# Big Data Frameworks

1. MapReduce
2. Data streaming
3. Spark
4. Graph processing

# 1. MapReduce

- The first framework for big data analysis (2004)
  - Invented par Google
    - Written in C++
    - Proprietary, protected by “non offensive” patents
- Unstructured, schema-less data
  - Stored in GFS
- For very big clusters
  - Tens of thousand nodes
  - Automatic partitioning and parallelization
- Many variations
  - Hadoop (Apache), Amazon Elastic MapReduce, etc.

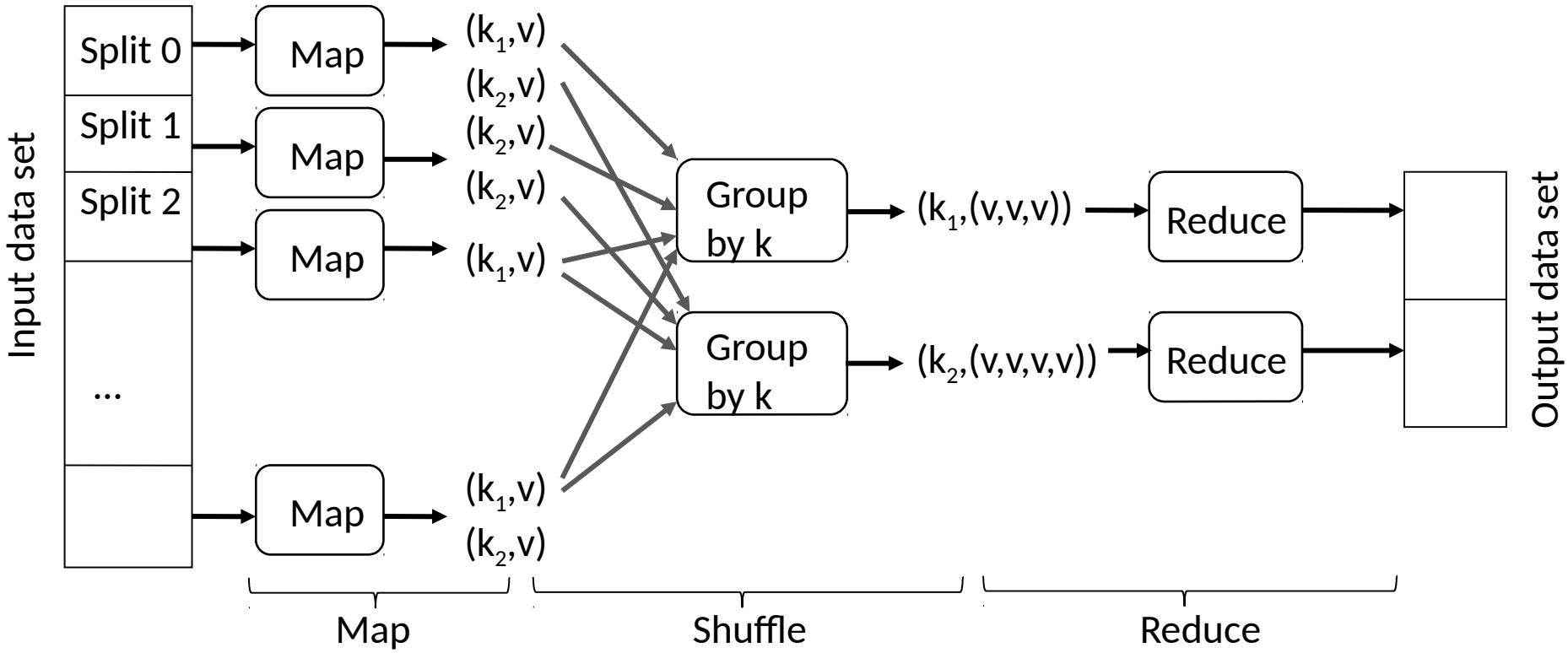
# MapReduce Typical Usages

- Counting the numbers of some words in a set of docs
- Distributed grep: text pattern matching
- Counting URL access frequencies in Web logs
- Computing a reverse Web-link graph
- Computing the term-vectors (summarizing the most important words) in a set of documents
- Computing an inverted index for a set of documents
- Distributed sorting

# MapReduce Model

- Data is of the form : (key, value) pairs
  - E.g. (doc-id, content), (word, count), etc.
- The programmer provides the code of two functions :
  1. Map (key, value) -> list(ikey, ivalue)
     - The same processing in parallel on partitionned data
     - Perfect parallelism
  2. Reduce (ikey, list(ivalue)) –> list(ikey, fvalue)
     - Aggregation of the data produced by Map
     - Less parallelism (remember Amdahl's law)
- Parallel processing of Map and Reduce
  - Data partitioning
  - Fault-tolerance
  - Scheduling of disk accesses
  - Monitoring

# MapReduce Processing




- Batch processing
  - Hence the term MapReduce jobs

# Exemple 1: size of a web server

- Given a large file containing metadata on the size of a collection of web pages
  - Rows of the form (server, page URL, page size, …)
- For each server, calculate the total size of the pages, i.e., the sum of the page sizes of all URLs of that server

# Example 1: pseudo-code

```
Map (key, value):
// key: file name; value: file rows
   for each line L (Server, Page url, Page size, ...) in value
        EmitIntermediate (Server, page size);

Reduce (key, values):
// key: a server name; values: a list of page sizes
   result = 0;
   for each size s in values:
        result += s;
   Emit (key, result);
```

# Example 2: group by

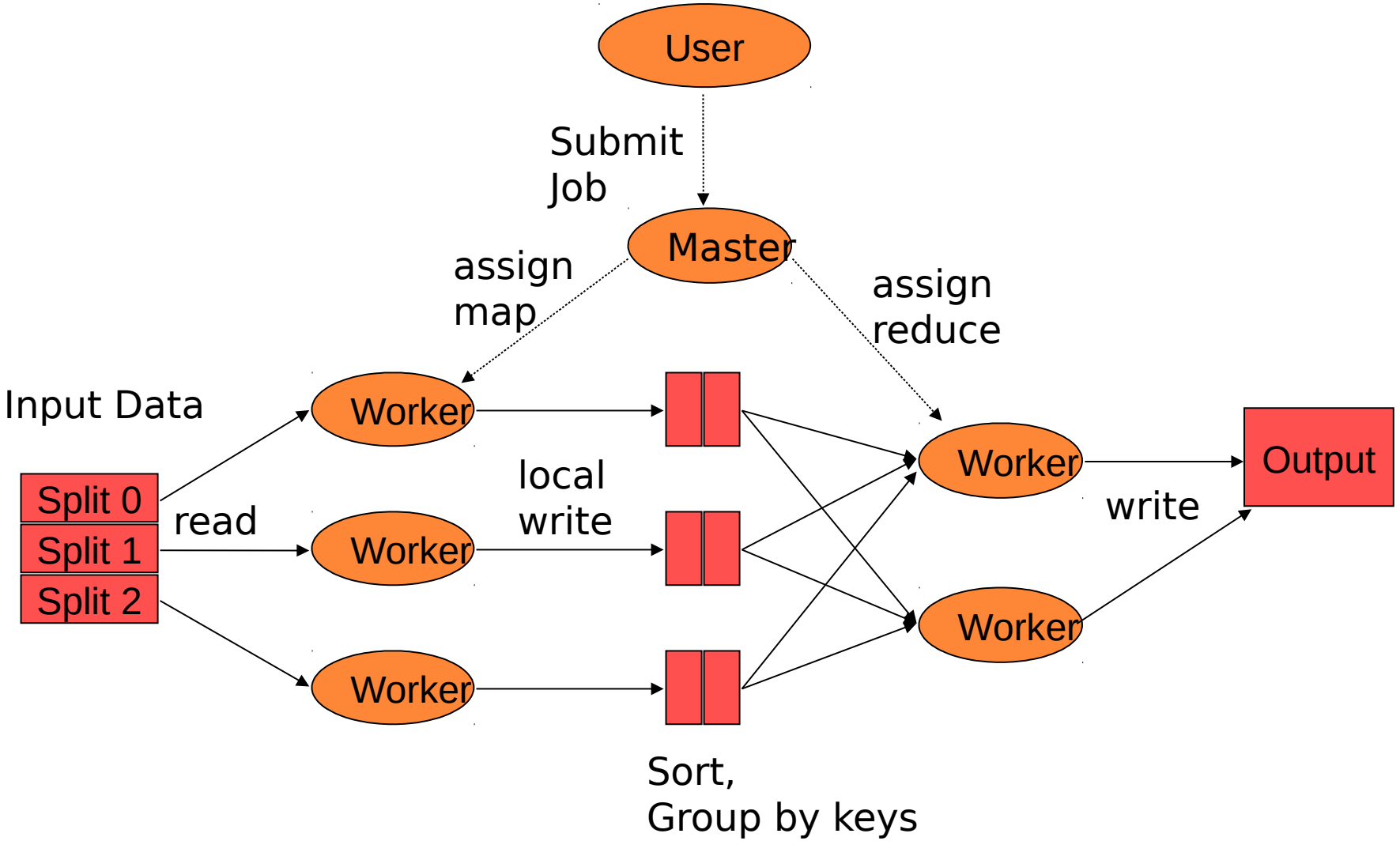
EMP (ENAME, TITLE, CITY)
Query: for each city, return the number of employees whose name is "Smith"

```
SELECT CITY, COUNT(*)
FROM EMP
WHERE  ENAME LIKE "\%Martin"
GROUP BY CITY
```

```
Map (Input (TID,emp), Output: (CITY,1))
// TID: tuple identifier, emp: une ligne de EMP
        if emp.ENAME like "%Martin"
            EmitIntermediate (CITY,1)
```

```
Reduce (Input (CITY,list(1)), Output: (CITY,SUM(list(1)))
   Emit (CITY,SUM(1*))
```

# Master-Worker Model



# Task Scheduling

- Dynamic approach
  - Status of a task: inactive, active, completed
  - Inactive tasks are enabled as workers become available
    - They are assigned to the workers who are closest to the input data
      - Ex. local disk or even rack, to reduce transfers between nodes
- When a task finishes, it sends the addresses and sizes of the intermediate data to the master
- When all Map tasks are completed, Reduce tasks start

# Fault-tolerance

- Fault-tolerance is fine-grain and well suited for large jobs
- Input and output data are stored in GFS
  - Already provides high fault-tolerance
- All intermediate data is written to disk
  - Helps checkpointing Map operations, and thus provides tolerance from soft failures
- If one Map node or Reduce node fails during execution (hard failure)
  - The tasks are made eligible by the master for scheduling onto other nodes
  - It may also be necessary to re-execute completed Map tasks, since the input data on the failed node disk is inaccessible

# Bilan MapReduce

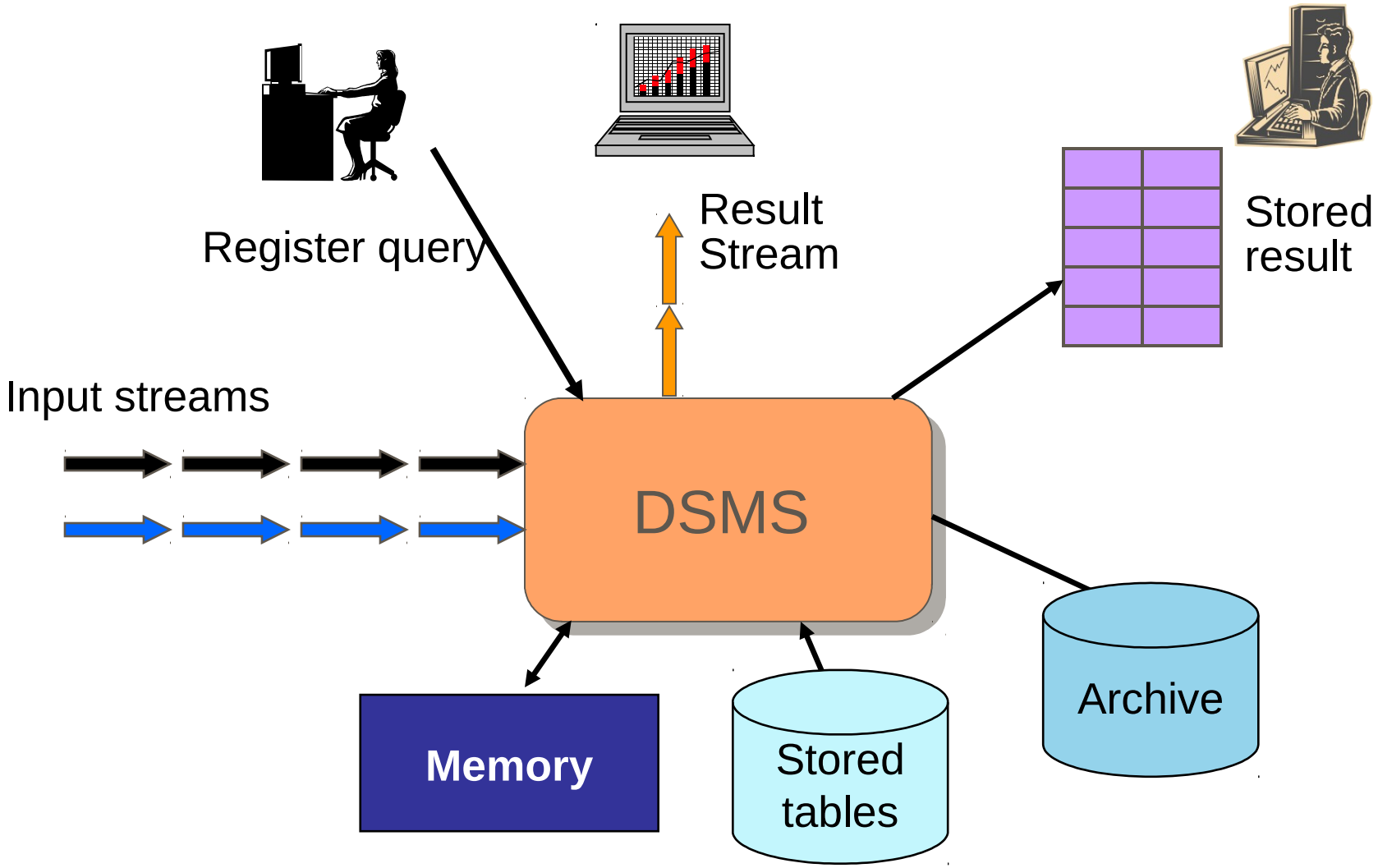
- Advantages
  - Simple for the programmer
  - Parallelization, fault tolerance, scalability
  - Well suited for unstructured data
- A very large community of developers
  - Strong early adoption
  - Google, Facebook, Amazon, Oracle, IBM, Microsoft, etc.
- Limitations
  - Low performance: disk-oriented
  - Batch applications only
- Abandonment in favor of Spark or others
  - Google Cloud Dataflow, Flink

# 2. Data Streaming

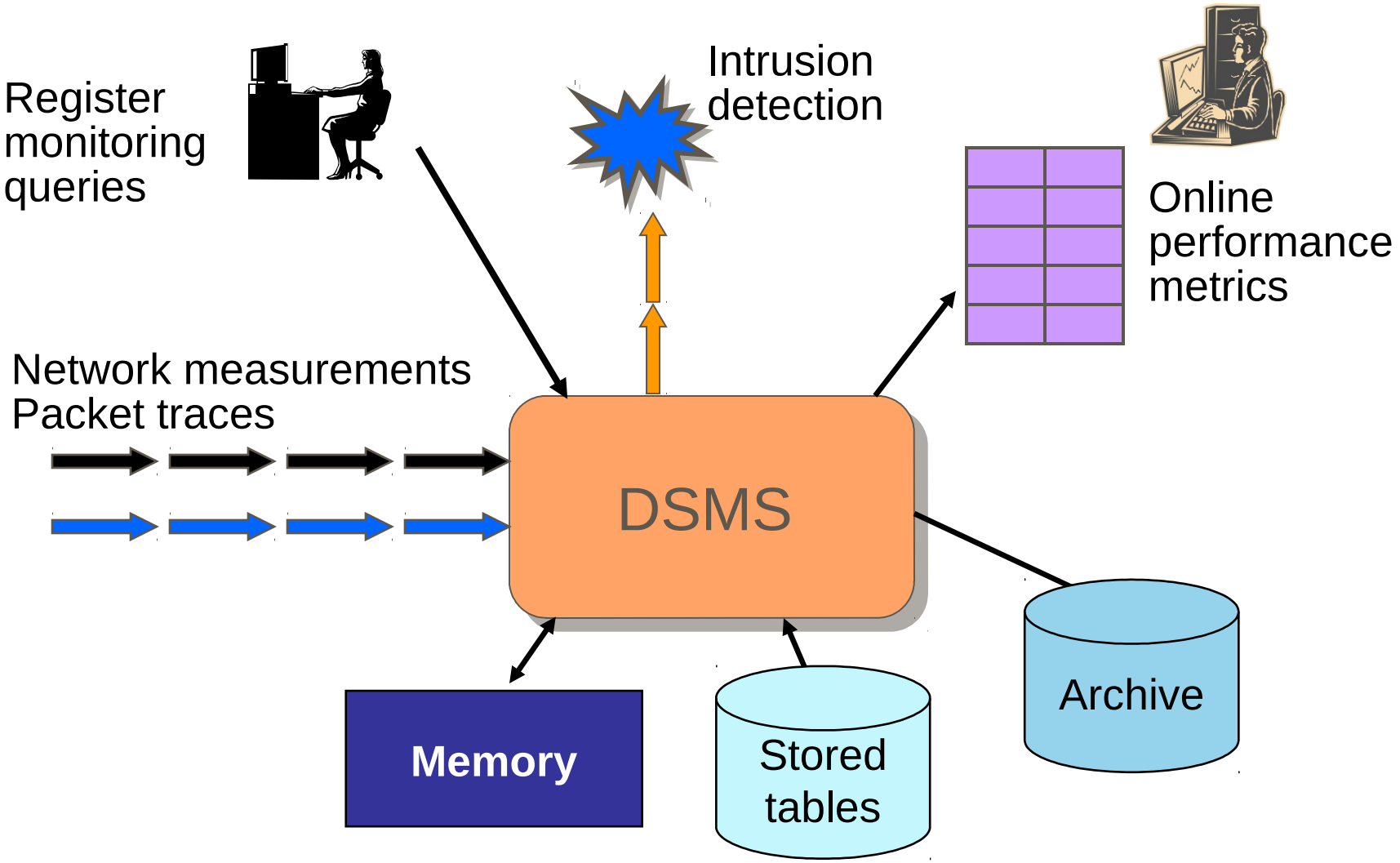
- Objective: extract outstanding events in continuous data streams
  - Unlimited, fast, and time-varying data
- Applications
  - Networks
    - Intrusion detection
  - Sensor networks
    - RFID, power generation adjustment
  - Finance
    - Trend analysis, fraud detection, risk analysis
  - Telecoms
    - Call analysis, fraud detection
  - Web
  - Log analysis, website statistics, adaptive targeting

- DSMS = Data Stream Management System

# DSMS Architecture



# Example: Network Monitoring

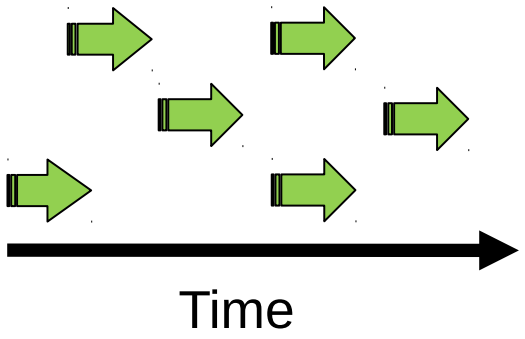


# Relation with CEP

- Complex Event Processing
  - Origin: simulation of discrete events, triggers (Event-Condition-Action rules) in DBMS, message-oriented middleware (MOM)
  - Objective: identify and respond to outstanding events (e.g. opportunities, threats, etc.) as quickly as possible
  - Technologies to filter, correlate, rapidly aggregate events and produce reactions
- For event-driven IS
  - SOA, Event-Driven Business Process Management, Business Application Monitoring (BAM)

# Event Management

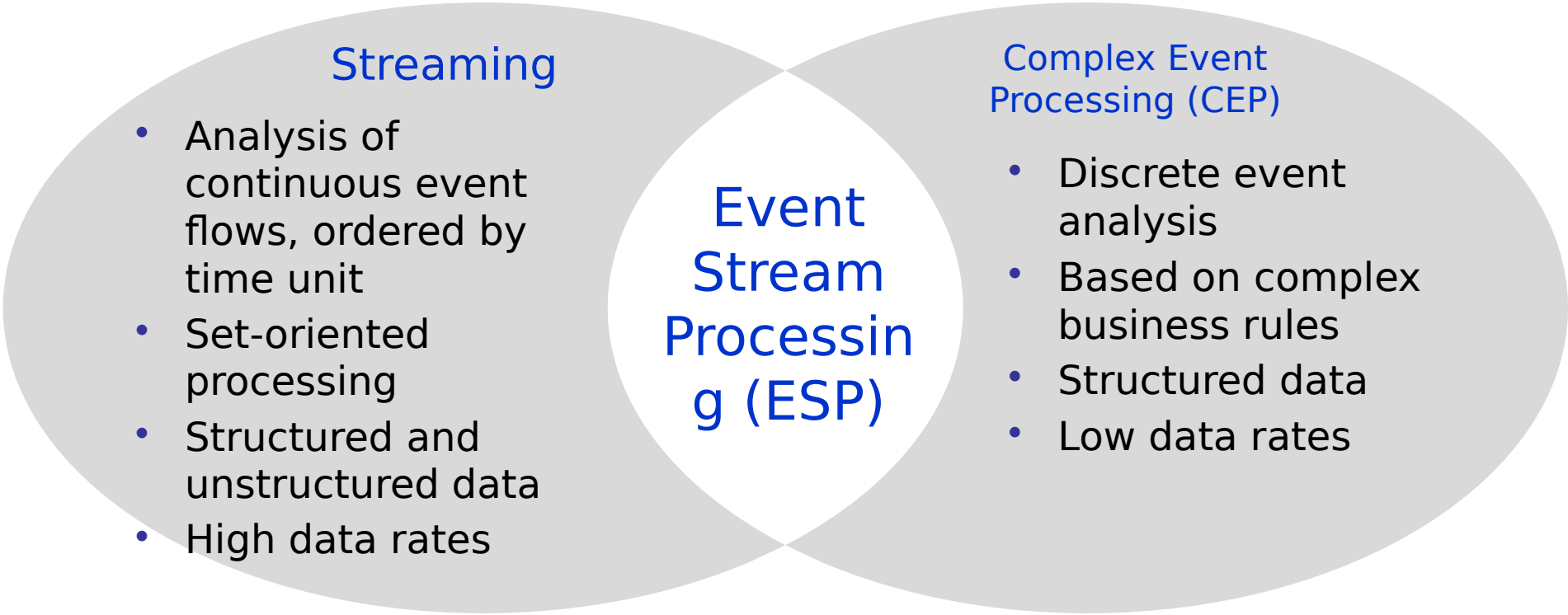
- Simple event
  - Corresponds to a real world event
    - A temperature reading
    - An order, a delivery, an invoice
    - Peter said to Mary, who said to..., who said to John
- Complex event
  - An interesting combination of simple or complex events
    - Temperature increase
  - An abnormal purchase (e.g. n orders of the same drug)
  - Jean learned indirectly through Pierre
- Relations between events
  - Time: *a* happened before *b*
  - Aggregation: *a* is composed of $a_1$, $a_2$ ... ,$a_n$
  - Causality: *a* must happen so *b* happens



# How are Events Created

- To create events that indicate the current activities, in two steps
  - Observation: access and observation of activities, in a non-intrusive way, without changing their behaviour
  - Adaptation: observations must be transformed into event objects to be processed
    - Typically with connectors
- The sources can be diverse
  - IS: components, databases, messages, etc.
  - Instrumentation: heartbeats, sensors, applications, etc.
  - CEP: events created by event processing

# Streaming versus CEP

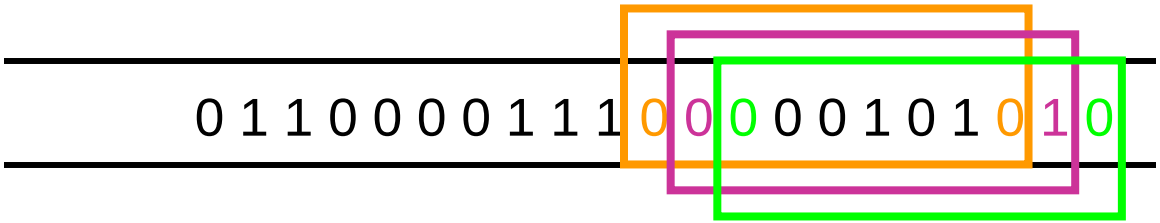



NB: the term CEP is often used to include both ESP and streams (i.e. CEP = DSMS)

# Approximation

- Why approximate?
  - Memory is not infinite
  - Streams cannot be processed fast enough to sustain requests
  - Queries can access or aggregate old data
- Solutions
  - Sliding window on data
  - Data summary of the elements of old streams
    - Like materialized views, but base data is unavailable
      - E.g. sampling, histograms

# Sliding Window Model




- Why?
  - Approximation technique for limited memory
  - Natural for applications (focuses on recent data)
  - Well defined
    - In terms of time, space, number of elements, etc.
- Requirements
  - SQL-like set-oriented language, query optimization
  - Timestamps (e.g. rowtime)
    - Explicit: injected by the source
    - Implicit: introduced by the DSMS

# SQL for Streaming

- No standard, several variants
  - Continuous Computation Language (Sybase)
  - StreamSQL (Streambase)
  - Continous query language (Stanford U.)
- Capabilities
  - Stream definition, view
    - Automatically generated Rowtime
  - Requests on sliding windows
    - Filtering, merging and aggregation of streams
    - Streams joining, joining with stored table
  - Extensibility of operators and functions
    - Java, C++, etc.

# Stream Aggregation

- Two streams

  Orders (orderID, customer, cost)
  Fulfillments (orderID, clerk)

- Total cost of orders processed by the employee "Sue" for the customer "Joe" during the day

```
SELECT  Sum(O.cost)
FROM Orders O, Fulfillments F [Range 1 Day]
WHERE O.orderID = F.orderID AND F.clerk = “Sue”
    AND O.customer = “Joe”
```

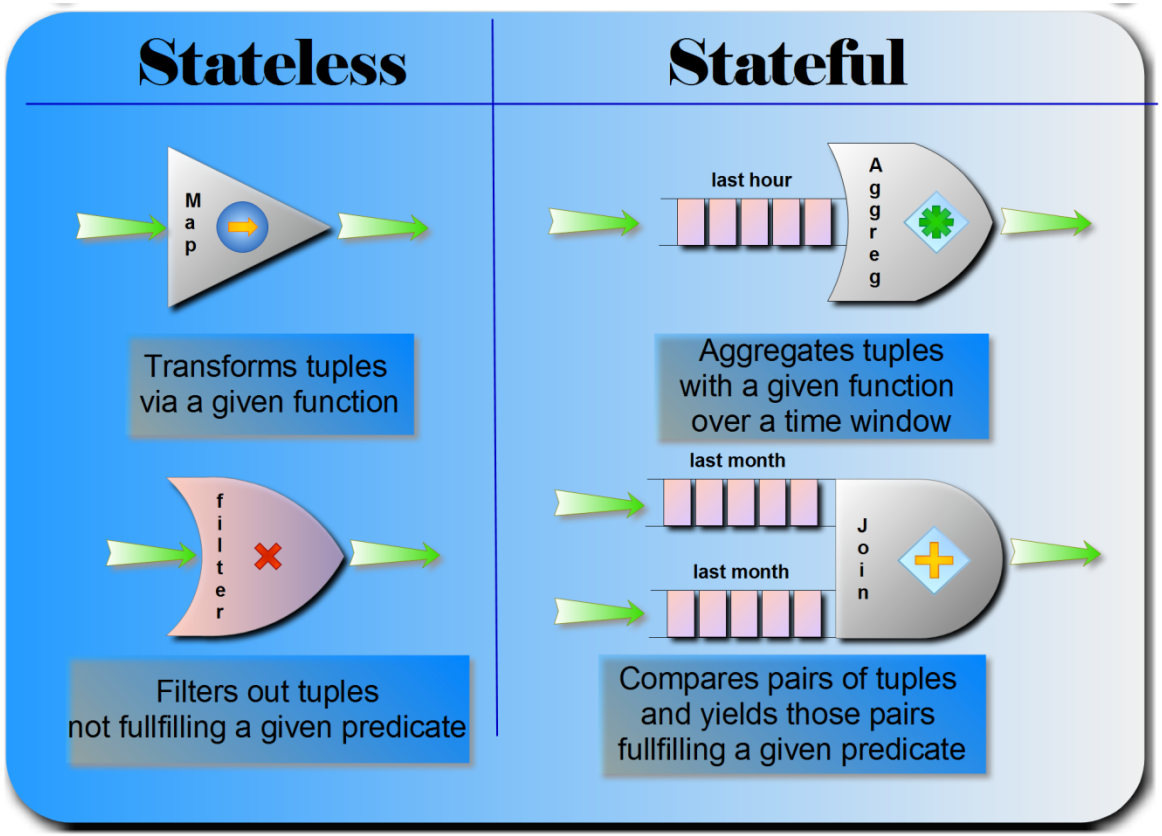
# Joining a Stream with a Table

- Let the Orders stream and the Shipments table, give all Orders delivered to New York within the last hour

```
SELECT STREAM *
FROM Orders OVER slw
    JOIN Shipments ON orders.id =
shipments.orderid
WHERE Shipments.city = 'New York'
WINDOW slw
    AS (RANGE INTERVAL '1' HOUR PRECEDING)
```

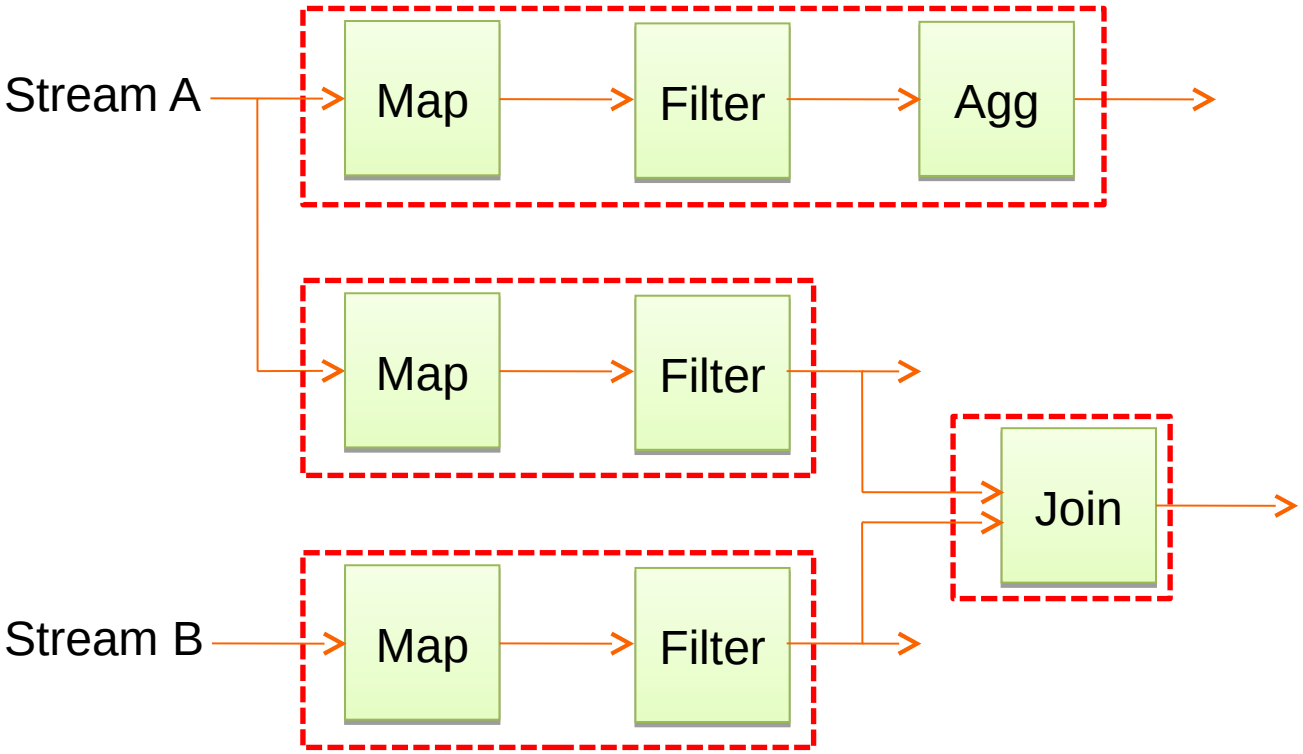
# Stream Processing

- Optimization and parallelization of continuous queries
  - Stateless operators: non-blocking
  - Operators that need state: blocking



# Parallelism

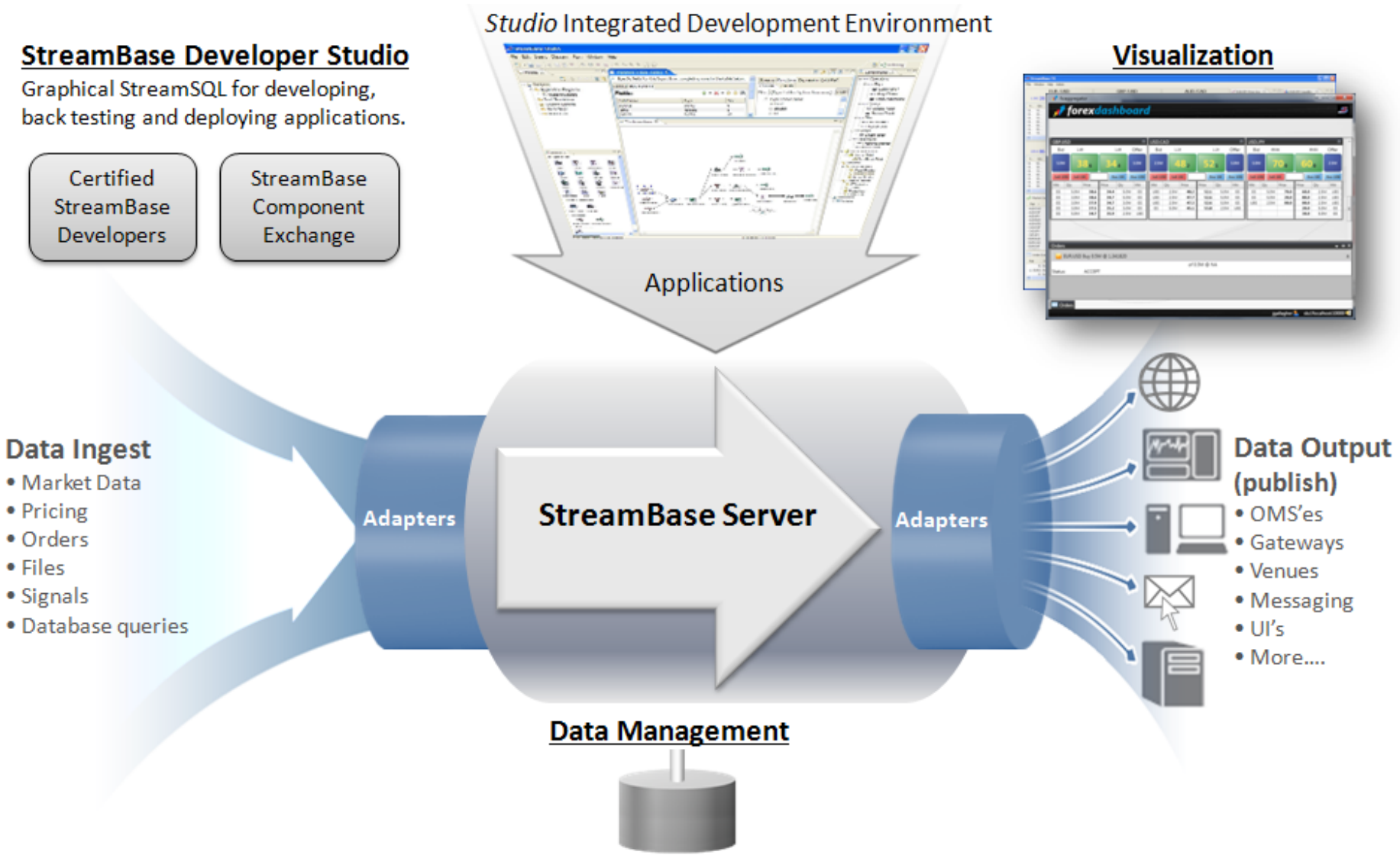
- Different types of parallelism
  - Pipeline, independent, hybrid



# Main DSMS

- RDBMS vendors
  - IBM, Microsoft, Oracle, SAP Sybase
  - Typically within an SOA offer
- New comers
  - Products: Tibco Streambase, Cisco Parstream
  - Open source: Storm, Kafka, Samza, Spark, Flink
  - Cloud service: Google Cloud Dataflow

# Tibco Streambase



# Apache Kafka (origin LinkedIn)

- Distributed platform widely used for large-scale streaming
  - PubSub-type message middleware
  - Storage of streams in clusters
    - Intra-cluster and inter-cluster replication (between data centers) for high availability
    - Partitioning a stream into topics and partitions in a cluster
    - Stream processing
      - Stream API for stream analysis, with Spark, Flink, Samza
- Interface with many systems and applications
  - Connector API

# Some Other DSMS

| Vendor | Product | Comment |
|---|---|---|
| Apache | Storm | Java framework, with continuous queries and parallel programming. Old. |
| Apache | Kafka | Origin: LinkedIn. PubSub parallel middleware, with Streaming API. |
| Apache | Samza | Origin: LinkedIn. Framework based on YARN and Kafka. Java API. |
| Google | Cloud Dataflow | Replacement of MapReduce for interactive data analysis. |
| IBM | Streaming Analytics | Stream analysis platform. Language with stream operators. |
| Microsoft | StreamInsight | Integrated within .NET framework. Use the Linq language. |
| Oracle | Oracle Event Processing | Middleware fro streaming apps. Support of Oracle CQL. |
| Cisco | Parstream | Column-store DSMS, SQL stream. |
| SAP | Sybase ESP | Support of Continuous Computation Language. Integration with SAP HANA and Sybase IQ. |

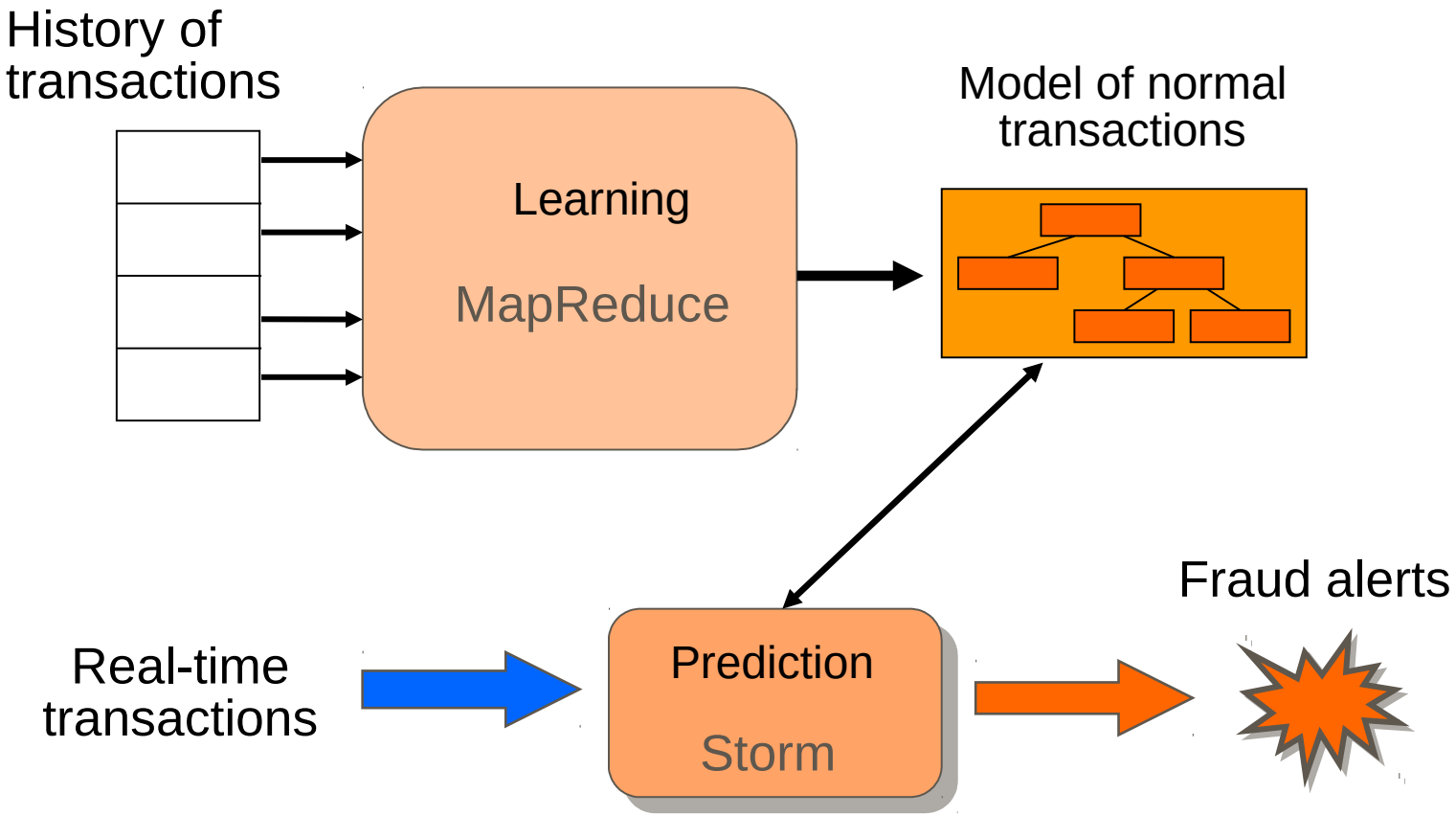
# Case Study: fraud detection

- Context: credit card transactions
- Objective: Real-time fraud detection
- Problem with traditional solutions
  - Applies to single transactions
    - E.g. Authentication, filtering rules (region, etc.)
  - Cannot detect suspicious behaviour
    - E.g. a large number of purchases in a very short time
    - E.g. a purchase that is out of the ordinary
- Approach
  - Combination of *sequence mining* and data streaming

# Solution

- Sequence mining
  - Detection of *outliers* (suspect transactions) in a data sequence (the transactions of the same card)
    - Need to code transactions
      - E.g. amount (low, normal, high), time since last transaction (short, normal, long), etc.
  - Two steps
    - Learning, from past transactions, of a model
    - Real-time detection of *outliers* on transaction streams
      - Outlier = "abnormal" transaction in relation to the model

# The Beymani Project



# 3. Spark

- Apache software, from UC Berkeley
  - Compatible with Hadoop (HDFS)
- Extension of MapReduce for two classes of analytics
  - Iterative processing (machine learning, graphs)
  - Interactive data mining (R, Excel, Python)
- Major performance improvement (up to 100*)
  - In-memory processing
  - Optimization of the task graph
- Major usability improvement
  - APIs for Java, Python, R and Scala (functional extension of Java)
  - Interactive use from a Scala interpreter
- Major adoption from industry
  - Databricks: a successful startup from UC Berkeley
  - Will replace MapReduce

# Spark Model

- Concept : Resilient Distributed Datasets (RDDs)
  - Collections of objects distributed in a cluster
  - Built from parallel transformations (map, filter, etc.)
  - Can reside in memory for efficient reuse
  - Preserve the properties of MapReduce
    - Fault-tolérance, data locality, scalability
- An RDD has its provenance information
  - How it is derived from other RDDs
- The user can control
  - Data persistence (disk or RAM memory)
  - Partitioning (hashing, range, [<k, v>])
- Operators: transformations and actions

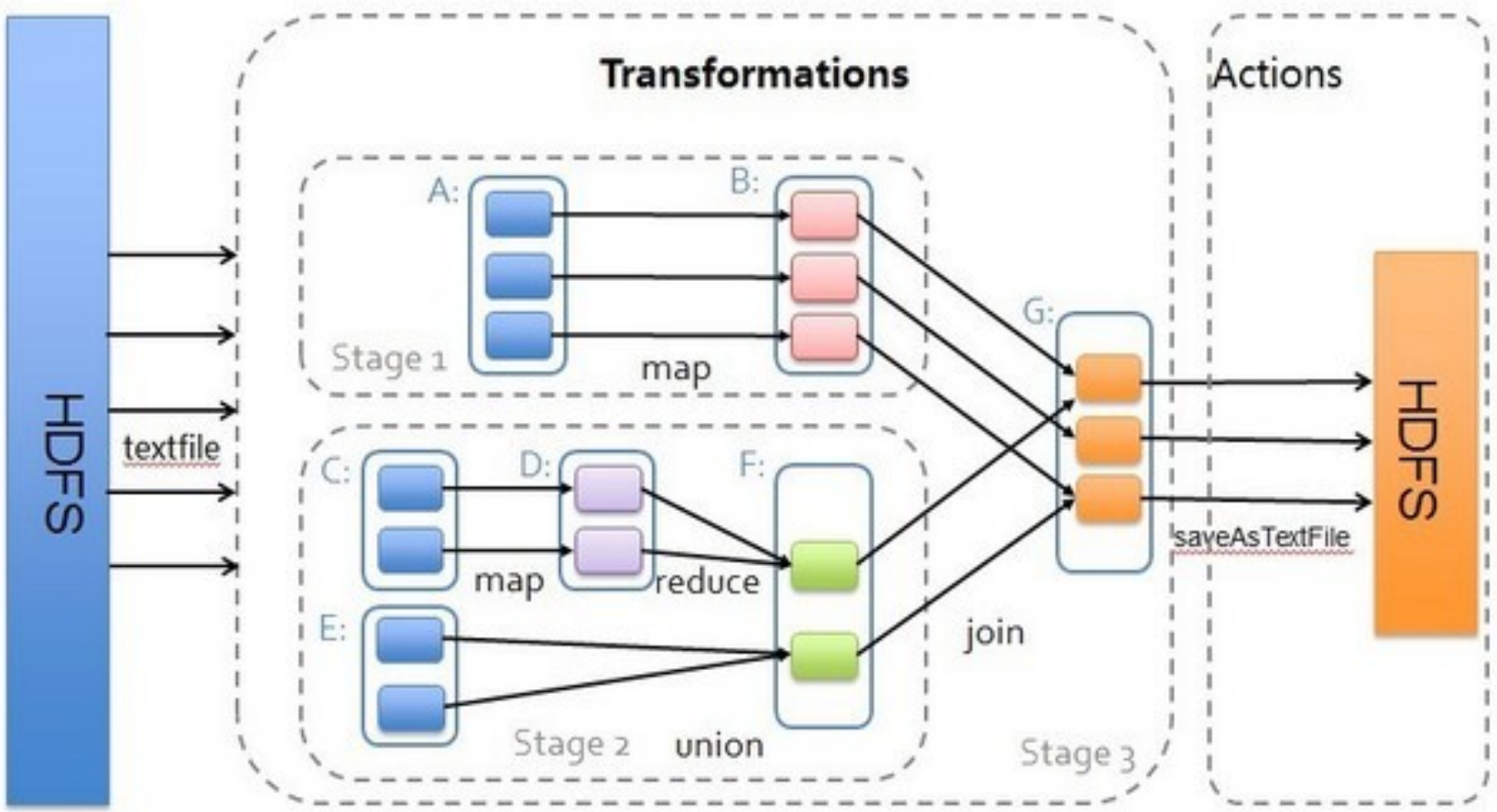
# Transformations

- Produce an RDD from an RDD
- map (f)
  - Produces an RDD by applying the function *f* to each element of the source RDD
- filter (f)
  - Produces an RDD by selecting the elements of the source RDD for which the function *f* yields true
- union (RDD)
  - Produces an RDD that contains the union of elements of the source RDD and the input RDD
- join (RDD, f)
  - Produces an RDD that contains the join based on *f* of the elements of the source RDD and the input RDD
- …

# Actions

- Return values to the program, from a source RDD
- reduce (f)
  - Aggregates the elements of the RDD with the function *f*
- collect ()
  - Returns the elements of the RDD as an array
- count()
  - Returns the number of elements in the RDD
- first()
  - Returns the first element of the RDD
- ...

# Transformations and Actions



# Memory Management

- 3 options for storing RDDs
  1. In memory, as Java objects
     - The fastest, since the JVM can access the RDD natively
  2. In memory, as serialized data
     - More efficient representation than Java object graphs when memory is limited
  3. Disk
     - For RDDs that do not fit in memory

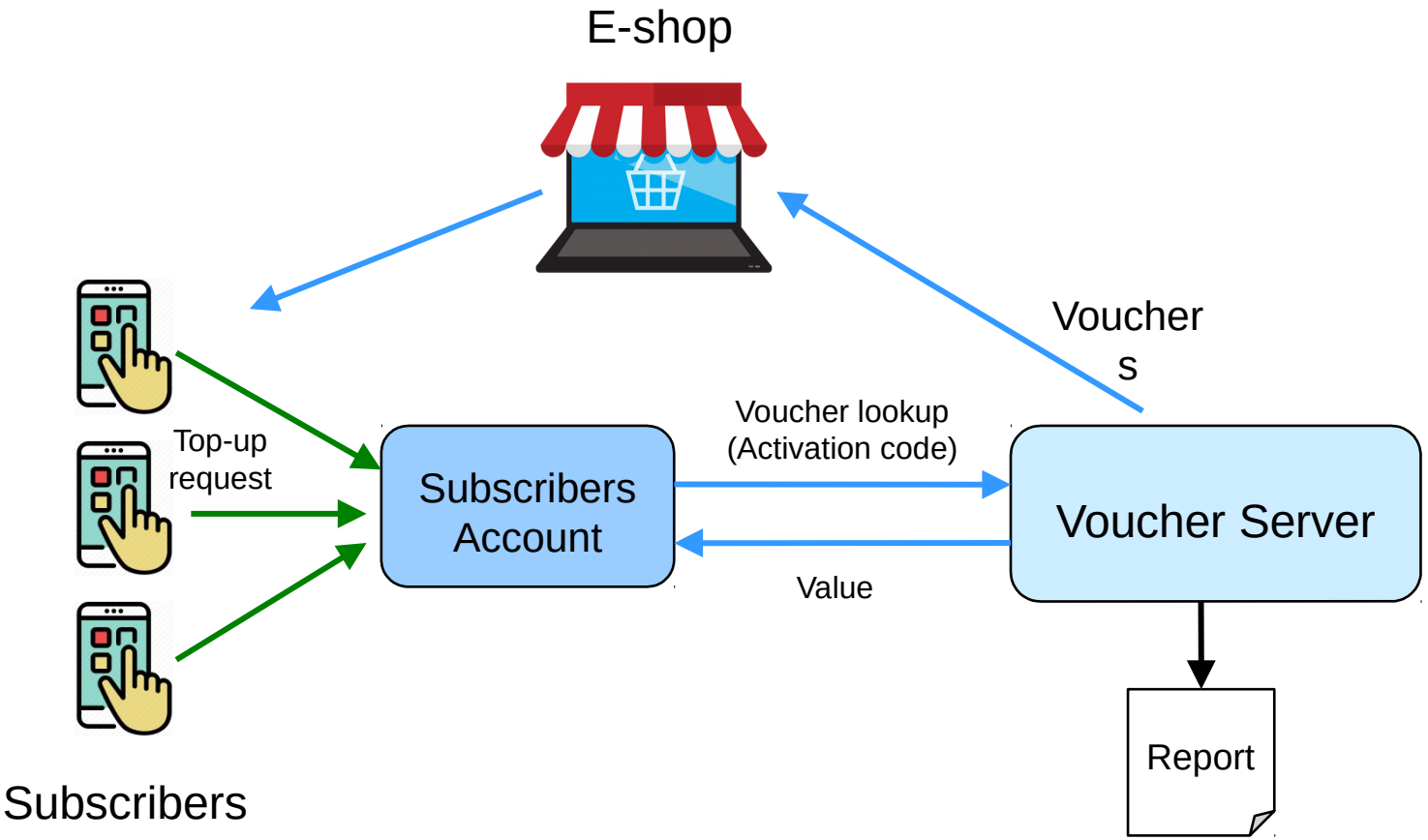
# Spark Tools

- Spark SQL
  - SchemaRDD = RDD with an associated schema
  - Manipulation with Scala, Java and Python
  - Manipulation with SQL with a JDBC driver
- Spark Streaming
  - To analyze data streams, processed in buckets (mini-batches)
- Mlib
  - Machine learning framework, with a library of classic algorithms
- GraphX
  - Graph processing framework, with an API to express graph manipulationl

# Case Study: Ericsson (Telecom)

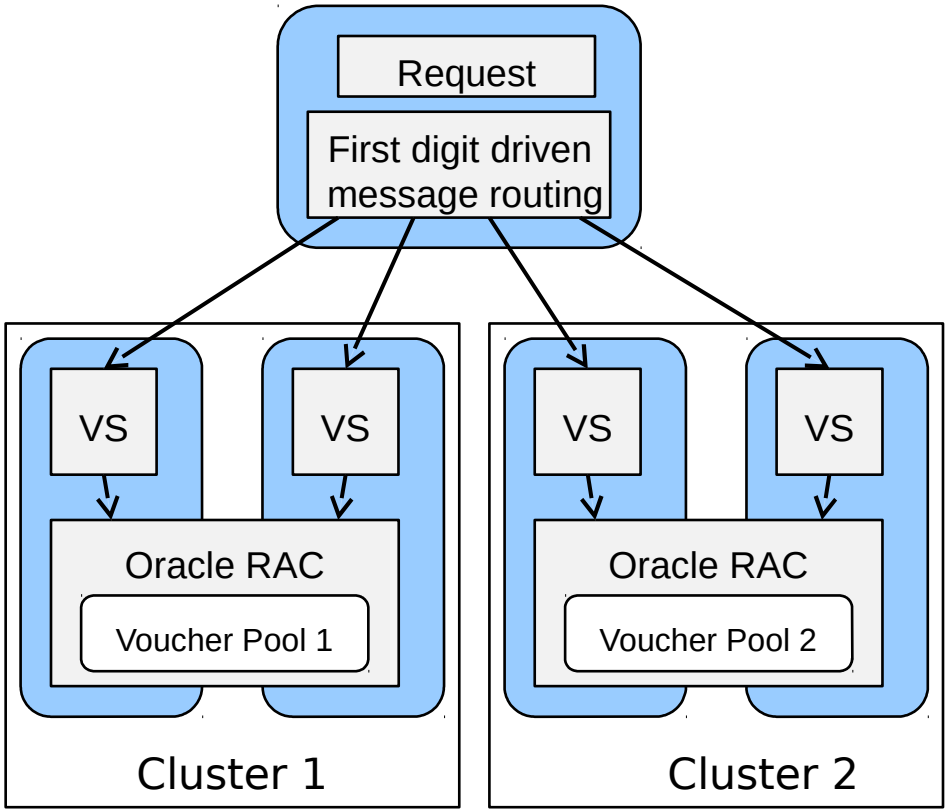
- Context : invoicing of prepaid phone cards
- Voucher server
  - Generates activation codes
    - Corresponds to a currency and a value
  - Handles top-up requests of pre-paid subscribers
    - Interprets the value of an activation code
    - Adds the corresponding value to the balance of the subscriber
  - Generate reports
- Scalability problem
  - Billions of vouchers

# Voucher Server



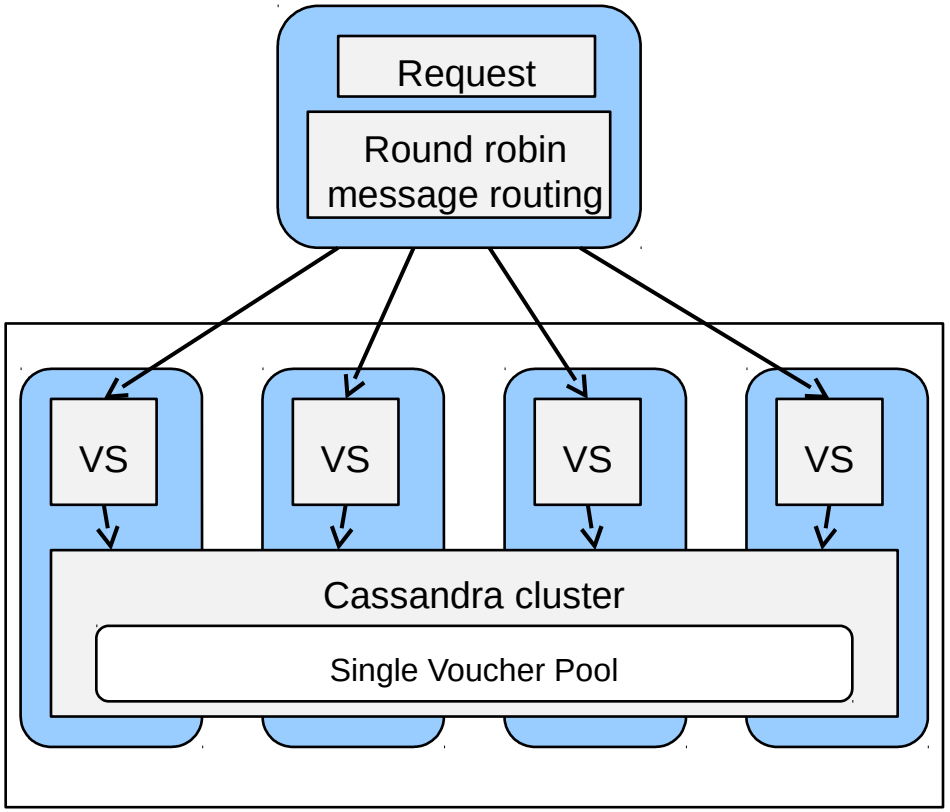
# Solution with Oracle

- Oracle RAC
- Scalability problem
  - Limited capacity
    - 1 cluster: 300 million vouchers
  - Hotspots
    - Access conflicts to shared disk



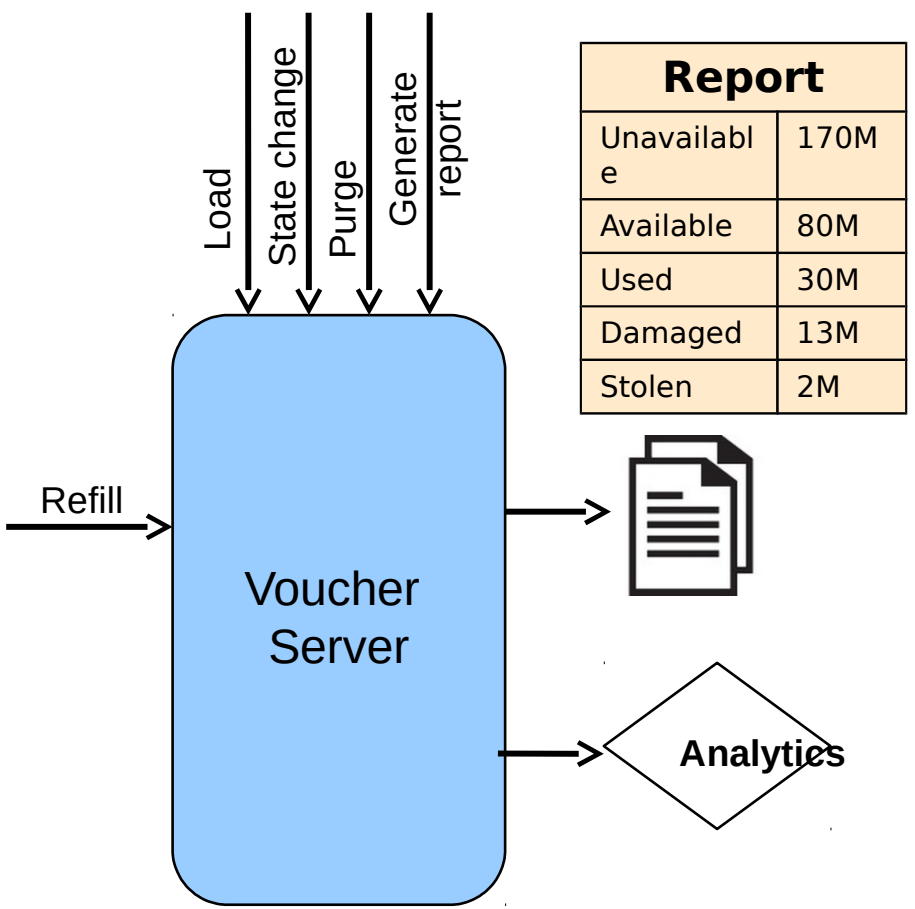
# Solution with Cassandra

- Advantages
  - Scalability
  - Performance
  - Nohotspots
- Weakness
  - Poor report generation



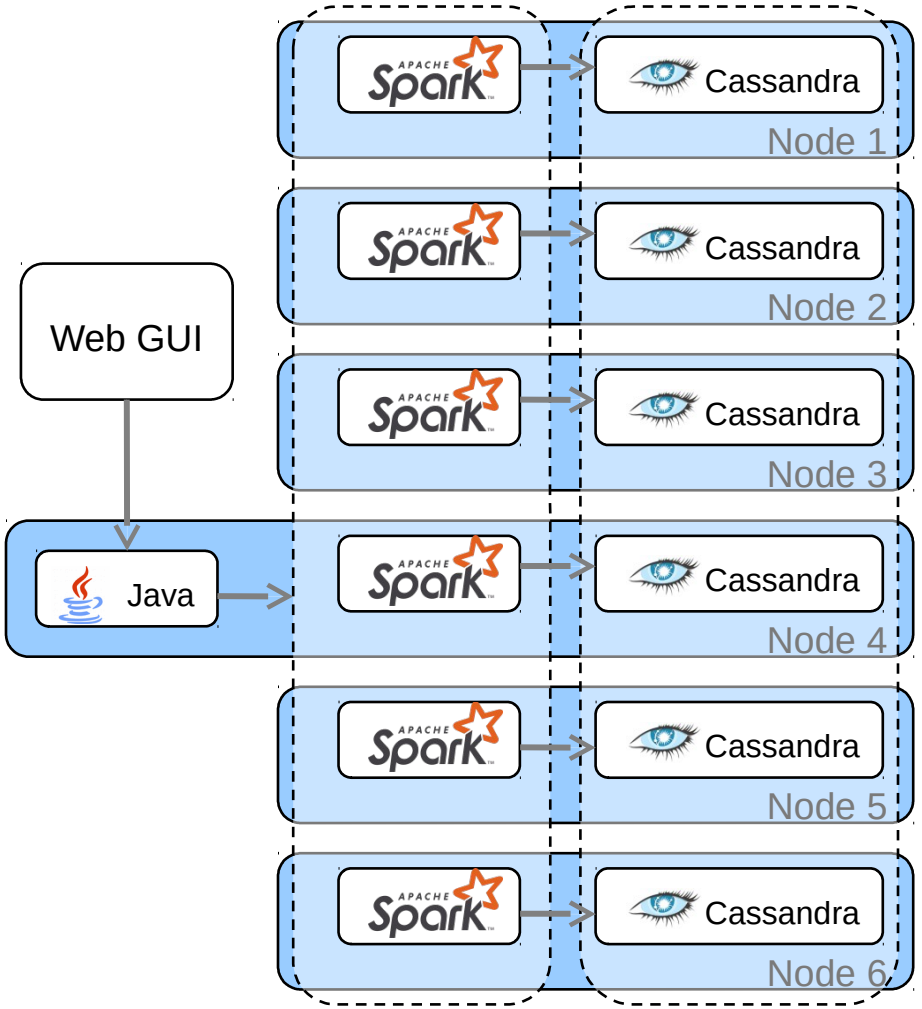
# Report Generation

- Voucher state distribution
  - Count vouchers in each state
  - Voucher reconcillation
  - Inventory
- Voucher data analysis
  - Fraud detection
  - Market prediction
- Challenges
  - Cassandra does not provide functions to count fields
  - Report generation requires full-table scan
    - 5-6 hours

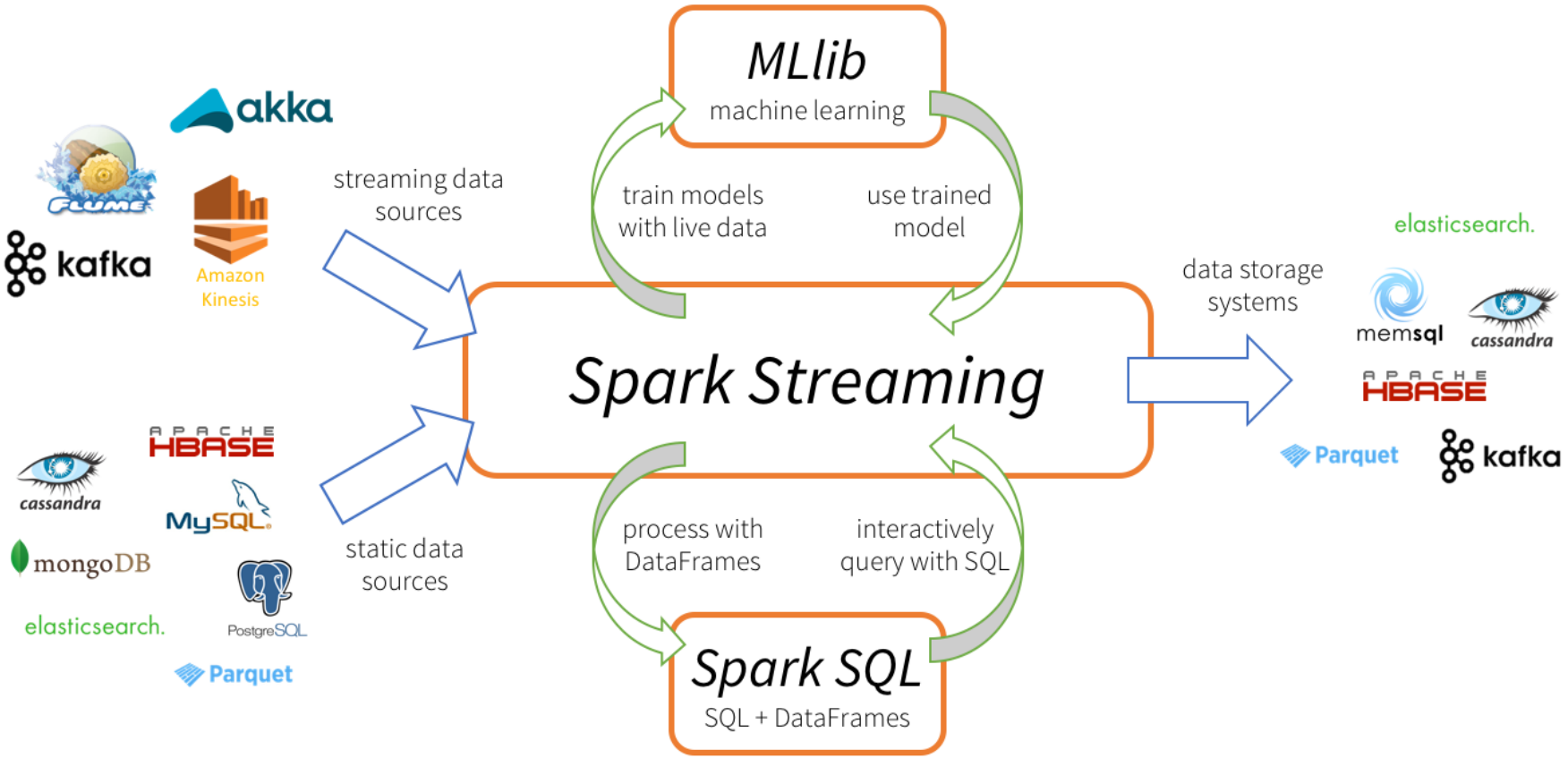


# Report Generation with Spark

- Parallel approach
  - One multi-task Spark instance per Cassandra node
  - Each Spark instance deals with one Cassandra partition and produces a partial report
- Advantages
  - Small network latency
  - Performance



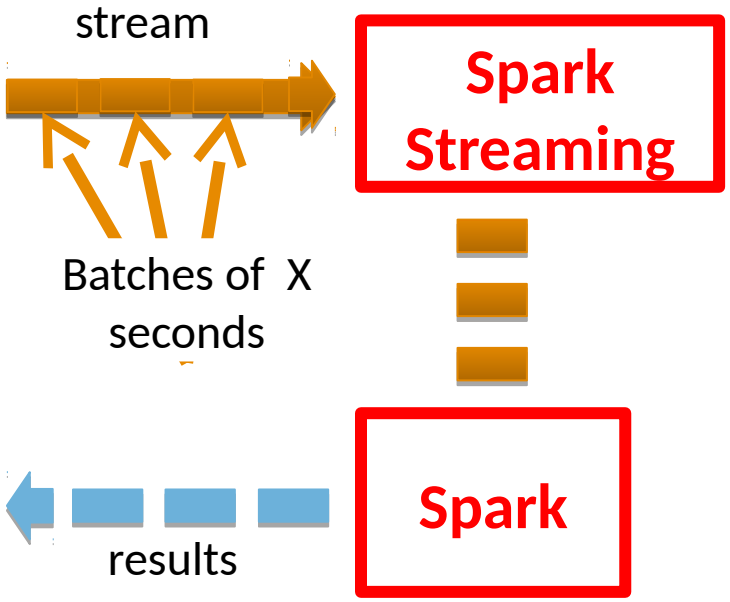
# Spark Streaming



# Discretization of Stream Processing

Decomposition of the streaming into a sequence of very small jobs (*mini-batches*)

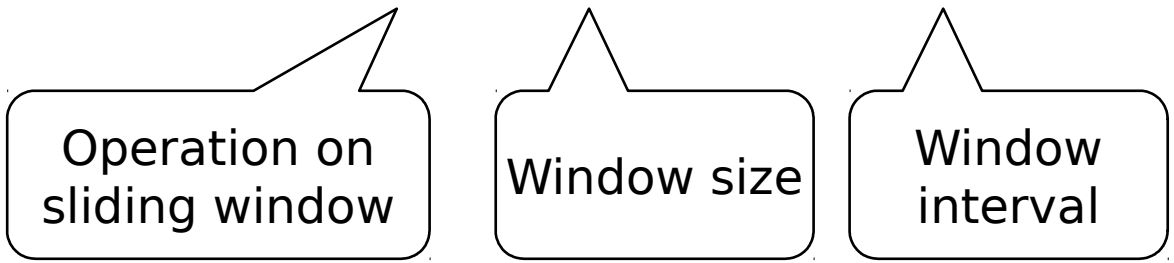
- Decomposition into batches of X seconds (e.g. X=0,5)
- Each batch is processed as an RDD
- The results of RDD operations are returned as batches



# Example

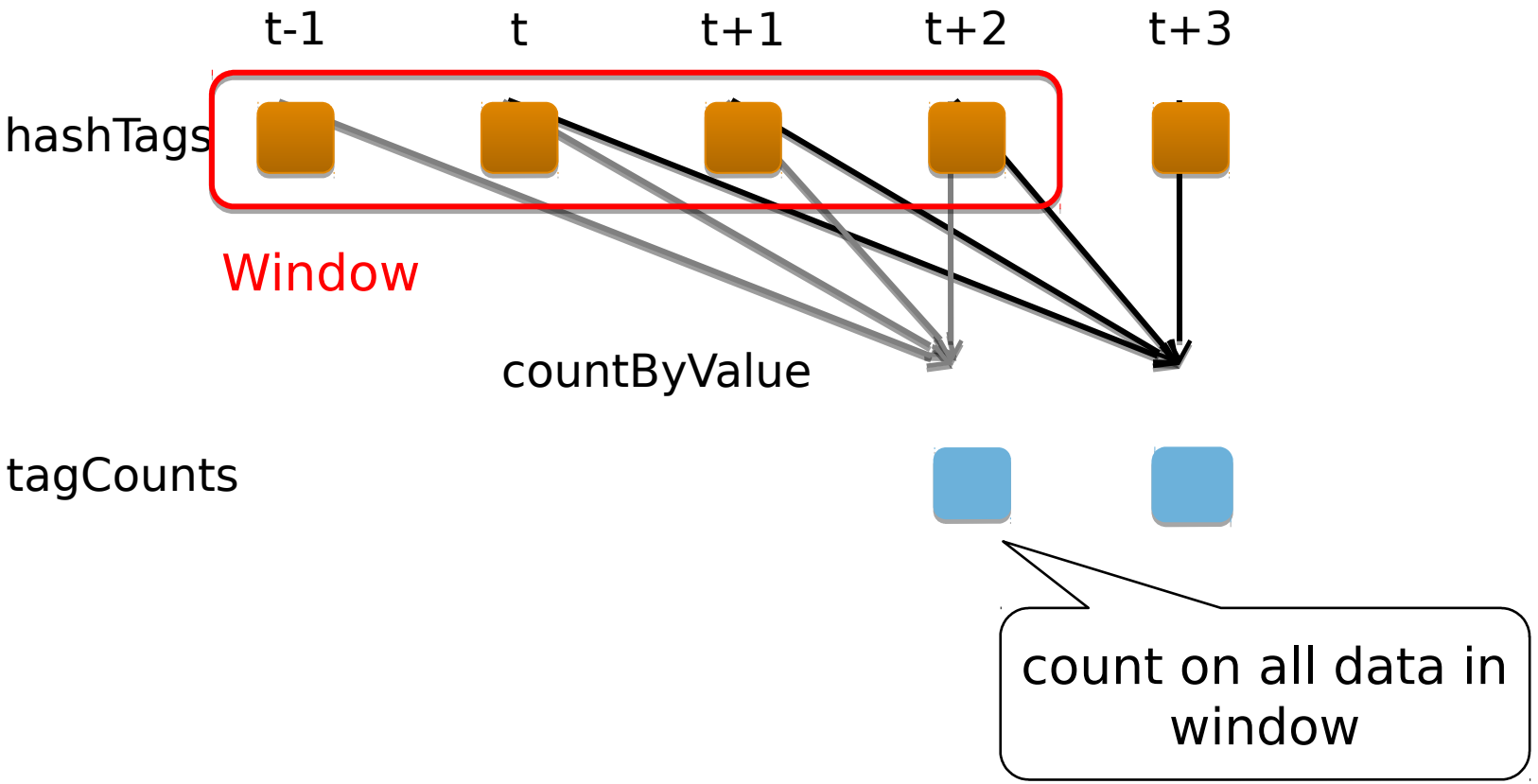
- Count hashtags in the last 10 minutes
  - Returns a set of pairs (hashtag, count)

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



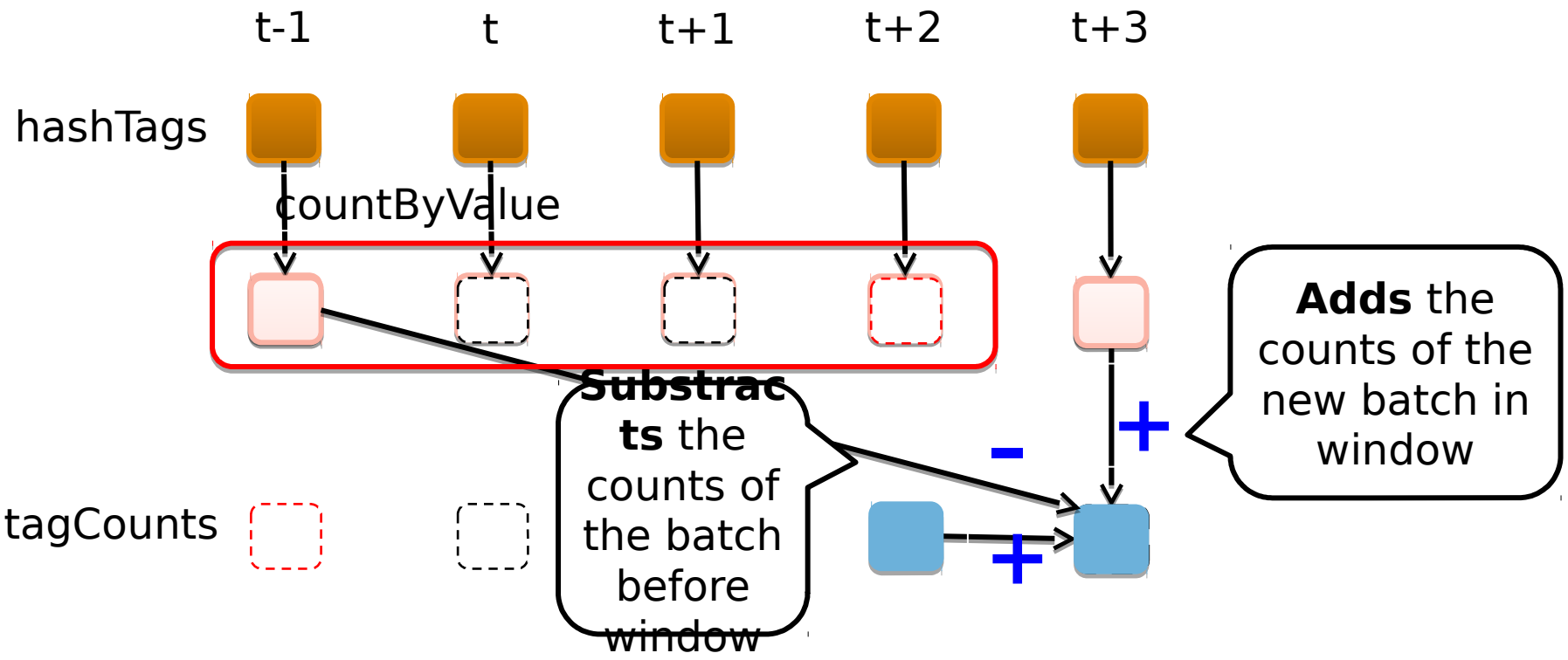
# Example

```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



# countByValue on Sliding Window

```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



# 4. Graph Processing

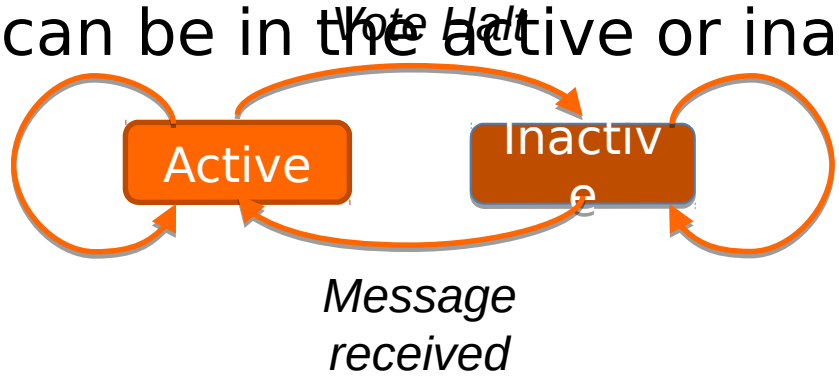
- Why use a graph framework (vs a graph DBMS, e.g. Neo4J)?
  - Storage independency
    - E.g. RDBMS, NoSQL, files, etc.
  - Possibility to control the partitioning of very large graphs for more parallelism and scalability
  - Flexibility in the expression of graph manipulation algorithms
- Examples of graph frameworks
  - Google Pregel
  - Apache Giraph (Facebook, LinkedIn, etc.)

# Google Pregel

- For Google applications that manipulate graphs at Internet scale
  - PageRank : calculation of the importance of a node in a graph (e. g. a web page), according to the number of incoming references
    - The more references, the higher the rank
- Scalable and fault-tolerant framework in a SN cluster
  - Facilitates the implementation of parallel algorithms for graph processing
- Computing model: “Think Like a Vertex”

# Computing Model

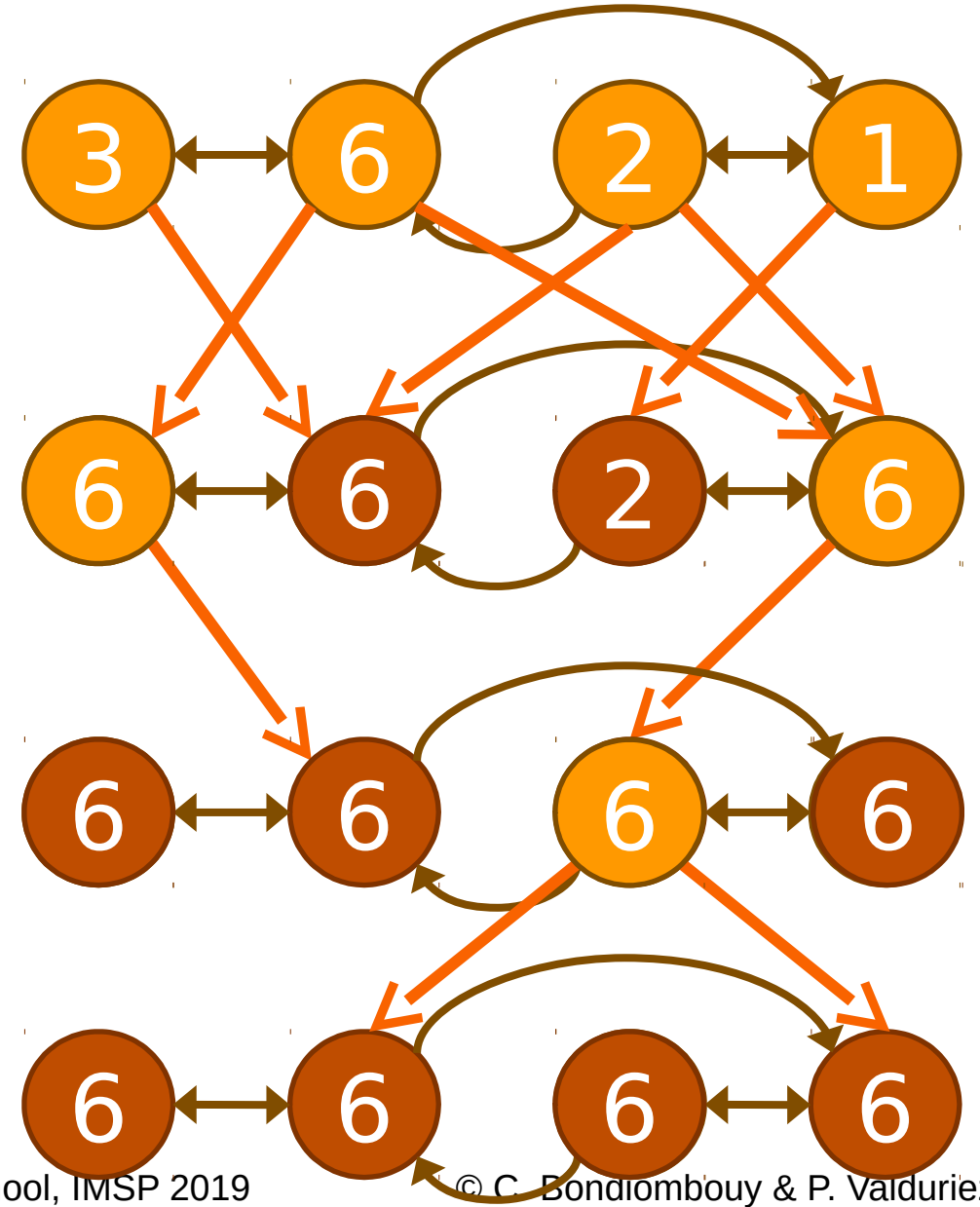
- Input: a directed graph (nodes and arcs)
  - Each node has a modifiable value
  - Each arc has a source node, a value, and a target node
- Node communication
  - A node can send messages through its outgoing arcs, and receive messages through its incoming arcs
  - A node can be in the active or inactive state



# Computing Model

- A graph processing is a sequence of steps called *supersteps*
  - Superstep 0: each node is active
  - At each superstep *S*, a user function *F* is executed by each active node *N*
    - *F* can read the messages sent to *N* at superstep *S-1*
    - *F* can send messages for superstep *S+1*
    - *F* can change the value of *N* and its outgoing arcs
  - Ends when all nodes are inactive

# Computing the Max Value




Messages

Dark nodes
have voted Halt

# Implementation

- C++ API
  - Class Vertex
    - Methods: SendMessage, VoteToHalt, Compute, …
- Master-Worker model
  - The Master partitions the input graph, distributes the partitions to the Workers and their function *F*, and schedules the execution of supersteps
- Fault tolerance
  - Checkpoints: workers write their data to disk
  - The Master sends ping messages to the Workers
    - In case of failure detection, reassignment of data to another Worker and re-execution of the superstep

# Case Study: Facebook

- Problems
  - Very large graphs: up to 100 billion links
  - Complex calculations on very large graphs
- Examples of killer applications
  - Inference of information, e.g. infer unknown site classifications from known site classifications by analyzing overlapping keywords
  - Pagerank applied to the social context
  - Friends of friends score: calculation of the strength of the relationship between 2 users according to their respective friends
- Solution with Giraph
  - Contributions to the Apache project with new techniques to scale up: composition of graph calculations, supersteps partitioning
  - Daily use by Facebook

# 5. Conclusion on Frameworks

- Alternative to traditional BI/RDBMS tools for unstructured data
  - Takeover of the developer/programmer of data analysis applications
- A field marked by a strong evolution
  - MapReduce abandoned by Google, and replaced by Cloud Dataflow
  - Hadoop MapReduce replaced by Apache Spark
    - Same architecture