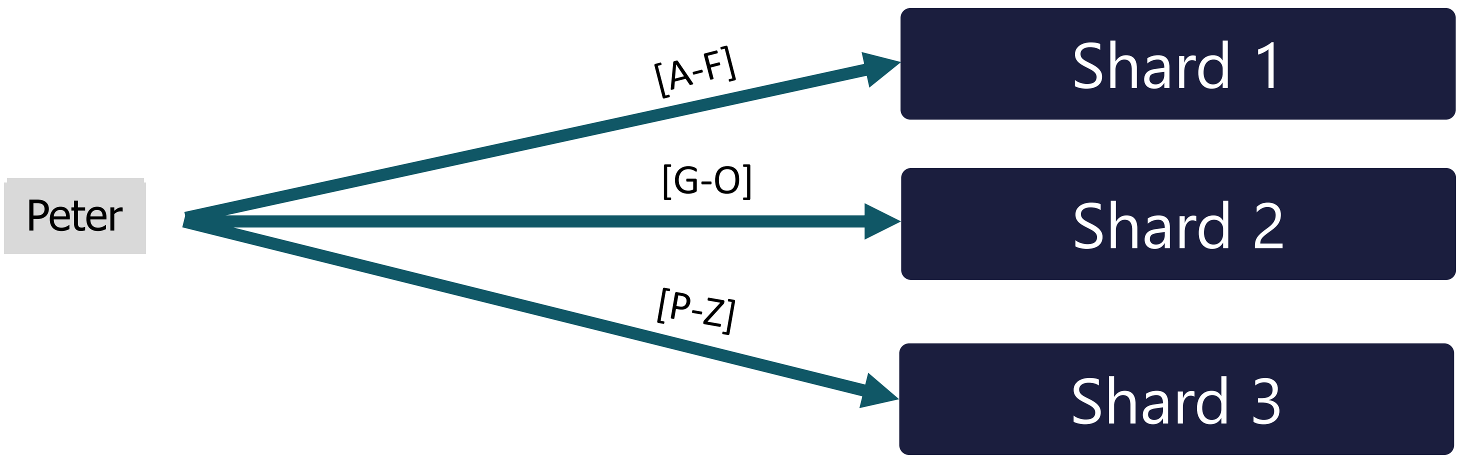


Functional
Techniques
Non-Functional
Scan Queries
ACID Transactions
Conditional or Atomic Writes
Joins
Sorting
Sharding
Range-Sharding
Hash-Sharding
Entity-Group Sharding
Consistent Hashing
Shared-Disk
Data Scalability
Write Scalability
Read Scalability
Elasticity

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) d
- **Pro**: Even distribution
- **Contra**: No data locality

Range-based Sharding

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

Entity-Group Sharding

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

Implemented in

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

Implemented in

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

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.

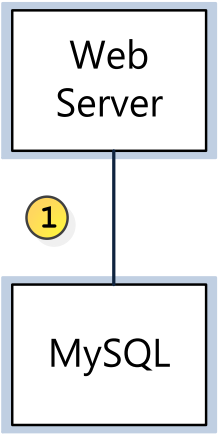
Problems of Application-Level Sharding

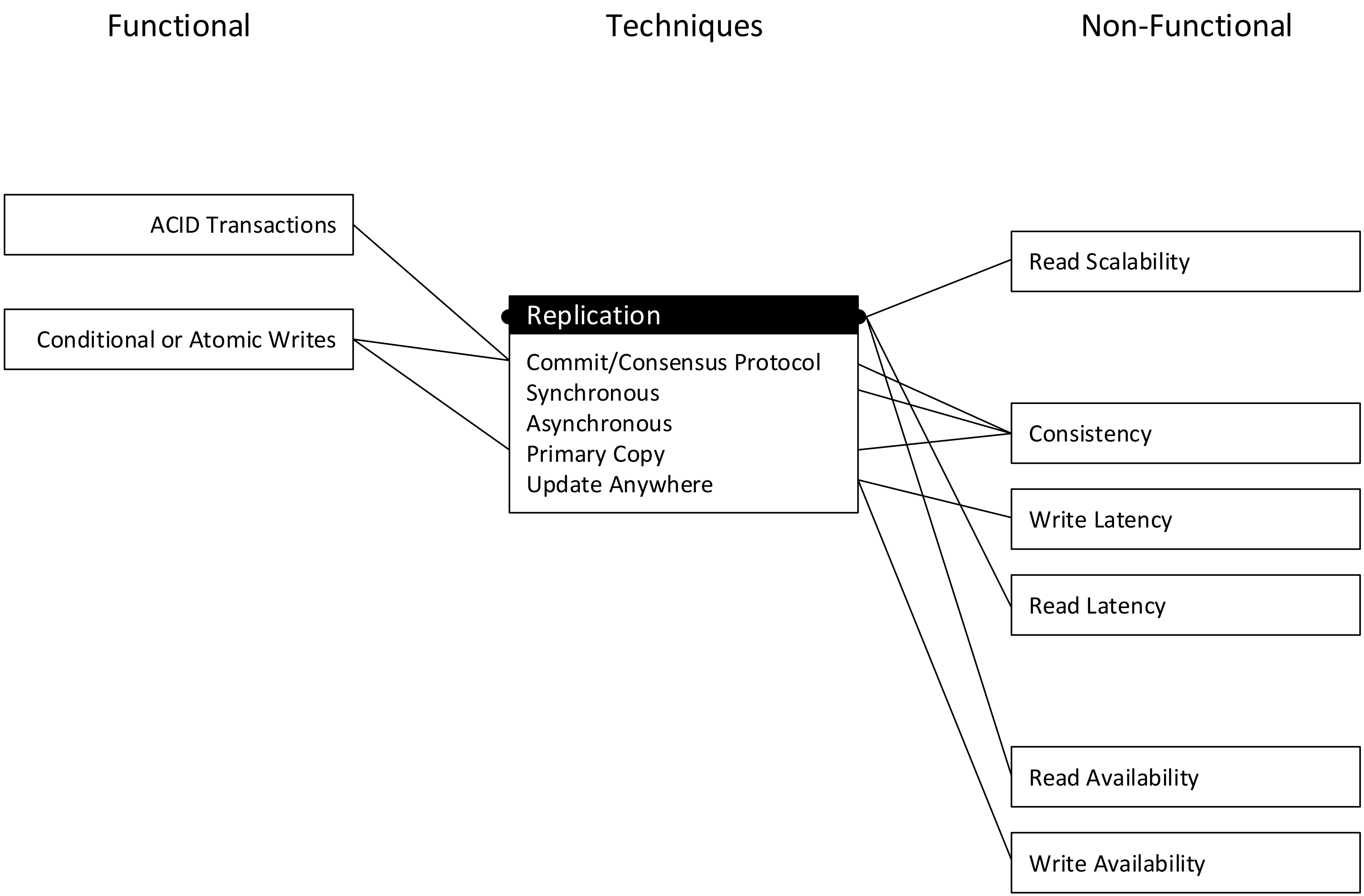
Example: **Tumblr**

- Caching
- Sharding from application

Moved towards:

- Redis
- HBase



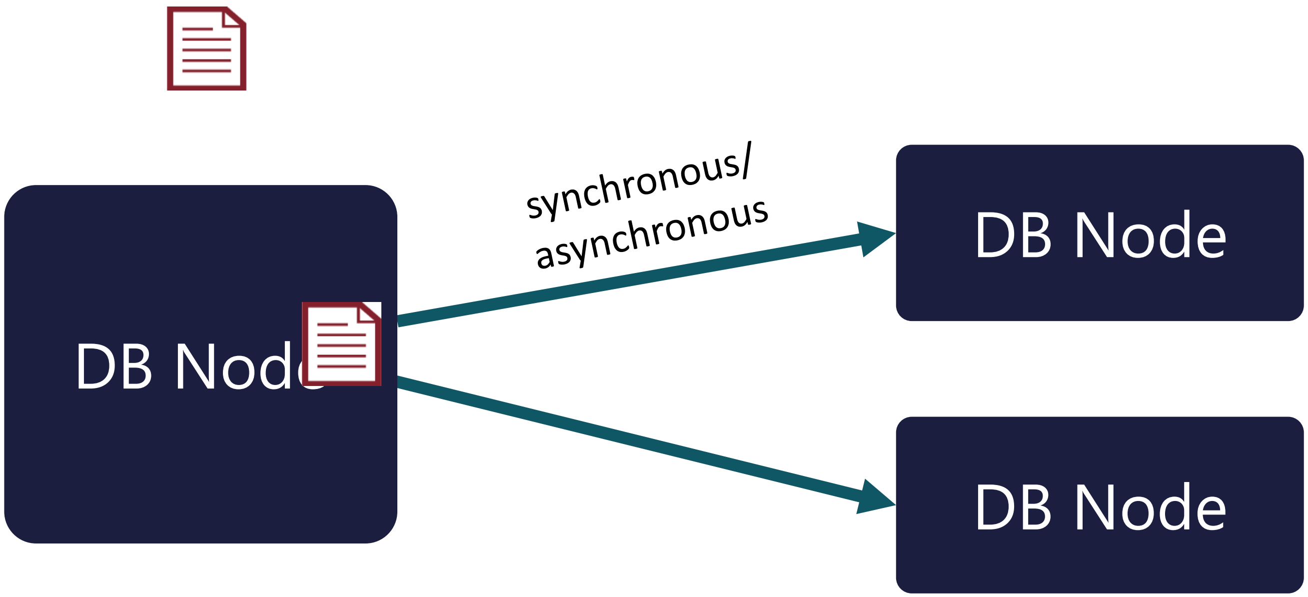


Functional
Techniques
Non-Functional
ACID Transactions
Conditional or Atomic Writes
Replication
Commit/Consensus Protocol
Synchronous
Asynchronous
Primary Copy
Update Anywhere
Read Scalability
Consistency
Write Latency
Read Latency
Read Availability
Write Availability

Replication

Read Scalability + Failure Tolerance

- Stores *N* copies of each data item



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

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

Replication: When

Asynchronous (lazy)

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

Implemented in

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

Synchronous (eager)

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

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

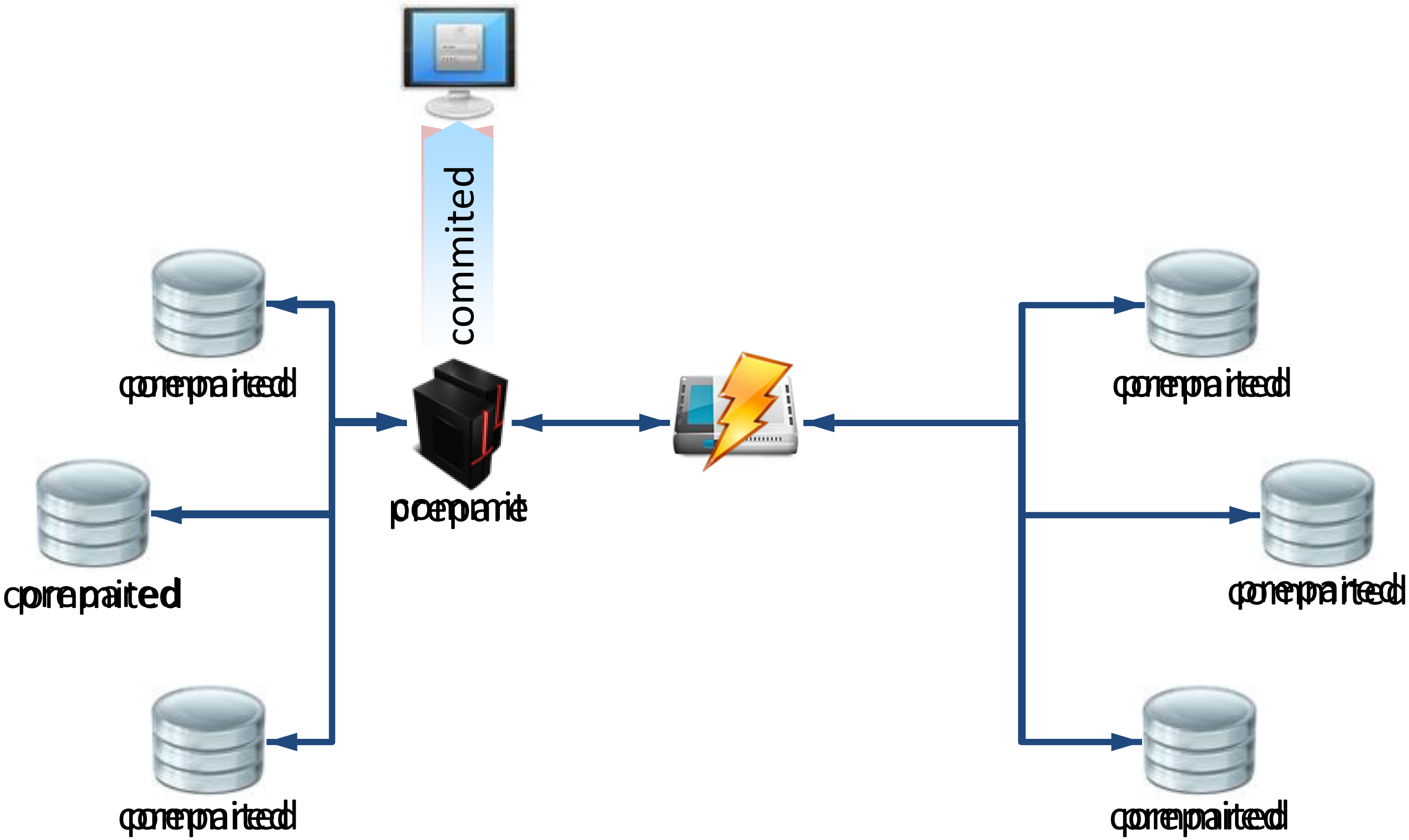
Master-Slave (*Primary Copy*)
- Only a dedicated master is allowed to accept writes, slaves are read-replicas
- **Pro**: reads from the master are consistent
- **Contra**: master is a bottleneck and SPOF

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

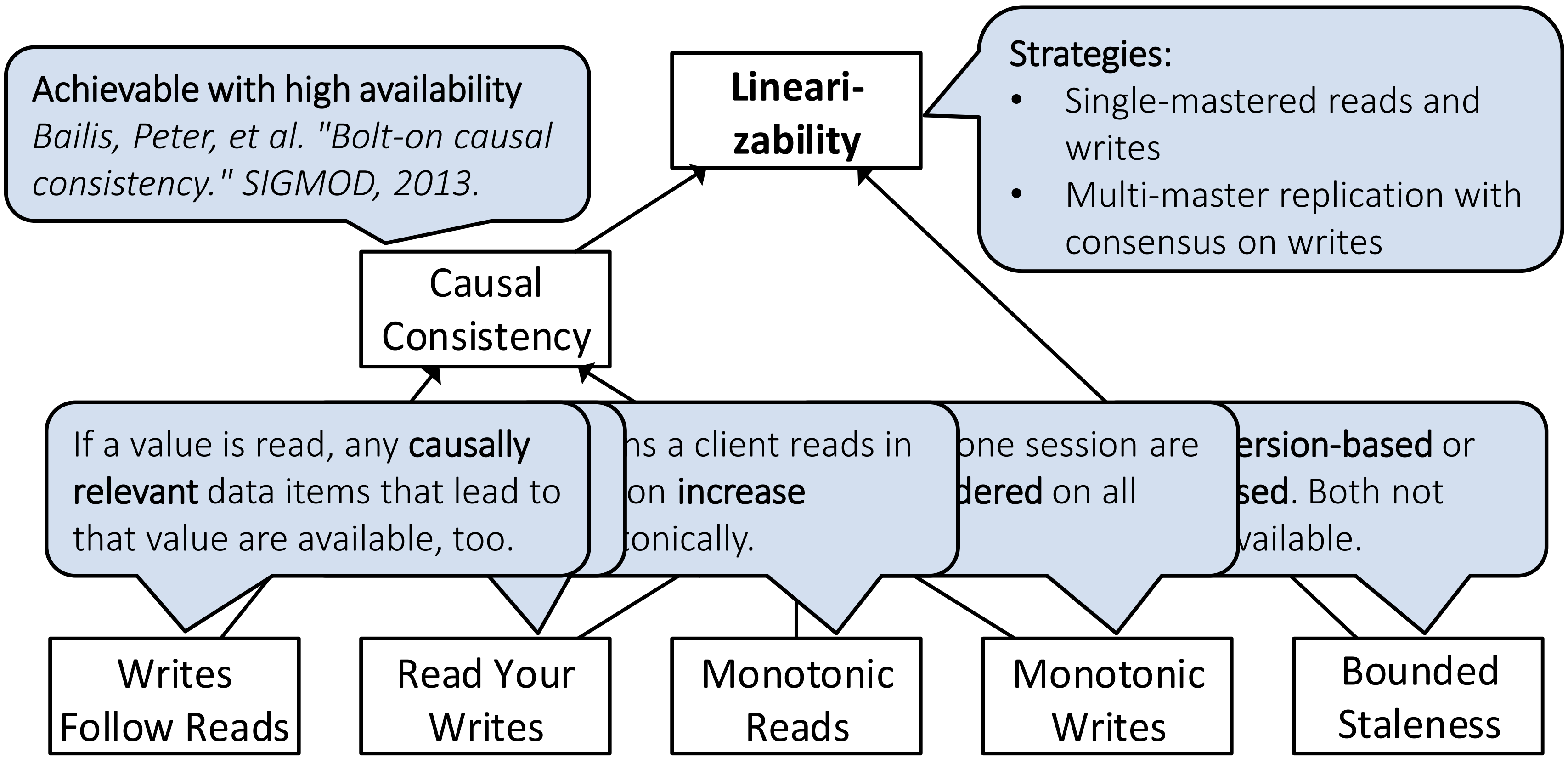
Charron-Bost, B., Pedone, F., Schiper, A. (eds.): Replication: Theory and Practice, Lecture Notes in Computer Science, vol. 5959. Springer (2010)

Synchronous Replication

Example: Two-Phase Commit is not partition-tolerant



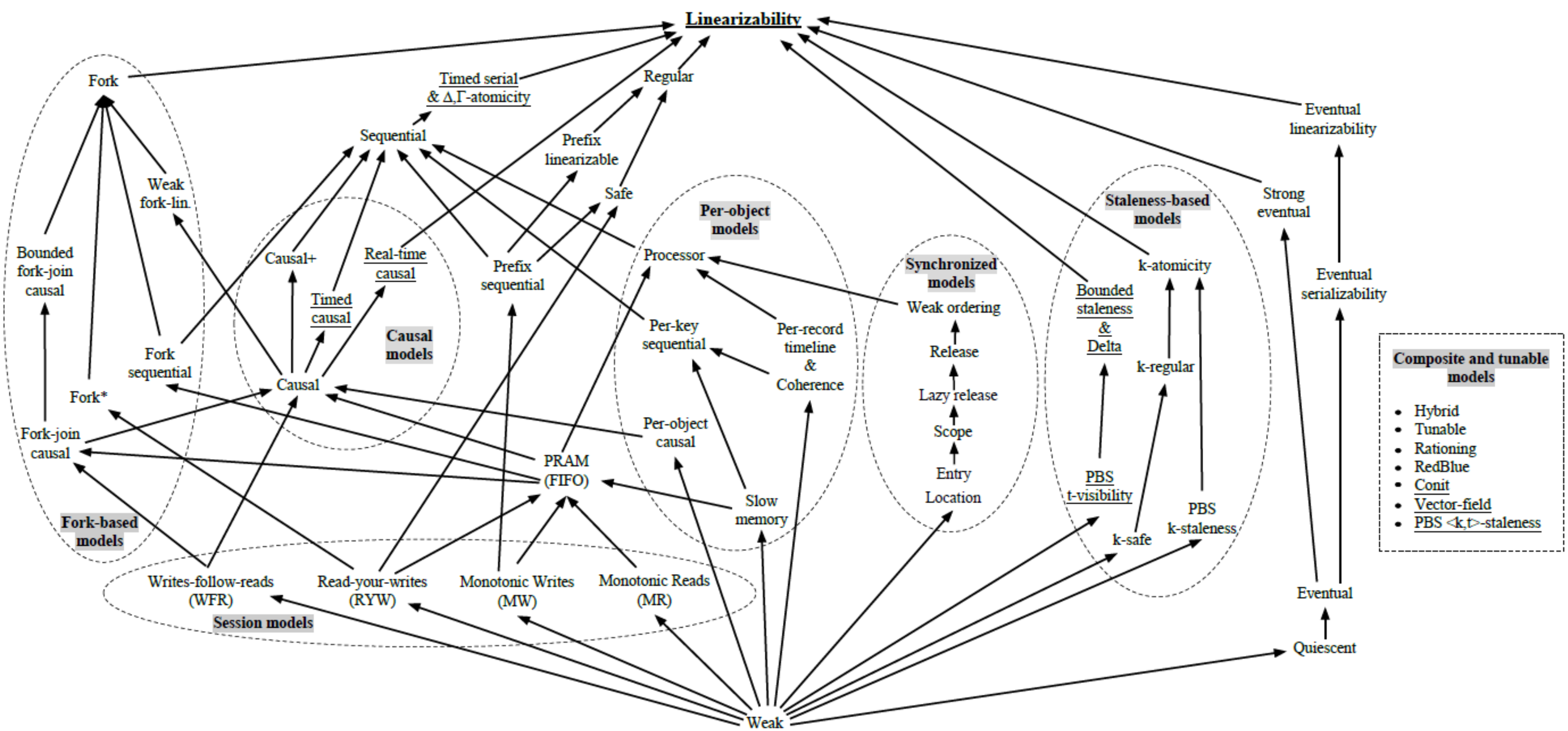
Consistency Levels



Viotti, Paolo, and Marko Vukolić. "Consistency in Non-Transactional Distributed Storage Systems." arXiv (2015).

Bailis, Peter, et al. "Highly available transactions: Virtues and limitations." Proceedings of the VLDB Endowment 7.3 (2013): 181-192.

Problem: Terminology

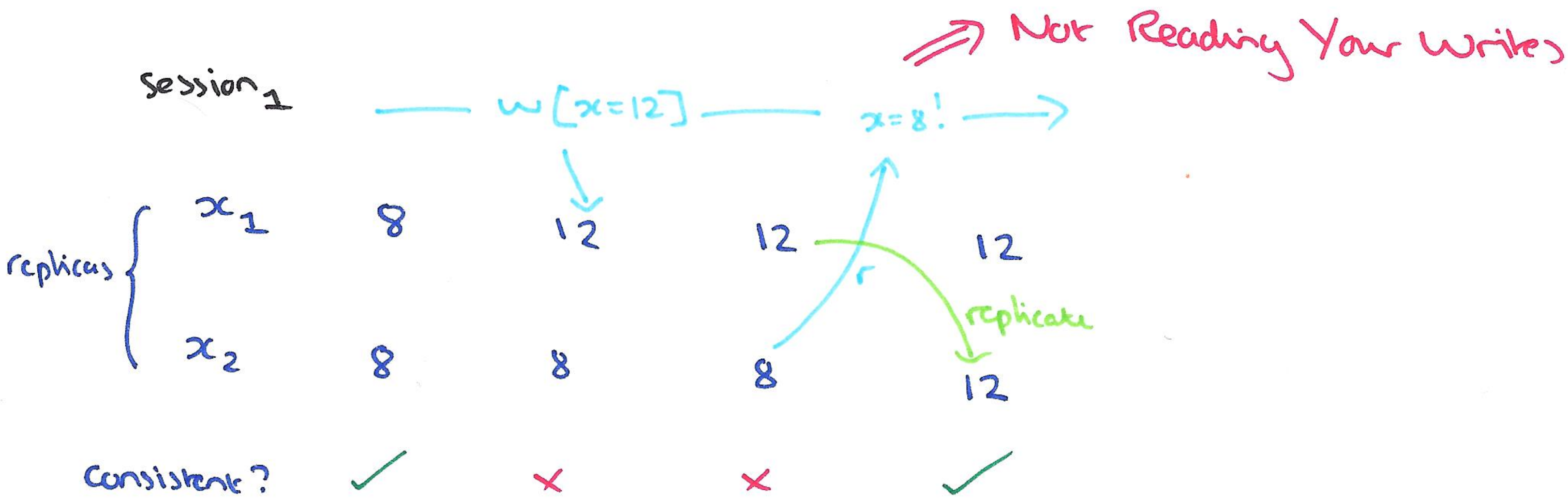


V., Paolo, and M. Vukolić. "Consistency in Non-Transactional Distributed Storage Systems." ACM CSUR (2016).

Bailis, Peter, et al. "Highly available transactions: Virtues and limitations." Proceedings of the VLDB Endowment 7.3 (2013): 181-192.

Read Your Writes (RYW)

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

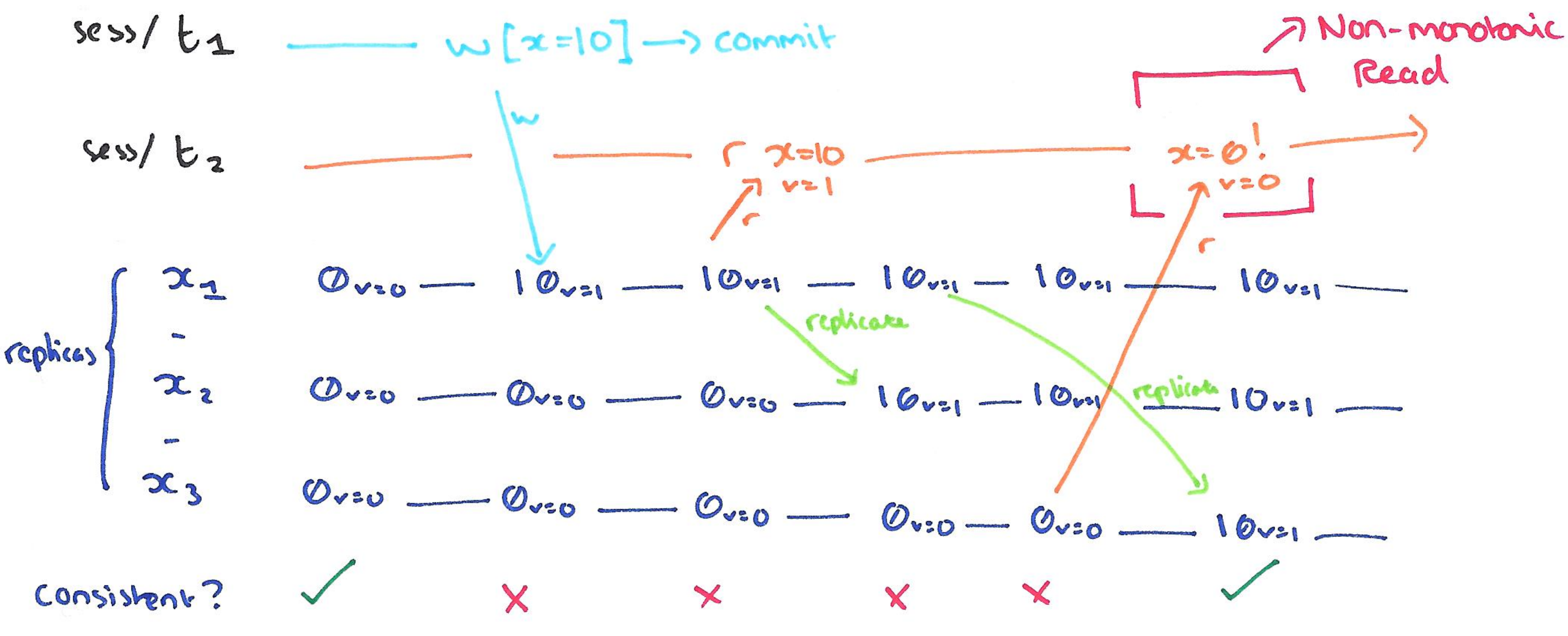


https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/

Wiese, Lena. Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases. De Gruyter, 2015.

Monotonic Reads (MR)

Definition: Once a user has read a version of a data item on one replica server, it will never see an older version on any other replica server

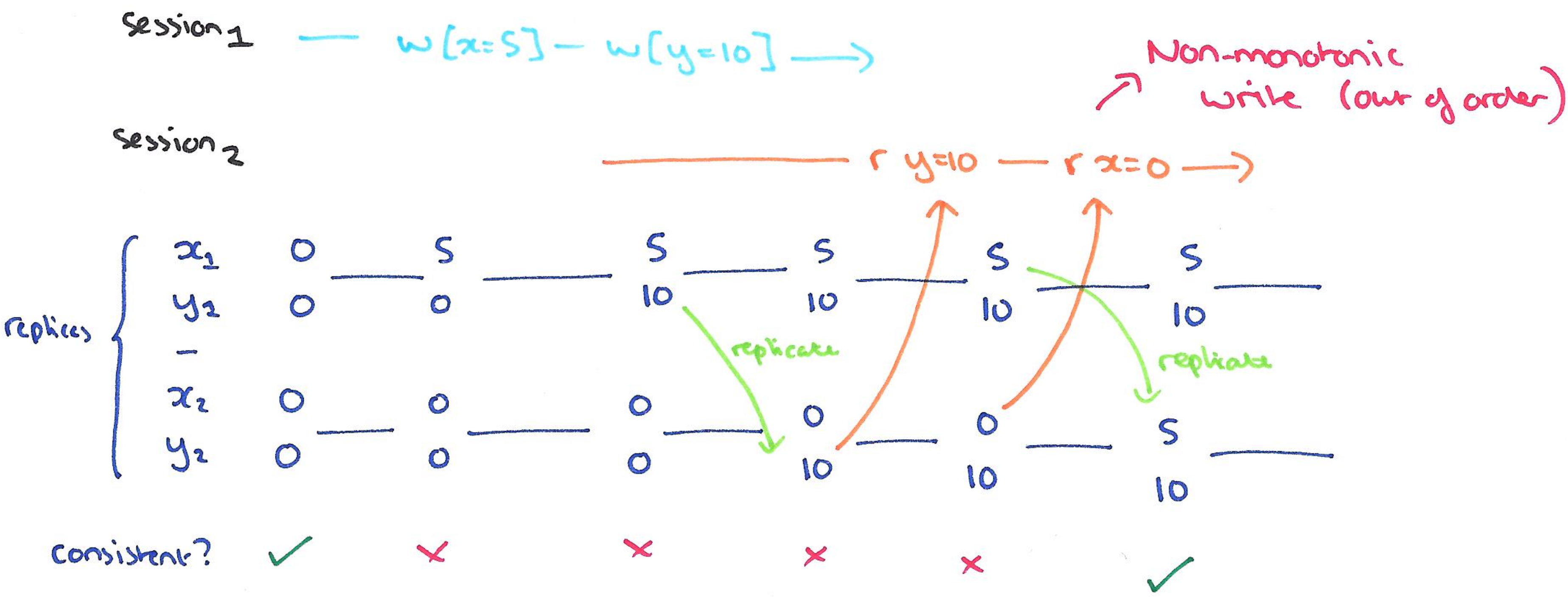


https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/

Wiese, Lena. Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases. De Gruyter, 2015.

Montonic Writes (MW)

Definition: Once a user has written a new value for a data item in a session, any previous write has to be processed before the current one. I.e., the order of writes inside the session is strictly maintained.

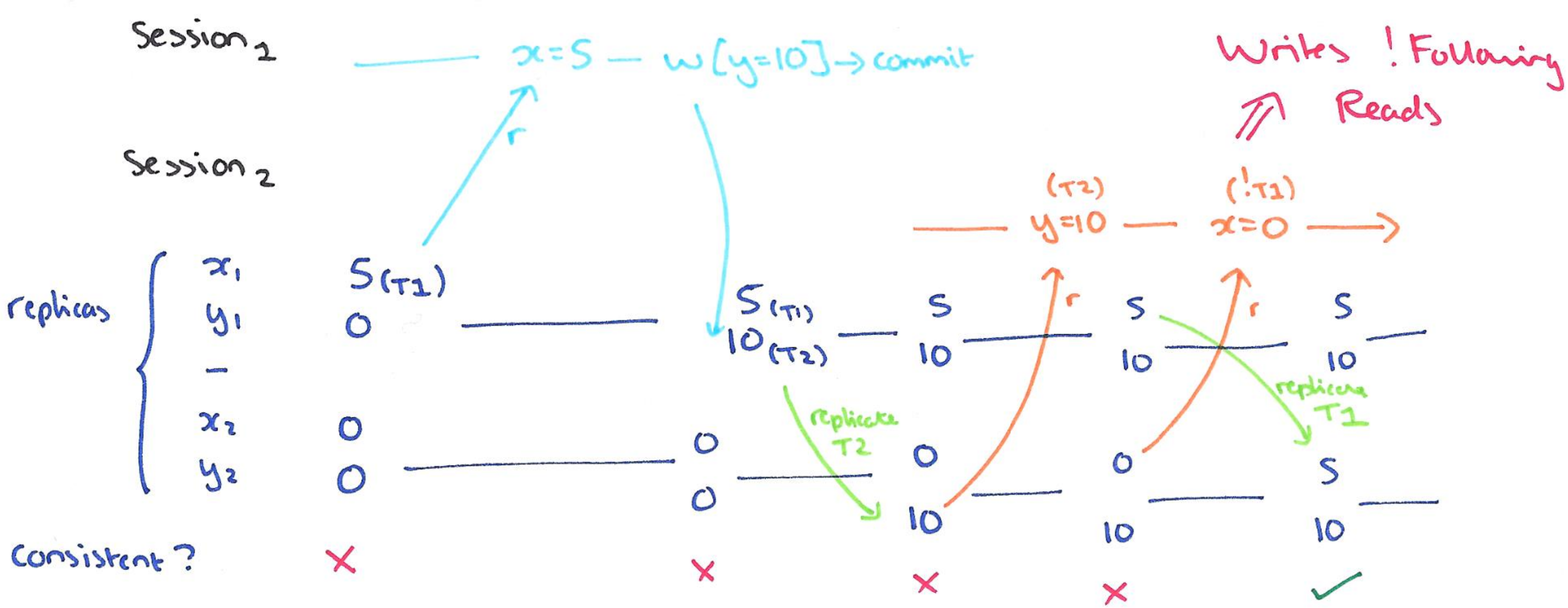


https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/

Wiese, Lena. Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases. De Gruyter, 2015.

Writes Follow Reads (WFR)

Definition: When a user reads a value written in a session after that session already read some other items, the user must be able to see those *causally relevant* values too.

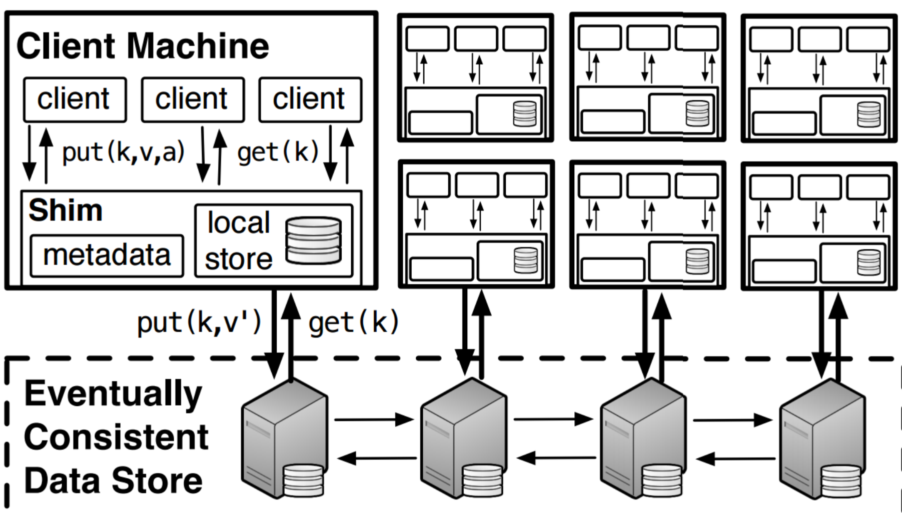


https://blog.acolyer.org/2016/02/26/distributed-consistency-and-session-anomalies/

Wiese, Lena. Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases. De Gruyter, 2015.

PRAM and Causal Consistency

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



Bailis, Peter, et al. "Bolt-on causal consistency." Proceedings of the 2013 ACM SIGMOD, 2013.

Bounded Staleness

- Either **time-based**:

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

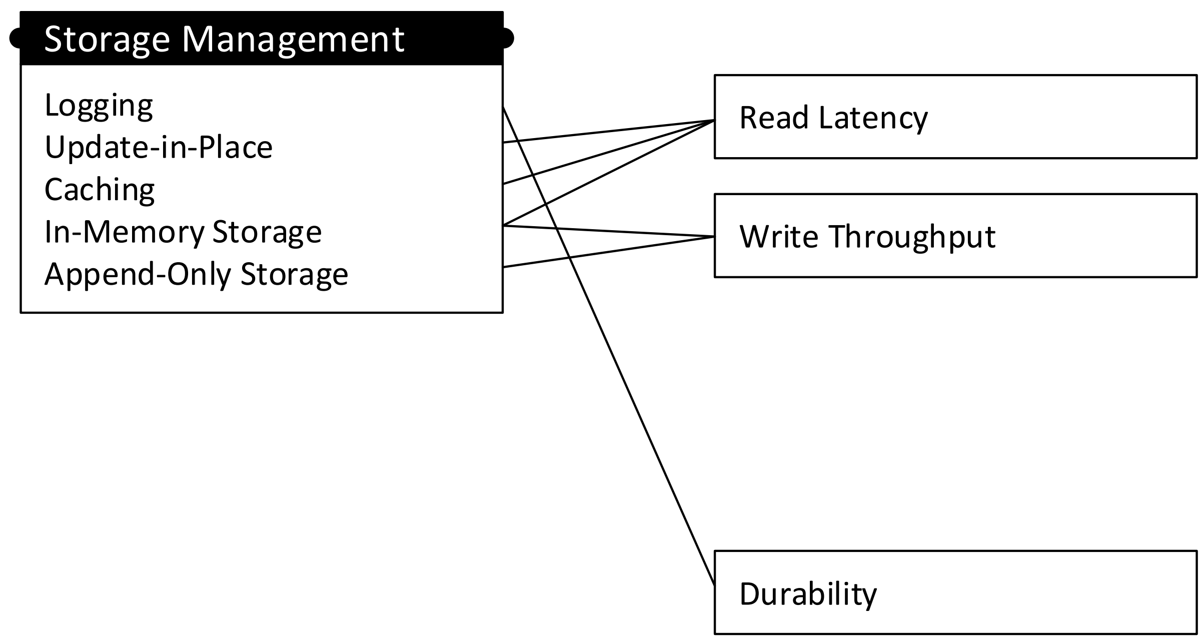
- Or **version-based:**

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

- Both are *not* achievable with high availability

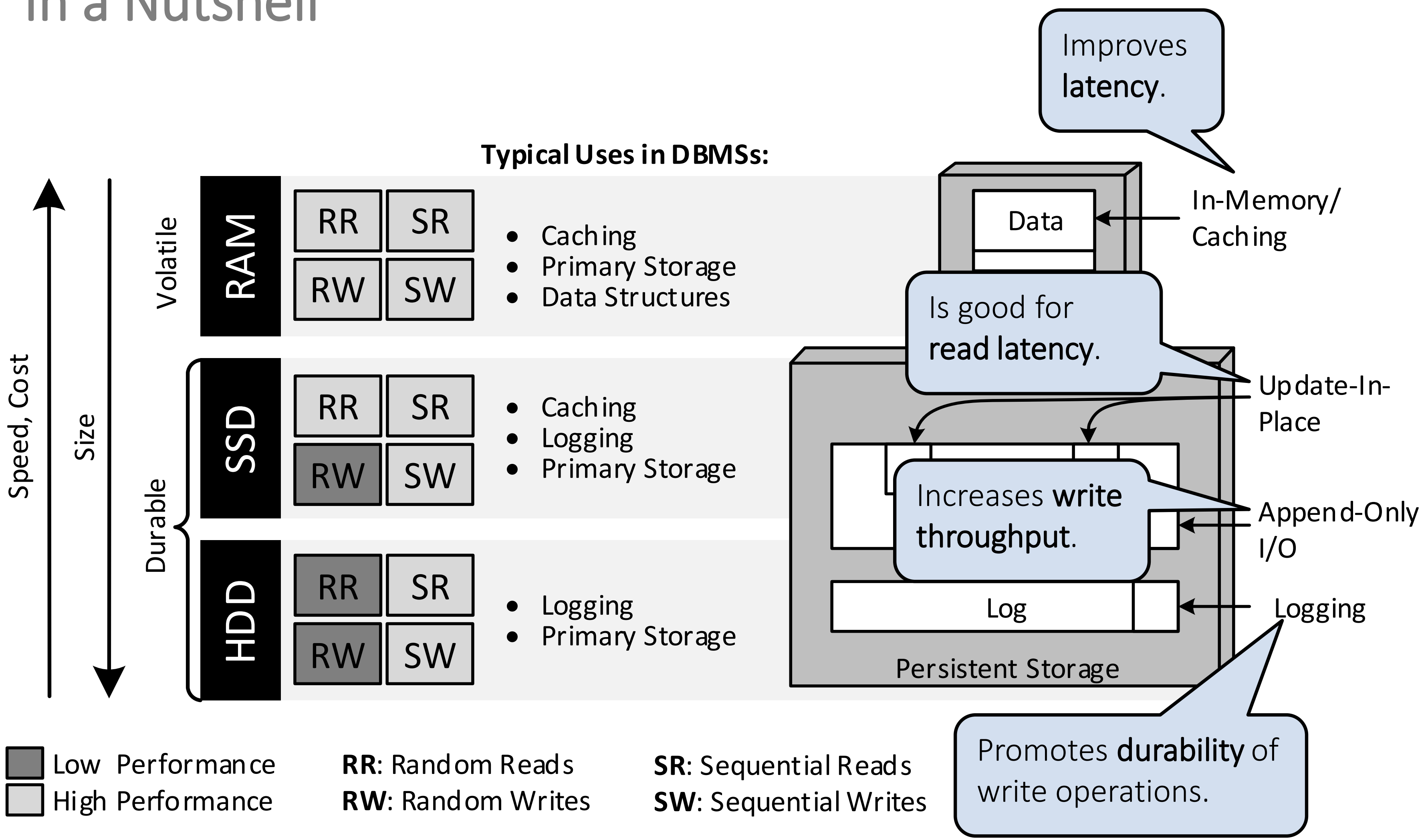
Wiese, Lena. Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases. De Gruyter, 2015.

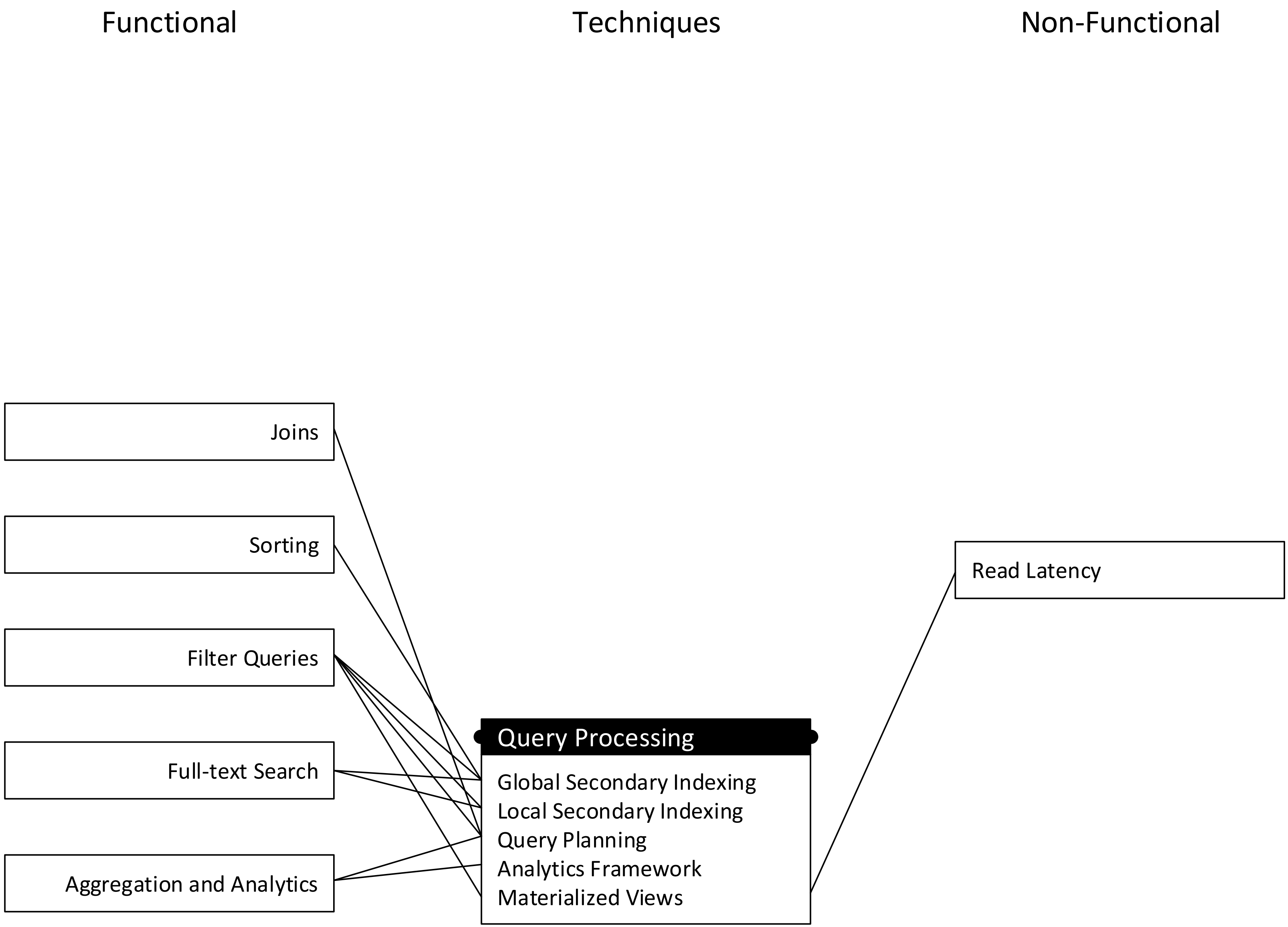
Functional Techniques Non-Functional



NoSQL Storage Management

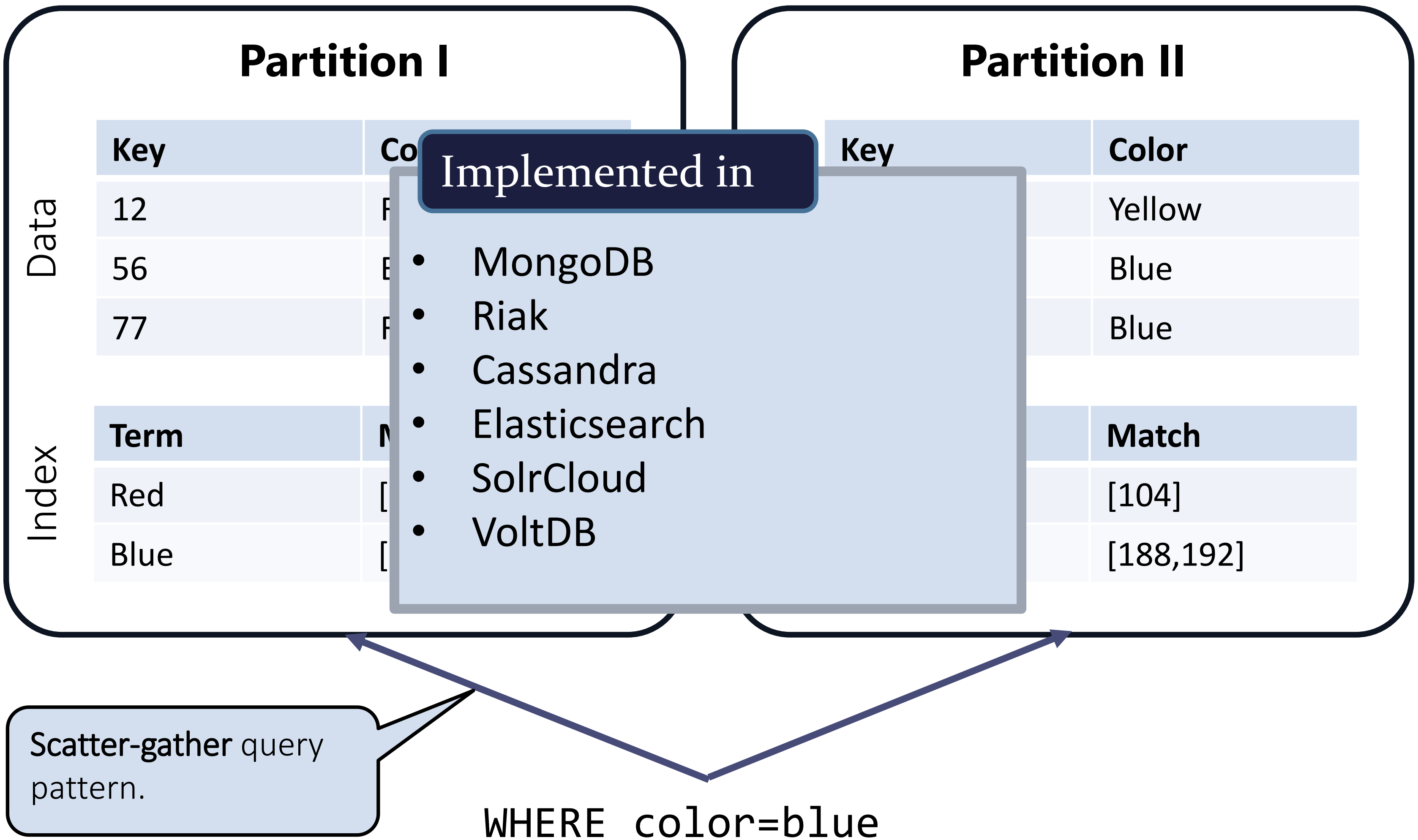
In a Nutshell




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

Local Secondary Indexing

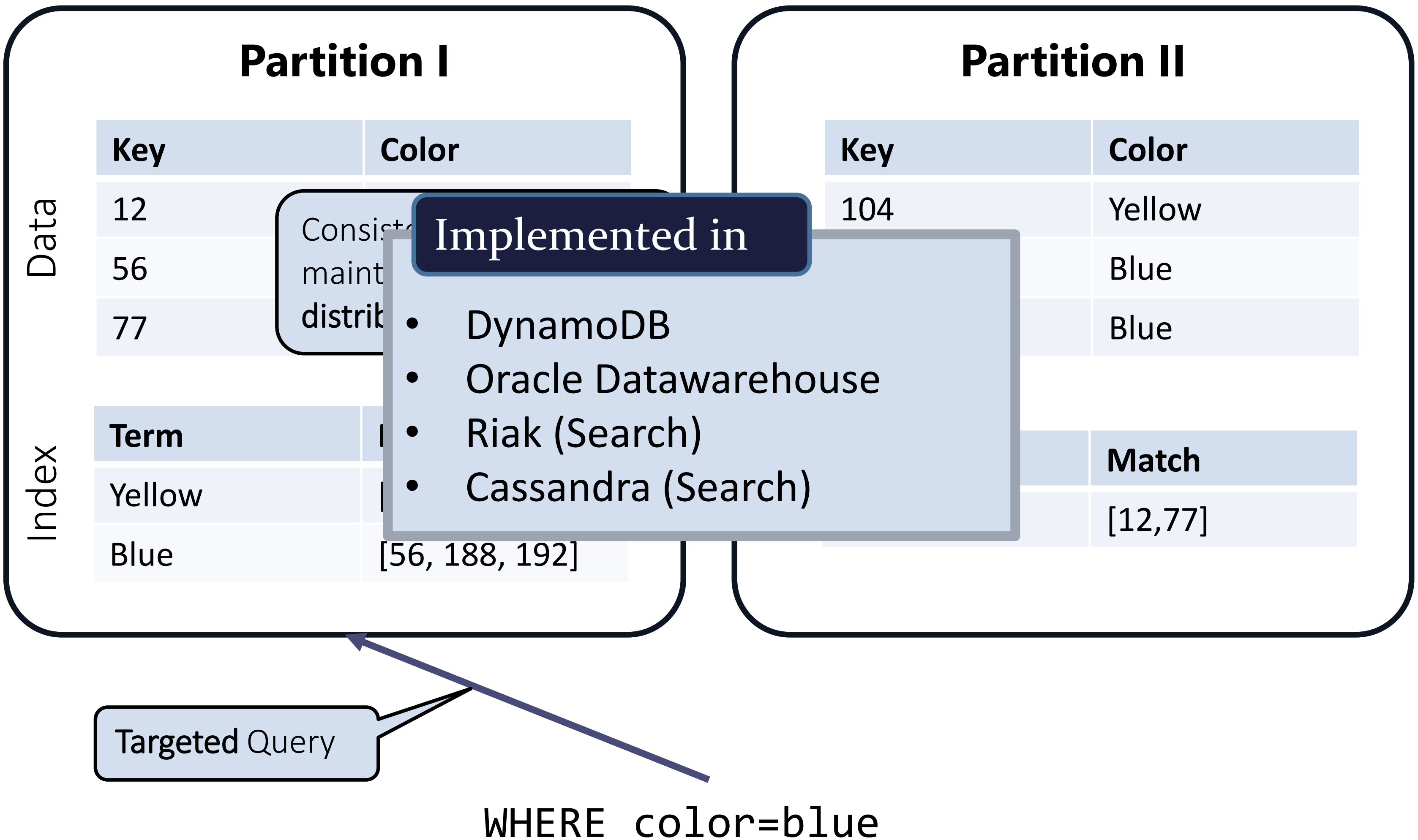
Partitioning By Document



Kleppmann, Martin. "Designing data-intensive applications." (2016).

Global Secondary Indexing

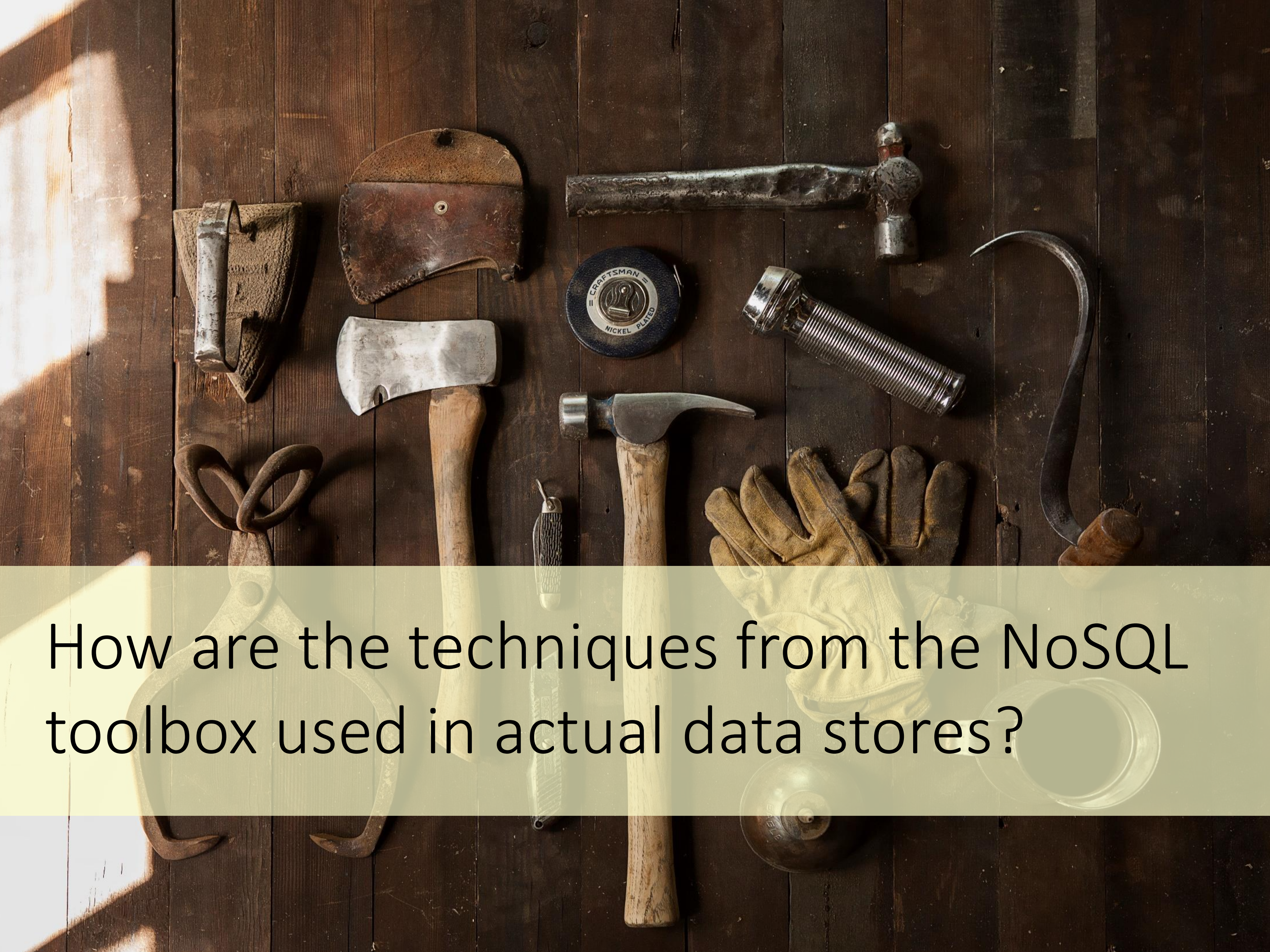
Partitioning By Term



Kleppmann, Martin. "Designing data-intensive applications." (2016).

Query Processing Techniques
Summary

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



How are the techniques from the NoSQL toolbox used in actual data stores?

Outline

NoSQL Foundations and Motivation

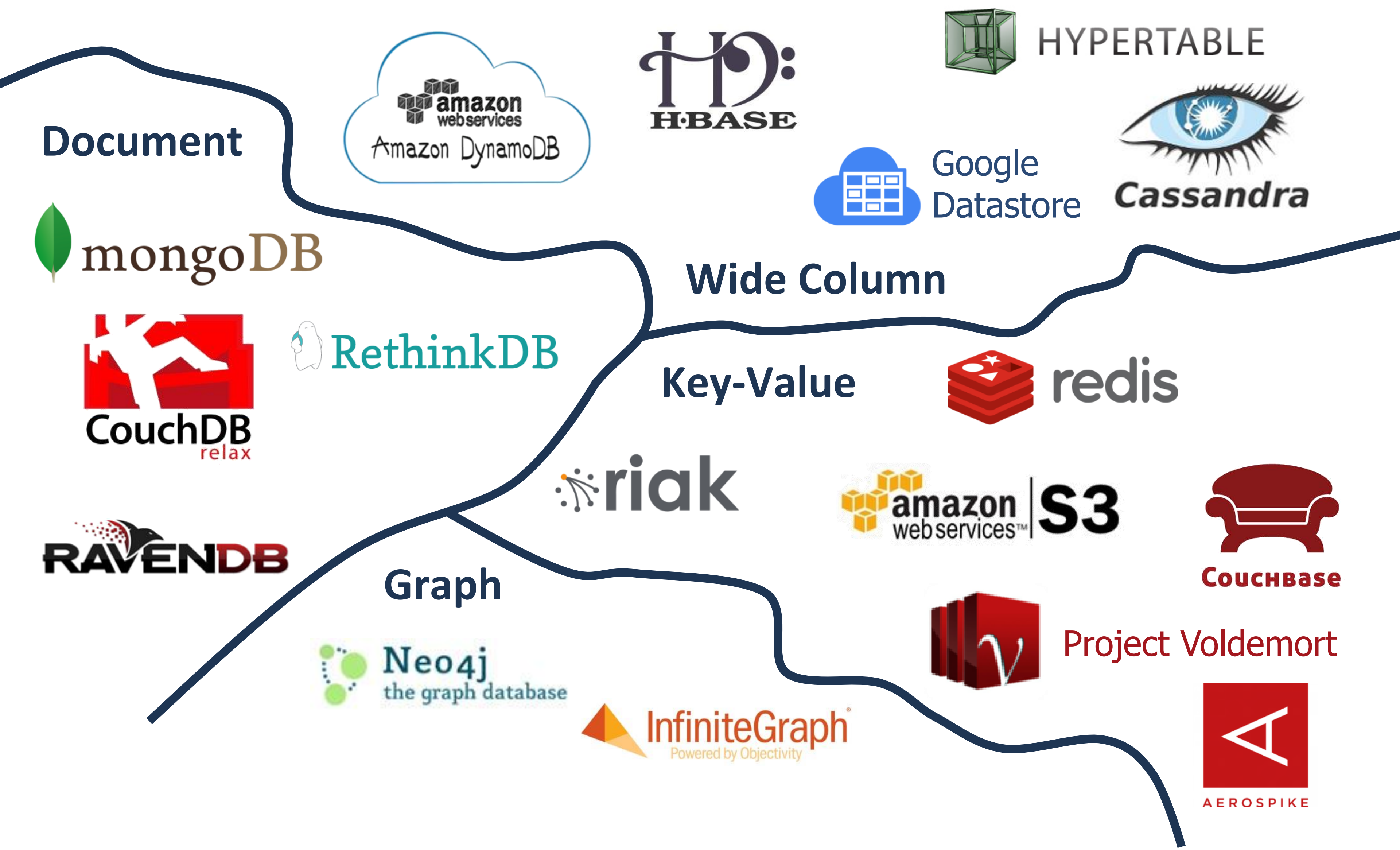
The NoSQL Toolbox: Common Techniques

NoSQL Systems & Decision Guidance

Scalable Real-Time Databases and Processing

- Overview & Popularity
- Core Systems:
  - Dynamo
  - BigTable
- Riak
- HBase
- Cassandra
- Redis
- MongoDB

NoSQL Landscape



Popularity (Feb 2019)

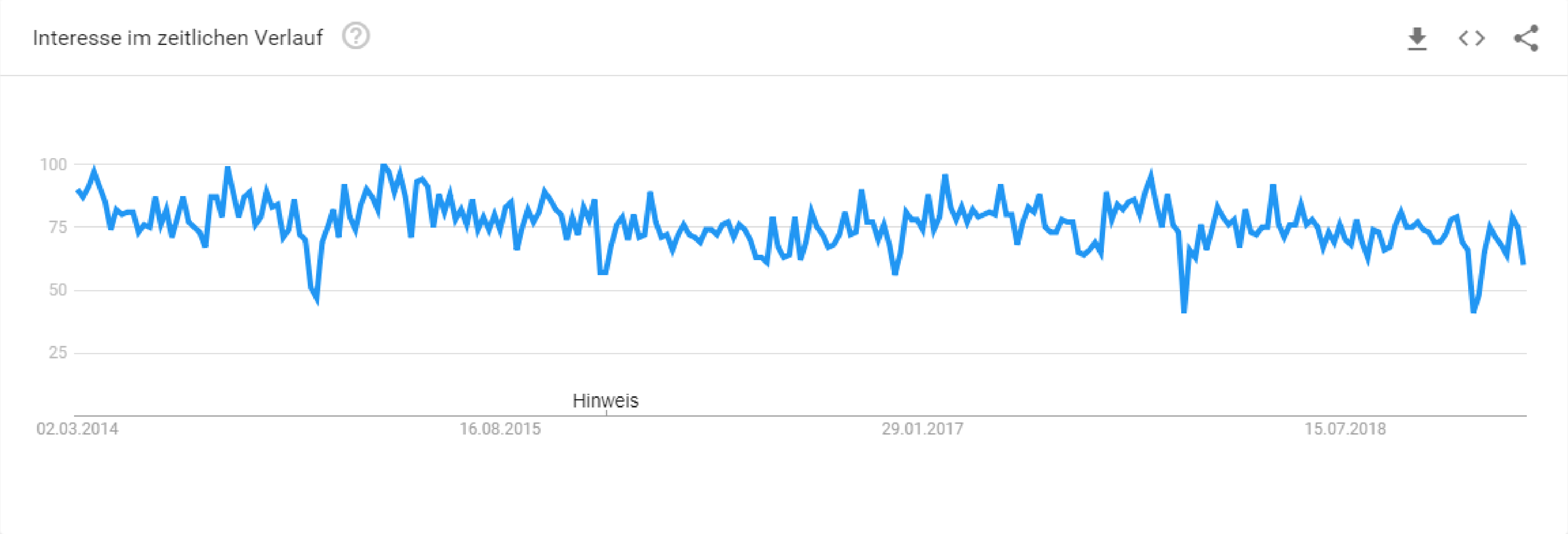
#	System	Model
1.	Oracle	Relational DBMS
2.	MySQL	Relational DBMS
3.	MS SQL Server	Relational DBMS
4.	PostgreSQL	Relational DBMS
5.	MongoDB	Document store
6.	DB2	Relational DBMS
7.	Microsoft Access	Relational
8.	Redis	Key-value store
9.	ElasticSearch	Search engine
10.	SQLite	Relational DBMS

11.	Cassandra	Wide column store
12.	MariaDB	Relational DBMS
13.	Splunk	Search engine
14.	Teradata	Search engine
15.	Hive	Relational
16.	Solr	Relational DBMS
17.	HBase	Relational DBMS
18.	FileMaker	Relational
19.	SAP Adaptive Server	Relational DBMS
20.	SAP HANA	Relational DBMS
21.	Amazon DynamoDB	Multi-model
22.	Neo4j	Graph DB
23.	Couchbase	Document store
24.	Memcached	Key-value store
25.	SQL Azure	Multi-model

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

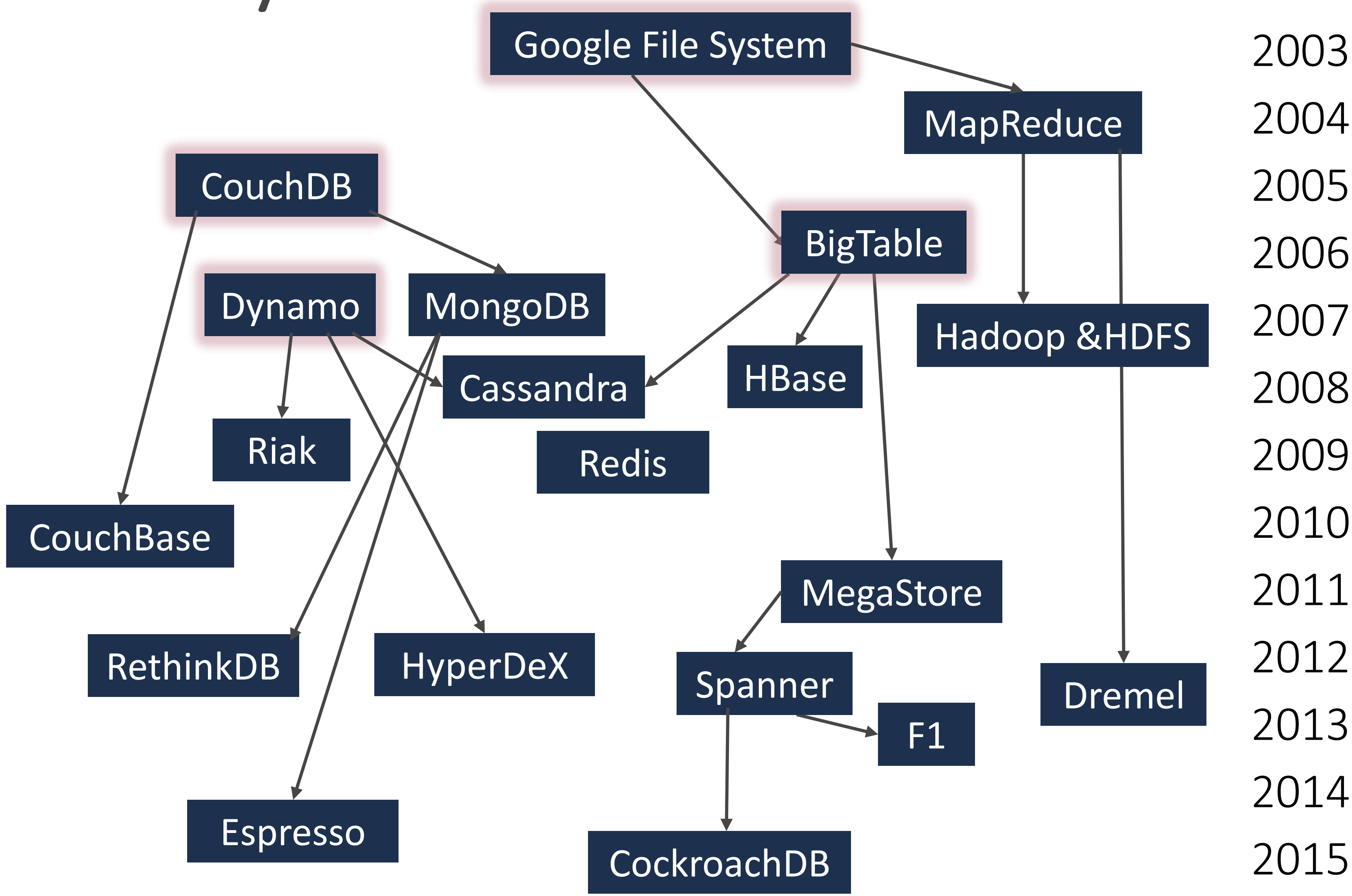
http://db-engines.com/de/ranking

NoSQL: Still a Thing in 2019



https://trends.google.com/trends/explore?date=today%205-y&q=nosql

History


Google File System
MapReduce
CouchDB
BigTable
Dynamo
MongoDB
Hadoop &HDFS
Cassandra
HBase
Riak
Redis
CouchBase
MegaStore
RethinkDB
HyperDeX
Spanner
Dremel
F1
Espresso
CockroachDB
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015

NoSQL foundations

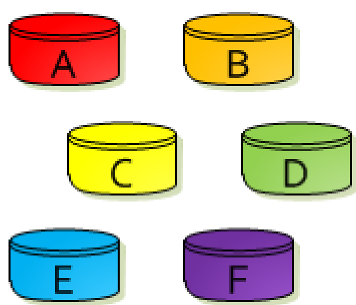
- **BigTable** (2006, Google)
  - **C**onsistent, **P**artition Tolerant
  - Wide-Column data model
  - Master-based, fault-tolerant, large clusters (1.000+ Nodes), **HBase**, **Cassandra**, **HyperTable, Accumolo**
- **Dynamo** (2007, Amazon)
  - **A**vailable, **P**artition tolerant
  - Key-Value interface
  - Eventually Consistent, always writable, fault-tolerant
  - **Riak**, **Cassandra**, **Voldemort**, **DynamoDB**

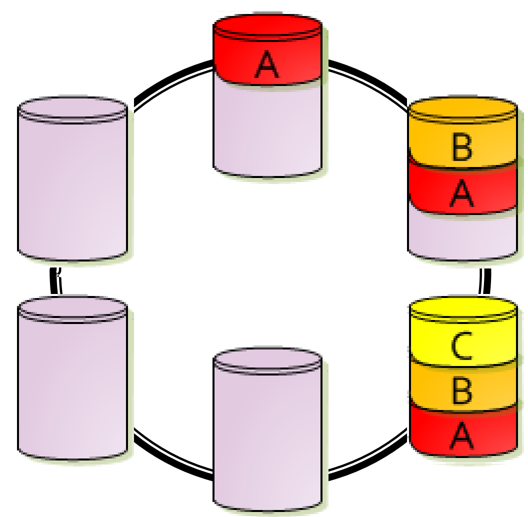
Chang, Fay, et al. "Bigtable: A distributed storage system for structured data."

DeCandia, Giuseppe, et al. "Dynamo: Amazon's highly available key-value store."

Dynamo (AP)

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

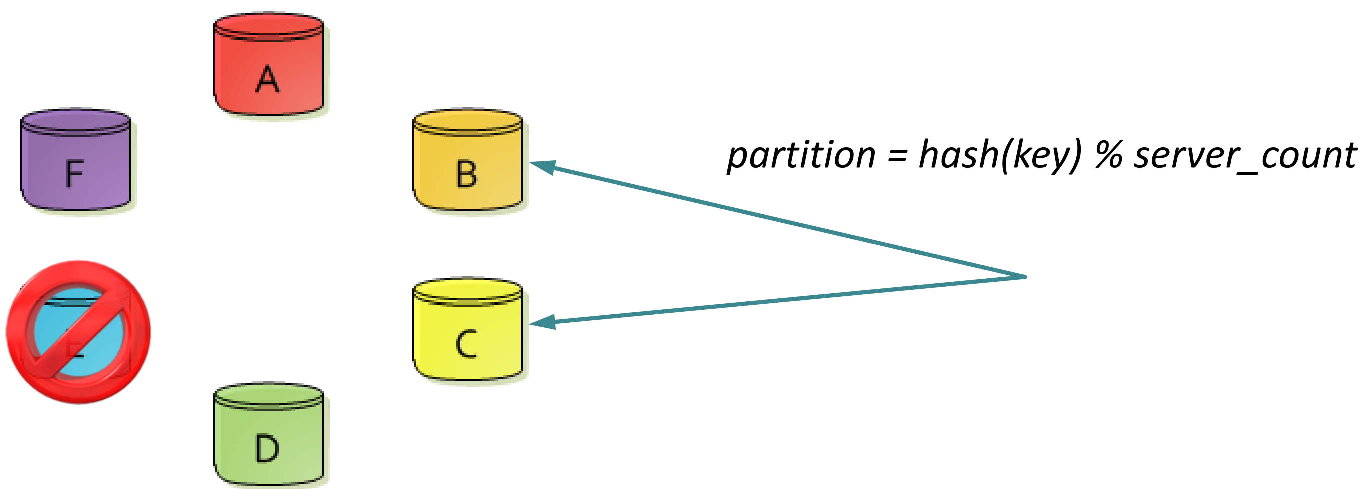




DeCandia, Giuseppe, et al. "Dynamo: Amazon's highly available key-value store."

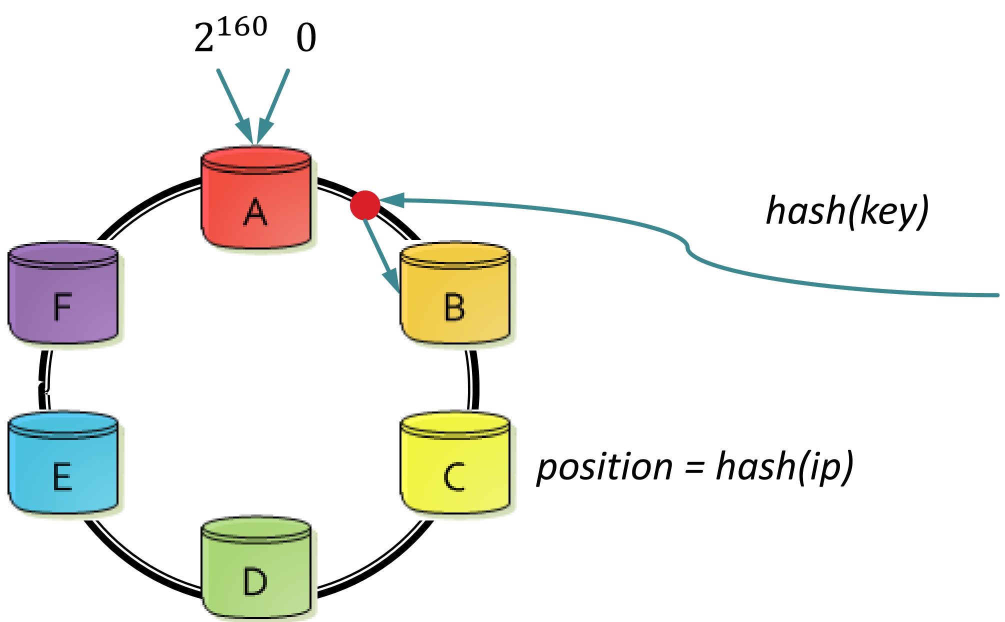
Consistent Hashing

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



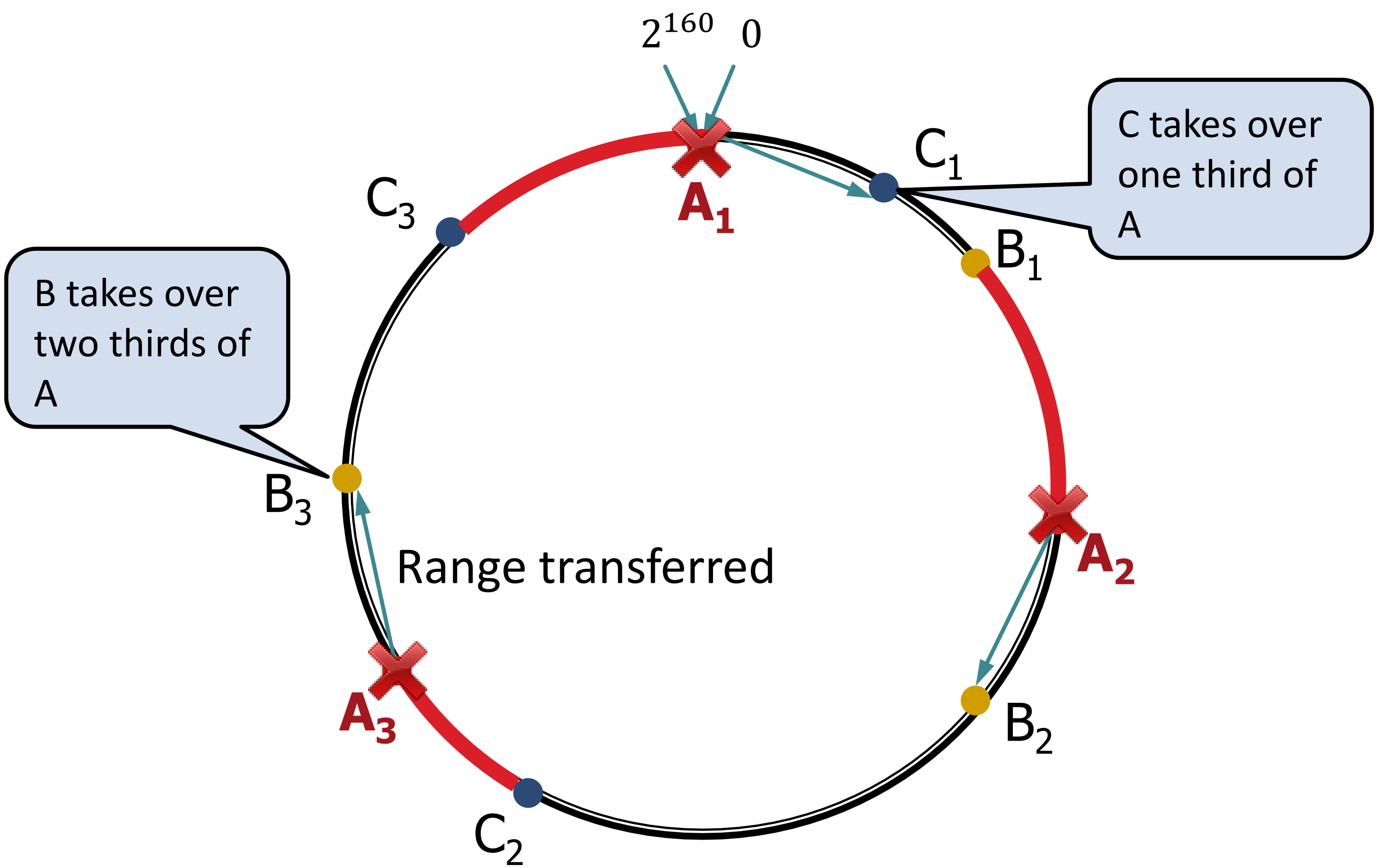
Consistent Hashing

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



Consistent Hashing

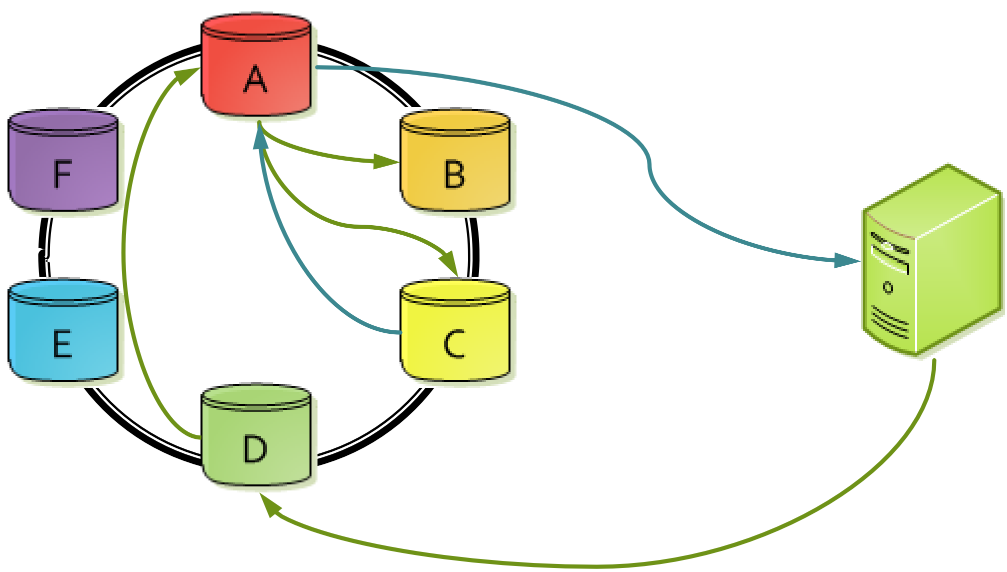
- Extension: **Virtual Nodes** for Load Balancing



Reading

Parameters R, W, N

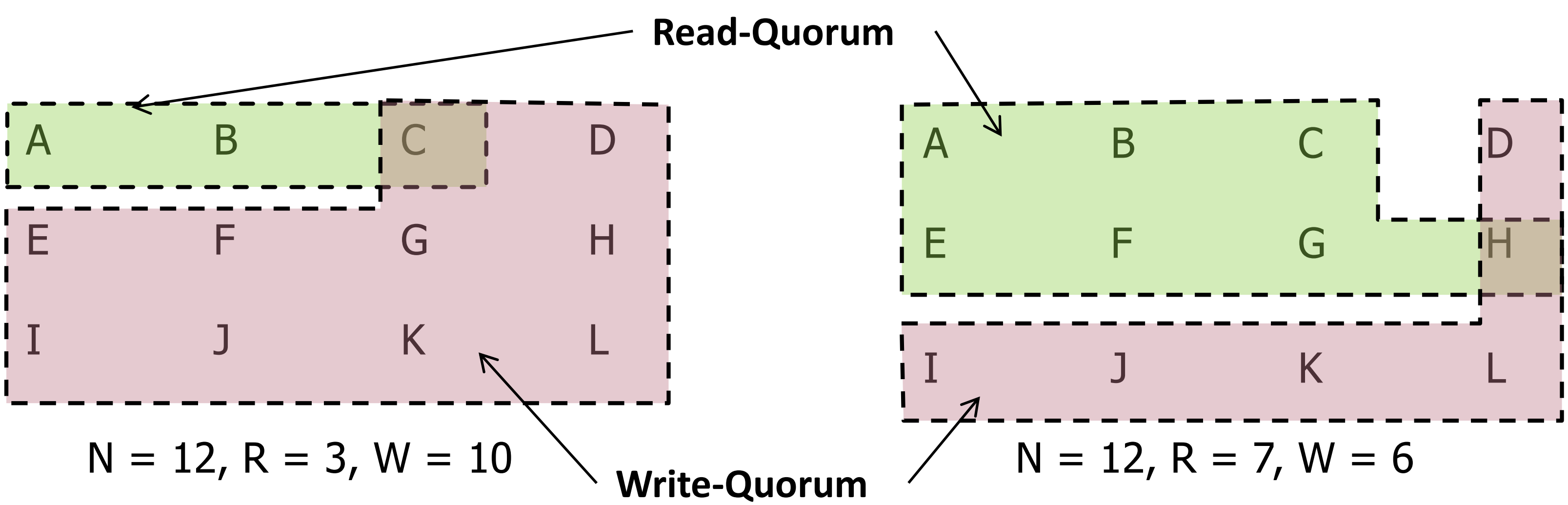
- An arbitrary node acts as a coordinator
- **N**: number of replicas
- **R**: number of nodes that need to confirm a read
- **W**: number of nodes that need to confirm a write



N=3
R=2
W=1

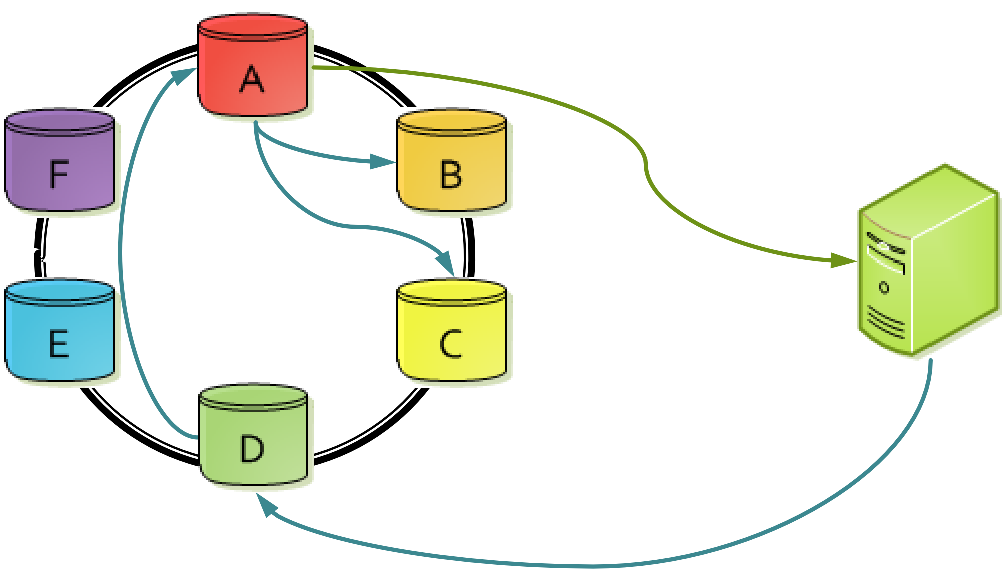
Quorums

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



Writing

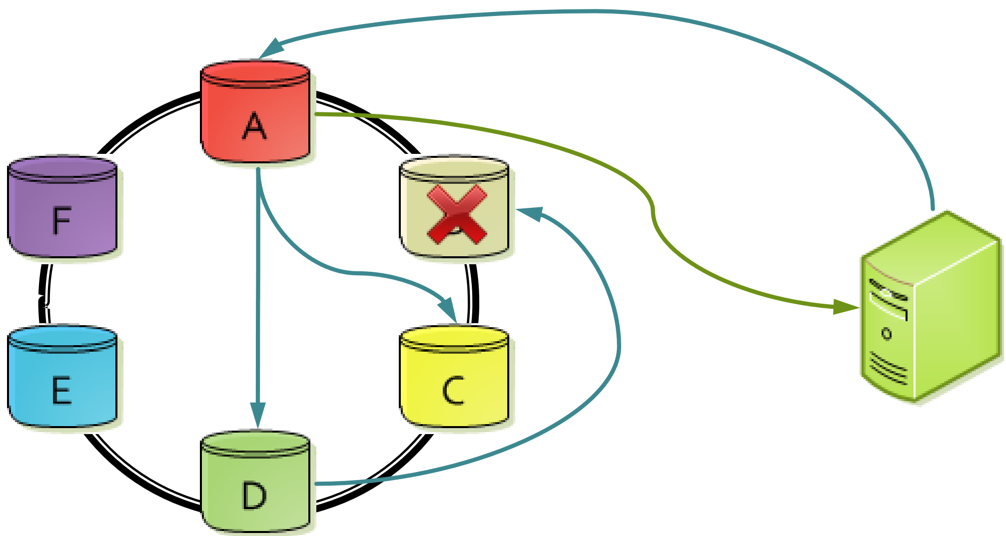
- **W** Servers have to acknowledge



N=3
R=2
W=1

Hinted Handoff

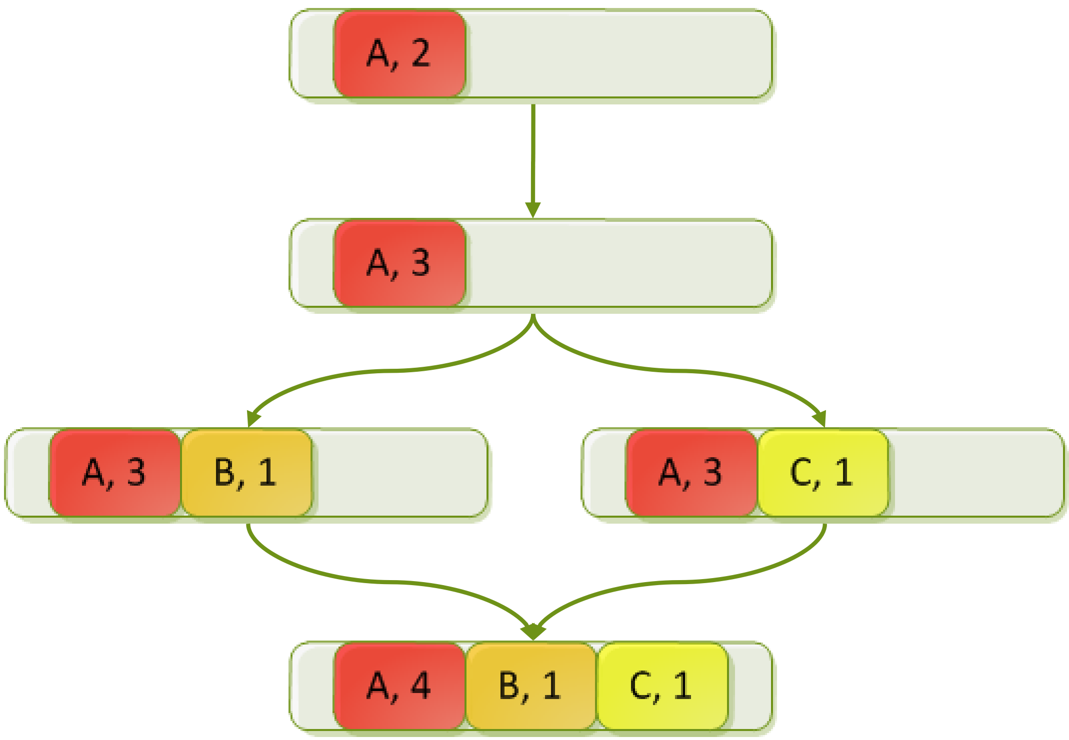
- Next node in the ring may take over, until original node is available again:



N=3
R=2
W=1

Vector clocks

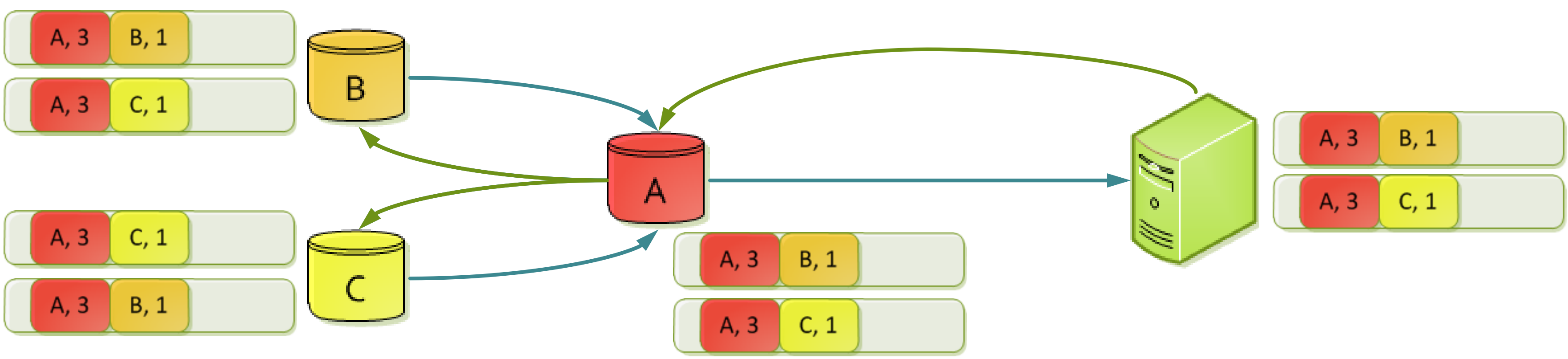
- Dynamo uses **Vector Clocks** for versioning



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

Versioning and Consistency

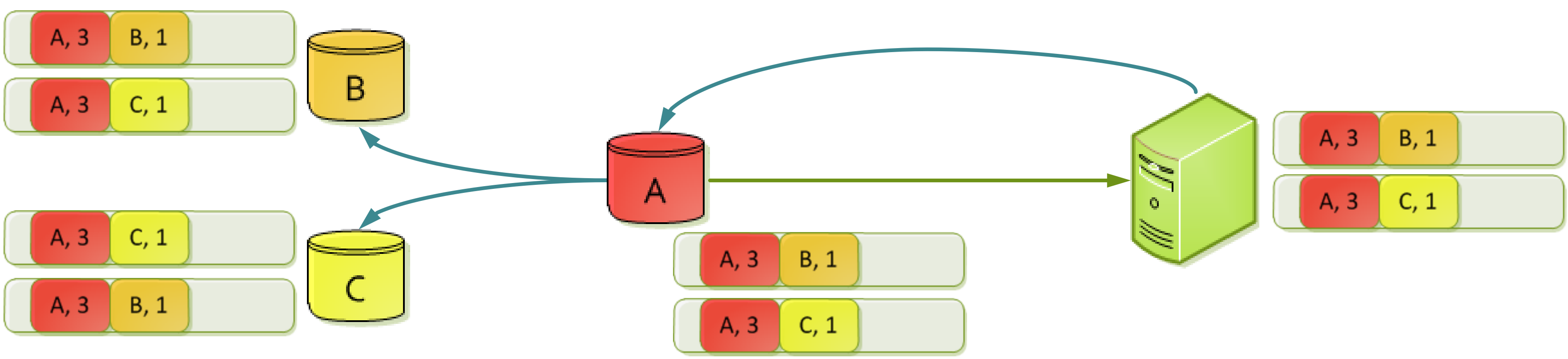
- $R + W \leq N \Rightarrow$ no consistency guarantee
- $R + W > N \Rightarrow$ newest acked value included in reads
- **Vector Clocks** used for versioning



Read Repair

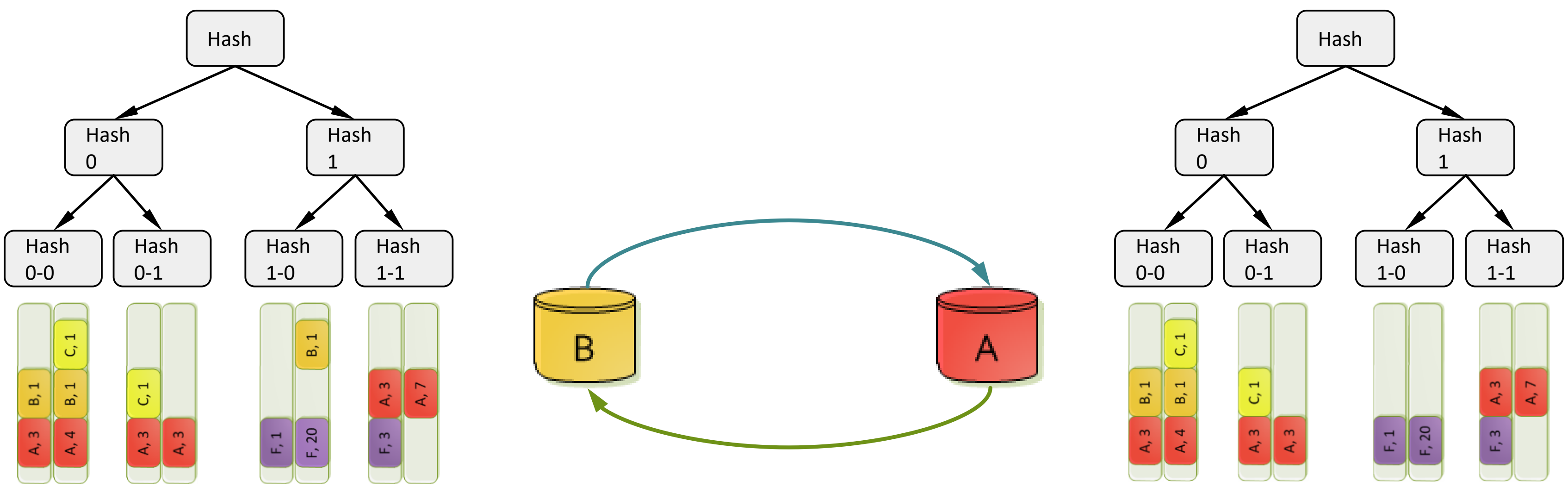
Conflict Resolution

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



Merkle Trees: Anti-Entropy

- Every Second: Contact random server and compare



Quorum

- Typical Configurations:

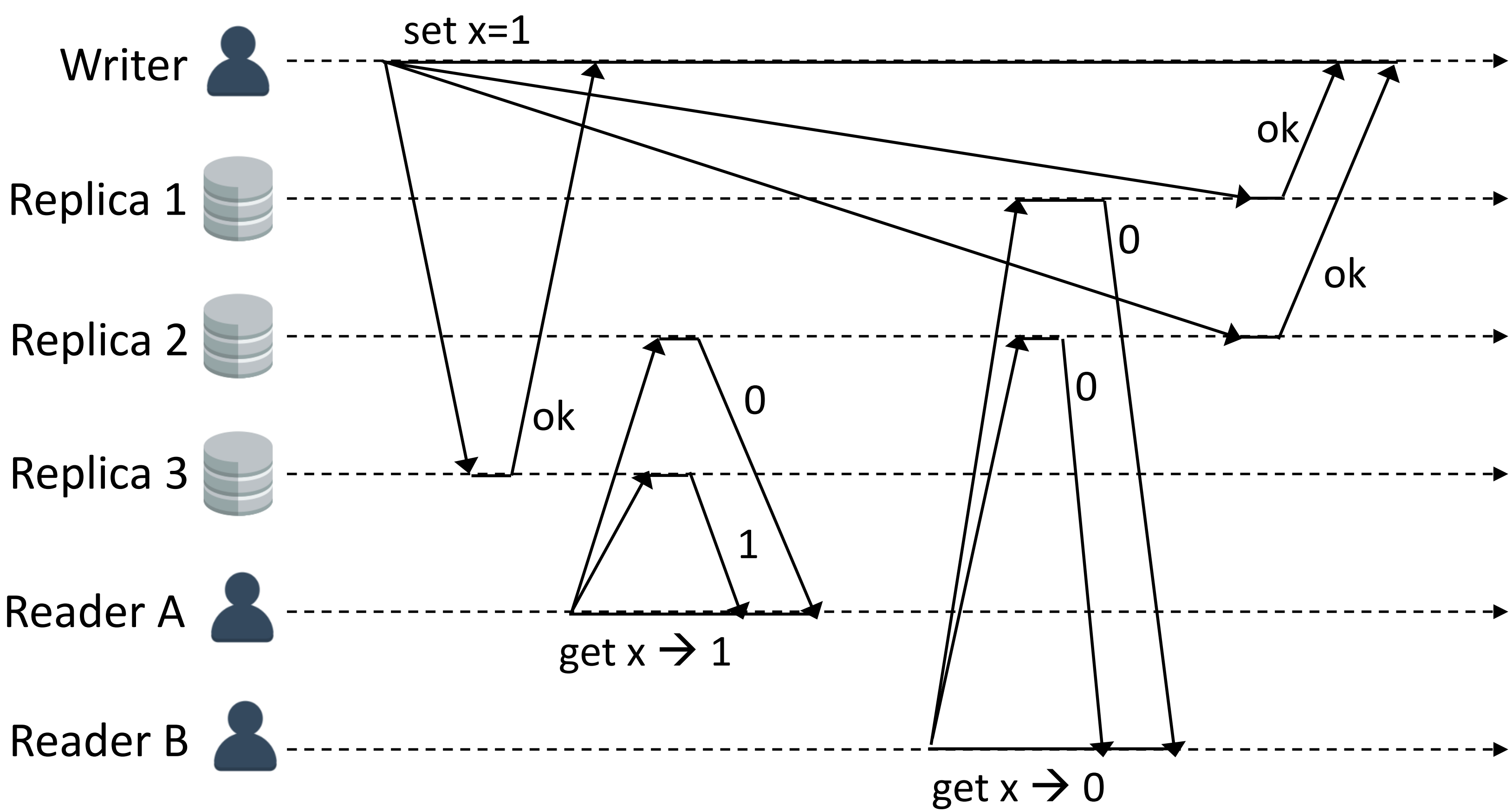
LinkedIn (SSDs):
$P(consistent) \geq 99.9\%$
nach $1.85\ ms$

Performance (Cassandra Default)	N=3, R=1, W=1
Quorum, fast Writing:	N=3, R=3, W=1
Quorum, fast Reading	N=3, R=1, W=3
Trade-off (Riak Default)	N=3, R=2, W=2

P. Bailis, PBS Talk: http://www.bailis.org/talks/twitter-pbs.pdf

$R + W > N$ does not imply linearizability

- Consider the following execution:

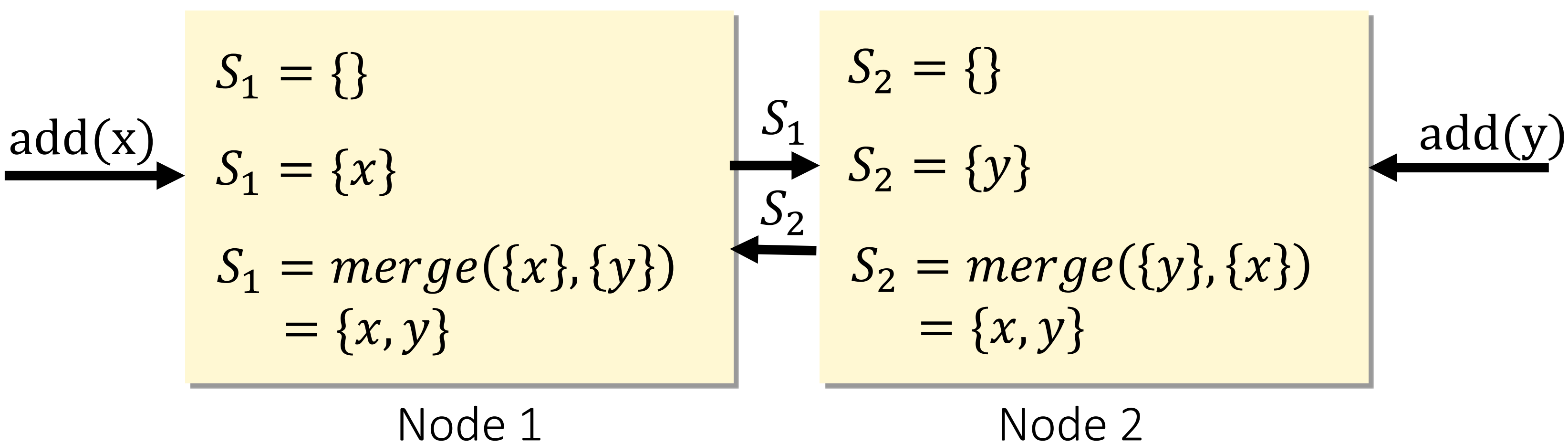


Kleppmann, Martin. "Designing data-intensive applications." (2016).

CRDTs

Convergent/Commutative Replicated Data Types

- **Goal**: avoid manual conflict-resolution
- **Approach**:
  - **State-based** – commutative, idempotent merge function
  - **Operation-based** – broadcasts of commutative upates
- Example: State-based Grow-only-Set (G-Set)

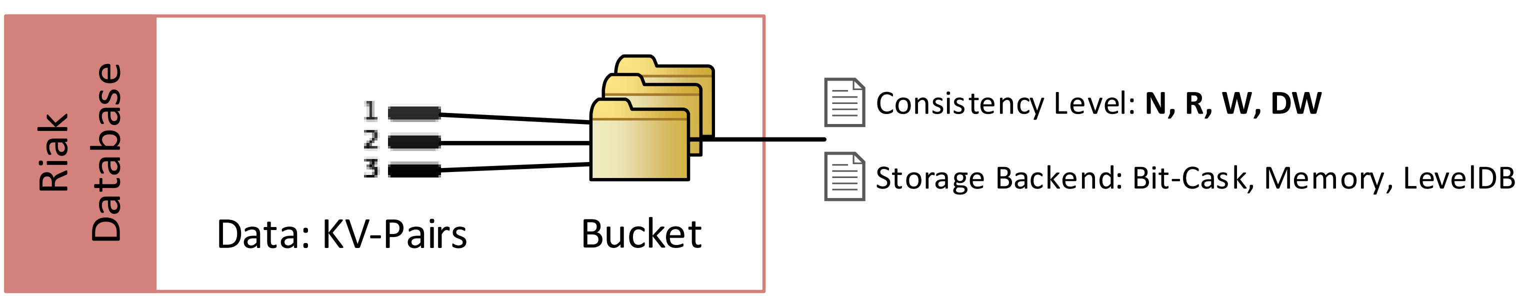


Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski "Conflict-free Replicated Data Types"

Riak (AP)

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

- Open-Source Dynamo-Implementation
- Extends Dynamo:
  - Keys are grouped to **Buckets**
  - KV-pairs may have **metadata** and **links**
  - Map-Reduce support
  - Secondary Indices, Update Hooks, Solr Integration
  - Option for **strongly consistent** buckets (experimental)
  - **Riak CS:** S3-like file storage, **Riak TS**: time-series database



Riak Data Types

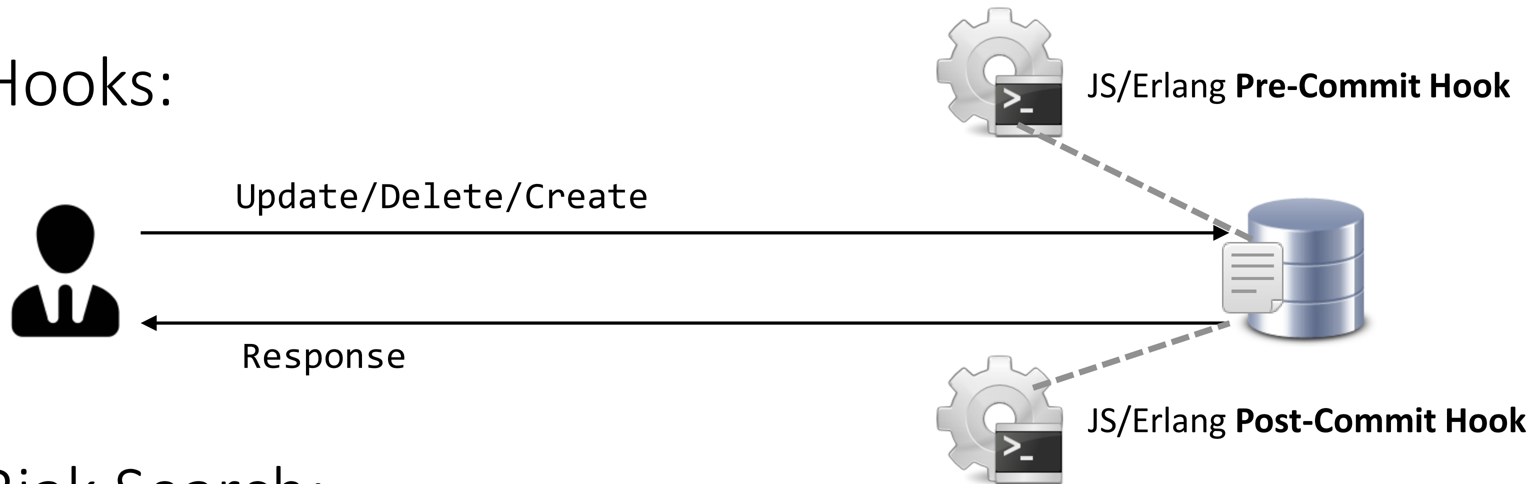
- Implemented as *state-based CRDTs*:

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

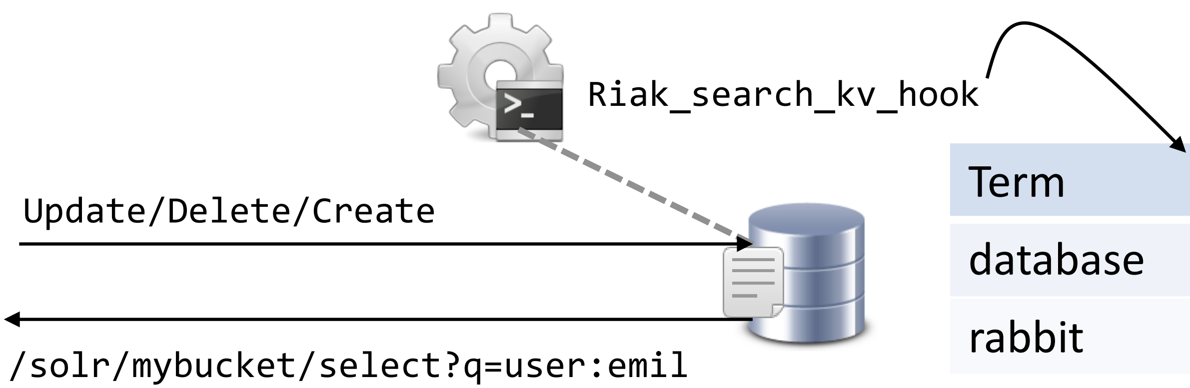
http://docs.basho.com/riak/kv/2.1.4/learn/concepts/crdts/

Hooks & Search

- Hooks:



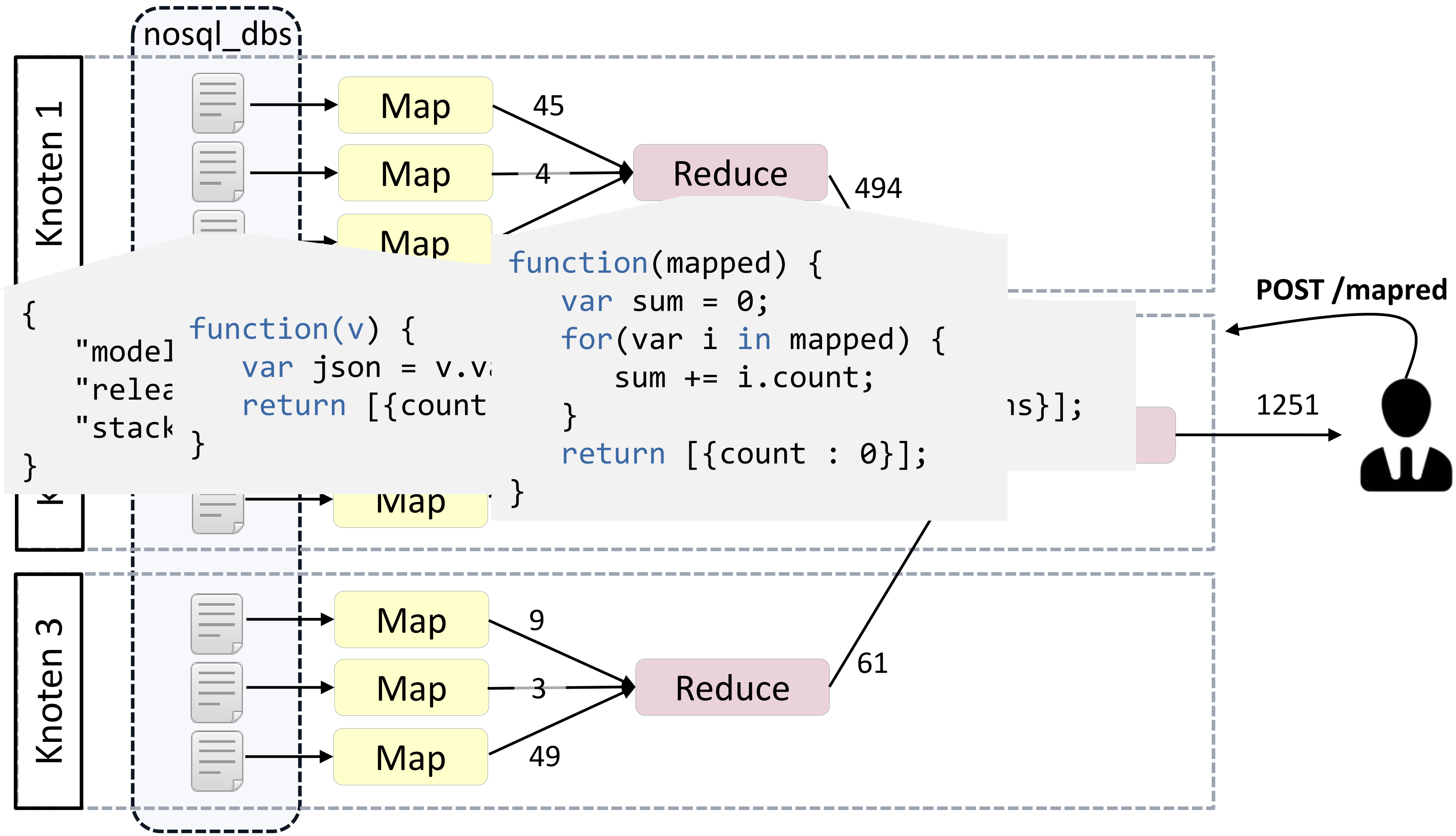
- Riak Search:



Term	Dokument
database	3,4,1
rabbit	2

Search Index

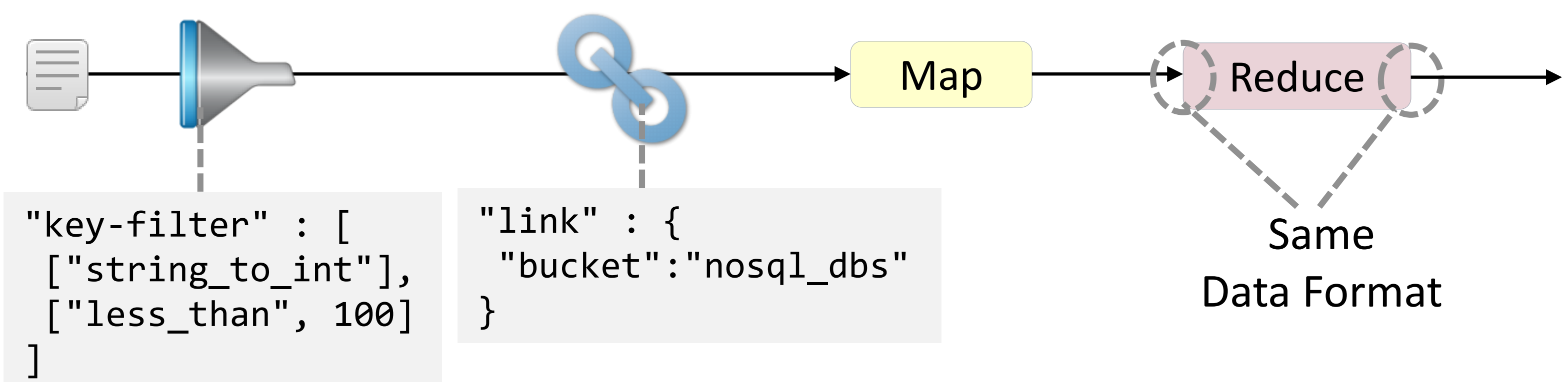
Riak Map-Reduce



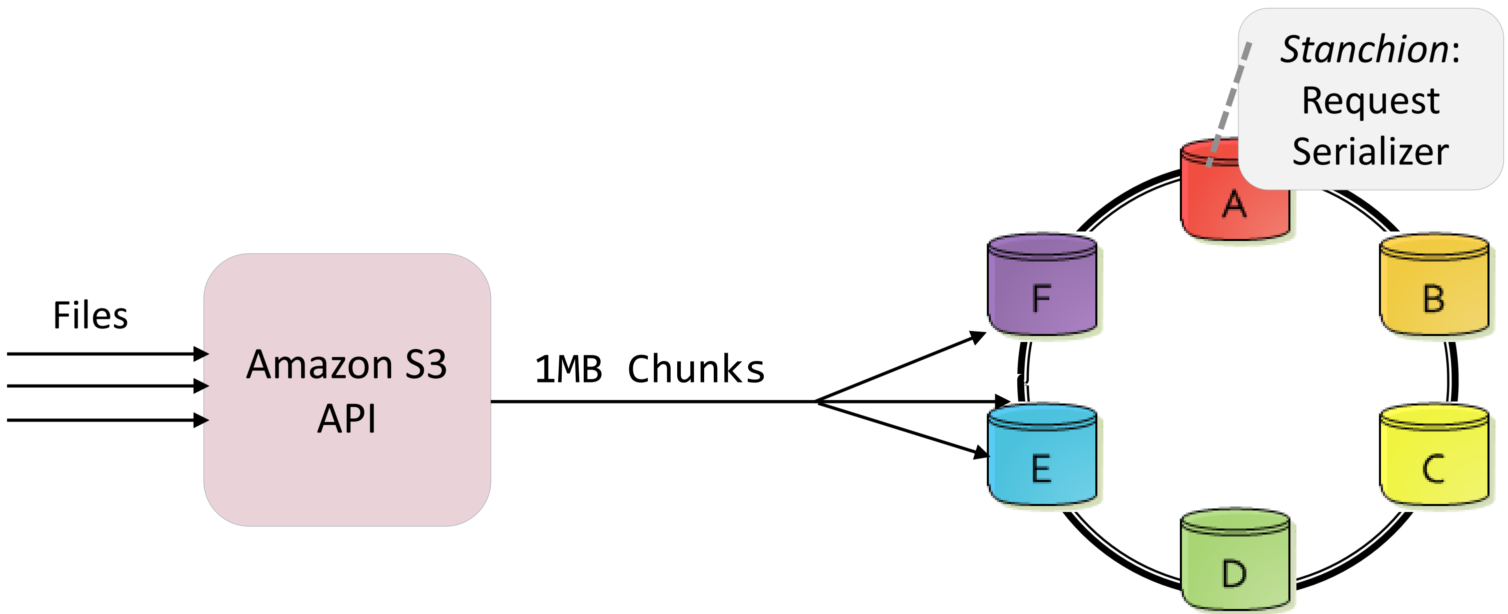
http://docs.basho.com/riak/latest/tutorials/querying/MapReduce/

Riak Map-Reduce

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



Riak Cloud Storage



Summary: Dynamo and Riak

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

Dynamo and Riak

Classification

Sharding	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared Disk
Replication	Trans-action Protocol	Sync. Replica-tion	Async. Replica-tion	Primary Copy	Update Anywhere
Storage Management	Logging	Update-in-Place	Caching	In-Memory	Append-Only Storage
Query Processing	Global Index	Local Index	Query Planning	Analytics	Materialized Views

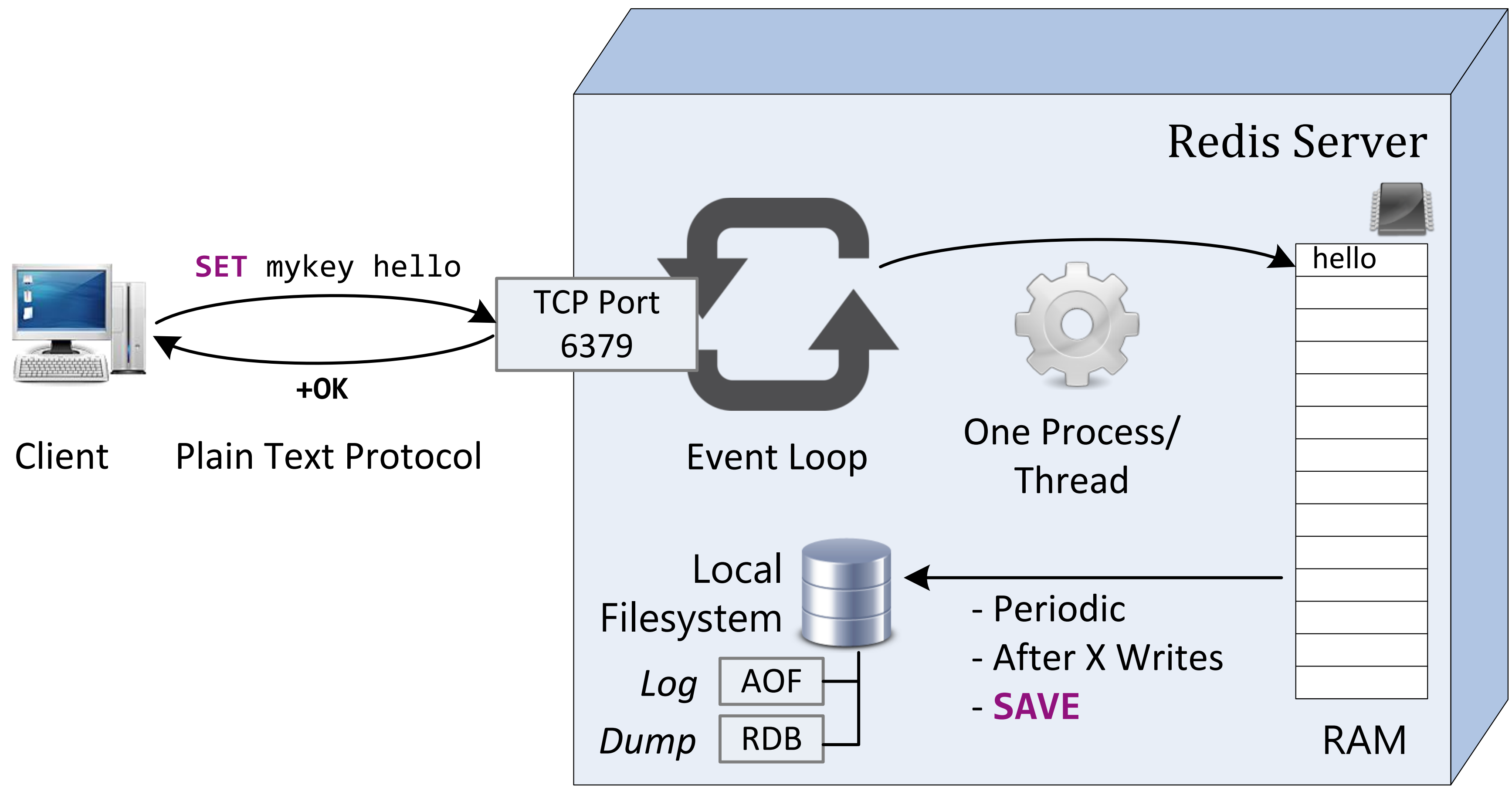
Redis (CA)

Redis
Model:
Key-Value
License:
BSD
Written in:
C

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

Redis Architecture

- Redis Codebase ≅ 20K LOC

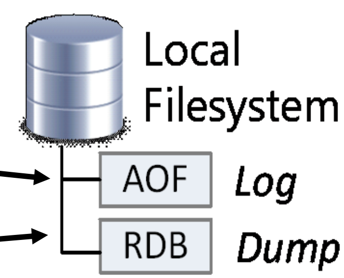


Persistence

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

```
config set appendonly everysec
```

`fsync()` every second



Snapshot every 60s,
if > 1000 keys changed

```
config set save 60 1000
```

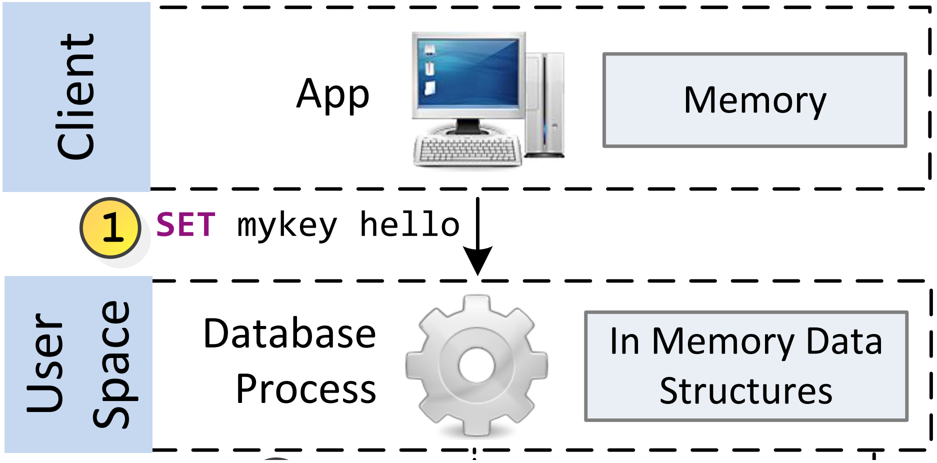
Persistence

1. Resistence to client crashes

2. Resistence to DB process crashes

3. Resistence to hardware crashes with *Write-Through*

4. Resistence to hardware crashes with *Write-Back*

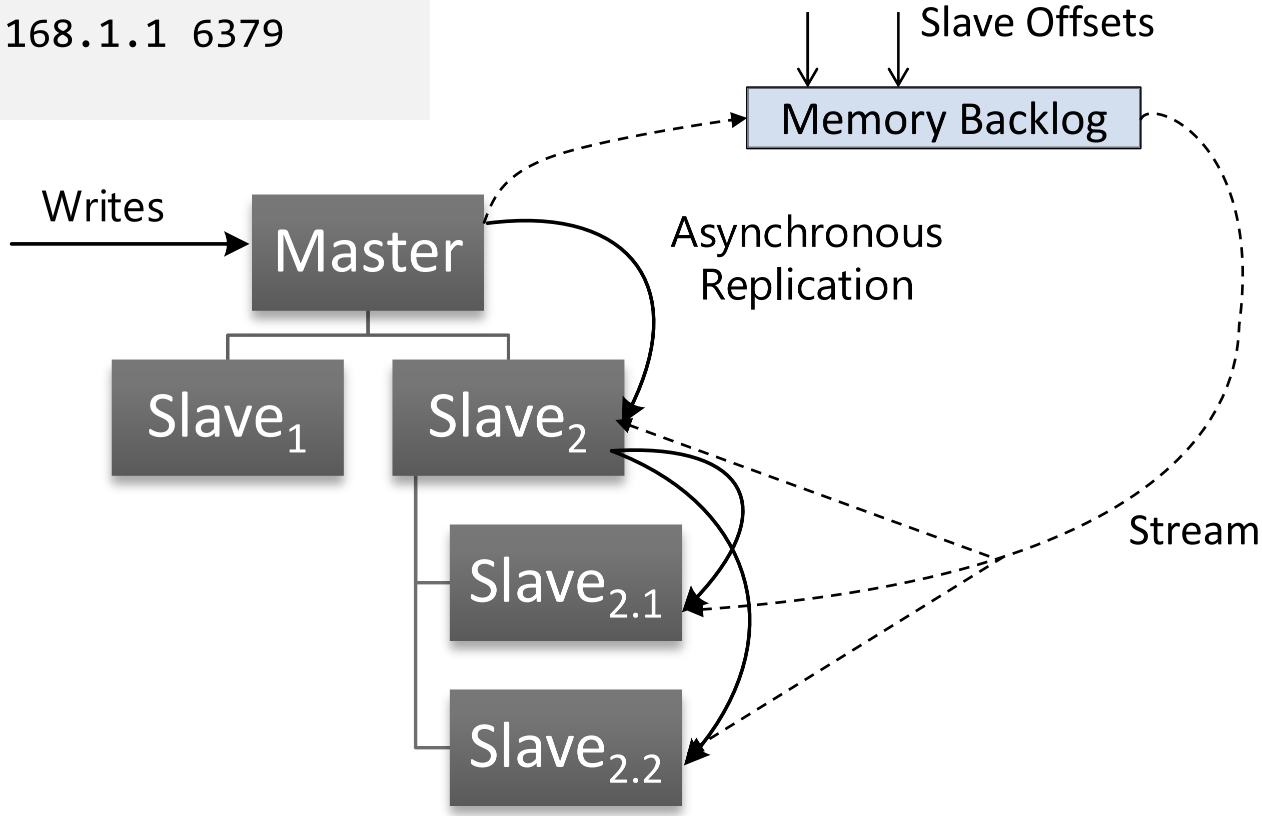


Persistence: Redis vs an RDBMS

PostgreSQL:	Redis:
> `synchronous_commit on`	> `appendfsync always`
Latency > Disk Latency, Group Commits, Slow	
> `synchronous_commit off`	> `appendfsync everysec`
periodic `fsync()`, data loss limited	
> `fsync false`	> `appendfsync no`
Data corruption and losspossible	Data loss possible, corruption prevented
> `pg_dump`	> `save` oder `bgsave`

Master-Slave Replication

```
> SLAVEOF 192.168.1.1 6379
< +OK
```



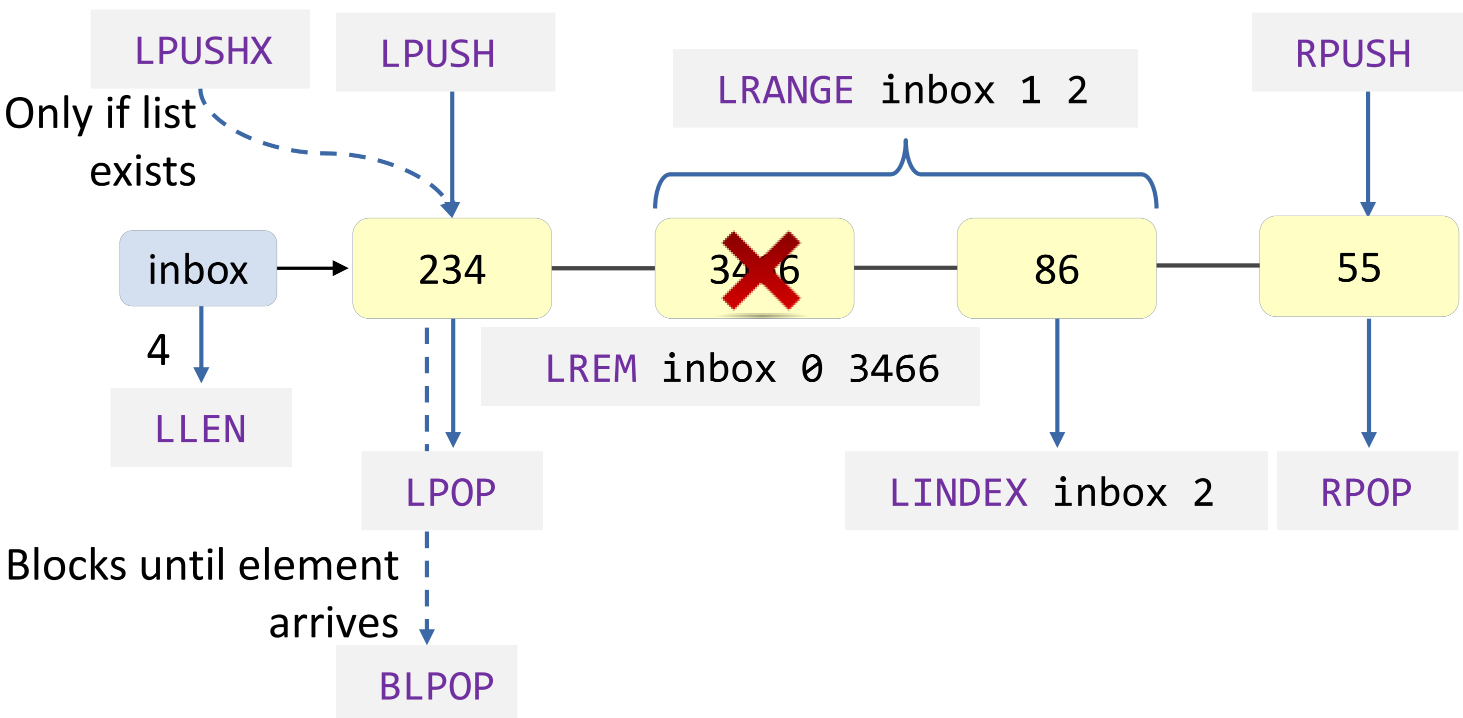
Data structures

- String, List, Set, Hash, Sorted Set

String	web:index →	"<html><head>…"
Set	users:2:friends →	{23, 76, 233, 11}
List	users:2:inbox →	[234, 3466, 86,55]
Hash	users:2:settings →	Theme → "dark", cookies → "false"
Sorted Set	top-posters →	466 → "2", 344 → "16"
Pub/Sub	users:2:notifs →	"{event: 'comment posted', time : …"

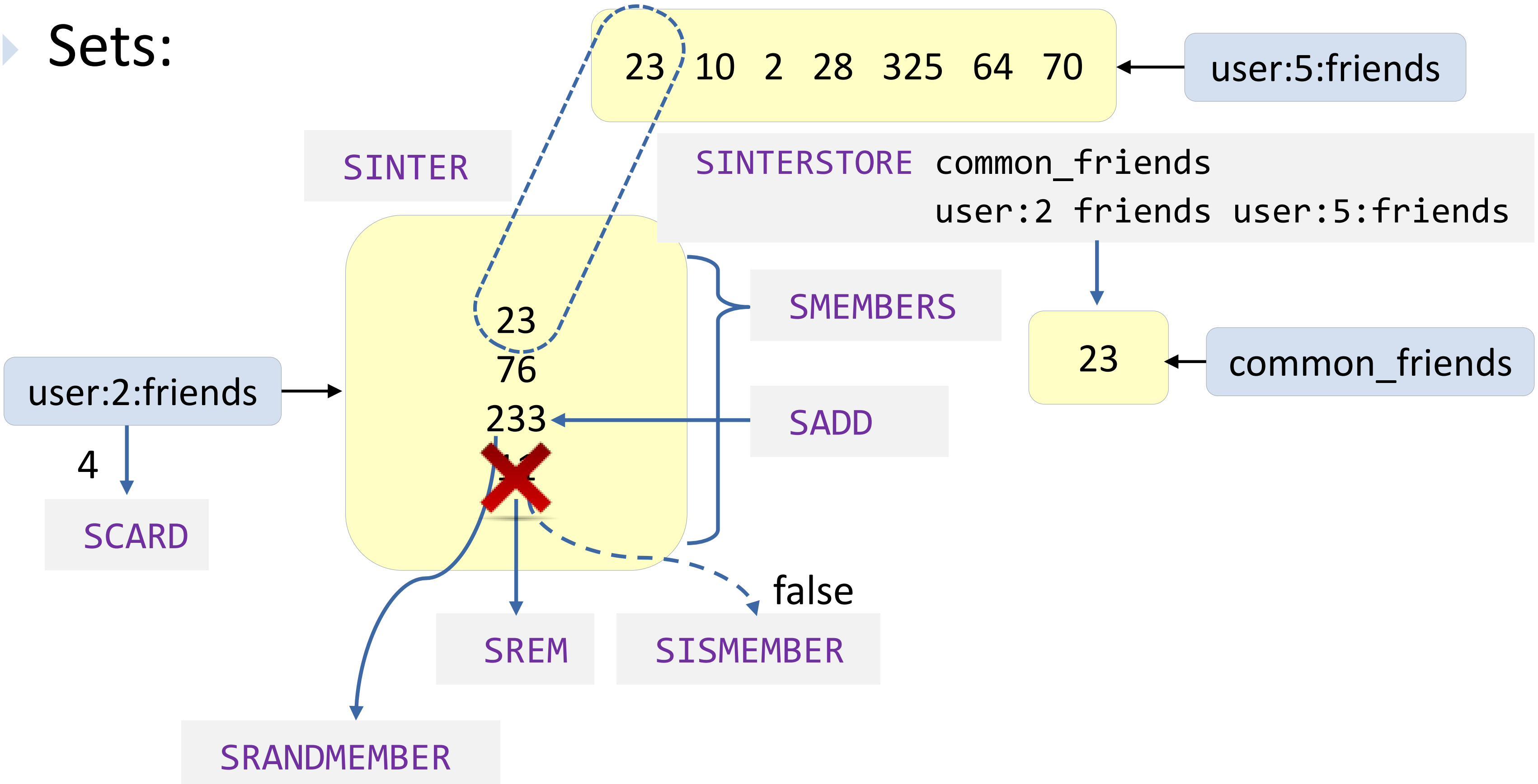
Data Structures

- (Linked) Lists:



Data Structures

- Sets:



Data Structures

- Pub/Sub: users:2:notifs → "{event: 'comment posted', time : …"

redis

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

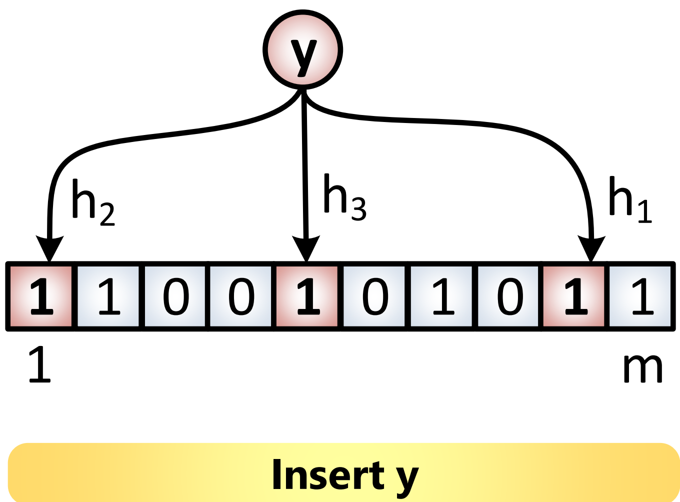
```
SUBSCRIBE user:2:notifs
```

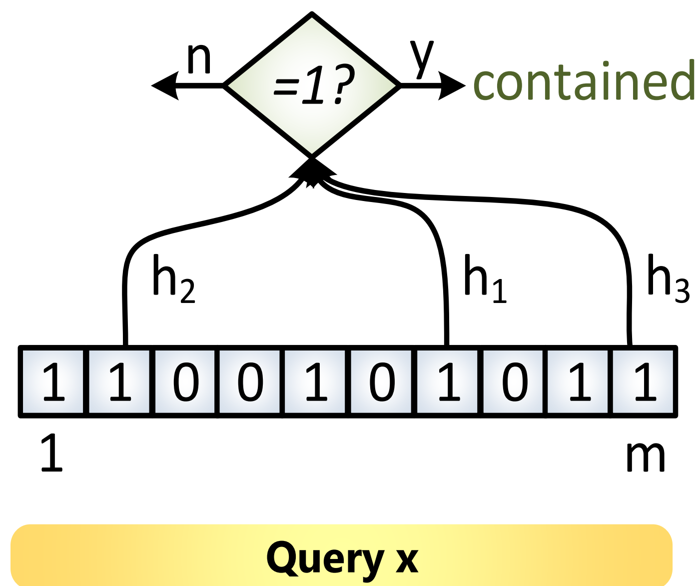
```
{
   event: 'comment posted',
   time : …
}
```

Example: Bloom filters

Compact Probabilistic Sets

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





https://github.com/Baqend/Orestes-Bloomfilter

Bloomfilters in Redis

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

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

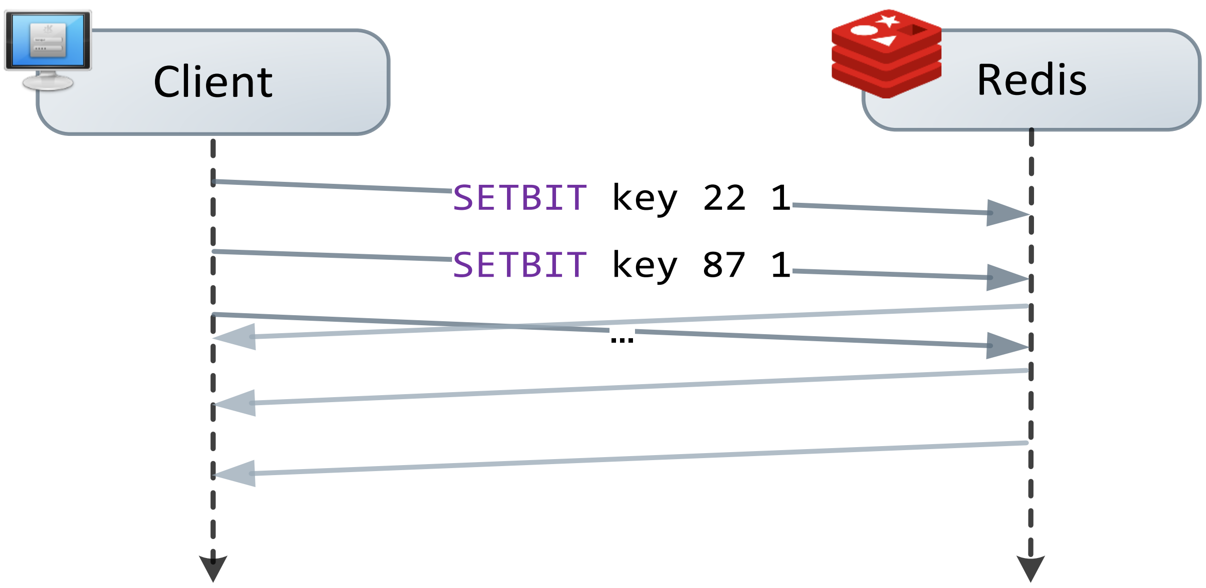
Jedis: Redis Client for Java

SETBIT creates and resizes automatically

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

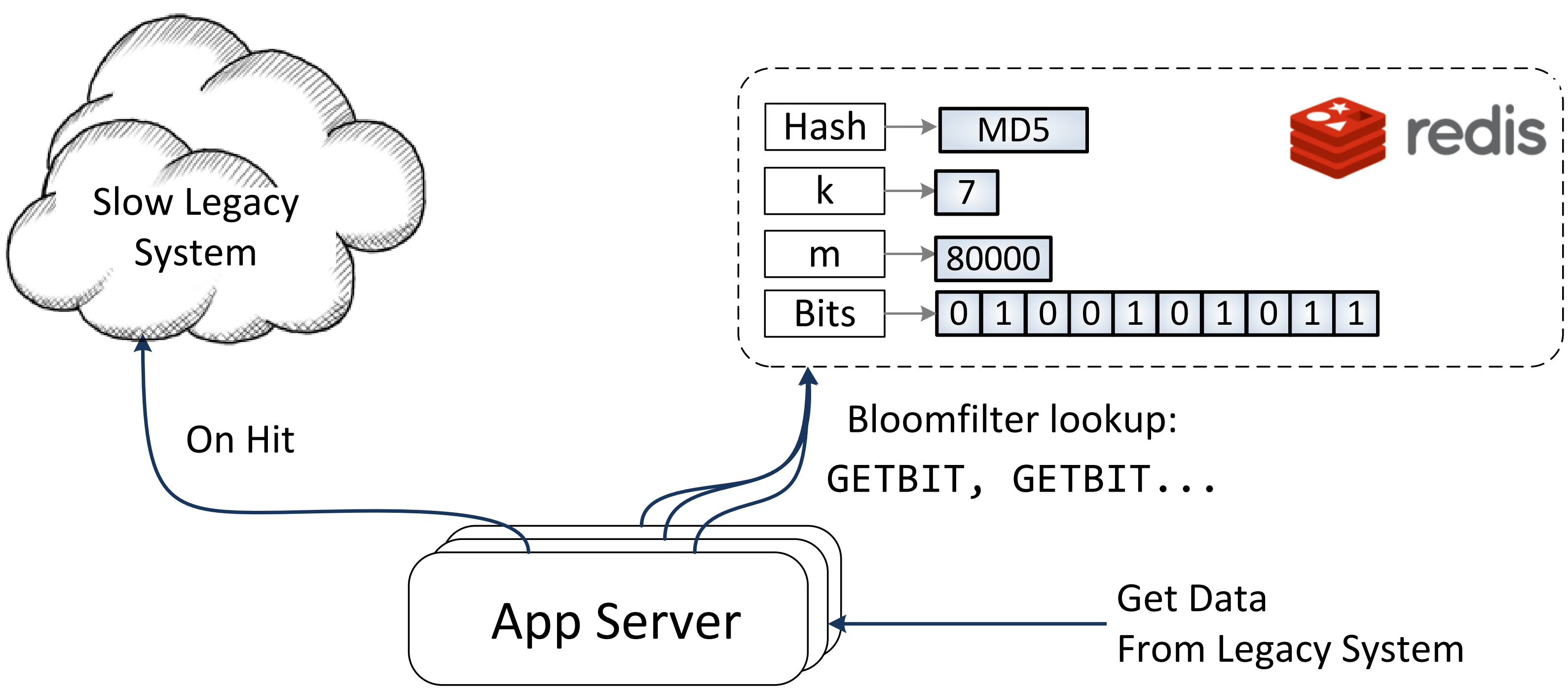
Pipelining

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



Redis for distributed systems

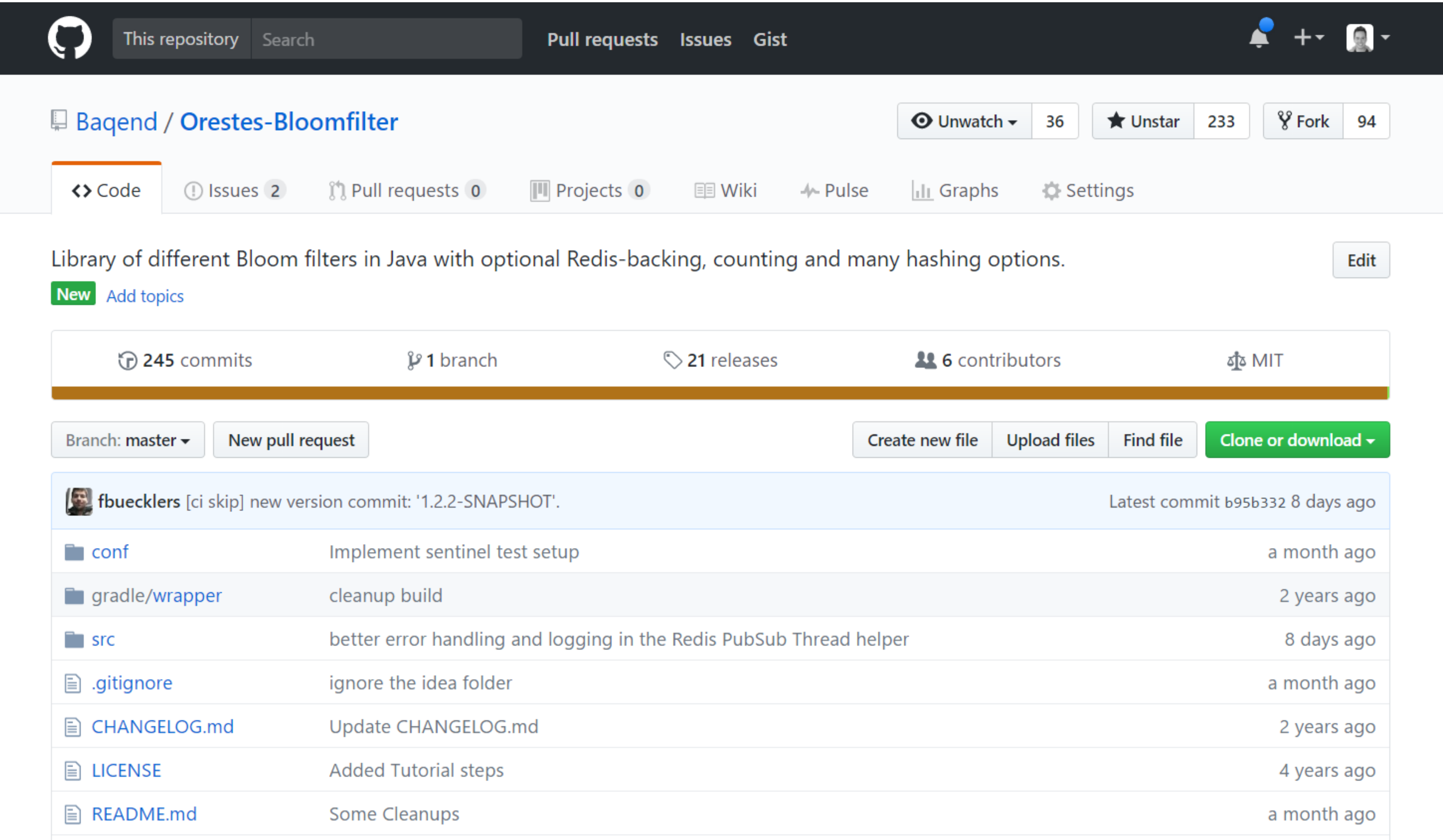
- Common Pattern: distributed system with **shared state** in Redis
- Example - Improve performance for legacy systems:



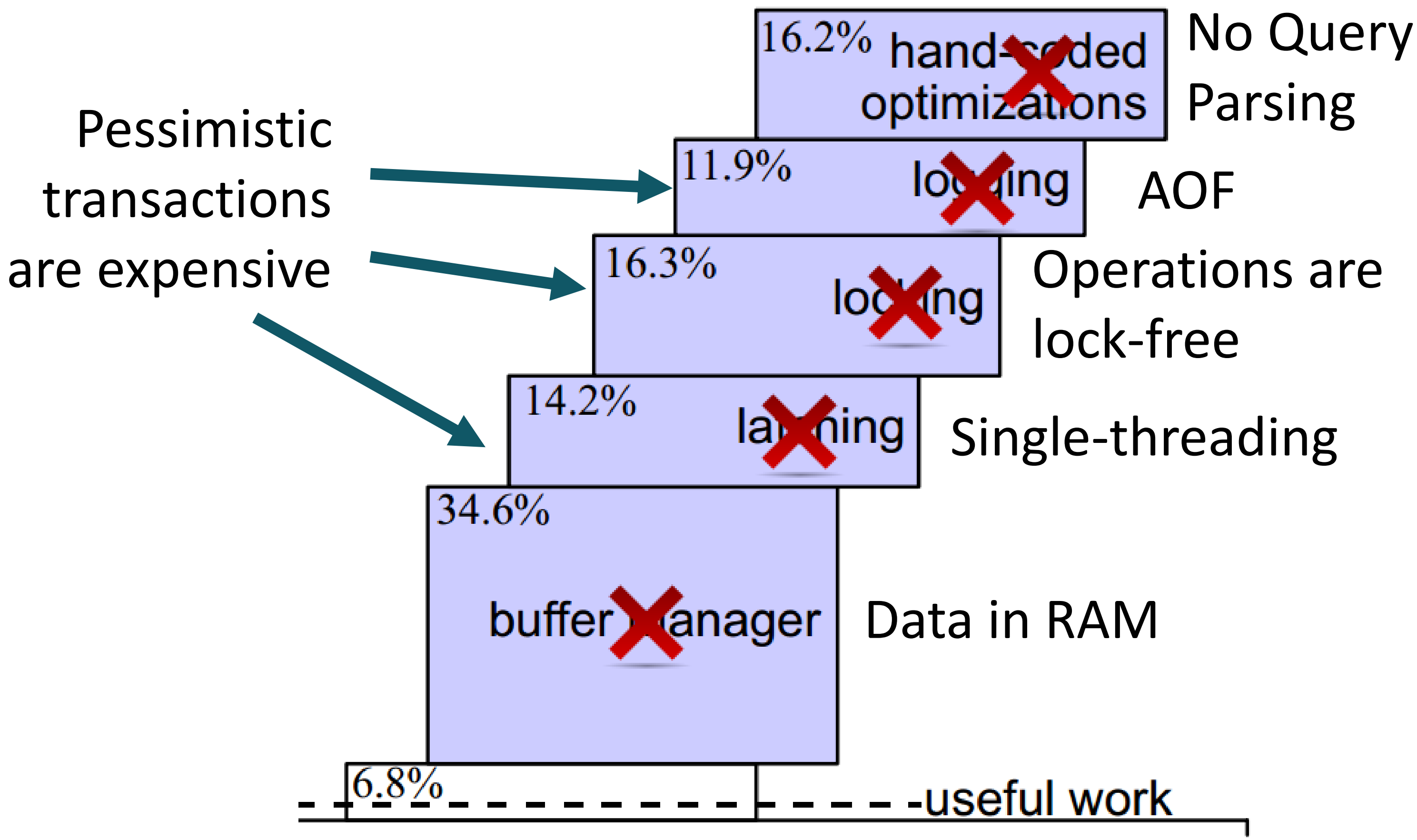
Redis Bloom filters

Open Source

https://github.com/Baqend/Orestes-Bloomfilter



Why is Redis so fast?



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

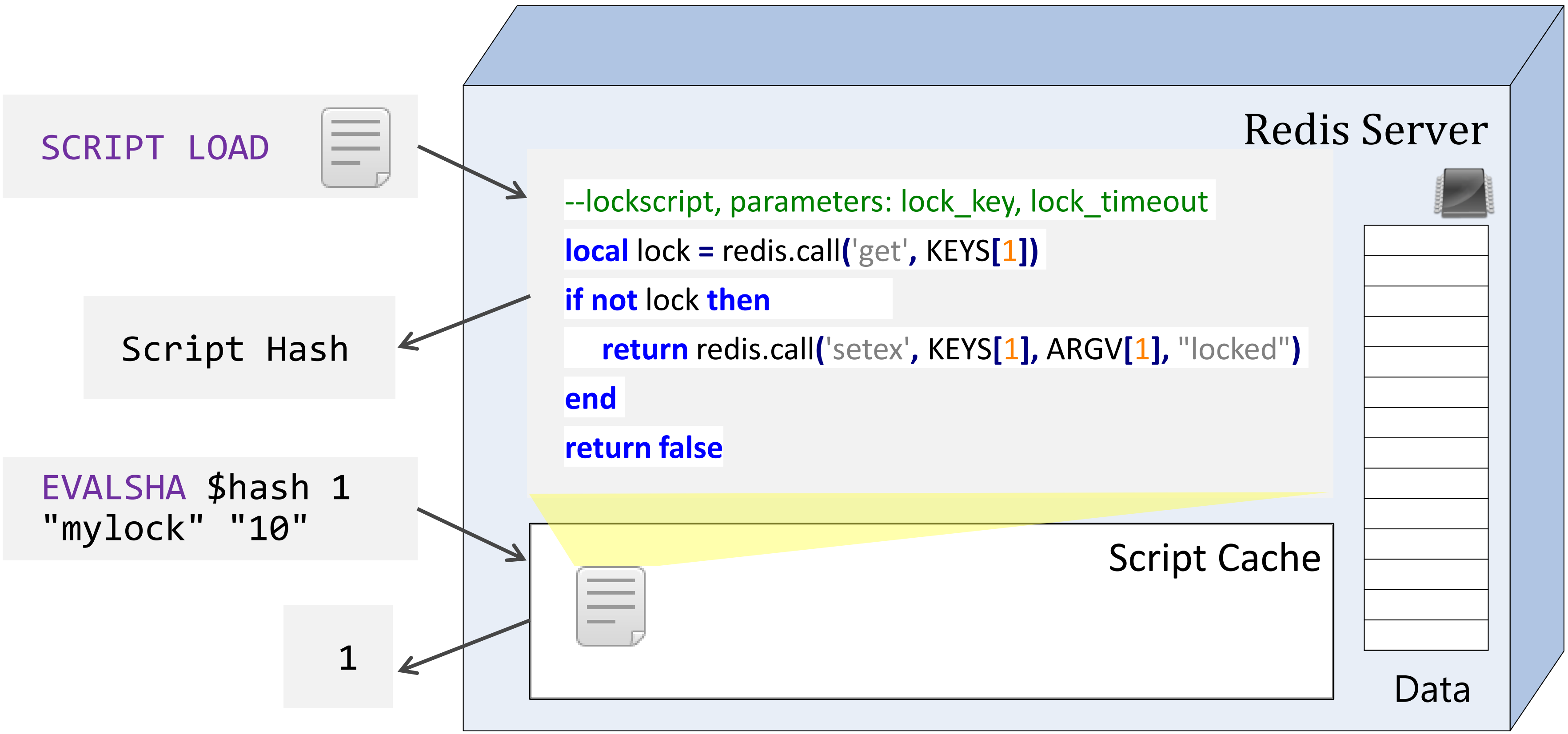
Optimistic Transactions

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

Only executed if bother keys are unchanged

```
WATCH users:2:followers, users:3:followers

MULTI

    SMEMBERS users:2:followers  →  Queued

    SMEMBERS users:3:followers  →  Queued

    INCR transactions  →  Queued

EXEC  →  Bulk reply with 3 results
```

Lua Scripting

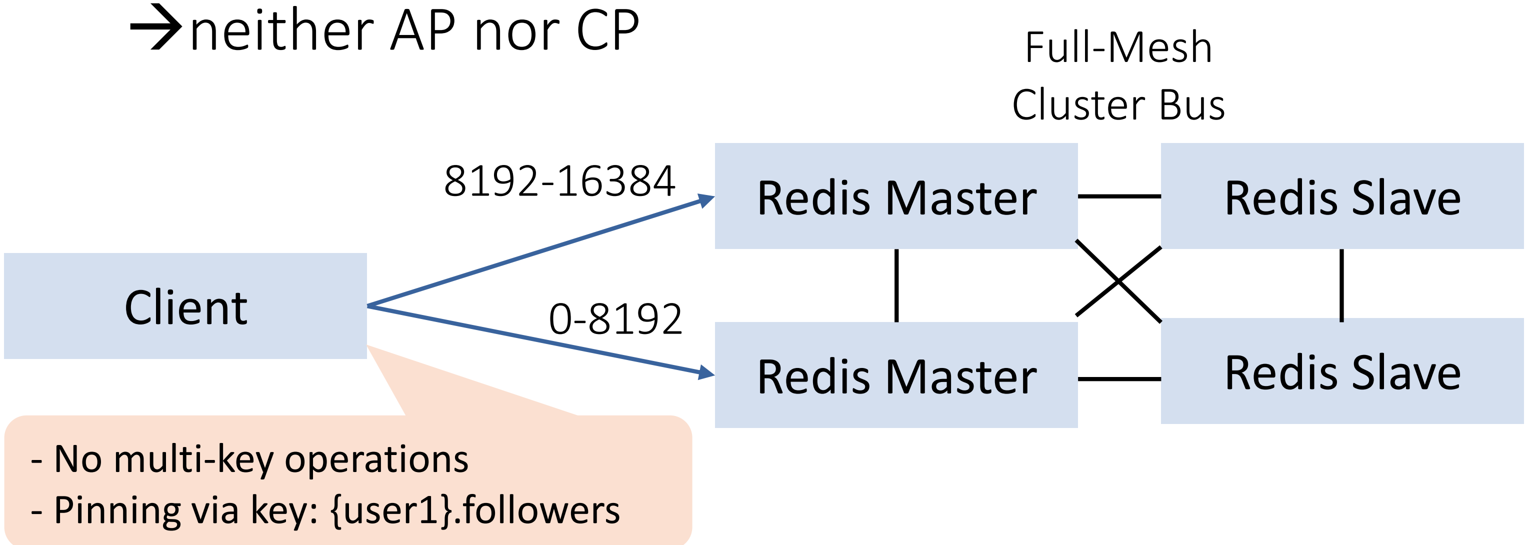


Ierusalimschy, Roberto. Programming in lua. 2006.

Redis Cluster

Native Sharding in Readis

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

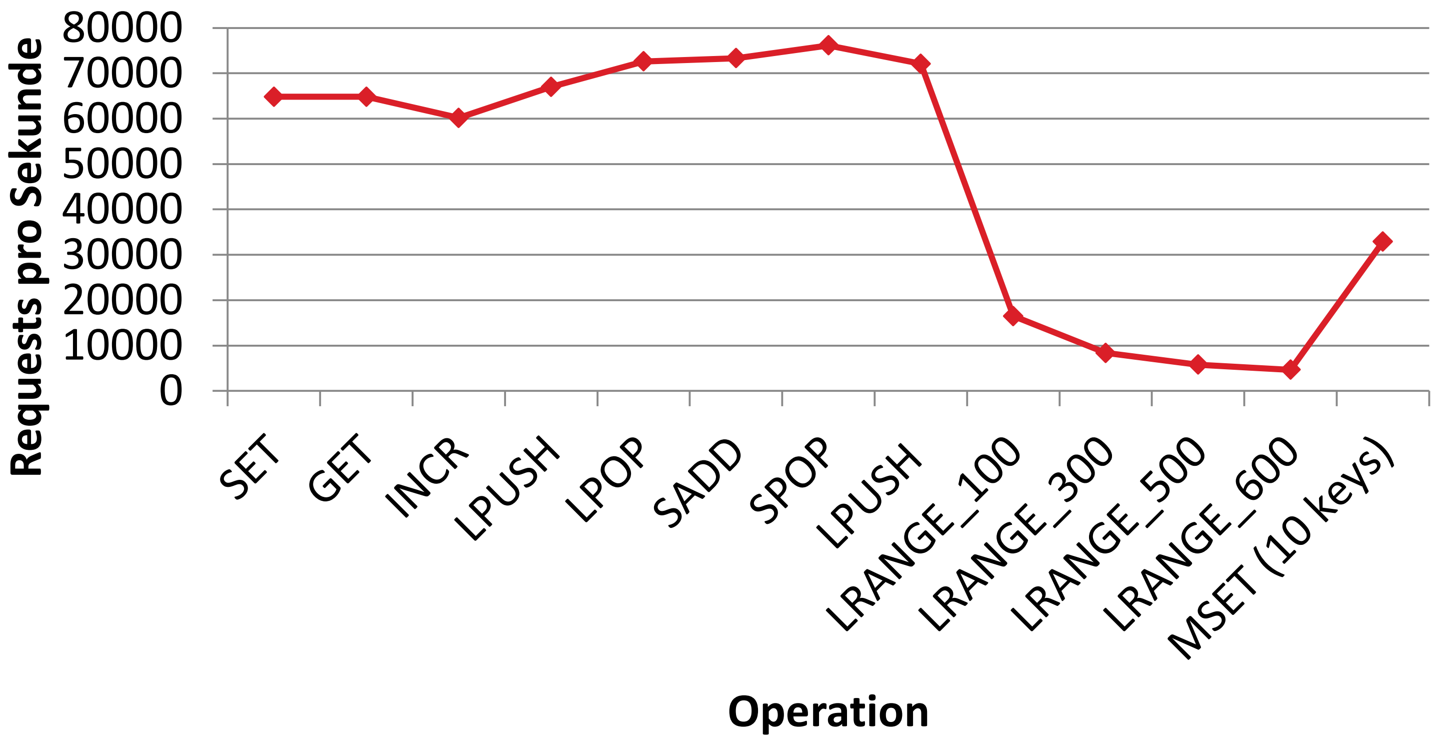


http://redis.io/topics/cluster-spec

Performance

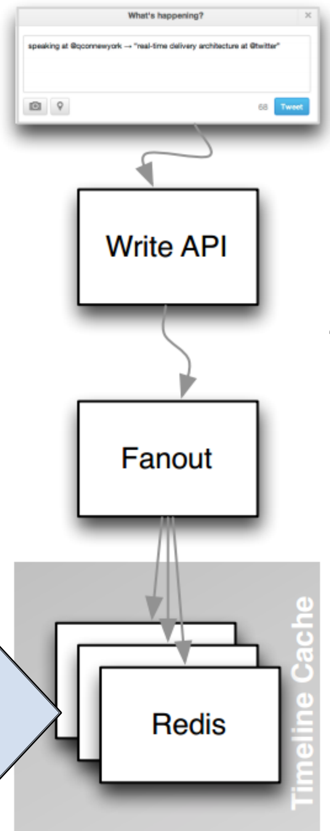
- Comparable to Memcache

```
> redis-benchmark -n 100000 -c 50
```



Example Redis Use-Case: Twitter

- Per User: one materialized timeline in Redis
- Timeline = List
- Key: User ID



>150 million users
~300k timeline querys/s

```
RPUSHX user_id tweet
```

http://www.infoq.com/presentations/Real-Time-Delivery-Twitter

Classification: Redis

Techniques

Sharding	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared Disk
Replication	Trans-action Protocol	Sync. Replica-tion	Async. Replica-tion	Primary Copy	Update Anywhere
Storage Management	Logging	Update-in-Place	Caching	In-Memory	Append-Only Storage
Query Processing	Global Index	Local Index	Query Planning	Analytics	Materialized Views

Google BigTable (CP)

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

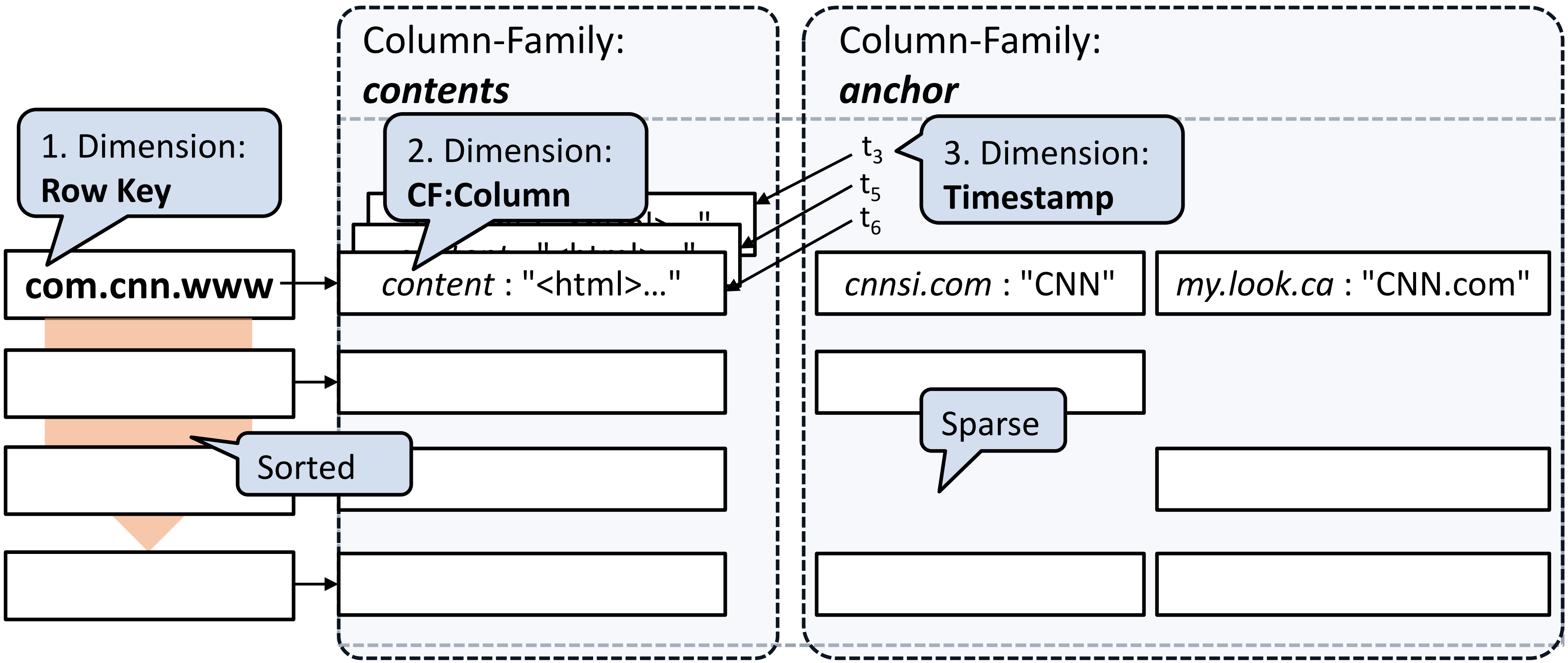
A Bigtable is a sparse, distributed, persistent multidimensional sorted map.

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

Chang, Fay, et al. "Bigtable: A distributed storage system for structured data."

Wide-Column Data Modelling

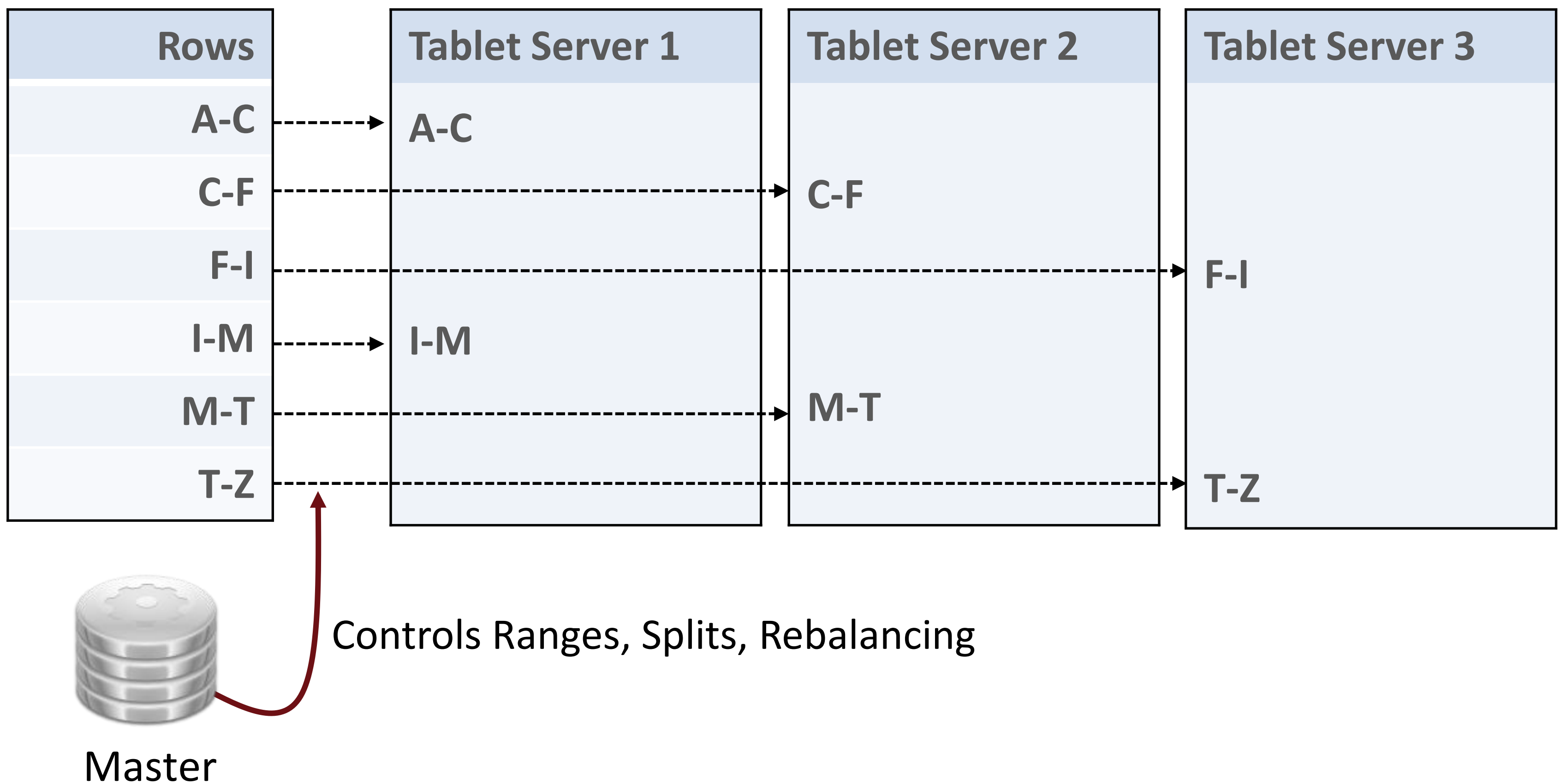
- Storage of crawled web-sites („Webtable“):



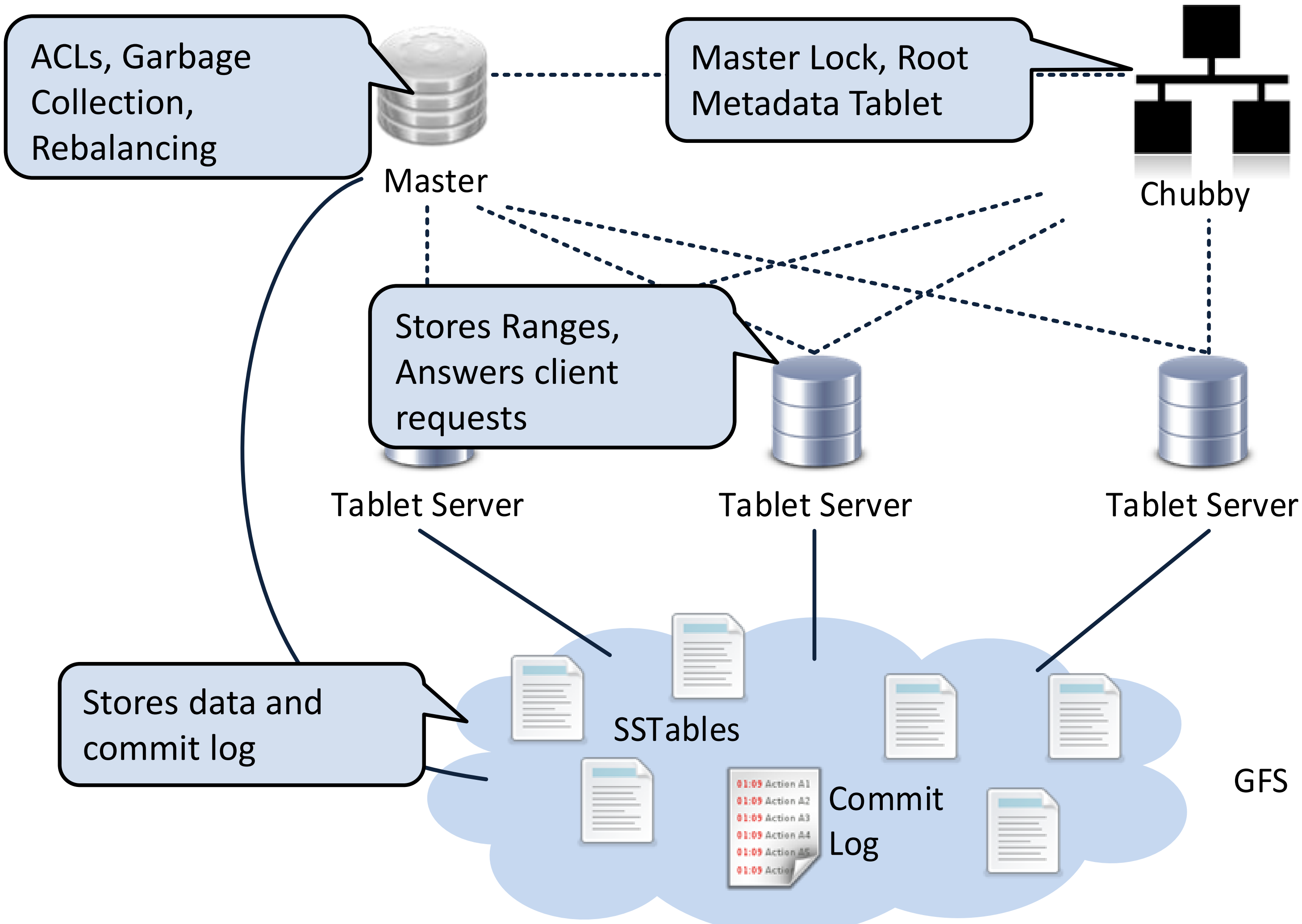
Range-based Sharding

BigTable Tablets

Tablet: Range partition of ordered records

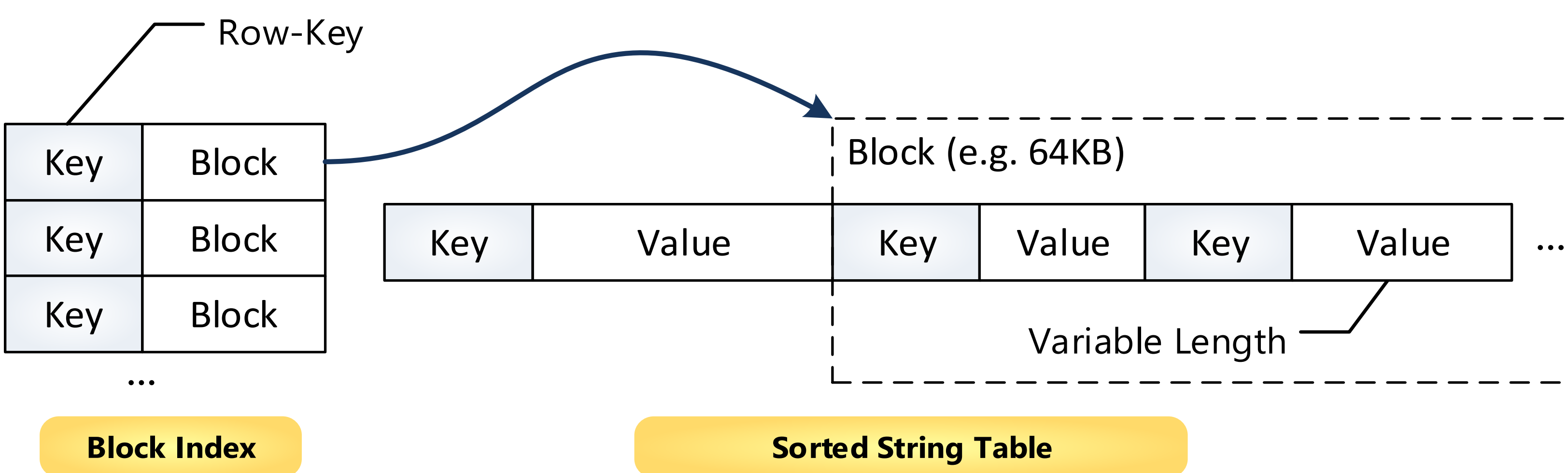


Architecture



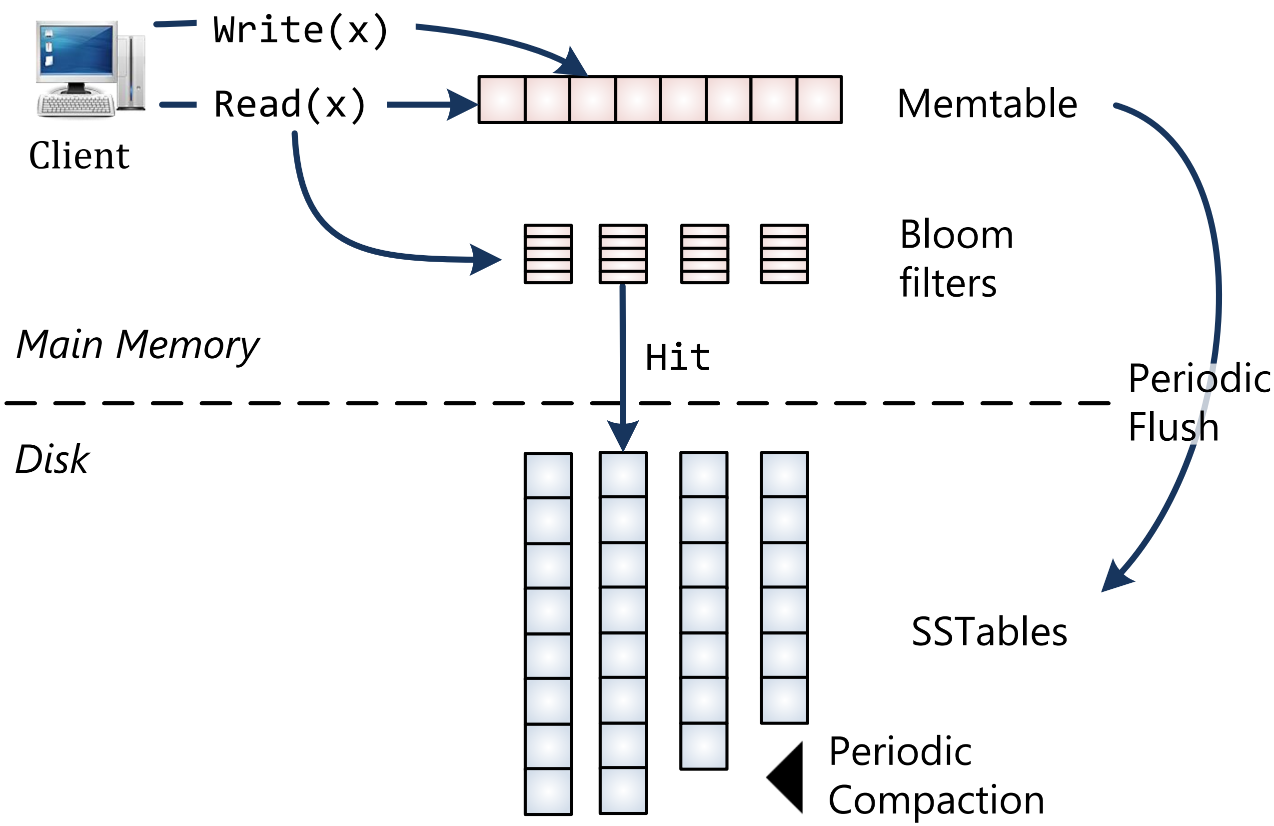
Storage: Sorted-String Tables

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



Storage: Optimization

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



Apache HBase (CP)

HBase
Model:
Wide-Column
License:
Apache 2
Written in:
Java

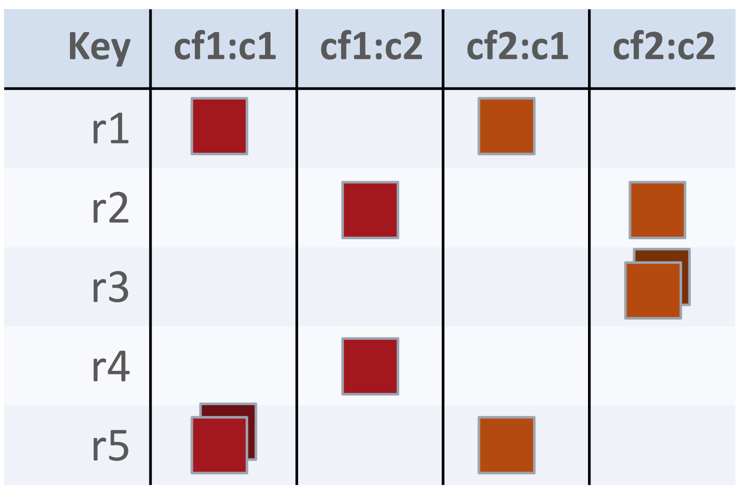
- Open-Source Implementation of BigTable
- Hadoop-Integration
  - Data source for Map-Reduce
  - Uses Zookeeper and HDFS
- Data modelling challenges: key design, tall vs wide
  - **Row Key**: only access key (no indices) → key design important
  - **Tall**: good for scans
  - **Wide**: good for gets,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

HBase Storage

- Logical to physical mapping:

In Value	r2:cf2:c2:t1:<value>
In Key	r2-<value>:cf2:c2:t1:_
In Column	r2:cf2:c2<value>:t1:_

Key Design – where to store data:




```
r1:cf2:c1:t1:<value>
r2:cf2:c2:t1:<value>
r3:cf2:c2:t2:<value>
r3:cf2:c2:t1:<value>
r5:cf2:c1:t1:<value>
```

HFile cf2

```
r1:cf1:c1:t1:<value>
r2:cf1:c2:t1:<value>
r3:cf1:c2:t1:<value>
r3:cf1:c1:t2:<value>
r5:cf1:c1:t1:<value>
```

HFile cf1

George, Lars. HBase: the definitive guide. 2011.

Example: Facebook Insights

MD5(Reversed Domain) + Reversed Domain + URL-ID — Row Key

6PM Total	6PM Male	…	01.01 Total	01.01 Male	…	Total	Male	…
10	7		100	65			567	

CF:Daily | CF:Monthly | CF:All

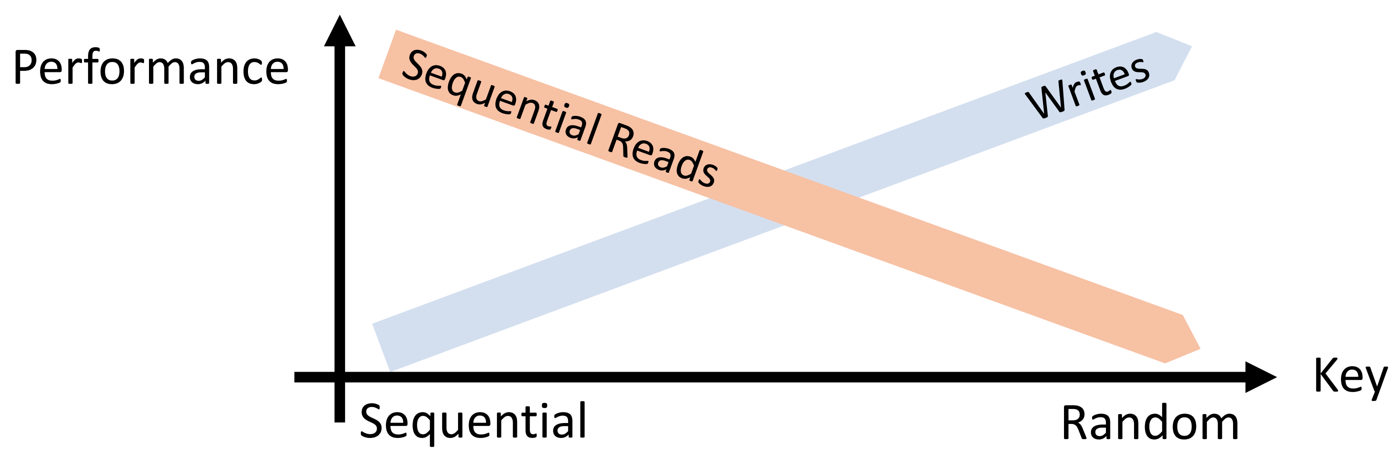
Atomic HBase **Counter**

TTL – automatic deletion of old rows

Lars George: "Advanced HBase Schema Design"

Schema Design

- Tall vs Wide Rows:
  - **Tall**: good for Scans
  - **Wide**: good for Gets
- Hotspots: Sequential Keys (z.B. Timestamp) dangerous



George, Lars. HBase: the definitive guide. 2011.

Schema: Messages

User ID	CF	Column	Timestamp	Message
12345	data	5fc38314-e290-ae5da5fc375d	1307097848	"Hi Lars, ..."
12345	data	725aae5f-d72e-f90f3f070419	1307099848	"Welcome, and ..."
12345	data	cc6775b3-f249-c6dd2b1a7467	1307101848	"To Whom It ..."
12345	data	dcbee495-6d5e-6ed48124632c	1307103848	"Hi, how are ..."

vs

ID:User+Message	CF	Column	Timestamp	Message
12345-5fc38314-e290-ae5da5fc375d	data		: 1307097848	"Hi Lars, ..."
12345-725aae5f-d72e-f90f3f070419	data		: 1307099848	"Welcome, and ..."
12345-cc6775b3-f249-c6dd2b1a7467	data		: 1307101848	"To Whom It ..."
12345-dcbee495-6d5e-6ed48124632c	data		: 1307103848	"Hi, how are ..."

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

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

http://2013.nosql-matters.org/cgn/wp-content/uploads/2013/05/HBase-Schema-Design-NoSQL-Matters-April-2013.pdf

API: CRUD + Scan

Setup Cloud Cluster:

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

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

Login, cluster size, etc.

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

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

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

API: Features

- Row **Locks** (MVCC): `table.lockRow()`, `unlockRow()`
  - Problem: Timeouts, Deadlocks, Ressources
- **Conditional Updates**: `checkAndPut()`, `checkAndDelete()`
- CoProcessors - registriered Java-Classes for:
  - Observers (`prePut`, `postGet`, etc.)
  - Endpoints (Stored Procedures)
- HBase can be a Hadoop **Source**:

```
TableMapReduceUtil.initTableMapperJob(
  tableName,  //Table
  scan,  //Data input as a Scan
  MyMapper.class, ...  //usually a TableMapper<Text,Text> );
```

Summary: BigTable, HBase

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

Classification: HBase

Techniques

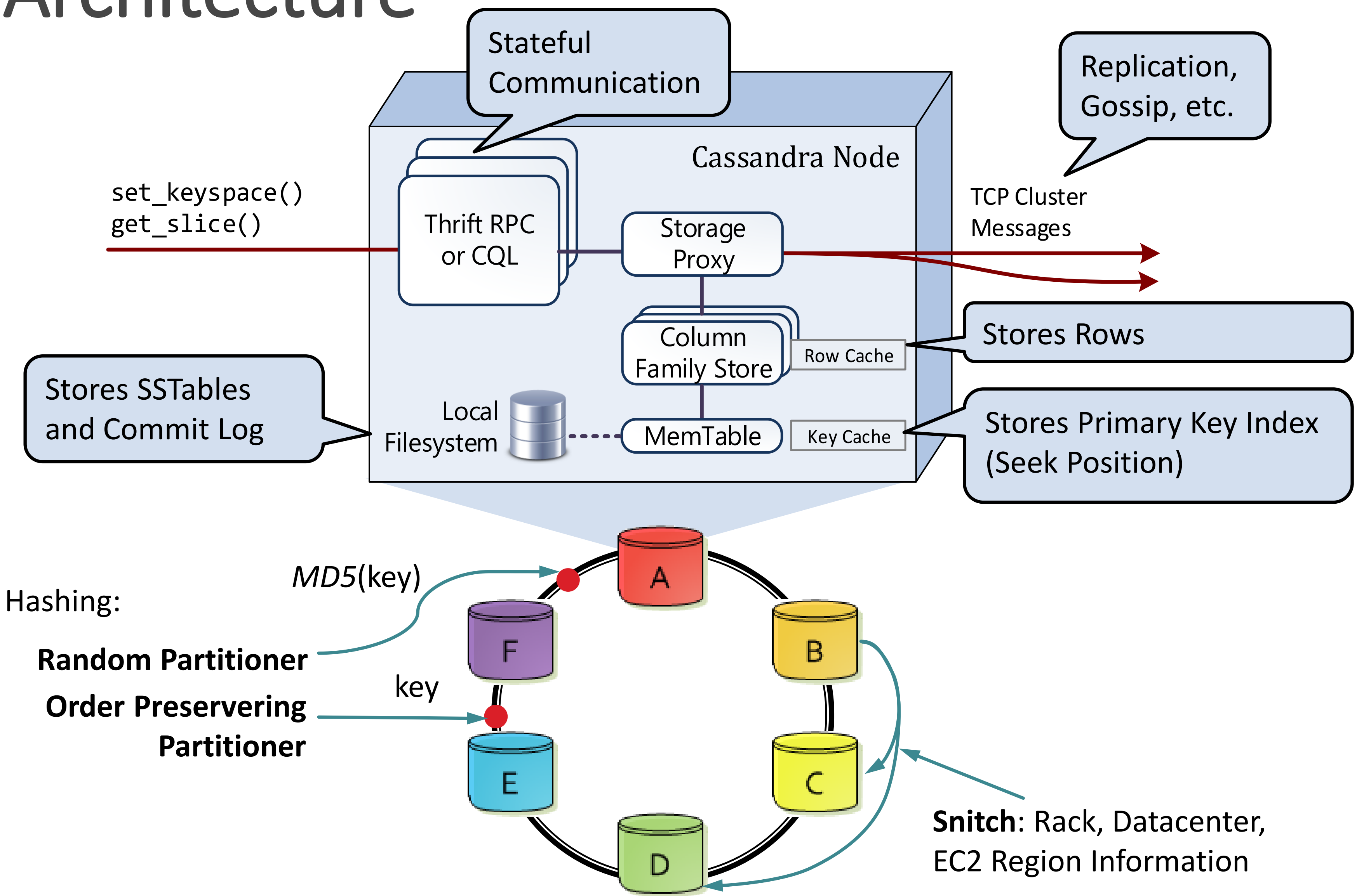
Sharding	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared Disk
Replication	Trans-action Protocol	Sync. Replica-tion	Async. Replica-tion	Primary Copy	Update Anywhere
Storage Management	Logging	Update-in-Place	Caching	In-Memory	Append-Only Storage
Query Processing	Global Index	Local Index	Query Planning	Analytics	Materialized Views

Apache Cassandra (AP)

Cassandra
Model:
Wide-Column
License:
Apache 2
Written in:
Java

- Published 2007 by Facebook
- **Idea**:
  - BigTable's wide-column data model
  - Dynamo ring for replication and sharding
- Cassandra Query Language (CQL): SQL-like query- and DDL-language
- **Compound indices**: *partition key* (shard key) + *clustering key* (ordered per partition key) → Limited range queries

Architecture



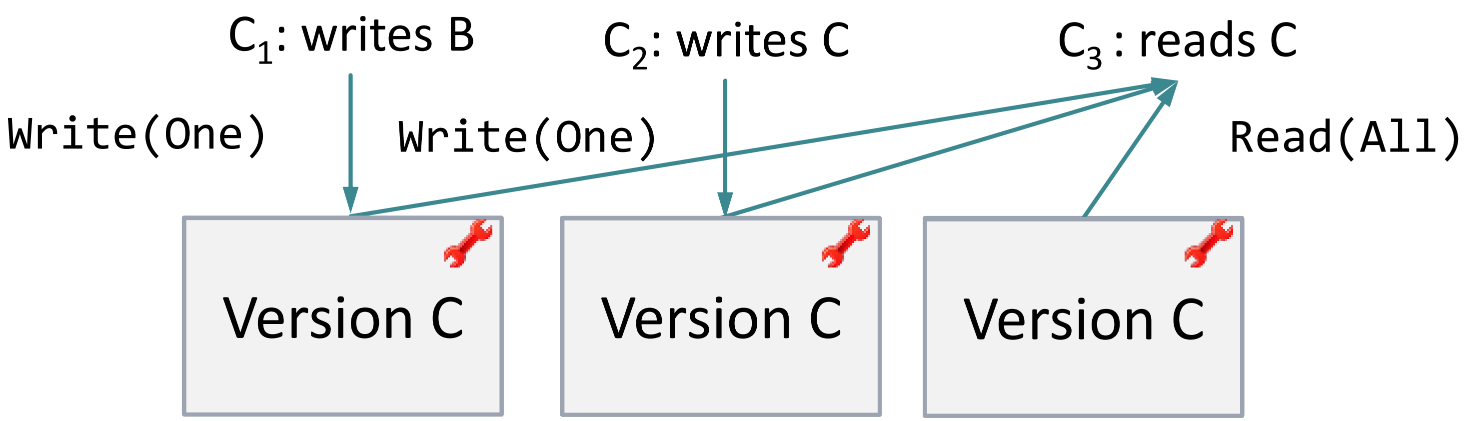
Consistency

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

Write	**Read**
Any	-
One	One
Two	Two
Quorum	Quorum
Local_Quorum / Each_Quorum	Local_Quorum / Each_Quorum
All	All

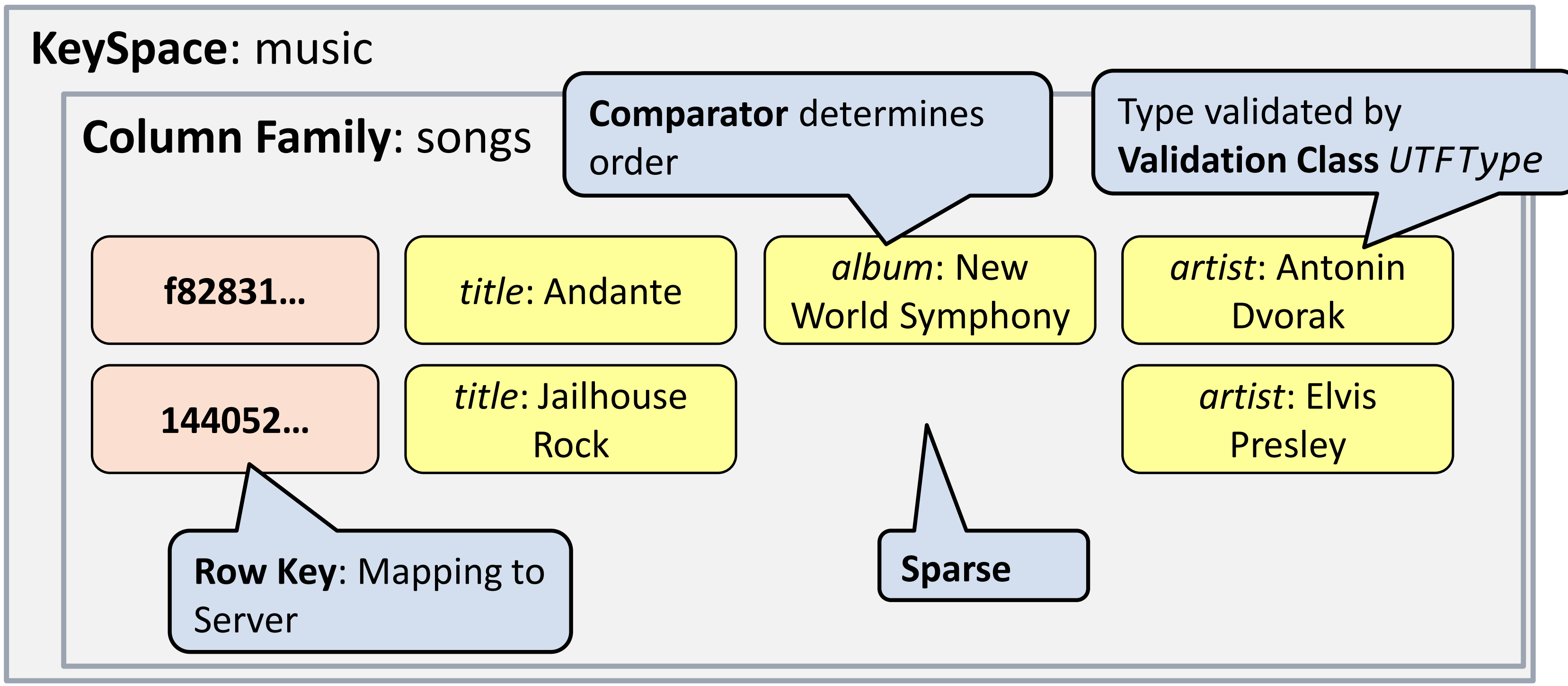
Consistency

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



Storage Layer

- Uses BigTables Column Family Format



http://www.datastax.com/dev/blog/cql3-for-cassandra-experts

CQL Example: Compound keys

- Enables Scans despite *Random Partitioner*

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

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

Partition Key

Clustering Columns:
sorted per node

id	song_order	song_id	artist
23423	1	64563	Elvis
23423	2	f9291	Elvis

Other Features

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

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

Classification: Cassandra

Techniques

Sharding	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared Disk
Replication	Trans-action Protocol	Sync. Replica-tion	Async. Replica-tion	Primary Copy	Update Anywhere
Storage Management	Logging	Update-in-Place	Caching	In-Memory	Append-Only Storage
Query Processing	Global Index	Local Index	Query Planning	Analytics	Materialized Views

MongoDB (CP)

MongoDB
Model:
Document
License:
GNU AGPL 3.0
Written in:
C++

- From hu**mongo**us ≅ gigantic
- Schema-free document database with tunable consistency
- Allows complex queries and indexing
- **Sharding** (either range- or hash-based)
- **Replication** (either synchronous or asynchronous)
- Storage Management:
  - **Write-ahead logging** for redos (*journaling*)
  - **Storage Engines:** memory-mapped files, in-memory, Log-structured merge trees (WiredTiger), …

Basics

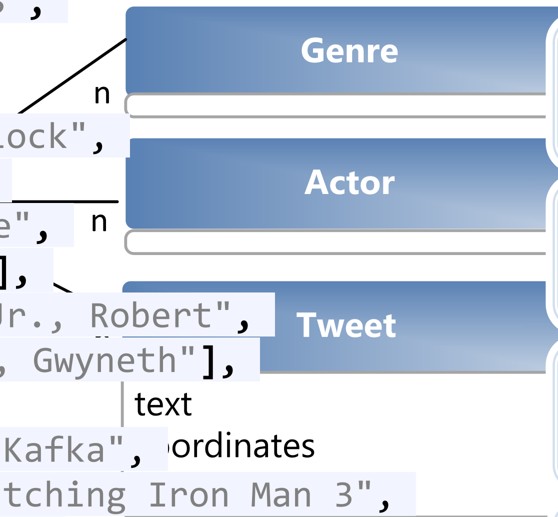
```
> mongod &
> mongo imdb
MongoDB shell version: 2.4.3
connecting to: imdb
> show collections
movies
tweets
> db.movies.findOne({title : "Iron Man 3"})
{
        title : "Iron Man 3",
        year : 2013 ,
        genre : [
                "Action",
                "Adventure",
                "Sci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

Properties

Arrays, Nesting allowed

Data Modelling

```
{
        "_id" : ObjectId("51a5d316d70beffe74ecc940")
        title : "Iron Man 3",
        year : 2013,
        rating : 7.6,
        director: "Shane Block",
        genre : [ "Action",
                  "Adventure",
                  "Sci -Fi"],
        actors : ["Downey Jr., Robert",
                  "Paltrow , Gwyneth"],
        tweets : [ {
           "user" : "Franz Kafka",
           "text" : "#nowwatching Iron Man 3",
           "retweet" : false,
           "date" : ISODate("2013-05-29T13:15:51Z")
        }]
}
```



Movie Document

Denormalisation instead of joins

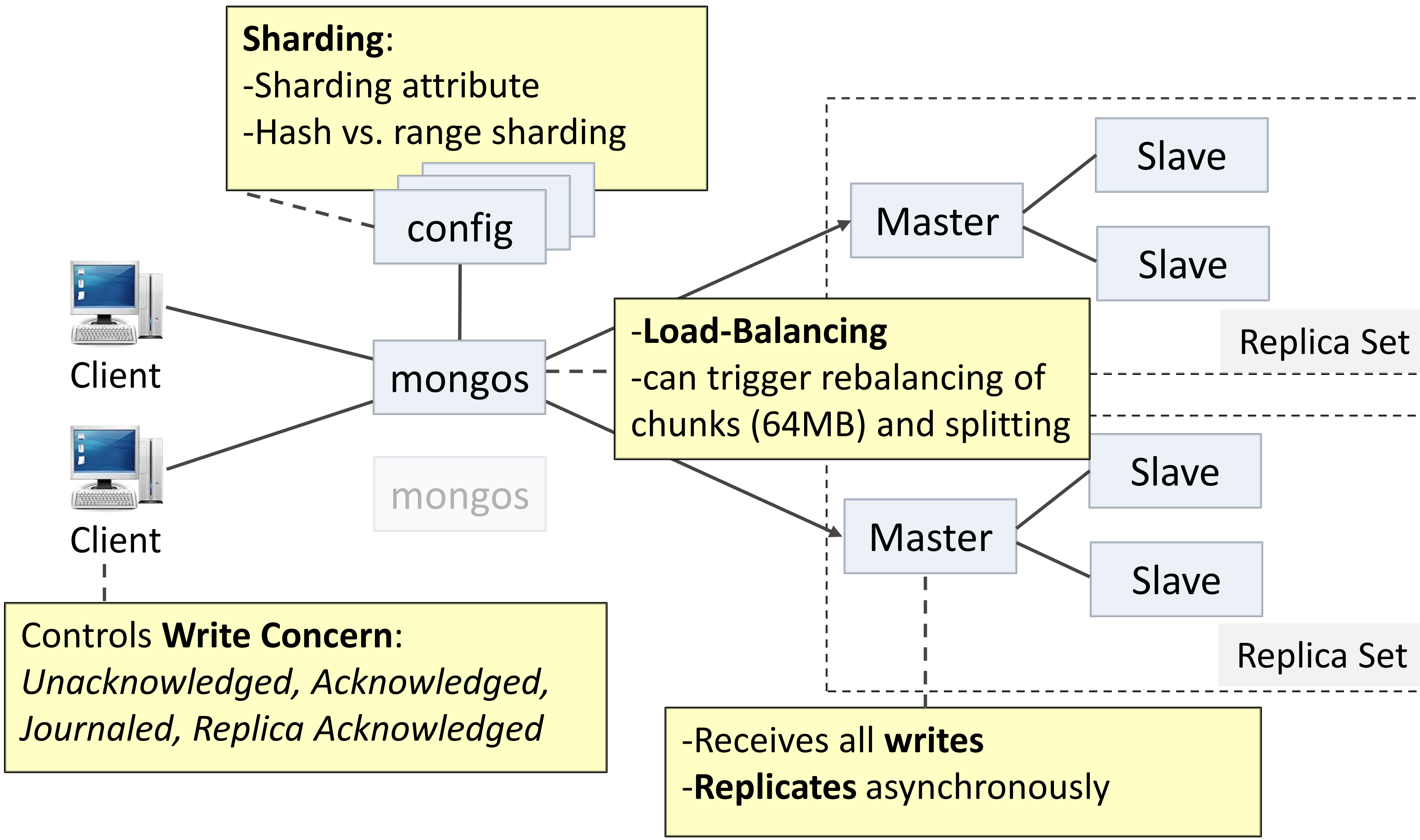
Nesting replaces 1:n and 1:1 relations

Schemafreeness: Attributes per document

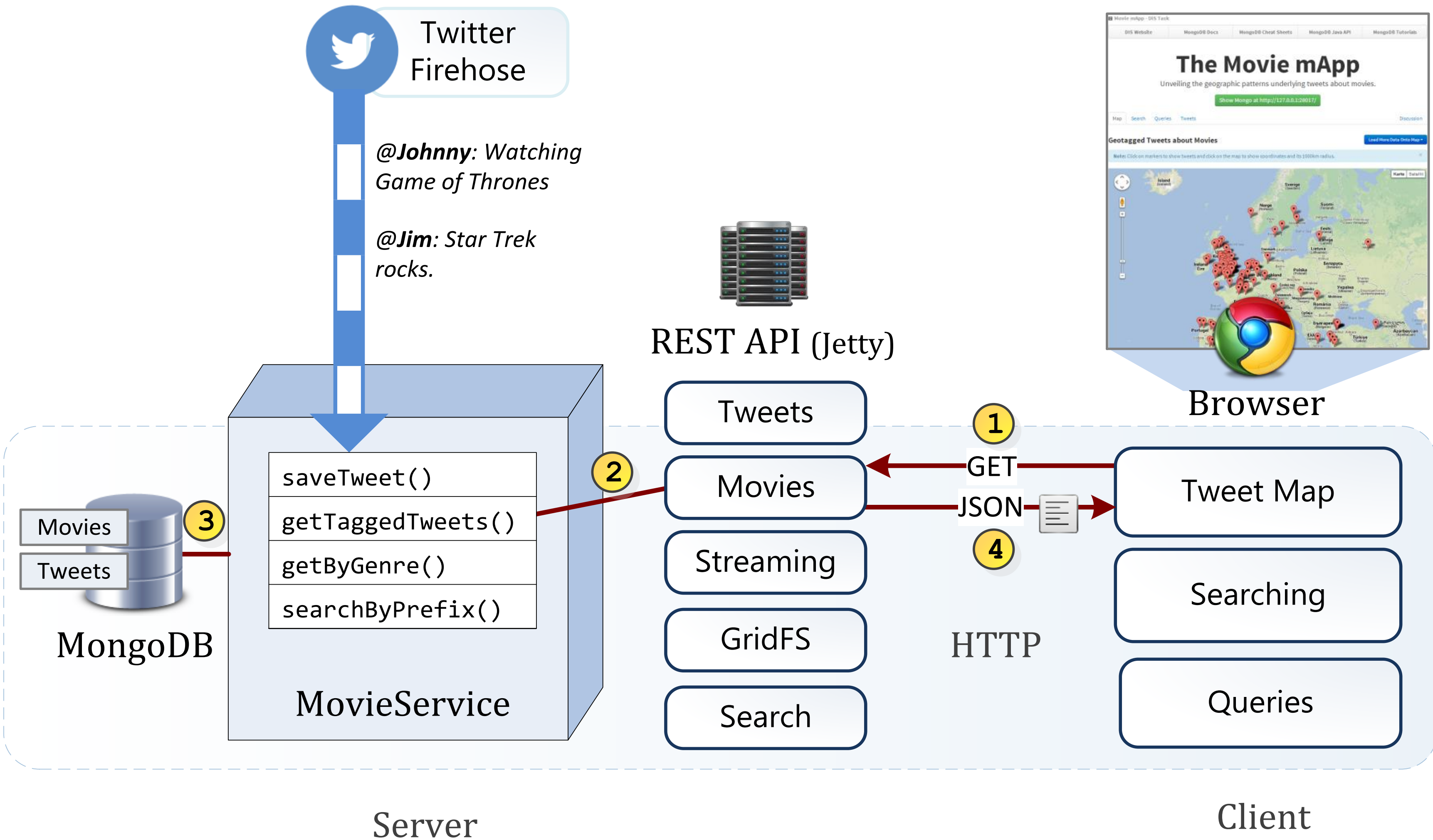
Unit of atomicity: document

Principles

Sharding und Replication



MongoDB Example App



MongoDB by Example

Movie mApp - DIS Task

DIS Website | MongoDB Docs | MongoDB Cheat Sheet | MongoDB Java API | MongoDB Tutorials

The Movie mApp

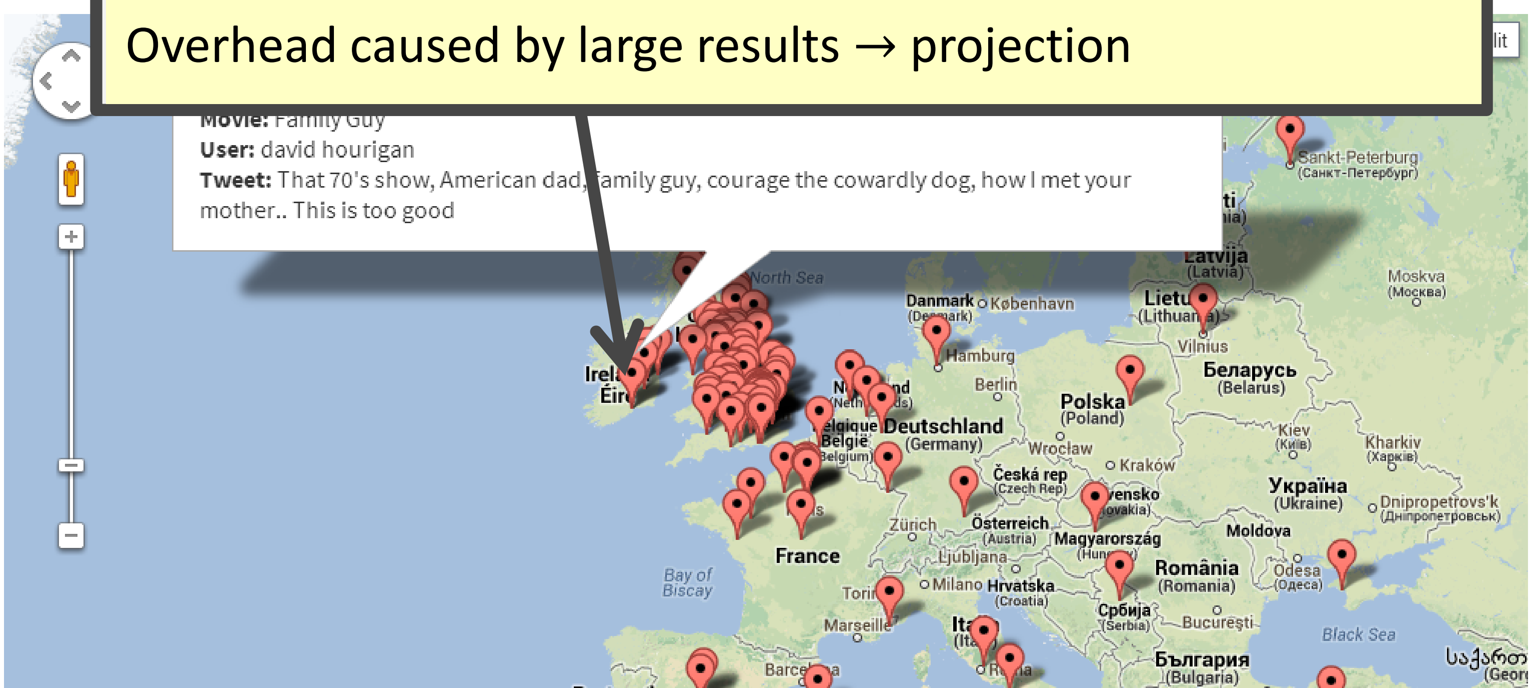
Unveiling the geographic patterns underlying tweets about movies.

```
DBObject query = new BasicDBObject("tweets.coordinates",
                      new BasicDBObject("$exists", true));
db.getCollection("movies").find(query);
```

Or in JavaScript:

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

Overhead caused by large results → projection



Movie mApp - DIS Task

DIS Website	MongoDB Docs	MongoDB Cheat Sheets	MongoDB Java API	MongoDB Tutorials

The Movie mApp

Unveiling the geographic patterns underlying tweets about movies.

Show Mongo at http://127.0.0.1:28017/

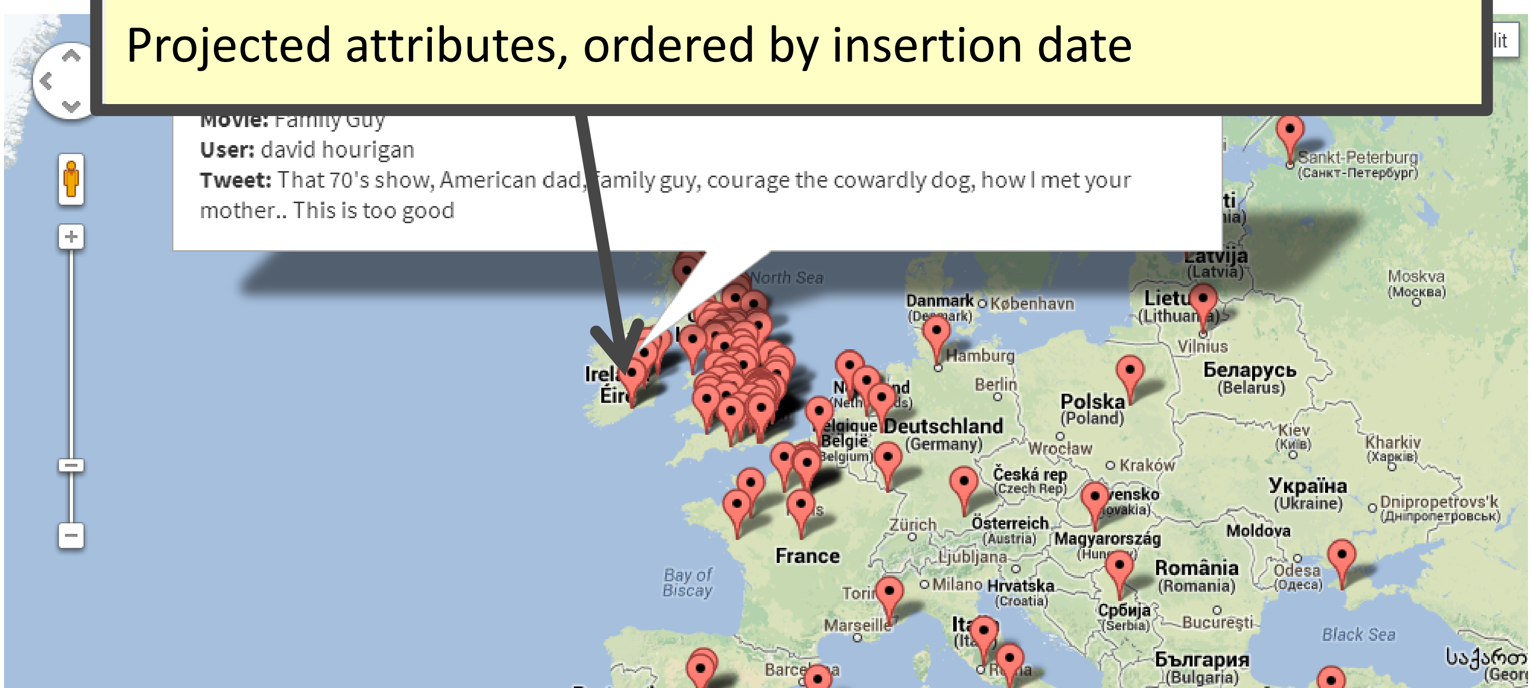
Map

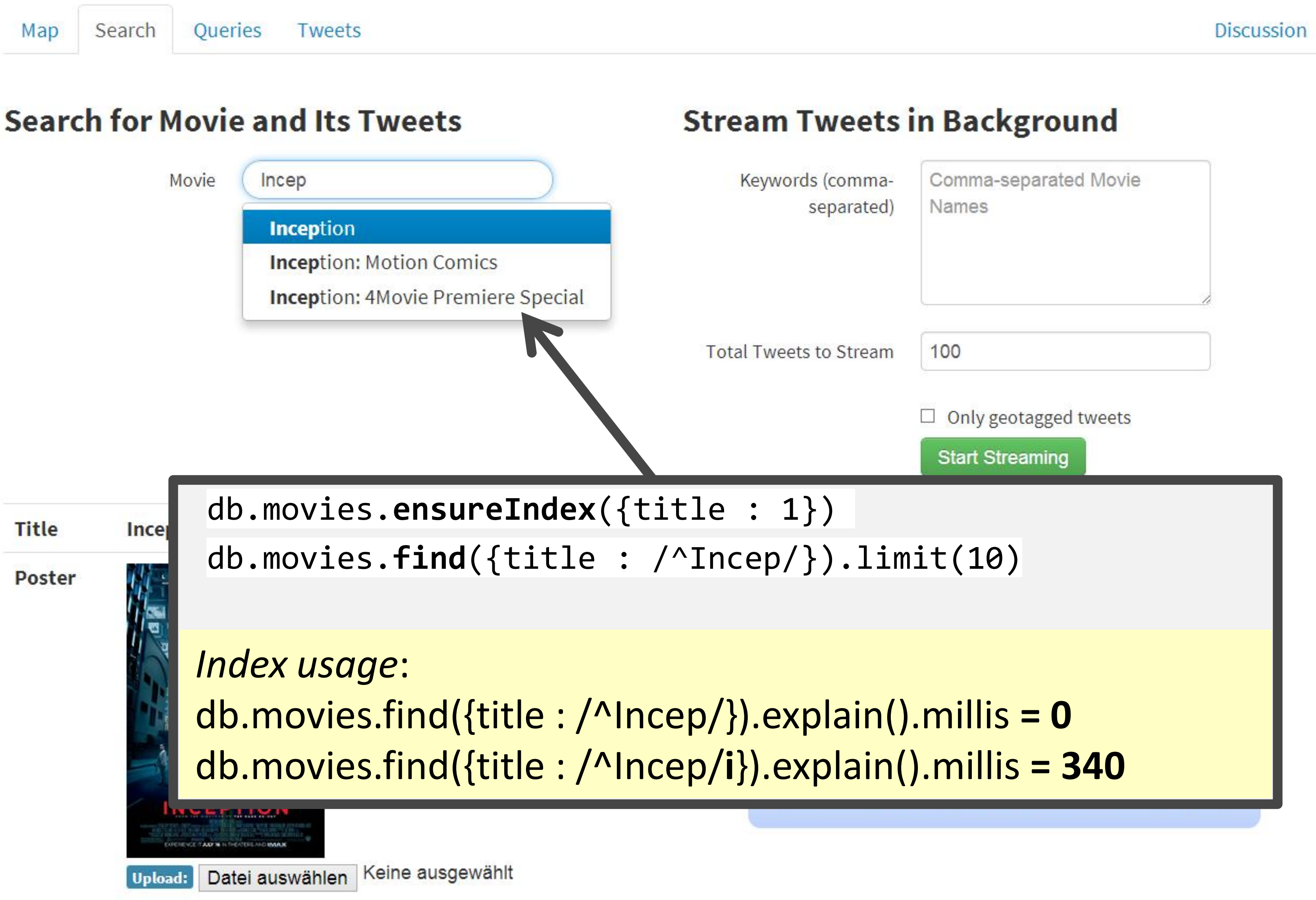
```
db.tweets.find({coordinates : {"$exists" : 1}},
  {text:1, movie:1, "user.name":1, coordinates:1})
  .sort({id:-1})
```

Projected attributes, ordered by insertion date

Geot

Note:



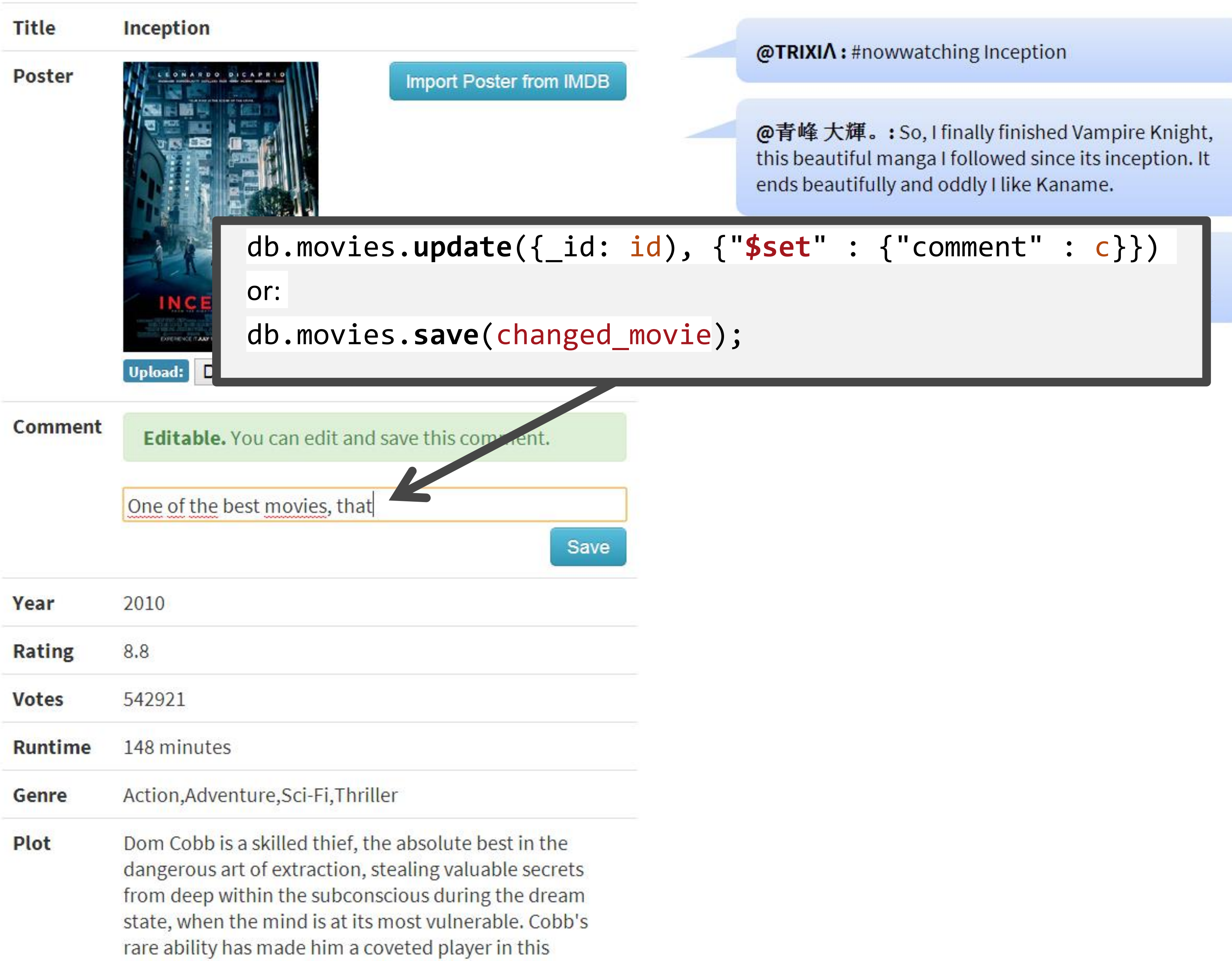


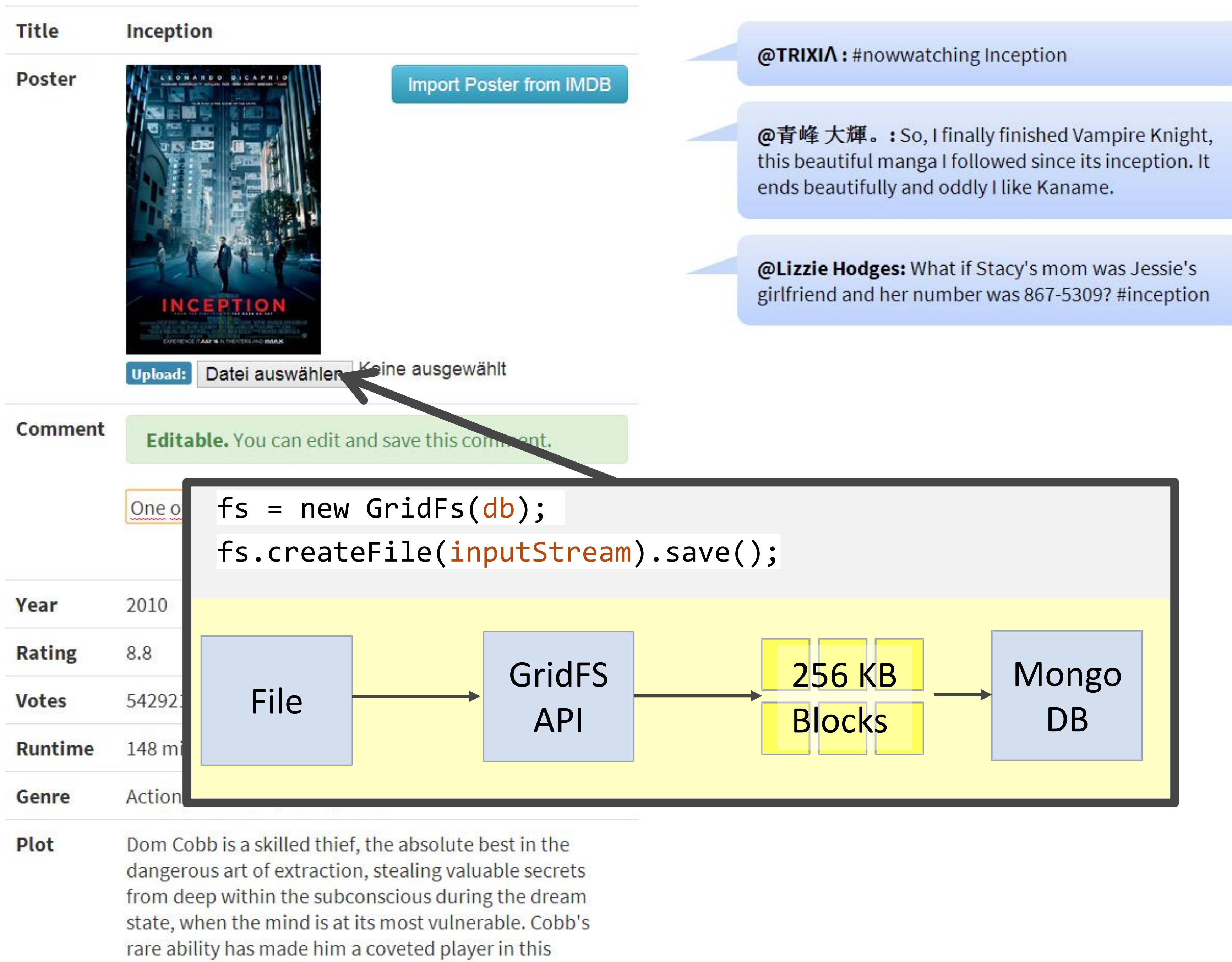

```
db.movies.ensureIndex({title : 1})
db.movies.find({title : /^Incep/}).limit(10)
```

Index usage:

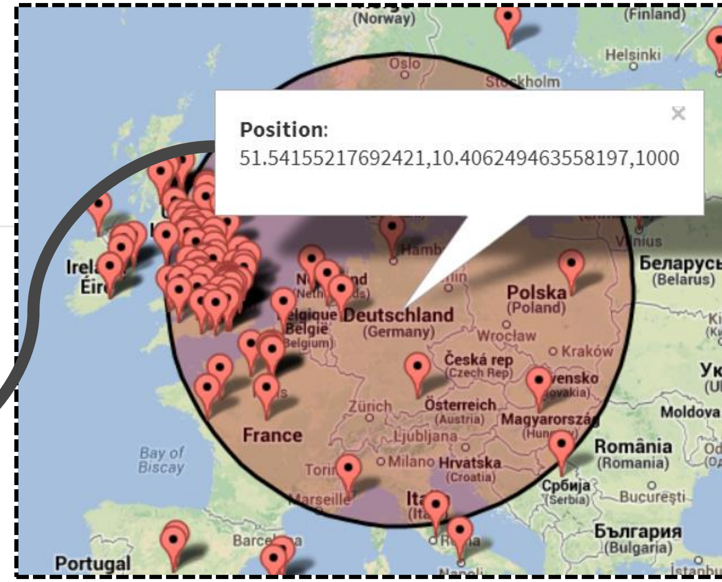
db.movies.find({title : /^Incep/}).explain().millis **= 0**

db.movies.find({title : /^Incep/**i**}).explain().millis **= 340**


Title Inception
Poster
LEONARDO DICAPRIO
INCE
Import Poster from IMDB
Upload:
@TRIXIΛ : #nowwatching Inception
@青峰 大輝。: So, I finally finished Vampire Knight, this beautiful manga I followed since its inception. It ends beautifully and oddly I like Kaname.
db.movies.update({_id: id), {"$set" : {"comment" : c}})
or:
db.movies.save(changed_movie);
Comment
Editable. You can edit and save this comment.
One of the best movies, that
Save
Year 2010
Rating 8.8
Votes 542921
Runtime 148 minutes
Genre Action,Adventure,Sci-Fi,Thriller
Plot Dom Cobb is a skilled thief, the absolute best in the dangerous art of extraction, stealing valuable secrets from deep within the subconscious during the dream state, when the mind is at its most vulnerable. Cobb's rare ability has made him a coveted player in this


Title Inception
Poster
Import Poster from IMDB
INCEPTION
Upload: Datei auswählen Keine ausgewählt
Comment
Editable. You can edit and save this comment.
One o
Year 2010
Rating 8.8
Votes 54292
Runtime 148 mi
Genre Action
Plot Dom Cobb is a skilled thief, the absolute best in the dangerous art of extraction, stealing valuable secrets from deep within the subconscious during the dream state, when the mind is at its most vulnerable. Cobb's rare ability has made him a coveted player in this
@TRIXIΛ : #nowwatching Inception
@青峰 大輝。: So, I finally finished Vampire Knight, this beautiful manga I followed since its inception. It ends beautifully and oddly I like Kaname.
@Lizzie Hodges: What if Stacy's mom was Jessie's girlfriend and her number was 867-5309? #inception
fs = new GridFs(db);
fs.createFile(inputStream).save();
File
GridFS API
256 KB Blocks
Mongo DB

Map | Search | Queries | Tweets



Query Tweets

Query: Get Tweets Near: lat,lng,radius-in-km [Go]

Parameter: 51.54155217692421,10.406249463558197,1000

Result Limit: 10

User	Tweet	Created at	Coordinates
MitchellyMonica			
J. Z.			
Party Hardy			
nadine stachowiak		2013	
PAIRSonnalites | DE	ES-News : http://t.co/WhBmNr6OM2: In vielen Staaten Afrikas gibt es	Wed May 29	13.46757813,52.5913198

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

Geospatial Queries:

- **Distance**
- **Intersection**
- **Inclusion**

Map Search Queries Tweets Discussion

Query Tweets

Query: Indexed Fulltext Search on Tweets [Go]

Parameter: StAr trek

Result Limit: 100

Show 25 search results per page

Filter search results:

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

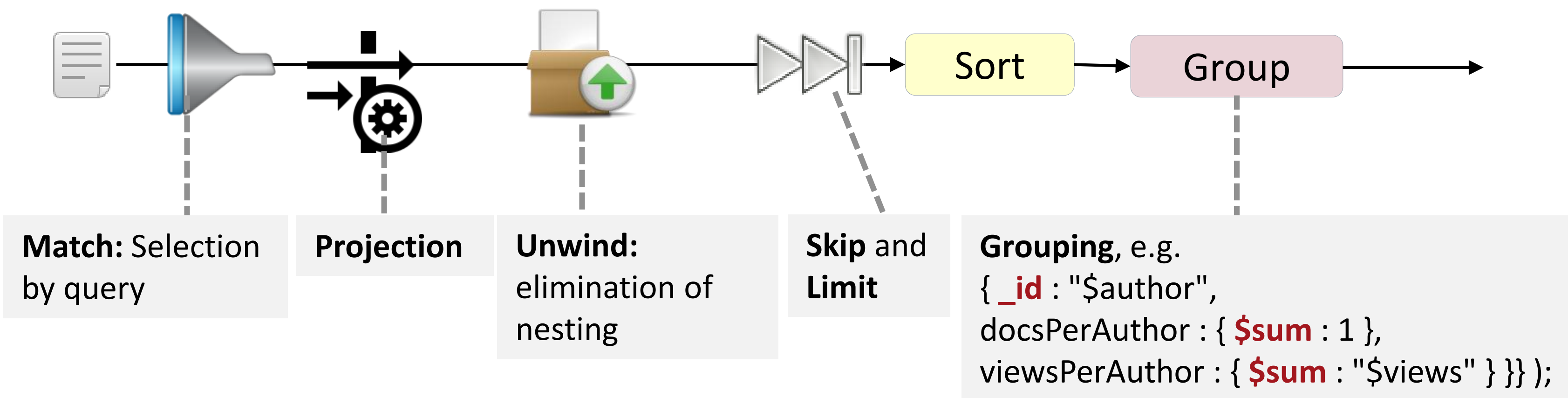
Full-text Search:

- **Tokenization, Stop Words**
- **Stemming**
- **Scoring**

User	Tweet	Created at	Coordinates
manwonman			
Mia Clrss Hrnndz ♡			
A N G G I_			
Stefany Ezra Elvina		2013	
Vanessa Yung	Star Trek into Darkness☐	Wed May 29 19:21:06 +0000 2013	-2.986771,53.404051
tam wilson	Finally getting to see Star Trek! (at @DCADundee Contemporary Arts for Star Trek Into Darkness 3D) http://t.co/0ojg4KMBL5	Wed May 29 18:48:56 +0000	-2.97489166,56.45753477

Analytic Capabilities

- Aggregation Pipeline Framework:

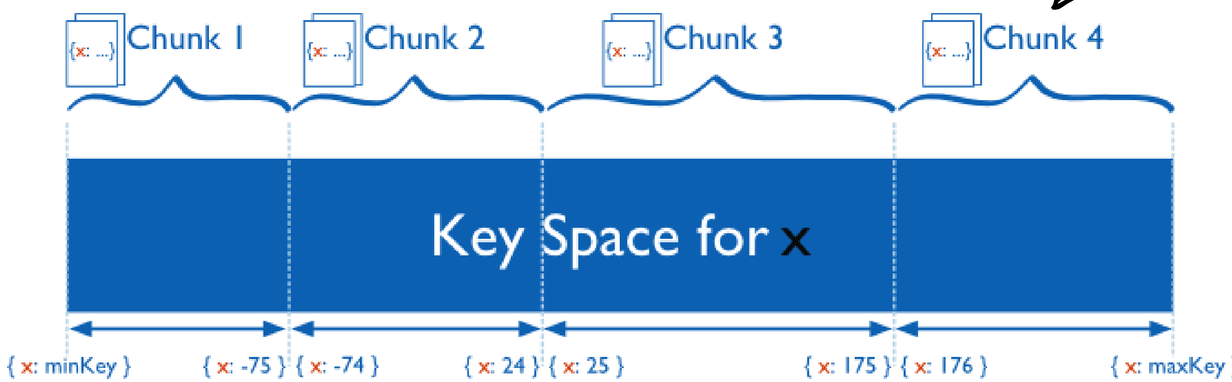


- Alternative: JavaScript MapReduce

Sharding

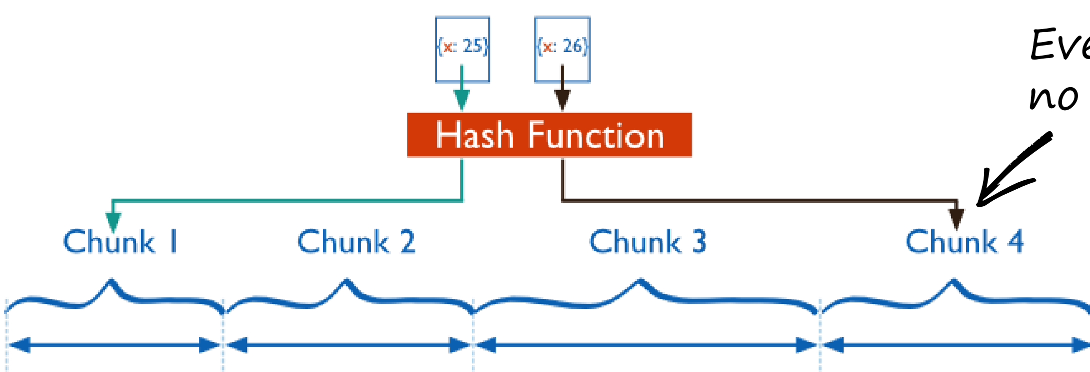
In the optimal case only one shard asked per query, else: Scatter-and-gather

- Range-based:



- Hash-based:

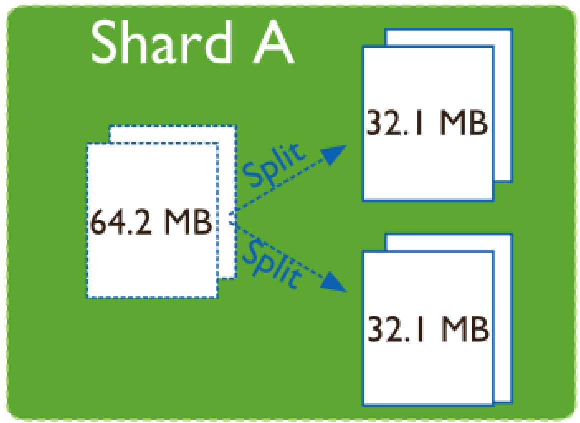
Even distribution, no locality



docs.mongodb.org/manual/core/sharding-introduction/

Sharding

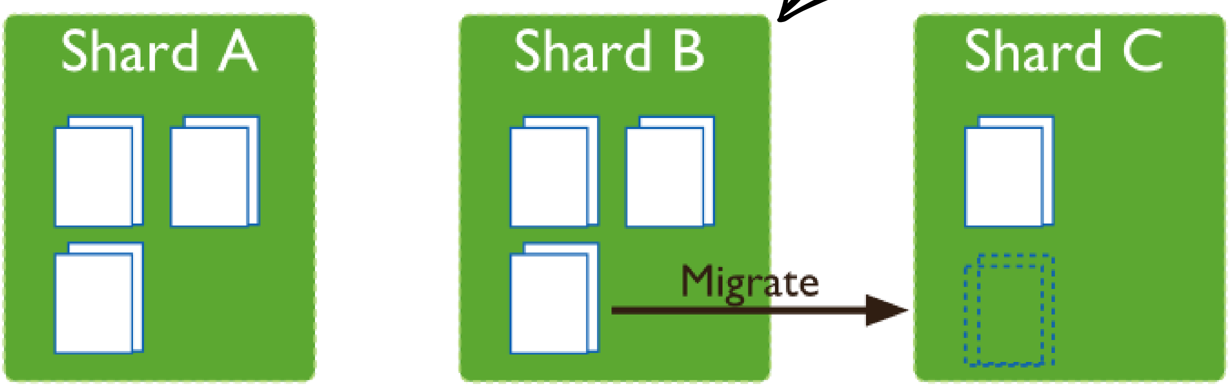
- Splitting:



Split chunks that are too large

- Migration:

Mongos Load Balancer triggers rebalancing



docs.mongodb.org/manual/core/sharding-introduction/

Classification: MongoDB

Techniques

Sharding	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared Disk
Replication	Trans-action Protocol	Sync. Replica-tion	Async. Replica-tion	Primary Copy	Update Anywhere
Storage Management	Logging	Update-in-Place	Caching	In-Memory	Append-Only Storage
Query Processing	Global Index	Local Index	Query Planning	Analytics	Materialized Views

Elasticsearch (CP)

Elasticsearch
Model:
Search Engine
License:
Elastic/Apache 2.0
Written in:
Java

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

Classification: Elasticsearch

Techniques

Sharding	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared Disk
Replication	Trans-action Protocol	Sync. Replica-tion	Async. Replica-tion	Primary Copy	Update Anywhere
Storage Management	Logging	Update-in-Place	Caching	In-Memory	Append-Only Storage
Query Processing	Global Index	Local Index	Query Planning	Analytics	Materialized Views

Other Systems

Graph databases

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

Other Systems

Key-Value Stores

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

Other Systems

Document Stores

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

Other Systems

Wide-Column Stores

- **Accumolo** (BigTable-style, cell-level security)
- **HyperTable** (BigTable-style, written in C++)

Other Systems

NewSQL Systems

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

Summary

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

Open Research Questions

For Scalable Data Management

- **Service-Level Agreements**
  - How can SLAs be guaranteed in a virtualized, multi-tenant cloud environment?
- **Consistency**
  - Which consistency guarantees can be provided in a geo-replicated system without sacrificing availability?
- **Performance & Latency**
  - How can a database deliver low latency in face of distributed storage and application tiers?
- **Transactions**
  - Can ACID transactions be aligned with NoSQL and scalability?

Distributed Transactions

ACID and Serializability

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

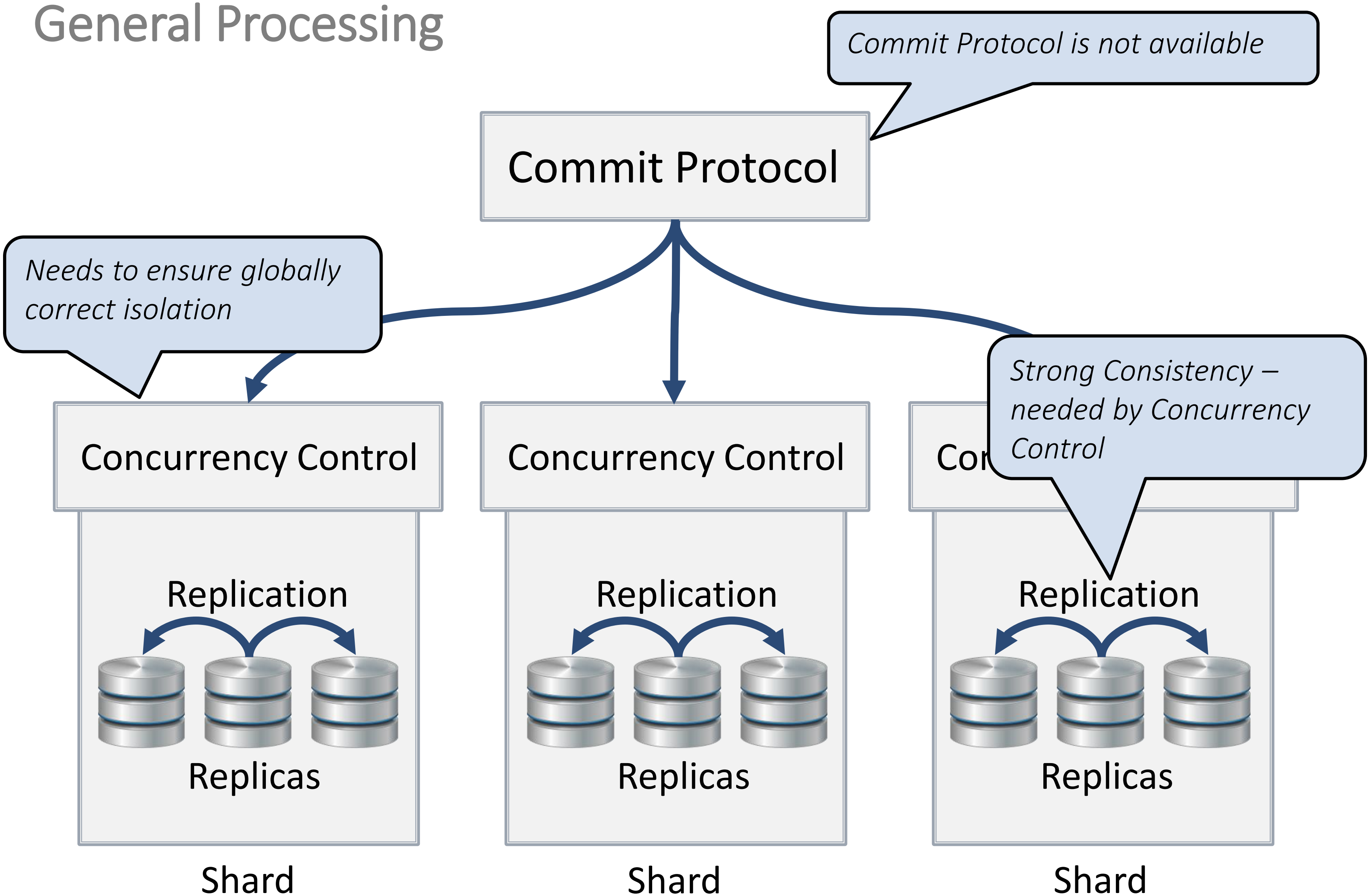
- Atomicity
- Consistency
- Isolation
- Durability

Isolation Levels:

1. Serializability
2. Snapshot Isolation
3. Read-Committed
4. Read-Atomic
5. …

Distributed Transactions

General Processing



Distributed Transactions

In NoSQL Systems – An Overview

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

Distributed Transactions

Megastore – Synchronous Wide-Area Replication

Spanner

Idea:

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

Transactions:

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

J. Corbett et al. "Spanner: Google's globally distributed database." TOCS 2013

Percolator

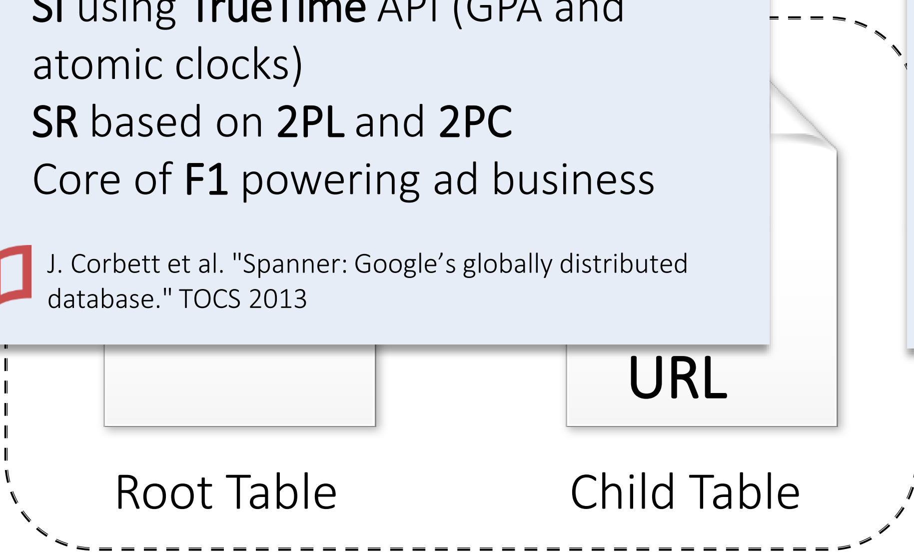
Idea:

- Indexing and transactions based on BigTable

Implementation:

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

Peng, Daniel, and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications." OSDI 2010.



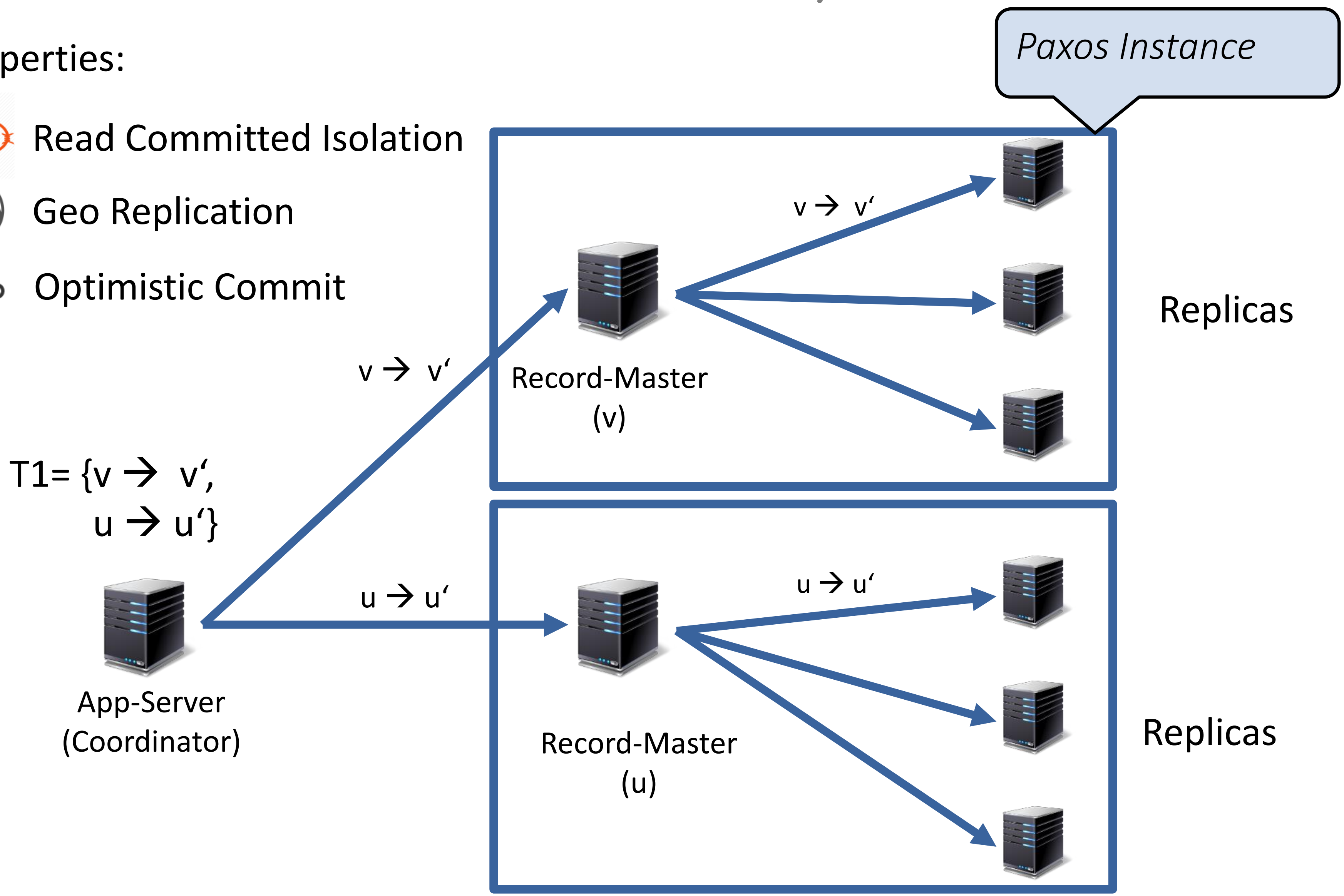
optimistic concurrency control

Distributed Transactions

MDCC – Multi Datacenter Concurrency Control

Properties:

- Read Committed Isolation
- Geo Replication
- Optimistic Commit



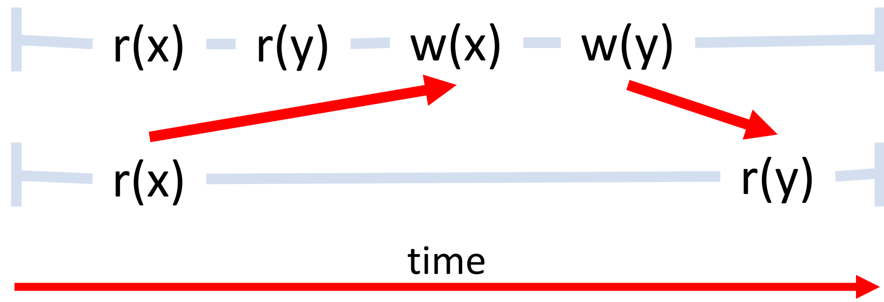
Distributed Transactions

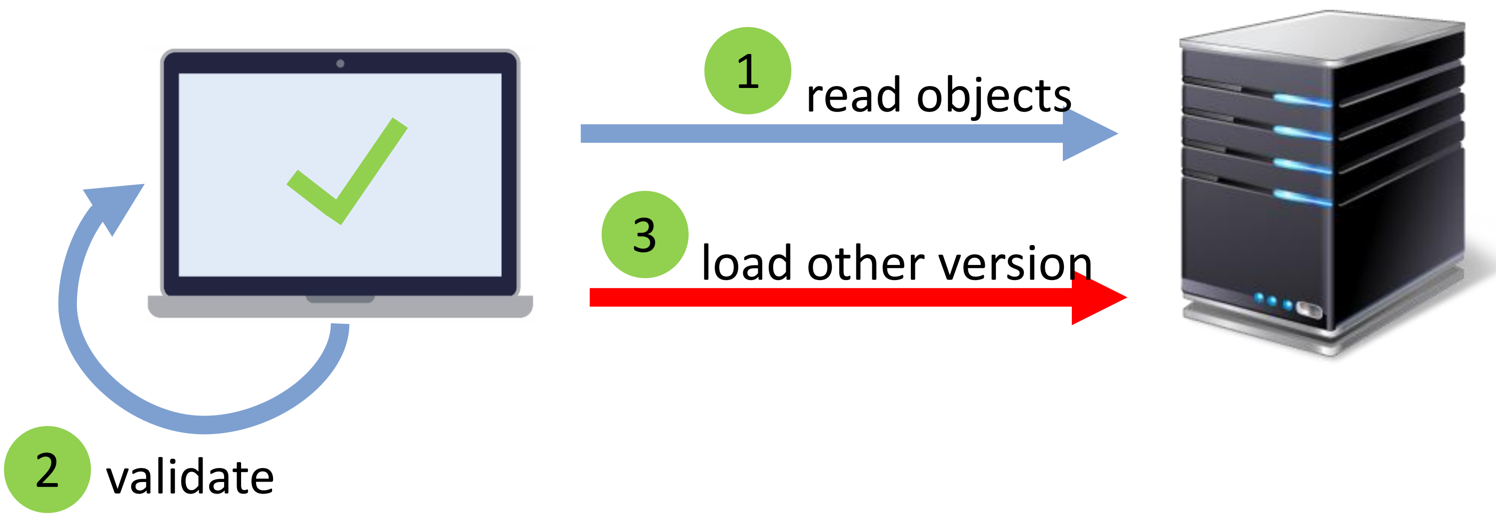
RAMP – Read Atomic Multi Partition Transactions

Properties:

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

Fractured Read

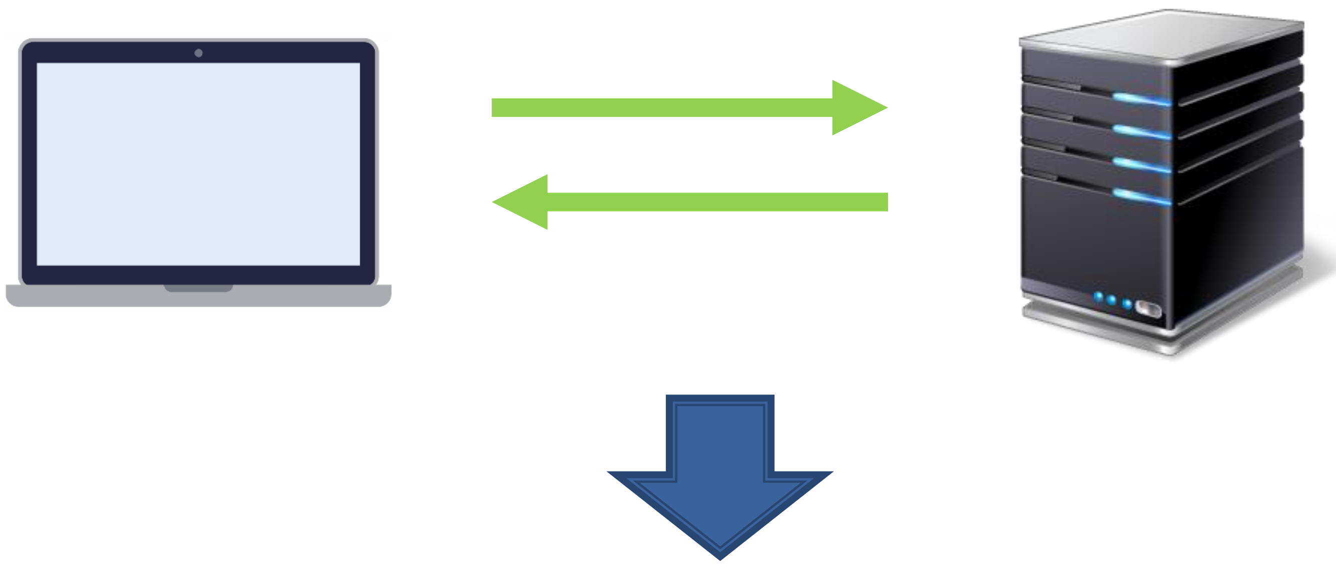




Distributed Transactions in the Cloud

The Latency Problem

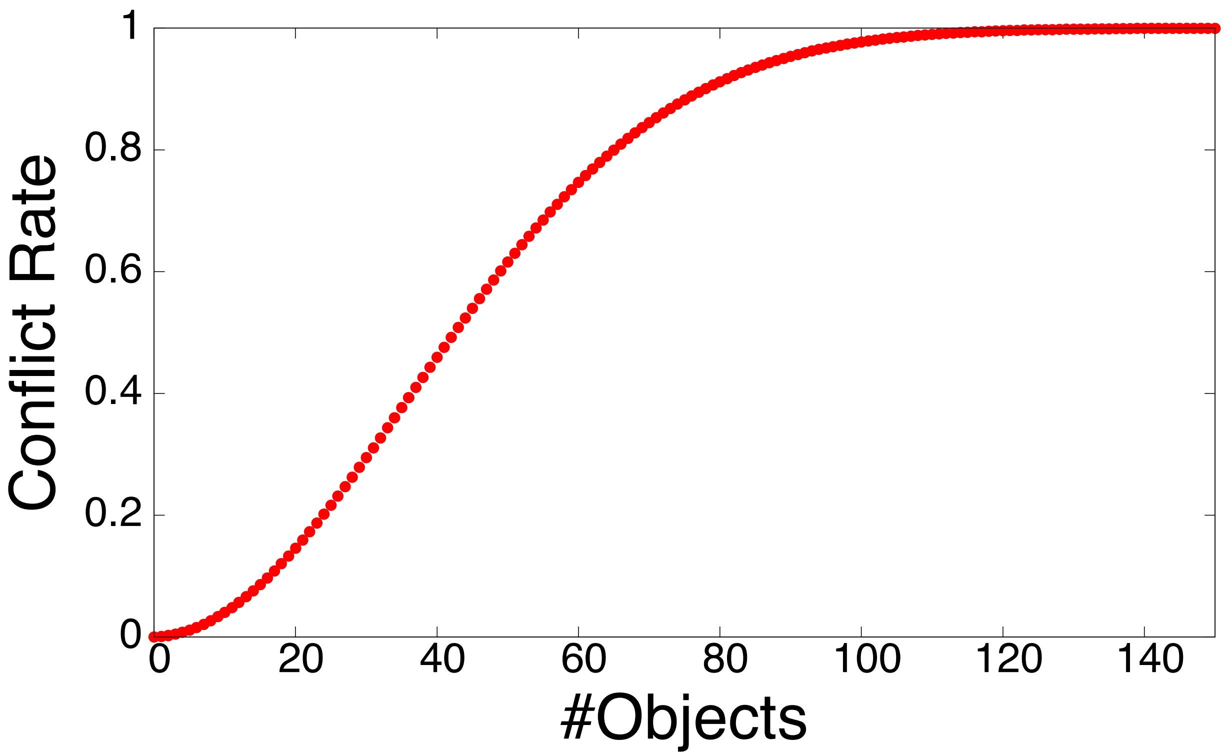
Interactive Transactions:



Optimistic Concurrency Control

Optimistic Concurrency Control

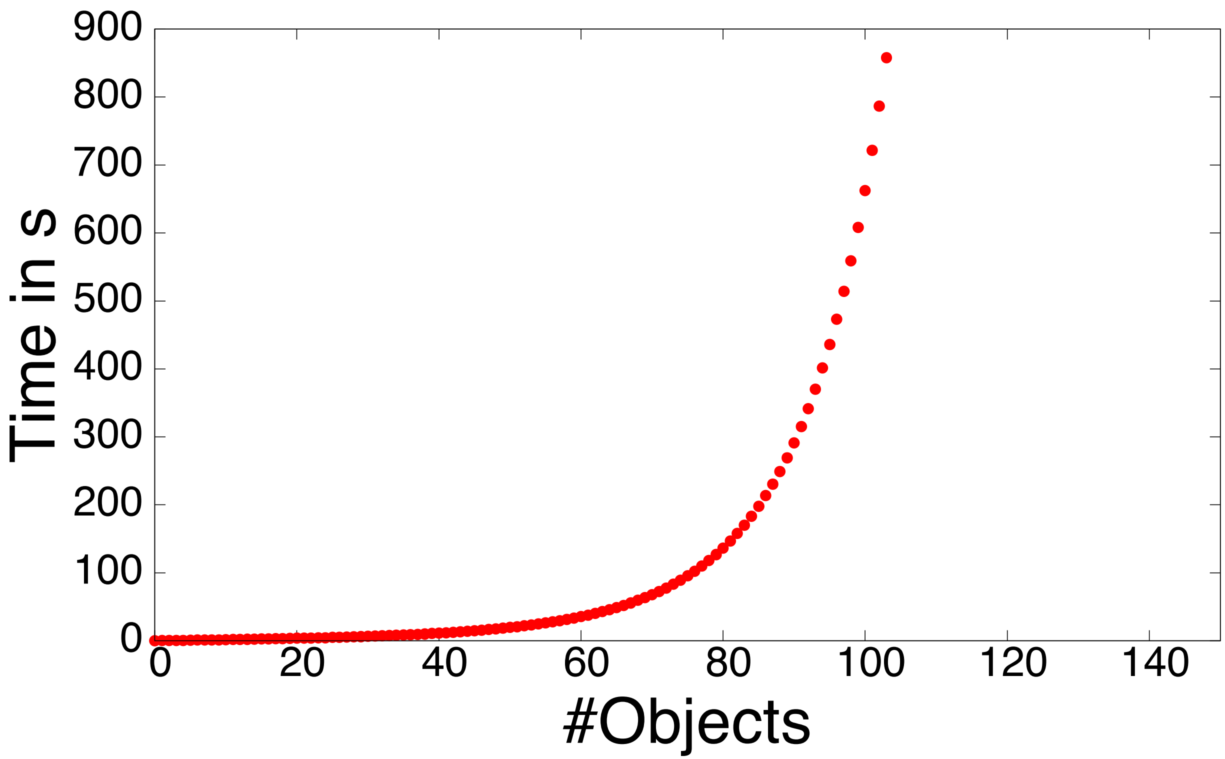
The Abort Rate Problem



- 10.000 objects
- 20 writes per second
- 95% reads

Optimistic Concurrency Control

The Abort Rate Problem

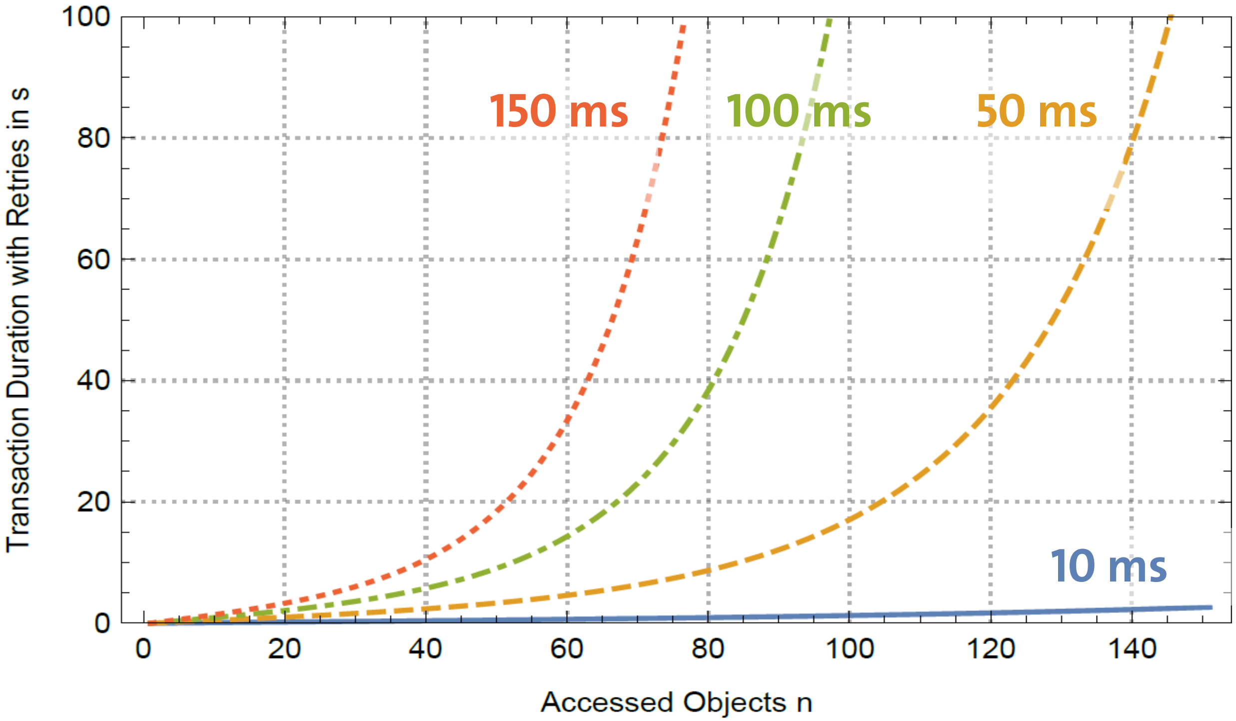


- 10.000 objects
- 20 writes per second
- 95% reads

Our line of work for improving scalable transaction processing.

Problem of Optimistic Transactions

Abort Rates Depend on Latency

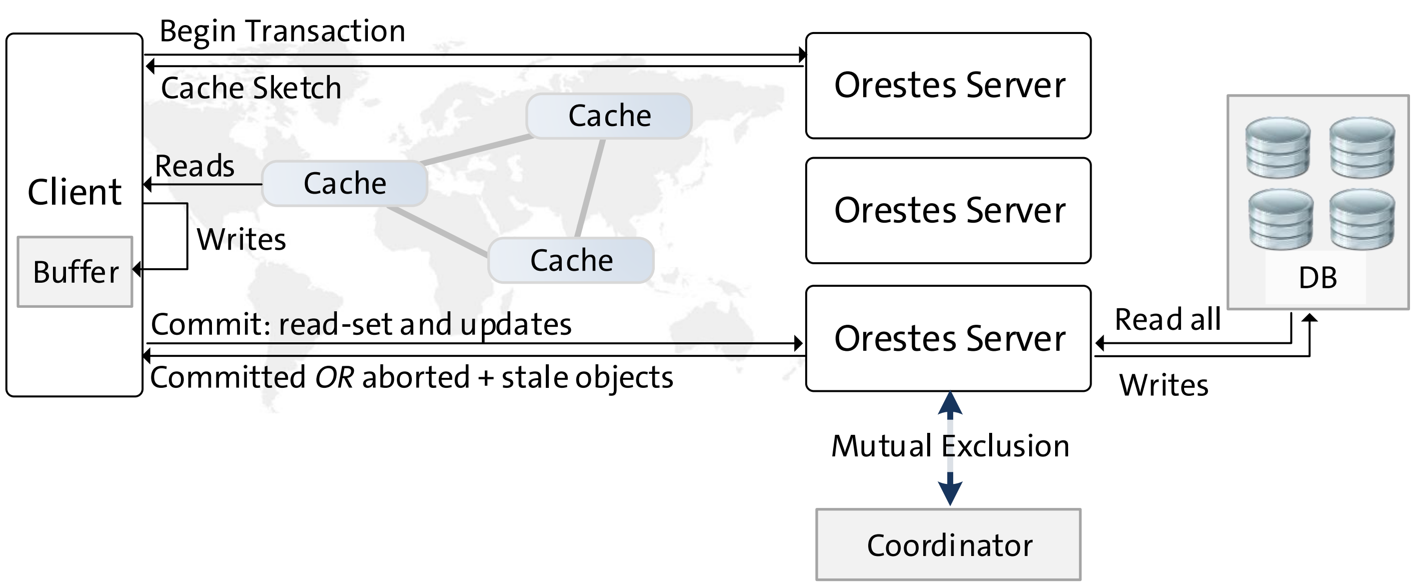


Transaction **Abort Rates** Increase **Exponentially** with **Latency**

Distributed Cache-Aware Transaction

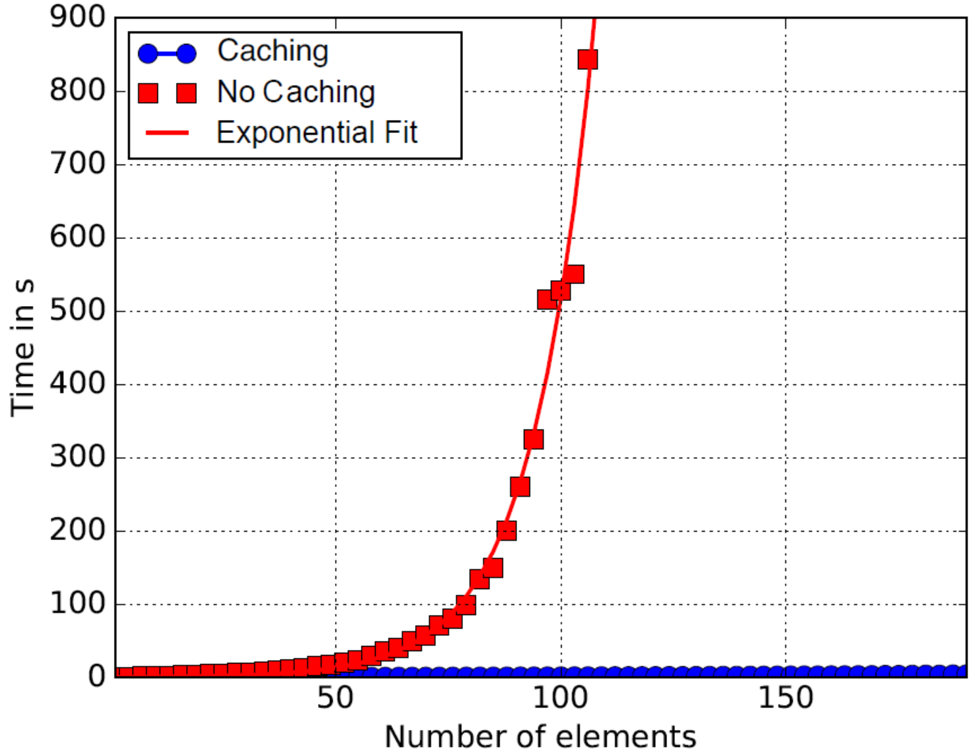
Scalable ACID Transactions

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



Distributed Cache-Aware Transaction

Speed Evaluation



15×

Faster Transactions

7×

More Objects Before Exceeding 2 Seconds

Selected Research Challanges

Encrypted Databases

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

SQL-Proxy

Encrypts and decrypts,

RDBMS

Relational Cloud

DBaaS Architecture:

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

C. Curino, et al. "Relational cloud: A database-as-a-service for the cloud.", CIDR 2011

- Early approach
- Not adopted in practice, yet

Dream solution:

Full Homomorphic Encryption

Research Challanges

Transactions and Scalable Consistency

	Consisten	mit	Data
Dynamo	Eventual		
Yahoo PNuts	Timeline pe		
COPS	Causality		
MySQL (async)	Serializable		
Megastore	Serializable	Static Partition	2 RT
Spann	Snapshot Isolation	Partition	2 RT
MDCC	Read		-

Google's F1

Idea:

- Consistent multi-data center replication with SQL and ACID transaction

Implementation:

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

Shute, Jeff, et al. "F1: A distributed SQL database that scales." Proceedings of the VLDB 2013.

Currently very few NoSQL DBs implement consistent Multi-DC replication

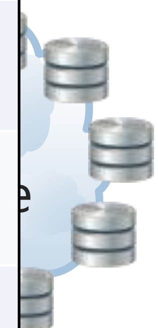
Selected Research Challanges

NoSQL Benchmarking

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

Read()

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



3. Popularity Distribution

Selected Research Challanges

NoSQL Benchmarking

YCSB++

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

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

YCSB+T

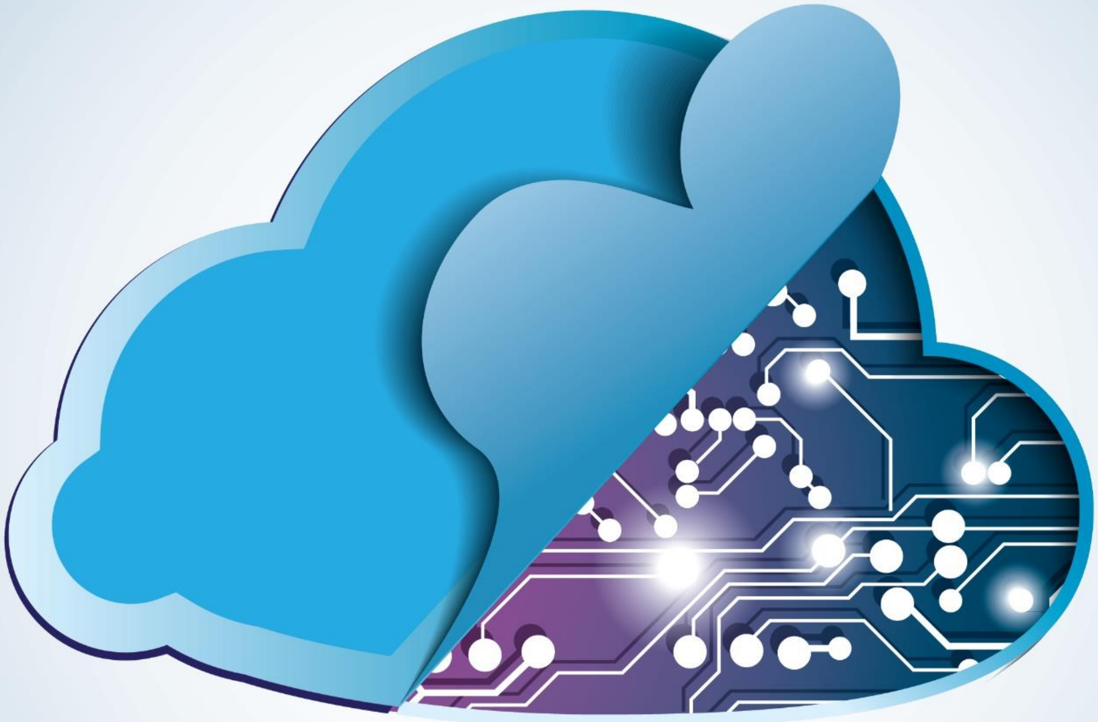
- **New workload:** Transactional Bank Account
- Simple anomaly detection for Lost Updates
- No comparison of systems

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

Weaknesses:

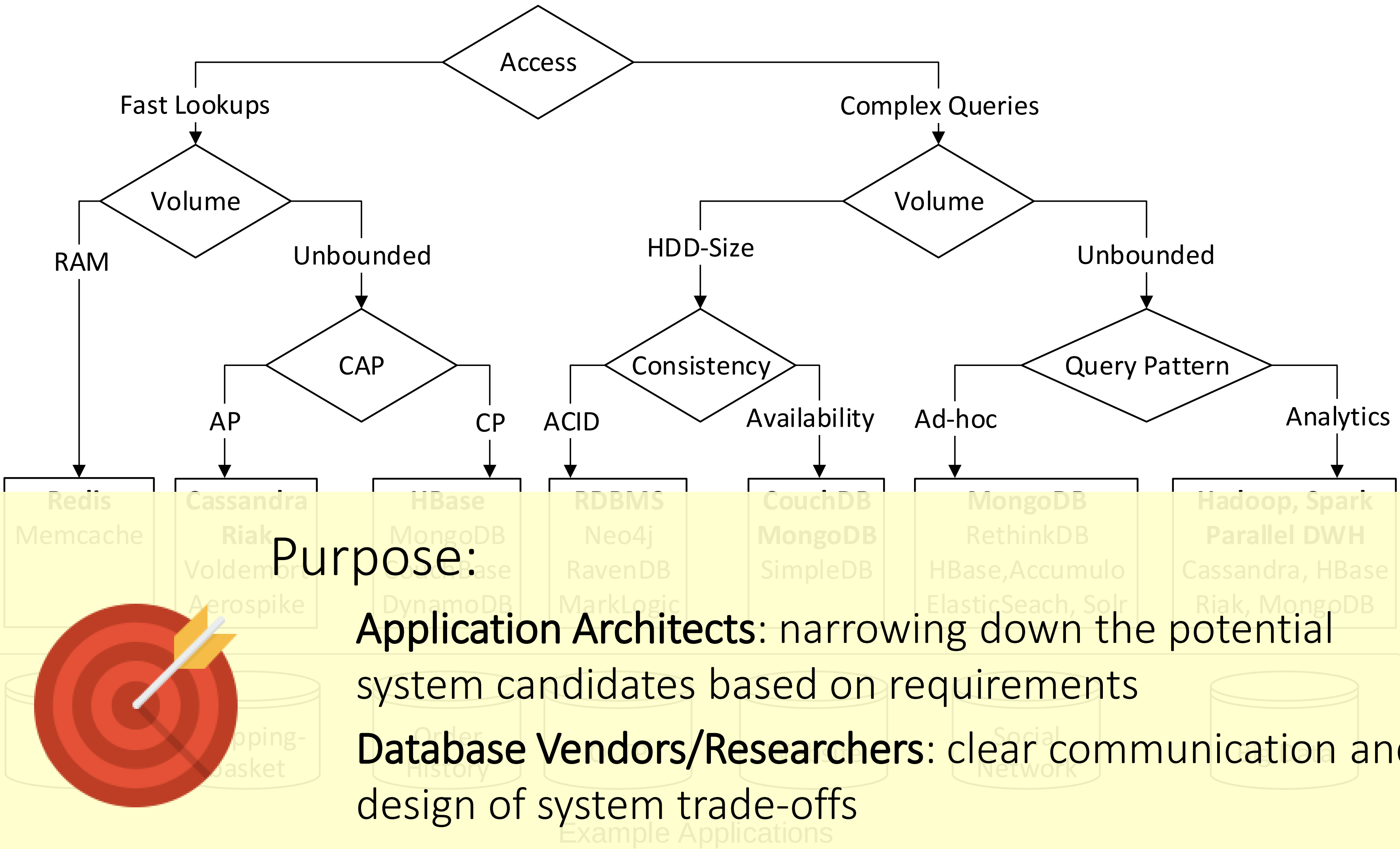
- Single client can be a bottleneck
- No consistency & availability measurement
- No Transaction Support

No specific application
→ CloudStone, CARE, TPC extensions?



How can the choices for an appro-
priate system be narrowed down?

NoSQL Decision Tree



Purpose:

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

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

System Properties

According to the NoSQL Toolbox

- For fine-grained system selection:

	Functional Requirements							
	Scan Queries	**ACID Transactions**	**Conditional Writes**	**Joins**	**Sorting**	**Filter Query**	**Full-Text Search**	**Analytics**
Mongo	x		x		x	x	x	x
Redis	x	x	x					
HBase	x		x		x			x
Riak							x	x
Cassandra	x		x		x		x	x
MySQL	x	x	x	x	x	x	x	x

System Properties

According to the NoSQL Toolbox

- For fine-grained system selection:

	Non-functional Requirements										
	Data Scalability	Write Scalability	Read Scalability	Elasticity	Consistency	Write Latency	Read Latency	Write Throughput	Read Availability	Write Availability	Durability
Mongo	x	x	x		x	x	x		x		x
Redis			x		x	x	x	x	x		x
HBase	x	x	x	x	x	x		x			x
Riak	x	x	x	x		x	x	x	x	x	x
Cassandra	x	x	x	x		x		x	x	x	x
MySQL			x		x						x

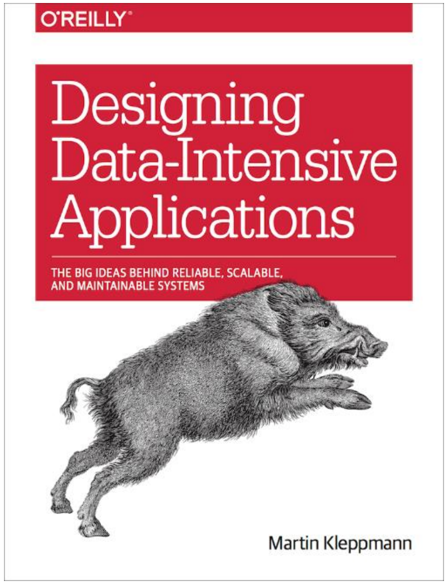
System Properties

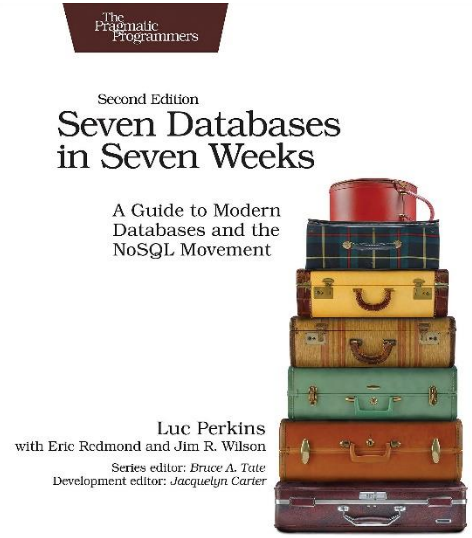
According to the NoSQL Toolbox

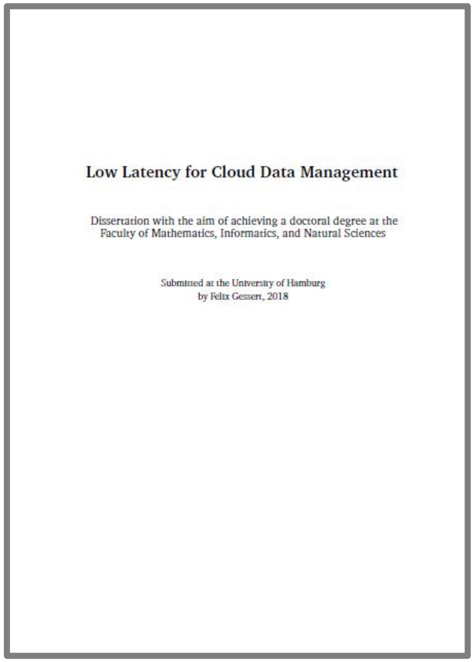
- For fine-grained system selection:

	Techniques																			
	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared-Disk	Transaction Protocol	Sync. Replication	Async. Replication	Primary Copy	Update Anywhere	Logging	Update-in-Place	Caching	In-Memory	Append-Only Storage	Global Indexing	Local Indexing	Query Planning	Analytics Framework	Materialized Views
Mongo	x	x					x	x	x		x		x	x	x		x	x	x	
Redis								x	x		x		x							
HBase	x						x		x		x		x		x					
Riak		x		x				x		x	x	x	x			x	x		x	
Cassandra		x		x				x		x	x		x		x	x	x			x
MySQL					x			x	x		x	x	x				x	x		

Literature



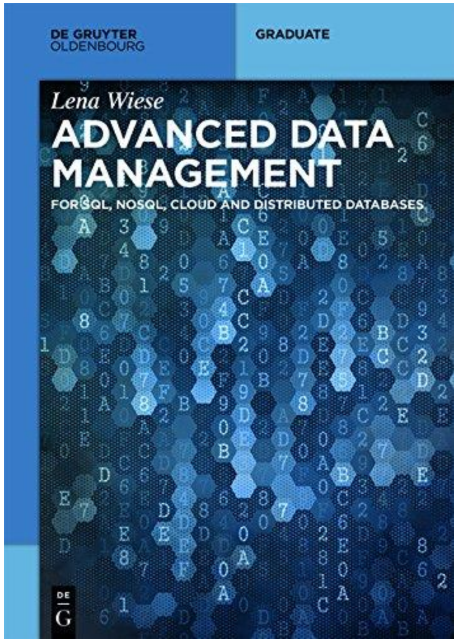




Low Latency for Cloud Data Management

Dissertation with the aim of achieving a doctoral degree at the Faculty of Mathematics, Informatics, and Natural Sciences

Submitted at the University of Hamburg
by Felix Gessert, 2018



Future Work

Online Collaborative Decision Support

- Select **Requirements** in Web GUI:

☐ Read Scalability ☑ Conditional Writes ☑ Consistent

- System makes **suggestions** based on data from *practitioners*, *vendors* and *automated benchmarks*:

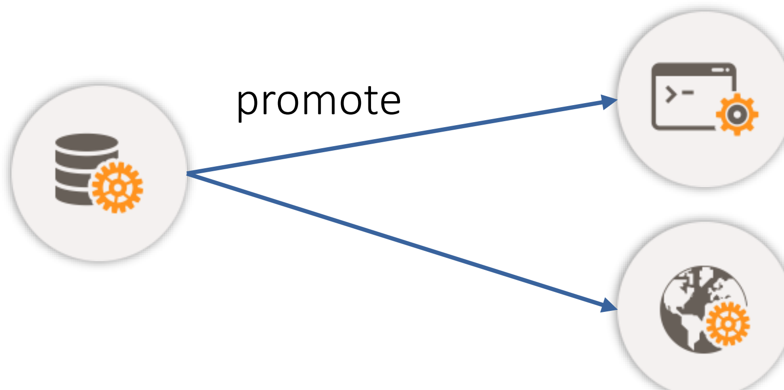
★ 4/5 4/5 3/5 redis

★ 4/5 5/5 5/5 mongoDB

Summary

- High-Level NoSQL Categories:
  - Key-Value, Wide-Column, Docuement, Graph
  - Two out of {Consistent, Available, Partition Tolerant}
- The **NoSQL Toolbox**: systems use similar techniques that promote certain capabilities

Techniques
Sharding, Replication, Storage Management, Query Processing



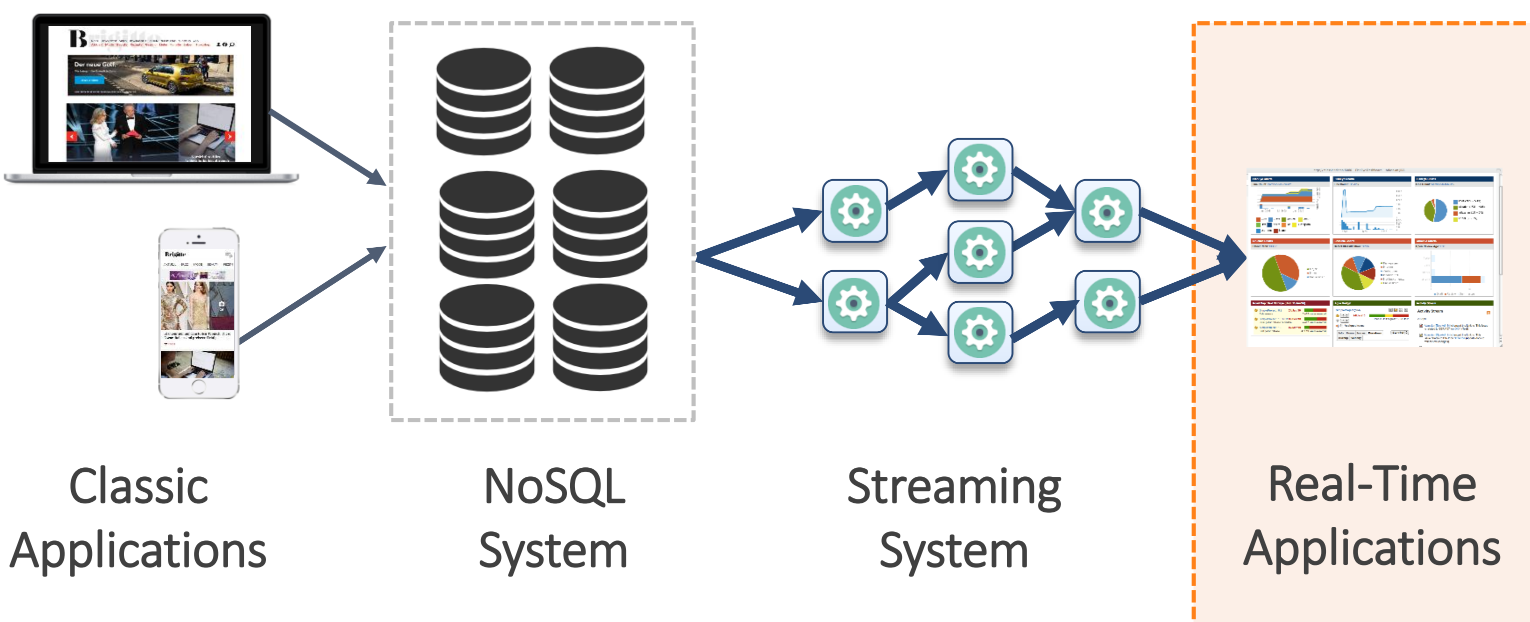
Functional Requirements

Non-functional Requirements

- **Decision Tree**

Summary

- Current NoSQL systems very good at scaling:
  - Data storage
  - Simple retrieval
- But how to handle real-time queries?



NoSQL & Real-Time Data Management

In Research & Practice – Part 2

Wolfram Wingerath, Felix Gessert, Norbert Ritter

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

March 5, BTW 2019, Rostock

Universität Hamburg

www.baqend.com

Outline

Introduction
Where From? Where To?

Stream Processing
Big Data + Low Latency

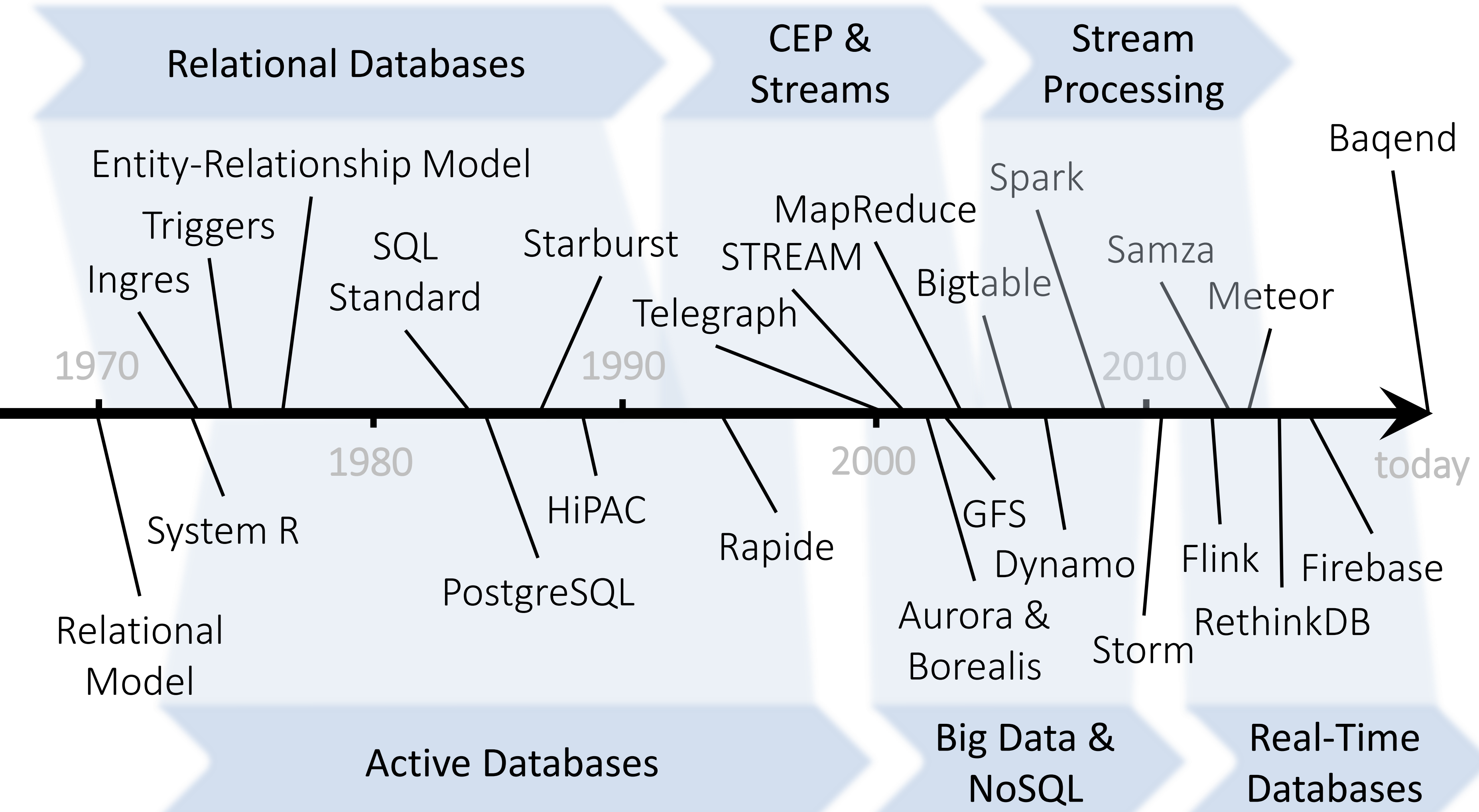
Real-Time Databases
Push-Based Collections

Future Directions
Current Research & Outlook

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

A Short History of Data Management

Hot Topics Through The Ages

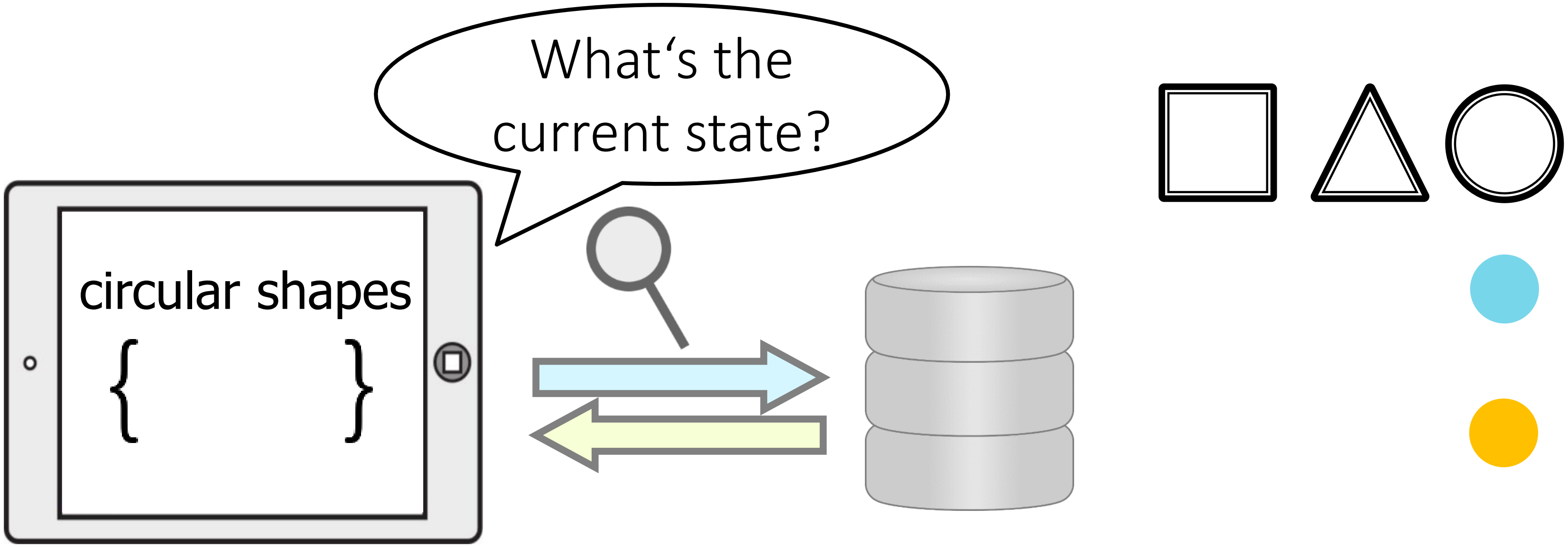


TRIGGERS & MORE

Active Database Features

Databases are Passive

Challenge: How to Build Reactive Applications?



Periodic Polling for query result maintenance:

→ inefficient

→ slow

Active Database Features

Modeling Behavioral Domain Aspects

Triggers: simple action-mechanisms

- Use cases:
  - (Referential) integrity
  - Change data capture

ECA rules: Event-Condition-Action

- Captures **composite events**
- More expressive than triggers (**rule languages**)
- Advanced use cases:
  - Materialized view maintenance
  - Pattern recognition
  - (complex) event processing

View Maintenance

Keeping Track of Query Results

Materialized Views: precomputed query results

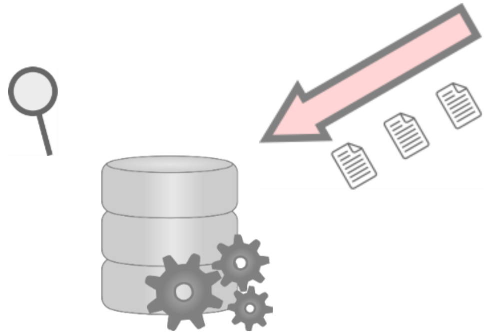
- Used to speed up pull-based queries, e.g in data warehouses
- Implementation aspects:
  - Eager vs. lazy
  - Incremental vs. recomputation-based
  - Partial maintenance vs. full maintenance
  - Self-maintainability vs. expressiveness

Change Notification Mechanisms: inform subscribers of possibly invalidated query results

- Used to invalidate caches in the middle tier (cf. 3-tier stack)

View Maintenance By Example

Matching Every Query Against Every Update



→ Potential ***bottlenecks***:

- *Number of queries*
- *Write throughput*
- *Query complexity*

Similar processing for:

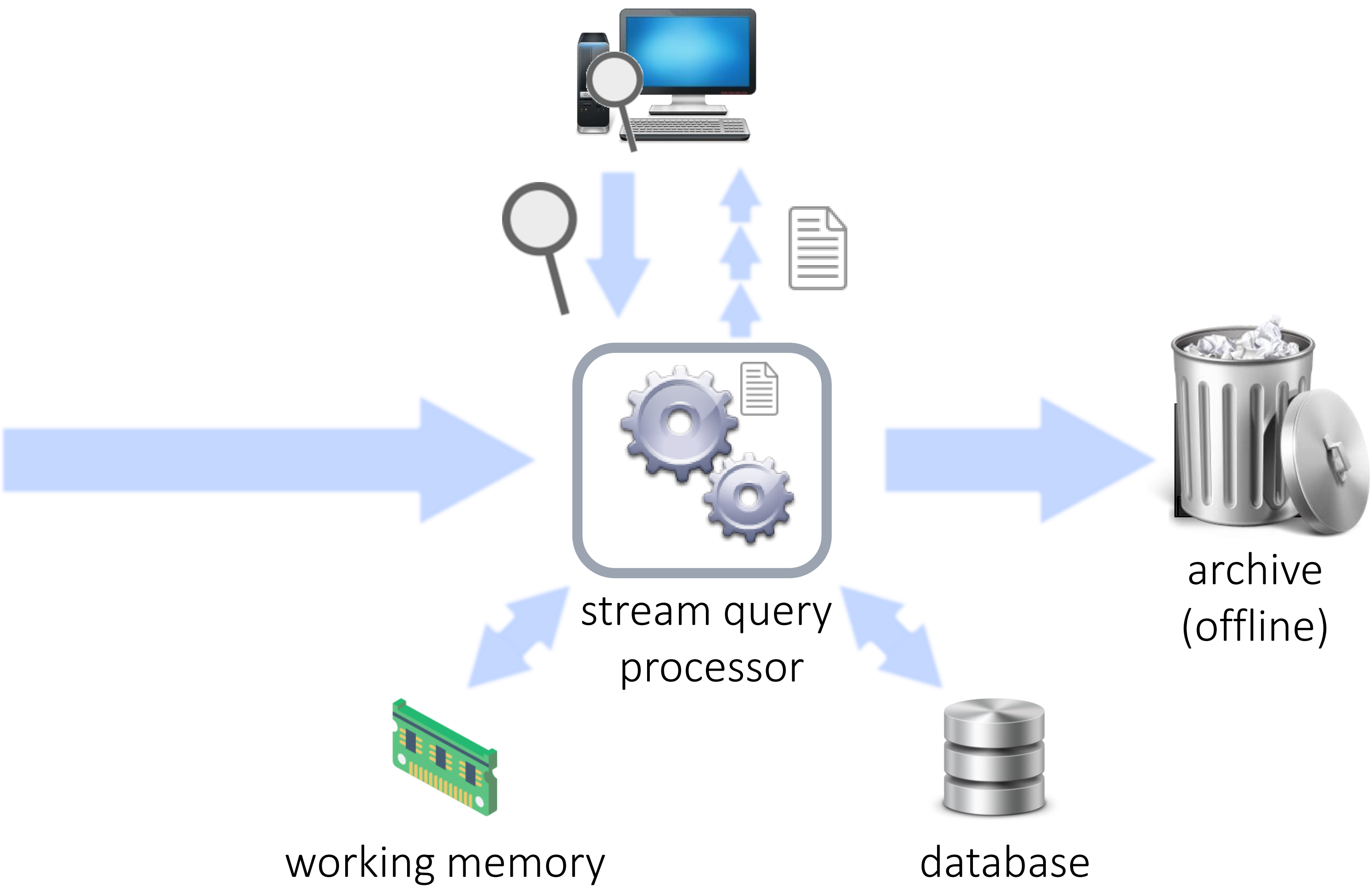
- Triggers
- ECA rules

EVOLVING DOMAINS

Data Stream Management

Data Stream Management Systems

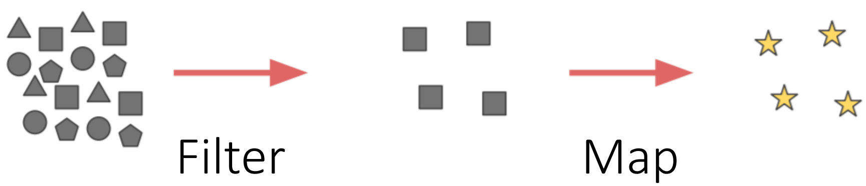
High-Level Architecture



Typical Stream Operators

Examples

Filter & Transform

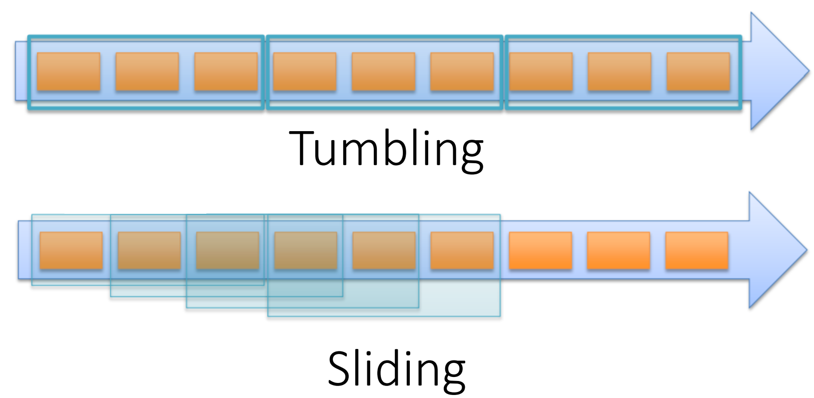


Group

Aggregates

SUM()
COUNT()

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**

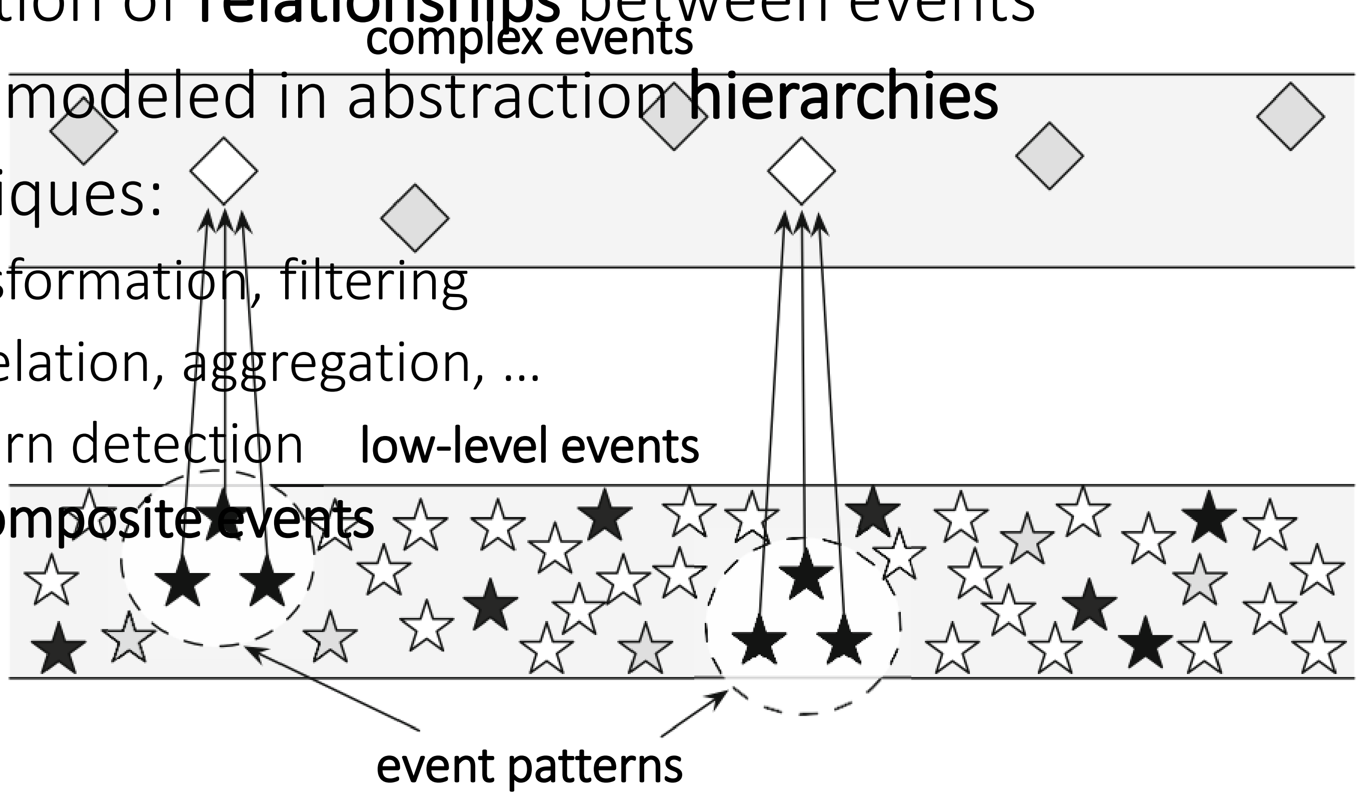


Illustration taken from: Bruns, R. & Dunkel, J, *Complex Event Processing: Komplexe Analyse von massiven Datenströmen mit CEP (2015). Springer Vieweg, 2015*

Notions of Time

Arrival Time vs. Event Time

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

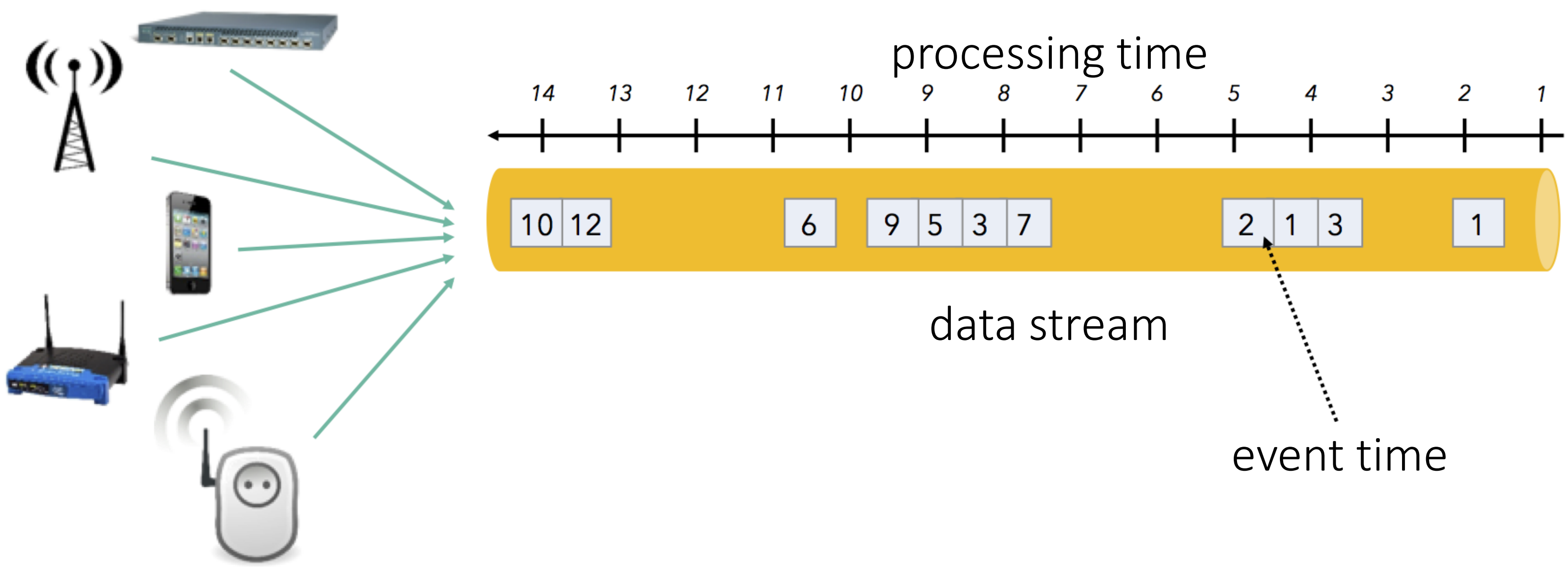
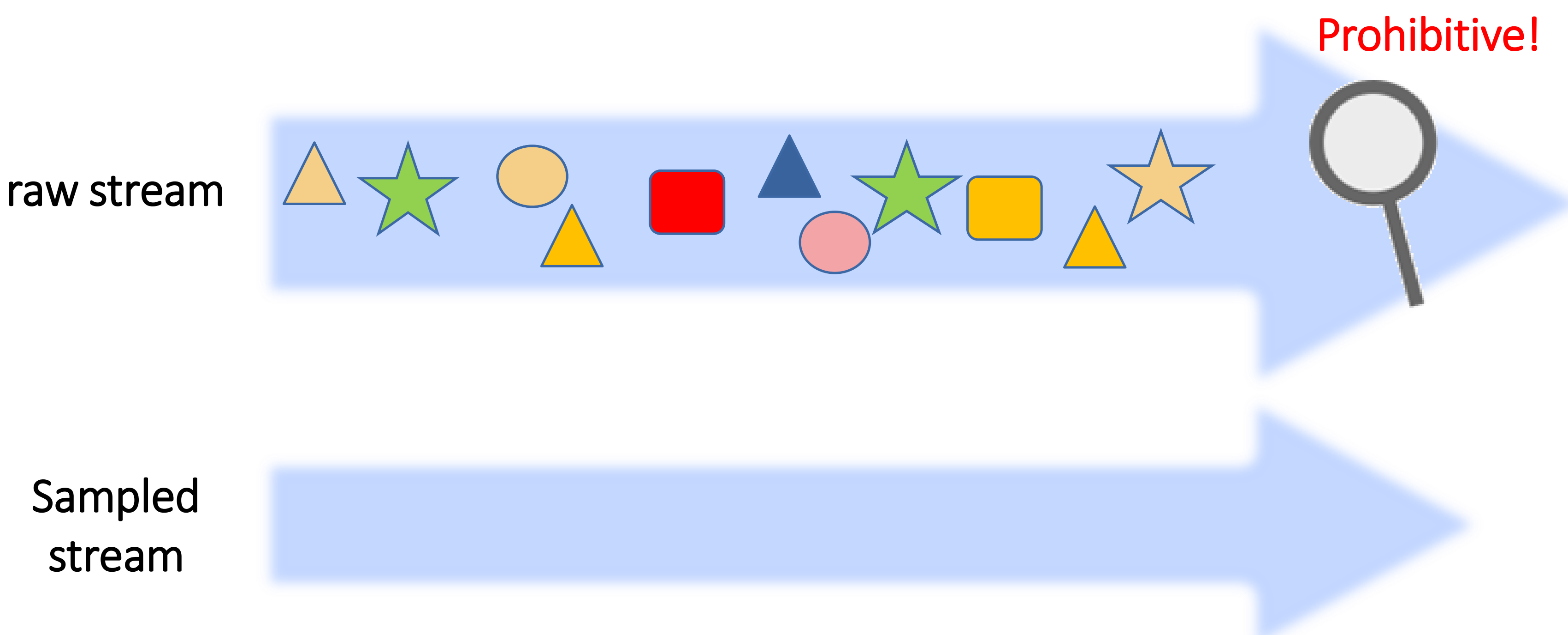


Illustration take from: Stephan Ewen, *How Apache Flink™ Enables New Streaming Applications, Part 1* (2015)
https://data-artisans.com/blog/how-apache-flink-enables-new-streaming-applications-part-1 (2018-03-16)

Approximation & Load Shedding

Provide the „Best“ Answer While Avoiding to Fall Behind

- **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

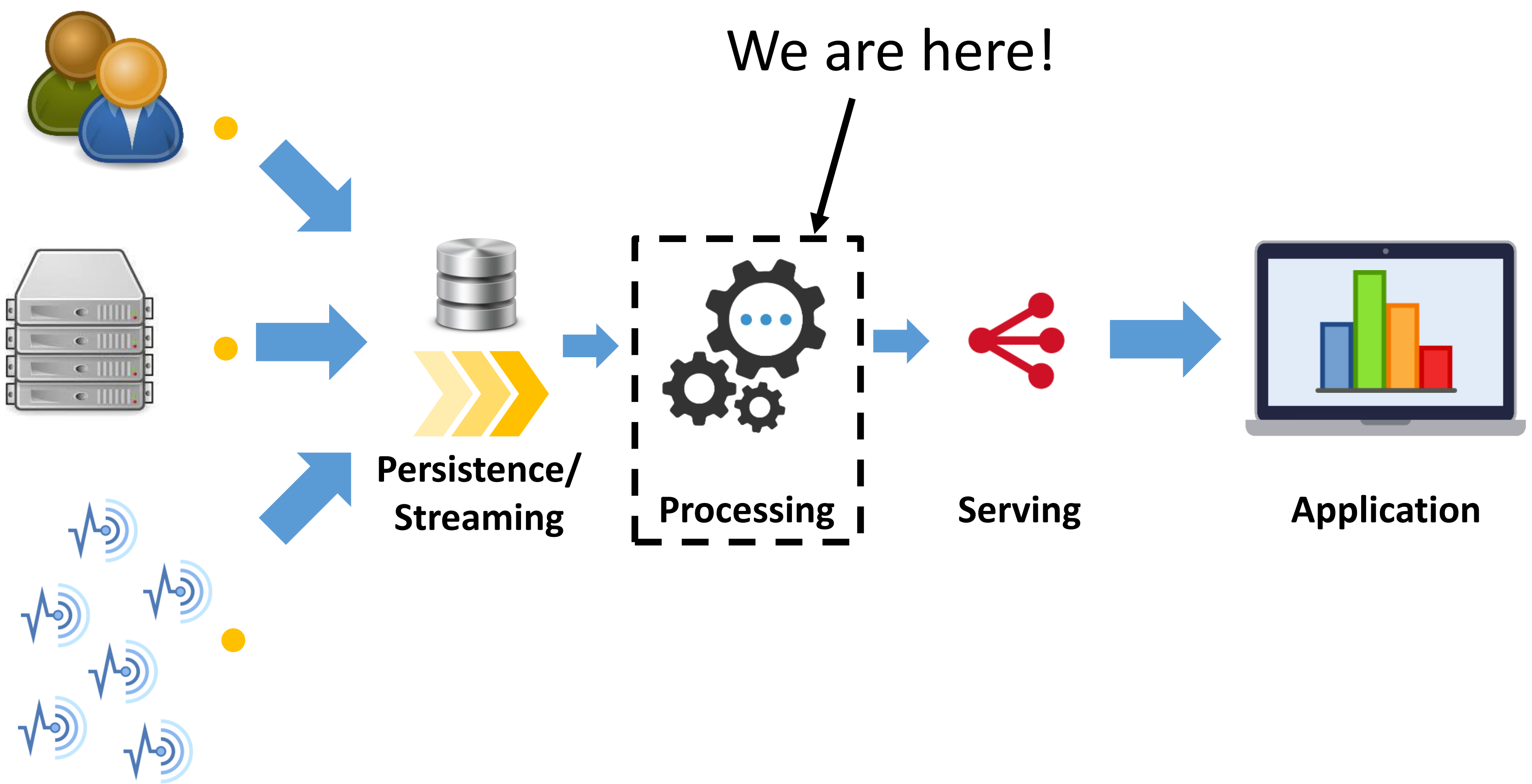
Future Directions
Current Research & Outlook

- **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.:

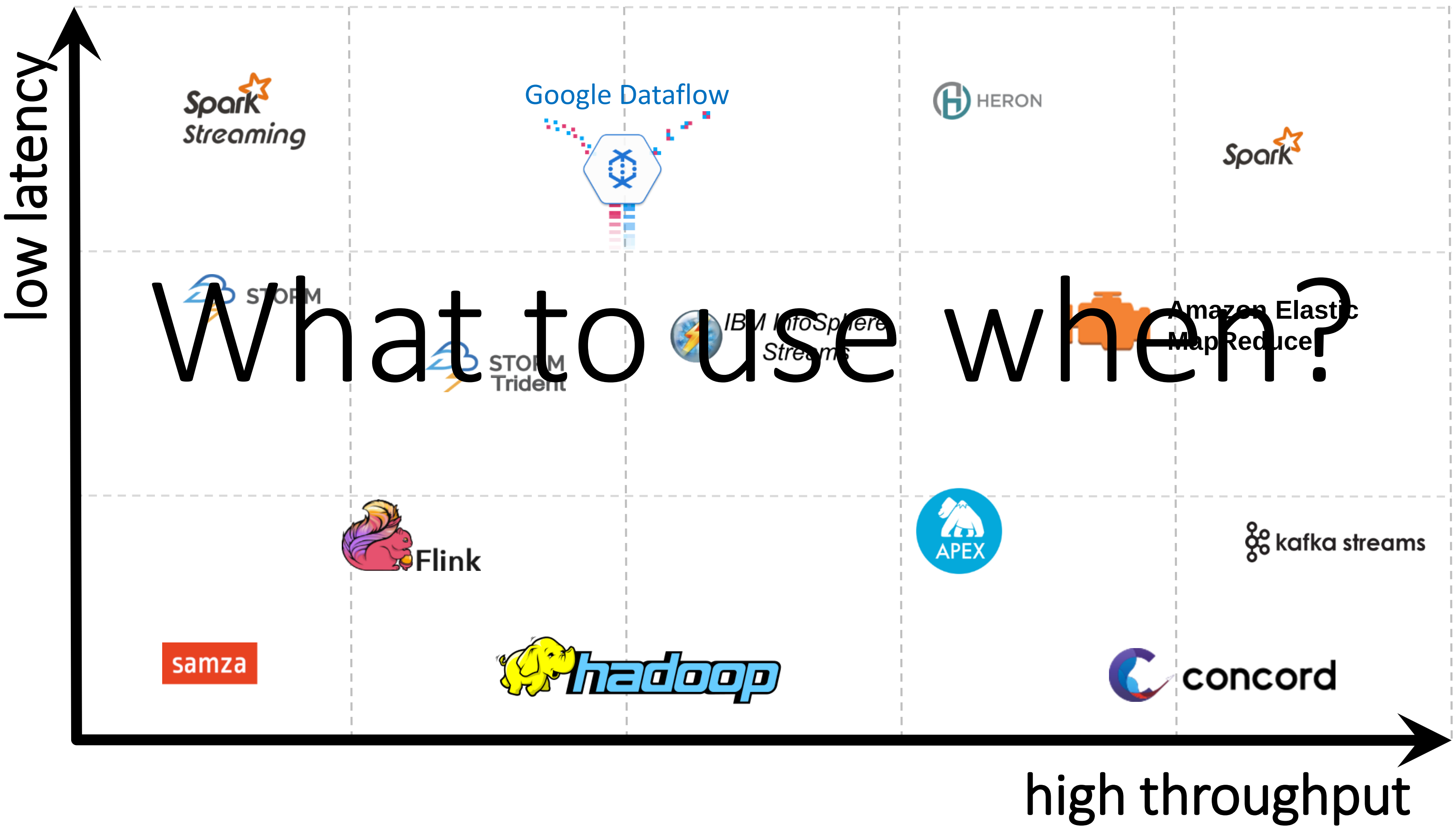
- **Deployment -** code distribution, starting/stopping work
- **Monitoring -** health checks, application stats
- **Scheduling -** assigning work, rebalancing
- **Fault-tolerance** - restarting workers, rescheduling failed work

Running in cluster

Running on single node

Big Data Processing Frameworks

What are your options?



CONCEPTS

Batch vs. Stream Processing

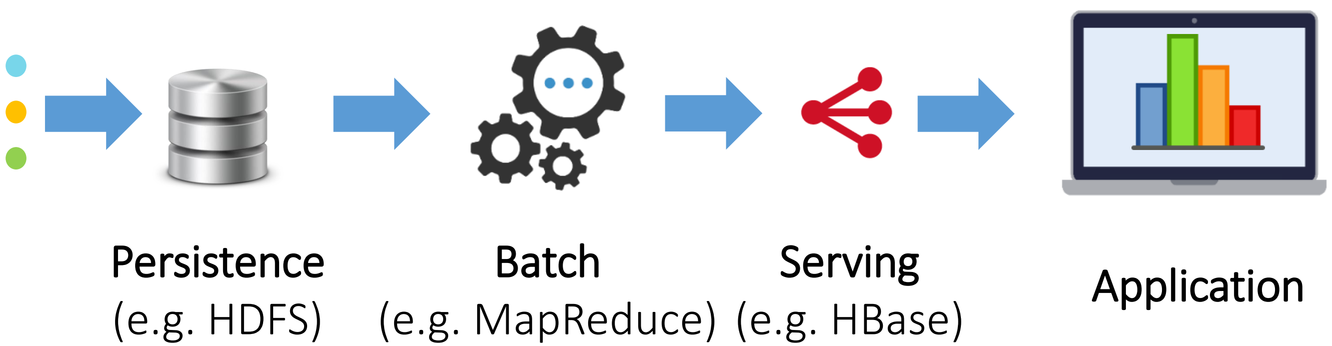
Batch Processing

„Volume“

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

But:

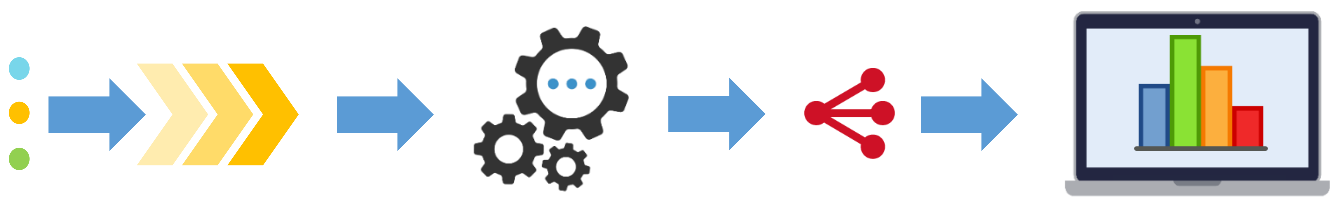
- High latency: periodic jobs (e.g. during night times)



Stream Processing

„Velocity“

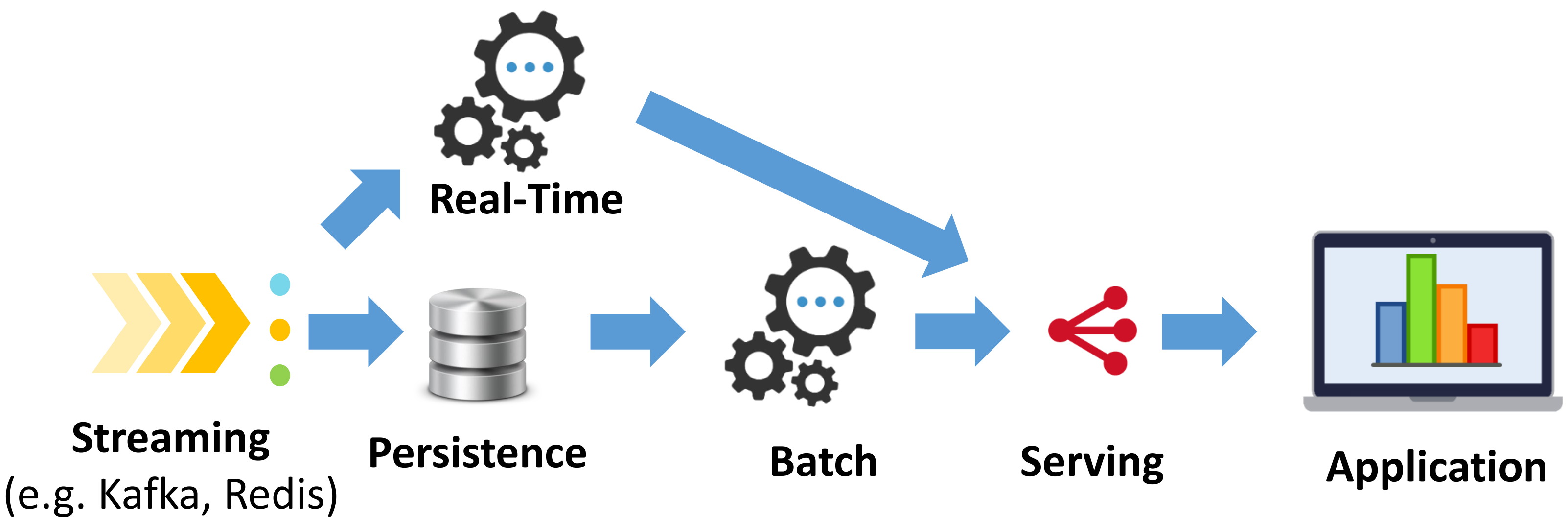
- **Low end-to-end latency**
- Challenges:
  - **Long-running jobs** - no downtime allowed
  - **Asynchronism** - data may arrive delayed or out-of-order
  - **Incomplete input** - algorithms operate on partial data
  - More: fault-tolerance, state management, guarantees, ...



Lambda Architecture

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

- **Fast** output (real-time)
- Data retention + reprocessing (batch)
  → **„eventually accurate“** merged views of real-time & batch
  Typical setups: Hadoop + Storm (→ Summingbird), Spark, Flink
- High complexity 2 code bases & 2 deployments

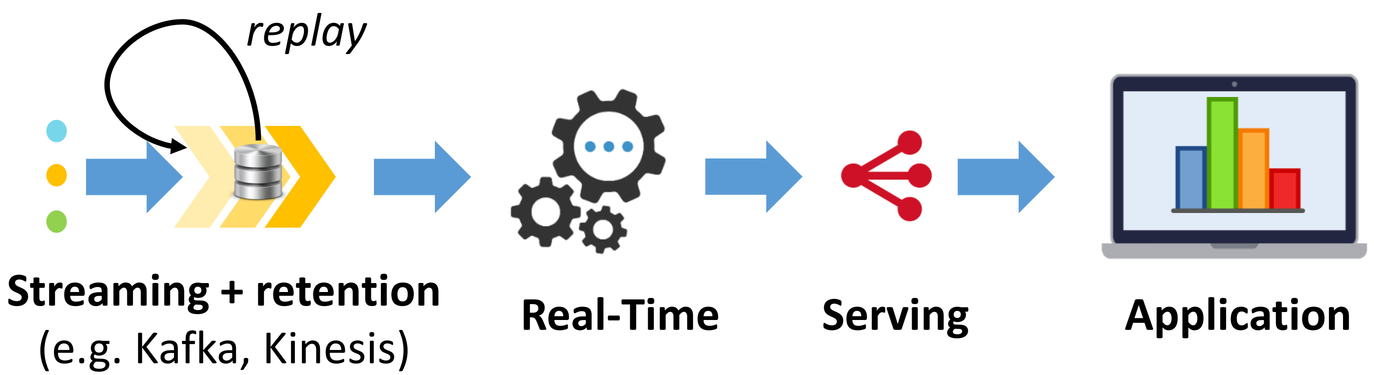


Nathan Marz, *How to beat the CAP theorem* (2011)
http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

Kappa Architecture

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 | **micro-batch** | **batch**

Flink

STORM Trident

samza

hadoop

Spark Streaming

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

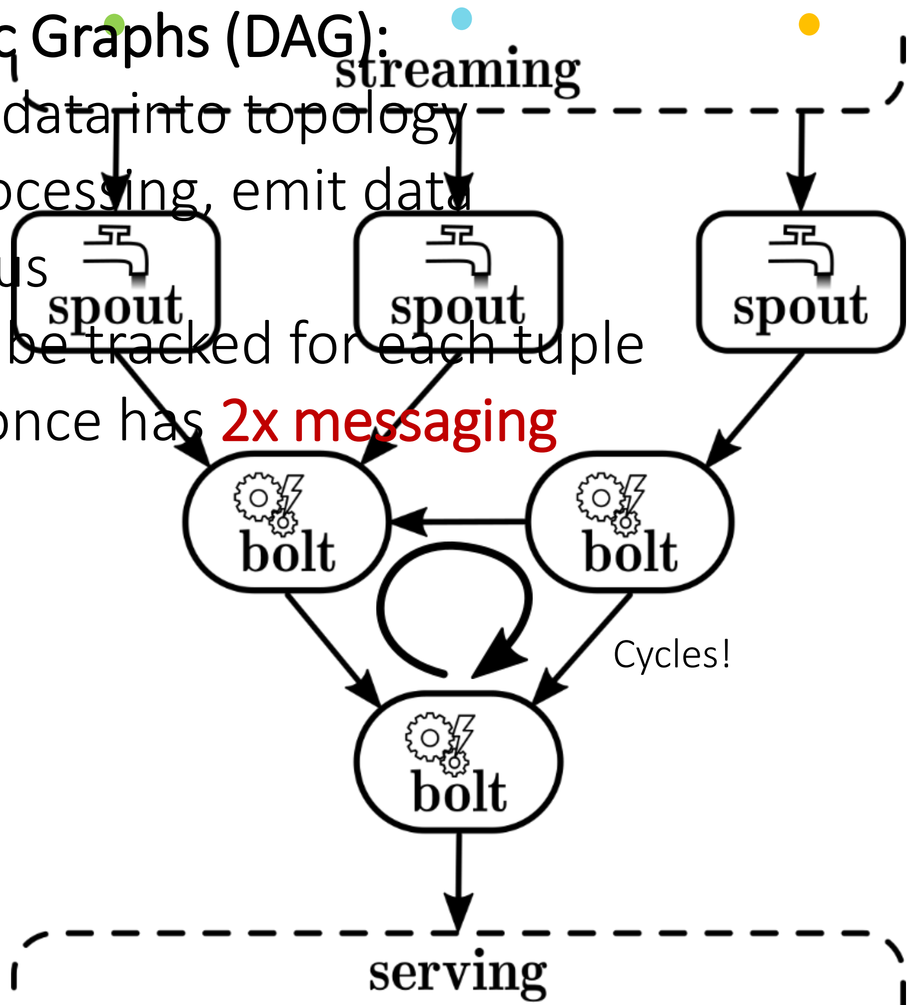
- **2010**: developed at BackType (acquired by Twitter)
- **2011**: open-sourced
- **2014**: Apache top-level project

Dataflow

STORM

Directed Acyclic Graphs (DAG):

- **Spouts**: pull data into topology
- **Bolts**: do processing, emit data
- Asynchronous
- Lineage can be tracked for each tuple
  → At-least-once has 2x messaging overhead

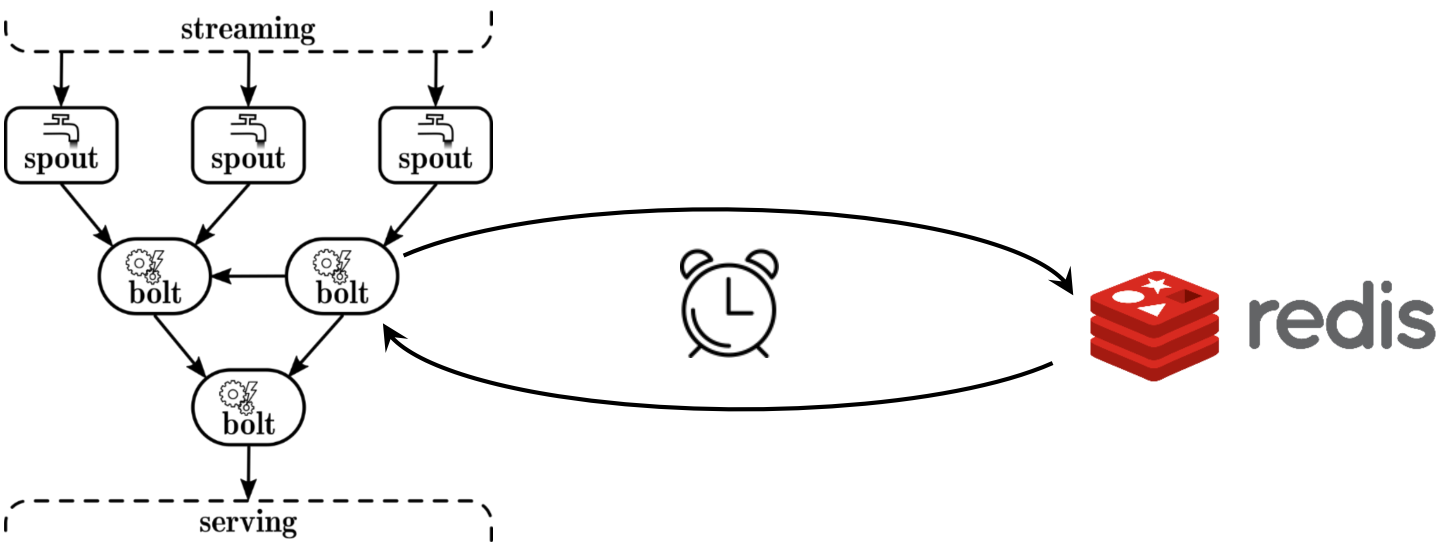


State Management

Recover State on Failure

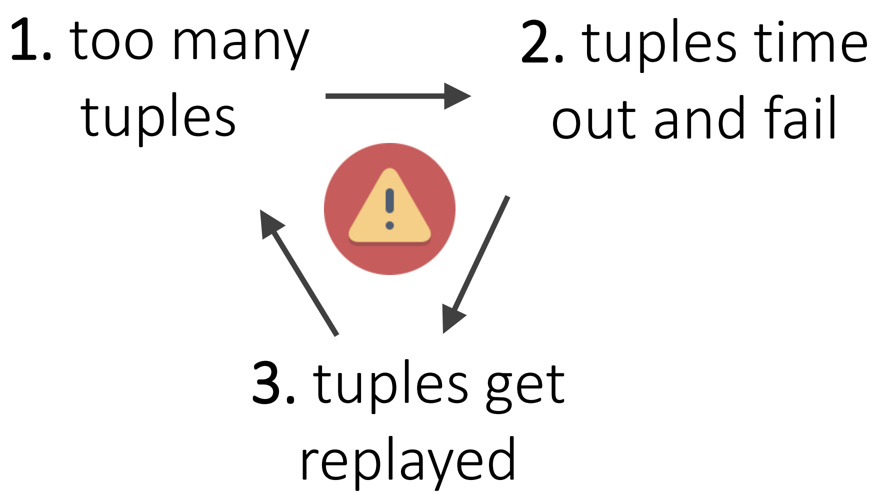
- **In-memory** or **Redis**-backed reliable state
- *Synchronous state communication* on the critical path

  → infeasible for large state



Back Pressure

Throttling Ingestion on Overload



Approach: monitoring bolts' inbound buffer

1. Exceeding **high watermark** → throttle!
2. Falling below **low watermark** → full power!

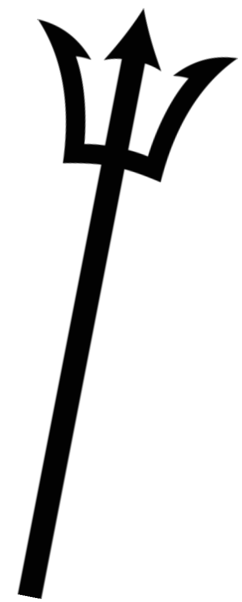
Trident

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

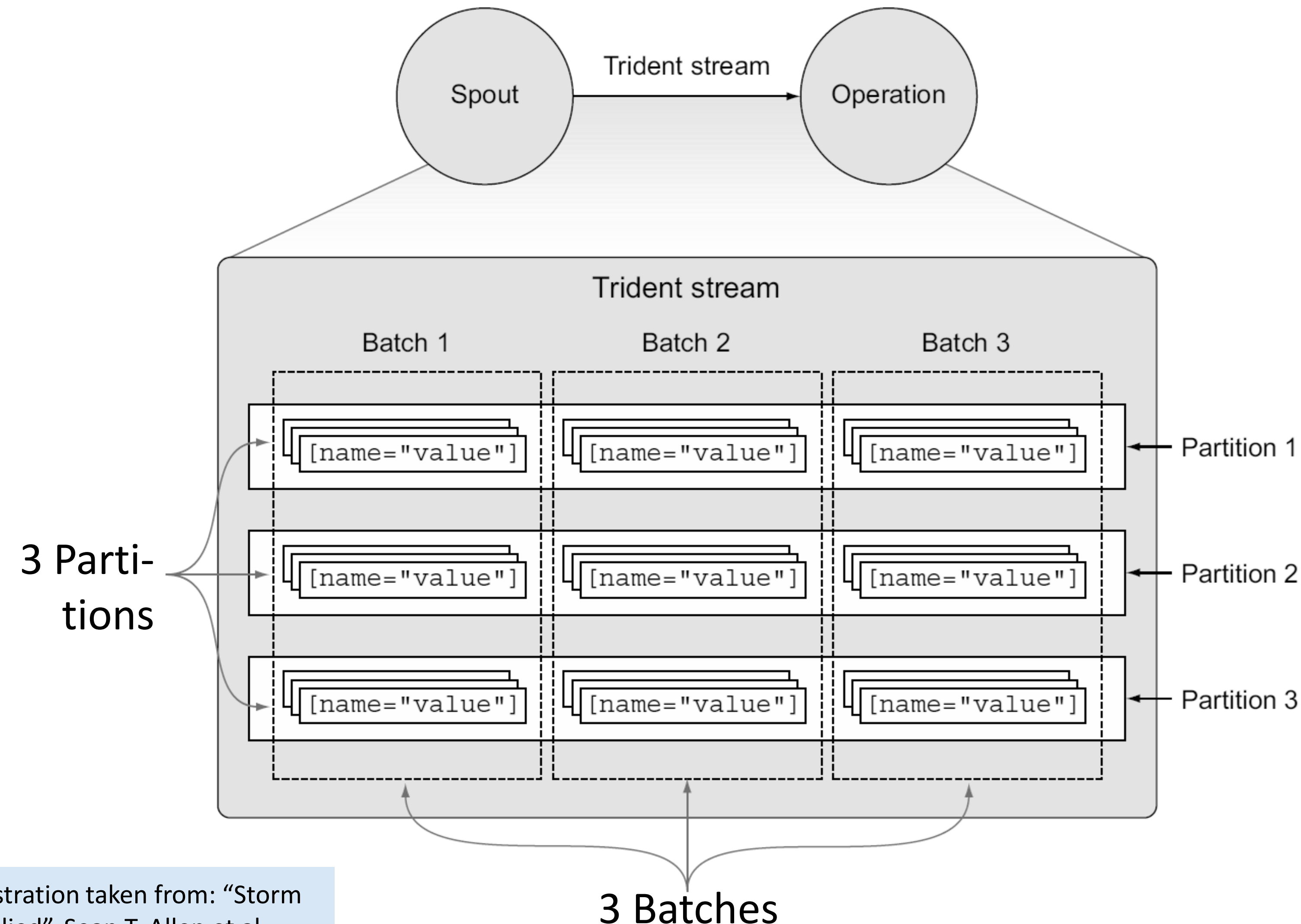
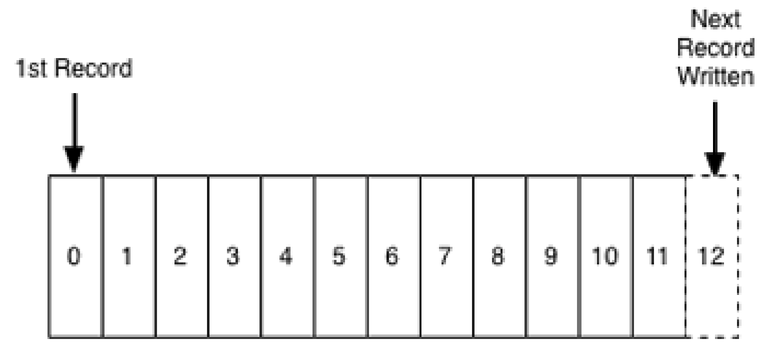


Illustration taken from: "Storm applied", Sean T. Allen et al.

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

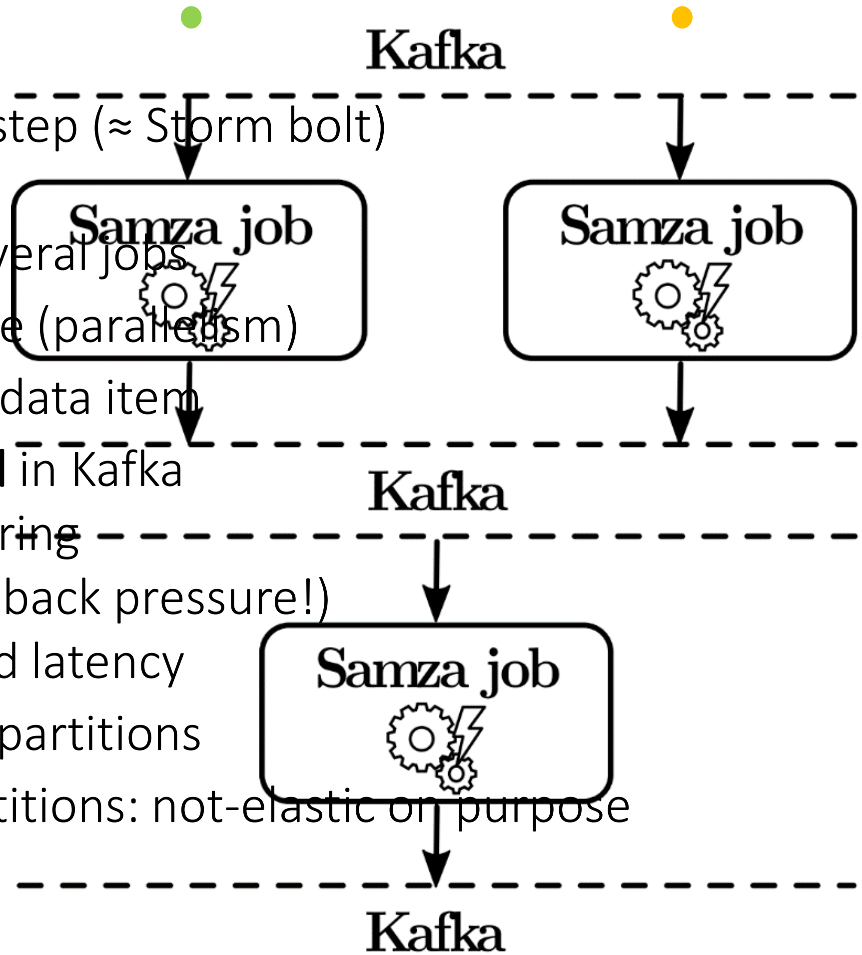
- Developed at **LinkedIn**
- **2013**: open-source (Apache Incubator)
- **2015**: Apache top-level project

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



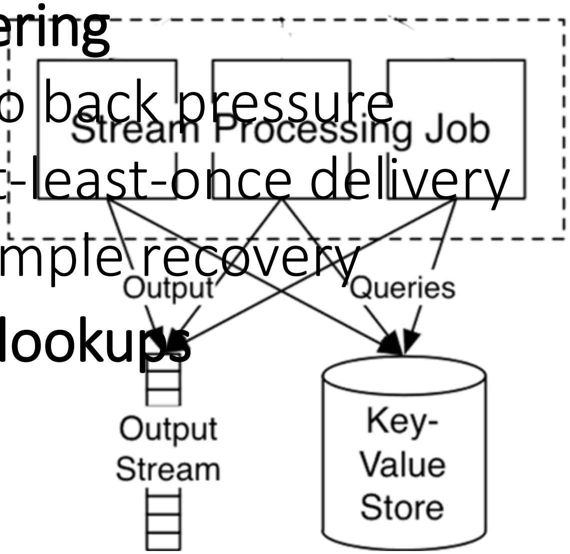
Martin Kleppmann, *Turning the database inside-out with Apache Samza* (2015)
https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/ (2017-02-23)

Samza

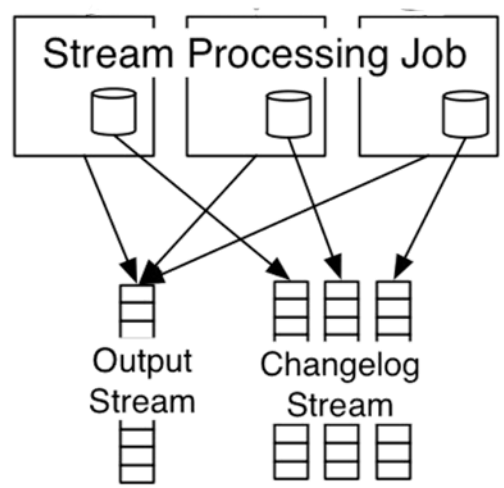
Local State

Advantages of local state:

- **Buffering**
  → No back pressure
  → At-least-once delivery
  → Simple recovery
- **Fast lookups**



Remote State



Local State

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)

State Management
Straightforward Recovery

samza

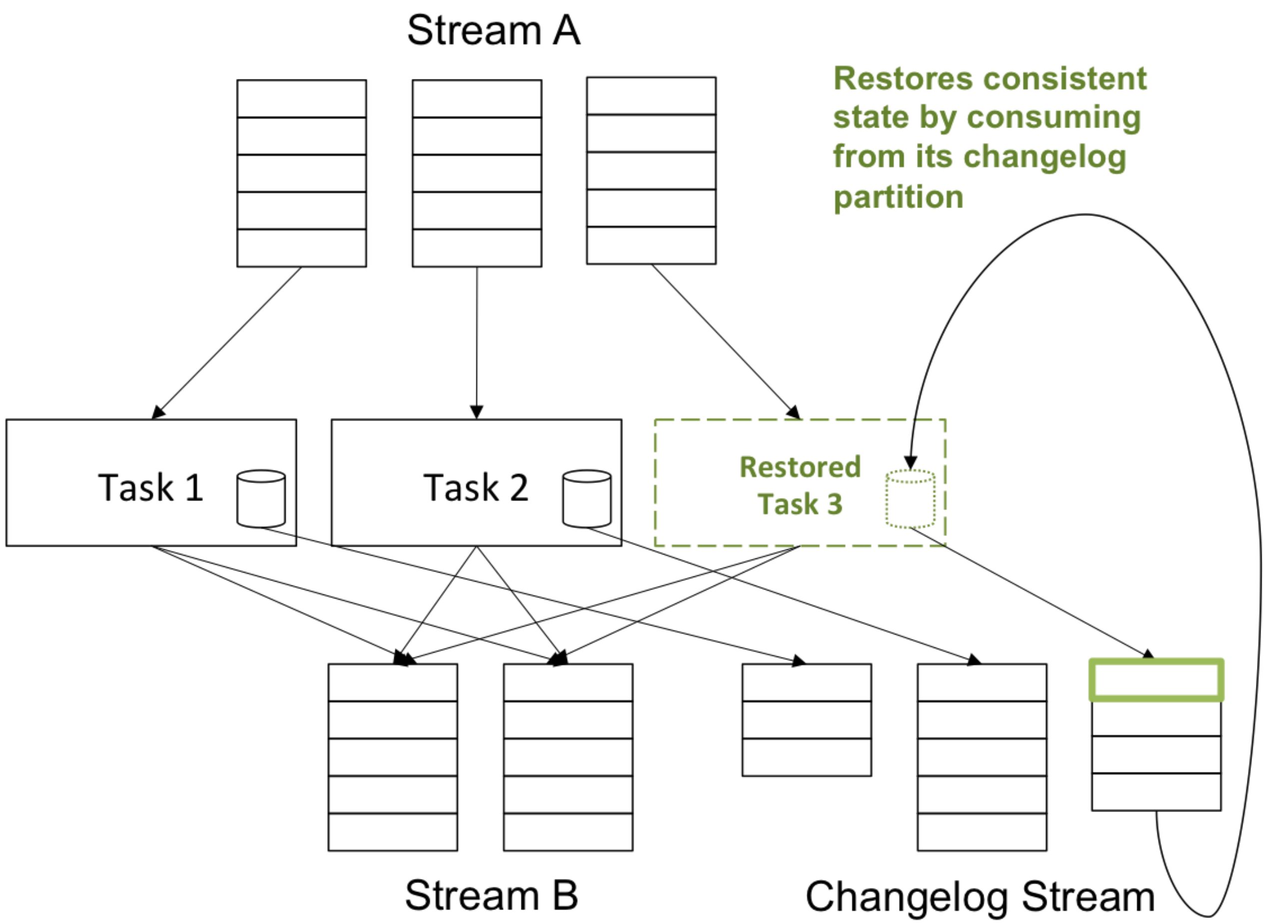


Illustration taken from: Navina Ramesh, *Apache Samza, LinkedIn's Framework for Stream Processing* (2015)
https://thenewstack.io/apache-samza-linkedins-framework-for-stream-processing (2017-02-26)

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

Spark Streaming

Resilient Distributed Data set (RDD)

- **Immutable** collection & **deterministic** operations
- **Lineage** tracking:
  → state can be reproduced
  → periodic checkpoints reduce recovery time

DStream: Discretized RDD

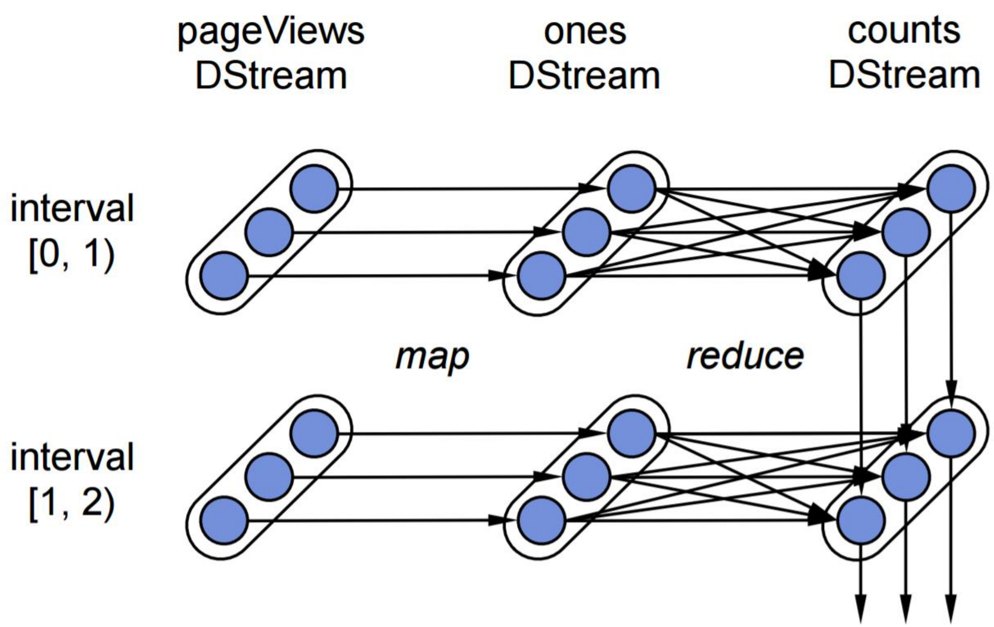
- **RDDs are processed in order**: no ordering within RDD
- RDD scheduling ~50 ms → latency >100ms

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

Example

Counting Page Views

```
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

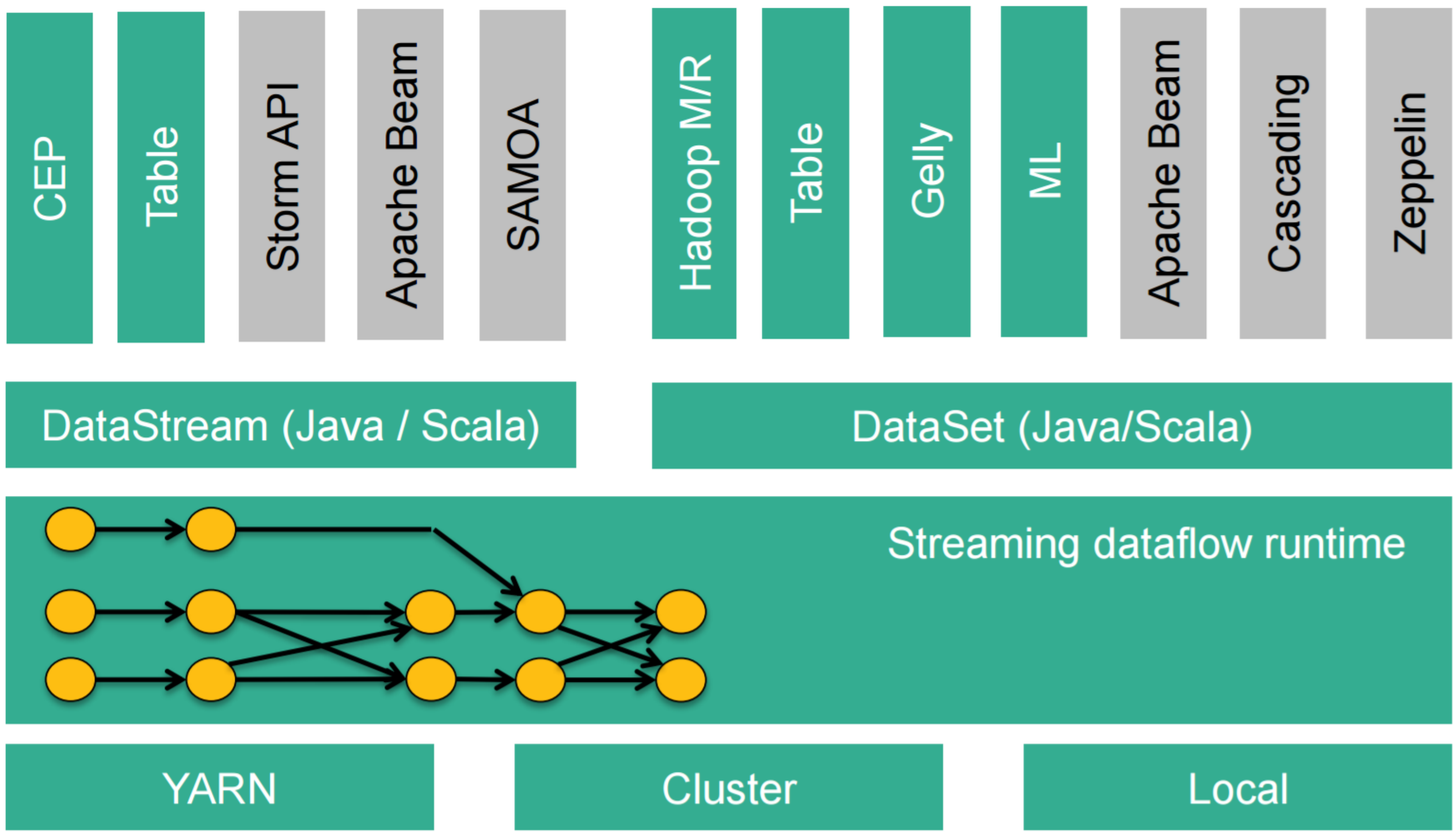
- **Native stream processor:** Latency <100ms feasible
- **Abstract API** for stream and batch processing, stateful, exactly-once delivery
- **Many libraries**: Table and SQL, CEP, Machine Learning , Gelly…
- **Users**: Alibaba, Ericsson, Otto Group, ResearchGate, Zalando…

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

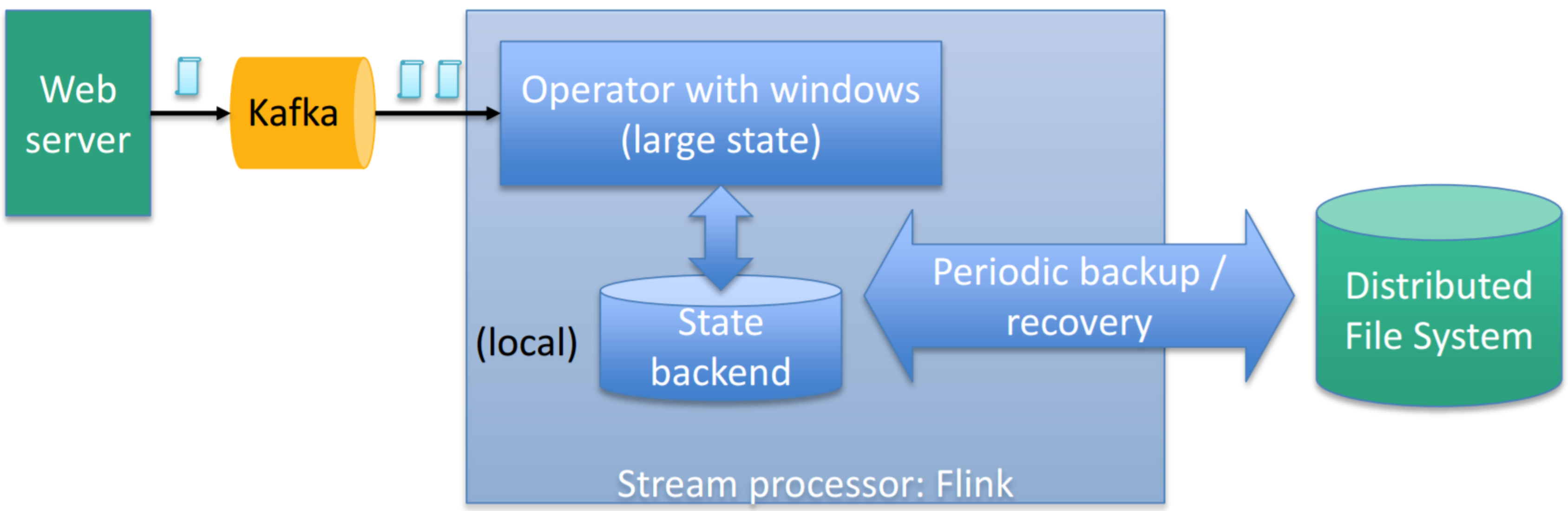


https://www.infoq.com/presentations/stream-processing-apache-flink

Managed State

Streaming + Batch

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

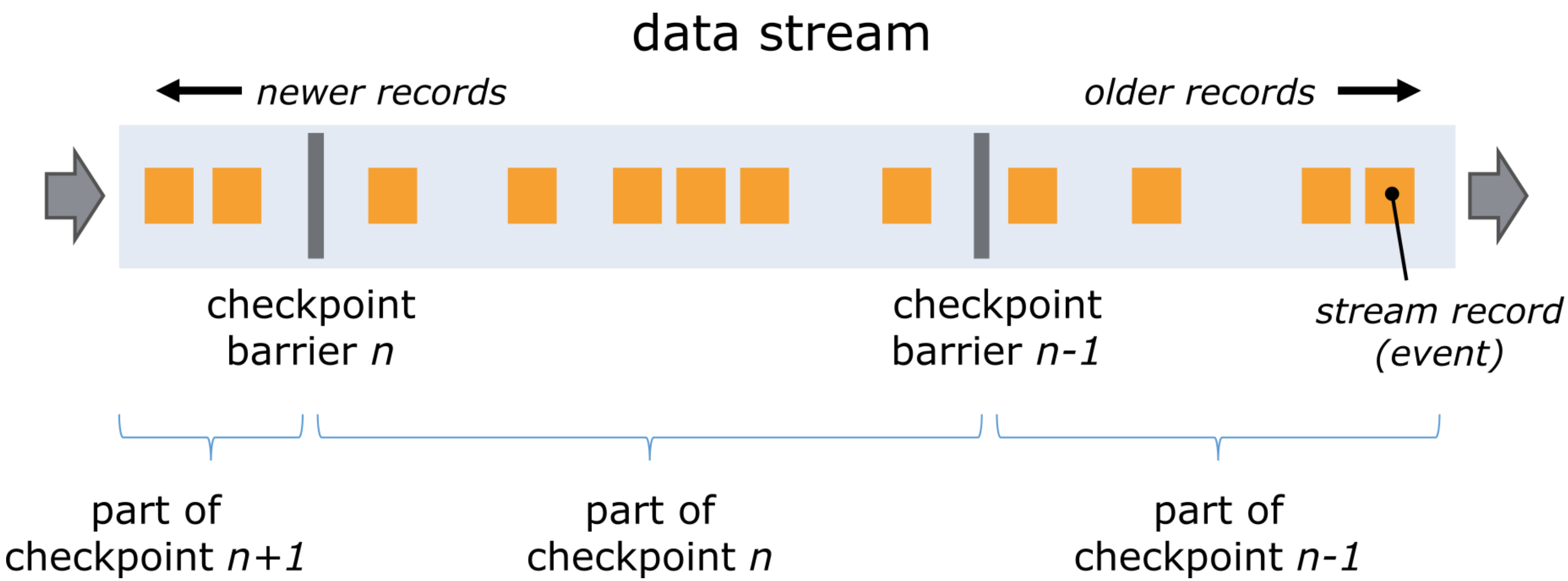


https://www.infoq.com/presentations/stream-processing-apache-flink

Highlight: Fault Tolerance

Distributed Snapshots

Flink



- **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

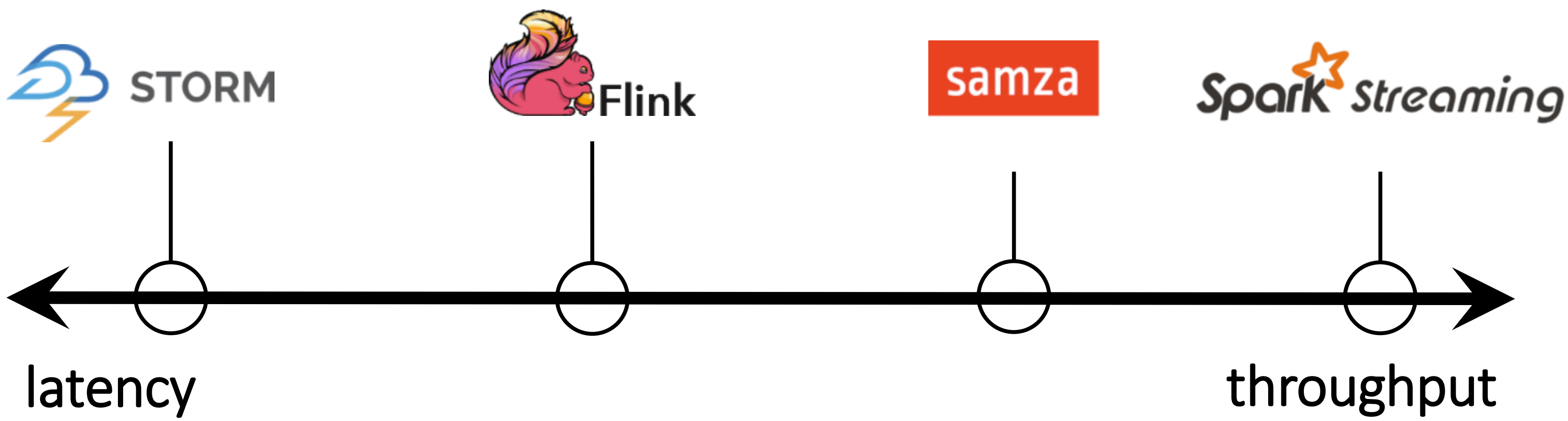
Kafka Streams

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

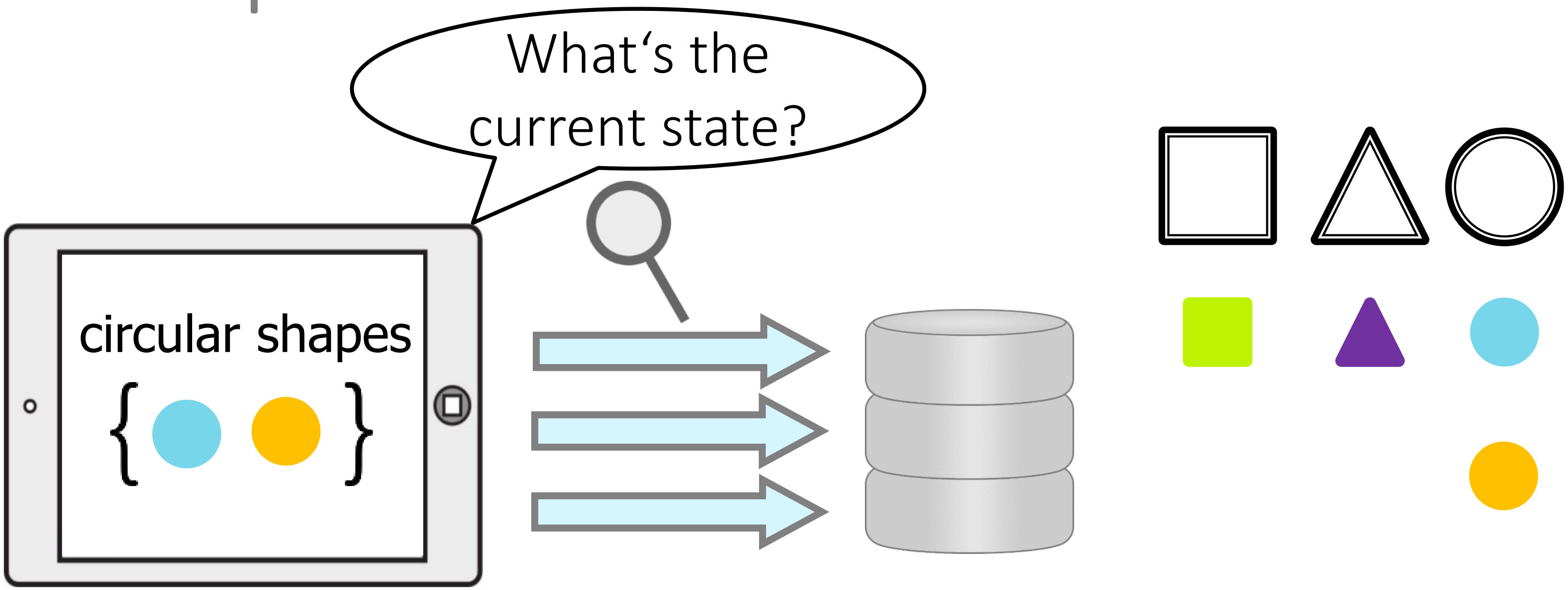
- **Big Picture:**
  - **Why** Push-Based Database Queries?
  - **Where** Do Real-Time Databases Fit in?
- **System Survey:**
  - Meteor
  - RethinkDB
  - Parse
  - Firebase
- **Discussion:**
  - Comparison Matrix
  - Other Systems

REAL-TIME DBS

Making Databases Push-Based

Traditional Database Access

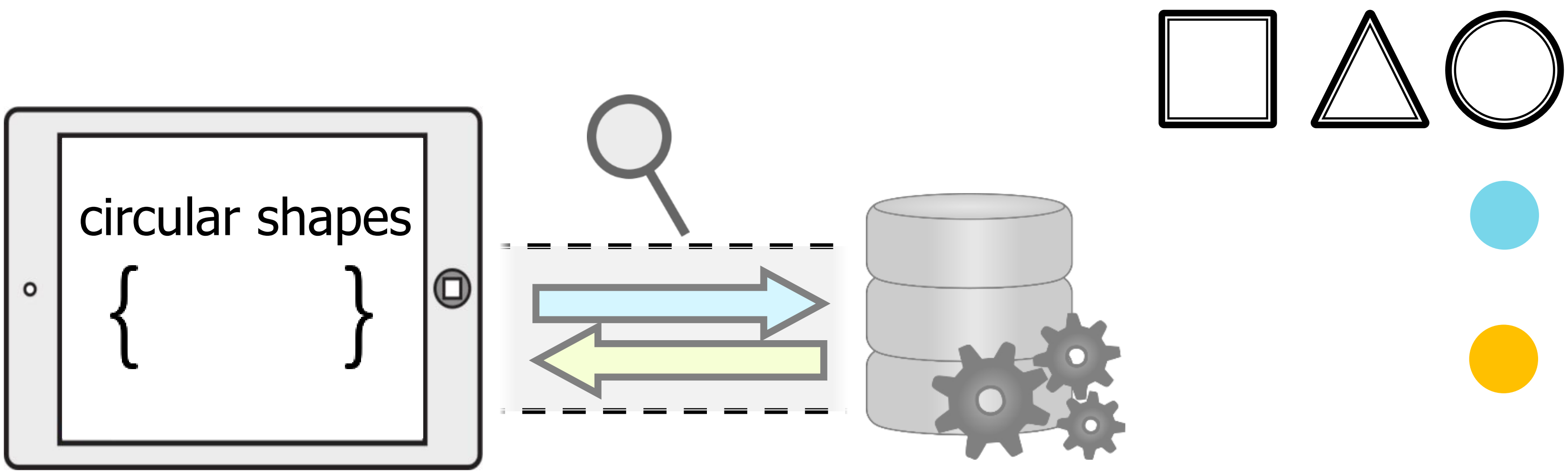
No Request? No Data!



Periodic Polling
→ inefficient
→ slow

Real-time Databases

Always In-Sync With Database State



Real-Time Queries for query result maintenance:

→ efficient

→ fast

Quick Comparison

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

Database Management

Real-Time Databases

Data Stream Management

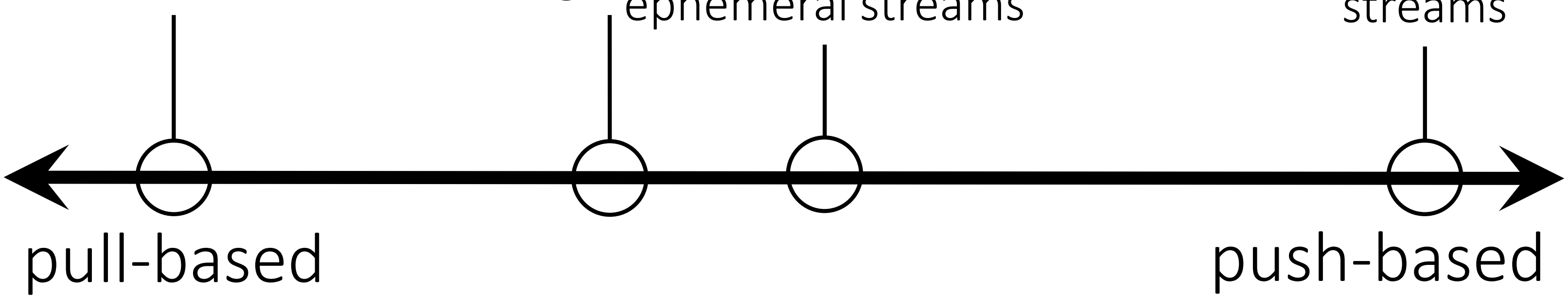
Stream Processing

static collections

evolving collections

persistent/ ephemeral streams

ephemeral streams



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)

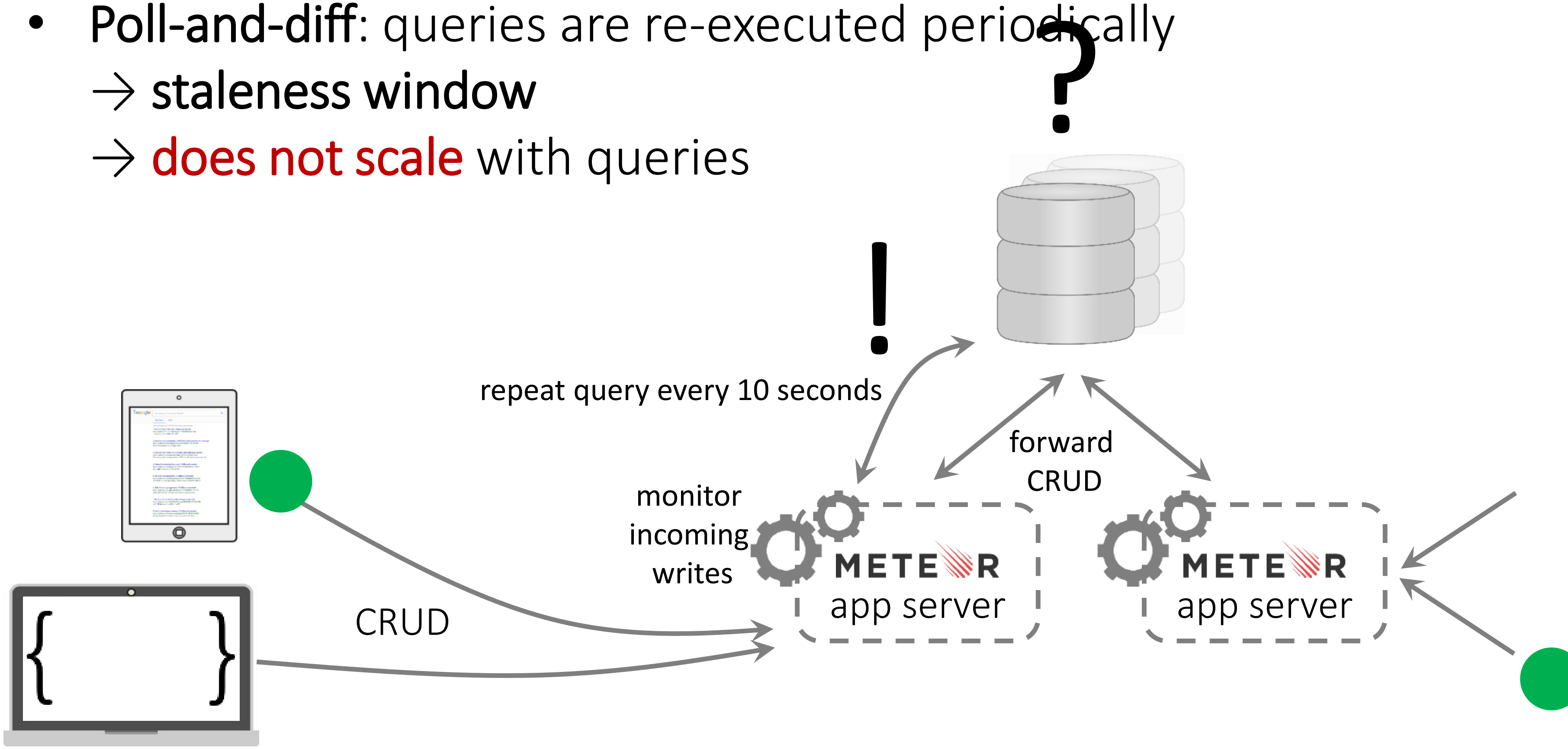
History:

- 2011: *Skybreak* is announced
- 2012: Skybreak is renamed to Meteor
- 2015: Managed hosting service Galaxy is announced

Live Queries

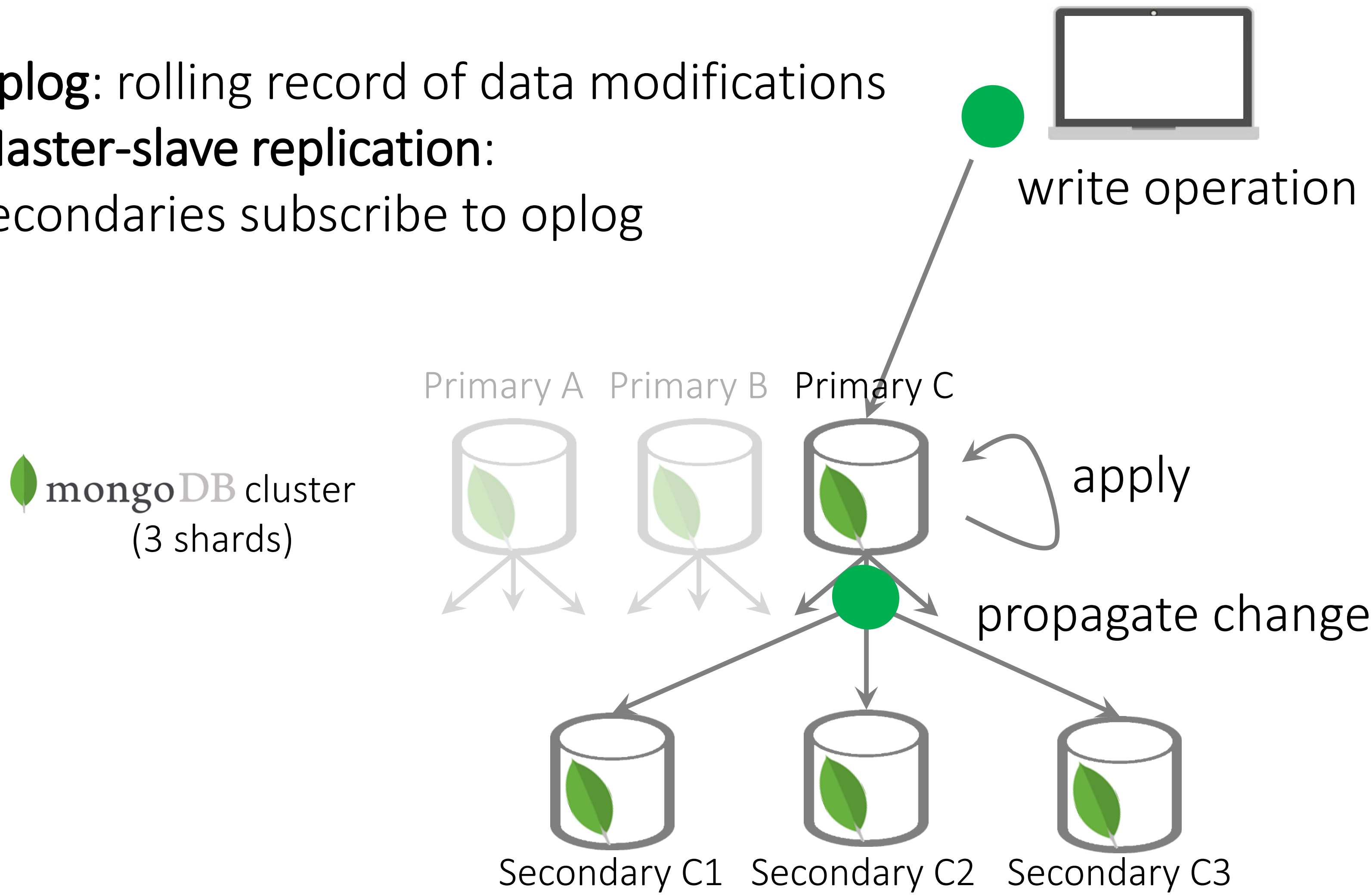
Poll-and-Diff

- **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

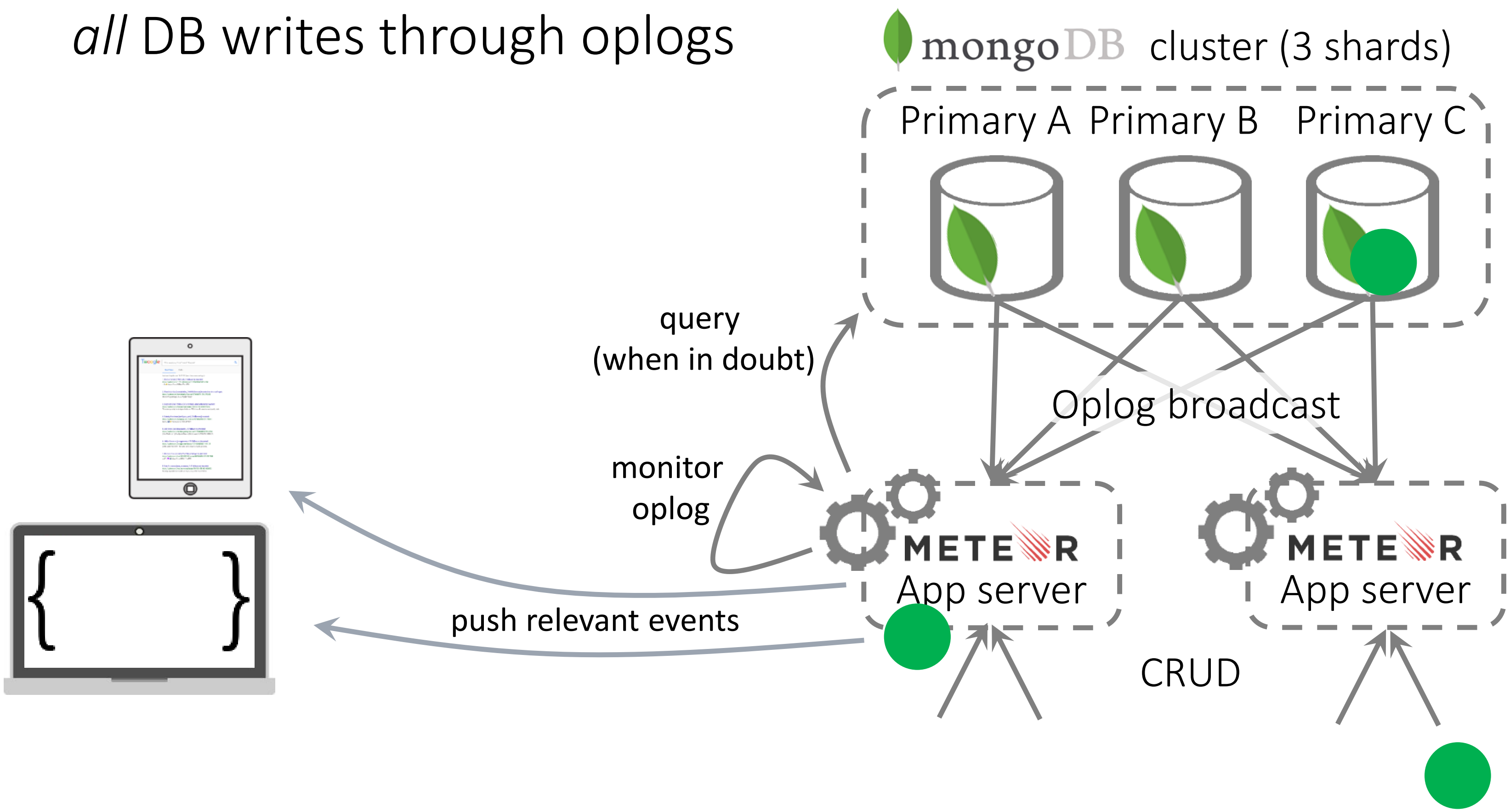
- **Oplog**: rolling record of data modifications
- **Master-slave replication**:
  Secondaries subscribe to oplog



Oplog Tailing

Tapping into the Oplog

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



Oplog Tailing

Oplog Info is Incomplete

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

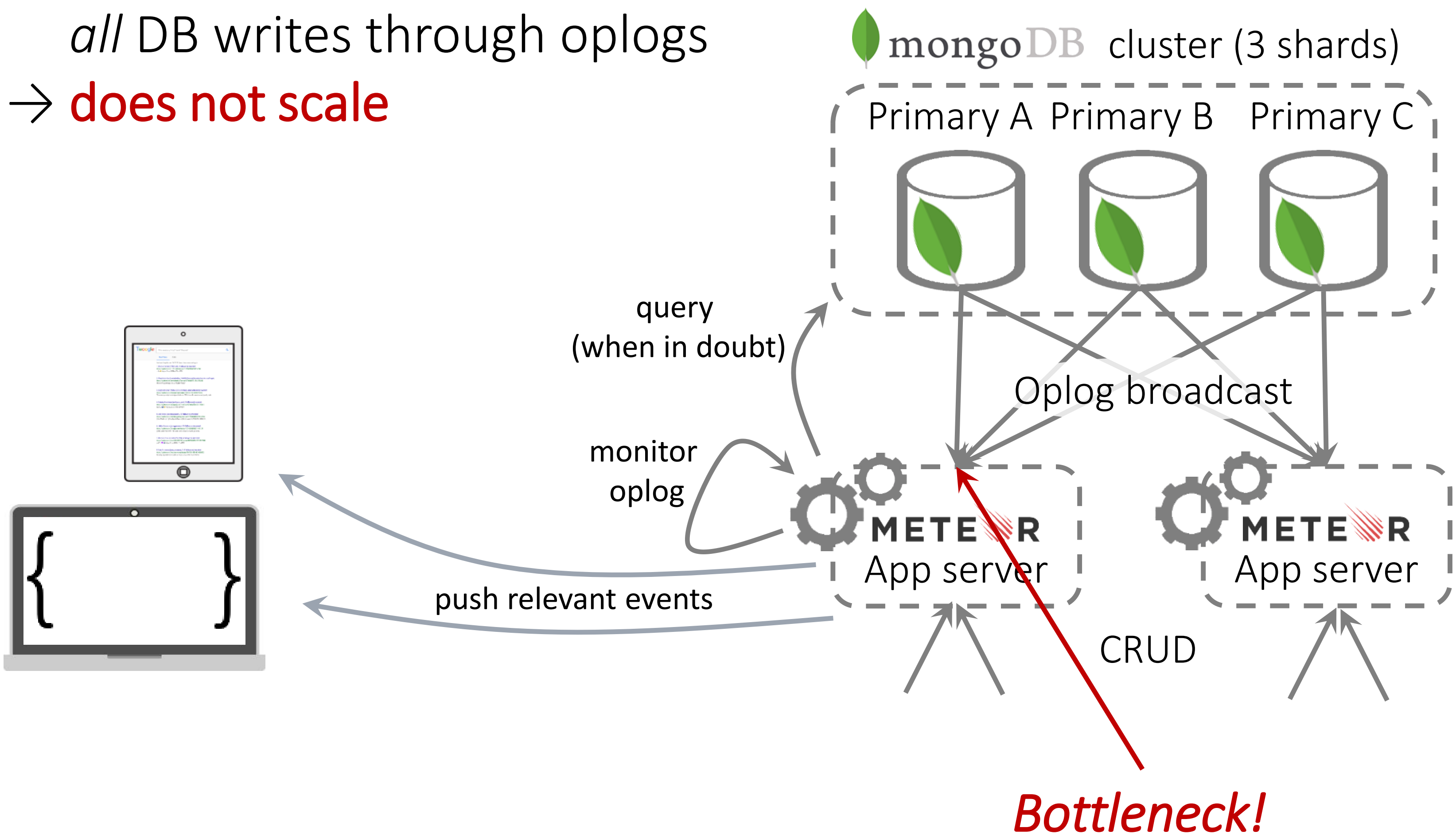
```
1. { name: „Joy", game: „baccarat", score: 100 }
2. { name: „Tim", game: „baccarat", score: 90 }
3. { name: „Lee", game: „baccarat", score: 80 }
```

Oplog Tailing

Tapping into the Oplog

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

→ does not scale



RethinkDB

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

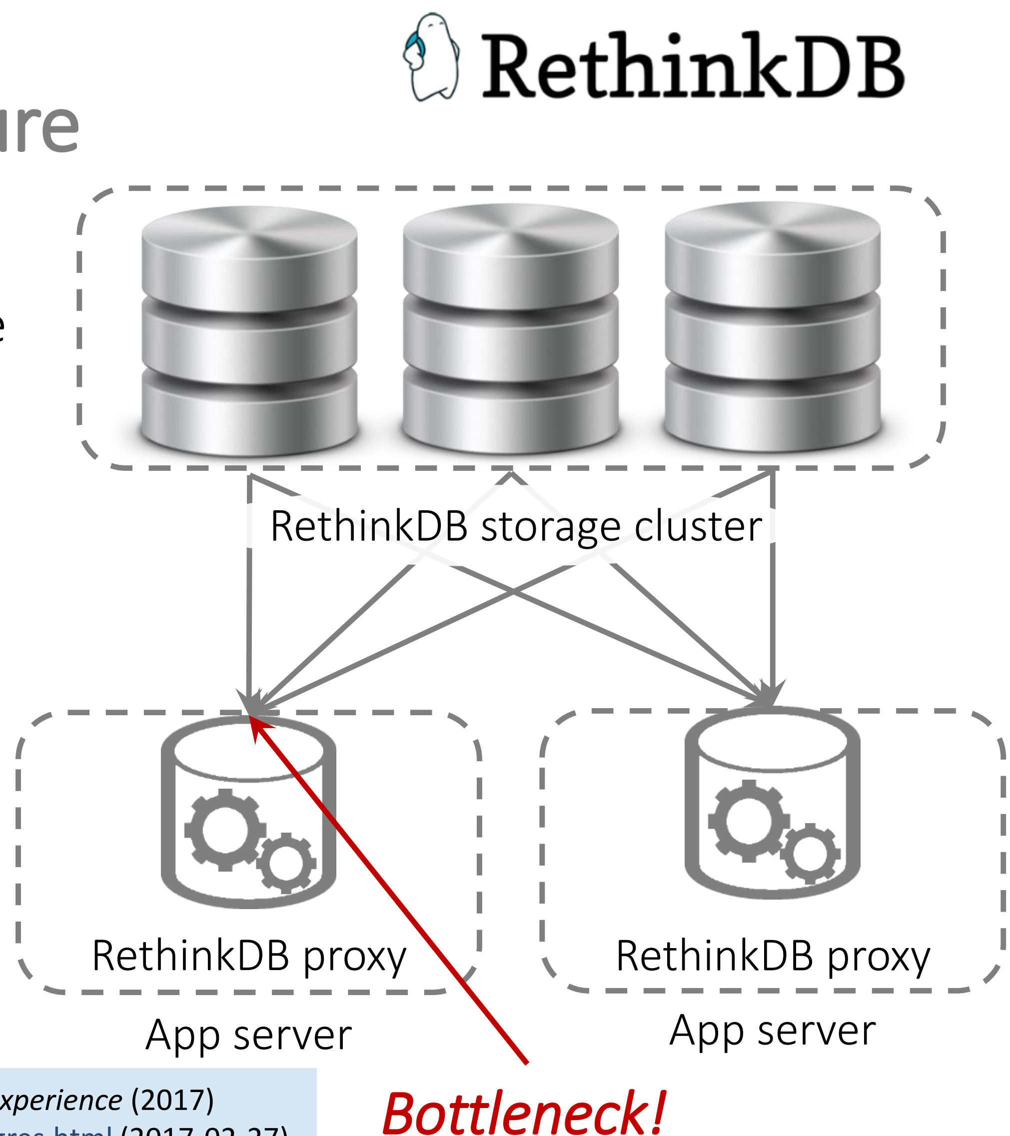
History:

- 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

Parse

Overview:

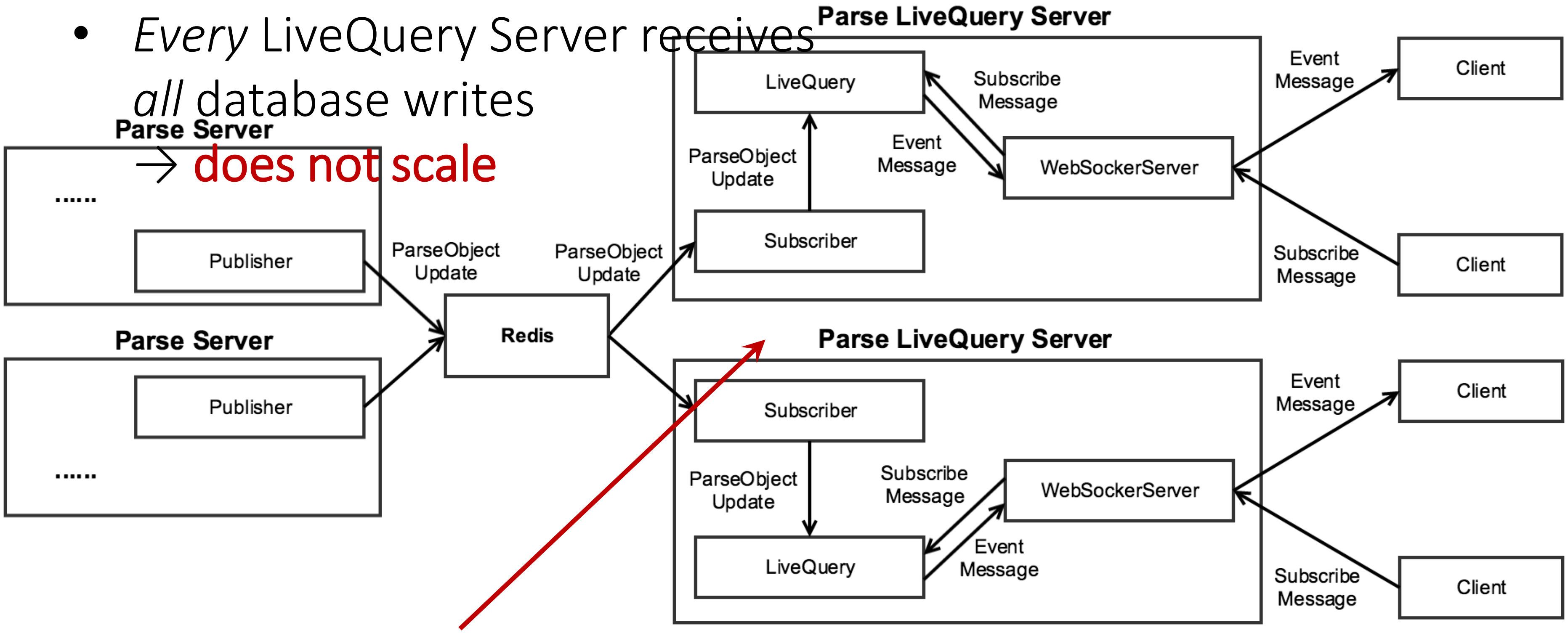
- **Backend-as-a-Service** for mobile apps
  - **MongoDB:** largest deployment world-wide
  - **Easy development**: great docs, push notifications, authentication, …
  - **Real-time** updates for most MongoDB queries
- **Open-source**: BSD license
- **Managed service**: discontinued

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

- **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

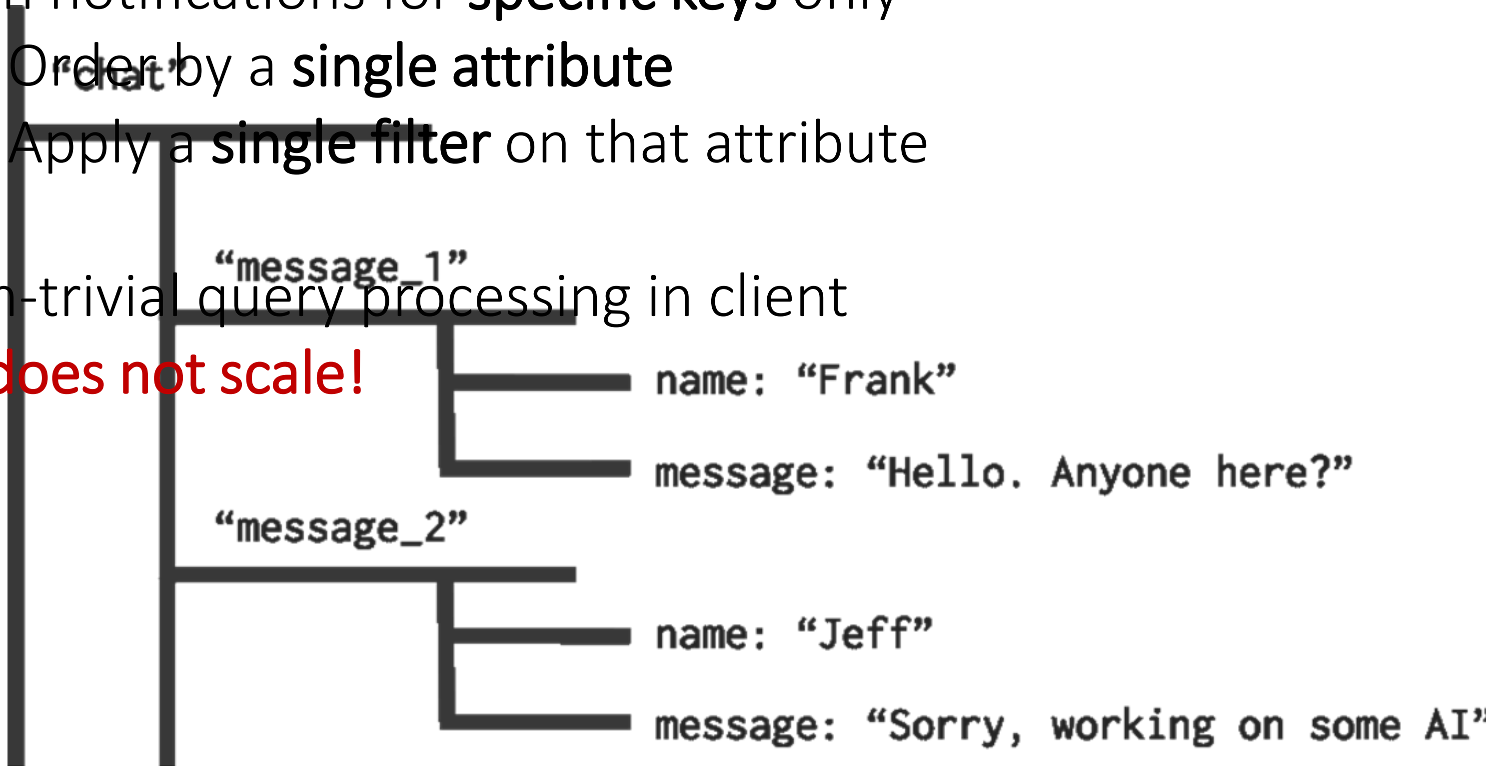
- **Tree data model**: application state ~JSON object
- **Subtree synching**: push notifications for specific keys only
  → Flat structure for fine granularity

→ 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."

Bottleneck!

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

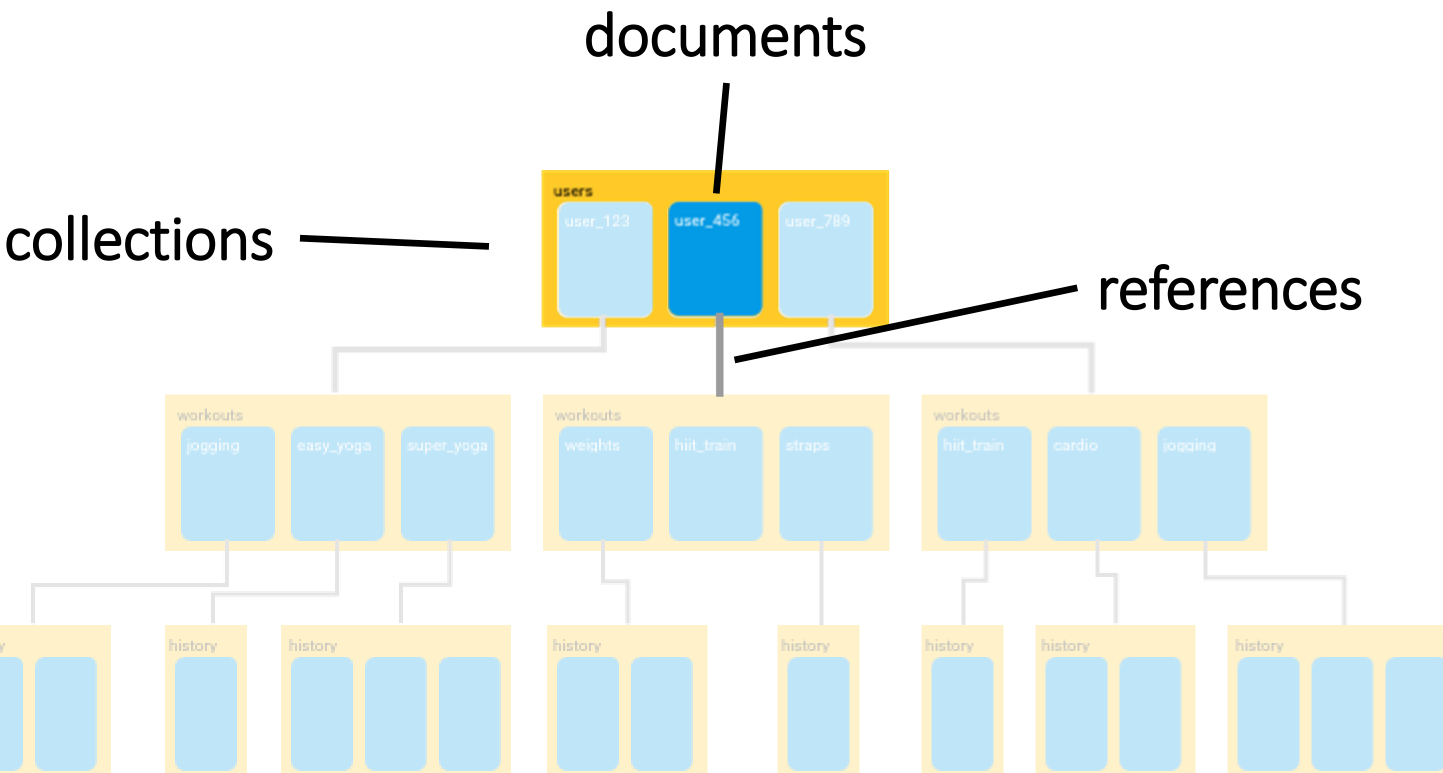


Illustration taken from: Todd Kerpelman, *Cloud Firestore for Realtime Database Developers (2017)*
https://firebase.googleblog.com/2017/10/cloud-firestore-for-rtdb-developers.html (2018-03-10)

Firebase

Firestore: New Model

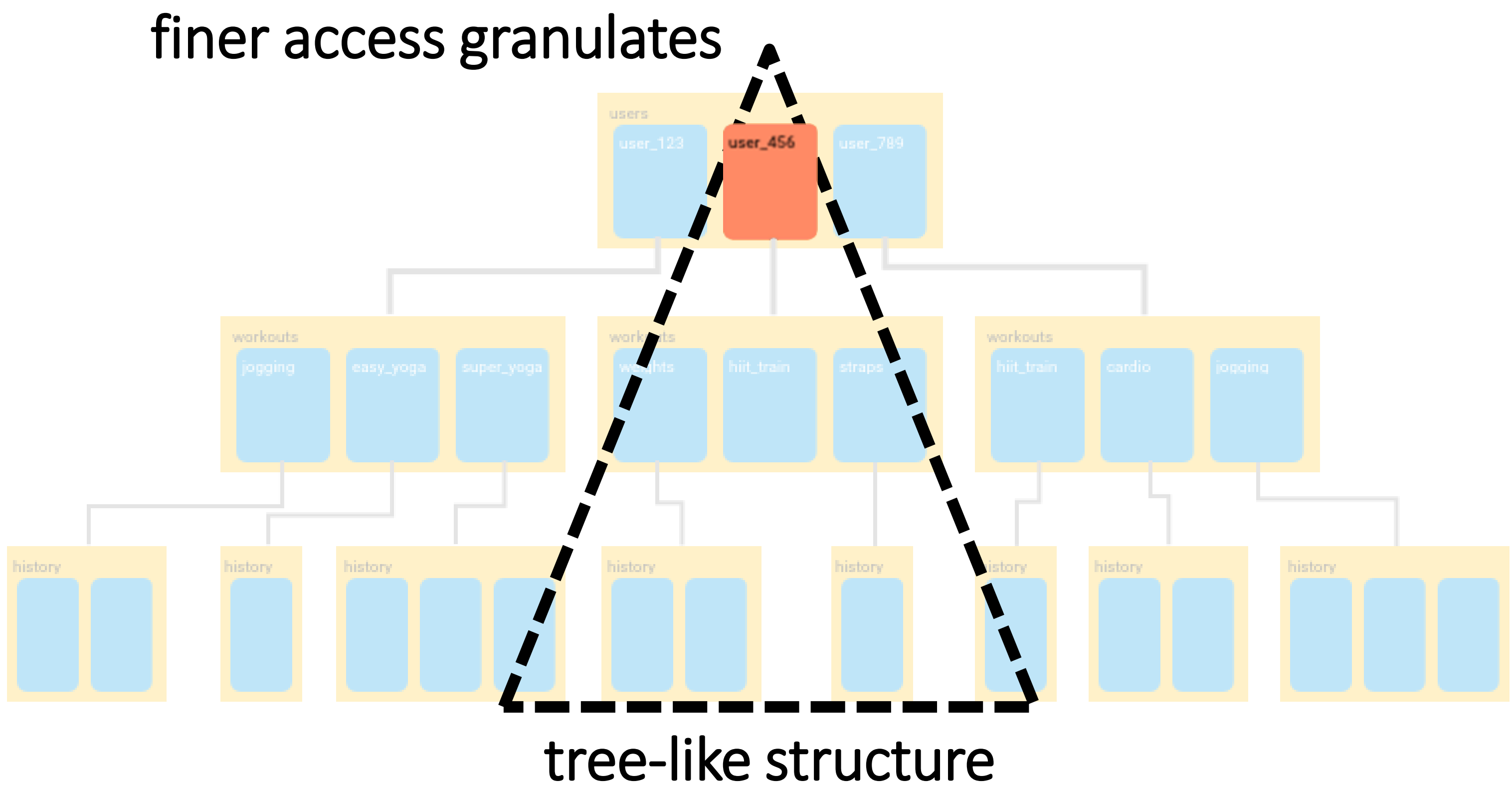


Illustration taken from: Todd Kerpelman, *Cloud Firestore for Realtime Database Developers (2017)*
https://firebase.googleblog.com/2017/10/cloud-firestore-for-rtdb-developers.html (2018-03-10)

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

	METEOR		RethinkDB	Parse	Firebase
	Poll-and-Diff	Change Log Tailing			Unknown
Write Scalability	✓	✗	✗	✗	✗
Read Scalability	✗	✓	✓	✓	? (100k connections)
Composite Filters (AND/OR)	✓	✓	✓	✓	○ (AND In Firestore)
Sorted Queries	✓	✓	✓	✗	○ (single attribute)
Limit	✓	✓	✓	✗	✓
Offset	✓	✓	✗	✗	○ (value-based)
Self-Maintaining Queries	✓	✓	✗	✗	✗
Event Stream Queries	✓	✓	✓	✓	✓

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

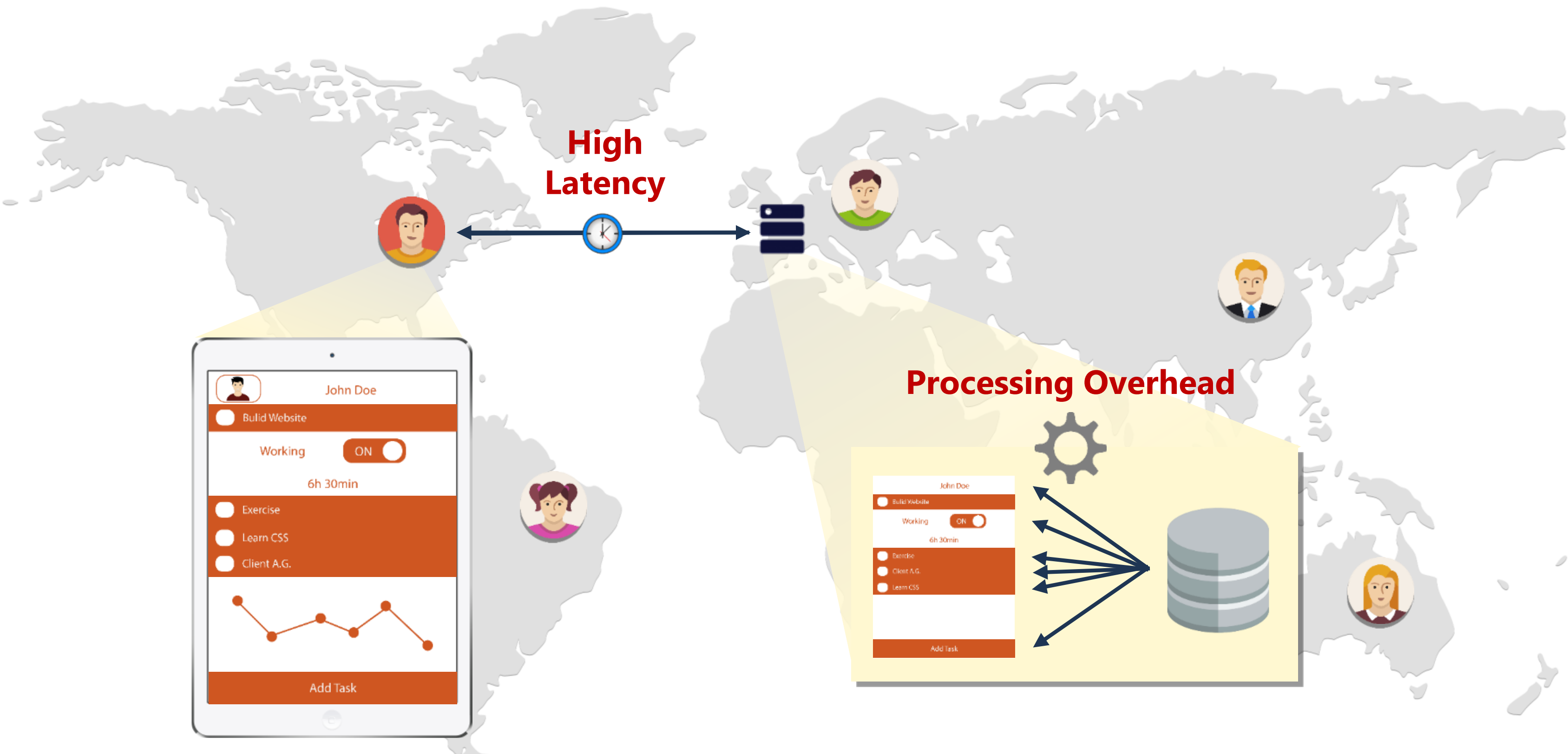
- **Caching Dynamic Data:**
  - Why is the Web Slow?
  - Caching to the Rescue!
  - Query Caching
- **Real-Time Queries:**
  - Scalability
  - Expressiveness
  - Legacy Compatibility
  - Use Cases
- **Open Challenges:**
  - TTLs & Transactions
  - Polyglot Persistence
- **Summary**

OUTLOOK

Our Research at the University of Hamburg

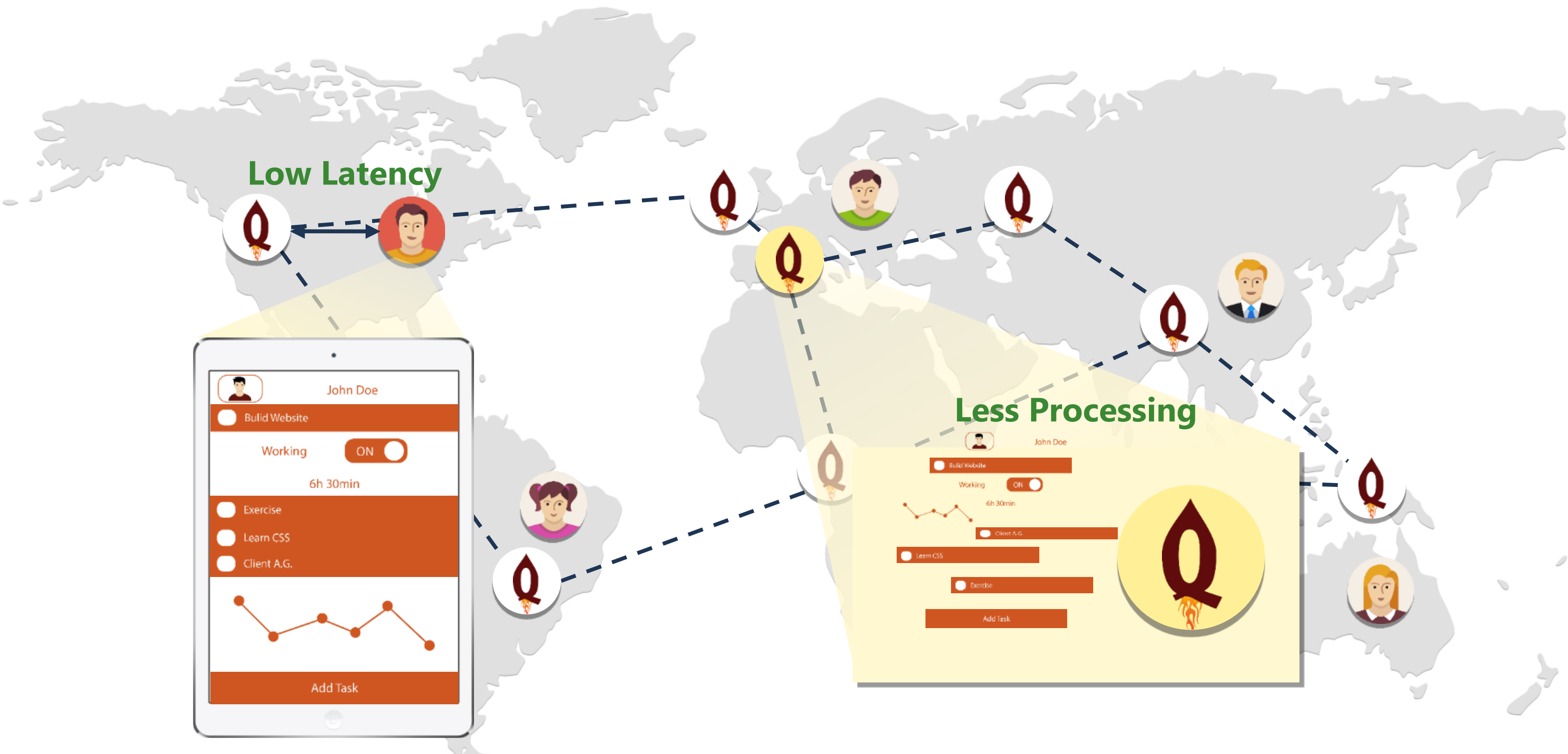
Problem: Slow Websites

Two Bottlenecks: Latency and Processing



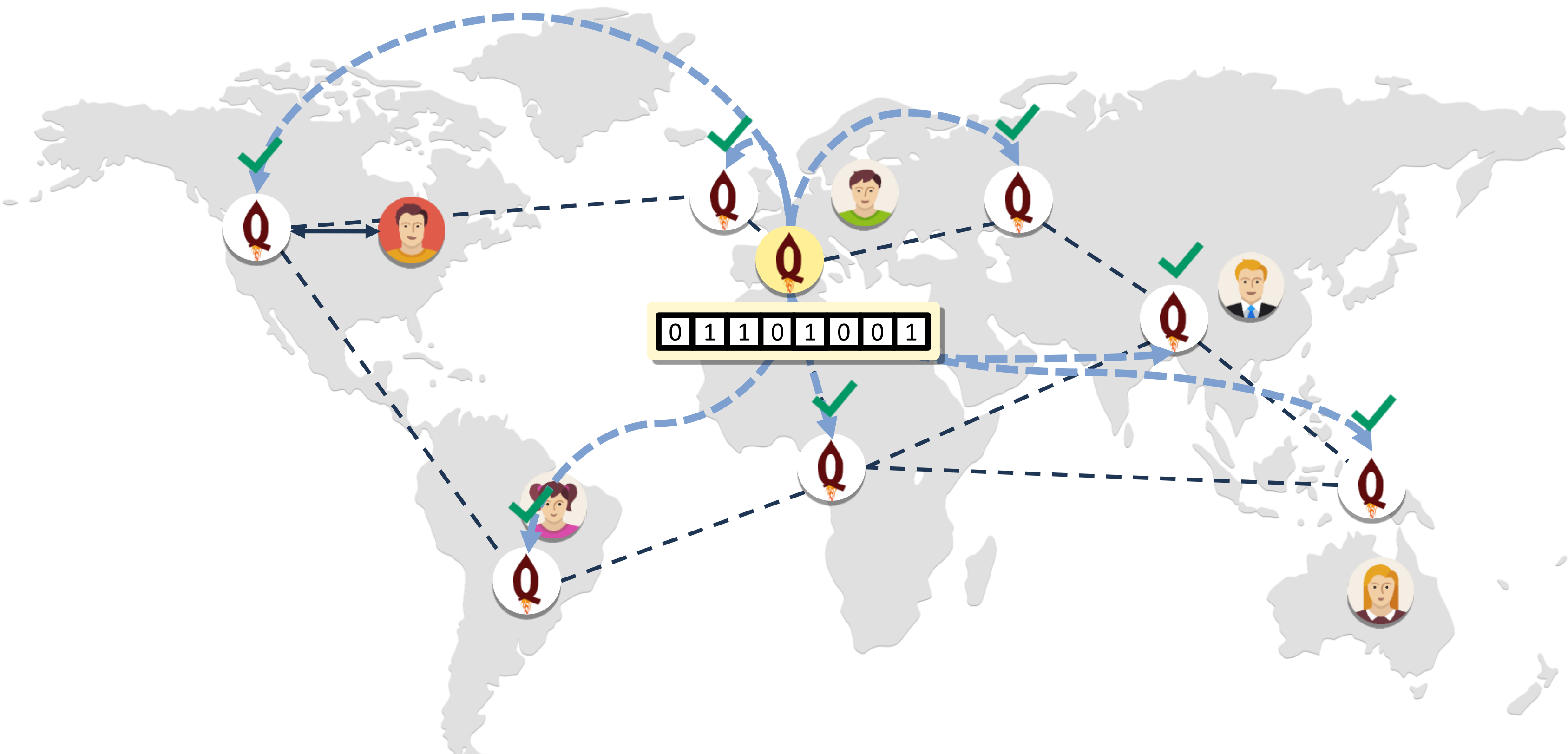
Solution: Global Caching

Fresh Data From Distributed Web Caches



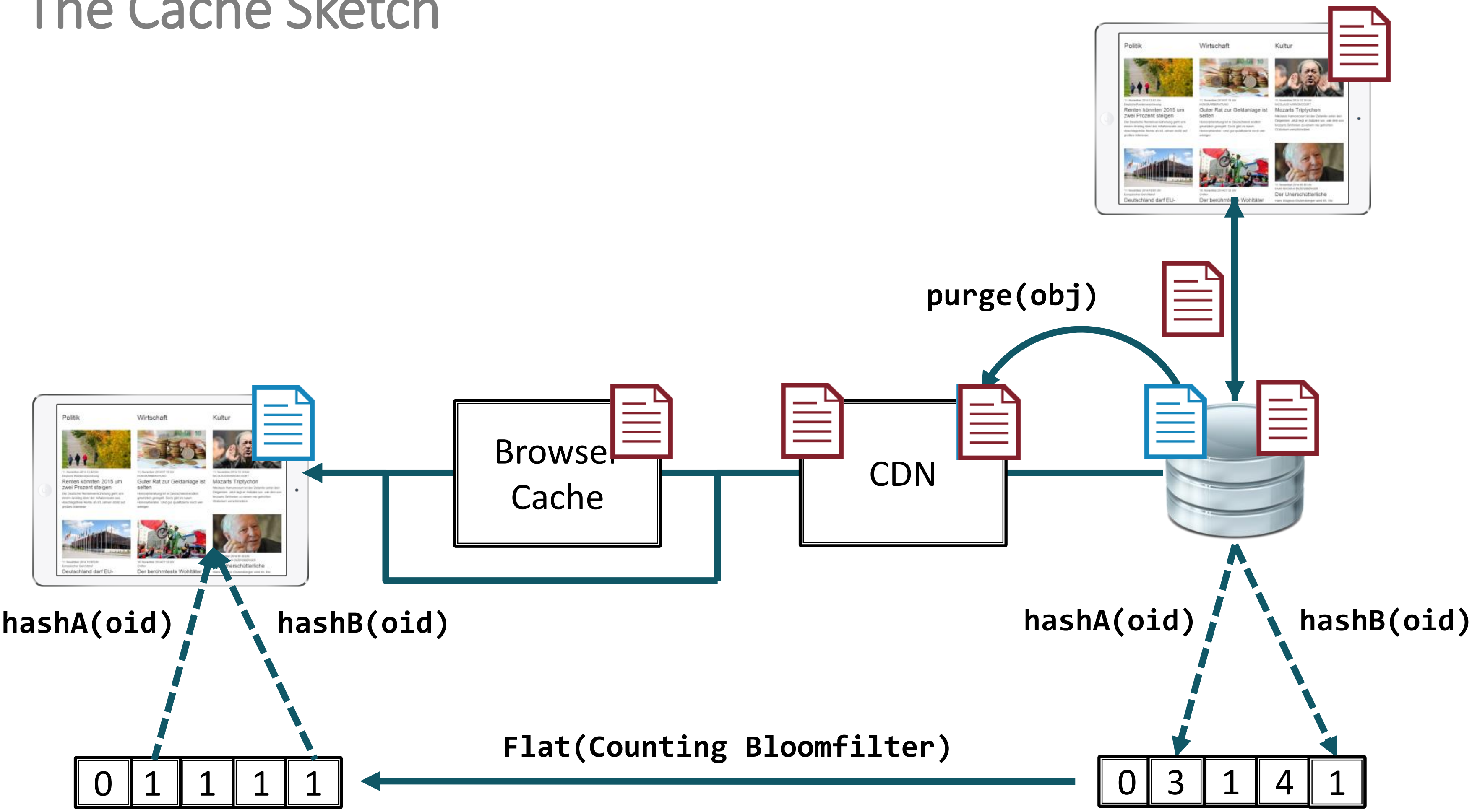
New Caching Algorithms

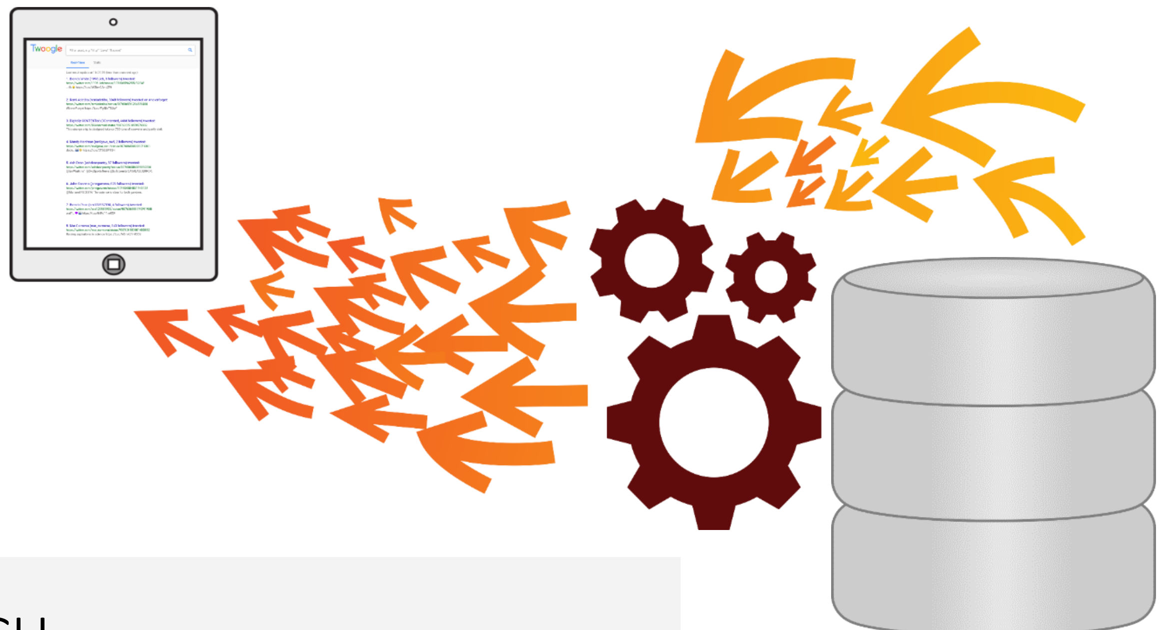
Solve Consistency Problem



Consistent Web Caching

The Cache Sketch





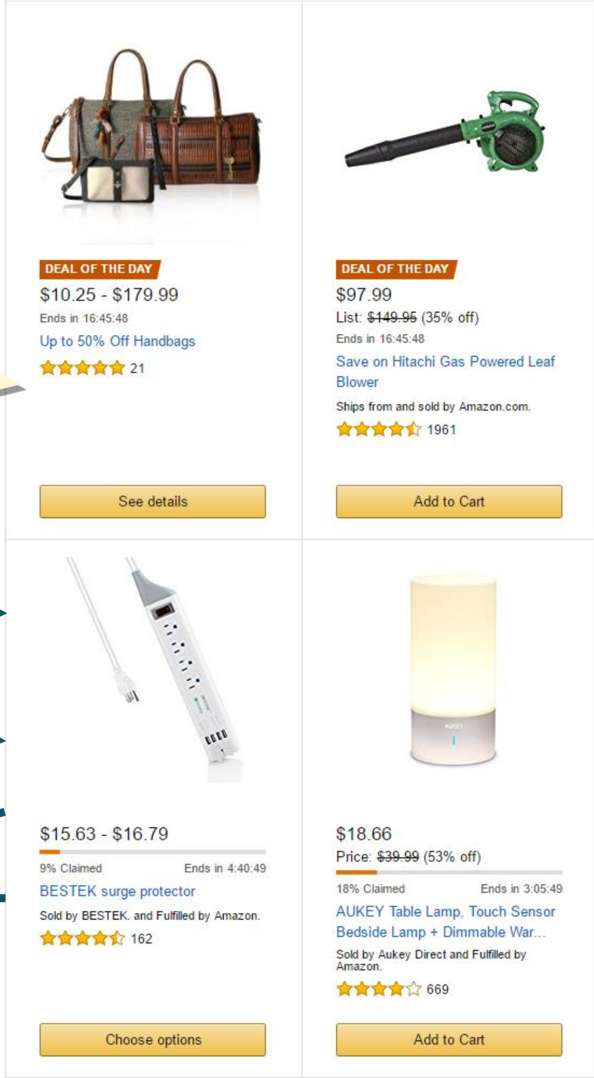
RESEARCH

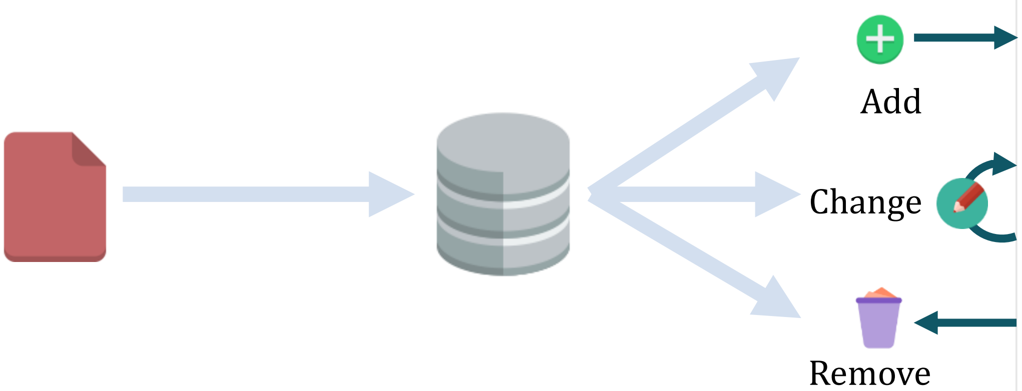
How to Invalidate DB Query Results?

InvaliDB

Invalidating DB Queries

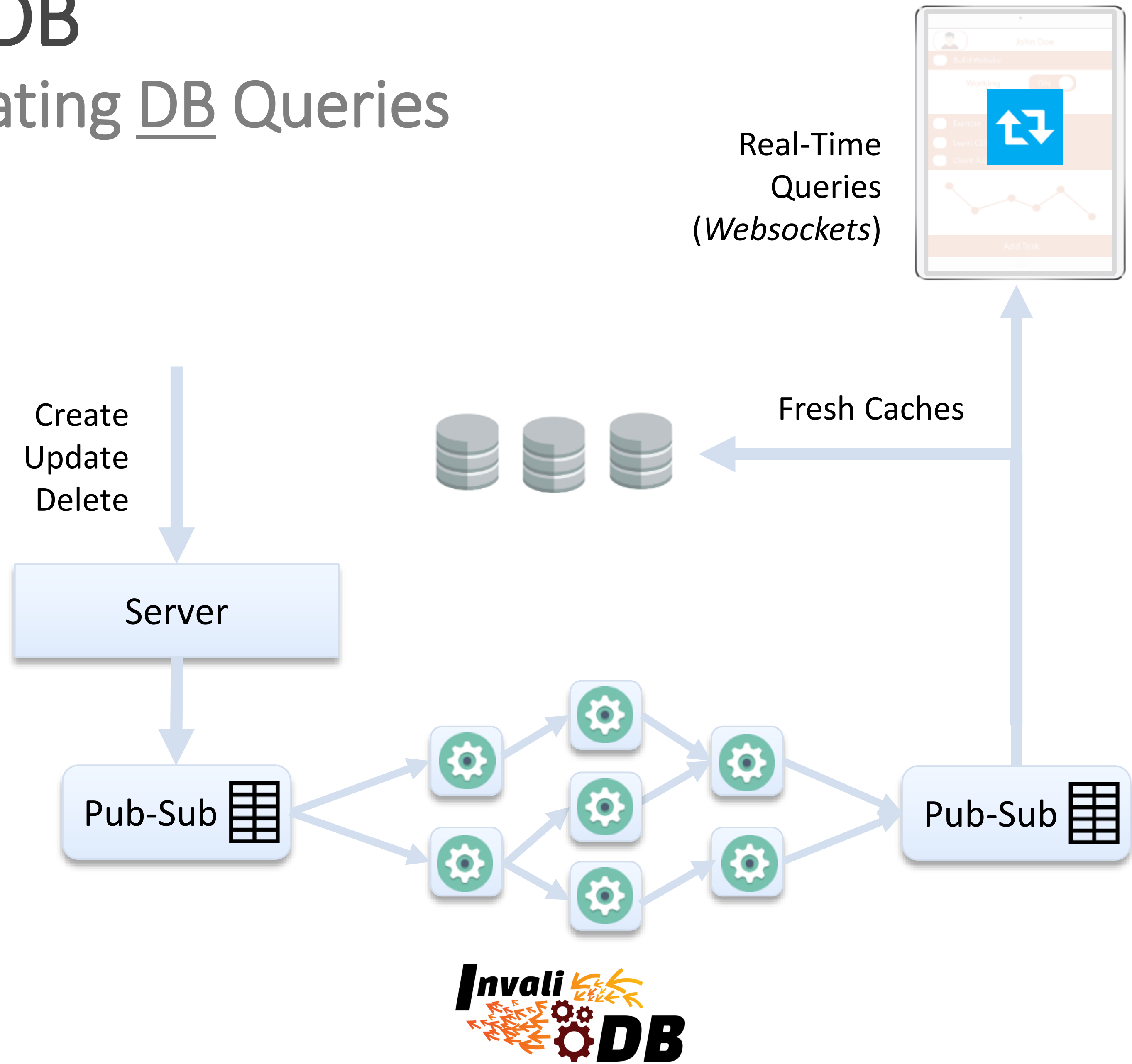
How to **detect changes to query results**:
„Give me the most popular products that are in stock.“





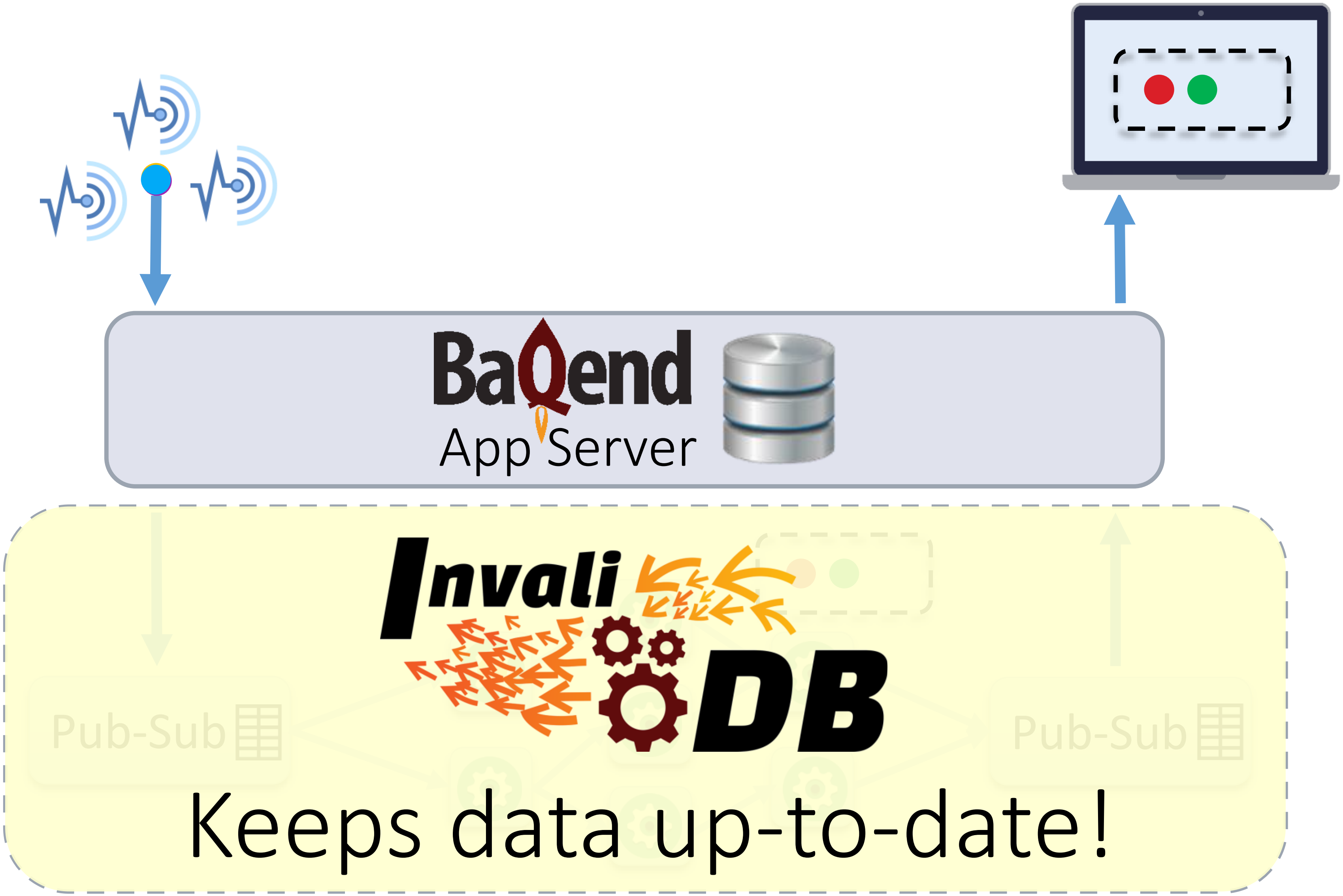
InvaliDB

Invalidating DB Queries



Baqend Real-Time Queries

Realtime Decoupled

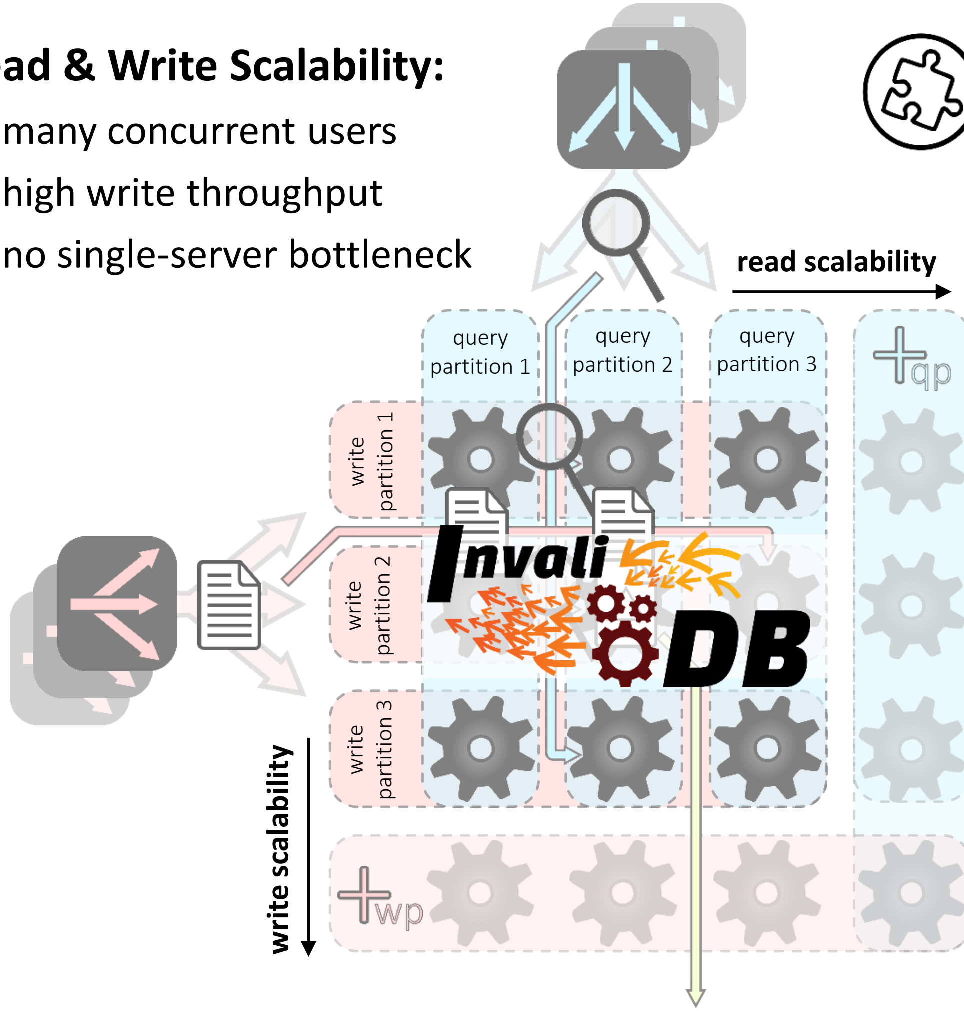


InvaliDB: A Scalable Real-Time Database Design

Two-Dimensional Workload Partitioning

Read & Write Scalability:

- many concurrent users
- high write throughput
- no single-server bottleneck



Pluggable Query Engine:

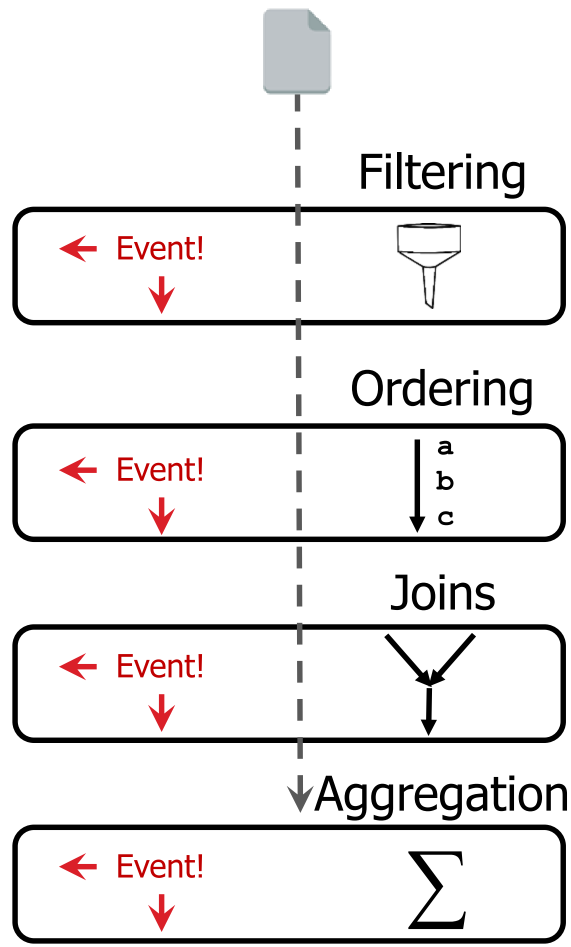
- legacy-compatibility
- multi-tenancy across databases

Baqend Real-Time Queries

Staged Real-Time Query Processing

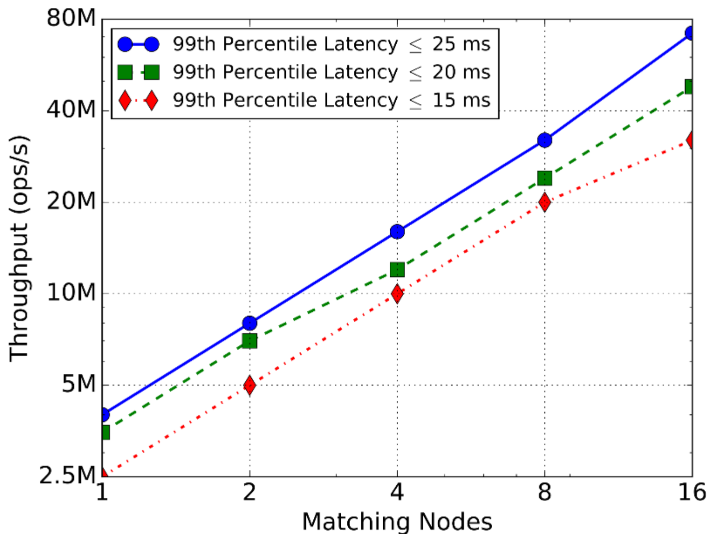
Change notifications go through different query processing stages:

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

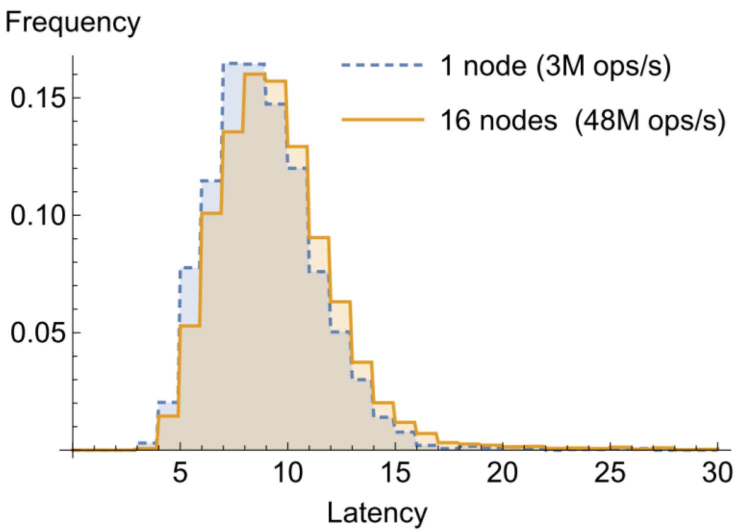


Baqend Real-Time Queries
Low Latency + Linear Scalability

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

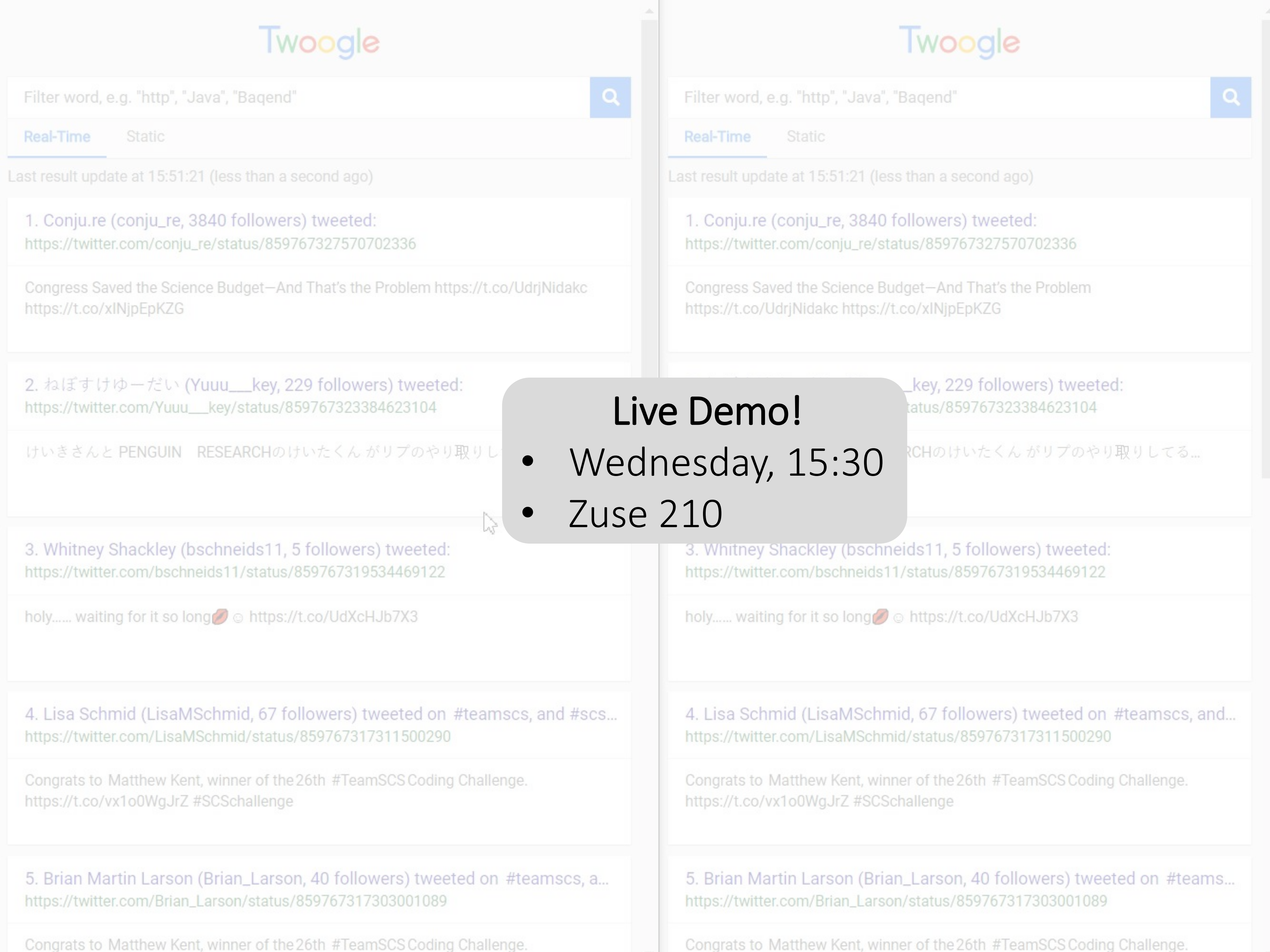
```
query.resultList(result => ...);
```

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
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.
Live Demo!
Wednesday, 15:30
Zuse 210

Baqend

Try It Out!

Platform

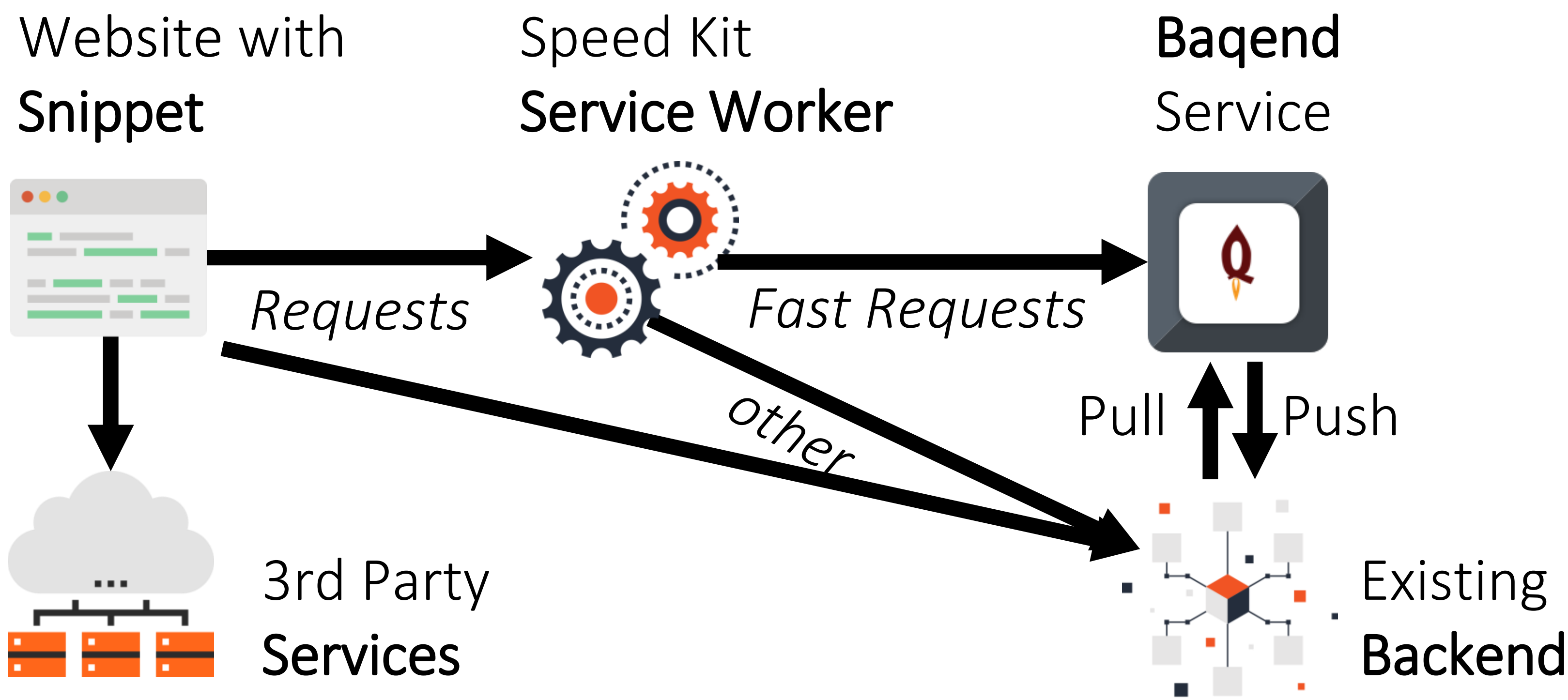
- Platform for building (Progressive) **Web Apps**
- **15x** Performance Edge
- Faster **Development**

Speed Kit

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

Speed Kit

Baqend Caching for Legacy Websites



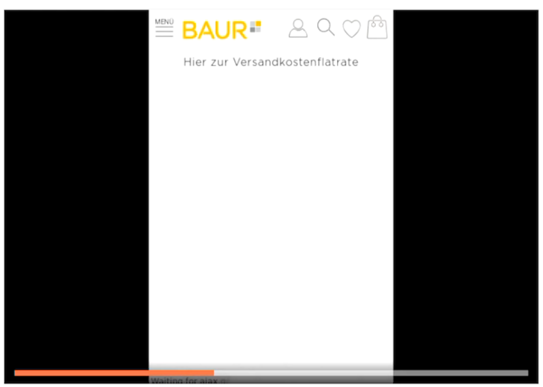
Speed Kit
Measure Your Page Speed!

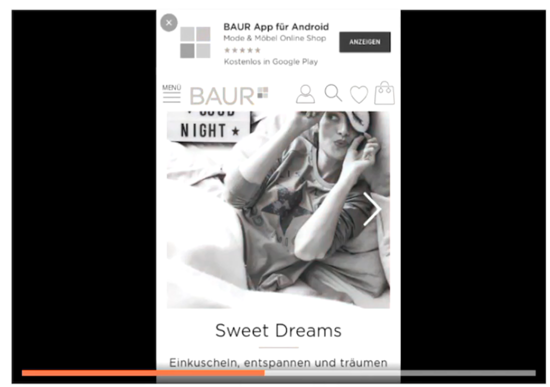
https://test.speed-kit.com/

https://www.baur.de/

You are using Speed Kit 1.12.1

Without Speed Kit		Your Website (Speed Kit 1.12.1)
1283 ms	1.9x Faster	662 ms





Without Speed Kit	Your Website
1.3s	0.7s

Average — Fast

Speed Kit
Built for Market Leaders

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

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

BAUR

A member of the otto group

FUTURE DIRECTIONS

Open Challenges

TTL Estimation

Quantifying Cacheability of Dynamic Content

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

  TTLs too short**:** Bad cache-hit rate

  TTLs too large: Bloom filter‘s false positive rate degrades

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

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

  Queries:

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

TTL Estimation

Learning Representations

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

Id-Lists

$\{id_1, id_2, id_3\}$

Less Invalidations

Object-Lists

$\{\ \{id{:}\,1, val{:}\,'a'\}, \{id{:}\,2, val{:}\,'b'\}, \{id{:}\,3, val{:}\,'c'\}\}$

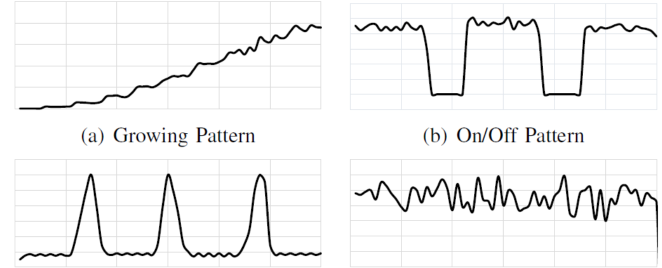
Less Round-Trips

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


(a) Growing Pattern (b) On/Off Pattern (c) Bursty Pattern (d) Random Pattern

„**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

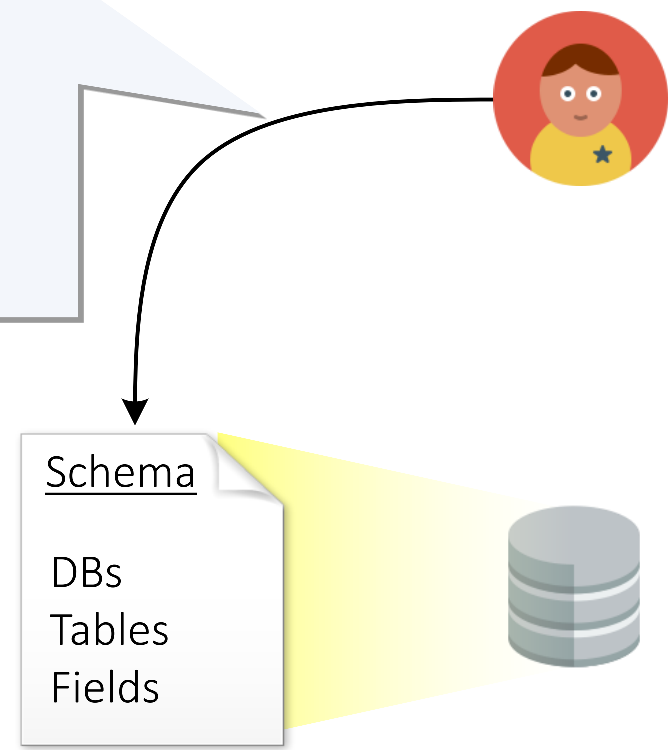
„**Reactive**“:

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

Polyglot Persistence Mediator

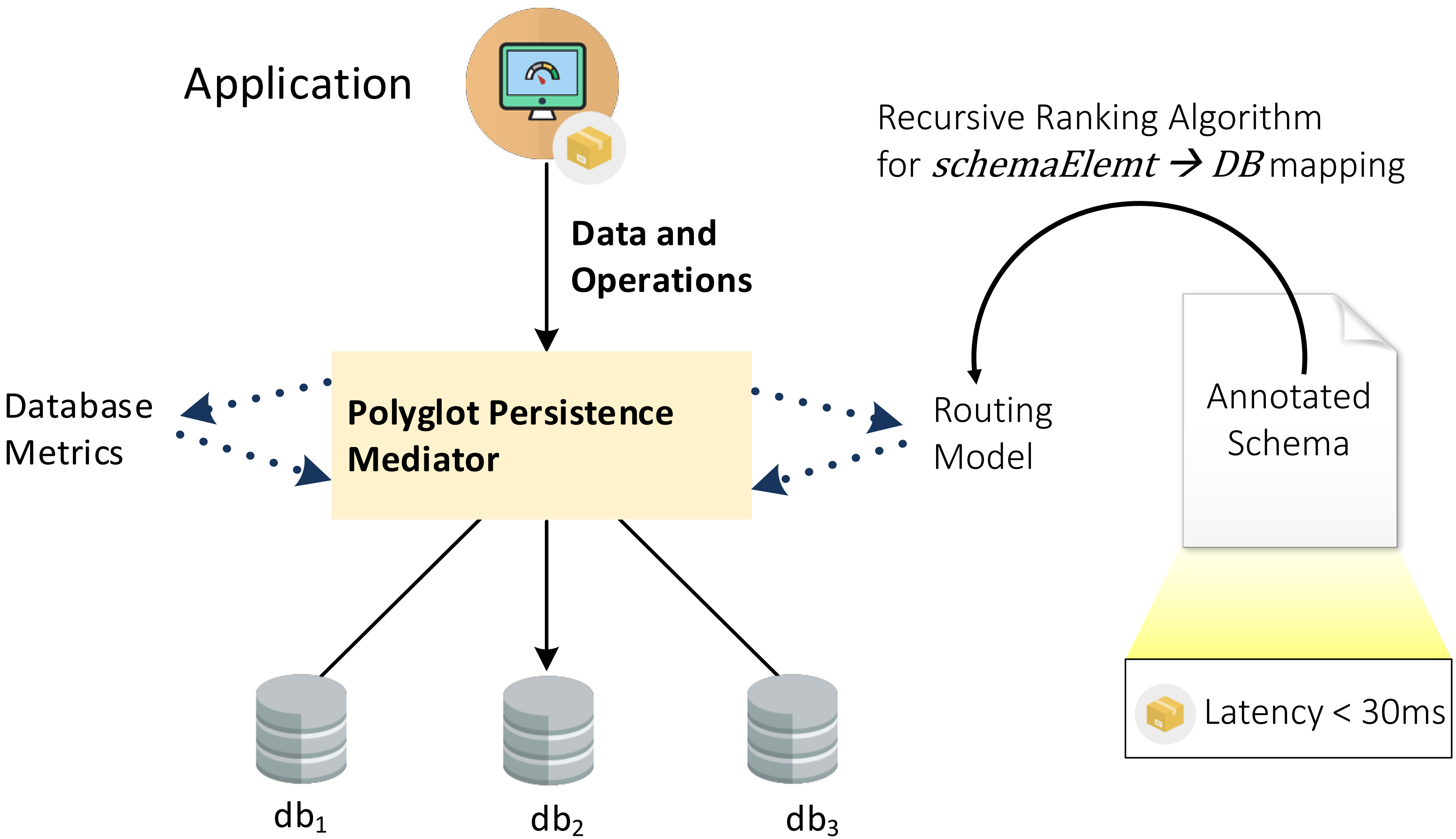
Schemas can be annotated with requirements/SLAs

- 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?

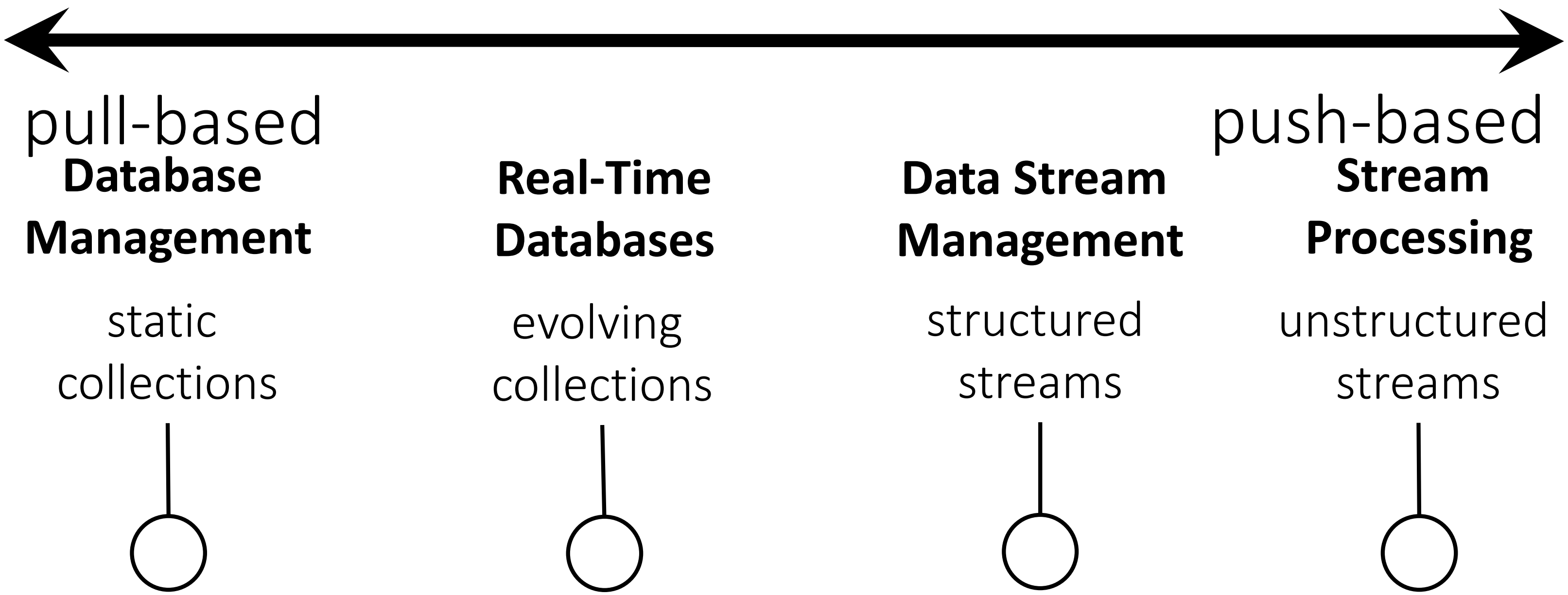
Transactional Visibility For Real-Time Queries: How to include transactions without introducing bottlenecks?

CLOSING TIME

Summary

Summary

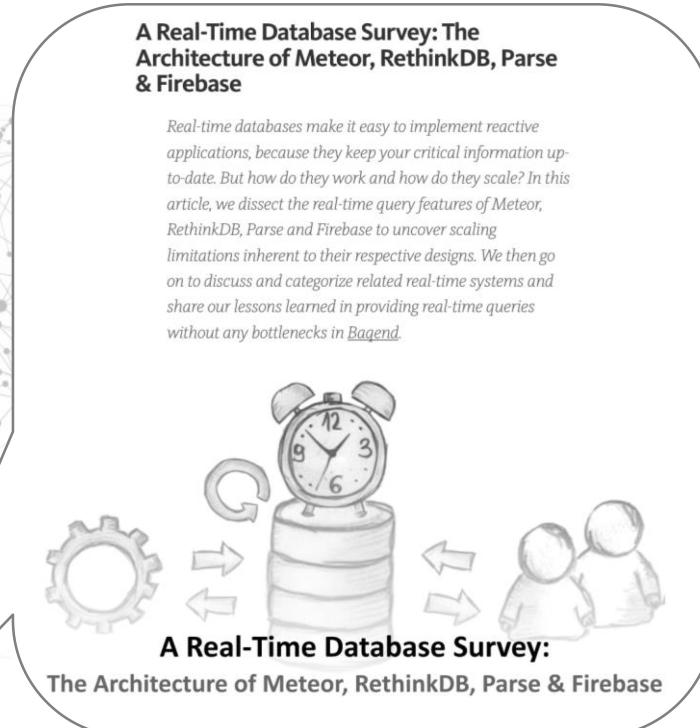
Real-Time Data Management



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



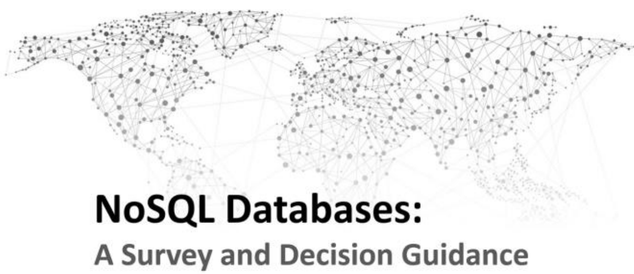
Blog Posts:

- *Real-Time Databases Explained: Why Meteor, RethinkDB, Parse and Firebase Don't Scale*
  Baqend Tech Blog (2017): https://medium.com/p/822ff87d2f87

Learn more at blog.baqend.com!

NoSQL Databases: a Survey and Decision Guidance

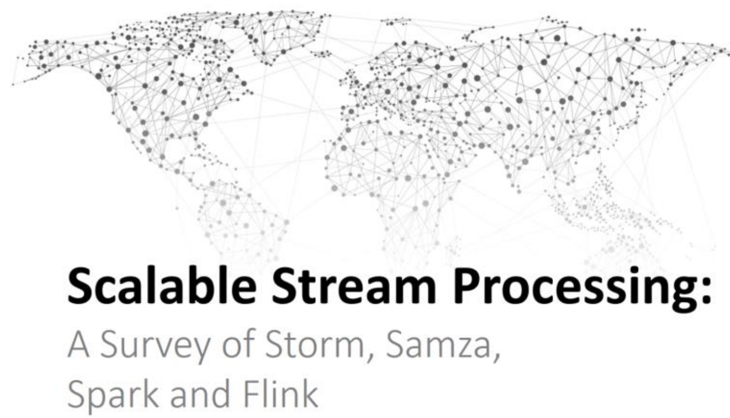
Together with our colleagues at the University of Hamburg, we—that is Felix Gessert, Wolfram Wingerath, Steffen Friedrich and Norbert Ritter—presented an overview over the NoSQL landscape at SummerSOC'16 last month. Here is the written gist. We give our best to convey the condensed NoSQL knowledge we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



TL;DR

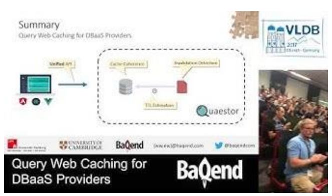
Today, data is generated and consumed at unprecedented scale. This has lead to novel approaches for scalable data management subsumed under the term "NoSQL" database systems to handle the ever-increasing data volume and request loads. However, the heterogeneity and diversity of the numerous existing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

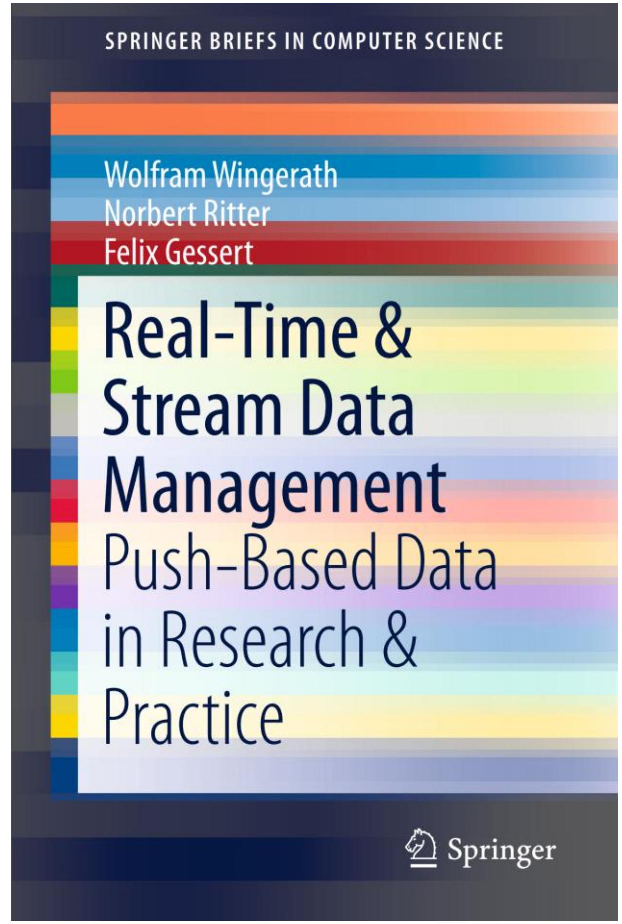
Scalable Stream Processing: A Survey of Storm, Samza, Spark and Flink



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 on blog.baqend.com!

For videos & book,
visit slides.baqend.com!

Thank you

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

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

@baqendcom

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