Stream Computing
(The engineer’s perspective)

Ilya Ganelin
Batch vs. Stream

• Batch
  • Process chunks of data instead of one at a time
  • Throughput over latency (seconds, minutes, hours)
  • E.g. MapReduce, Spark, Tez

• Stream
  • Data processed one at a time
  • Latency over throughput (microseconds, milliseconds)
  • E.g. Storm, Flink, Apex, KafkaStreams, GearPump
Scalability, Performance, Durability, Availability

• How do we handle more data?

• Quickly?

• Without ever losing data or compute?

• And ensure the system keeps working, even if there are failures?
What are the tradeoffs?

- If we focus on **scalability**, it’s harder to guarantee
  - Durability – more moving pieces, more coordination, more failures
  - Availability – more failures, harder to stay operational
  - Performance – bottlenecks and synchronization

- If we focus on **availability**, it’s harder to guarantee
  - Performance – monitoring and synchronization overhead
  - Scalability and performance
  - Durability – must recover without losing data

- If we focus on **durability**, it’s harder to guarantee
  - Performance
  - Scalability
Batch compute has it easy.

- Get **scale-out** and **performance** by adding hardware and **taking longer**

- Get **durability** with a durable data store and **recompute**

- Get **availability** by **taking longer to recover** (this makes life easier!)

- In **stream** processing, **you don’t have time**!
It’s not about performance and scale.

• Most platforms handle large volume of data relatively quickly

• It’s about:
  • **Ease of use** – how quickly can I build a complex application? Not word count.
  • **Failure-handling** – what happens when things break?
  • **Durability** – how do I avoid losing data without sacrificing performance?
  • **Availability** – how can I keep my system operational with a minimum of labor and without sacrificing performance?
Next: Case Studies in Open-Source Streaming

- Storm
- Flink
- Apex
Apache Storm

• Tried and true, was deployed on 10,000 node clusters at Twitter
  • Scalable
  • Performant
  • Easy to use

• Weaknesses:
  • Failure handling
  • Operationalization at scale
  • Flexibility

• Obsolete?
How does it work?
How does it work?
How does it work?
Failure Detection

Reliable Processing

Acks are delivered via a system-level bolt
Failure Detection

No durability of data in flight or guarantee of exactly once processing!
Where do the weakness come from?

• Nimbus was a single point of failure (fixed as of 1.0.0 release)
• Upstream bolt/spout failure triggers re-compute on entire tree
  • Can only create parallel independent stream by having separate redundant topologies
• Bolts/spouts share JVM → Hard to debug
• Failed tuples cannot be replayed quicker than 1s (lower limit on Ack)
• No dynamic topologies
• Cannot add or remove applications without service interruption
• Poor resource sharing in large clusters
Enter the Competition – Apache Flink

• Declarative functional API (like Spark)
• But, true streaming platform (sort of) with support for CEP
• Optimized query execution

• Weaknesses:
  • Depends on network micro-batching under the hood!
  • Not battle-tested
  • Failures still affect the entire topology
How does it work?
Failure Handling

1. **align barriers**
2. **checkpoint state**
3. **emit barrier and continue**
So what’s different from Storm?

• Flink handles planning and optimization for you
• Abstracts lower level internals
• Clear semantics around windowing (which Storm has lacked)
• Failure handling is lightweight and fast!
• Exactly once processing (given appropriate connectors at start/end)
• Can run Storm
What can’t it do?

• Dynamically update topology
• Dynamically scale
• Recover from errors without stopping the entire DAG
• Allow fine-grained control of how data moves through the system – locality, data partitioning, routing
  • You can do these individually, but not all at once
  • The high level API is a curse!
• Run in production (Maybe?)
So what else is there?
Which are unique?

• Apache Beam (Google’s baby - unifies all the platforms)

• Apache Apex (Robust architecture, scalable, fast, durable)

• IBM InfoSphere Streams (proprietary, expensive, the best)
Let’s look at Apex

• Unique provenance
  • Built for the business at Yahoo – not a research project
  • Built for reliability and strict processing semantics, not performance
  • Apex just works

• Strengths
  • Dynamism
  • Scalability
  • Failure-handling

• Weaknesses
  • No high-level API
  • More complex architecture
How does it work?
Failure Handling
So it’s the best? Sort of!

• Most robust failure-handling
• Allows fine-tuning of data flows and DAG setup
• Excellent exploratory UI

• But
  • Learning curve
  • No high-level API
  • No machine learning support
  • Built for business, not for simplicity
Streaming is great – what about state?

• What if I need to persist data?

• Across operators?

• Retrieve it quickly?

• Do complex analytics?

• And build models?
Why state?

• Historical features (e.g. spend amount over 30 days)

• Statistical aggregates

• Machine learning model training

• Why Cross operator? Because of how data is partitioned, allows aggregation over multiple fields.
Distributed In-Memory Databases

• Can support low-latency streaming use cases

• Durability becomes complicated because memory is volatile

• Memory is expensive and limited

• Examples: Memcached, Redis, MemSQL, Ignite, Hazelcast, Distributed Hash Tables