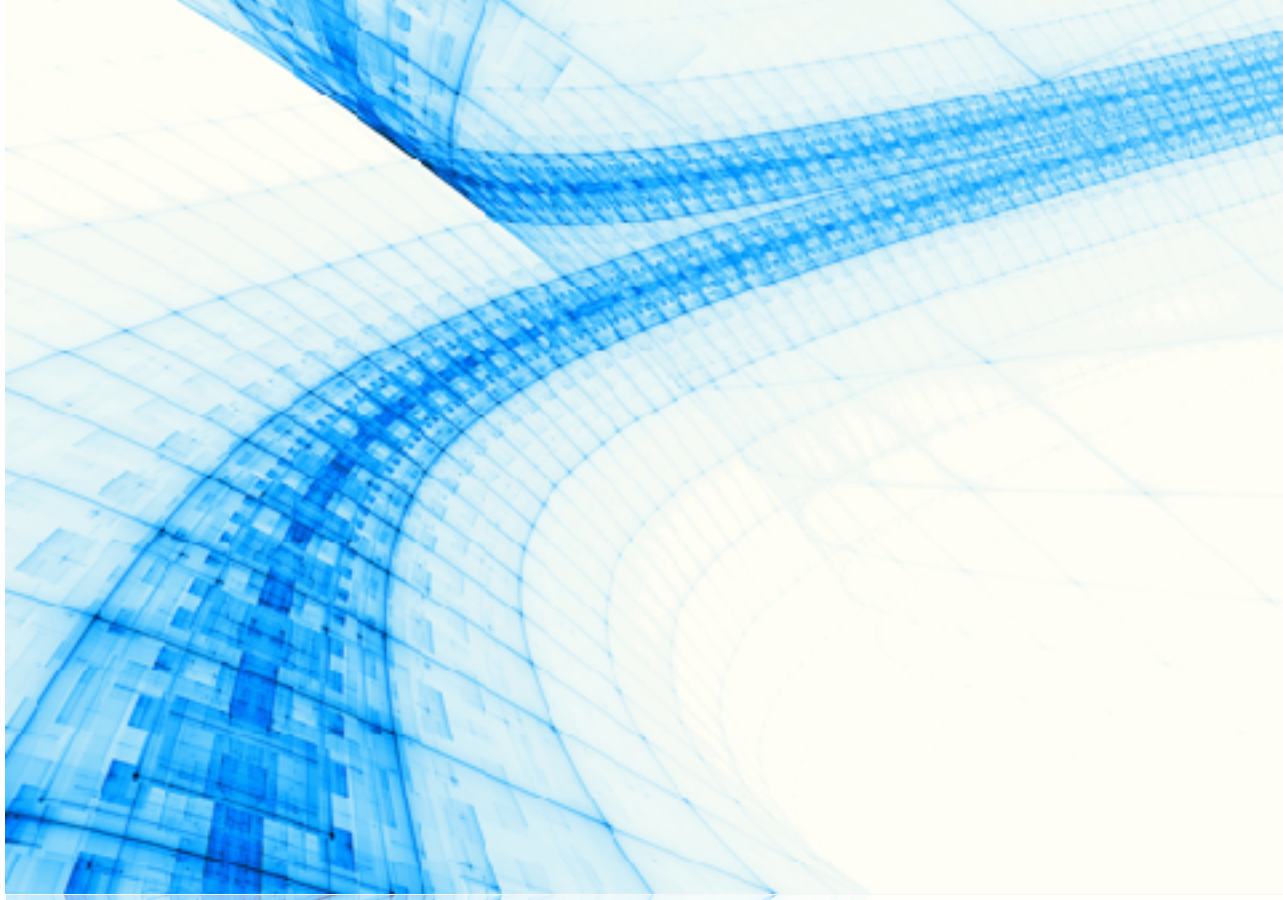


Systems for data stream

AsterixDB and Storm

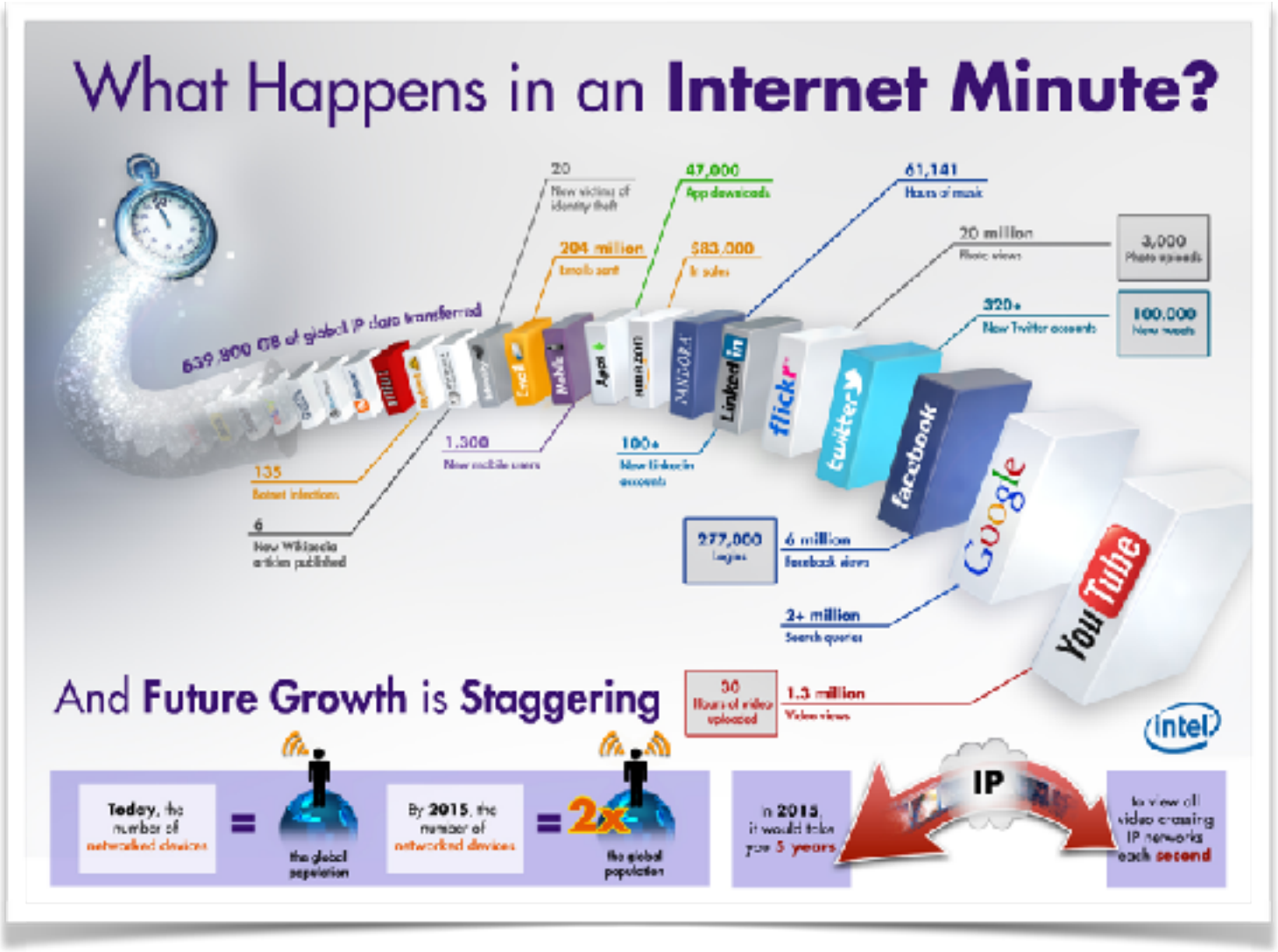
Aims

- To know how streaming data are handled in real-world systems: AsterixDB and Storm

Content

- Motivation
- Introduction to Storm
- Comparison between Storm and Spark
- Introduction to AsterixDB & Feeds

Streaming Data generation



Dealing with Streaming Data

- Continuous processing, aggregation and analyzing data **when created**
- System made like **directed acyclic graph**
- Data reduced before being centralized
- Processing messages one at time
- Collect structured, semi-structured and unstructured data from different sources

Streaming Data

- Actions in real time:
  - Monitoring trends
  - Communicate
  - Recommendation
  - Searching
- Expected response from 500 ms to one minute
- Data message flow through a chain of processors until the result reaches the final destination.
- To guarantee reliable data processing, necessary to restart processing in case of failures

Streaming Data (2)

- Delivery Semantics (Message Guarantees):
  - **At most once:**
    messages may be lost but never redelivered.
  - **At least once:**
    messages will never be lost but may be redelivered.
  - **Exactly once:**
    messages are never lost and never redelivered, perfect message delivery.

Fault-Tolerance

- **Soft Failure:** Inability to pre-process or ingest a received record either due to unexpected (null) value, duplicate record or due to inherent bug(s) in user provided code (UDF).
- **Hard Failure:** Inability to support dataflow due to loss of one of more participant nodes

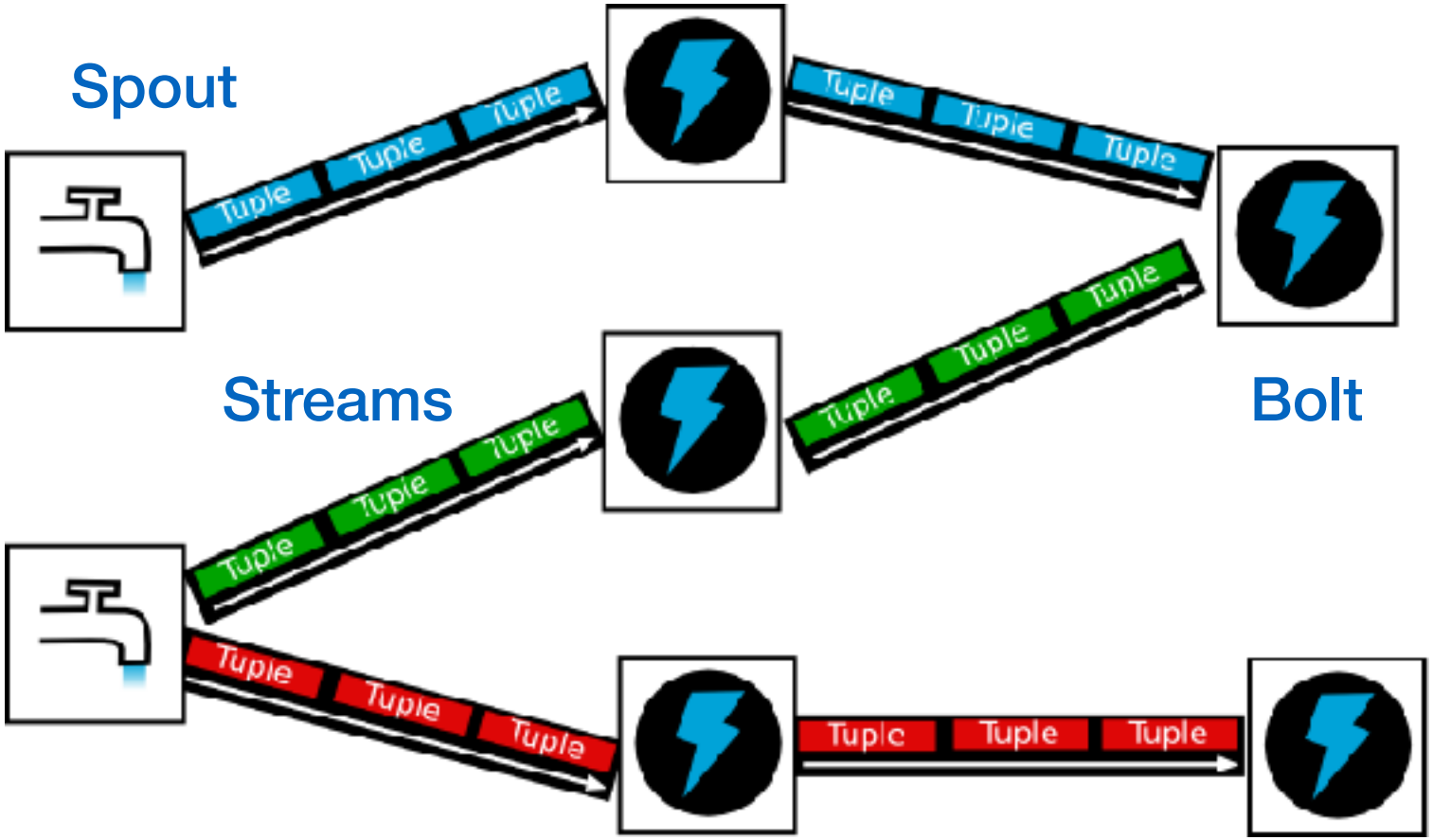
STORM

An open source, scalable, fault-tolerant, **distributed real-time computation system**.

Storm Concept

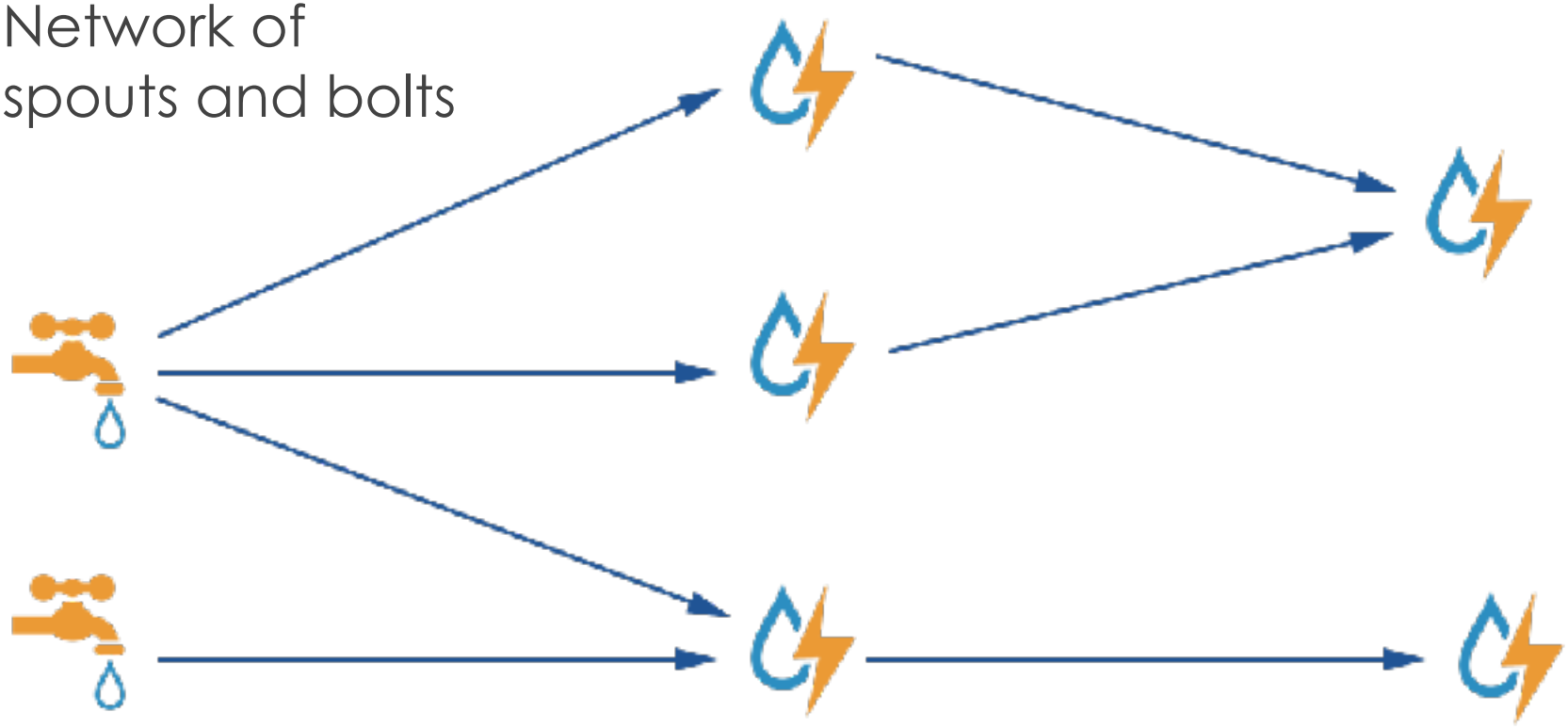
- **Streams**
  - Unbounded sequence of tuples
- **Spout**
  - Source of Stream
  - E.g. Read from Twitter streaming API
- **Bolts**
  - Processes input streams and produces new streams
  - E.g. Functions, Filters, Aggregation, Joins

Storm Concept



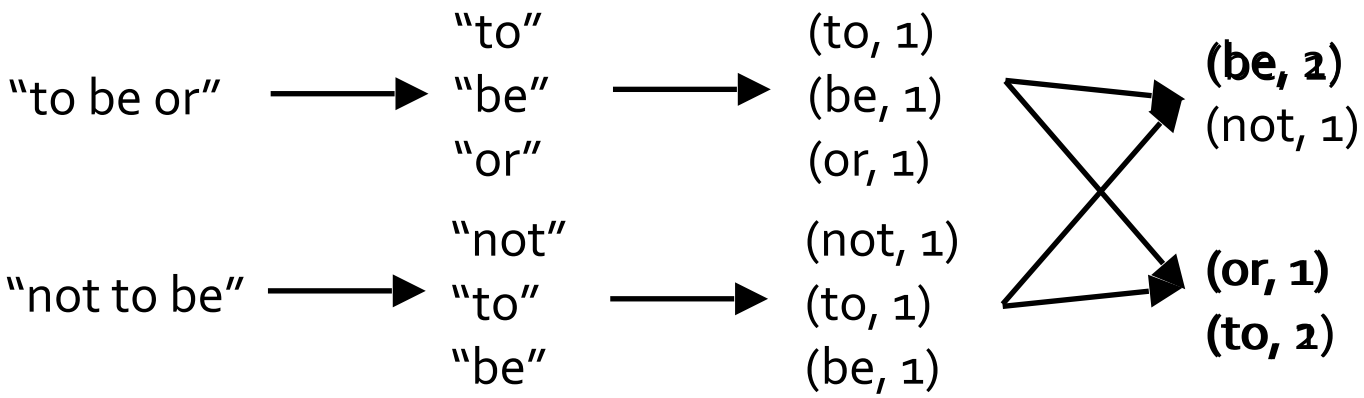
Storm Concept

- **Topologies**
  - Network of spouts and bolts

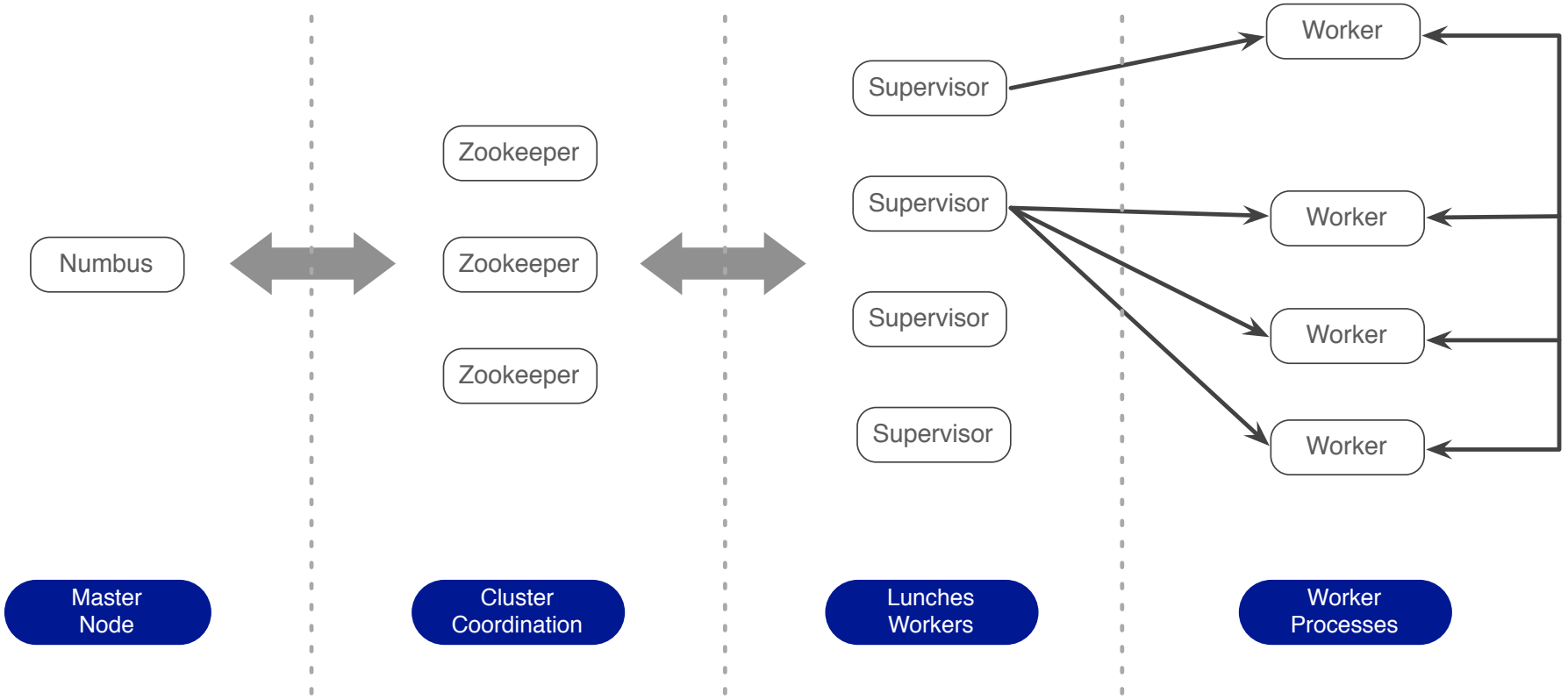


Word counting w/ Storm

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts = topology.newStream("spout1", spout)
        .each(new Fields("sentence"), new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
        .persistentAggregate(new MemoryMapState.Factory(), new Count(),
                new Fields("count")).parallelismHint(6);
```



Storm Architecture



Storm vs. Spark (2)

- Scale
- Latency
- Iterative Processing
  - Are there suitable non-iterative alternatives?
- Use What You Know
- Code Reuse
- Maturity

Storm Advantages

- **Fault tolerance:**
  - if worker threads die, or a node goes down, the workers are automatically restarted;
- **Scalability:**
  - throughput rates of even one million 100 byte messages per second per node can be achieved
- **Ease of use** in deploying and operating the system

VS.

	Storm	Spark
Processing Model	Event-Streaming	Micro-Batching / Batch (Spark Core)
Delivery Guarantees	At most once / At least once	Exactly Once
Latency	Sub-second	Seconds
Language Options	Java, Clojure, Scala, Python, Ruby	Java, Scala, Python
Development	Use other tool for batch	Batching and streaming are very similar

Spark Recommended

- Iterative Batch Processing (most Machine Learning)
  - Restricted alternatives
  - Some scale issues
- ETL - Extract, Transform and Load
- Shark/Interactive Queries

Spark Recommended (2)

- < 1 TB (or memory size of your cluster)
- Tuning to run well can be a pain
- Data Bricks and others are working on scaling.
- Streaming all μ-batch:
  - latency at least 1 sec
- Still single points of failure in streaming
- All streaming inputs replicated in memory

Storm recommended

- Latency < 1 second (single event at a time)
  - There is little else (especially not open source)
- "Real Time" …
  - Analytics
  - Budgeting
  - ML
  - Anything

- Lower Level API than Spark
- No built-in concept of look back aggregations
- Takes more effort to combine batch with streaming

VS.

Storm:

- Tends to be driven by creating classes and implementing interfaces
- Advantage of broader language support (code written in R or any other language not natively supported by Spark)
- DAG's as natural processing model
  Tuple natural interface for the data passed between nodes
- Processing excels at computing transformations as data are ingested with sub-second latencies

VS.

Spark:

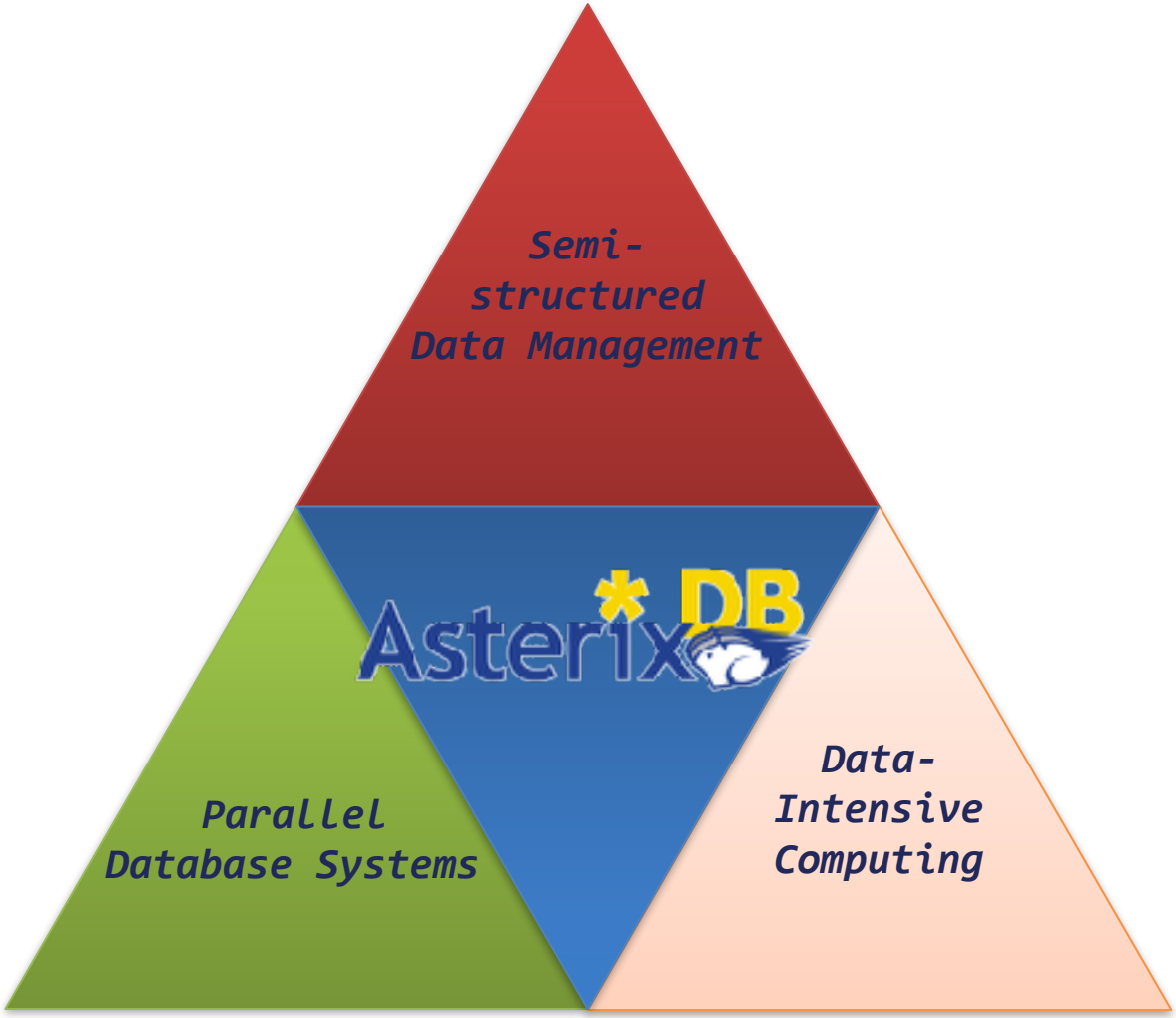
- Has more of a "functional" flavor, where working with the API is driven more by chaining successive method calls to invoke primitive operations
- Tuples can feel awkward in Java but with this is going benefit of compile-time
- Use existing Hadoop or Mesos cluster
- Micro-batching trivially gives stateful computation, making windowing an easy task.

Conclusion: Neither approach is better or worse

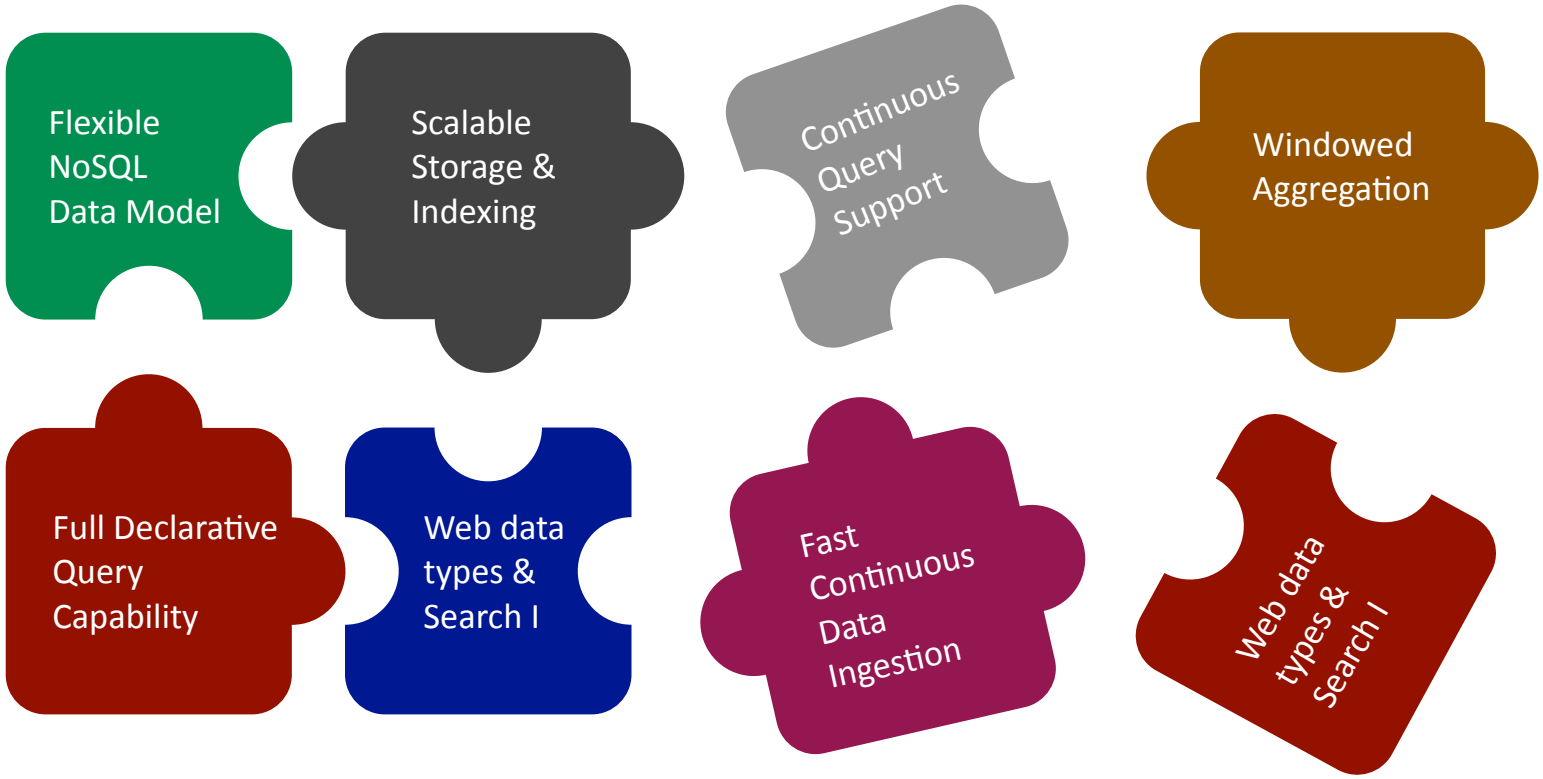
An Open source BigData Management System

Existing Solution(s)

AsterixDB: "One Size Fits a Bunch"



AsterixDB


Flexible NoSQL Data Model
Scalable Storage & Indexing
Continuous Query Support
Windowed Aggregation
Full Declarative Query Capability
Web data types & Search I
Fast Continuous Data Ingestion
Web data types & Search I

AsterixDB Features

Data feeds

Process incoming live data, and store it seamlessly.

Scalable runtime

AsterixDB is built on Hyracks, a parallel dataflow engine scale tested on hundreds of nodes.

External data

Query and index your data in HDFS alongside data stored in AsterixDB.

Semi-structured data model

Give your data as much or as little schema as you want, in one system.

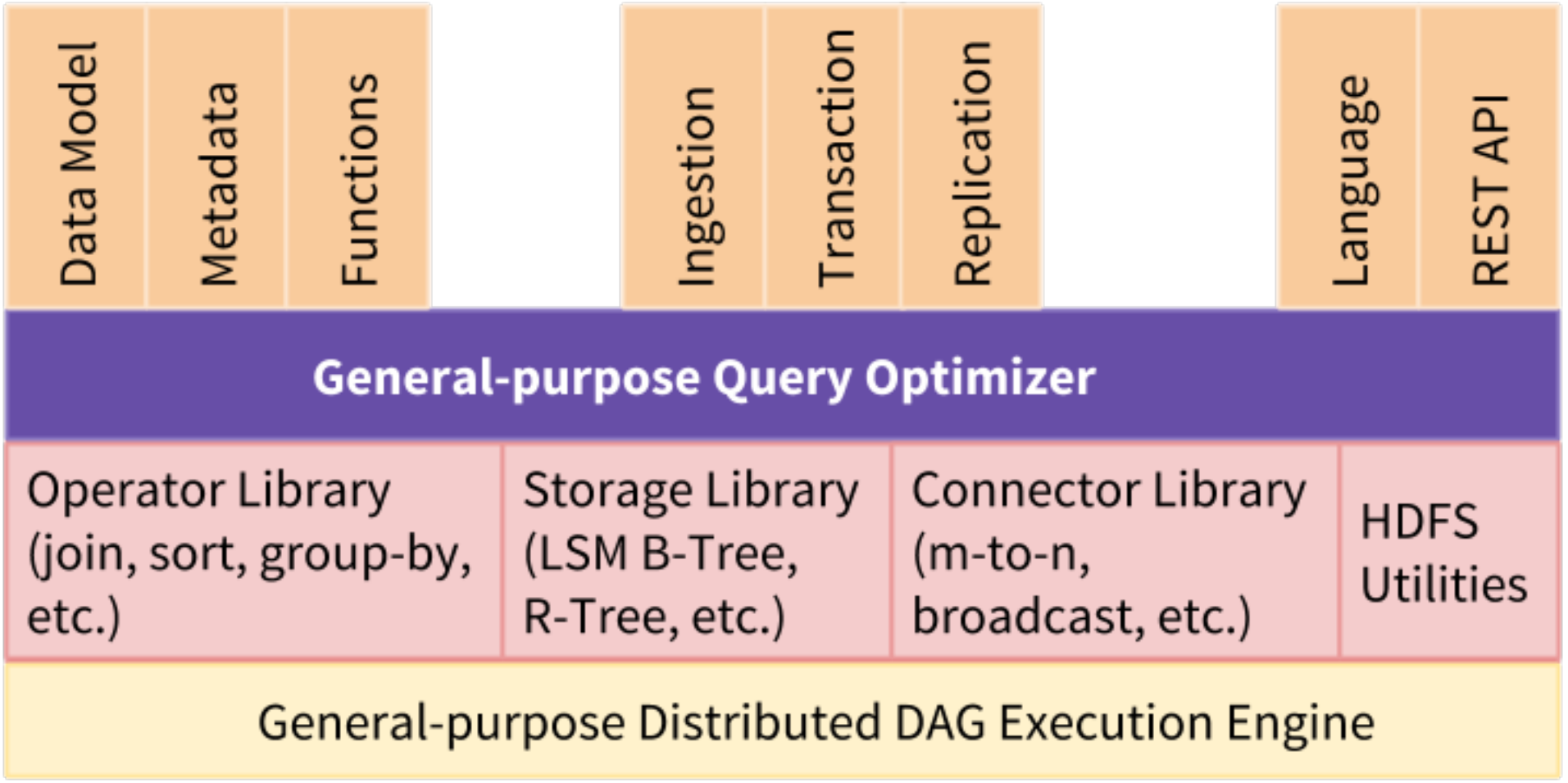
Log-structured storage

AsterixDB uses state of the art indexing techniques to support high speed data ingestion.

Rich datatypes

AsterixDB extends JSON datatypes with support for spatial and temporal data.

AsterixDB Architecuture



AsterixDB Data Model

```
create type TwitterUser as open {
screen-name: string,
   lang: string,
   friends_count: int32,
   statuses_count: int32,
   name: string,
   followers_count: int32
};
```

```
create type ProcessedTweet {
tweetid: string,
   userid: string,
   send-time: string,
   message-text: string,
   language: string,
   location: point?,
   referred-topics: {{string}}
};
```

```
create type RawTweet as open
{  tweetid: string,
   user: TwitterUserType,
   latitude: double?,
   longitude: double?,
   send-time: string,
   message-text: string,
   language: string
};
```

```
create dataset RawTweets(RawTweet)
primary key tweetid;

create dataset
ProcessedTweets(ProcessedTweet)
primary key tweetid;

create index locationIdx on
ProcessedTweets(location) type rtree;
```

Scalable Storage and Indexing

- Data is hash-partitioned on the basis of primary key
- Primary Index partition on each node with co-located secondary index partitions.
- **L**og **S**tructured **M**erge (LSM)-based Index
  A deferred-update, append-only data structure optimized for insert-intensive workloads [P. O'Neil et al, Acta Informatica, 1996]
  - Adopted in popular NoSQL systems
- "Storage Management in AsterixDB", VLDB, 2015

Full Declarative Query Support

Popularity of tourist spots as measured by the number of tweets mentioning a tourist spot:

```
for $tweet in dataset ProcessedTweets
for $spot in dataset TouristSpots
where some $topic in $tweet.referred-topics
satisfies contains($topic, $spot.name)
group by $spot.name with $tweet
return {
    ”spot": $spot.name,
    "count": count($tweet)
}
```

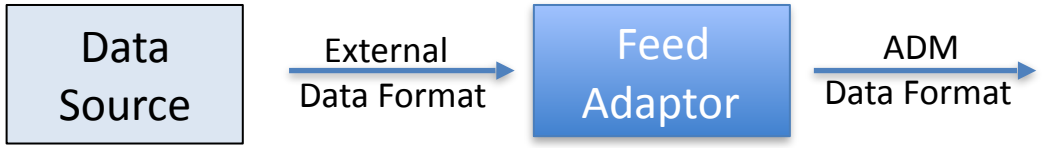
Focus: Data Feeds

Data Feed: Flow of data from an external source into persistent (indexed) storage inside a **B**ig **D**ata **M**anagement **S**ystem (BDMS).

Data Feed Management: Task of maintaining the continuous flow of data.

Data Ingestion Basics

- AsterixDB Query Language (AQL) with built-in support for data ingestion.



- Functionality of establishing connection with a data source and receiving, parsing and translating data into ADM records -> Feed Adaptor

- **Pull/Push-based ingestion**

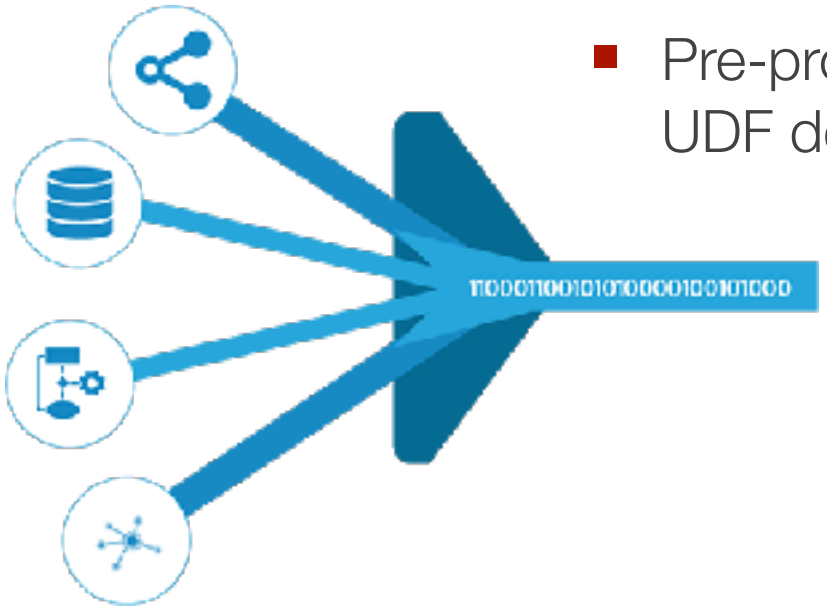
```
create feed TwitterFeed using TwitterAdaptor
("interval"=60);

create feed TwitterFeed using TwitterPushAdaptor;

create feed CNNFeed using CNNAdaptor
("topics"="politics, sports");
```

Pre-processing Collected Data

- Pre-processing arriving data;
  - E.g., adding attributes, sentiment analysis, feature extraction, etc.
- Pre-processing expressed as a UDF defined in **AQL** or **Java**.



Pre-processing Collected Data

```
create type RawTweet as open
{   tweetid: string,
    user: TwitterUserType,
    latitude: double?,
    longitude: double?,
    send-time: string,
    message-text: string,
    language: string
};
```

UDF

```
create type ProcessedTweet as open {
    tweetid: string,
    userid: string,
    send-time: string,
    message-text: string,
    language: string,
    location: point?,
    referred-topics: {{string}}
};
```

```
create feed TwitterFeed using
TwitterAdaptor
("query"="Obama", "interval"=60)
apply function addFeatures;
```

Feed Adaptor/UDF act as pluggable components

Challenge solved: Genericity/Extensibility

Fetch Once, Compute Many!

- Multiple applications may wish to consume the data ingested from a feed
  - **Primary Feed:** A feed that gets data from an external source.
  - **Secondary Feed:** A feed that derives its data from another feed.
- Pre-processing required by each application might overlap and could be done in an incremental fashion.

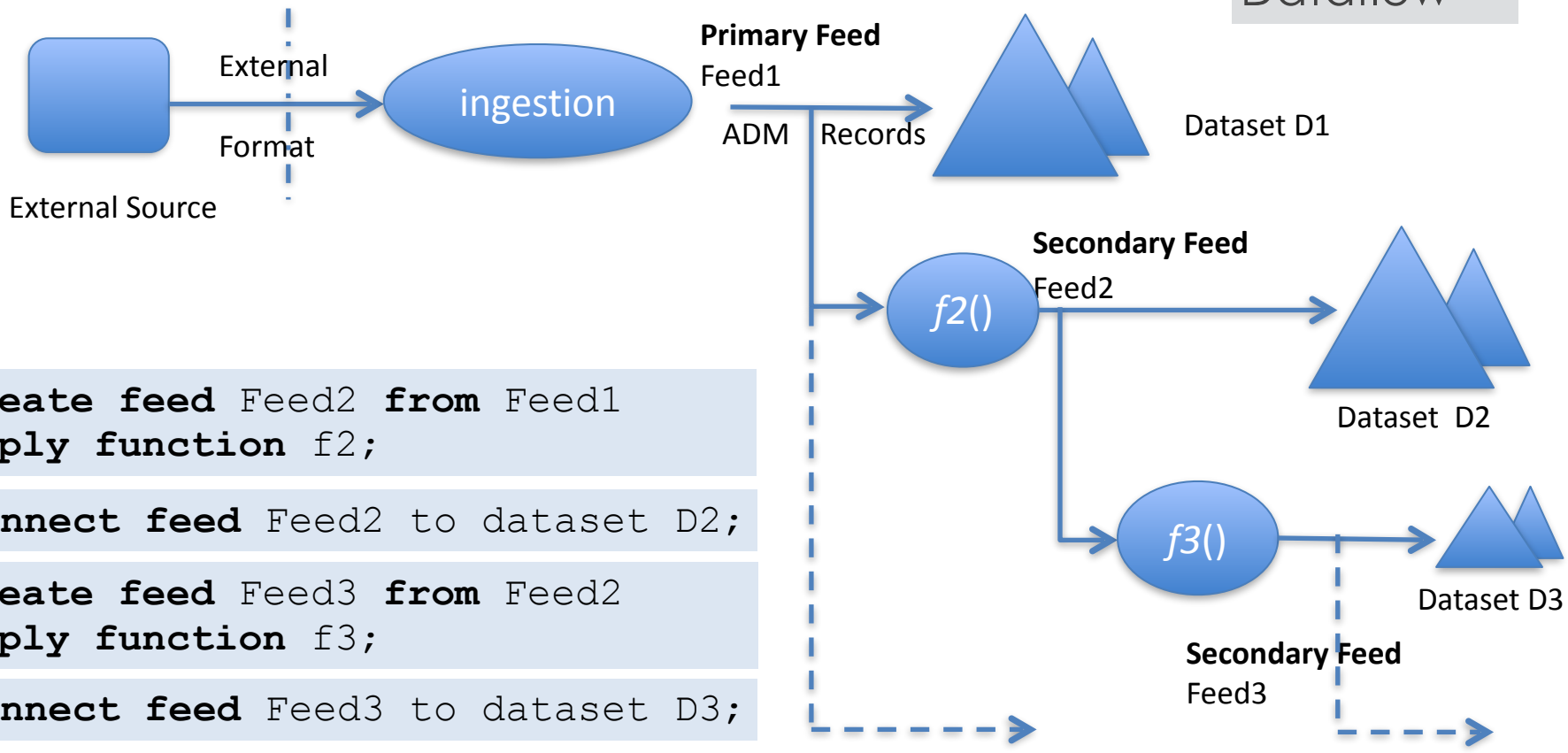
Fetch Once, Compute Many! (2)

- **Secondary feed**
  - can be persisted to a target dataset
  - can be used to derive other secondary feeds to form a Cascade Network

Challenge solved: Fetch Once, Compute Many

Building a Cascade Network of Feeds


```
create feed Feed1 using myAdaptor (…);
connect feed Feed1 to dataset D1;
```




```
create feed Feed2 from Feed1
apply function f2;
```

```
connect feed Feed2 to dataset D2;
```

```
create feed Feed3 from Feed2
apply function f3;
```

```
connect feed Feed3 to dataset D3;
```

Lifecycle of a Feed

- Subsequent to a connect feed statement, a feed is said to be in a connected state.
- Multiple feeds can simultaneously be connected to a single dataset.
- Connecting a secondary feed does not require the parent (or any ancestor) feed to be in a connected state.
- Additional feeds can be added to an existing hierarchy and connected to a dataset at any time.

```
disconnect feed TwitterFeed
from dataset Tweets;
```

Policies for Data Ingestion

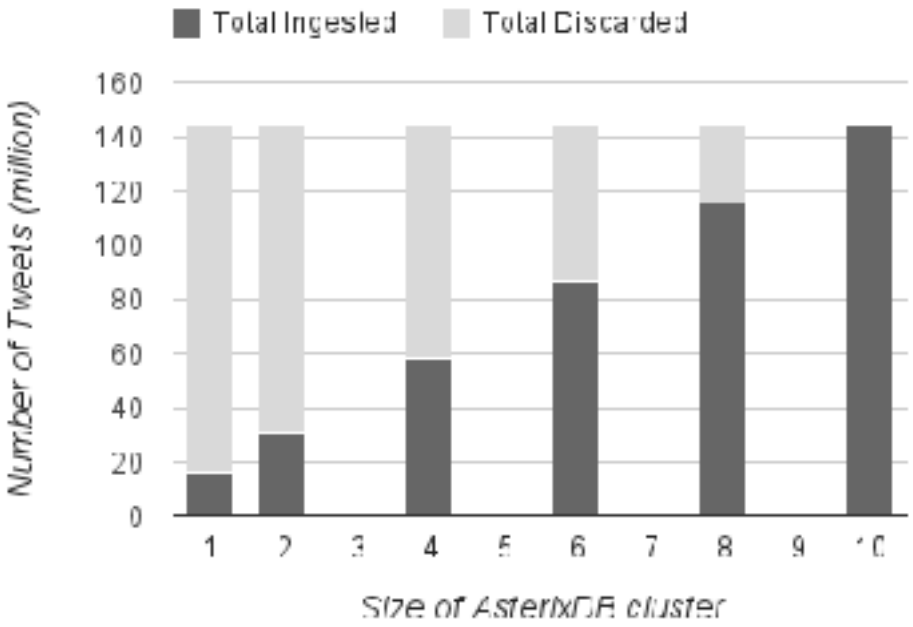
- Concurrent feeds (and queries) compete for resources (CPU cycles, network bandwidth, disk IO).
- Each feed is different
  - Rate of flow of data
  - Complexity involved in pre-processing
  - Constraints imposed by consuming application

Policies for Data Ingestion (2)

- A strategy for allocation of resources or handling of failures may work for one feed but may be a misfit or an over-fit for another!
- AsterixDB allows each data feed to have its own associated "**Ingestion Policy**"

Scalability

- 10 node cluster; each node 8gb of RAM
- Custom Tweet Generator (TweetGen): Output JSON tweets at a configurable rate (twps) and a pattern.
- Custom feed adaptor – TweetGenAdaptor
- UDF with 3 ms execution time.
- 6 instances of TweetGen for a duration of 20 minutes, each generating at 20k twps



Performance Metric: **Data Loss**

Use '**Discard**' policy to evaluate ability to successfully ingest data as a function of available

Data "Indigestion"

- Data arriving from external source may be subjected to expensive pre-processing (UDF) prior to persistence.
- Data arrival rate may exceed the rate at which it can be pre-processed!
- End user may form a custom policy; e.g., spill excess records to disk and throttle if the spillage crosses a configured threshold.

Data "Indigestion" (2)

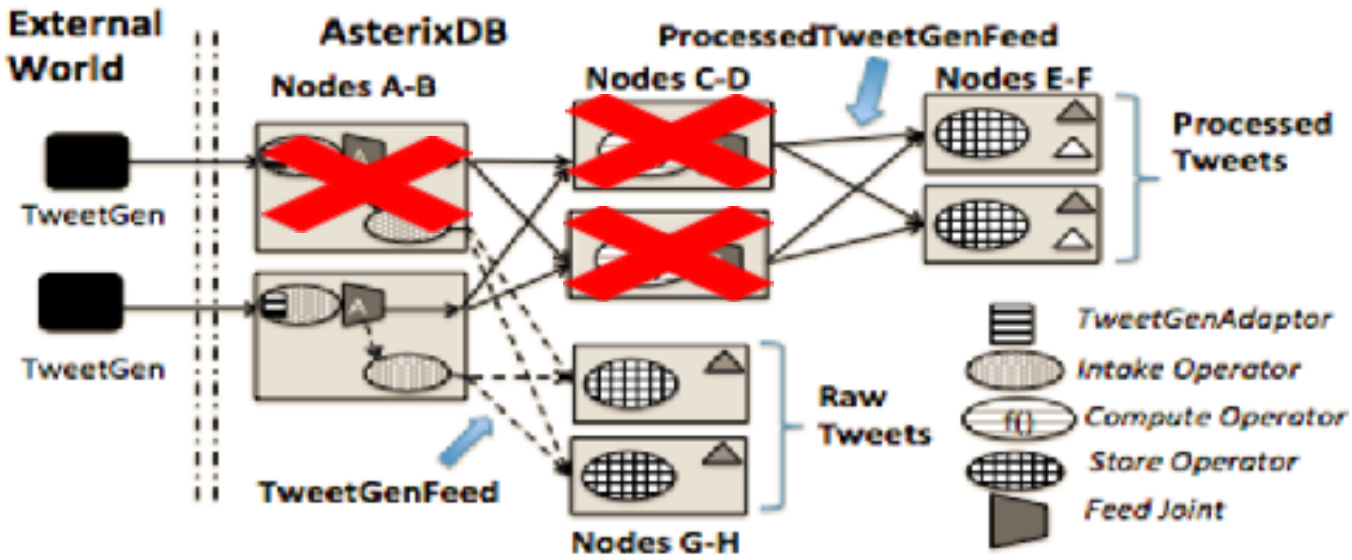
Policy	Approach to handling excess records
Basic	Buffer excess records in memory for deferred processing
Spill	Spill excess records to disk for deferred processing
Discard	Discard excess records altogether
Throttle	Randomly filter out records to regulate the rate of arrival
Elastic	Scale out/in to adapt to the rate of arrival

```
connect feed TwitterFeed to dataset Tweets
using policy Elastic;
```

Fault Tolerance

- System can recover from **soft and hard failures**
- Recovery from kinds of failures of a data ingestion data flow dictated by associated **ingestion policy**

Fault-Tolerance (contd.)



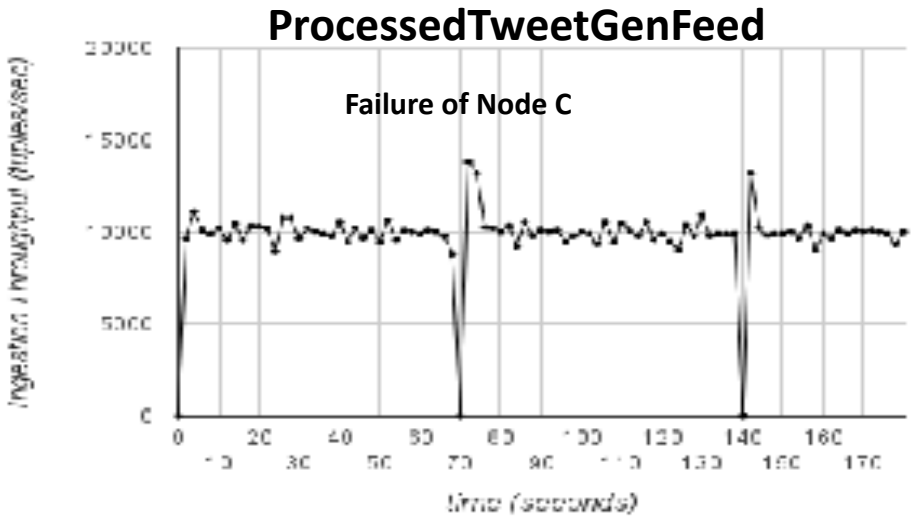
At t=70 seconds

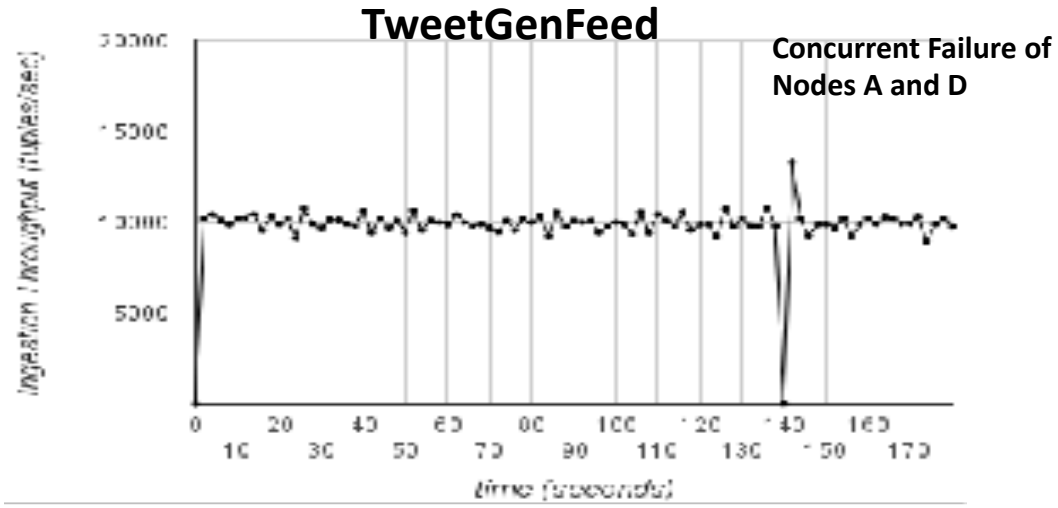
At t=140 seconds

To make things interesting…

```
connect feed ProcessedTweetGenFeed to dataset
ProcessedTweets
using policy FaultTolerant;
connect feed TweetGenFeed to dataset RawTweets
using policy FaultTolerant;
```

Fault-Tolerance (contd.)





- **Fault Isolation**
- **Recovery Time**

Summary

- Scalable, Fault-Tolerant, and Elastic data ingestion facility
- Plug-and-Play model to cater to wide variety of data sources.
- Custom built solution – Storm + MongoDB did not compare – user experience or performance characteristics (see paper).
- We can reduce the number of moving parts needed to handle Big Data.
- AsterixDB now an **Apache Incubator Project**.

Links

- Storm:
  - http://storm.apache.org/index.html
  - Tutorial: http://storm.apache.org/tutorial.html
- AsterixDB
  - https://asterixdb.apache.org
  - Documentation: https://ci.apache.org/projects/asterixdb/index.html