Big Data Analytics at Scale

Designing Complex Data Pipelines for Continuous Insights

Invited Talk at Kristu Jayanti College
March 01, 2024
Wolle

Material Available at https://wolle.science
A Presentation in 2 Parts

Part 1: Big Data Analytics Frameworks
(Stream Processing Systems)

Part 2: Data Validation at Scale
(Real-World Example)

Note: Not all material is included in the live presentation, but the slides and the above talks contain all the details!
Research:
• Stream Processing
• Real-Time Databases
• NoSQL & Cloud Systems
• ...

Practice:
• Web Caching
• Big Data Analytics
• Anger Management
• ...

I Am Wolle

Slides: wolle.science
Part 1: Big Data Analytics Frameworks

1. Processing Paradigms
   Where is the difference between batch & stream processing and what are their resp. benefits?

2. Stream Processing Systems
   How did the stream processing system space evolve and what are the landmark systems?

3. System Comparison & Trade-Offs
   What are fundamental design decisions and how do they reflect in the individual systems?
A Data Processing Pipeline

Part 1
(Stream Processing)

Part 2
(Industry Example)
Data Management Through the Ages: Historical Context

Relational Databases
- Entity-Relationship Model
- Triggers
- Ingres
- SQL Standard
- Starburst
- Telegraf
- HiPAC
- PostgreSQL
- Rapide
- Relational Model

CEP & Streams
- MapReduce
- Spark
- Bigtable
- GFS
- Dynamo
- Aurora & Borealis
- Storm
- Flink
- Samza
- Meteor
- Baqend

Stream Processing
- PostgreSQL
- HiPAC
- Starburst
- Telegraf
- Rapide
- Relational Model

Relational Databases
- Active Databases
- Big Data & NoSQL
- Real-Time Databases

1970
1980
1990
2000
today

Data Management Through the Ages: Historical Context

Relational Databases
- Entity-Relationship Model
- Triggers
- Ingres
- SQL Standard
- Starburst
- Telegraf
- HiPAC
- PostgreSQL
- Rapide
- Relational Model

CEP & Streams
- MapReduce
- Spark
- Bigtable
- GFS
- Dynamo
- Aurora & Borealis
- Storm
- Flink
- Samza
- Meteor
- Baqend

Stream Processing
- PostgreSQL
- HiPAC
- Starburst
- Telegraf
- Rapide
- Relational Model

Relational Databases
- Active Databases
- Big Data & NoSQL
- Real-Time Databases

1970
1980
1990
2000
today

Data Management Through the Ages: Historical Context

Relational Databases
- Entity-Relationship Model
- Triggers
- Ingres
- SQL Standard
- Starburst
- Telegraf
- HiPAC
- PostgreSQL
- Rapide
- Relational Model

CEP & Streams
- MapReduce
- Spark
- Bigtable
- GFS
- Dynamo
- Aurora & Borealis
- Storm
- Flink
- Samza
- Meteor
- Baqend

Stream Processing
- PostgreSQL
- HiPAC
- Starburst
- Telegraf
- Rapide
- Relational Model

Relational Databases
- Active Databases
- Big Data & NoSQL
- Real-Time Databases

1970
1980
1990
2000
today

Data Management Through the Ages: Historical Context

Relational Databases
- Entity-Relationship Model
- Triggers
- Ingres
- SQL Standard
- Starburst
- Telegraf
- HiPAC
- PostgreSQL
- Rapide
- Relational Model

CEP & Streams
- MapReduce
- Spark
- Bigtable
- GFS
- Dynamo
- Aurora & Borealis
- Storm
- Flink
- Samza
- Meteor
- Baqend

Stream Processing
- PostgreSQL
- HiPAC
- Starburst
- Telegraf
- Rapide
- Relational Model

Relational Databases
- Active Databases
- Big Data & NoSQL
- Real-Time Databases

1970
1980
1990
2000
today
Data processing frameworks hide complexities of scaling, e.g.:

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

Running in cluster

Running on single node

Scaling out
Big Data Processing Frameworks: What are your options?

What to use when?

- Google Dataflow
- Apache Spark
- Apache Storm
- Apache Flink
- Apache Hadoop
- Apache Kafka
- IBM InfoSphere Streams
- Apache Apex
- Apache Heron

Low latency vs. high throughput
Processing Models: Batch vs. Micro-Batch vs. Stream

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

- **Micro-batch**
  - Hadoop
  - Spark Streaming
  - High throughput

- **Batch**
  - Amazon Elastic MapReduce
  - Low latency
  - High throughput
Batch Processing: "Volume"

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

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

---

Persistence (e.g. HDFS) → Batch (e.g. MapReduce) → Serving (e.g. HBase) → Application
• 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
Streaming + retention (e.g. Kafka, Kinesis)

- **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

**Kappa Architecture:** Stream(D_{all}) = Batch(D_{all})
• 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
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**

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

- **Storm Slave**
  - JVM for each worker (runs spouts and bolts as tasks)

- **Storm Slave**
  - Supervisor
  - Worker
  - Worker

- **Storm Slave**
  - Supervisor
  - Worker
  - Worker
• **In-memory** or **Redis**-backed reliable state
• *Synchronous state communication* on the critical path
  → infeasible for large state
**Approach**: monitoring bolts’ inbound buffer
1. Exceeding **high watermark** → throttle!
2. Falling below **low watermark** → full power!
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 & scalability bottleneck
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
Data Flow: 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
- **Task = Kafka partitions**: not-elastic on purpose
  - **Ordering** within partitions

---

Advantages of local state:

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

Illustrations taken from: Jay Kreps, *Why local state is a fundamental primitive in stream processing* (2014)
Example: the *enriched clickstream* is available to every team within the organization.

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

Overview

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

◦ **Community**: 1000+ contributors in 2015

◦ **Big users**: Amazon, eBay, Yahoo!, IBM, Baidu, ...

History

◦ **2009**: developed at UC Berkeley

◦ **2010**: open-sourced

◦ **2014**: Apache top-level project
Overview

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

History

- **2011**: start of development
- **2013**: Spark Streaming becomes part of Spark Core
Resilient Distributed Data set (RDD)

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

**DStream**: Discretized RDD

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

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

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

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

History

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

DataStream (Java / Scala)  DataSet (Java/Scala)

Streaming dataflow runtime

YARN  Cluster  Local

https://www.infoq.com/presentation/s/stream-processing-apache-flink
• Automatic **Backups** of local state
• Stored in **RocksDB**, Savepoints written to **HDFS**
• **Ordering** within stream partitions
• Periodic **checkpoints**
• Recovery:
  1. *reset state* to checkpoint
  2. *replay data* from there

<table>
<thead>
<tr>
<th></th>
<th>Storm</th>
<th>Trident</th>
<th>Samza</th>
<th>Spark Streaming</th>
<th>Flink (streaming)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strictest Guarantee</strong></td>
<td>at-least-once</td>
<td>exactly-once</td>
<td>at-least-once</td>
<td>exactly-once</td>
<td>exactly-once</td>
</tr>
<tr>
<td><strong>Achievable Latency</strong></td>
<td>≪100 ms</td>
<td>&lt;100 ms</td>
<td>&lt;100 ms</td>
<td>&lt;1 second</td>
<td>&lt;100 ms</td>
</tr>
<tr>
<td><strong>State Management</strong></td>
<td>(small state)</td>
<td>(small state)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Processing Model</strong></td>
<td>one-at-a-time</td>
<td>micro-batch</td>
<td>one-at-a-time</td>
<td>micro-batch</td>
<td>one-at-a-time</td>
</tr>
<tr>
<td><strong>Backpressure</strong></td>
<td>✓</td>
<td>✓</td>
<td>no (buffering)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Ordering</strong></td>
<td>✗</td>
<td>between batches</td>
<td>within partitions</td>
<td>between batches</td>
<td>within partitions</td>
</tr>
<tr>
<td><strong>Elasticity</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
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**."

And even more: Kinesis, Gearpump, MillWheel, Muppet, S4, Photon, ...
Wrapup: Continuous Data Processing at Scale

Stream Processors:

- STORM
- Flink
- Samza
- Spark Streaming

Many Dimensions of Interest: consistency guarantees, state management, backpressure, ordering, elasticity, ...
Part 2: Data Validation at Scale

1. The Importance of Data Validation
   Where is data validation integrated into data science pipelines and what is its impact?

2. Data Quality & Constraints
   What dimensions of data quality are there and how can they be ensured?

3. Scalability-Related Challenges
   Why is data validation difficult in data-intensive domains?
How and Why Acceleration: speedhub.org

THE LARGEST SYSTEMATIC STUDY OF

Mobile Site Speed and the Impact on E-Commerce

Your Email

Subscribe for Insights

Google

Baqend

Universität Hamburg
Split Testing for Web Performance

Speed Kit Users vs. Normal Users

- **Speed Kit enabled**

- **Speed Kit disabled** (no acceleration)

  - **Measurable uplift:**
    - Performance
    - User engagement
    - Revenue
    - ...

Running **Example**: Web Performance Analysis Pipeline

Collection | Ingestion | Analytics | Reporting
---|---|---|---
Tracking (RUM) | | | Performance Dashboard
| | | QA Dashboard
| | | Real-Time Alerting
| | | Ad-hoc SQL Interface
| | | Custom Reporting

Running **Example**: Web Performance Analysis Pipeline

**Collection**

- Tracking (RUM)

**Ingestion**

- Flink

**Analytics**

**Reporting**

- Performance Dashboard
- QA Dashboard
- Real-Time Alerting
- Ad-hoc SQL Interface
- Custom Reporting

---

Running Example: Web Performance Analysis Pipeline

**Collection**

- Tracking (RUM)

**Ingestion**

- Flink

**Analytics**

**Reporting**

- Performance Dashboard
- QA Dashboard
- Real-Time Alerting
- Ad-hoc SQL Interface
- Custom Reporting

---


---

Wolfram Wingerath (KJC, March 01, 2024) Slides: wolle.science 44 / 78
Running **Example**: Web Performance Analysis Pipeline

Tracking (RUM)
Running Example: Web Performance Analysis Pipeline

Collection → Ingestion → Processing & Storage → Analysis, Reporting & ML

Dashboarding → Data Warehousing → Machine Learning
- **Goal**: verify that data in the pipeline is in an acceptable state for downstream processing, e.g.
  - External reporting (statistics, visualizations & dashboarding)
  - Internal reporting (debugging, product optimizations)
  - Decision-making (analytics, machine learning)

- Data validation can be integrated **in and between all stages**
There are different dimensions of data quality, especially:

- **Completeness**: Do we have all the data we need to assess page load performance?
- **Consistency**: Does data have a valid format and does it comply with business semantics?
- **Accuracy**: Do data items represent their corresponding real-world entities well?
- **Uniqueness**: Are duplicate records known and are all unique attributes actually distinct?
There are different dimensions of data quality, especially:

- **Completeness**: Do we have all the data we need to assess page load performance?
- **Consistency**: Does data have a valid format and does it comply with business semantics?
- **Accuracy**: Do data items represent their corresponding real-world entities well?
- **Uniqueness**: Are duplicate records known and are all unique attributes actually distinct?
• Is beacon loss a problem at all?

• **When** is beacon loss a problem?
  → For which beacon types? For which beacon strategies?

• **Where** is beacon loss a problem?
  → For which browsers? For which device types?
Unload Beacon Reliability by Strategy

```javascript
addEventListener("unload", (event) => {
    navigator.sendBeacon(url, JSON.stringify(data));
});

addEventListener("beforeunload", (event) => {
    navigator.sendBeacon(url, JSON.stringify(data));
});

addEventListener("pagehide", (event) => {
    navigator.sendBeacon(url, JSON.stringify(data));
});

addEventListener("visibilitychange", (event) => {
    if (document.visibilityState === 'hidden') {
        navigator.sendBeacon(url, JSON.stringify(data));
    }
});
```

- available on all platforms
- experimental feature (only available as origin trial in Chrome)

```javascript
var beacon = new window.PendingPostBeacon(
    url,
    {
        timeout: 60000,
        backgroundTimeout: 0
    }
);
beacon.setData(JSON.stringify(data));
```
Unload Beacon Reliability by Strategy

```javascript
addEventListener("unload", (event) => {
  navigator.sendBeacon(url, JSON.stringify(data));
});
```

Available on all platforms currently only available as origin trial in Chrome

```javascript
addEventListener("beforeunload", () => {
  navigator.sendBeacon(url, JSON.stringify(data));
});
```

```javascript
addEventListener("pagehide", () => {
  navigator.sendBeacon(url, JSON.stringify(data));
});
```

```javascript
if (document.visibilityState === 'hidden') {
  navigator.sendBeacon(url, JSON.stringify(data));
}
```

Sophie Ferrlein. Data Viz for Engineers: Optimizing Insight & Decision Making Through Visualization, code.talks (2023)

Slides: wolle.science
## Unload Beacon Reliability by Strategy & Device

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Mobile</th>
<th>Desktop</th>
</tr>
</thead>
<tbody>
<tr>
<td>PendingPostBeacon</td>
<td>97%</td>
<td>99%</td>
</tr>
<tr>
<td>visibilitychange</td>
<td>90%</td>
<td>91%</td>
</tr>
<tr>
<td>pagehide</td>
<td>81%</td>
<td>88%</td>
</tr>
<tr>
<td>unload</td>
<td>78%</td>
<td>87%</td>
</tr>
<tr>
<td>beforeunload</td>
<td>28%</td>
<td>73%</td>
</tr>
</tbody>
</table>

*Based on 52 million page views on a globally operating e-commerce site measured by Speed Kit's real user-monitoring.*

*Desktop data may be overrepresented!*

---


Wolfram Wingerath (KJC, March 01, 2024)  
Slides: wolle.science
Unload Beacon Reliability by Strategy & Browser

<table>
<thead>
<tr>
<th></th>
<th>Chrome</th>
<th>Safari</th>
<th>Firefox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pending PostBeacon</td>
<td>98%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>visibility change</td>
<td>92%</td>
<td>89%</td>
<td>95%</td>
</tr>
<tr>
<td>pagehide</td>
<td>88%</td>
<td>80%</td>
<td>91%</td>
</tr>
<tr>
<td>unload</td>
<td>86%</td>
<td>80%</td>
<td>90%</td>
</tr>
<tr>
<td>before unload</td>
<td>83%</td>
<td>3%</td>
<td>91%</td>
</tr>
</tbody>
</table>

Based on 52 million page views on a globally operating e-commerce site measured by Speed Kit's real user-monitoring | August 2023

Unload Beacon Reliability: The Ideal Combo Strategy

- `visibilitychange + pagehide + beforeunload + unload`
  - 91.7%
  - Cross

- `visibilitychange + pagehide + beforeunload`
  - 91.7%
  - Cross

- `visibilitychange + pagehide`
  - 91.3%
  - Tick

Based on 52 million page views on a globally operating e-commerce site measured by Speed Kit's real user monitoring | August 2023

Can be used with Backward-Forward Cache?
• There are different dimensions of data quality, especially:
  
  o Completeness: Do we have all the data we need to assess page load performance?
  
  o Consistency: Does data have a valid format and does it comply with business semantics?
  
  o Accuracy: Do data items represent their corresponding real-world entities well?
  
  o Uniqueness: Are duplicate records known and are all unique attributes actually distinct?

Note: Browser values should be unified for all records in the same session!

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Pageload ID</th>
<th>Browser</th>
<th>LCP (Performance)</th>
<th>Session ID</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:05:04.578</td>
<td>37ab08</td>
<td>Edge</td>
<td>&quot;670ms&quot;</td>
<td>123</td>
<td>null</td>
</tr>
<tr>
<td>13:26:48.139</td>
<td>9cddf7</td>
<td>Firefox</td>
<td>692 654</td>
<td>456</td>
<td>abc.de/red</td>
</tr>
<tr>
<td>13:28:23.857</td>
<td>0b577a</td>
<td>Firefox</td>
<td>0.256</td>
<td>456</td>
<td>abc.de/blue</td>
</tr>
<tr>
<td>13:29:17.468</td>
<td>faf55e</td>
<td>Edge</td>
<td>1.598</td>
<td>456</td>
<td>abc.de/sold</td>
</tr>
<tr>
<td>20:45:38.941</td>
<td>faf55e</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>abc.de/sold</td>
</tr>
</tbody>
</table>
There are different dimensions of data quality, especially:

- Completeness: Do we have all the data we need to assess page load performance?
- Consistency: Does data have a valid format and does it comply with business semantics?
- Accuracy: Do data items represent their corresponding real-world entities well?
- Uniqueness: Are duplicate records known and are all unique attributes actually distinct?

**Note:** Browser values should be unified for all records in the same session! → value may be replaced with majority vote (another reasonable option: replace old values with latest one)
Dimensions of Data Quality: Consistency

There are different dimensions of data quality, especially:

- Completeness: Do we have all the data we need to assess page load performance?
- Consistency: Does data have a valid format and does it comply with business semantics?
- Accuracy: Do data items represent their corresponding real-world entities well?
- Uniqueness: Are duplicate records known and are all unique attributes actually distinct?

Note: All values represent milliseconds, but formats differ depending on browser.
• There are different **dimensions of data quality**, especially:
  
  o **Completeness**: Do we have all the data we need to assess page load performance?
  
  o **Consistency**: Does data have a valid format and does it comply with business semantics?
  
  o **Accuracy**: Do data items represent their corresponding real-world entities well?
  
  o **Uniqueness**: Are duplicate records known and are all unique attributes actually distinct?

---

**Note**: All values represent milliseconds, but formats differ depending on browser. → values may be converted to integer.
There are different dimensions of data quality, especially:

- **Completeness**: Do we have all the data we need to assess page load performance?
- **Consistency**: Does data have a valid format and does it comply with business semantics?
- **Accuracy**: Do data items represent their corresponding real-world entities well?
- **Uniqueness**: Are duplicate records known and are all unique attributes actually distinct?

Note: Despite being in the right format, one value does not represent a reasonable timer value.
Dimensions of Data Quality: **Accuracy**

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Pageload ID</th>
<th>Browser</th>
<th>LCP (Performance)</th>
<th>Session ID</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:05:04.578</td>
<td>37ab08</td>
<td>Edge</td>
<td>670</td>
<td>123</td>
<td>null</td>
</tr>
<tr>
<td>13:26:48.139</td>
<td>9cddf7</td>
<td>Firefox</td>
<td>null</td>
<td>456</td>
<td>abc.de/red</td>
</tr>
<tr>
<td>13:28:23.857</td>
<td>0b577a</td>
<td>Firefox</td>
<td>256</td>
<td>456</td>
<td>abc.de/blue</td>
</tr>
<tr>
<td>13:29:17.468</td>
<td>faf55e</td>
<td>Firefox</td>
<td>1598</td>
<td>456</td>
<td>abc.de/sold</td>
</tr>
<tr>
<td>20:45:38.941</td>
<td>faf55e</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>abc.de/sold</td>
</tr>
</tbody>
</table>

**Note:** Despite being in the right format, one value does not represent a reasonable timer value. → broken value may be removed

- There are different **dimensions of data quality**, especially:
  - **Completeness**: Do we have all the data we need to assess page load performance?
  - **Consistency**: Does data have a valid format and does it comply with business semantics?
  - **Accuracy**: Do data items represent their corresponding real-world entities well?
  - **Uniqueness**: Are duplicate records known and are all unique attributes actually distinct?
There are different dimensions of data quality, especially:

- Completeness: Do we have all the data we need to assess page load performance?
- Consistency: Does data have a valid format and does it comply with business semantics?
- Accuracy: Do data items represent their corresponding real-world entities well?
- Uniqueness: Are duplicate records known and are all unique attributes actually distinct?

### Dimensions of Data Quality: **Uniqueness**

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Pageload ID</th>
<th>Browser</th>
<th>LCP (Performance)</th>
<th>Session ID</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:05:04.578</td>
<td>37ab08</td>
<td>Edge</td>
<td>670</td>
<td>123</td>
<td>null</td>
</tr>
<tr>
<td>13:26:48.139</td>
<td>9cddf7</td>
<td>Firefox</td>
<td>null</td>
<td>456</td>
<td>abc.de/red</td>
</tr>
<tr>
<td>13:28:23.857</td>
<td>0b577a</td>
<td>Firefox</td>
<td>256</td>
<td>456</td>
<td>abc.de/blue</td>
</tr>
<tr>
<td>13:29:17.468</td>
<td>faf55e</td>
<td>Firefox</td>
<td>1598</td>
<td>456</td>
<td>abc.de/sold</td>
</tr>
<tr>
<td>20:45:38.941</td>
<td>faf55e</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>abc.de/sold</td>
</tr>
</tbody>
</table>

**Note**: The ID field should be unique, but two different records share the same value!
There are different dimensions of data quality, especially:

- Completeness: Do we have all the data we need to assess page load performance?
- Consistency: Does data have a valid format and does it comply with business semantics?
- Accuracy: Do data items represent their corresponding real-world entities well?
- Uniqueness: Are duplicate records known and are all unique attributes actually distinct?
There are different **dimensions of data quality**, especially:

- **Completeness**: Do we have all the data we need to assess page load performance?
- **Consistency**: Does data have a valid format and does it comply with business semantics?
- **Accuracy**: Do data items represent their corresponding real-world entities well?
- **Uniqueness**: Are duplicate records known and are all unique attributes actually distinct?

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Pageload ID</th>
<th>Browser</th>
<th>LCP (Performance)</th>
<th>Session ID</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:05:04.578</td>
<td>37ab08</td>
<td>Edge</td>
<td>670</td>
<td>123</td>
<td>null</td>
</tr>
<tr>
<td>13:26:48.139</td>
<td>9cddf7</td>
<td>Firefox</td>
<td>null</td>
<td>456</td>
<td>abc.de/red</td>
</tr>
<tr>
<td>13:28:23.857</td>
<td>0b577a</td>
<td>Firefox</td>
<td>256</td>
<td>456</td>
<td>abc.de/blue</td>
</tr>
<tr>
<td>13:29:17.468</td>
<td>faf55e</td>
<td>Firefox</td>
<td>1598</td>
<td>456</td>
<td>abc.de/sold</td>
</tr>
<tr>
<td>20:45:38.941</td>
<td>faf55e</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>abc.de/sold</td>
</tr>
</tbody>
</table>

*Note*: The ID field should be unique, but two different records share the same value! → merge duplicates into a single record
• **Constraints** are rules, conditions, or limits that data must adhere to

• **Type Checks** represent expectations on the data format, e.g.
  - *Value range* for numerical data (e.g. \([0, \text{MAX\_INTEGER})\) for load timers)
  - *Format or pattern* for string-valued data (e.g. ISO 8601 for timestamps)
  - *Structure* for complex attributes (e.g. required keys for JSON objects)

• **Complex conditions** can further describe complex semantics such as
  - Cross-field or cross-record relationships (e.g. same browser within sessions)
  - Referential integrity between records in different collections
  - Custom constraints for domain semantics
```
import pandas as pd
import pandera as pa
from pandera import Column, DataFrameSchema, Check

# Define the schema
schema = DataFrameSchema(
    {
        "Timestamp": Column(pa.DateTime),
        "Pageload ID": Column(pa.String, Check(lambda x: x.str.len() == 8)),
        "Browser": Column(pa.String, Check(lambda x: x.isin(['Chrome', 'Edge', 'Firefox']))),
        "LCP": Column(pa.Int, Check(lambda x: (x >= 0) & (x <= 600000))),
        "Session ID": Column(pa.Int),
    }
)
```
# valid data item

```python
good = {
    "Timestamp": [pd.Timestamp("2023-05-10 13:26:48.139")],
    "PageLoad ID": ["9cddf7"],
    "Browser": ["Firefox"],
    "LCP": [256],
    "Session ID": [456],
}
```

# invalid data item

```python
bad = {
    "Timestamp": [pd.Timestamp("2023-05-10 09:05:04.578")],
    "PageLoad ID": ["37ab08"],
    "Browser": ["Edge"],
    "LCP": [692654], # timer value out of bounds
    "Session ID": [123],
}
```
for record in [good, bad]:
    record_id = record['Pageload ID'][0]
    try:
        validated = schema(pd.DataFrame(record))
        print(f"\nValidation passed for record {record_id}!"")
    except pa.errors.SchemaError as e:
        print(f"\nValidation FAILED for record {record_id}:")
        print(e)

Validation passed for record 9cddf7!

Validation FAILED for record 37ab08
<Schema Column(name=LCP, type=DataType(int64))> failed element-wise validator 0:
<Check <lambda>>
failure cases:
  index  failure_case
  0      692654

Example: Declarative Constraints With Pandera (3/3)
Fundamental Challenge: Scalability

One Month in Data Errors at Baqend: April 2023

Data Errors by Type:
- Aggregation (66.89%)
- Attribute (32.83%)
- Internal (0.16%)

Data Errors by Cause:
- Missing Value (63.91%)
- Blank Value (11.92%)
- ValueOutOfRange (8.60%)
- Ambiguous Value (7.98%)
- StringLengthExceeded (4.30%)
- ValueOutOfRange (2.46%)
### Challenging at Scale: Complexity

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Pageload ID</th>
<th>Browser</th>
<th>LCP (Performance)</th>
<th>Session ID</th>
<th>URL</th>
<th>Session Length</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>09:05:04.578</td>
<td>37ab08</td>
<td>Edge</td>
<td>670</td>
<td>123</td>
<td>null</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>13:26:48.139</td>
<td>9cddf7</td>
<td>Firefox</td>
<td>null</td>
<td>456</td>
<td>abc.de/red</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>13:28:23.857</td>
<td>0b577a</td>
<td>Firefox</td>
<td>256</td>
<td>456</td>
<td>abc.de/blue</td>
<td>...</td>
<td>3</td>
</tr>
<tr>
<td>13:29:17.468</td>
<td>faf55e</td>
<td>Firefox</td>
<td>1598</td>
<td>456</td>
<td>abc.de/sold</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>20:45:38.941</td>
<td>faf55e</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>abc.de/sold</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- Manual constraint definition is often infeasible, because of...
  - ... inherent data complexity (often **hundreds** of attributes)
  - ... aggregation, derived storage, and evolving schemas
  - ... a plethora of other data stores to integrate!

→ **Automation** is necessary!
Computing validation metrics from scratch periodically can be infeasible, because of ...

- ... strict timing requirements
- ... efficiency or cost reasons
- ... data privacy reasons

→ Incremental computation can be the only option!
Specifying generalized constraints can be difficult in large deployments, because of...

- temporal fluctuations (e.g. throughout the day, on black Friday, or during holidays)
- multi-tenancy (e.g. different data patterns by customer timezone or domain)

→ **Elasticity & Multi-Tenancy** requirements can be challenging!
Deployments can break things (and cause sudden load spikes for validation)

- **Availability**
  Can this take down your data pipeline?
- **Attribution**
  Who or what is causing the problem?
- **Responsibility**
  Can you fix the problem?
Advanced Techniques
- Inferring constraints
- Adapting to schema changes
- Incremental computation of complex measures

Tooling & Frameworks
- Validation libraries such as Great Expectations, Pandera, TFDV, or Deequ
- Preprocessing and validation with Apache Spark and Apache Flink

Further Challenges
- Handling distribution (validation per partitioning, avoiding skew, ...)
- Efficiency and performance (load distribution, approximation, ...)
- Operational challenges (anomaly detection, fixing, load shedding, ...)

So How Do You Handle All This?
• **Data Quality** can be measured along dimensions such as completeness, consistency, accuracy, and uniqueness

• **Constraints** specify expectations about the data and can be used to enforce them

• **Data Validation** is the process of ensuring high data quality for processes like analysis, modeling, and decision-making

• Data Validation **Challenges at Scale** include
  
  o **Complexity**: schemas are often too complex to define constraints manually
  
  o **Volatility**: data varies throughout the day, by season, or by customer
  
  o **Continuity**: incremental processing is required when computation from scratch is infeasible
For videos & books, visit https://wolle.science!
• The **classic challenges** of Big Data management are also known as the „3 Vs of Big Data“:
  o **Volume**: How to reliably process huge amounts of data?
  o **Velocity**: How to keep throughput up and latency down?
  o **Variety**: How to handle all kinds of structured & unstructured data?

• But there are **additional challenges**:
  o **Veracity**: Is your data (source) trustworthy / meaningful?
  o **Visualization**: How to communicate? insights & knowledge?
  o **Value**: How to use data for (machine) learning, optimization, ...?
  o (Volatility, Vulnerability, Validity, ...)
Thanks! Questions?

Material Available at https://wolle.science