Wolfram Wingerath

Real-Time Processing Explained
A Survey of Storm, Samza, Spark & Flink

Architecture
PhD Thesis & Research

Research:
- Real-Time Databases
- Stream Processing
- NoSQL & Cloud Databases
- ...

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

About me
Wolfram Wingerath

Distributed Systems Engineer
Who We Are

Our Product

Speed Kit:
- Accelerates Any Website
- Pluggable
- Easy Setup

test.speed-kit.com

Our Services

- Web & Data Management Workshops
- Performance Auditing
- Implementation Services

consulting@baqend.com

Wolfram Wingerath

ww@baqend.com
Distributed Systems Engineer
Outline

- **Introduction**
  - Big Data in Motion

- **System Survey**
  - Big Data + Low Latency

- **Wrap-Up**
  - Summary & Discussion

- **Future Directions**
  - Real-Time Databases

**Big Picture:**
- A Typical Data Pipeline
- Processing Frameworks

**Processing Models:**
- Batch Processing
- Stream Processing

**Streaming Architectures:**
- Lambda Architecture
- Kappa Architecture
- Typical Operators
- Exemplary Use Case
IN PRACTICE

Scalable Data Processing
A Data Processing Pipeline

Today’s topic!
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
INTRODUCTION

Batch vs Stream Processing
Big Data Processing Frameworks
What are your options?

What to use when?

- Spark Streaming
- Google Dataflow
- HERON
- Spark
- STORM
- STORM Trident
- IBM InfoSphere Streams
- Amazon Elastic MapReduce
- Flink
- APEX
- kafka streams
- samza
- hadoop
- concord
Batch Processing

„Volume“

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

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

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

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

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

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

![Diagram of Lambda Architecture](http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html)
Kappa Architecture

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

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

---

Jay Kreps, *Questioning the Lambda Architecture* (2014)
[https://www.oreilly.com/ideas/questioning-the-lambda-architecture](https://www.oreilly.com/ideas/questioning-the-lambda-architecture)
Typical Stream Operators

Examples

Filter & Transform
- Filter
- Map

Group
- GroupByKey

Aggregates
- SUM()
- COUNT()

Windows
- Tumbling
- Sliding

https://www.infoq.com/presentations/stream-processors-databases
https://www.infoq.com/presentations/stream-processing-apache-flink
Typical Use Case
Example from Yahoo!

Input
• **Read** Ad tracking data **from Kafka**

Filter
• **Discard** useless data

Project
• **Extract** relevant fields

Group
• By Ad **campaign**

Window
• Ad views per **10-min-window**

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
Outline

Introduction
Big Data in Motion

System Survey
Big Data + Low Latency

Wrap-Up
Summary & Discussion

Future Directions
Real-Time Databases

- Processing Models: Stream ↔ Batch
- Stream Processing Frameworks:
  - Storm
  - Trident
  - Samza
  - Flink
  - Other Systems
SURVEY

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

- **Stream**
  - Low latency
  - Flink, Storm Trident, Samza

- **Micro-batch**
  - High throughput
  - Apache Spark Streaming, Amazon Elastic MapReduce

- **Batch**
  - 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
Directed Acyclic Graphs (DAG):
- **Spouts**: pull data into topology
- **Bolts**: do processing, emit data
- Asynchronous
- Lineage can be tracked for each tuple
  → At-least-once has 2x messaging overhead
Cluster Architecture

How Storm Scales

Submit Topology

Nimbus

Scheduling & Monitoring

Zookeeper

Handles coordination

Supervisor

Worker
Worker
Worker

Worker
Worker
Worker

Storm Slave

Supervisor

Worker
Worker
Worker

Worker
Worker
Worker

Storm Slave

JVM for each worker (runs spouts and bolts as tasks)
State Management
Recover State on Failure

• In-memory or Redis-backed reliable state
• Synchronous state communication on the critical path
  → infeasible for large state
Back Pressure
Throttling Ingestion on Overload

1. too many tuples → 2. tuples time out and fail
3. tuples get replayed

Approach: monitoring bolts’ inbound buffer
1. Exceeding high watermark → throttle!
2. Falling below low watermark → full power!
Trident
Stateful Stream Joining on Storm

Overview:

- Abstraction layer on top of Storm
- Released in 2012 (Storm 0.8.0)
- **Micro-batching**
- **New features:**
  - High-level API: aggregations & joins
  - Strong ordering
  - Stateful exactly-once processing
    → Performance penalty
Trident
Partitioned Micro-Batching

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

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

History

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

Illustration taken from: Jay Kreps, Questioning the Lambda Architecture (2014)
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

Samza
Local State

Advantages of local state:

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

• Fast lookups

Illustrations taken from: Jay Kreps, *Why local state is a fundamental primitive in stream processing* (2014)
Dataflow
Example: Enriching a Clickstream

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

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

Spark
„MapReduce successor“

Overview

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

History

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

Overview

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

History

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

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

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

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

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

Overview

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

History

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

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

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

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

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

Illustration taken from:
Outline

- **Introduction**
  - Big Data in Motion

- **System Survey**
  - Big Data + Low Latency

- **Wrap-Up**
  - Summary & Discussion

- **Future Directions**
  - Real-Time Databases

- **Future Directions**
  - Comparison Matrix
  - Other Systems
  - One-Line Takeaway
WRAP UP

Side-by-side comparison
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Storm</th>
<th>Trident</th>
<th>Samza</th>
<th>Spark Streaming</th>
<th>Flink (streaming)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strictest Guarantee</strong></td>
<td>at-least-once</td>
<td>exactly-once</td>
<td>at-least-once</td>
<td>exactly-once</td>
<td>exactly-once</td>
</tr>
<tr>
<td><strong>Achievable Latency</strong></td>
<td>&lt;&lt;100 ms</td>
<td>&lt;100 ms</td>
<td>&lt;100 ms</td>
<td>&lt;1 second</td>
<td>&lt;100 ms</td>
</tr>
<tr>
<td><strong>State Management</strong></td>
<td>![Circle] (small state)</td>
<td>![Circle] (small state)</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
</tr>
<tr>
<td><strong>Processing Model</strong></td>
<td>one-at-a-time</td>
<td>micro-batch</td>
<td>one-at-a-time</td>
<td>micro-batch</td>
<td>one-at-a-time</td>
</tr>
<tr>
<td><strong>Backpressure</strong></td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>no (buffering)</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
</tr>
<tr>
<td><strong>Ordering</strong></td>
<td>![X] between batches</td>
<td>within partitions</td>
<td>between batches</td>
<td>within partitions</td>
<td></td>
</tr>
<tr>
<td><strong>Elasticity</strong></td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![X]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
</tr>
</tbody>
</table>
Performance
Yahoo! Benchmark

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

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

Other Systems

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

Introduction
Big Data in Motion

System Survey
Big Data + Low Latency

Wrap-Up
Summary & Discussion

Future Directions
Real-Time Databases

- Real-Time Databases:
  - Why Push-Based Database Queries?
  - Where Do Real-Time Databases Fit in?

- Comparison Matrix:
  - Meteor
  - RethinkDB
  - Parse
  - Firebase
  - Baqend

- Use Cases at Baqend:
  - Query Caching
  - Real-Time Queries
REAL-TIME DBS

Combining databases with streaming
Traditional Databases

No Request? No Data!

What's the current state?

circular shapes

Query maintenance: periodic polling
→ Inefficient
→ Slow
**Push-Based Access For Evolving Domains**

Self-Maintaining Results

Find people in Room B:

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

- 1. Erik (5/10)
- 2. Wolle (20/18)
- 3. Wolle (22/8)
- Flo (4/3)
- Erik (10/3)
- Erik (15/11)
- Erik (5/10)
Data Management Overview

DBMS vs. Real-Time DB vs. Stream Processing

Database Management

- static collections
- pull-based

Real-Time Databases

- evolving collections

Stream Processing

- ephemeral streams
- push-based
# Real-Time Databases

## In a Nutshell

<table>
<thead>
<tr>
<th></th>
<th>Meteor</th>
<th>RethinkDB</th>
<th>Parse</th>
<th>Firebase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scales with write TP</strong></td>
<td>![Checkmark]</td>
<td>![Cross]</td>
<td>![Cross]</td>
<td>![Cross]</td>
</tr>
<tr>
<td><strong>Scales with no. of queries</strong></td>
<td>![Cross]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
</tr>
<tr>
<td><strong>Composite queries (AND/OR)</strong></td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
</tr>
<tr>
<td><strong>Sorted queries</strong></td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Cross]</td>
</tr>
<tr>
<td><strong>Limit</strong></td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Cross]</td>
</tr>
<tr>
<td><strong>Offset</strong></td>
<td>![Checkmark]</td>
<td>![Checkmark]</td>
<td>![Cross]</td>
<td>![Cross]</td>
</tr>
</tbody>
</table>
Presentation is loading
Why latency matters

Average: 9.3s

- 1000ms Loading...
- 1% Revenue
- 1% Visitors
- 1% Traffic
The Problem
Two Bottlenecks: Backend & Latency

High Latency

Backend
Solution: Global Caching
Fresh Data from Ubiquitous Web Caches

Low Latency

Less Processing
New Caching Algorithms
Solve Consistency Problem
New Caching Algorithms Solve Consistency Problem

8 Years Research & Development

Universität Hamburg
InvaliDB
Invalidating DB Queries

How to detect changes to query results:
“Give me the most popular products that are in stock.”
Going Real-Time
Query Caching & Subscribing

Keeps data up-to-date
InvaliDB
Filter Queries: Distributed Query Matching

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

Implementation:
• Apache Storm & Java
• MongoDB query language
• Pluggable engine
Baqend Real-Time Queries
Low Latency + Linear Scalability

Linear Scalability

Stable Latency Distribution

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

JavaScript API

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

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

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

- **Stream Processors:**
  - STORM
  - Flink
  - samza
  - Spark Streaming

- **Real-Time Databases** integrate Storage & Streaming

- **Learn more:** [slides.baqend.com](slides.baqend.com)