Dataflow Programming for Big Data Stream Processing

Kai-Uwe Sattler
Database & Information Systems Group
TU Ilmenau
www.tu-ilmenau.de/dbis

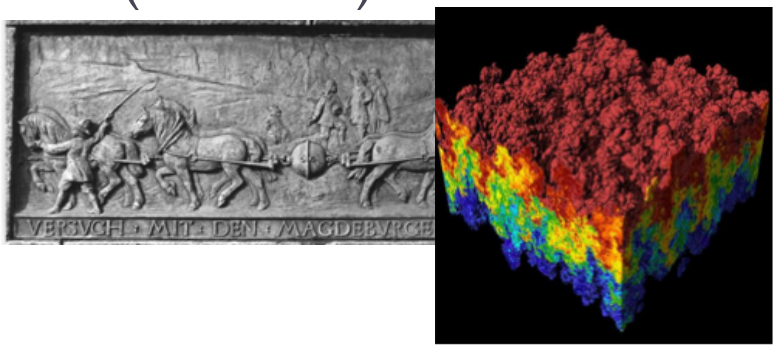
Outline

- Big Dynamic Data: Use Cases & Requirements
- Foundations of Stream Processing
- APIs, Query & Dataflow Languages
- Piglet: Language & Compiler
- Conclusion & Outlook

Quelle: pixabay.com

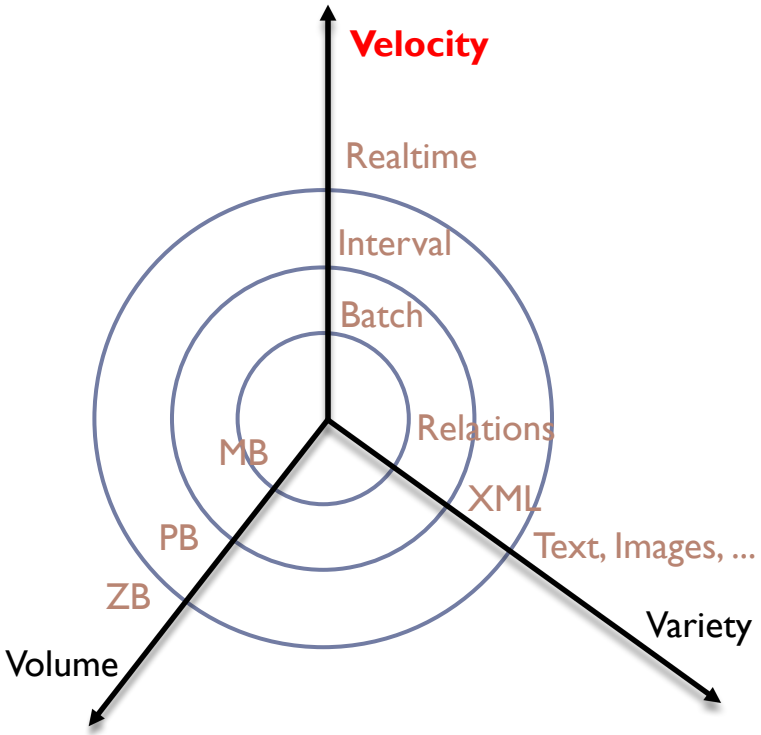
The 4th Paradigm & Big Data

- J. Gray: shift of paradigm in natural sciences
  - 1000 years ago: empirical experiments
  - Since 100 years: grand theories, models, ..
  - Last decades: Computational Sciences (Simulation)
  - today: data exploration (eSciences)



Fotos: Wikipedia

- Big Data = „too big“ for traditional methods

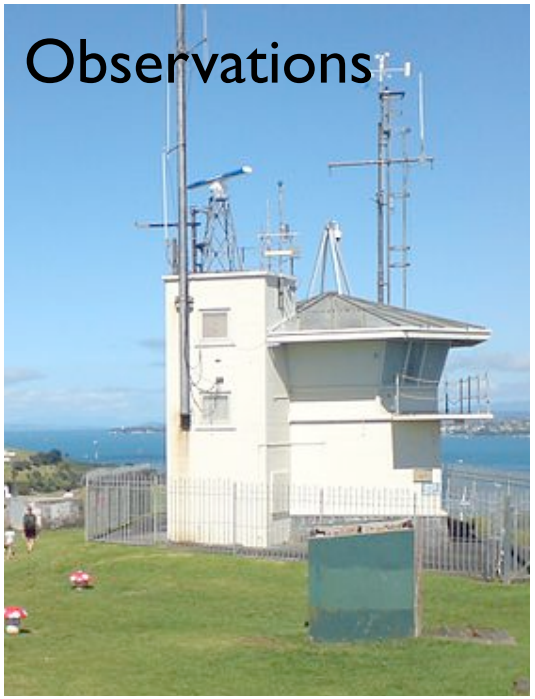


Dynamic Data in the Big Data Age

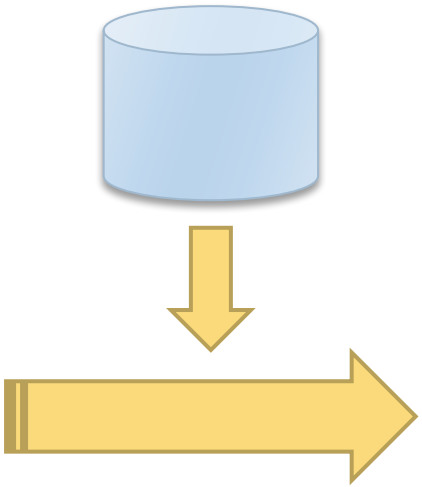
- Availability of sensors connecting IT and physical world
  - Enormous volume of data from real-world observations (=„dirty“ data)
  - Many heterogeneous sources incl. potentially infinite streams
  - Valid or useful only for a short period of time
  - Not inspectible by humans anymore

Use Case: Linked Stream Data

- Combine and analyze sensor data e.g. from weather stations (e.g. www.weatherground.com)
- Link sensor data with metadata (sensors, stations, GeoNames)
- SRBench https://www.w3.org/wiki/SRBench



Metadata

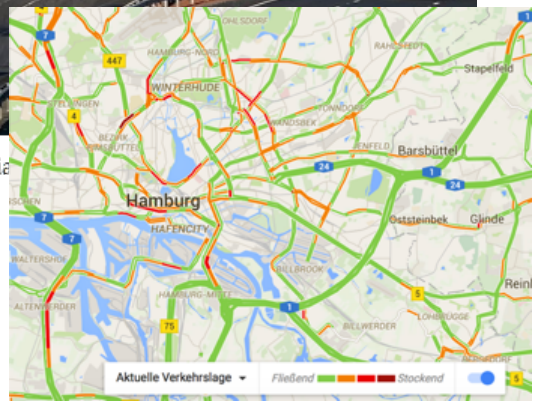


Detect a hurricane, detect a station observing a blizzard, hourly average temperature, ...

Use Case: Traffic Monitoring

- Traffic monitoring using
  - Toll stations,
  - Induction loops,
  - smartphones, ...
- tasks:
  - Toll calculation
  - Speed control
  - Detection of traffic jam and accidents
- Further applications:
  - Variable toll (based on time, congestion, …)
  - Dynamic route planning

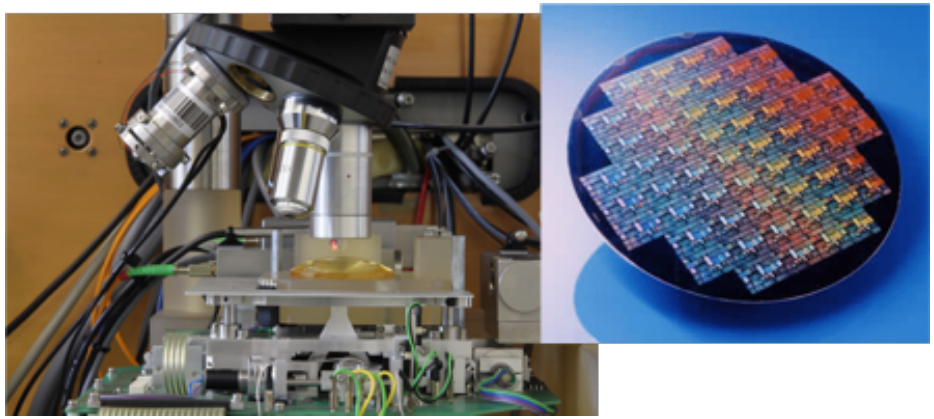
Fotot: Wikipedia

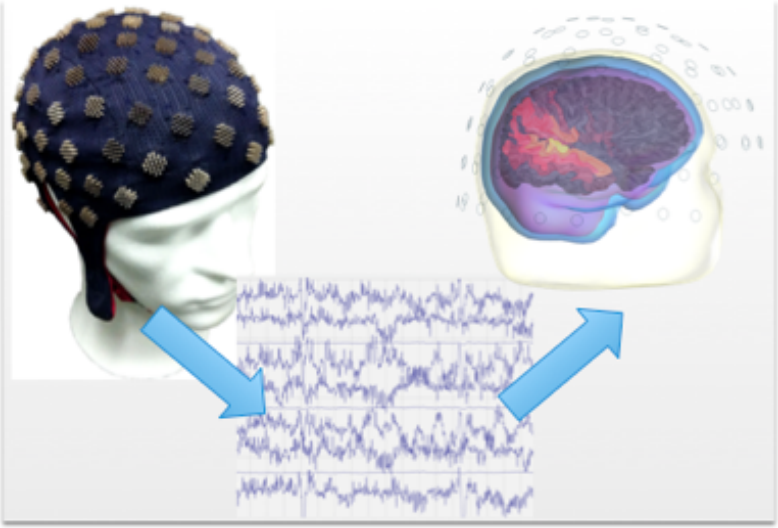


Use Case: Engineering Data

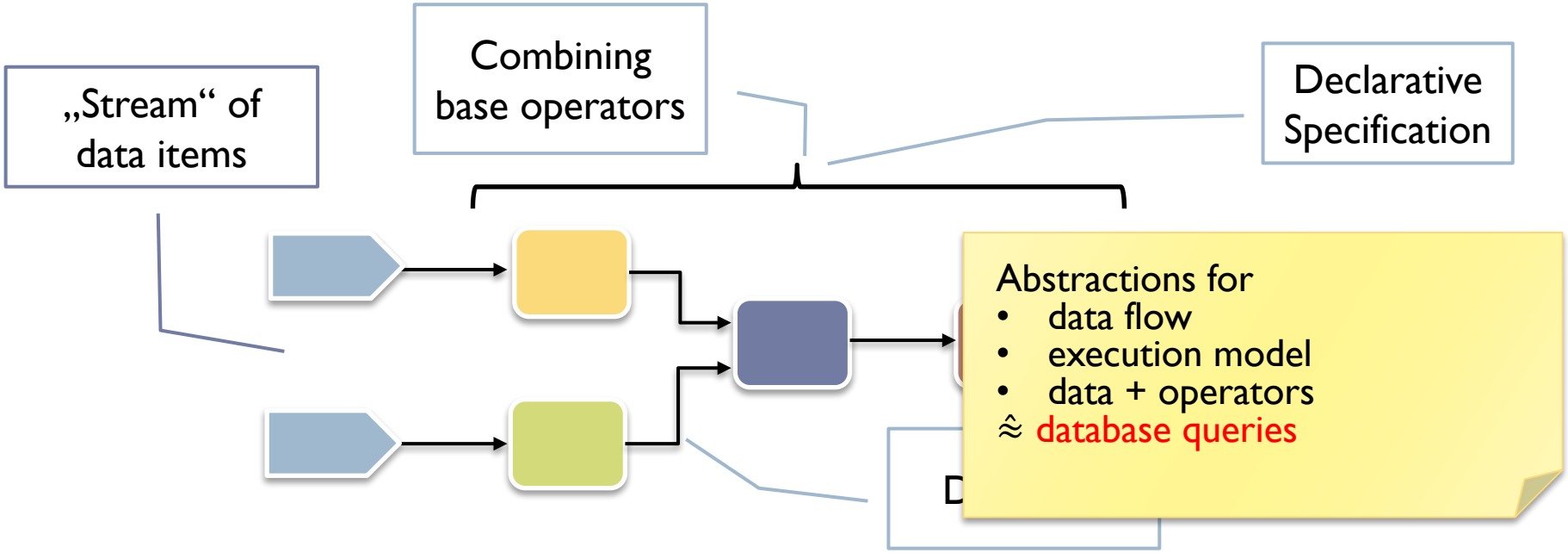
- Find defects in nano structures (e.g. electronic components)
- 3D surface measurement (x,y position, z measure)
- Resolution: nm (100s MB - 10s TBs per scan)

- online source localization in human brain
- EEG/MEG sensors, data rate: 600-5000 Hz
- Complex analysis pipeline: signal filtering, decomposition, matrix inversion, time-based folding





Common Concepts



- Further applications
  - System monitoring, smart grid, process automation (Industry 4.0), Social media analysis, ...

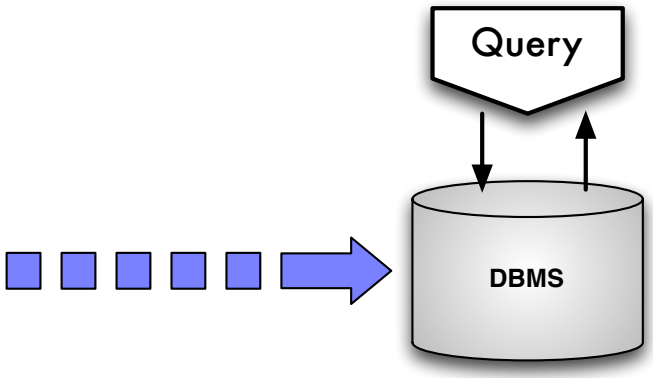
Outline

- Big Dynamic Data: Use Cases & Requirements
- Foundations of Stream Processing
  - Processing Model
  - Windows
  - Basic Operators
  - Scalable Stream Processing
  - Examples
- APIs, Query & Dataflow Languages
- Piglet: Language & Compiler
- Conclusion & Outlook

Quelle: pixabay.com

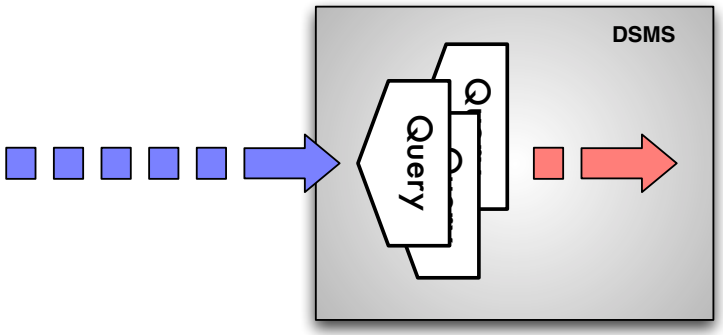
Processing models

- Batch Processing = Store & Process



- Traditional databases
- MapReduce

- Online Processing = Continuous Query Processing (CQP)

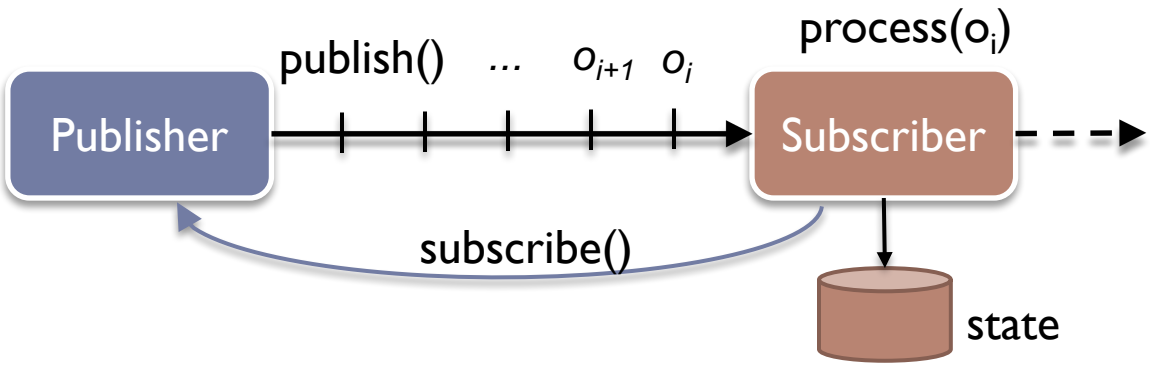


- Data Stream Management Systems (DSMS), Stream Processing Engines (SPE)

Stream Processing: Basic Principles

- Data stream := < o_i, o_{i+1}, o_{i+2}, … >
- Stream element: o_i = (*data items*, *timestamp*)

- Publish subscribe paradigm

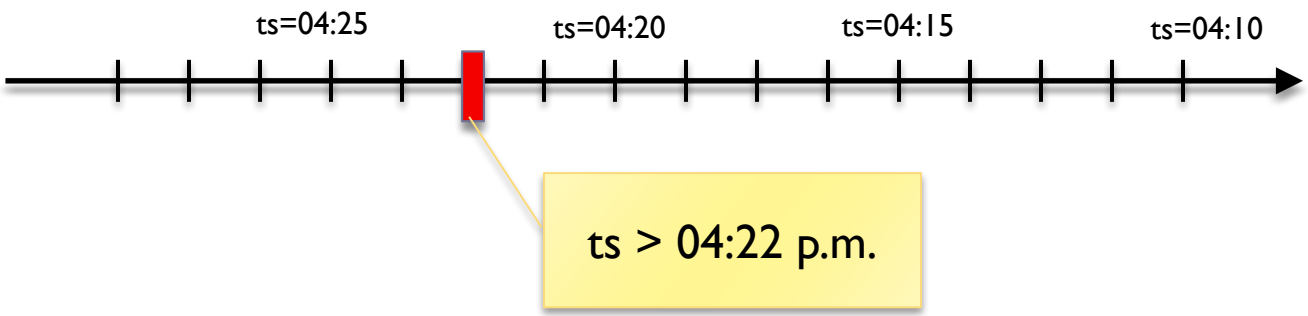


- Challenges
  - (Theoretically) unbounded stream
  - Limited resources (memory, CPU time, ...) for state (computation and representation)

Controlling Stream Processing: Punctuations

- Punctuations mark the end of a substream
  - Conceptually, as a predicate that evaluates to false for every element following the punctuation
  - But represented as data item (=stream element)
- Simple form = heart beat
- Advanced forms: end of window, end of stream, end of session, end of auction, ...
- Processing punctuations:
  - match(t: Tuple, p: Punctuation) → bool
  - combine(p: Punctuation, q: Punctuation) → Punctuation

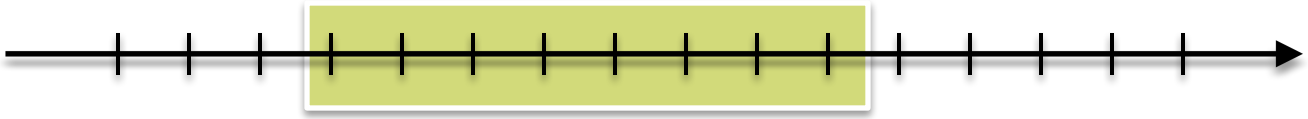
Punctuations: Example



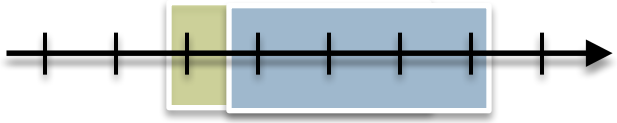
- Allows to
  - implement e.g. window semantics
  - Avoid blocking
  - Optimize processing of aggregates, etc.
  - Out-of-order processing

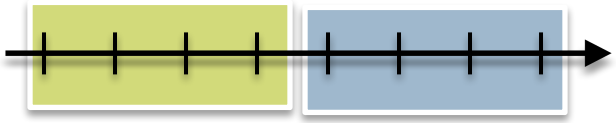
Windows on Data Streams

- Aggregations, joins etc. would require access to <u>all</u> data elements ➟ blocking operators
- Solution: define a window (subset) over a data stream

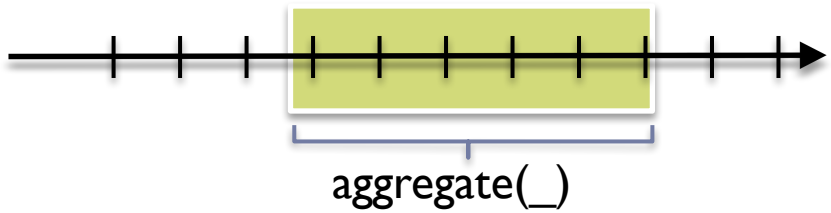


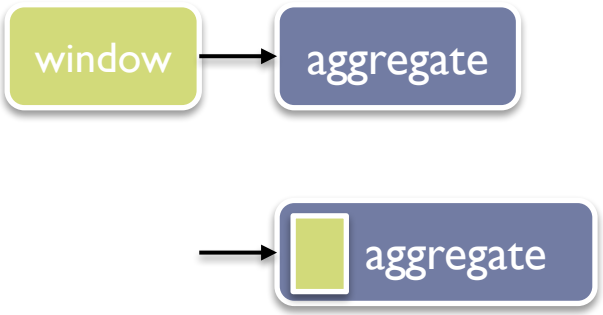
- Different possible semantics:
  - Time-based: last 5 minutes
  - Count-based: 100 data items
  - Data-driven: Web session
- Eviction strategies: sliding vs. tumbling vs. landmark



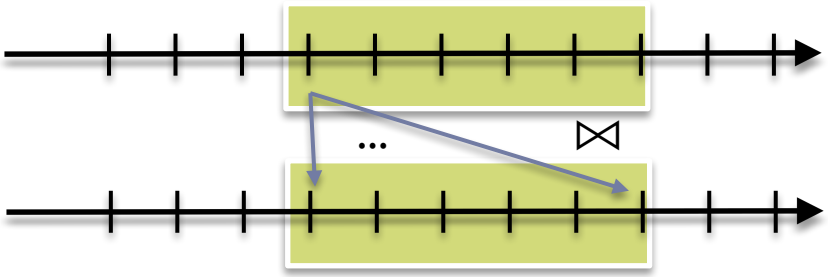


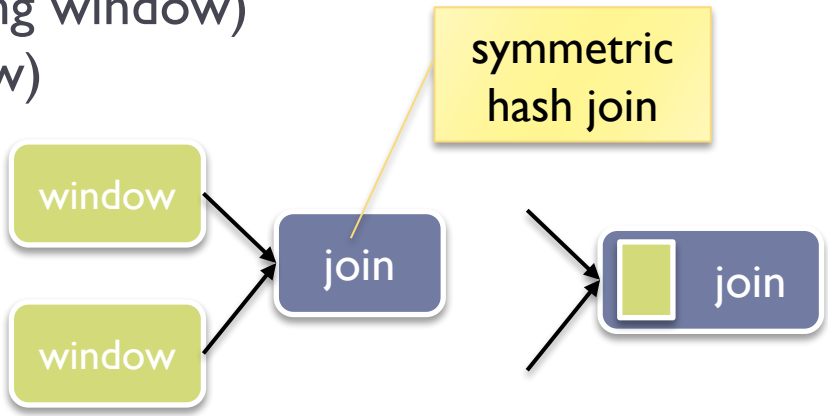
Windowed Aggregation & Joins





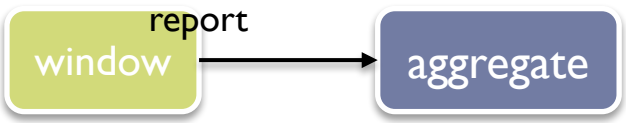
- Examples:
  - Moving average over 5 minutes (sliding window)
  - URL access per day (tumbling window)





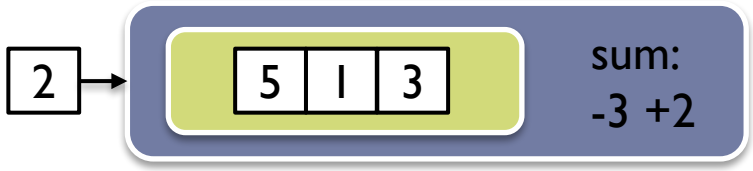
- Example:
  - Correlated observations, e.g. high temperatures within a certain period of time

Window Implementation



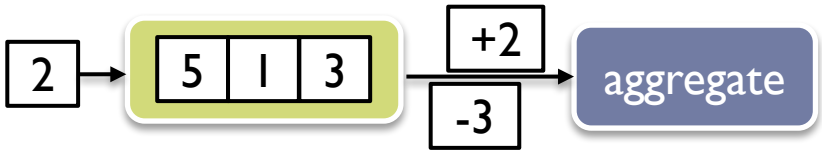
- on report:
  - Re-evaluation
  - Incremental evaluation

- Strategies for incremental evaluation
  - Input triggered



  - Possible delay in evaluation if no new tuple arrives
  - Hard to separate window

  - Negative tuples



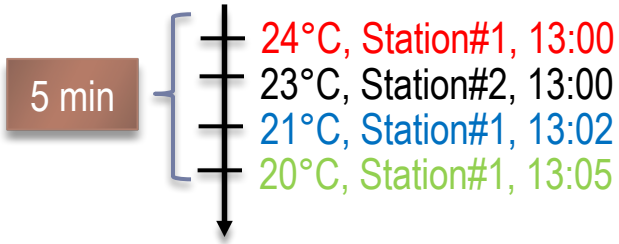
  - Allows to implement windows separately
  - Avoids delay
  - Increases the number of tuples

Complex Event Processing (CEP)

- Detecting complex patterns of events in a stream of data
  - Event = stream element; complex event = sequence of events (not necessarily consecutively)
  - Defined using logical and temporal conditions

Data values + combinations

Within a given period of time

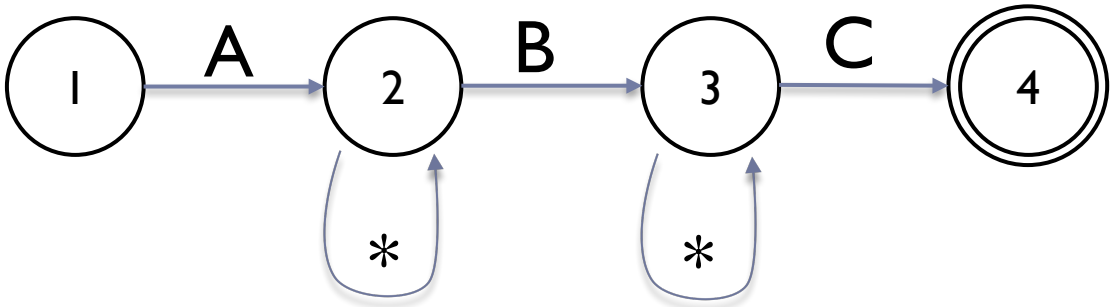



```
SEQ(A, B, C) WITH
  A.Temp > 23°C &&
  B.Station = A.Station && B.Temp < A.Temp &&
  C.Station = A.Station && A.Temp-C.Temp > 3
```

CEP: Implementation

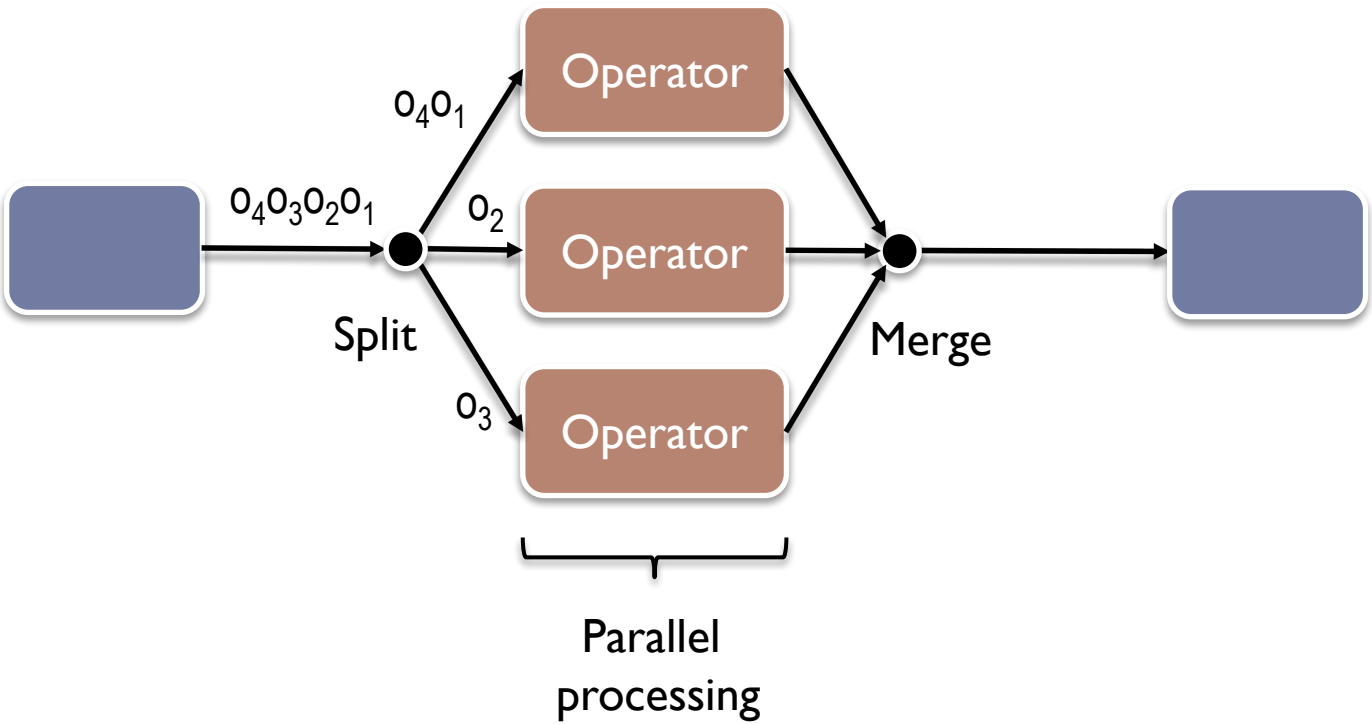
- Composite events constructed e.g. by
  - SEQ, AND, OR, NEG, ...
  - $SEQ(e_1, e_2) \rightarrow (e_1, t_1) \wedge (e_2, t_2) \wedge t_1 \leq t_2 \wedge e_1, e_2 \in \mathbb{W}$

- Implemented by constructing a NFA
- Example: SEQ(A, B, C)



Scalability: Data-parallel processing

- High volume of data, high arrival rates
  ➟ parallel processing (Multicore CPUs, Cluster, ...)
  ➟ requires partitioning of the data stream



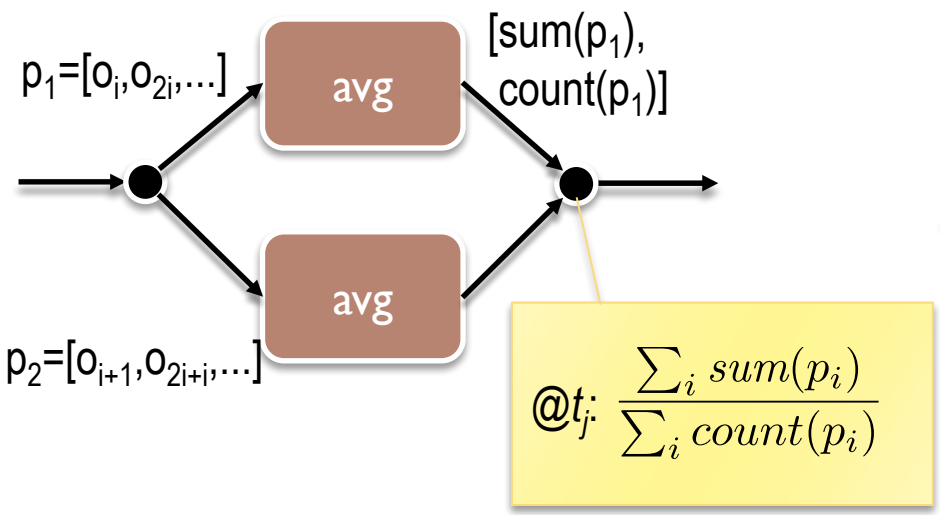
Data parallelism in data streams

- Non-trivial problem
  - Stateful operators (e.g. aggregates, ...)
  - Operators relying on ordering of tuples (e.g. windows, CEP, ...)
  - Dynamic load balancing / elasticity in case of changing workloads or skewed data
  - State migration

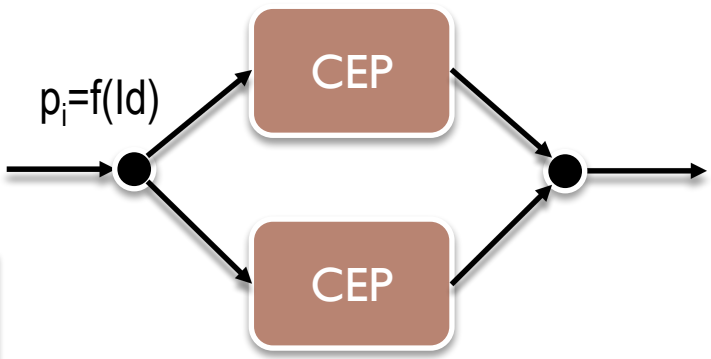
- Tasks
  - Partitioning (split + merge)
  - Safe auto-parallelization
  - State management + migration

Partitioning

- Some possible solutions
  - Partitioning: hash-based (e.g. consistent hashing), range-based, ...
  - Operator-specific merge
  - „semantic " partitioning

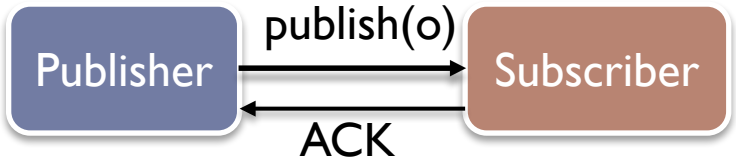


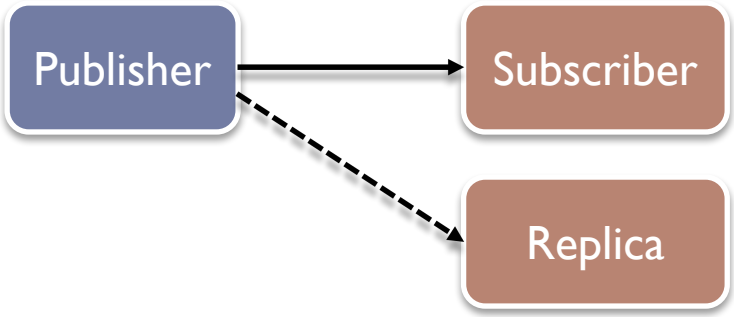
SEQ(A, B) WITH B.Id = A.Id && ...



Fault tolerance

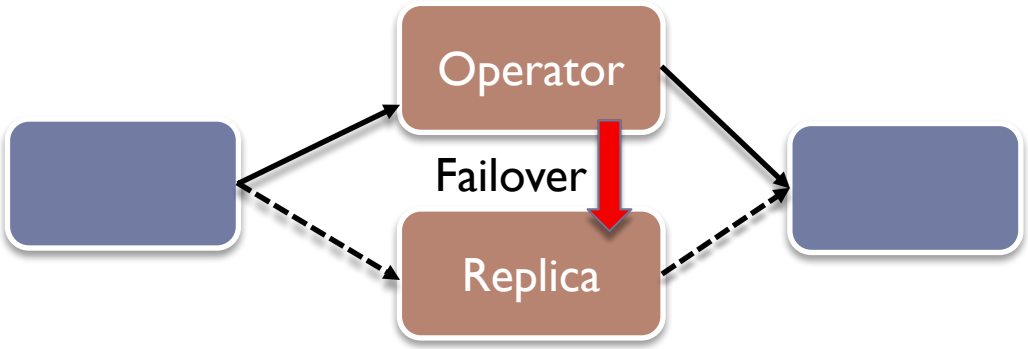
- Continuous queries on unbounded data streams require to deal with
  - Loss of message
  - Guaranteed message delivery:
    - Best effort
    - At least once
    - Exactly once
  - Node failures
  - Redundant processing and/or recovery





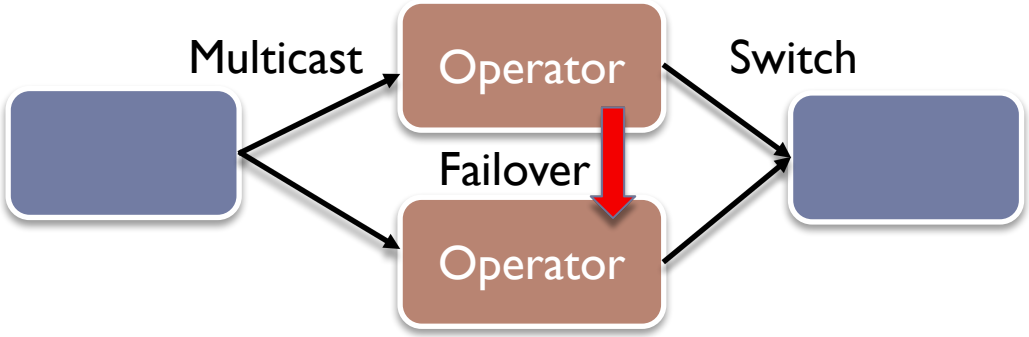
Fault Tolerance: Standby

- Standby



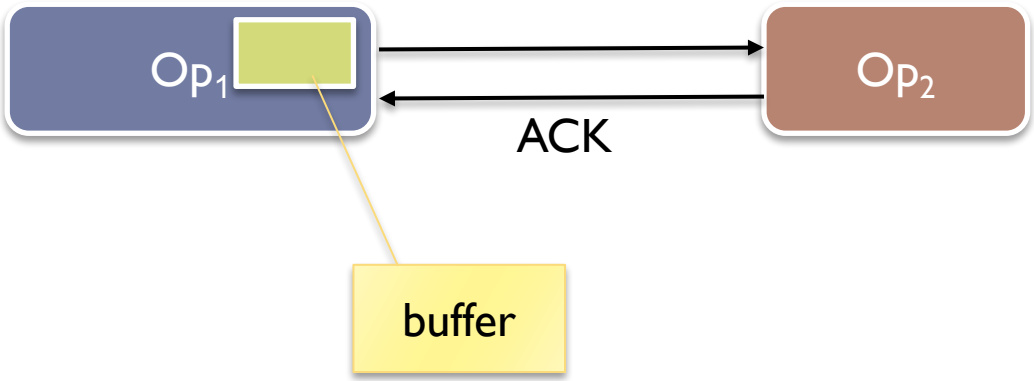
- Simple for stateless operators like filters, etc.
- But for stateful operators (e.g. aggregates, ...)??

- Hot Standby



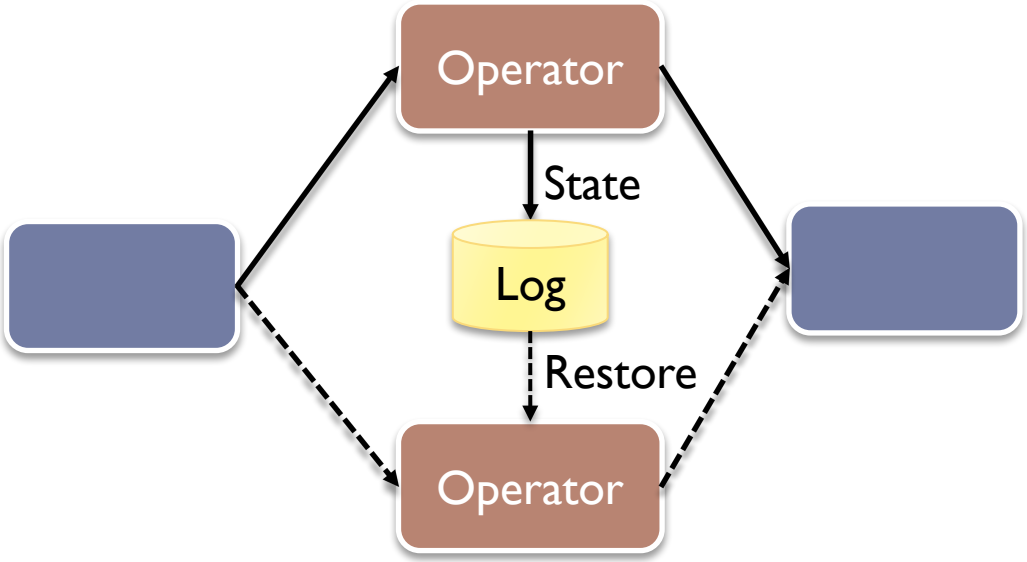
Fault Tolerance: Upstream Backup

- Operator buffers data sent to the the next operator until acknowledgement
- In case of failure: retransmit
- Can be combined with standby/replicas



Fault Tolerance: Checkpointing

- Checkpointing/ Logging



- Basic idea:
  - Periodically/on changes: saving the state (checkpointing, snapshot) or saving state changes (logging)
  - In case of failures: recover the state from the log
- Challenges:
  - State management (buffer, routing, processing), overhead
  - Consistency among multiple operators during recovery

Challenges in Dataflow Compilation

- Optimization: finding the best plan for a declaratively specified dataflow
  - Choosing algorithms (for operators) and data structures (for windows, state representation, ...)
  - Reordering operators, e.g. pushdown of selection
  - Auto-parallelization & partitioning of data streams for distributed/parallel processing
  - Placement of operators at computing nodes
  - Resource allocation (memory, CPU time, ...)
  - ...
- Extensibility: integrating and optimizing new operators (possible as black box) and new target platforms

Optimization

- requires
  - Knowledge about algebraic properties of operators such as associativity, commutativity ➡ reordering
  - Transformations and their properties
  - Cost model: expected processing costs, arrival rates, data distribution, ...
- Challenges
  - Operators beyond standard (relational) operators = user-defined code: functions, operators, ...
  - Cost model: parameters, statistics for streams

Optimizations in Stream Processing

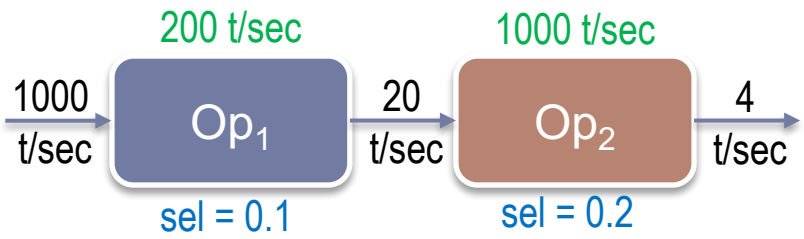
- Hirzel et. al: A Catalog of Stream Processing Optimizations, ACM Comp. Surveys 46(4), 2014.

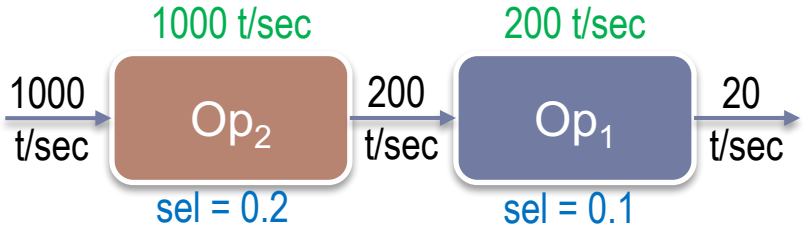
Transformation	Graph	Semantics	Dynamic
Operator reordering	changed	unchanged	(depends)
Redundancy elimination	changed	unchanged	(depends)
Operator separation	changed	unchanged	static
Fusion	changed	unchanged	(depends)
Fission	changed	(depends)	(depends)
Placement	unchanged	unchanged	(depends)
Load balancing	unchanged	unchanged	(depends)
State sharing	unchanged	unchanged	static
Batching	unchanged	unchanged	(depends)
Algorithm selection	unchanged	(depends)	(depends)
Load shedding	unchanged	changed	dynamic

Cost model

- Traditional cost-based query optimization is based on cardinality estimation ➟ inadequate for unbounded streams
- Possible solution: rate-based cost estimation (Viglas et al.: Rate-based query optimization for streaming information sources, SIGMOD 2002)

$$output\ rate = \frac{\#outputs\ transmitted}{time\ for\ transmission}$$





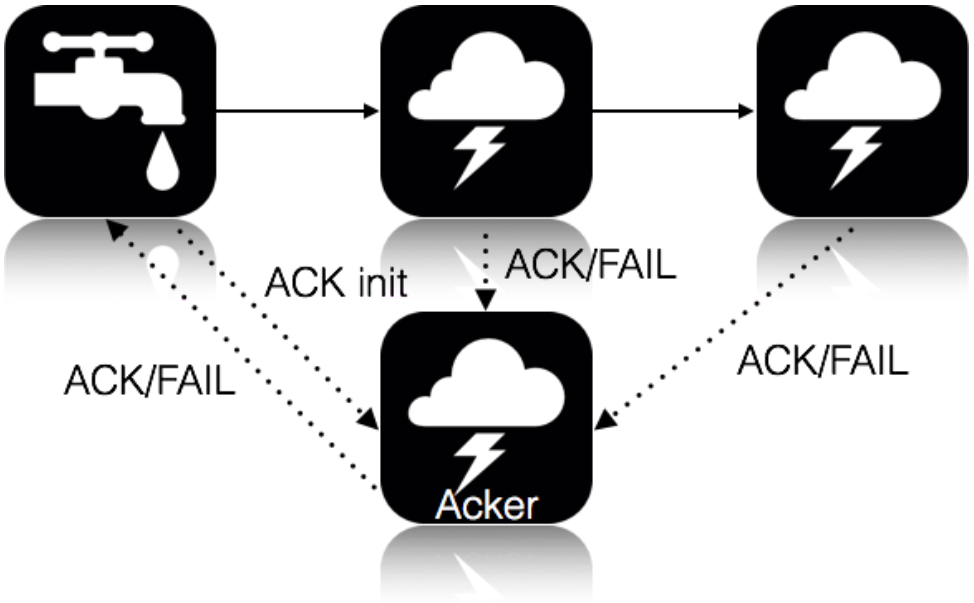
- Challenges:
  - Fluctuating streams
  - Data-parallel processing

Examples: Storm

- scalable stream processing platform by Twitter
  - Tuple-wise computation model ➟ low latency
  - Processing components can be defined in any language
  - Stateless processing
- Components:
  - Spouts: source of tuples (from e.g. MQs)
  - Bolts: consumes & processes input streams and passes them to other bolts or external systems
  - Wiring spouts and bolts leads to a directed graph called topology
  - Spouts and bolts associated with set of tasks that run in parallel on multiple machines

Storm: Fault Tolerance

- At-least-once guarantee via Record ACKs



- Acker:
  - „logs“ the progress of each tuple emitted by spout

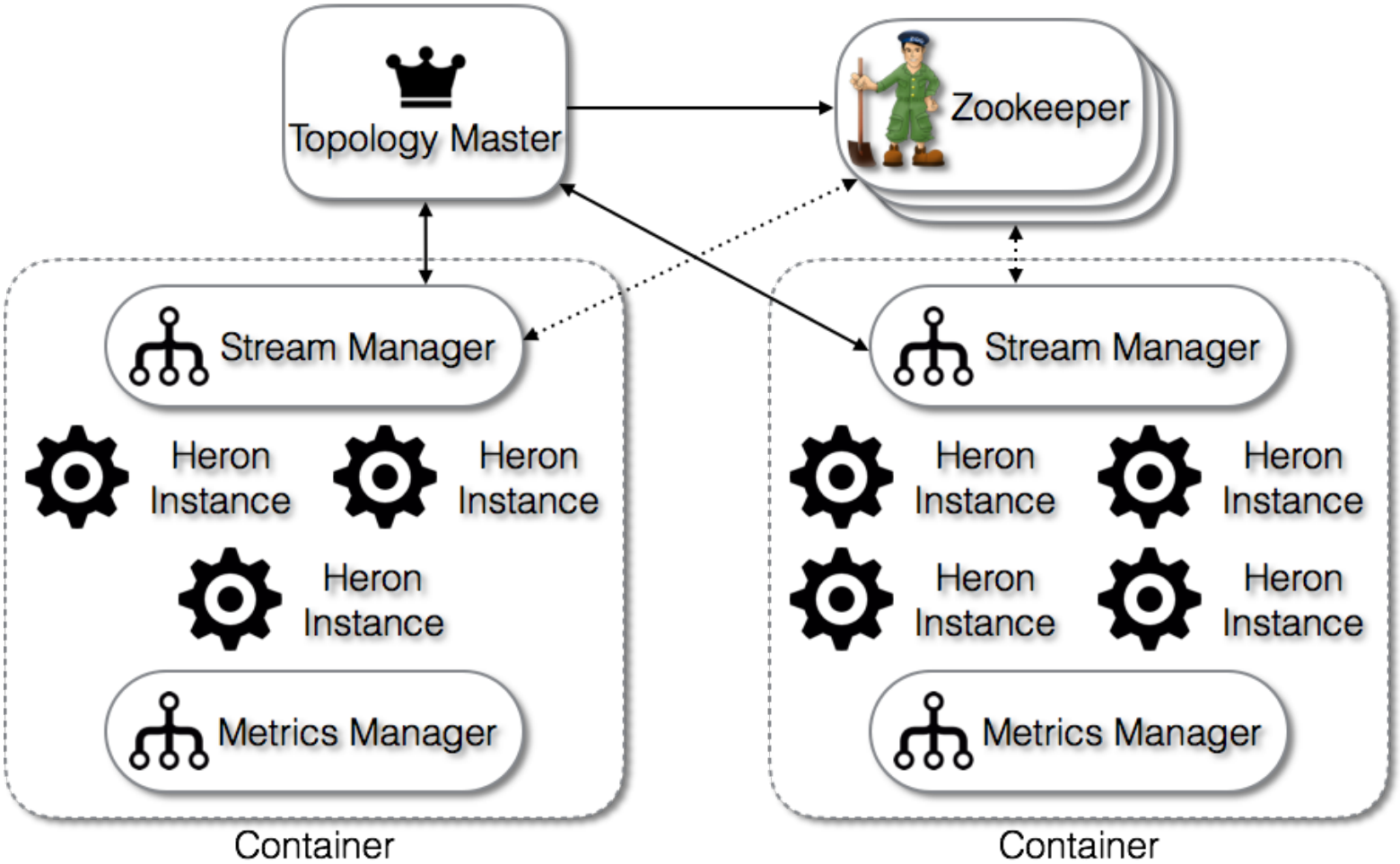
Storm & Trident

- High-level abstraction built on top of Storm core: operators like filter, join, groupBy, ...
- Stream-oriented API + UDFs
- Stateful, incremental processing
- Micro-Batch oriented (ordered & partitionable)
- Exactly-once semantics
- Trident topology compiled into spouts and bolt

Heron

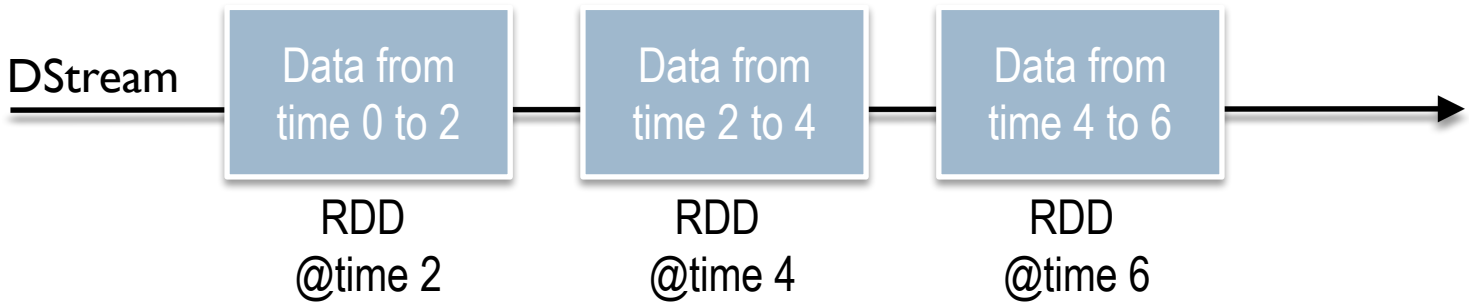
- New real-time streaming system based on Storm
- Introduced June 2015 by Twitter (SIGMOD)
- Fully compatible with Storm API
- Container-based implementation
- Back pressure mechanism
- Easy debugging of heron topologies through UI
- better performance than Storm (latency + throughput)

Heron: Topology Architecture



Examples: Spark Streaming

- extension of the core Spark API for stream processing
- Data sources: Kafka, Flume, Twitter, ZeroMQ, TCP sockets, ...
- Key abstraction: discretized streams (DStream)
  - micro-batch = series of RDDs
  - Stream computation = series of deterministic batch computation at a given time interval

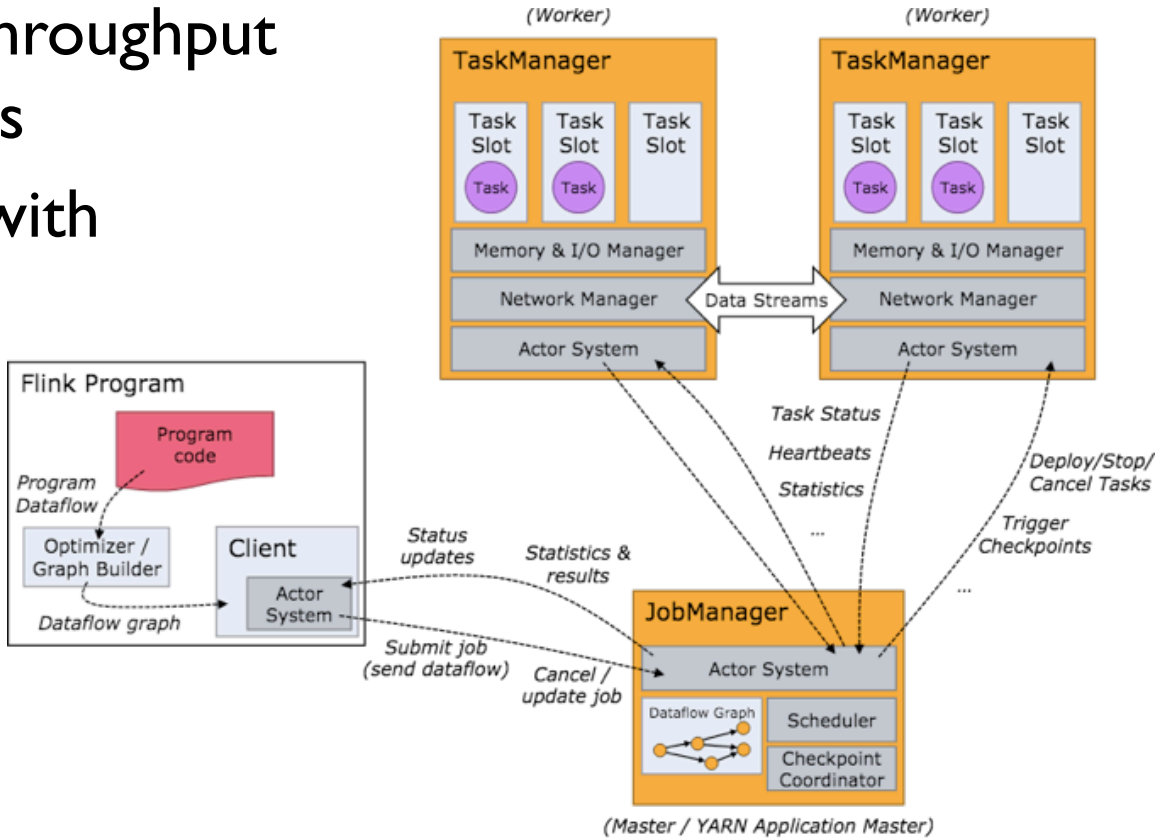


Examples: Spark Streaming

- API very similar to Spark core (Java, Scala, Python)
  - (stateless) transformations on DStreams: map, filter, reduce, repartition, cogroup, ...
  - Stateful operators: time-based window operations, incremental aggregation, time-skewed joins
- Fault tolerance:
  - Exactly-once semantics using checkpoints (asynchronous replication of state RDDs)

Examples: Flink

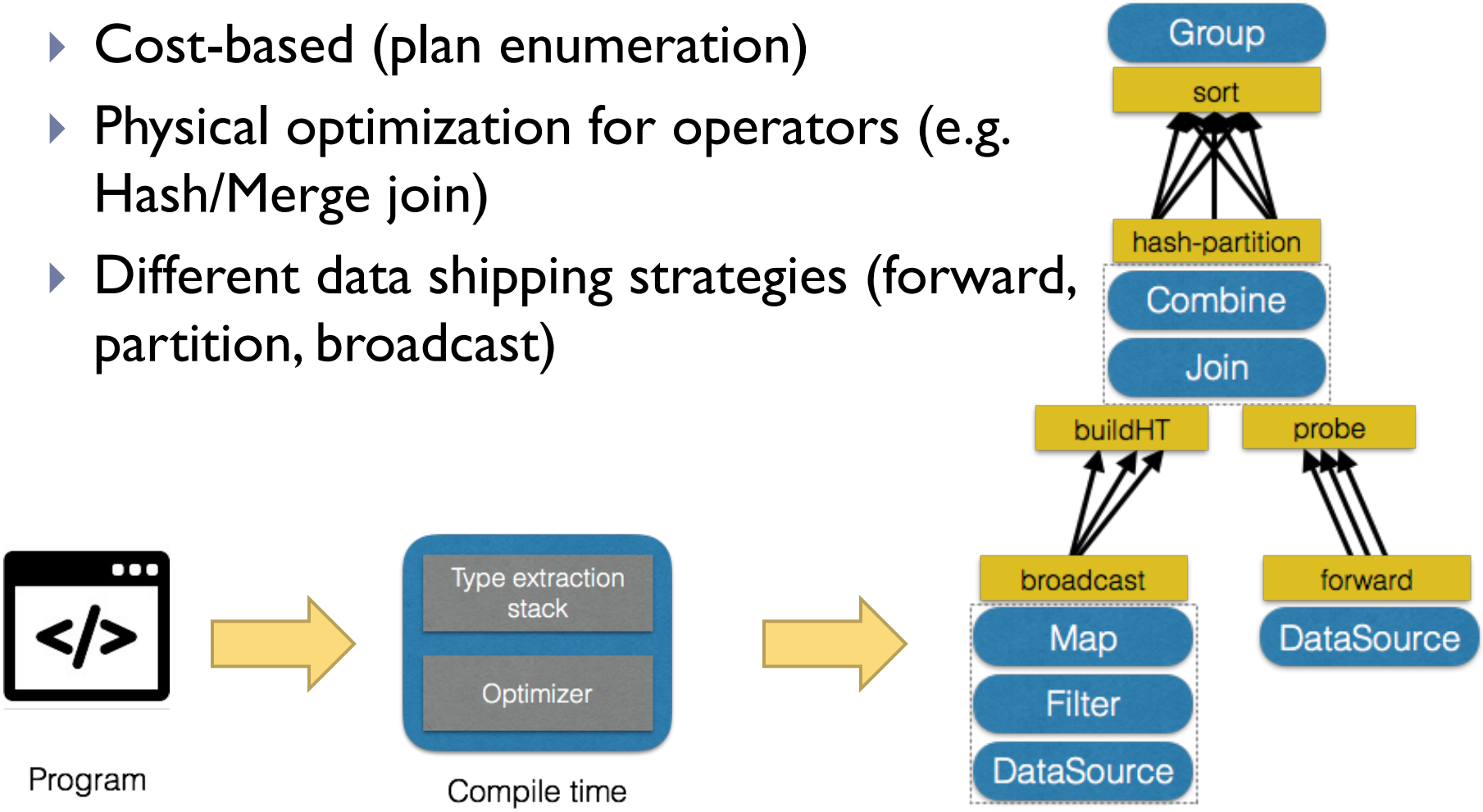
- Open source distributed stream processing platform
- Batch support built on top
- Adjustable latency/throughput through tuple buffers
- Pipelining principle with natural flow control
- Stateful processing



Source: https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/runtime.html

Flink: Optimizer

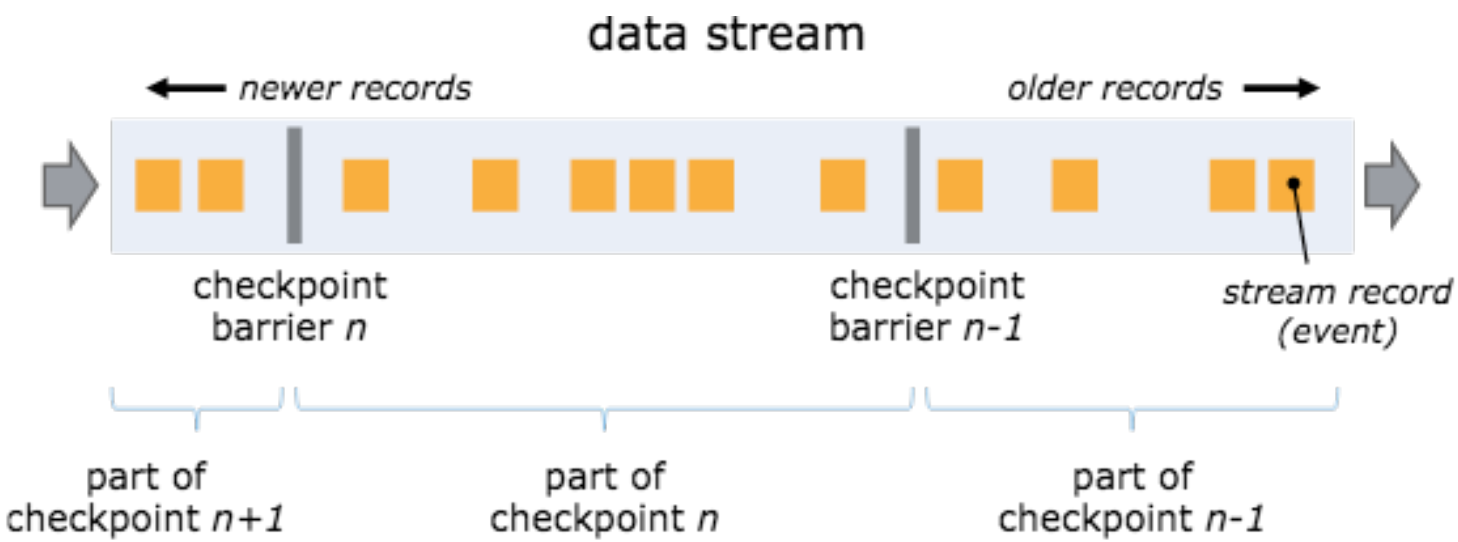
- Cost-based (plan enumeration)
- Physical optimization for operators (e.g. Hash/Merge join)
- Different data shipping strategies (forward, partition, broadcast)



Source: http://de.slideshare.net/Hadoop_Summit/apache-flink-deep-dive

Flink: Fault Tolerance

- Asynchronous Barrier Snapshotting (ABS)
- Stream barriers alongside the data (≙ punctuations)
- Alignment at the operators
- State report to state backend (e.g. HDFS)



Source: https://ci.apache.org/projects/flink/flink-docs-release-1.2/internals/stream_checkpointing.html

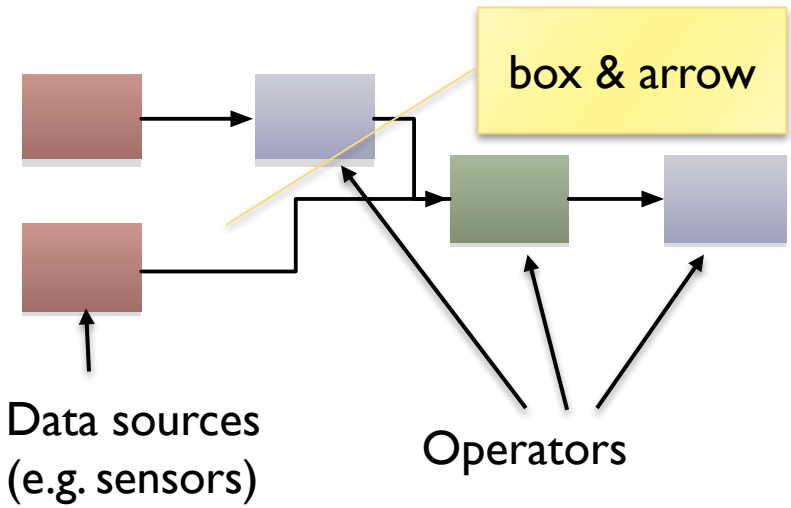
Outline

- Big Dynamic Data: Use Cases & Requirements
- Foundations of Stream Processing
- APIs, Query & Dataflow Languages
- Piglet: Language & Compiler
- Conclusion & Outlook

Quelle: pixabay.com

Languages for Data Stream Processing

- Declarative specification of stream processing pipeplines using visual or textual languages



query & dataflow languages

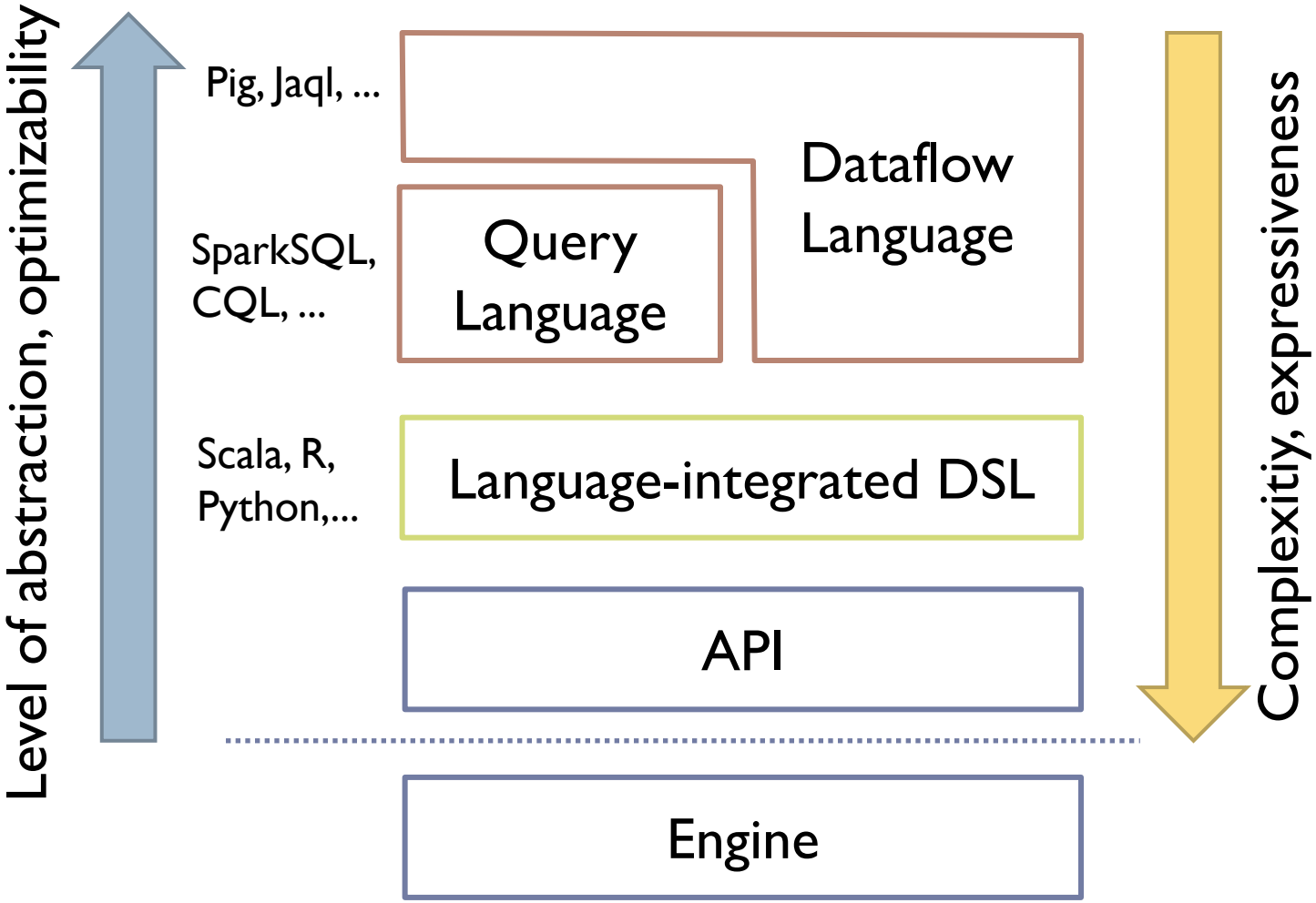
```
SELECT Lane, AVG(Speed)
FROM Traffic [RANGE 1 HOUR]
GROUP BY Lane
```

```
in = SOCKET_READ '*:8080' AS (...);
b = WINDOW in RANGE 1 HOUR;
c = GROUP b BY Lane;
out = FOREACH c
        GENERATE b.Lane, AVG(b.Speed);
```

```
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
val hashTags = tweets.flatMap (tweet => extractTags(tweet))
val counts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
counts.foreachRDD(rdd => rdd.foreach(elem => println(elem)))
```

DSL (Scala, Python, ...)

APIs, DSLs, and Query Languages



DSL for Big Data (Stream) Processing

- Functional programming languages
  - Inspired by LINQ, Java Streaming API, ...
  - Easy construction of complex dataflows from simple transformation operators

- Lazy evaluation
  - Evaluation on demand instead of immediately

- Processing
  - First, construct a DAG only – don't execute
  - Only some actions (print, save) trigger execution

optimization

How to implement a DSL?

- Example: Scala DSL for Apache Spark
  - RDD abstraction for dataset
  - transformations as methods of RDD class

```
class RDD[T] {
    val plan: DAG  /* plan representation */
    def op1(): RDD[T] = { ...  plan.append("op1") /* add operator to DAG */ }
    def action(): RDD[T] = { /* process plan & execute */ }
}
```

- Usage:
  - create a RDD object
  - apply transformations

```
val rdd = ...
rdd.op1()
    .op2()
    .action()
```

DSL for the Non-Programmer?

- You have to ...
  - ... know Scala, Python or Java.
  - ... know the API of your underlying platform (Spark with RDD or DataFrame or ..., Flink, ...).
  - ... write boilerplate code to represent your data (generic lists, tuples, case classes).
  - ... implement, build, and deploy your user-defined operators.
  - ... optimize your program still by hand (but there is some work addressing this issue).

Outline

- Big Dynamic Data: Use Cases & Requirements
- Foundations of Stream Processing
- APIs, Query & Dataflow Languages
- Piglet: Language & Compiler
  - Overview
  - Language Features
  - Extensible Rewriting
  - Code Generation
- Conclusion & Outlook

Quelle: pixabay.com

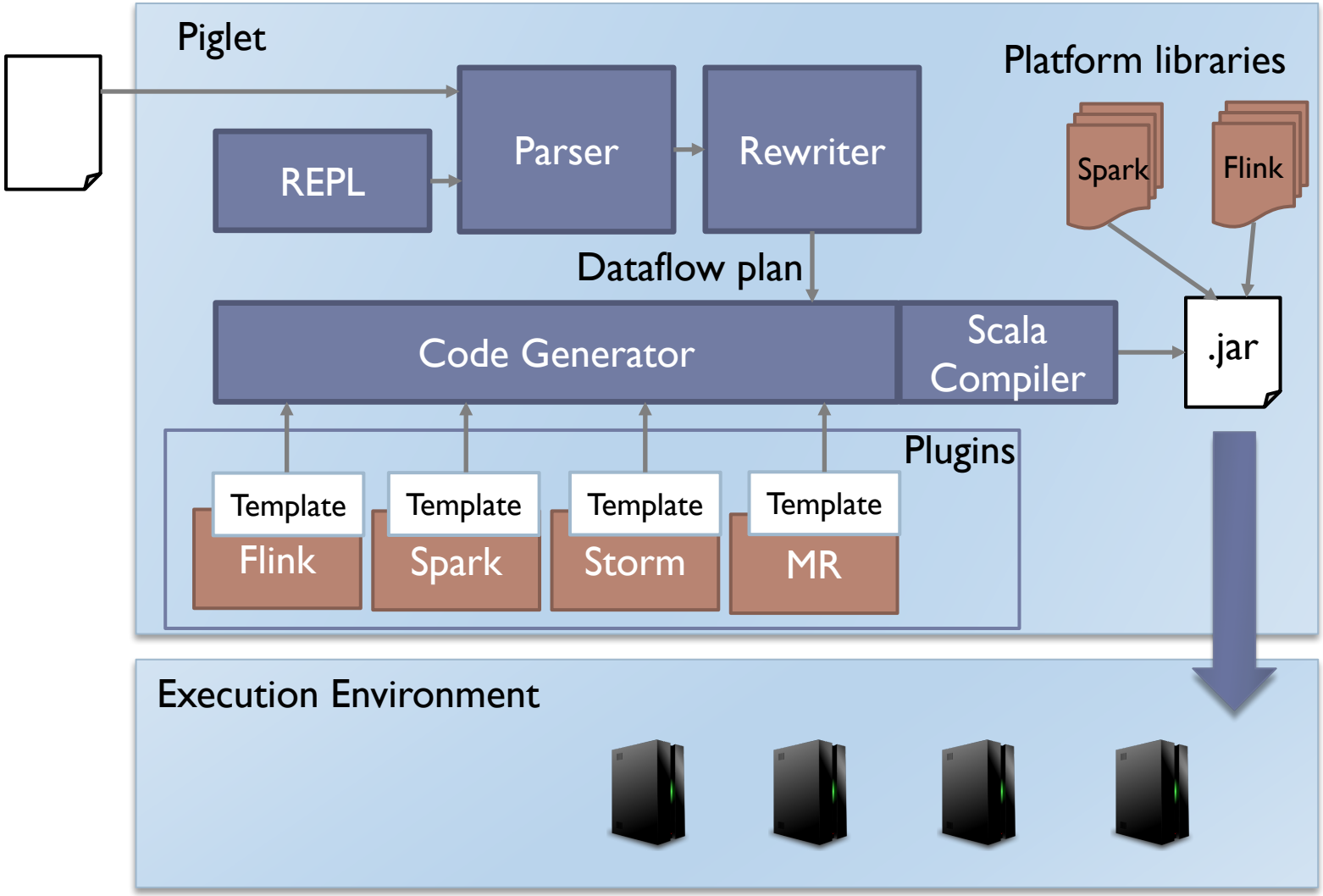
Piglet = Pig Latin++

- Language extensions + compiler for a Pig Latin derivate
  - support for different target backends, e.g. Spark, Flink, ...

```
> piglet --backend spark --interactive --master yarn-client
Welcome to PigREPL ver. 0.3
pigsh>
```

  - language extensions for data stream processing, spatial data, SPARQL, CEP, ...
  - seamless integration of user-defined code, operators & rewriting
  - REPL interface
- available at https://github.com/dbis-ilm/piglet

Piglet Compiler Architecture



Comparison with Related Work

- Why Pig Latin language?

  Language syntax doesn't really matter.
- Apache Pig Latin
  - Hadoop backend, some initial(?) support for Spark
  - Fixed & limited set of optimization rules
- Apache Spark with Scala, SparkSQL, R, ...
  - Multiple APIs & languages on the same backend
- Google Dataflow, Apache Beam
  - Supports (or will support) both streaming + batch
  - Java API only
  - Could be used as target backend, too

Piglet: Language Extensions

- Stream processing
- Complex event processing
- Support for spatial data
- SPARQL integration
- Integration of matrix data and R

Stream Processing

- stream processing engine as backend: Storm, Spark Streaming, Flink Streaming, ...
- streaming sources & sinks:

```
aBag = SOCKET_READ '127.0.0.1:8080' MODE Kafka
        USING PigStream(',') AS (...);
...
SOCKET_WRITE anotherBag TO '141.44.24.124:9000';
```

- support for windows: sliding & tumbling, row-based & range-based

```
bwin = WINDOW stream RANGE 10 SECONDS;
grpd = GROUP bwin BY key;
cntd = FOREACH grpd GENERATE group, COUNT(bwin);
```

Mapping of Window Operations

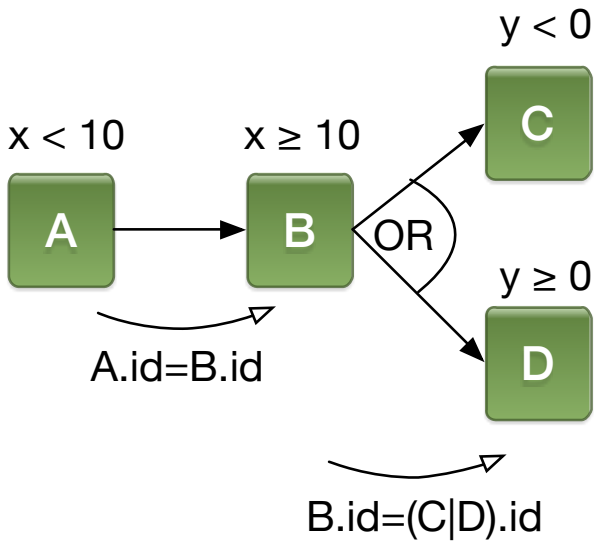
- Spark Streaming
  - Time - sliding/tumbling
- Flink Streaming
  - Global, time, count, delta - sliding/tumbling
  - Various interpretations of time
  - Sliding scope: (slide length – window length)
- Storm
  - Not directly supported, instead in Trident: partitionPersist and stateQuery functions

Complex Event Processing

- Primitive event = stream element
- Complex event = sequence or combination of events

within 60 seconds

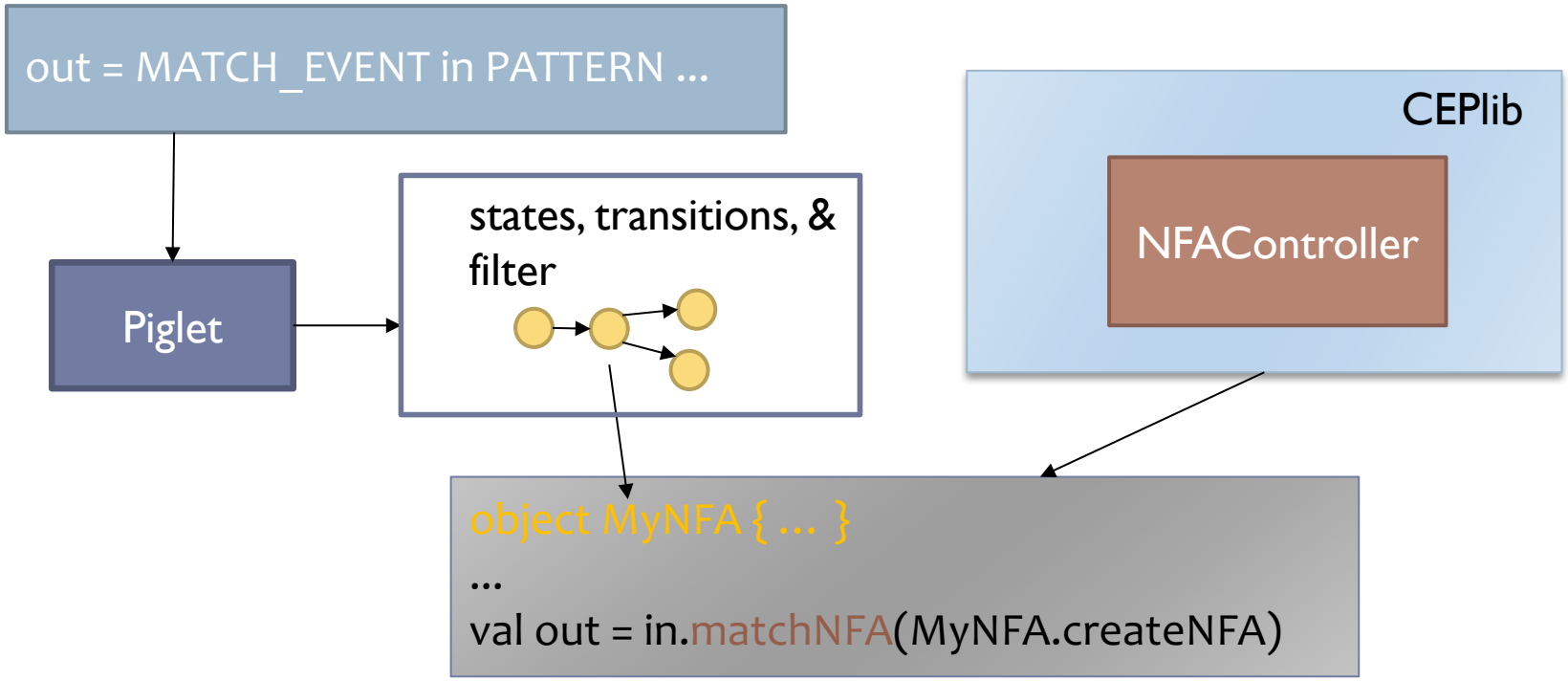
tuple A (id = 1, x = 4, y = 0)
…
tuple B (id = 1, x = 11, y = 0)
…
tuple D (id = 1, x = 4, y = 7)




```
win = WINDOW seq RANGE 60 SECONDS;
out = MATCH_EVENT win PATTERN SEQ(A,B,OR(C,D))
        WITH (A:  x < 10, B:  id == A.id && x > 10, … );
```

Complex Event Processing

- CEP library for Spark and Flink
  - Implementation of a generic NFA
- Code generator for states & transitions



Spatial Data

- New data type geometry = wrapper for point, polyline, polygon
- Spatial predicates for FILTER

```
res = SPATIAL_FILTER bag
      BY intersects($1, geometry(POLYGON((1.0 1.0, 1.0 2.0, ...))));
```

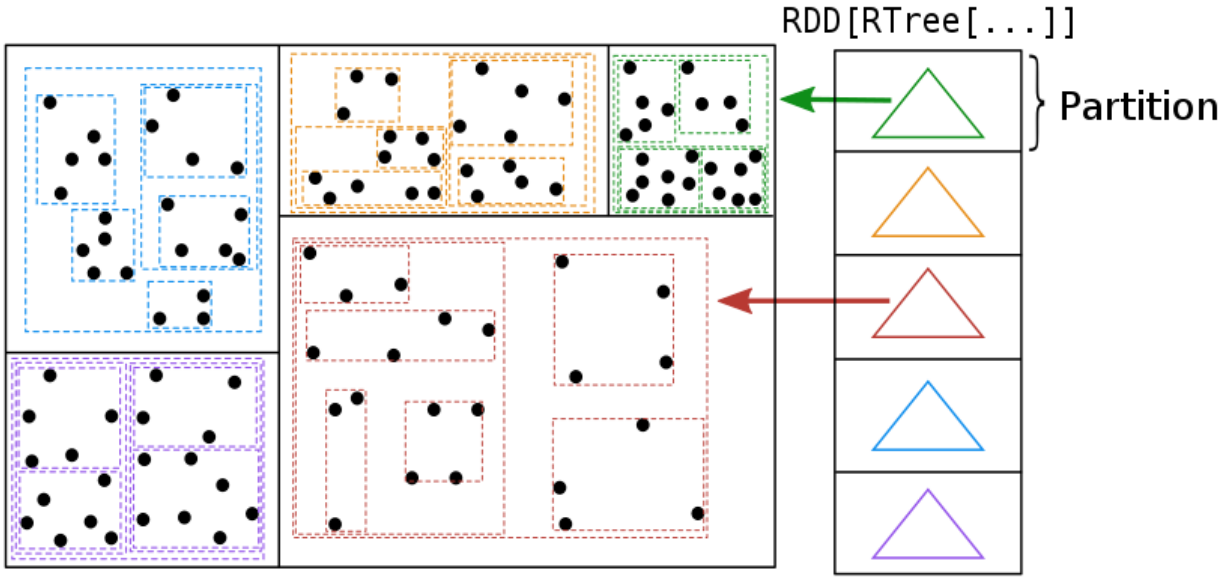
- Spatial join

```
res = SPATIAL_JOIN bag1, bag2 ON intersects(bag1.$1, bag2.$1);
```

- Currently for Spark only
  - Uses SpatialRDD (R-tree based) if available

SpatialRDD

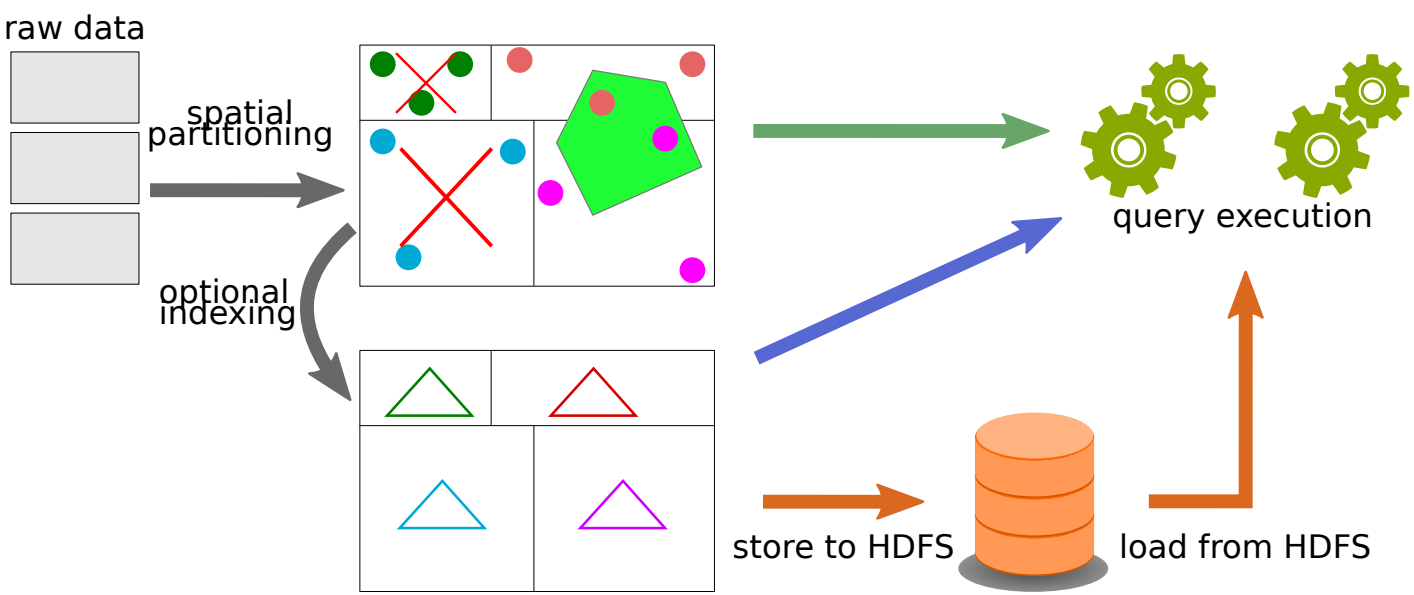
- Basic idea: use geo position as key for data partitioning to exploit locality of objects



- special type of Pair-RDD, i.e. RDD[(Geometry, V)]
  - provides spatial operations: intersection, contains, join
- seamless integration via Scala implicits to add spatial functions to RDD

Space Partitioning & Indexing

- Spatial partitioning by binary space or grid partitioning
  - partitions (internal represented as R tree) are used for partition pruning
  - can be materialized as index on disk and reused in other scripts



SPARQL Integration

- Use „triple bag“ format instead of plain RDF triples for efficient processing (avoid expensive self joins)

```
{ predicate: bytearray,
  stmts: { (subject: bytearray, object: bytearray) }
 }
```

- Two additional operators
  - TUPLIFY: produce triple bags from bag of triples
  - BGP_FILTER: filter operator with basic graph pattern (BGP)
    - can be implemented by joins or set of filters on triple bags

```
result = BGP_FILTER triples BY {
                    ?a <produced> ?record . ?a <country> ?country .
                    ?record <release> “2015” . };
```

Optimizations in Piglet

- Reordering, eliminating or substituting nodes (operators)
  - Pushdown of FILTER and FOREACH ...GENERATE
  - Combine WINDOW with CEP, aggregates and joins
  - join ordering
- Choose among different implementations of operators
  - Implementation of BGP_FILTER: sequence of joins vs. FOREACH + FILTER on triple bags
  - Use (or construct) SpatialRDD for spatial predicates
- Spark tuning
  - Level of parallelism
  - Persisting intermediate datasets
  - Indexing
  - ...

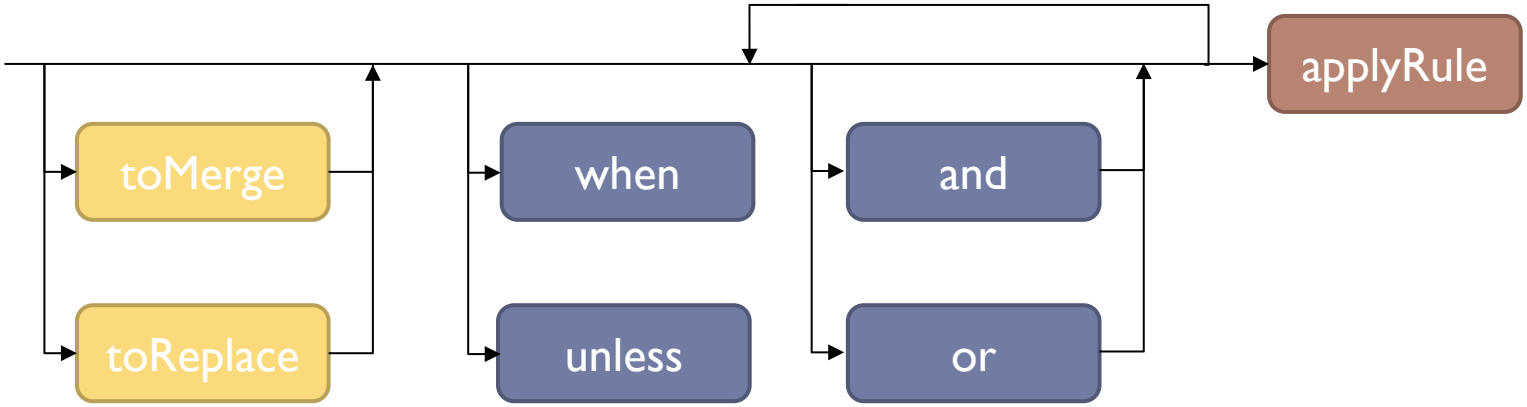
Most tasks can be implemented by transformation rules!

Rewriting in Piglet

- Integrating new operators and supporting new targets requires transformation of dataflow graphs
  - replace nodes by others (e.g., SPLIT INTO by multiple FILTERs, BGP_FILTER by multiple joins)
  - use special operator implementations (e.g. JOIN on WINDOW, spatial predicates in FILTER)
  - exploit algebraic properties of user-defined (black box) operators
  - exploit indexes (e.g. spatial indexes) and materialization

A DSL for Rewriting

- Scala-based DSL



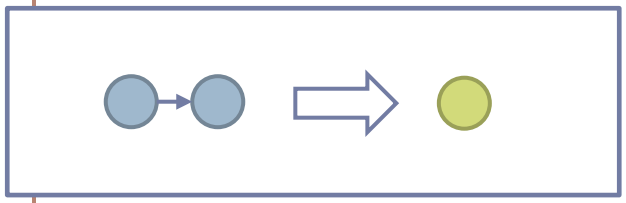

```
toMerge[Filter,Filter] when {
 case (f1, f2) if f1.predicate != f2.predicate =>
} applyRule {
  case (f1, f2) => Some(Filter(…, And(f1.predicate, f2.predicate)))
}
```

Rewriting Rules

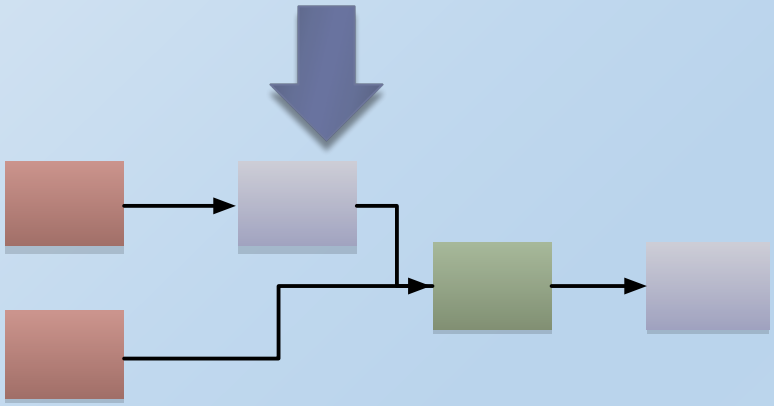
- Builtin rules as part of the dataflow compiler
  - e.g. SPLIT INTO → FILTER,
  - BGP_FILTER → JOIN, ...
- target-specific rules
  - e.g. handling of WINDOW for stream processing backends,
  - Exploiting SpatialRDD for spatial predicates in FILTER
- User-defined rules
  - e.g. for user-defined operators

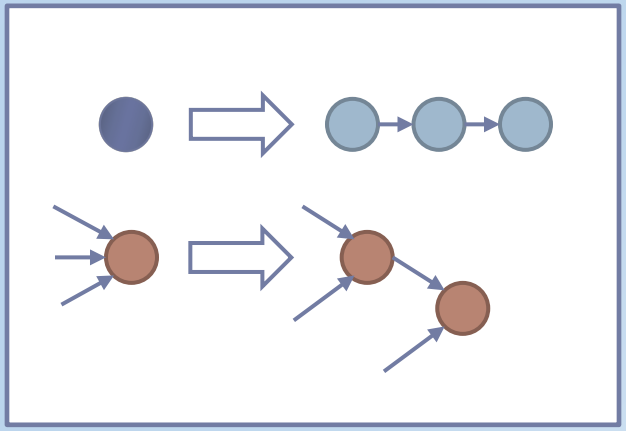
Extensible Rewriting

```
in = SOCKET_READ '*:8080' AS (...);
b = WINDOW in RANGE 1 HOUR;
c = GROUP b BY Lane;
out = FOREACH c
        GENERATE b.Lane, AVG(b.Speed);
```



user-provided rules





builtin rule set

Platform-specific code

Extensible Rewriting: Examples

- Adaptive spatial indexing

```
A = LOAD ....
B = FOREACH A GENERATE geometry($0) as geo ;
C = SPATIAL_FILTER B BY containedBy(geo, geometry("POLYGON((...))"));
```

Insert operator for creating and using index

```
A = LOAD ....
B = FOREACH A GENERATE geometry($0) as geo ;
X = INDEX B ON geo;
C = SPATIAL_FILTER X BY containedBy(geo, geometry("POLYGON((...))"));
```

- Possible reuse of index in other scripts ...

Extensible Rewriting: BGP_FILTER

- Mapping of BGP filter depends on structure of input data

```
result = BGP_FILTER inp BY  { ?s  wgs84:lat ?o1 . ?s wgs84:long ?o2 . };v
```

- Input = bag of plain triples ➠ self join

```
inp1 = FILTER inp BY predicate == == 'wgs84:lat';
inp2 = FILTER inp BY predicate == == 'wgs84:long';
result = JOIN inp1 BY (subject), inp2 BY (subject);
```

- Input = bag of tuples (bag of statements)

```
tmp = FOREACH inp {
        r1 = FILTER stmts BY predicate == 'wgs84:lat';
        r2 = FILTER stmts BY predicate == ' wgs84:long>';
        GENERATE *, COUNT(r1) AS cnt1, COUNT(r2) AS cnt2;
};
result = FILTER tmp BY cnt1 > 0 AND cnt2 > 0;
```

User-defined Operators

- Standard approach in Pig: STREAM THROUGH
  - but requires to implement a special API
- Embedded code simplifies wrapping of existing code
  - Example: integration of a DBSCAN implementation for Spark

```
REGISTER 'dbscan-0.1.jar';
<%
def clustering(sc: SparkContext, inp: RDD[List[Double]],
                        eps: Double, minPts: Int) = {
val input = inp.map(t =>.Vectors.dense(t(0), t(1)))
val op = new dbis.dbscan.DBScan().setEpsilon(eps).setMinPts(minPts)
op.run(sc, input) }
%>
data = LOAD 'points.csv' USING PigStorage(',') AS (c1:double, c2:double);
result = STREAM pdata THROUGH clustering(0.1, 122);
…
```

User-defined Operators

- Macros to encapsulate STREAM THROUGH

```
REGISTER 'dbscan-0.1.jar';
<%
def clustering(sc: SparkContext, inp: RDD[List[Double]],
                        eps: Double, minPts: Int) = {  … }
%>

DEFINE dbscan(in, eps, minPts) RETURNS out {
      out = STREAM in THROUGH clustering(eps, minPts);
};
```

```
IMPORT 'dbscan.pig';

data = LOAD 'points.csv' USING PigStorage(',') AS (c1:double, c2:double);
result = dbscan(data, 0.1, 122);
…
```

User-defined Rewriting Rules

- Define a transformation rule as embedded code

```
<%
@rule
toMerge[Filter,Filter] when {
 case (f1, f2) if f1.predicate != f2.predicate =>
} applyRule {
  case (f1, f2) => Some(Filter(… , And(f1.predicate, f2.predicate)))
}
%>
in = LOAD ‘file.csv’ USING PigStorage(‘,’);
out1 = FILTER in BY $1 > 0;
out2 = FILTER out1 BY $2 < 0;
```

- Code sections can be imported via IMPORT
- Allows special rewriting of user-defined operators

Piglet: Code Generation

- compiling Pig → Spark/Flink generally easier than Pig → MapReduce (esp. for Scala)
- For basic operators mostly 1:1 mapping of Pig to Spark operators via template engine
- main task: dealing with schemas in a type-safe and efficient way
  - generating Scala case class for each schema (=type)
  - with typed parameters representing fields
  - Scala's Option type allows to implement null values

Outline

- Big Dynamic Data: Use Cases & Requirements
- Foundations of Stream Processing
- APIs, Query & Dataflow Languages
- Piglet: Language & Compiler
- Conclusion & Outlook

Quelle: pixabay.com

Conclusion & Outlook

- Dynamic data becomes more and more important as big data
- Highly active research topic
  - Various systems, e.g. Spark, Flink, Kafka (Apache), Storm, Heron (Twitter), Beam, MillWheel (Google), ...
  - and commercial products, e.g. IBM InfoSphere Streams, Microsoft StreamInsight, ...
- But also a lot of existing work on stream processing
- New challenges: scalability, state management for large-scale systems, optimization of user-defined code, ...

Photo: Dolina Wiedzy CC BY-NC 2.0

Further Readings

- Historical papers on STREAM, Aurora, TelegraphCQ, Borealis, CQL, ...
- Papers and blogs on Storm, Heron, Flink, Spark Streaming, ...
- Windows & Semantics
  - Ghanem et al.: Incremental Evaluation of Sliding-Window Queries over Data Streams, TKDE 19(1), 2007
  - Tucker et al.: Exploiting Punctuation Semantics in Continuous Data Streams, TKDE 15(3), 2003
  - Krämer et al.: Sematics and Implementation of Continuous Sliding Window Queries over Data Streams, TODS 34(1), 2009
  - Botan et al.: SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems, VLDB 2010
- CEP:
  - Wu et al.: High-Performance Complex Event Processing over Streams, SIGMOD 2006
  - Schultz-Moeller et al.: Distributed Complex Event Processing with Query Rewriting, DEBS 2009
- Fault Tolerance:
  - Hwang et al.: High-availability algorithms for distributed stream processing, ICDE 2005
  - Zaharia et al.: Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. HotCloud, 2012.
  - Fernandez et al.: Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management, SIGMOD 2013
- Partitioning & Optimization:
  - Hirzel et al.: A Catalog of Stream Processing Optimizations, ACM Comp. Surveys 46(4), 2014.
  - Gedik et al.: Elastic Scaling for Data Streams, TPDS 25(6), 2014.
  - Viglas et al.: Rate-Based Query Optimization for Streaming Information Sources, SIGMOD 2002