Advanced topics in Apache Flink™

Maximilian Michels
mxm@apache.org
@stadtlegende

Ufuk Celebi
uce@apache.org
@iamuce

EIT ICT Summer School 2015

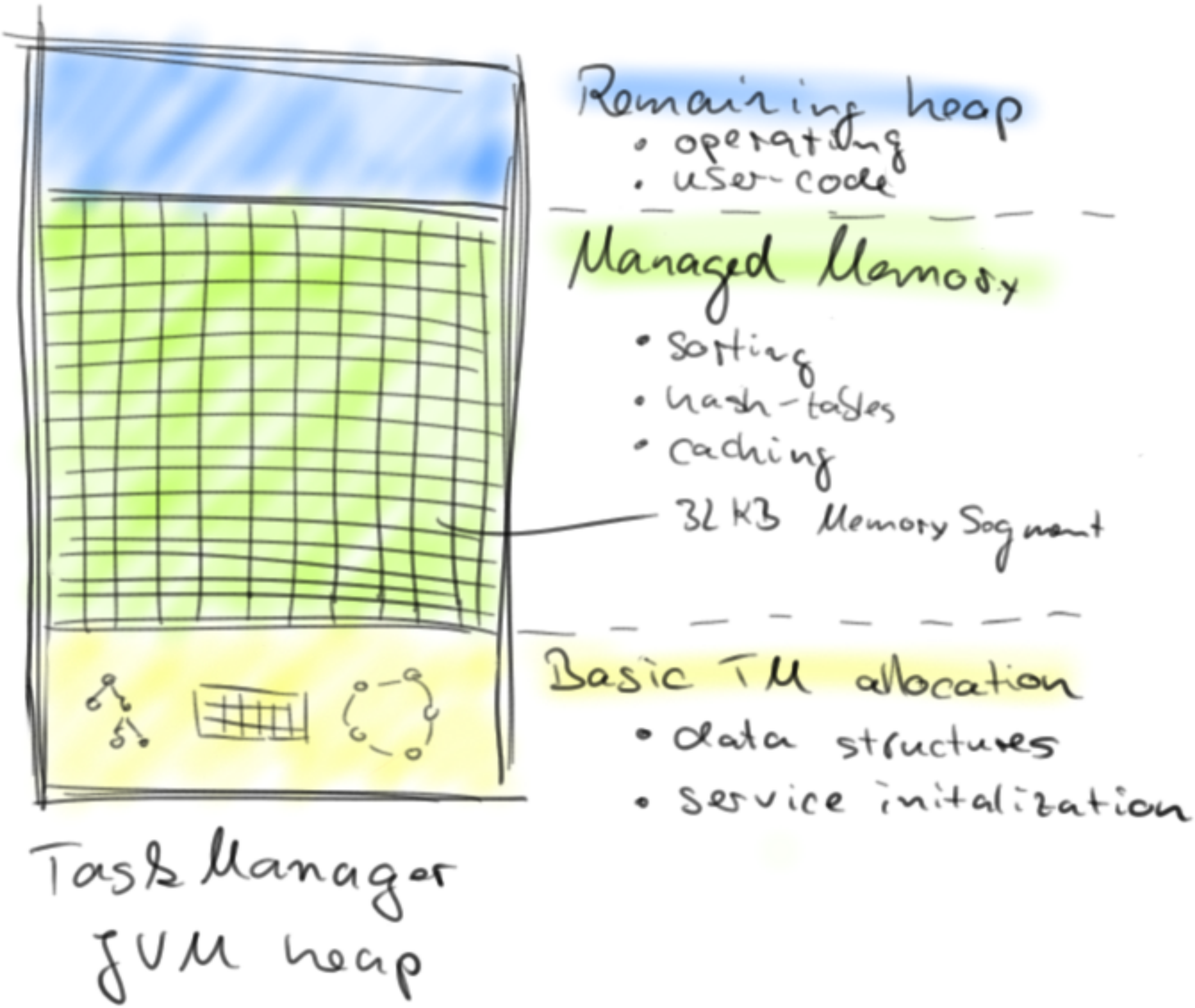
dataArtisans

Agenda

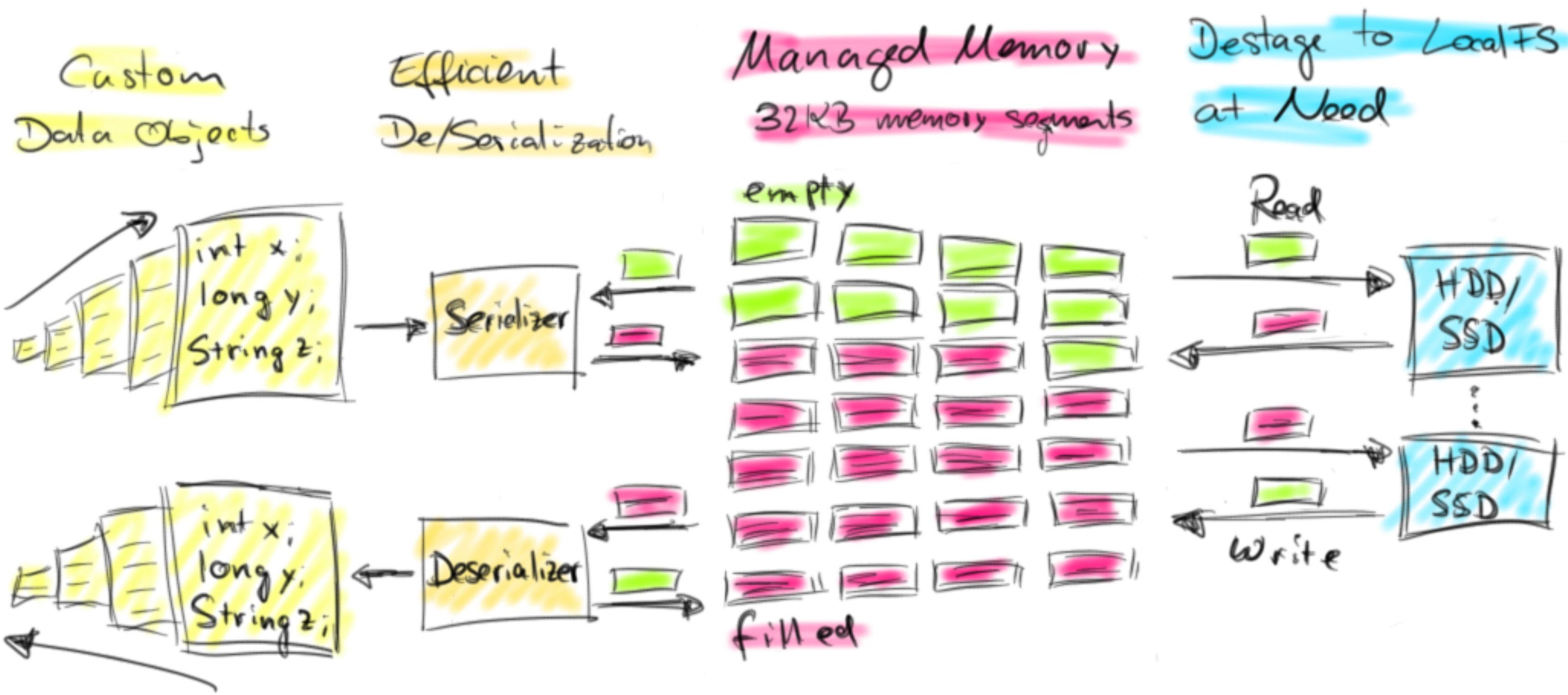
- Batch analytics
- Iterative processing
- Fault tolerance
- Data types and keys
- More transformations
- Further API concepts

Batch analytics

Memory Management

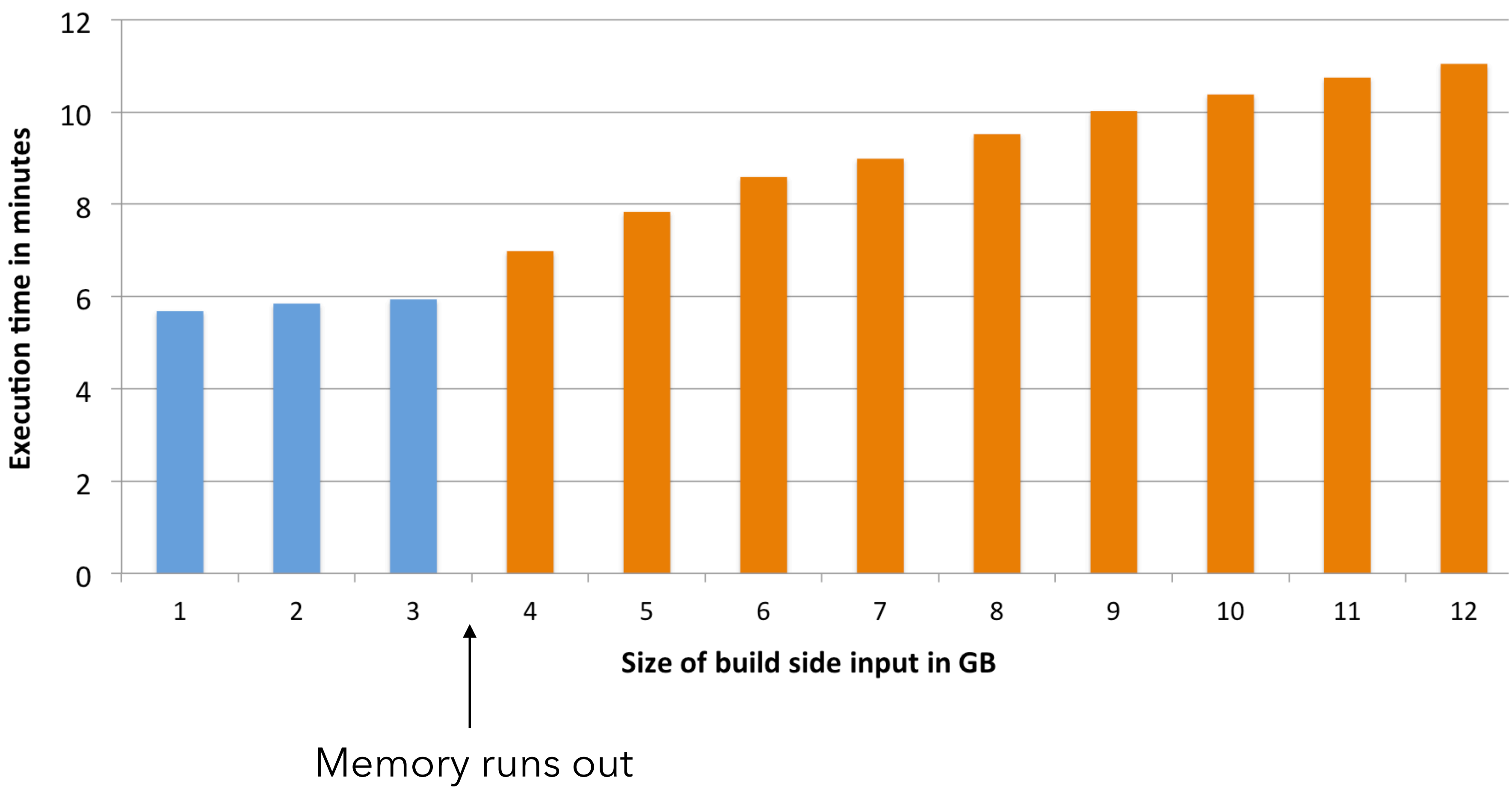


Memory Management



Managed memory in Flink

Hash Join
Probe side: 64GB



See also: http://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html

Cost-based optimizer

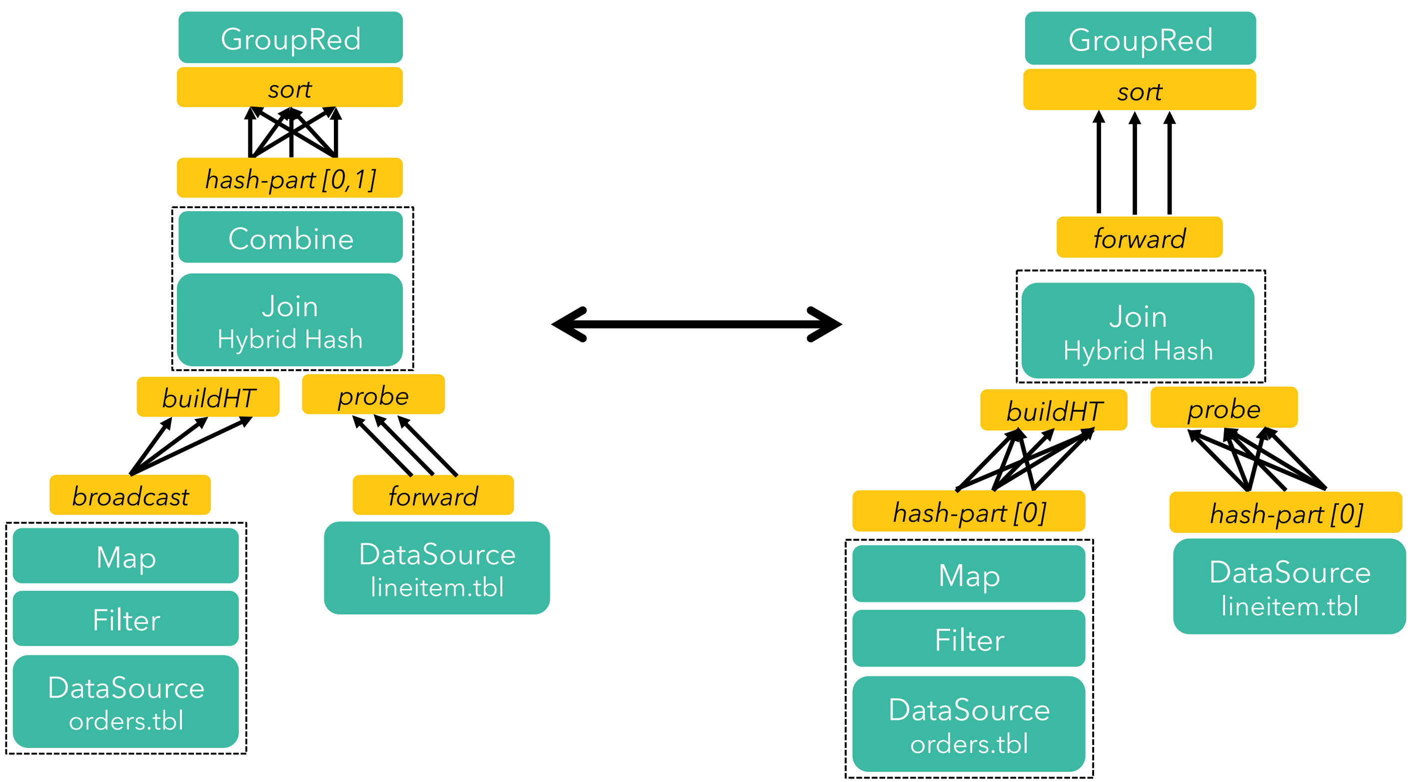
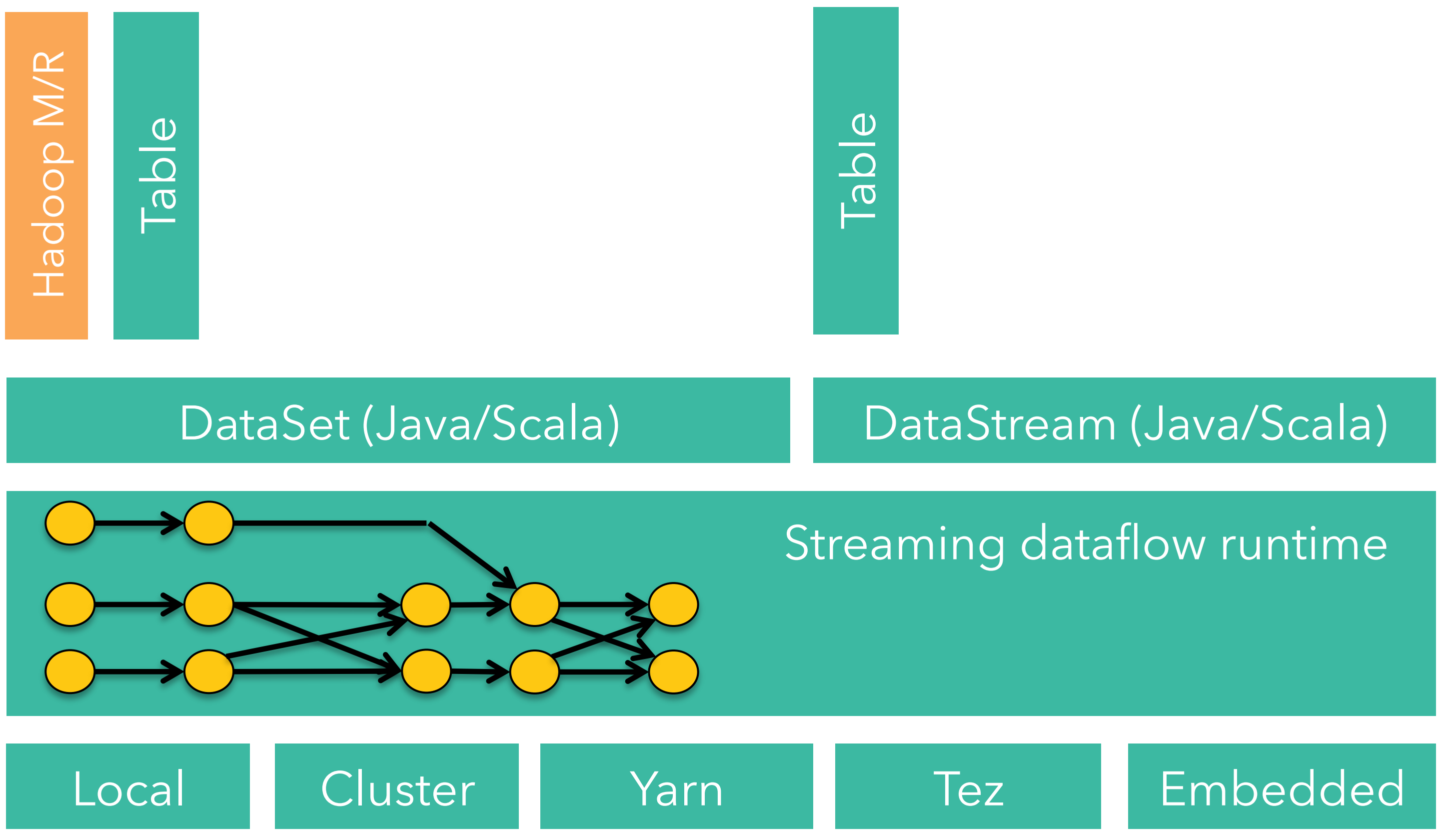


Table API

```
val customers = env.readCsvFile(…).as('id, 'mktSegment)
      .filter("mktSegment = AUTOMOBILE")

val orders = env.readCsvFile(…)
      .filter( o => dateFormat.parse(o.orderDate).before(date) )
      .as("orderId, custId, orderDate, shipPrio")

val items = orders
      .join(customers).where("custId = id")
      .join(lineitems).where("orderId = id")
      .select("orderId, orderDate, shipPrio,
          extdPrice * (Literal(1.0f) - discount) as revenue")

val result = items
      .groupBy("orderId, orderDate,  shipPrio")
      .select('orderId, revenue.sum, orderDate, shipPrio")
```

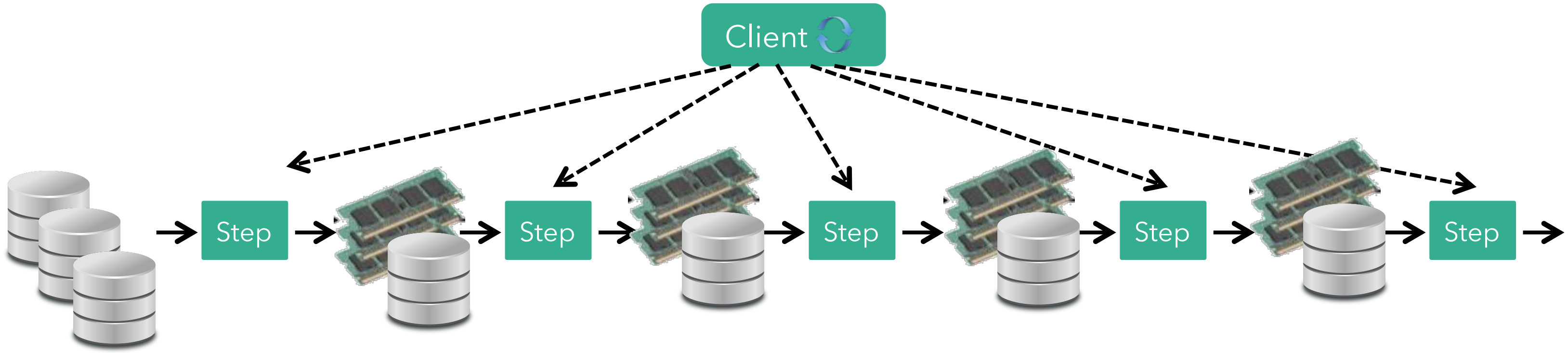
Flink stack


Hadoop M/R
Table
Table
DataSet (Java/Scala)
DataStream (Java/Scala)
Streaming dataflow runtime
Local
Cluster
Yarn
Tez
Embedded

Iterative processing

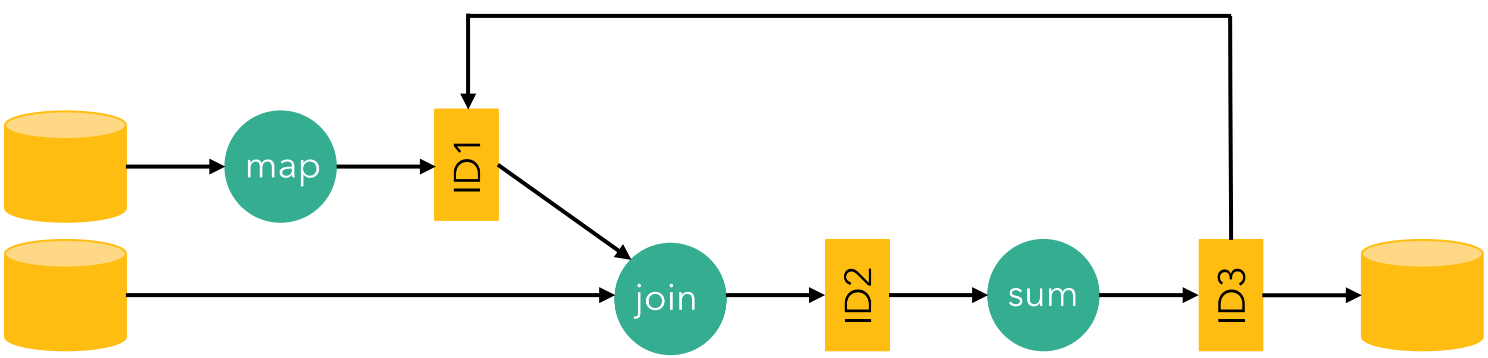
Non-native iterations

```
for (int i = 0; i < maxIterations; i++) {
    // Execute MapReduce job
}
```



Iterative processing in Flink

Flink offers **built-in** iterations and **delta iterations** to execute ML and graph algorithms efficiently.



FlinkML

- API for **ML pipelines** inspired by *scikit-learn*
- Collection of packaged algorithms
  - SVM, Multiple Linear Regression, Optimization, ALS, ...

```
val trainingData: DataSet[LabeledVector] = ...
val testingData: DataSet[Vector] = ...

val scaler = StandardScaler()
val polyFeatures = PolynomialFeatures().setDegree(3)
val mlr = MultipleLinearRegression()

val pipeline = scaler.chainTransformer(polyFeatures).chainPredictor(mlr)

pipeline.fit(trainingData)

val predictions: DataSet[LabeledVector] = pipeline.predict(testingData)
```

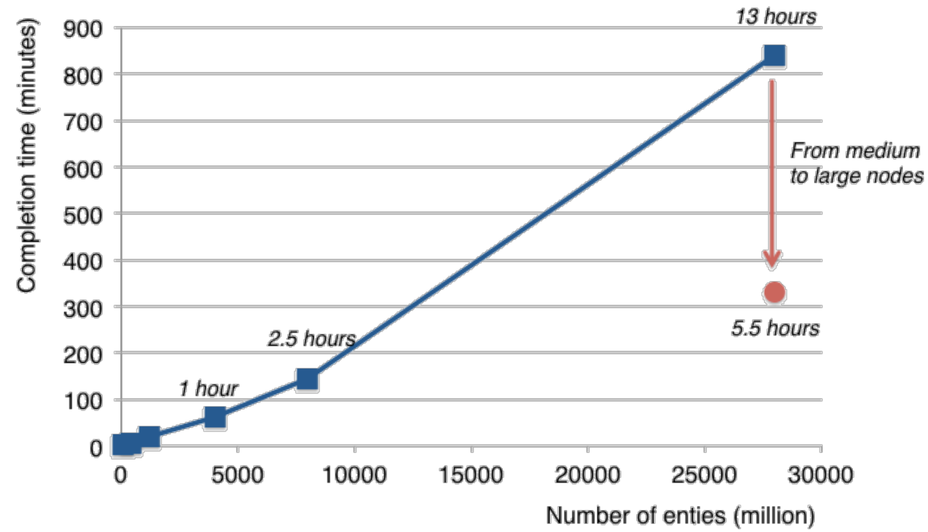
Gelly

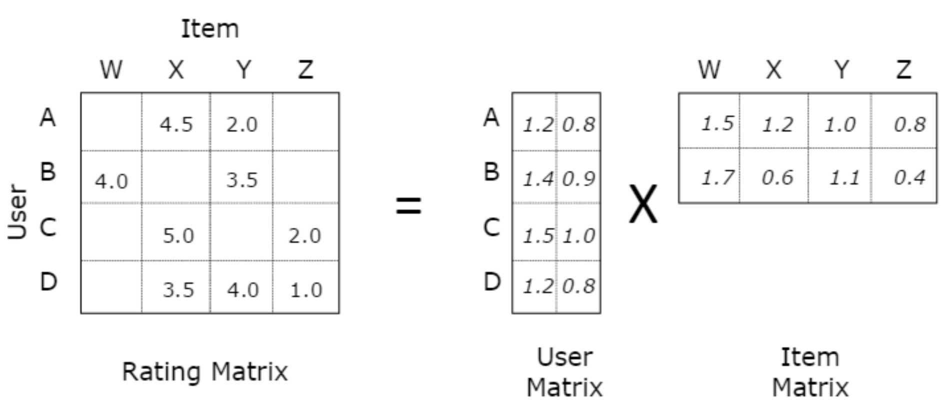
- **Graph** API: various graph processing paradigms
- Packaged algorithms
  - PageRank, SSSP, Label Propagation, Community Detection, Connected Components

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

Graph<Long, Long, NullValue> graph = ...

DataSet<Vertex<Long, Long>> verticesWithCommunity = graph.run(
                    new LabelPropagation<Long>(30)).getVertices();

verticesWithCommunity.print();

env.execute();
```

Example: Matrix Factorization

Factorizing a matrix with 28 billion ratings for recommendations





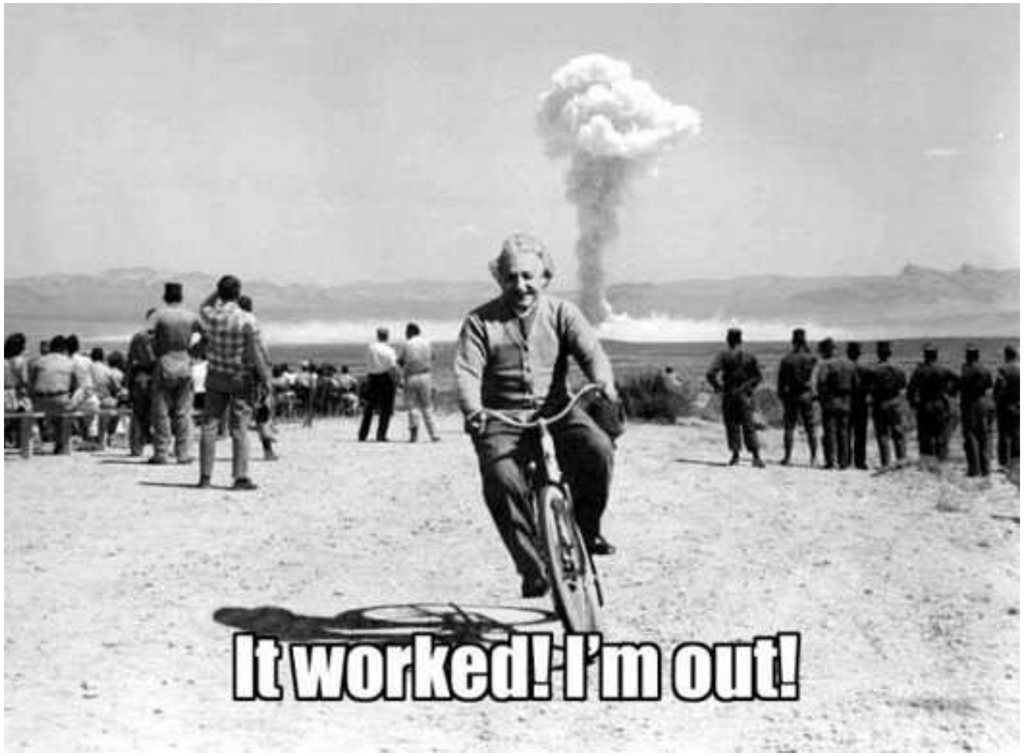
More at: http://data-artisans.com/computing-recommendations-with-flink.html

Fault tolerance

Why be fault tolerant?

- Failures are rare **but** they occur
- The larger the cluster, the more likely failures

- Types of failures
  - Hardware
  - Software



Recovery strategies

Batch

- Simple strategy: restart the job/tasks
- Resume from partially crated intermediate results or re-read and process the entire input again

Streaming

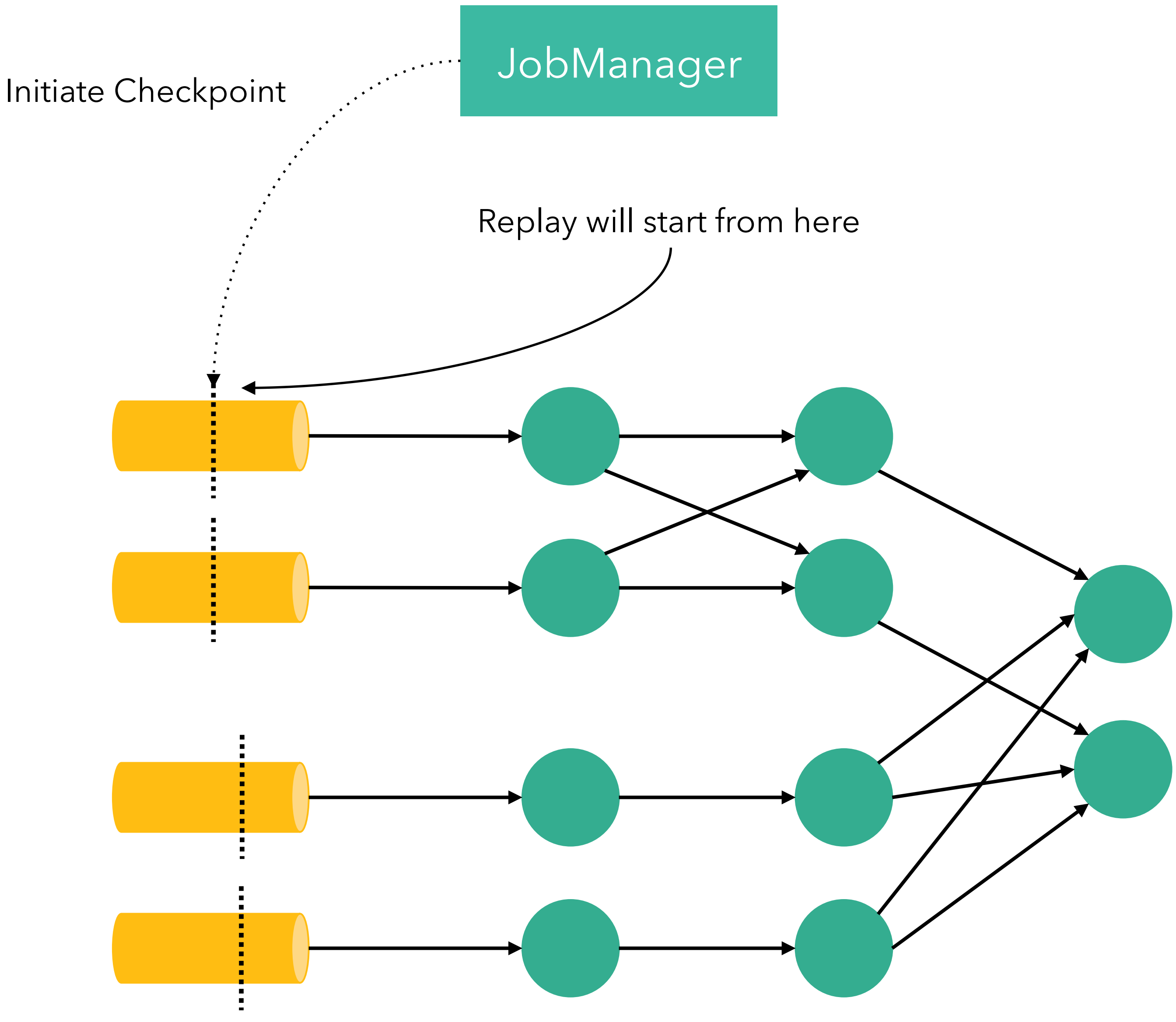
- Simple strategy: restart job/tasks
- We loose the state of the operators
- Goal: find the correct offset to resume from

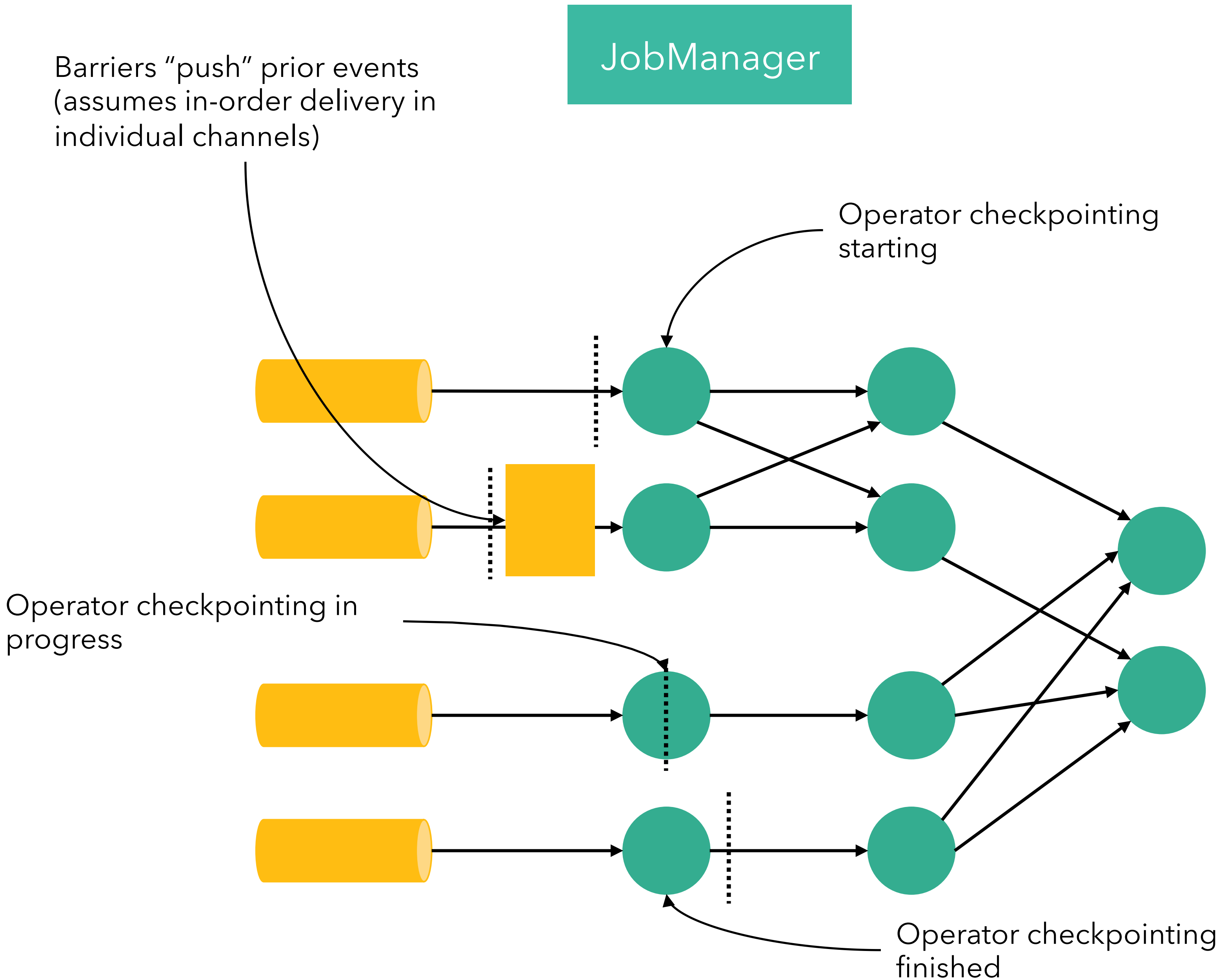
Streaming fault tolerance

- Ensure that operators see all events
  - "At least once"
  - Solved by replaying a stream from a checkpoint, e.g., from a past Kafka offset
- Ensure that operators do not perform duplicate updates to their state
  - "Exactly once"
  - Several solutions

Exactly once approaches

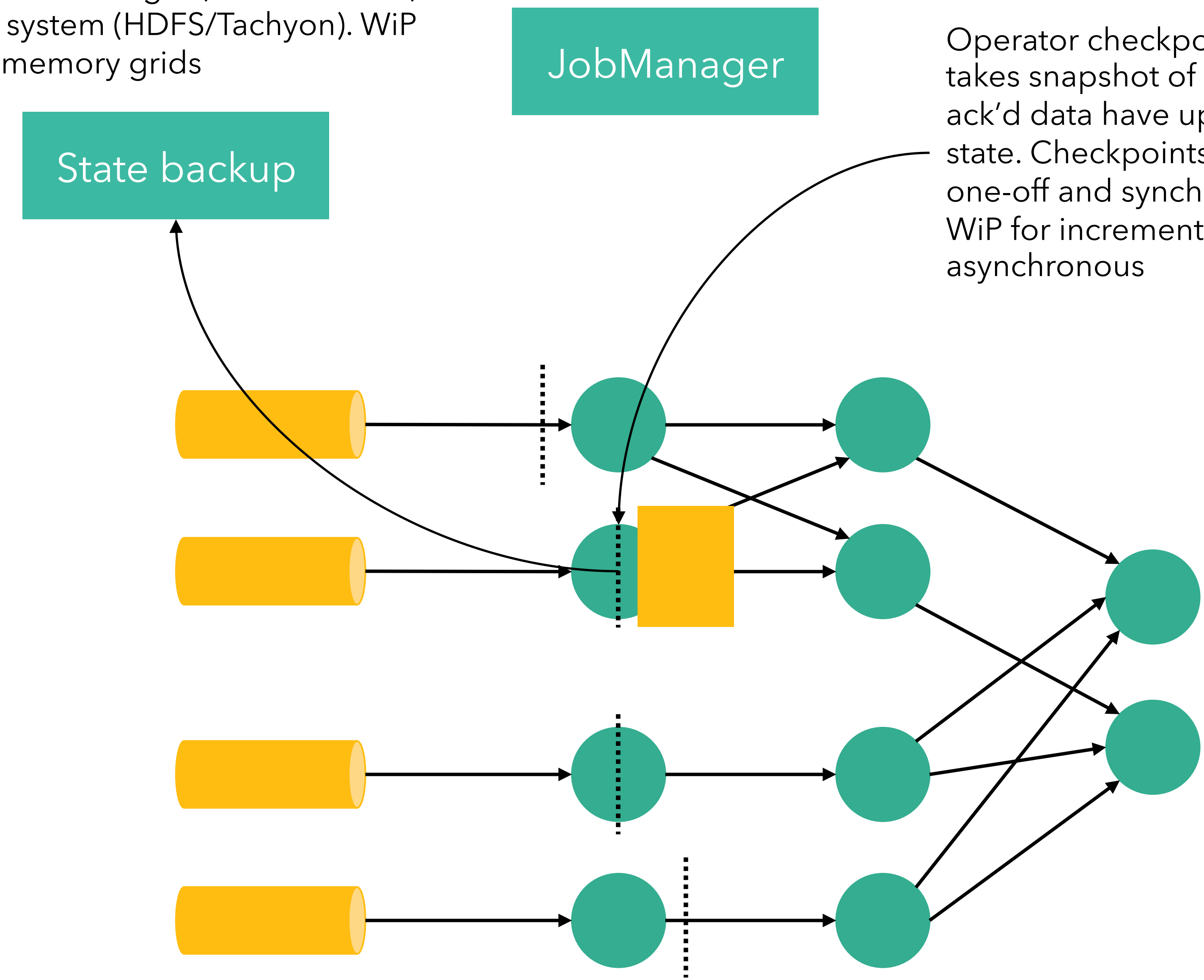
- Discretized streams (Spark Streaming)
  - Treat streaming as a series of small atomic computations
  - "Fast track" to fault tolerance, but restricts computational and programming model (e.g., cannot mutate state across "mini-batches", window functions correlated with mini-batch size)
- MillWheel (Google Cloud Dataflow)
  - State update and derived events committed as atomic transaction to a high-throughput transactional store
  - Requires a very high-throughput transactional store ☺
- Chandy-Lamport distributed snapshots (Flink)

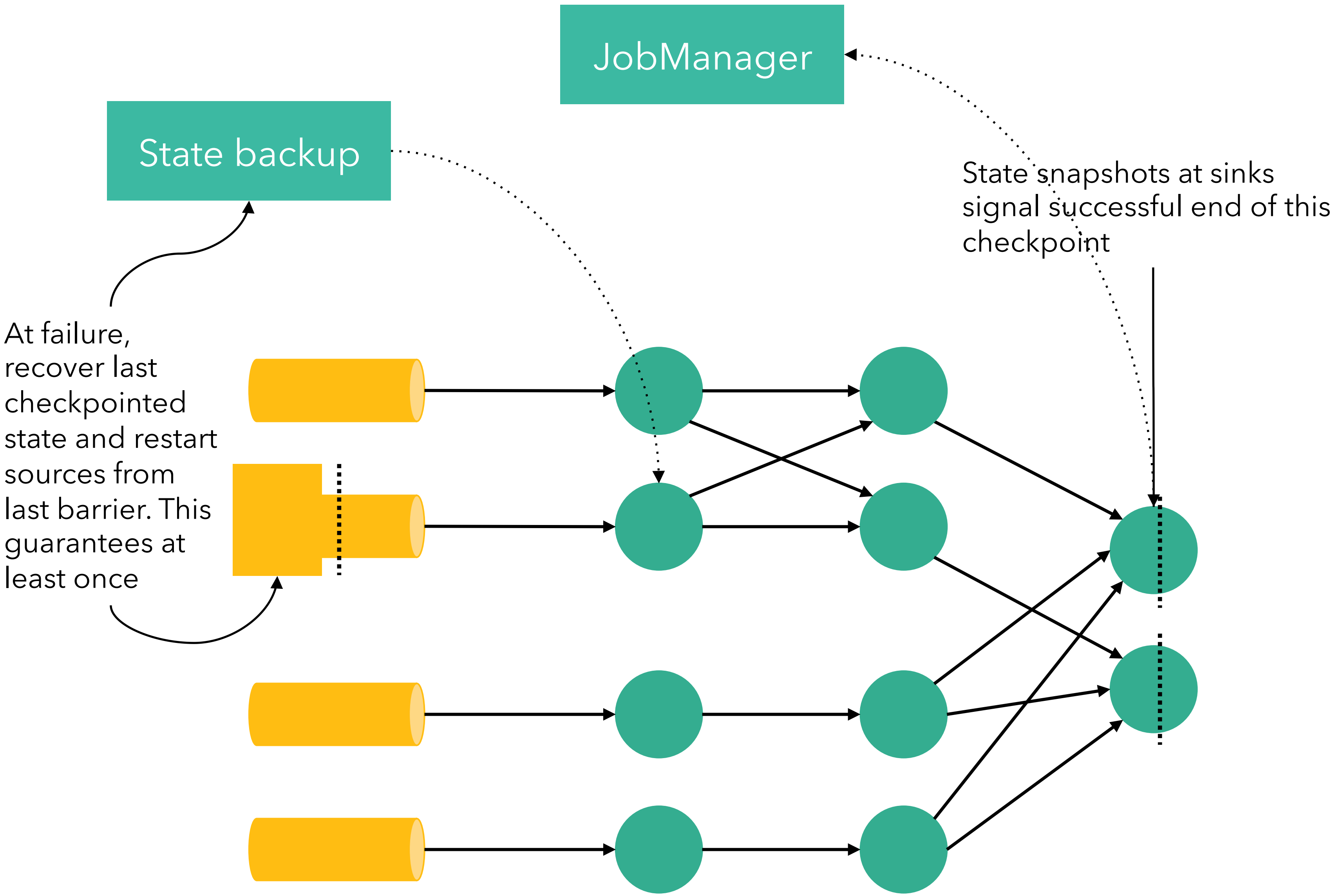

JobManager
Initiate Checkpoint
Replay will start from here


JobManager
Barriers “push” prior events (assumes in-order delivery in individual channels)
Operator checkpointing starting
Operator checkpointing in progress
Operator checkpointing finished

Pluggable mechanism. Currently either JobManager (for small state) or file system (HDFS/Tachyon). WiP for in-memory grids

Operator checkpointing takes snapshot of state after ack'd data have updated the state. Checkpoints currently one-off and synchronous, WiP for incremental and asynchronous




JobManager
State backup
State snapshots at sinks signal successful end of this checkpoint
At failure, recover last checkpointed state and restart sources from last barrier. This guarantees at least once

Best of all worlds for streaming

- Low latency
  - Thanks to pipelined engine
- Exactly-once guarantees
  - Variation of Chandy-Lamport
- High throughput
  - Controllable checkpointing overhead
- Separates app logic from recovery
  - Checkpointing interval is just a config parameter

Fault Tolerance Demo

What kind of data can Flink handle?

Type System and Keys

Apache Flink's Type System

- Flink aims to support all data types
  - Ease of programming
  - Seamless integration with existing code

- Programs are analyzed before execution
  - Used data types are identified
  - Serializer & comparator are configured

Apache Flink's Type System

- Data types are either
  - Atomic types (like Java Primitives)
  - Composite types (like Flink Tuples)

- Composite types nest other types

- Not all data types can be used as keys!
  - Flink groups, joins & sorts DataSets on keys
  - Key types must be comparable

Atomic Types

Flink Type	Java Type	Can be used as key?
BasicType	Java Primitives (Integer, String, …)	Yes
ArrayType	Arrays of Java primitives or objects	No
WritableType	Implements Hadoop's Writable interface	Yes, if implements WritableComparable
GenericType	Any other type	Yes, if implements Comparable

Composite Types

- Are composed of fields with other types
  - Fields types can be atomic or composite
- Fields can be addressed as keys
  - Field type must be a key type!
- A composite type can be a key type
  - All field types must be key types!

PojoType

- Any Java class that
  - Has an empty default constructor
  - Has publicly accessible fields
    (Public or getter/setter)

```
public class Person {
  public int id;
  public String name;
  public Person() {};
  public Person(int id, String name) {…};
}

DataSet<Person> p =
  env.fromElements(new Person(1, ”Bob”));
```

PojoType

- Define keys by field name

```
DataSet<Person> p = …
// group on “name” field
d.groupBy(“name”).groupReduce(…);
```

Scala CaseClasses

- Scala case classes are natively supported

```
case class Person(id: Int, name: String)
d: DataSet[Person] =
     env.fromElements(new Person(1, “Bob”))
```

- Define keys by field name

```
// use field “name” as key
d.groupBy(“name”).groupReduce(…)
```

Composite & nested keys

```
DataSet<Tuple3<String, Person, Double>> d = …
```

- Composite keys are supported

```
// group on both long fields
d.groupBy(0, 1).reduceGroup(…);
```

- Nested fields can be used as types

```
// group on nested “name” field
d.groupBy(“f1.name”).reduceGroup(…);
```

- Full types can be used as key using “*” wildcard

```
// group on complete nested Pojo field
d.groupBy(“f1.*”).reduceGroup(…);
```

  - “*” wildcard can also be used for atomic types

Join & CoGroup Keys

- Key types must match for binary operations!

```
DataSet<Tuple2<Long, String>> d1 = …
DataSet<Tuple2<Long, Long>> d2 = …

// works
d1.join(d2).where(0).equalTo(1).with(…);
// works
d1.join(d2).where("f0").equalTo(0).with(…);
// does not work!
d1.join(d2).where(1).equalTo(0).with(…);
```

KeySelectors

- Keys can be computed using KeySelectors

```
public class SumKeySelector implements
        KeySelector<Tuple2<Long, Long>, Long> {

  public Long getKey(Tuple2<Long, Long> t) {
    return t.f0 + t.f1;
 }}

DataSet<Tuple2<Long,Long>> d = …
d.groupBy(new SumKeySelector()).reduceGroup(…);
```

Getting data in and out

Advanced Sources and Sinks

Supported File Systems

- Flink build-in File Systems:
  - LocalFileSystem (file://)
  - Hadoop Distributed File System (hdfs://)
  - Amazon S3 (s3://)
  - MapR FS (maprfs://)

- Support for all Hadoop File Systems
  - NFS, Tachyon, FTP, har (Hadoop Archive), …

Input/Output Formats

- FileInputFormat
  (recursive directory scans supported)
  - DelimitedInputFormat
    - TextInputFormat (Reads text files linewise)
    - CsvInputFormat (Reads field delimited files)
  - BinaryInputFormat
  - AvroInputFormat (Reads Avro POJOs)
- JDBCInputFormat (Reads result of SQL query)
- HadoopInputFormat
  (Wraps any Hadoop InputFormat)

Hadoop Input/OutputFormats

- Support for all Hadoop I/OFormats

- Read from and write to
  - MongoDB
  - Apache Parquet
  - Apache ORC
  - Apache Kafka (for batch)
  - Compressed file formats (.gz, .zip, ...)
  - and more…

Using InputFormats

```
ExecutionEnvironment env = …

// read text file linewise
env.readTextFile(…);

// read CSV file
env.readCsvFile(…);

// read file with Hadoop FileInputFormat
env.readHadoopFile(…);

// use regular Hadoop InputFormat
env.createHadoopInput(…);

// use regular Flink InputFormat
env.createInput(…);
```

Transformations & Functions

Transformations

- DataSet Basics presented:
  - Map, FlatMap, GroupBy, GroupReduce, Join

- Reduce
- CoGroup
- Combine
- GroupSort
- AllReduce & AllGroupReduce
- Union

- see documentation for more transformations

GroupReduce (Hadoop-style)

- GroupReduceFunction gives iterator over elements of group
  - Elements are streamed (possibly from disk), not materialized in memory
  - Group size can exceed available JVM heap

- Input type and output type may be different

Reduce (FP-style)

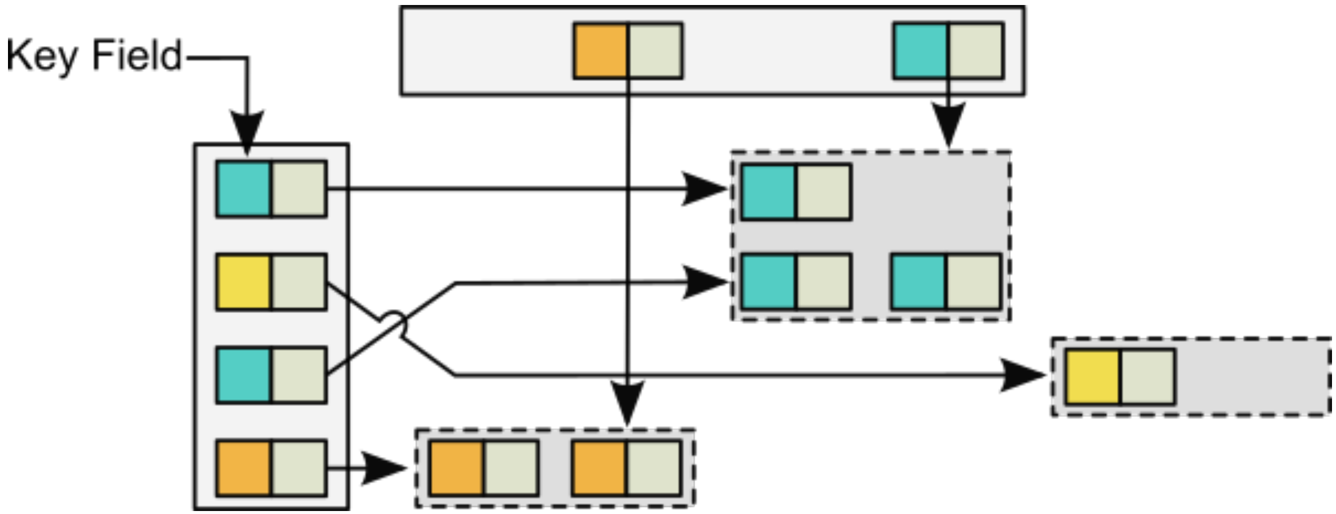
- Reduce like in functional programming
- Less generic compared to GroupReduce
  - Function must be commutative and associative
  - Input type == Output type
- System can apply more optimizations
  - Always combinable
  - May use a hash strategy for execution (future)

Reduce (FP-style)

```
DataSet<Tuple2<Long,Long>> sum = data
    .groupBy(0)
    .reduce(new SumReducer());

public static class SumReducer implements
              ReduceFunction<Tuple2<Long,Long>> {

  @Override
  public Tuple2<Long,Long> reduce(
                        Tuple2<Long,Long> v1,
                        Tuple2<Long,Long> v2) {
      v1.f1 += v2.f1;
     return v1;
   }
}
```

CoGroup

- Binary operation (two inputs)
  - Groups both inputs on a key
  - Processes groups with matching keys of both inputs
- Similar to GroupReduce on two inputs



CoGroup

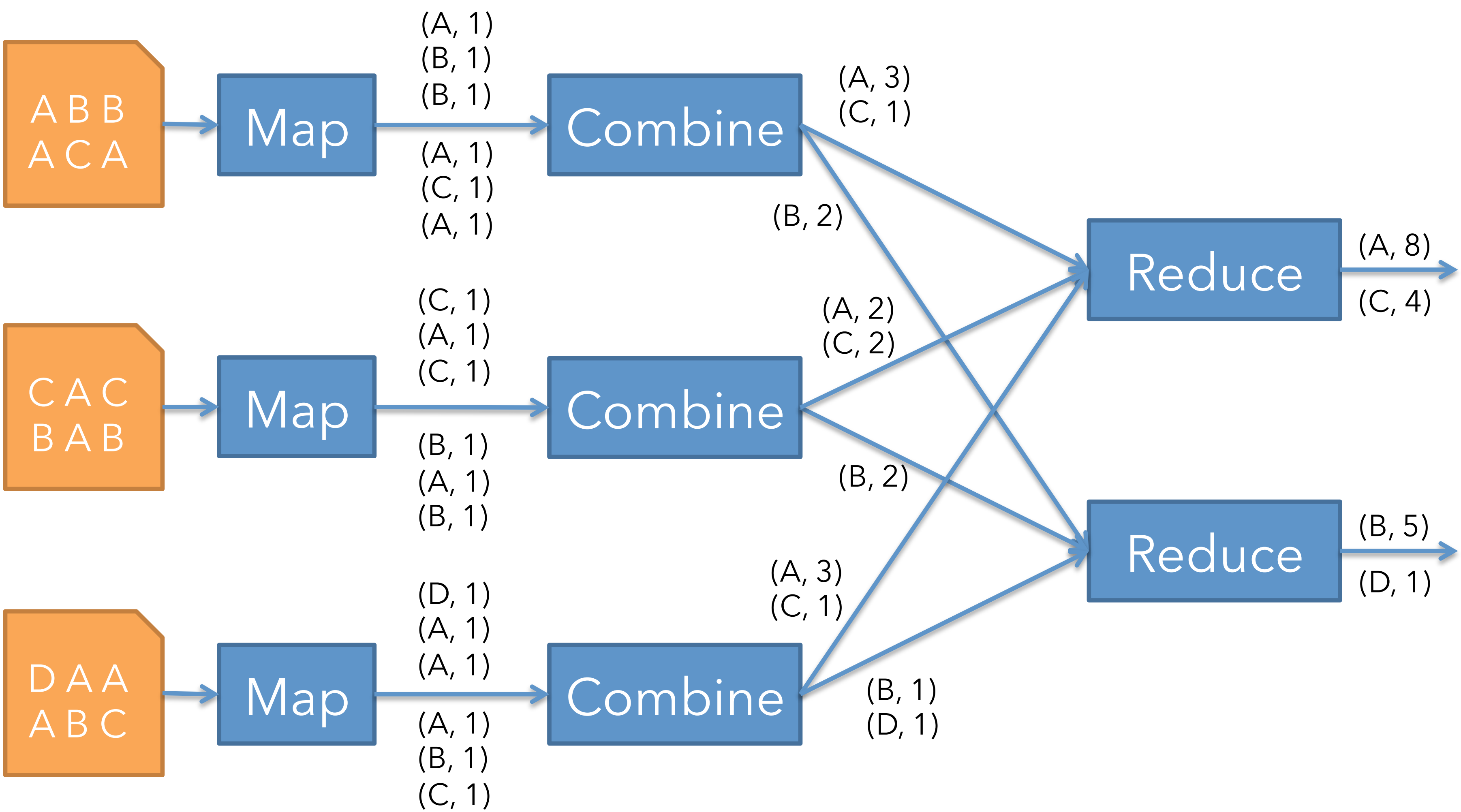
```
DataSet<Tuple2<Long,String>> d1 = …;
DataSet<Long> d2 = …;
DataSet<String> d3 =
   d1.coGroup(d2).where(0).equalTo(1).with(new CoGrouper());

public static class CoGrouper implements
CoGroupFunction<Tuple2<Long,String>,Long,String>{

  @Override
  public void coGroup(Iterable<Tuple2<Long,String> vs1,
                Iterable<Long> vs2, Collector<String> out) {
    if(!vs2.iterator.hasNext()) {
      for(Tuple2<Long,String> v1 : vs1) {
        out.collect(v1.f1);
      }
    }
} }
```

Combiner

- Local pre-aggregation of data
  - Before data is sent to GroupReduce or CoGroup
  - (functional) Reduce injects combiner automatically
  - Similar to Hadoop Combiner

- Optional for semantics, crucial for performance!
  - Reduces data before it is sent over the network

Combiner WordCount Example



Use a combiner

- Implement `RichGroupReduceFunction<I,O>`
  - Override `combine(Iterable<I> in, Collector<O>);`
  - Same interface as `reduce()` method
  - Annotate your `GroupReduceFunction` with `@Combinable`
  - Combiner will be automatically injected into Flink program

- Implement a `GroupCombineFunction`
  - Same interface as `GroupReduceFunction`
  - `DataSet.combineGroup()`

GroupSort

- Sort groups before they are handed to GroupReduce or CoGroup functions
  - More (resource-)efficient user code
  - Easier user code implementation
  - Comes (almost) for free
  - Aka secondary sort (Hadoop)

```
DataSet<Tuple3<Long,Long,Long> data = …;

data.groupBy(0)
    .sortGroup(1, Order.ASCENDING)
    .groupReduce(new MyReducer());
```

AllReduce / AllGroupReduce

- Reduce / GroupReduce without GroupBy
  - Operates on a single group -> Full DataSet
  - Full DataSet is sent to one machine
  - Will automatically run with parallelism of 1

- Careful with large DataSets!
  - Make sure you have a Combiner

Union

- Union two data sets
  - Binary operation, same data type required
  - No duplicate elimination (SQL UNION ALL)
  - Very cheap operation

```
DataSet<Tuple2<String, Long> d1 = …;
DataSet<Tuple2<String, Long> d2 = …;

DataSet<Tuple2<String, Long> d3 =
   d1.union(d2);
```

RichFunctions

- Function interfaces have only one method
  - Single abstract method (SAM)
  - Support for Java8 Lambda functions
- There is a “Rich” variant for each function.
  - RichFlatMapFunction, …
  - Additional methods
    - `open(Configuration c)`
    - `close()`
    - `getRuntimeContext()`

RichFunctions & RuntimeContext

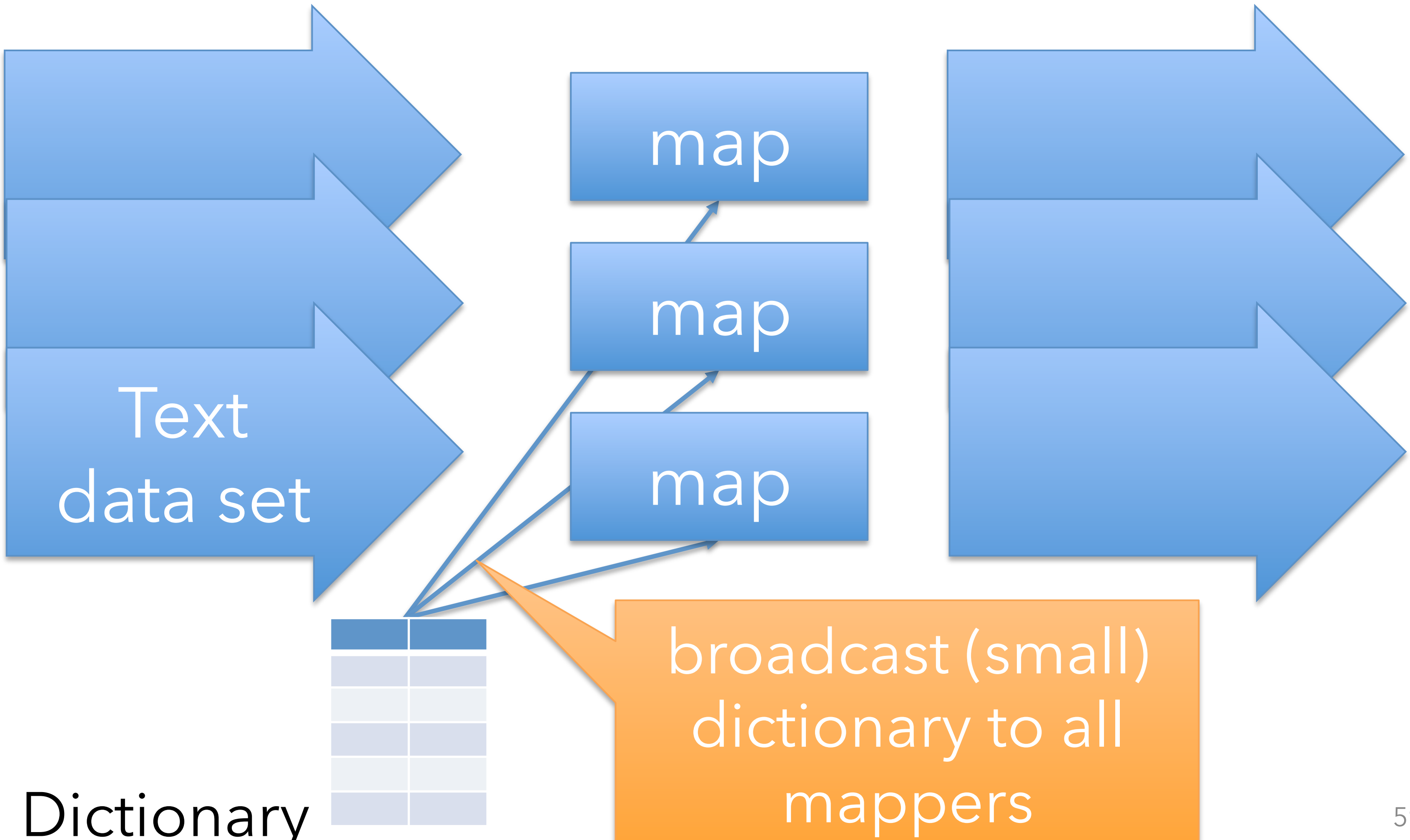
- RuntimeContext has useful methods:
  - `getIndexOfThisSubtask ()`
  - `getNumberOfParallelSubtasks()`
  - `getExecutionConfig()`

- Gives access to:
  - Accumulators
  - DistributedCache

Further API Concepts

Broadcast Variables

Example: Tag words with IDs in text corpus



Broadcast variables

- register any DataSet as a broadcast variable
- available on all parallel instances

```
// 1. The DataSet to be broadcasted
DataSet<Integer> toBroadcast = env.fromElements(1, 2, 3);

// 2. Broadcast the DataSet
map().withBroadcastSet(toBroadcast, "broadcastSetName");

// 3. inside user defined function
getRuntimeContext().getBroadcastVariable("broadcastSetName");
```

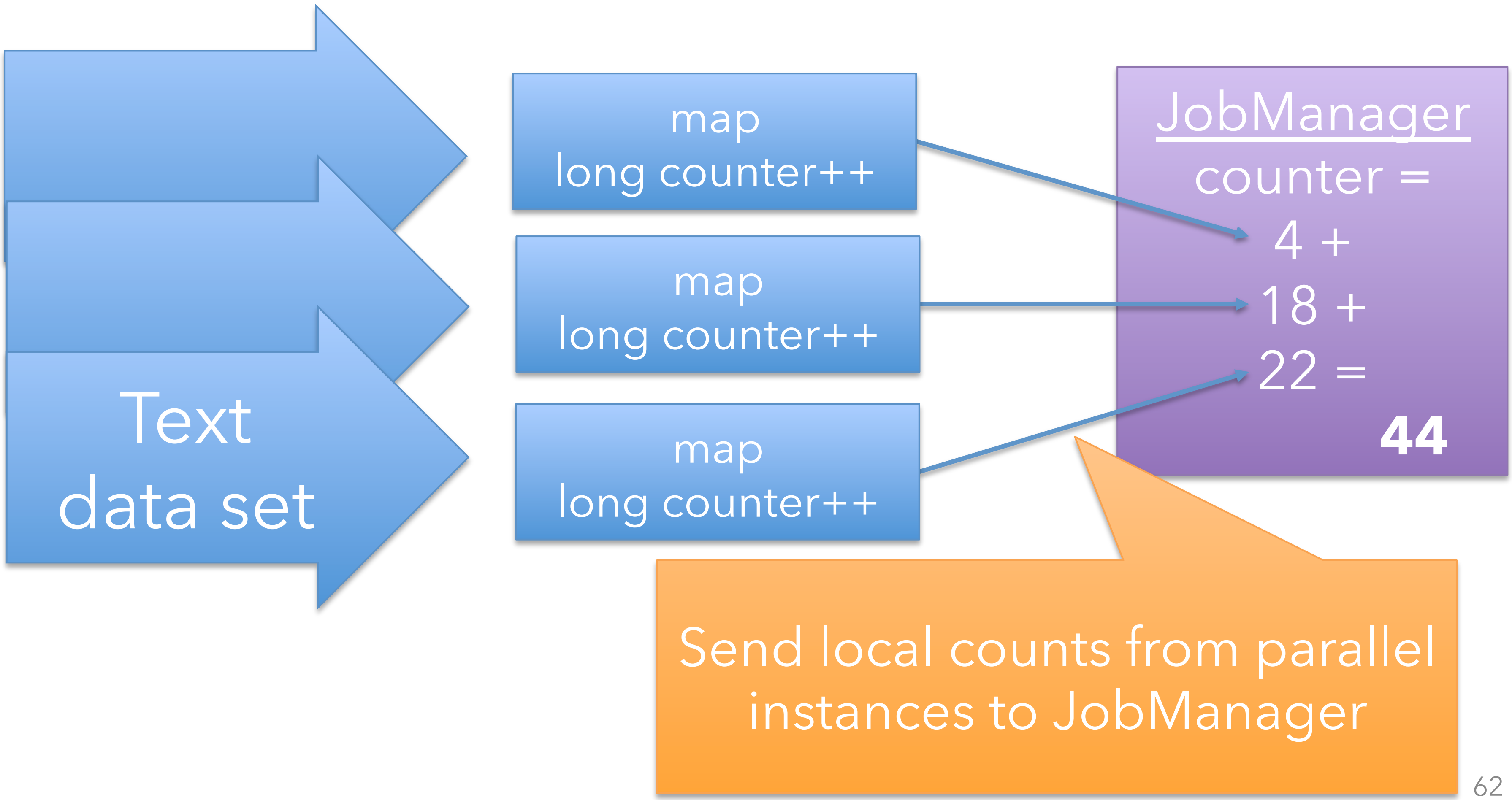
Accumulators

- Lightweight tool to compute stats on data
  - Useful to verify your assumptions about your data
  - Similar to Counters (Hadoop MapReduce)
- Build in accumulators
  - Int and Long counters
  - Histogramm
- Easily customizable

Accumulators

Example:

Count total number of words in text corpus



Using Accumulators

- Use accumulators to verify your assumptions about the data

```
class Tokenizer extends
            RichFlatMapFunction<String, String>> {

  @Override
  public void flatMap(String val,
                      Collector<String> out) {
      getRuntimeContext()
        .getLongCounter("elementCount").add(1L);
     // do more  stuff.
   }
}
```

Get Accumulator Results

- Accumulators are available via JobExecutionResult
  - returned by env.execute()

```
JobExecutionResult result = env.execute("WordCount");
long ec = result.getAccumulatorResult("elementCount");
```

- Accumulators are displayed
  - by CLI client
  - in the JobManager web frontend

Closing

I ♥ , do you?

- Get involved and start a discussion on Flink's mailing list
- { user, dev }@flink.apache.org

- Subscribe to news@flink.apache.org
- Follow flink.apache.org/blog and @ApacheFlink on Twitter

Flink Forward

BERLIN 12/13 OCT 2015

flink-forward.org

October 12-13, 2015

Call for papers deadline:
August 14, 2015

Discount code: FlinkEITSummerSchool25

Thank you for listening!

Flink compared to other projects

Batch & Streaming projects

Batch only

Streaming only

Hybrid

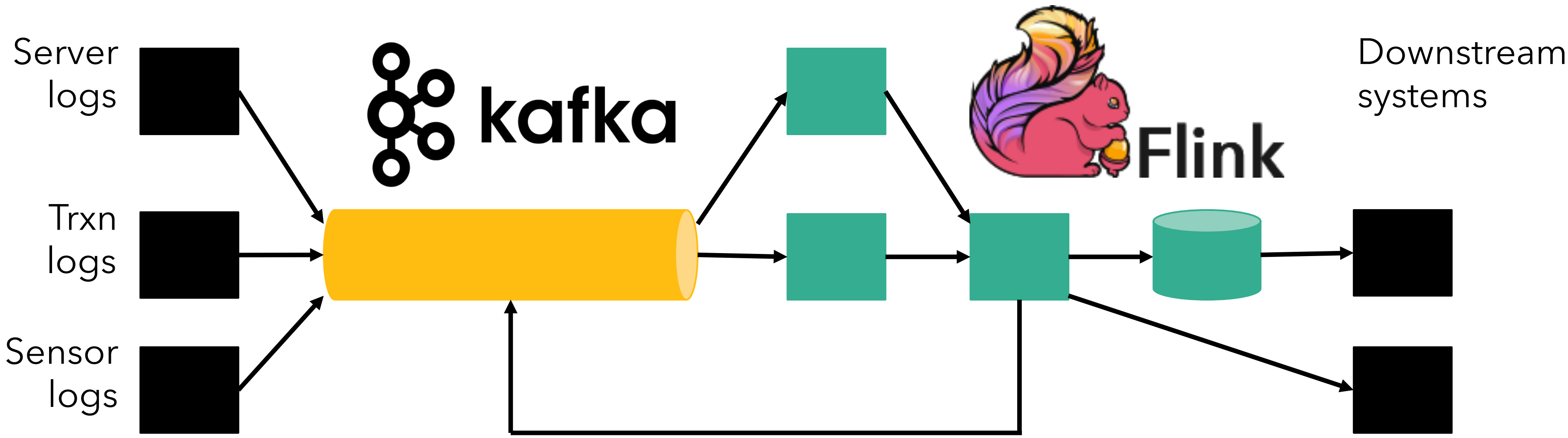
Batch comparison

API	low-level	high-level	high-level
Data Transfer	batch	batch	pipelined & batch
Memory Management	disk-based	JVM-managed	Active managed
Iterations	file system cached	in-memory cached	streamed
Fault tolerance	task level	task level	job level
Good at	massive scale out	data exploration	heavy backend & iterative jobs
Libraries	many external	built-in & external	evolving built-in & external

Streaming comparison

Streaming	"true"	mini batches	"true"
API	low-level	high-level	high-level
Fault tolerance	tuple-level ACKs	RDD-based (lineage)	coarse checkpointing
State	not built-in	external	internal
Exactly once	at least once	exactly once	exactly once
Windowing	not built-in	restricted	flexible
Latency	low	medium	low
Throughput	medium	high	high

Stream platform architecture



- Gather and backup streams
- Offer streams for consumption
- Provide stream recovery

- Analyze and correlate streams
- Create derived streams and state
- Provide these to downstream systems

What is a stream processor?

1. Pipelining
2. Stream replay

Basics

3. Operator state
4. Backup and restore

State

5. High-level APIs
6. Integration with batch

App development

7. High availability
8. Scale-in and scale-out

Large deployments

See http://data-artisans.com/stream-processing-with-flink.html

Engine comparison

	hadoop MapReduce	TEZ	Spark	Flink
API	MapReduce on k/v pairs	k/v pair Readers/Writers	Transformations on k/v pair collections	Iterative transformations on collections
Paradigm	MapReduce	DAG	RDD	Cyclic dataflows
Optimization	none	none	Optimization of SQL queries	Optimization in all APIs
Execution	Batch sorting	Batch sorting and partitioning	Batch with memory pinning	Stream with out-of-core algorithms