A view of Big Data Processing from Stratosphere

Asterios Katsifodimos, TU Berlin
asterios.katsifodimos@tu-berlin.de

Outline

- Background: Query Processing Fundamentals
  - Relational Algebra
  - Query Optimization Basics

- Background: Parallel Databases Primer
  - Pipelined vs. Data Parallelism
  - Data partitioning

- Stratosphere / Flink – Big Data meets Parallel Databases
  - Architecture
  - Programming API
  - Iterations

- After the Coffee break
  - Hands-on: programming on Stratosphere.

  *prepare your laptops

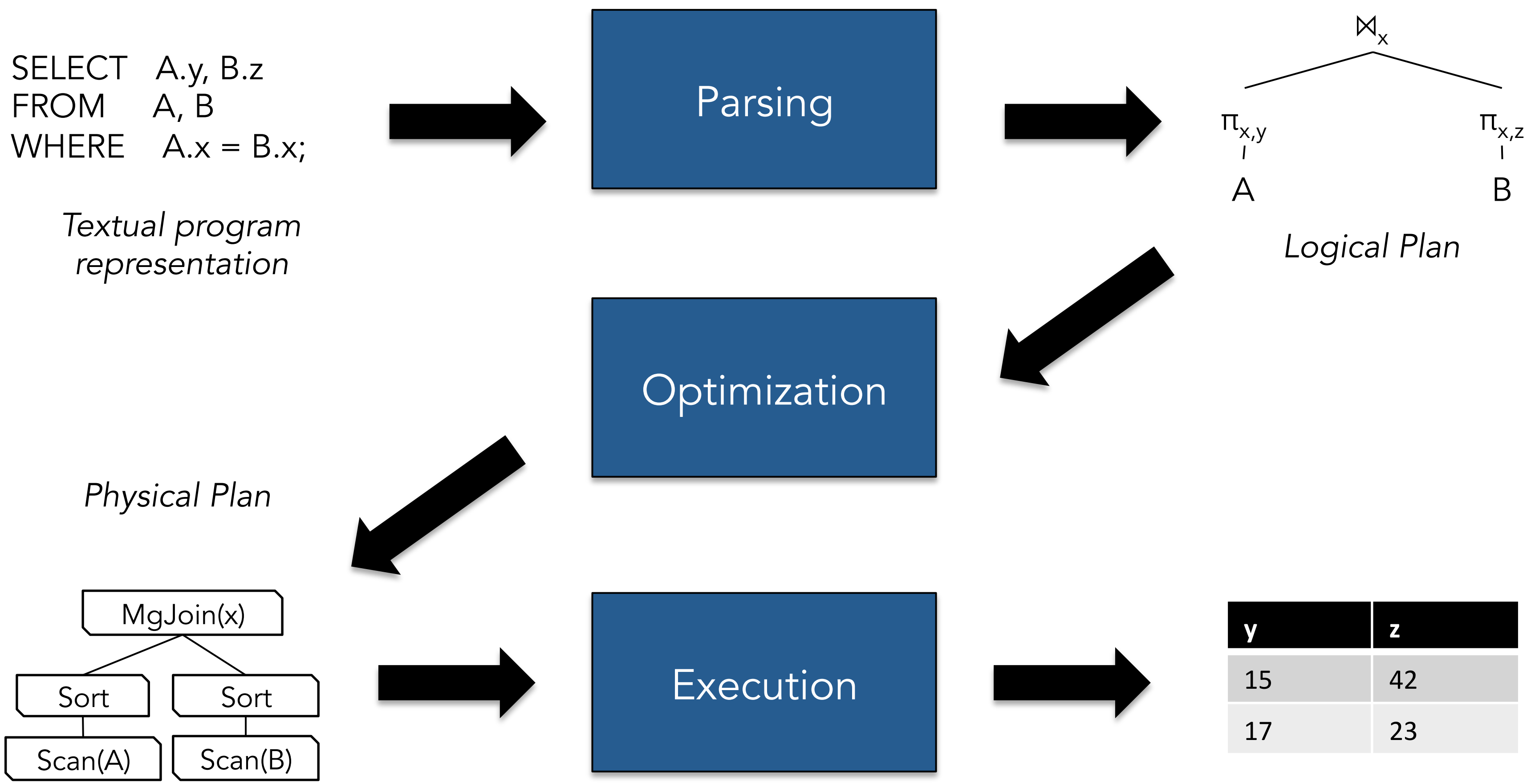
Query Processing Fundamentals

Database Engines 101

Database engines provide high-level abstractions for efficient data management

- Storage
  - "Store {some data} as {X}."
- Querying
  - "Retrieve {something} from {X_1}, …, {X_n} where {some condition} applies."
- Transactions
  - Ensures a minimal set of operational abstractions (ACID)
  - Required mostly in scenarios with concurrent read/write access. *Not discussed today!*

Query lifecycle



y	z
15	42
17	23

Relational Algebra

Operator	Semantics	Description
R, S, T, ...	Bag / Set	a collection of elements
$\pi_{\mathbf{A}}(R)$	$\{ \pi_{\mathbf{A}}(x) \mid x \in R \}$	projects a set of attributes **A**
$\sigma_{\mathbf{p}}(R)$	$\{ x \mid x \in S, p(x) \}$	filters elements where *p* is *false*
$R \times S$	$\{ (x, y) \mid x \in R, y \in S \}$	all pairs (Cartesian product)
$R \bowtie_{\theta} S$	$\{ (x, y) \mid x \in R, y \in S, \theta(x, y) \}$	theta-join, cross + filter
$R \bowtie_{\mathbf{A}} S$	$\{ (x, y) \mid x \in R, t \in S, \pi_{\mathbf{A}}(x) = \pi_{\mathbf{A}}(y) \}$	equi-join , *s, t* select the join key
$\gamma_{\mathbf{A}}(R)$	$\{ G_x = \{ y \mid y \in R, \pi_{\mathbf{A}}(y) = \pi_{\mathbf{A}}(x) \} \mid x \in R \}$	group by **A**
$\gamma_{\mathbf{A}, a}(R)$	$\{ \{ \pi_{\mathbf{A}}(y), \mathit{agg}(G_x) \} \mid x \in R \}$	group and compute aggregate *agg*
$S \cup T$ / $S \uplus T$		bag / set union
$S - T$ / $S \dot{-} T$		bag / set difference

Query Optimization

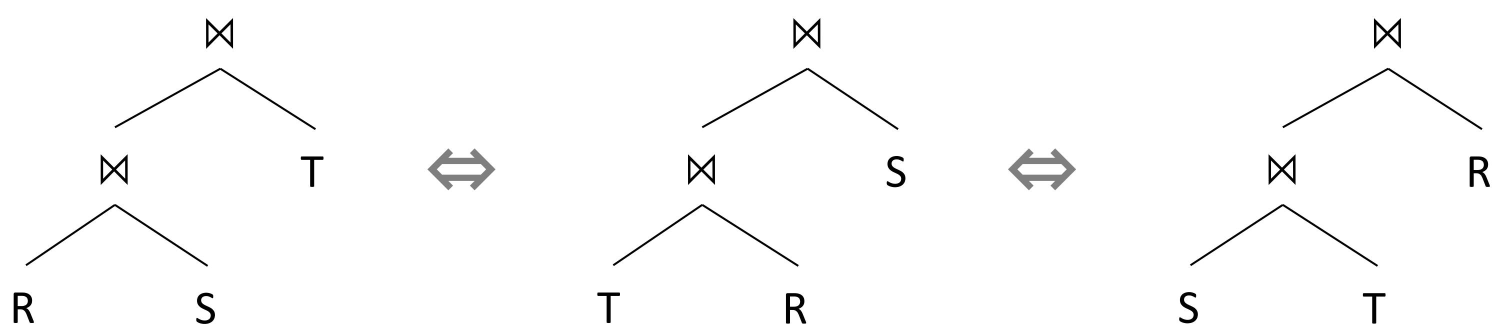
- Parsed query is represent as a relational algebra expression
  - (abstract) logical plan with well-defined semantics

- The optimizer translates it into a (concrete) execution plan (physical plan)

- Multiple degrees of freedom during the compilation process
  - Algebraic rewrites, e.g., join order
  - Algorithm selection, e.g., type of join (merge/hash/…)

Algebraic Rewrites

- Makes use of equivalences of relational algebra operators
  - Join operator is associative
  - Selections & projections can may commute joins, sometimes aggregates



join commutativity

Algorithm Selection

- Logical operators can be realized in different ways
  - Different time / space requirements depending on the concrete algorithm choice
  - Applicability depends on certain "physical properties" of the inputs (e.g. sorting)

S\T	1	3	1	4
7	(7,1)	(7,3)	(7,1)	(7,4)
4	(4,1)	(4,3)	(4,1)	**(4,4)**
1	**(1,1)**	(1,3)	**(1,1)**	(1,4)
4	(4,1)	(4,3)	(4,1)	**(4,4)**
3	(3,1)	**(3,3)**	(3,1)	(3,4)
2	(2,1)	(2,3)	(2,1)	(2,4)

Nested-Loops Join

S\T	1	1	3	4
1	**(1,1)**	**(1,1)**	(1,3)	(1,4)
2	(2,1)	(2,1)	(2,3)	(2,4)
3	(3,1)	(3,1)	**(3,3)**	(3,4)
4	(4,1)	(4,1)	(4,3)	**(4,4)**
4	(4,1)	(4,1)	(4,3)	**(4,4)**
7	(7,1)	(7,1)	(7,3)	(7,4)

Sort-Merge Join
(S, T sorted on join key)

Optimization Search Space

- Exponential search space
  - Can be restricted using heuristics, branch & bound pruning
- Finding the optimal execution plan is non-trivial
  - Decision can be cost-based or rule-based
- Algorithm costs are data-dependent
  - Typically dominated by a bottleneck resource (I/O or network)
  - Information required to calculate them can be approximated using data statistics

Parallel Databases Primer

Parallel DB Architectures

- Shared Memory
  - Several CPUs share a single memory space and (multiple) disks
  - Communication over a single common bus

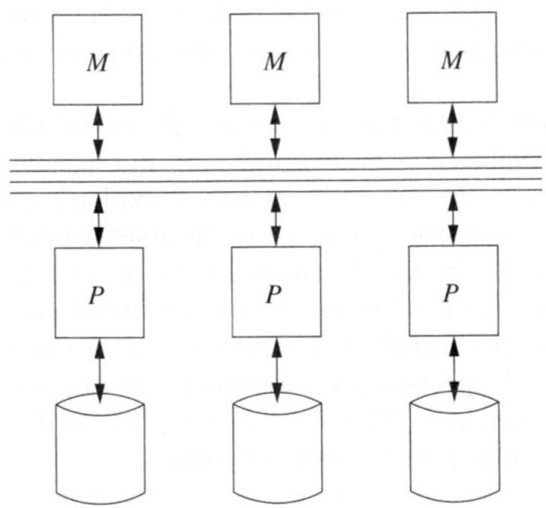


Figure 20.1: A shared-memory machine

Source:
Garcia-Molina et al., „Database Systems – The Complete Book. Second Edition"

Parallel DB Architectures

- Shared Disk
  - Several nodes with multiple CPUs, each node has its private memory
  - Single attached disk (array): Often NAS, SAN, etc…

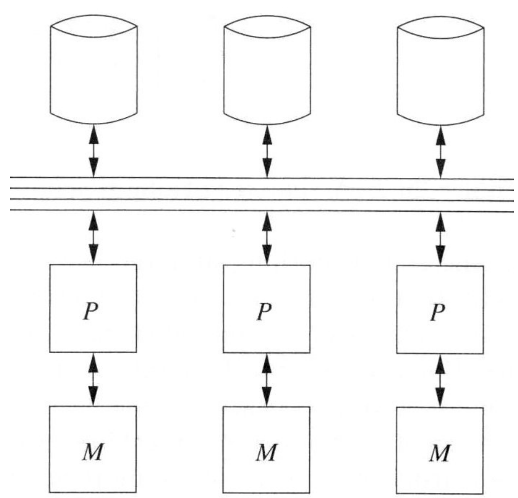


Figure 20.2: A shared-disk machine

Source:
Garcia-Molina et al., „Database Systems – The Complete Book. Second Edition"

Parallel DB Architectures

- Shared Nothing
  - Each node has it own set of CPUs, memory and disks attached
  - Data needs to be partitioned over the nodes
  - Data is exchanged through direct node-to-node communication

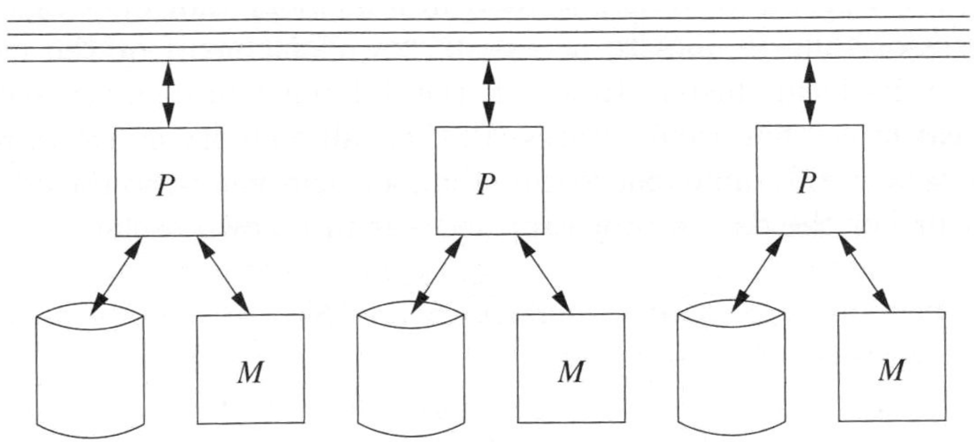
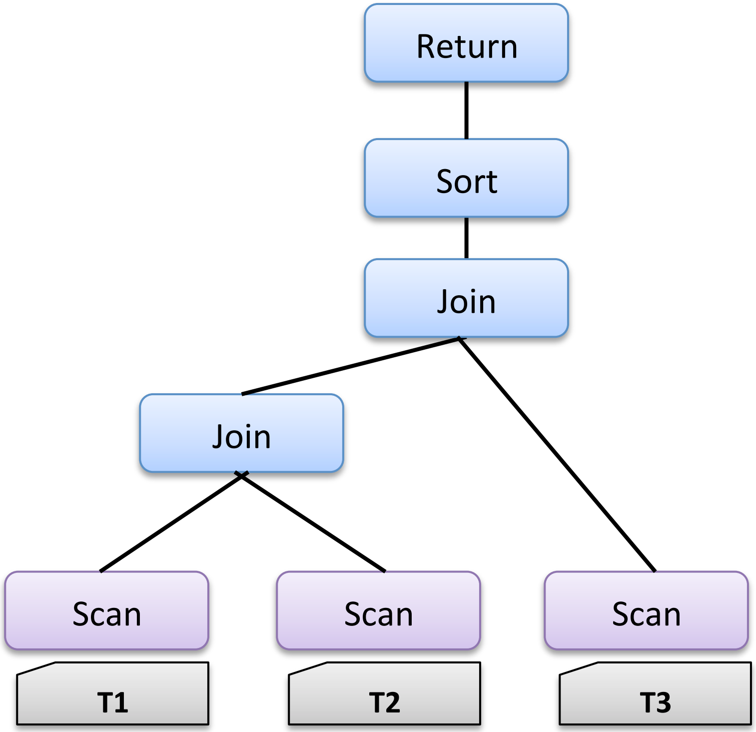


Figure 20.3: A shared-nothing machine

Source:
Garcia-Molina et al., „Database Systems – The Complete Book. Second Edition"

Parallelizing Query Plans

How would you parallelize this query plan?



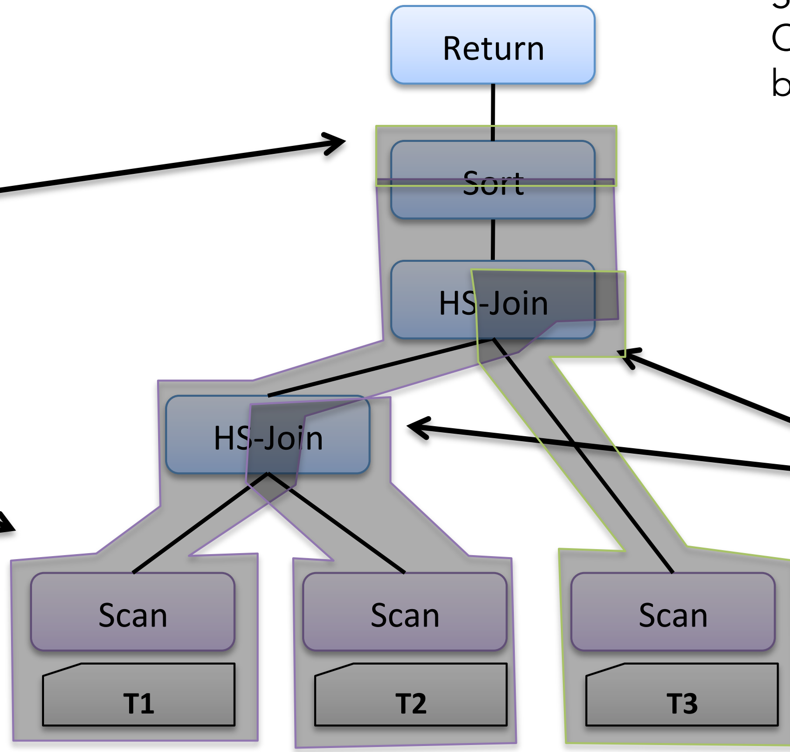
Parallelism in databases

- **Inter-query Parallelism:** Multiple queries run in parallel
- **Pipeline Parallelism**: Multiple parts of the plan run in parallel
- **Data Parallelism**: Multiple threads work on the same operator

Possible to use all at the same time

Pipeline Parallelism

Step 2:
One thread scans the table, probes the hash tables. Second thread starts the sort (sorting sub-lists, merging the first lists)



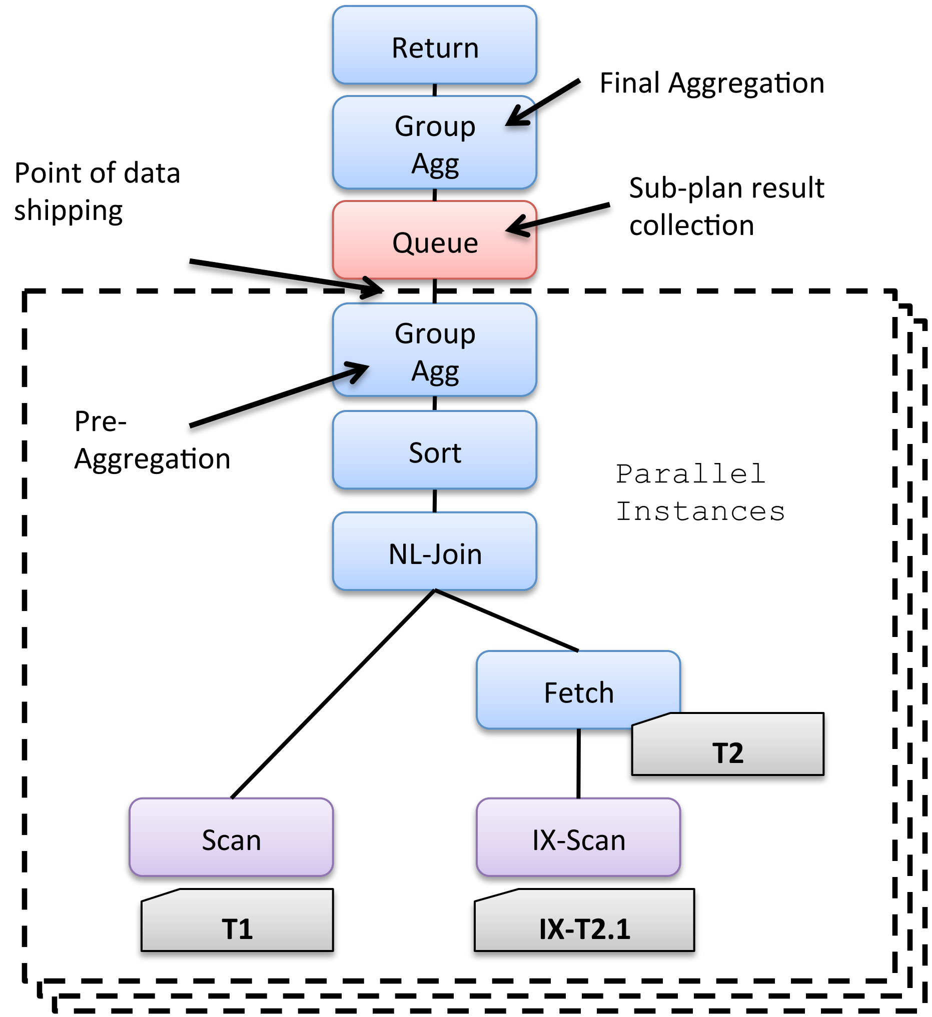
Step 3:
One thread, return result, business as usual…

Step 1:
Two threads scan one base table each and build the hash tables for the joins.

Maximum degree of parallelism in pipeline parallelism?

Data Parallelism

- Multiple instances of a sub-plan are executed on different computers.
- The instances operate on different splits/partitions of the data.
- At some points, results from the sub-plans are collected.
- For more complex queries, results are not collected but re-distributed, for further parallel processing.



Data Parallelism

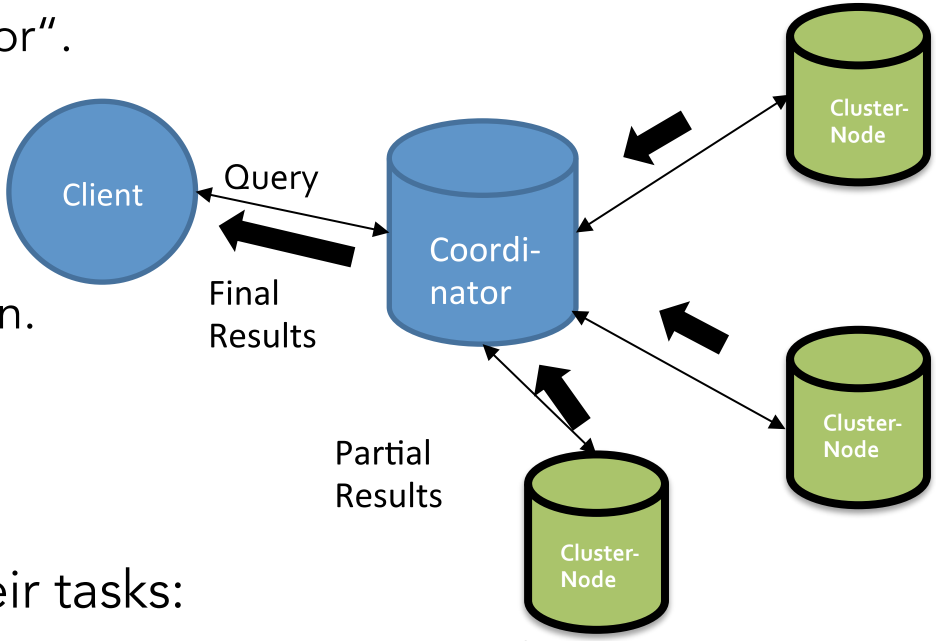
- Data is divided into several partitions
  - Most operations don't need a complete view of the data!
    - E.g. $\sigma_p(\cdot)$ looks only at a single tuple at a time.
  - Partitions can be processed independently and in parallel

- (max) Degree of Parallelism = number of possible partitions
  - For $\sigma_p(\cdot)$ as high as the number of tuples

- BUT: Some operations possibly need a view of larger portions of the data:
  - E.g. some Grouping/Aggregation operations need **all tuples** with the same grouping key, e.g., Median

Data Parallelism Workflow

- Client send a SQL query to one of the cluster nodes:
  - Node becomes the "coordinator".
- Coordinator compiles the query:
  - Parsing, Checking, Optimization.
  - Parallelization.
- Sends partial plans to the other cluster nodes that describes their tasks:
  - Coordinator also executes the partial plan on his part of the data.
- Collects partial results and finalizes them (see next slide)



Data Partitioning

- Partitioning the data means creating multiple disjoint sub-sets
  - Example: Sales data, every year gets its own partition.

- For shared-nothing, data must be partitioned across nodes:
  - If it were replicated, it would effectively become a shared-disk with the local disks acting like a cache (must be kept coherent).

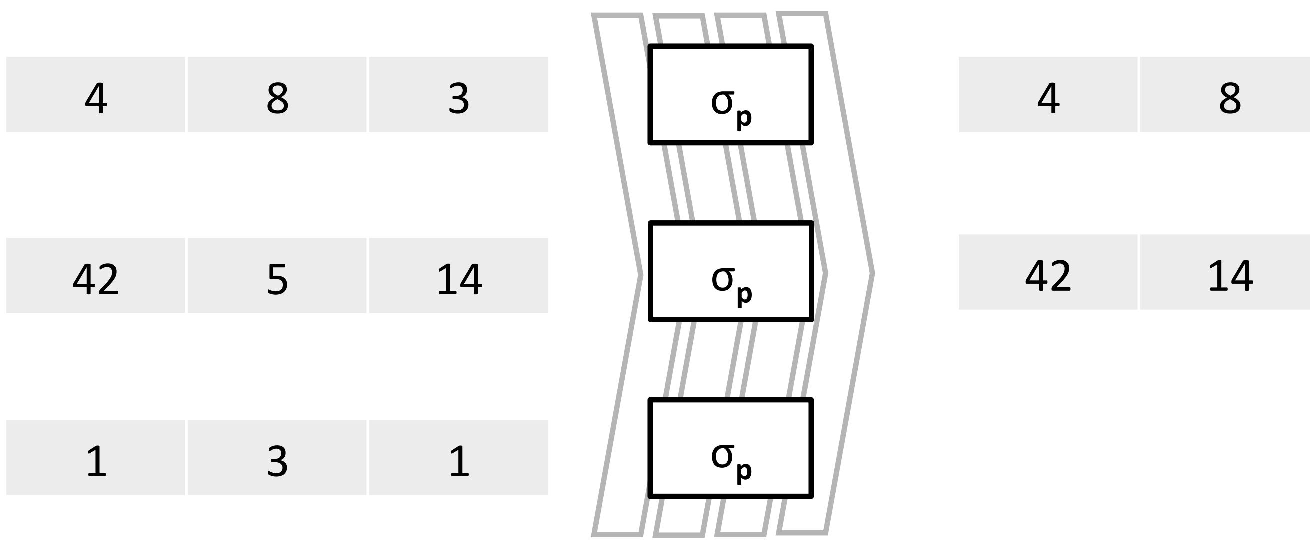
- Partitioning with certain characteristics has more advantages:
  - Some queries can be limited to operate on certain sets only, if it is provable that all relevant data (passing the predicates) is in that partition.
  - Partitions can be simply dropped as a whole (data is rolled out) when it is no longer needed (e.g. discard old sales).

Data Partitioning

- **Round robin**: Each set gets a tuple in a round, all sets have guaranteed equal amount of tuples, no apparent relationship between tuples in one set.
- **Hash Partitioned**: Define a set of partitioning columns. Generate a hash value over those columns to decide the target set. All tuples with equal values in the partitioning columns are in the same set.
- **Range Partitioned**: Define a set of partitioning columns and split the domain of those columns into ranges. The range determines the target set. All tuples on one set are in the same range.

Parallel Selection

- Each node performs the selection on its existing local partition.
  - Selection needs no context.
  - Data can be partitioned in a arbitrary way.
  - Partial results *union*-ed afterwards.



Parallel Aggregation

- Re-partition dataset on grouping column set
  - Tuples with the same grouping values will end up on the same machine
  - Apply local grouping/aggregation algorithm to each partition in parallel

- Not possible if the aggregation function requires sorting
  - E.g. Median

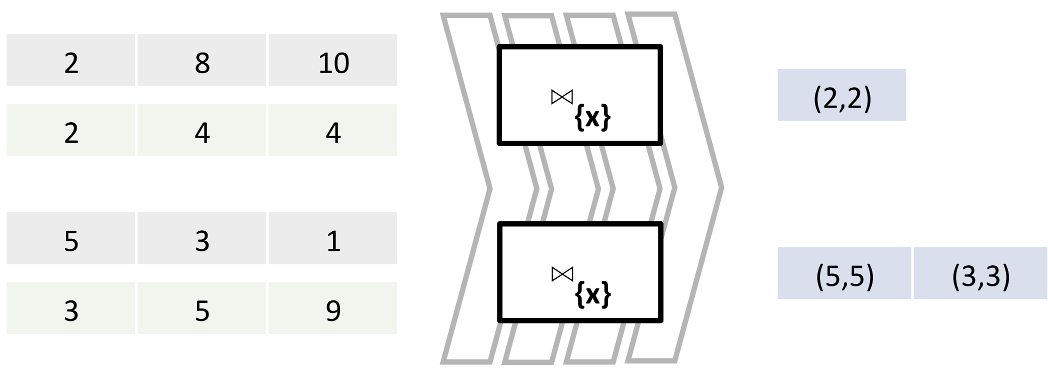
Parallel Equi-Joins

- A special class of joins suited for parallelization are Equi-Joins.
  - Only look at tuple pairs that share the same join key
  - Partition relations R and S using the same partitioning scheme over the join key
  - All values of R and S with the same join key end up at the same node
  - All joins can be performed locally

- Multiple partitioning strategies possible:
  - Co-Located Join
  - Directed Join
  - Re-Partitioning Join

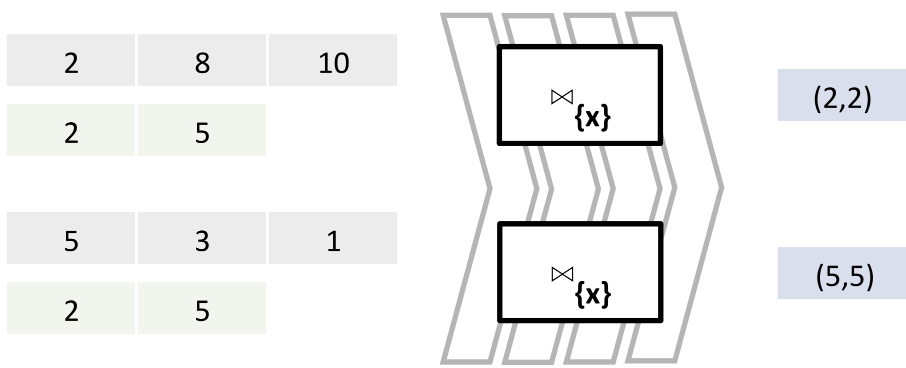
Co-Located Join

- Data is already partitioned on the join key
  - No re-partioning needed
  - Local joins work “out of the box”



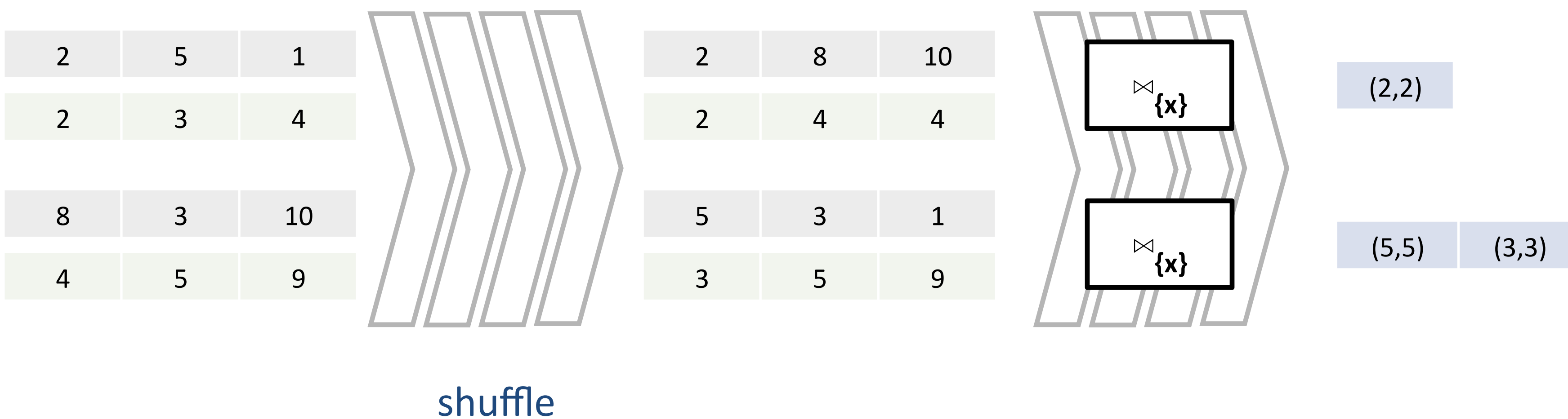
Directed Join (Broadcast Join)

- One side is fully replicated on each node
  - No re-partitioning for the other side needed
  - Works best if one side is much smaller than the other



Re-Partitioning Join

- Both-sides re-partitioned on the join key
  - Fallback strategy for inputs with similar size



STRATOSPHERE

What is Flink

- Probable renaming of Stratosphere system
  - German for “fast, swift”
- Apache Incubator project
  - Proposed by core developers of Stratosphere project
  - Core processing engine developed in the Stratosphere research project
- Community lives at dev@flink.incubator.apache.org
- For this talk, I will still use the name “Stratosphere”

What is Stratosphere

- General purpose computation engine for Hadoop data on YARN clusters
  - Backed by database-inspired execution and optimization
  - Focusing on making the user's life easy
  - Orders of magnitude faster than Hadoop MapReduce

- www.stratosphere.eu
- Follow @stratosphere_eu

Databases ➤ "Big Data"

- Tables ➤ Tables and unstructured files
  - Schema on read
- Parallel ➤ More parallel, commodity, shared clusters
  - Mid-query fault tolerance, resource allocation
- SQL ➤ SQL and Java, Scala, Python, you name it
  - General object manipulation
- Data warehousing ➤ Logs, ML, Graphs, also DW
  - Iterative processing, user-defined functions
- Proprietary ➤ Open source
  - New project structures (and monetization strategies)

Stratosphere: general-purpose programming + database execution

Take from database Technology

- Declarativity
- Query optimization
- Robust out-of-core

Add

- Iterations
- Advanced Dataflows
- General APIs

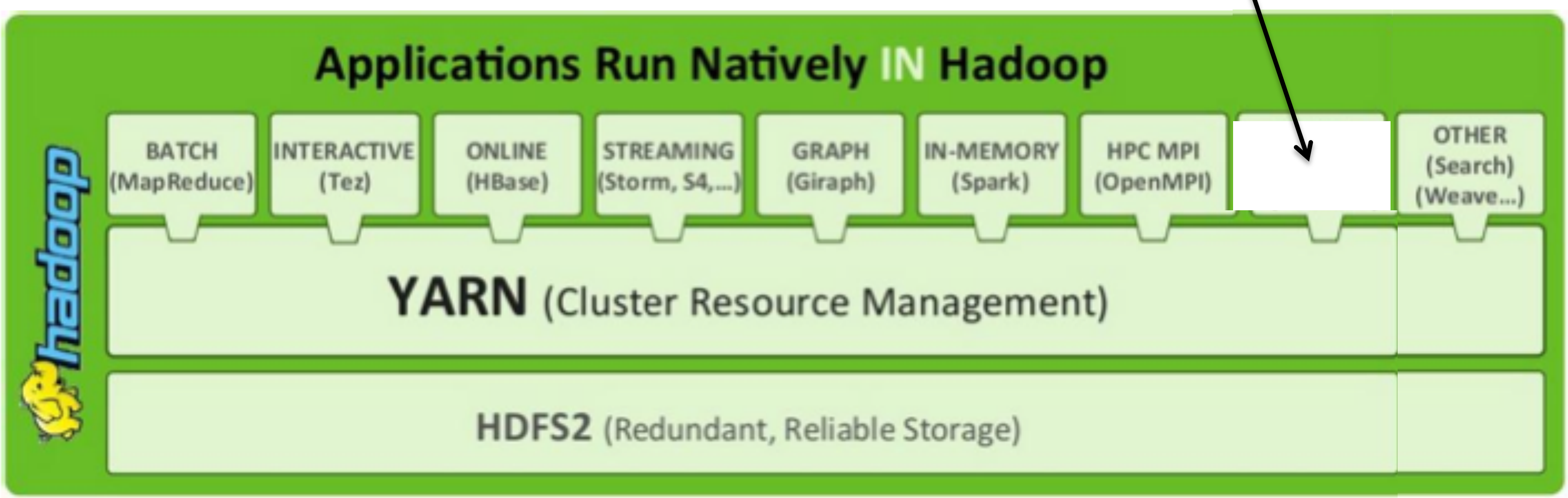
Take from MapReduce technology

- Scalability
- User-defined functions
- Complex data types
- Schema on read

Placement in Hadoop stack

- Analyzes HDFS data directly
- Runs on top of YARN

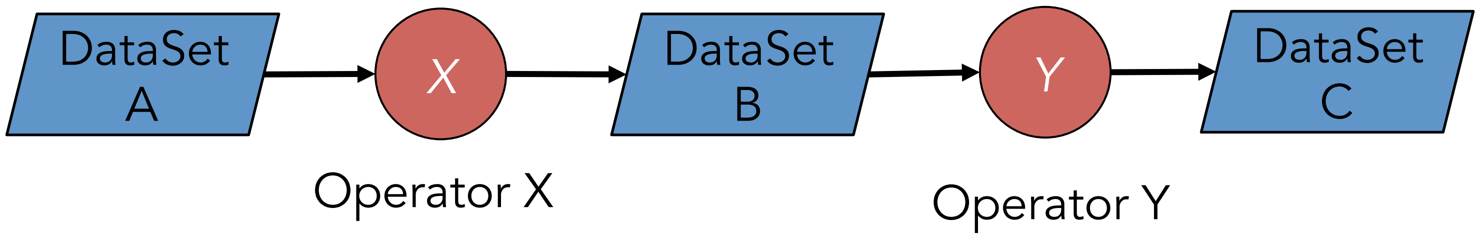
Big Data looks tiny from Stratosphere



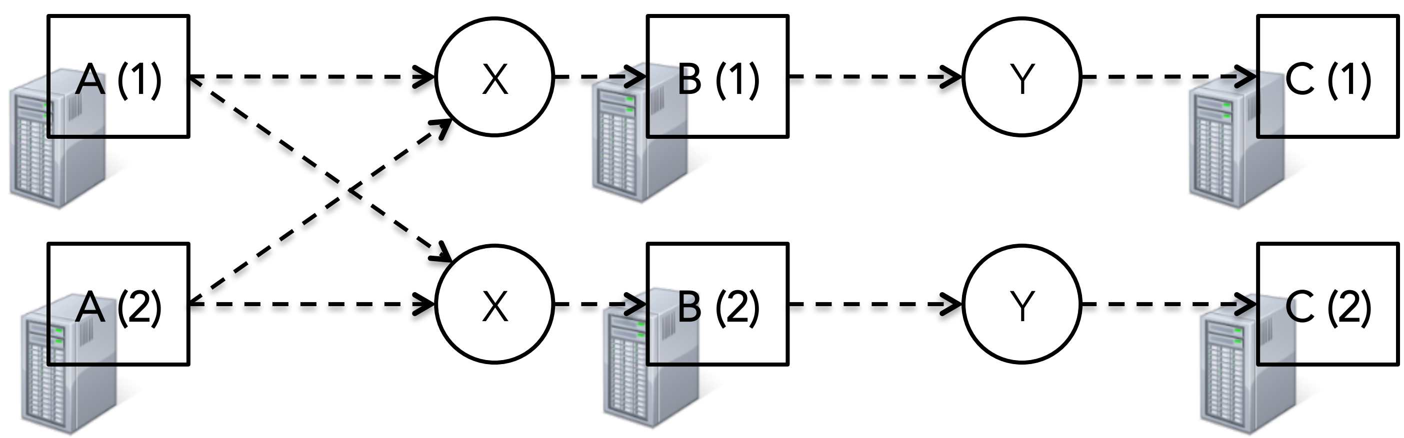
Programming Model and APIs

Data sets and operators

Program

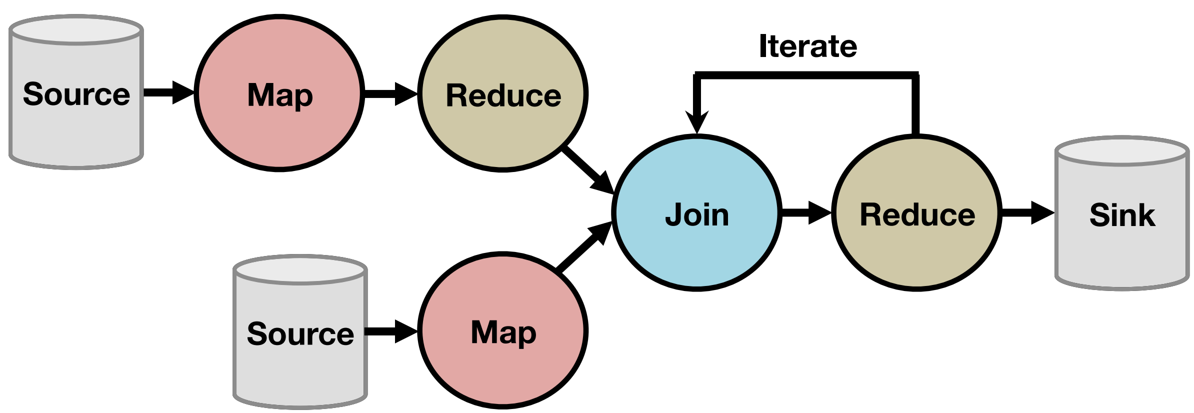


Parallel Execution



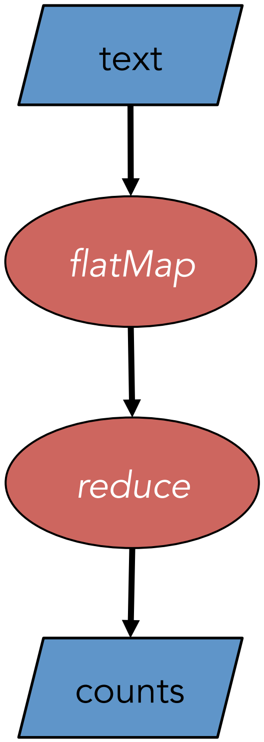
Rich set of operators

Map, Reduce, Join, CoGroup, Union, Iterate, Delta Iterate, Filter, FlatMap, GroupReduce, Project, Aggregate, Distinct, Vertex-Update, Accumulators



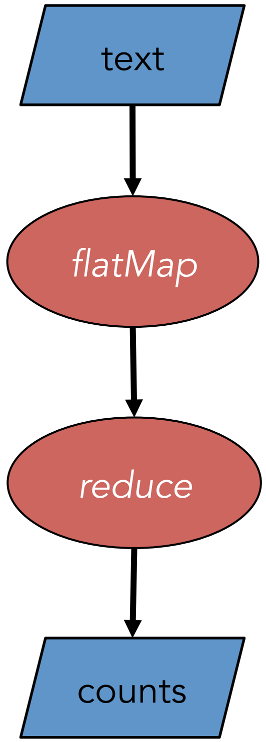
WordCount in Java

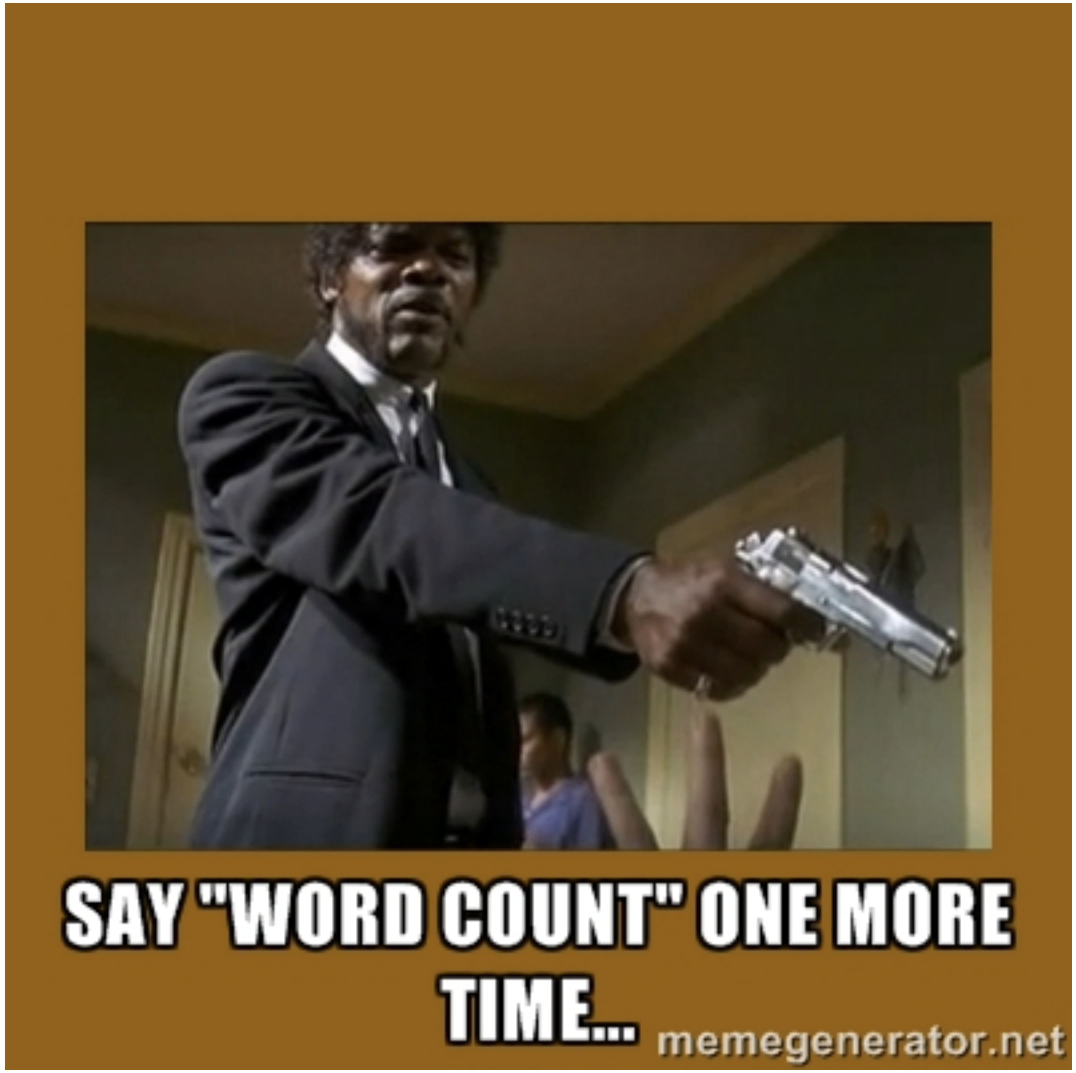
```
final ExecutionEnvironment env =
  ExecutionEnvironment.getExecutionEnvironment();

DataSet<String> text = readTextFile (input);

DataSet<Tuple2<String, Integer>> counts= text
  .flatMap(new LineSplitter())
  .groupBy(0)
  .count();

env.execute("Word Count Example");


public static final class LineSplitter extends
  FlatMapFunction<String, Tuple2<String, Integer>> {

  public void flatMap(String line, Collector<Tuple2<String, Integer>> out) {
    for (String word : line.split(" ")) {
      out.collect(new Tuple2<String, Integer>(word, 1));
    }
  }
}
```



WordCount in Scala

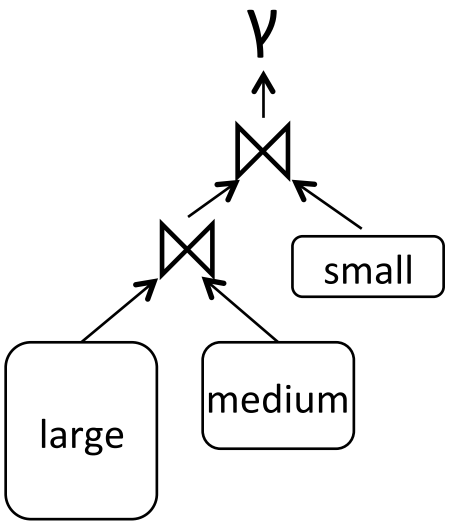
```
val input = TextFile(textInput)

val counts = input
    .flatMap { line => line.split(“\\W+”) }
    .groupBy { word => word }
    .count()
```



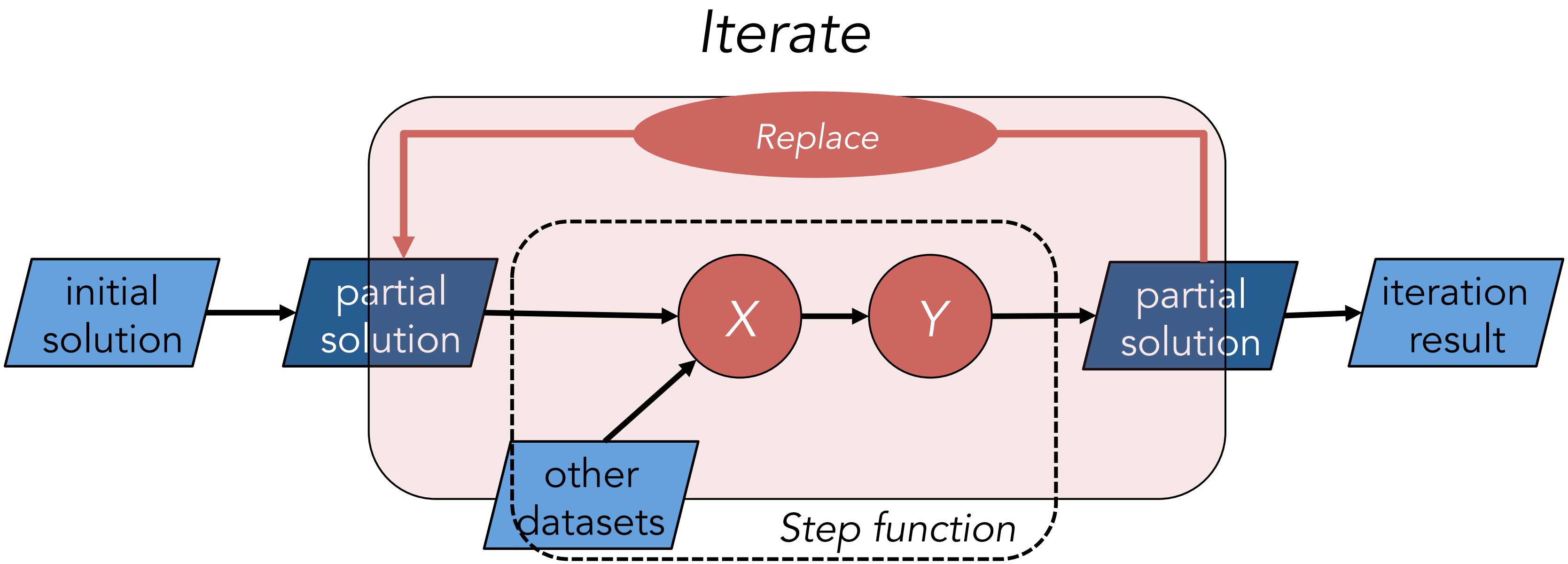

SAY "WORD COUNT" ONE MORE TIME...
memegenerator.net

Longer Operator Pipelines

```
DataSet<Tuple...> large = env.readCsv(...);
DataSet<Tuple...> medium = env.readCsv(...);
DataSet<Tuple...> small = env.readCsv(...);

DataSet<Tuple...> joined1 = large
                .join(medium)
                .where(3).equals(1)
                .with(new JoinFunction() { ... });

DataSet<Tuple...> joined2 = small
                .join(joined1)
                .where(0).equals(2)
                .with(new JoinFunction() { ... });

DataSet<Tuple...> result = joined2
                .groupBy(3)
                .max(2);
```



"Iterate" operator



- Built-in operator to support looping over data
- Applies step function to partial solution until convergence
- Step function can be arbitrary Stratosphere program
- Convergence via fixed number of iterations or custom convergence criterion

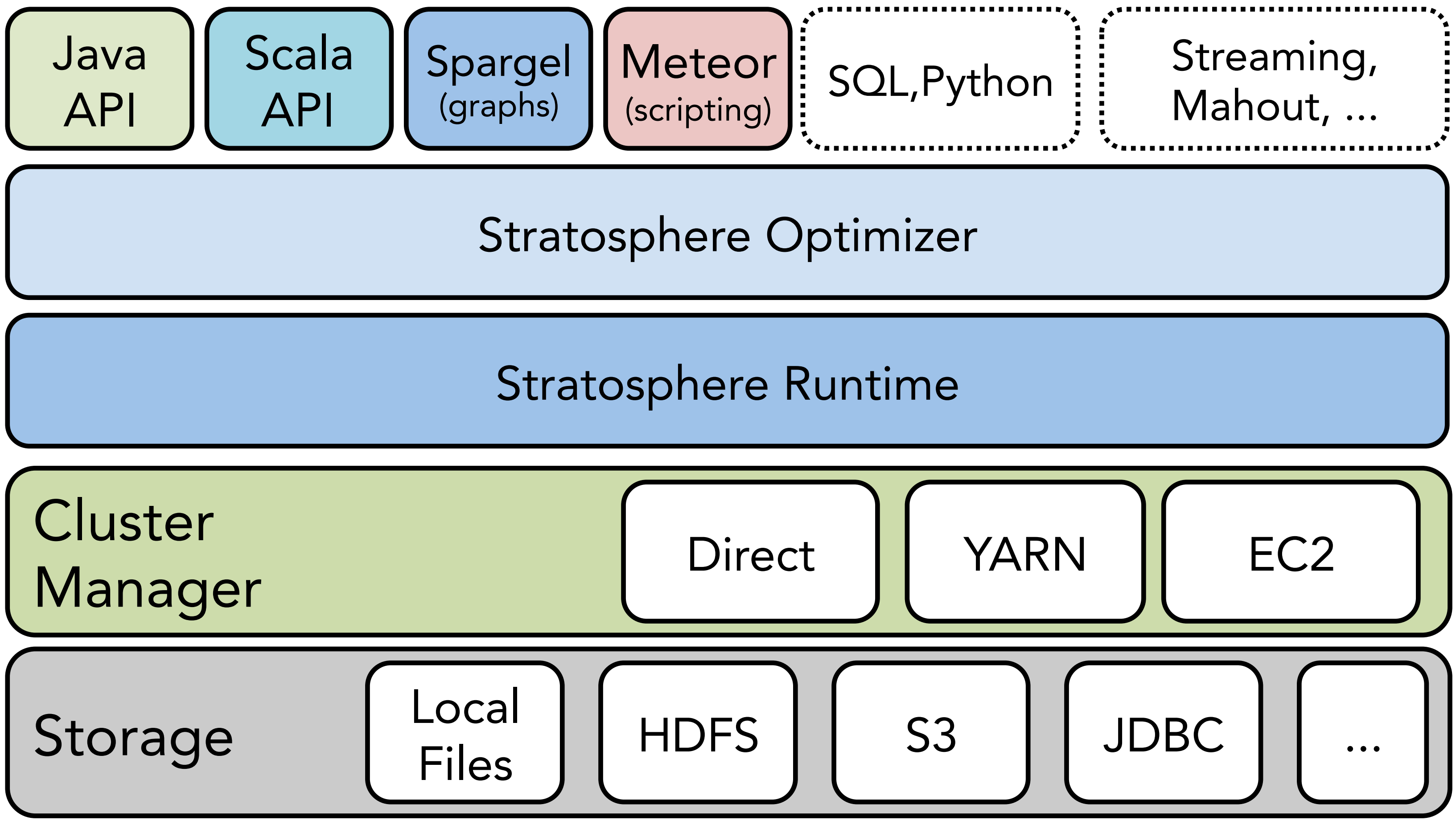
Using Spargel: The graph API

```
ExecutionEnvironment env = getExecutionEnvironment();

DataSet<Long> vertexIds = env.readCsv(...);
DataSet<Tuple2<Long, Long>> edges = env.readCsv(...);

DataSet<Tuple2<Long, Long>> vertices = vertexIds.map(new IdAssigner());

DataSet<Tuple2<Long, Long>> result = vertices.runOperation(
        VertexCentricIteration.withPlainEdges(
                edges, new CCUpdater(), new CCMessager(), 100));

result.print();
env.execute("Connected Components");
```

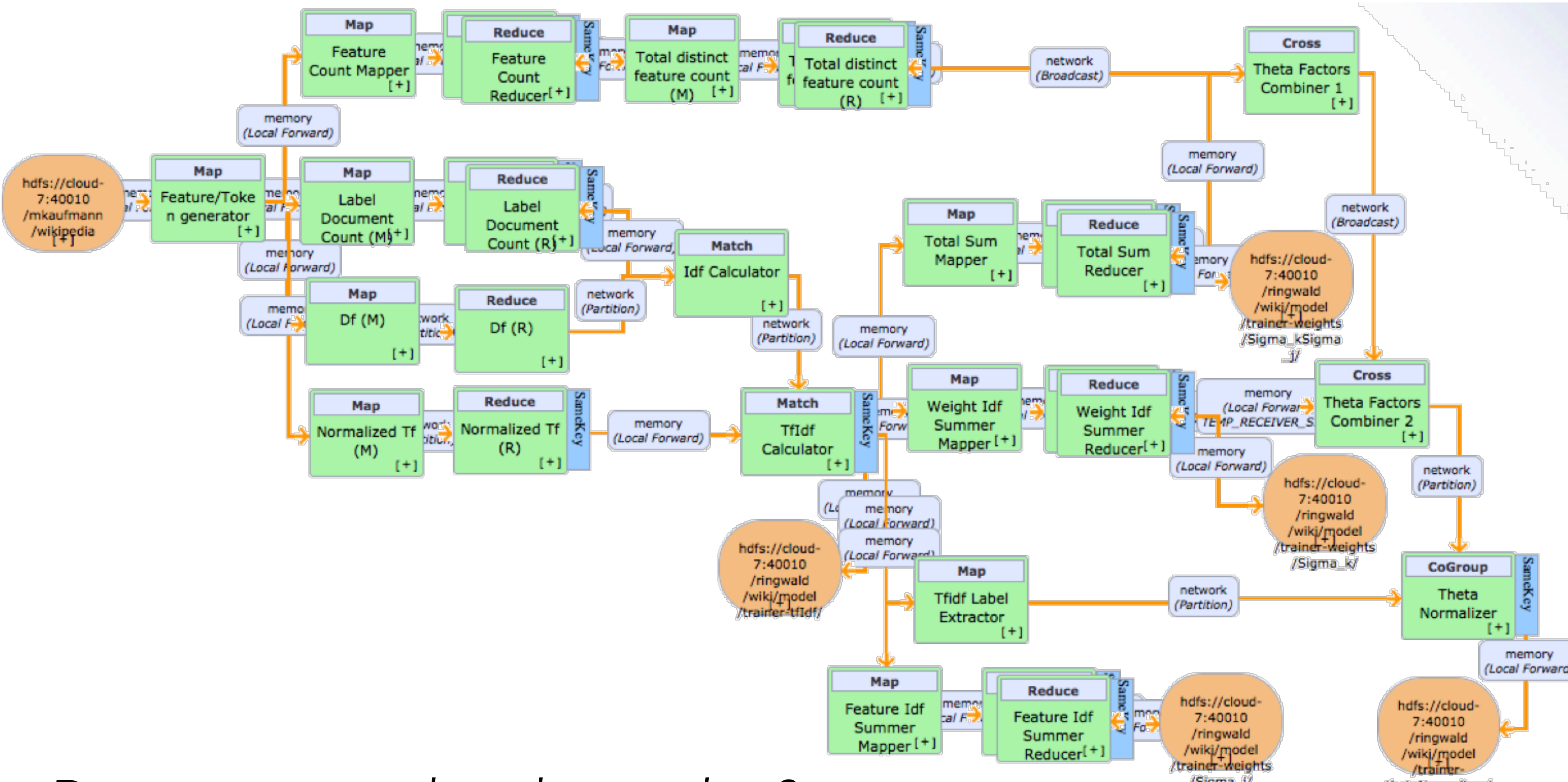
Pregel/Giraph-style Graph Computation

Stratosphere Internals

Stratosphere stack

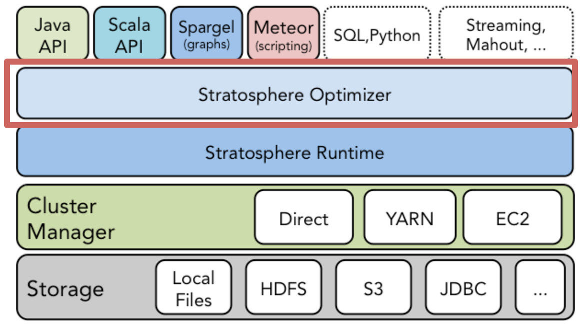


Why optimization ?



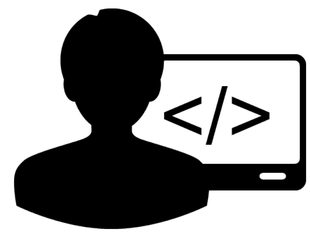
Do you want to hand-tune that?

Stratosphere optimizer



- There are many ways to execute a program
- What you write is ***not*** what is executed
- No need to hardcode execution strategies
- Optimizer decides:
  - Pipelines and operator placement
  - Sort- vs. hash- based execution
  - Data exchange (partition vs. broadcast)
  - Data partitioning steps
  - In-memory caching

Effect of optimization



Hash vs. Sort
Partition vs. Broadcast
Caching
Reusing partition/sort

Execution Plan A

Execution Plan B

Execution Plan C

Run on a sample on the laptop

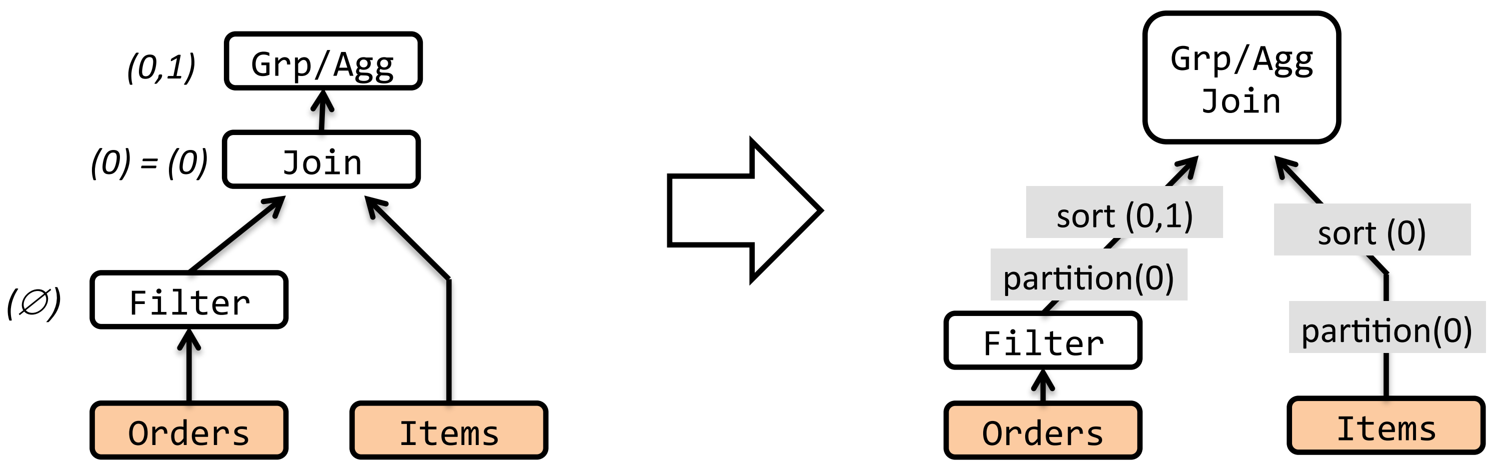
Run on large files on the cluster

Run a month later after the data evolved

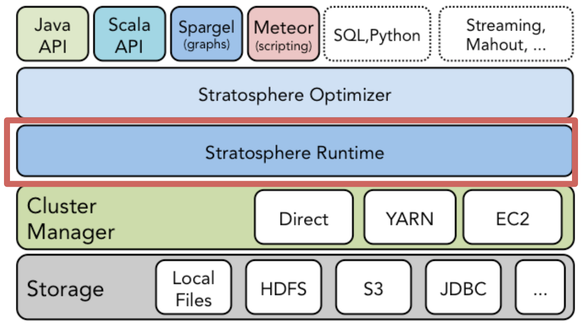
Optimization Example

```
case class Order(id: Int, priority: Int, ...)
case class Item(id: Int, price: double, )
case class PricedOrder(id, priority, price)
```

```
val orders = DataSource(...)
val items  = DataSource(...)

val filtered = orders filter { ... }

val prio = filtered join items where { _.id } isEqualTo { _.id }
                    map {(o,li) => PricedOrder(o.id, o.priority, li.price)}

val sales = prio groupBy {p => (p.id, p.priority)} aggregate ({_.price},SUM)
```



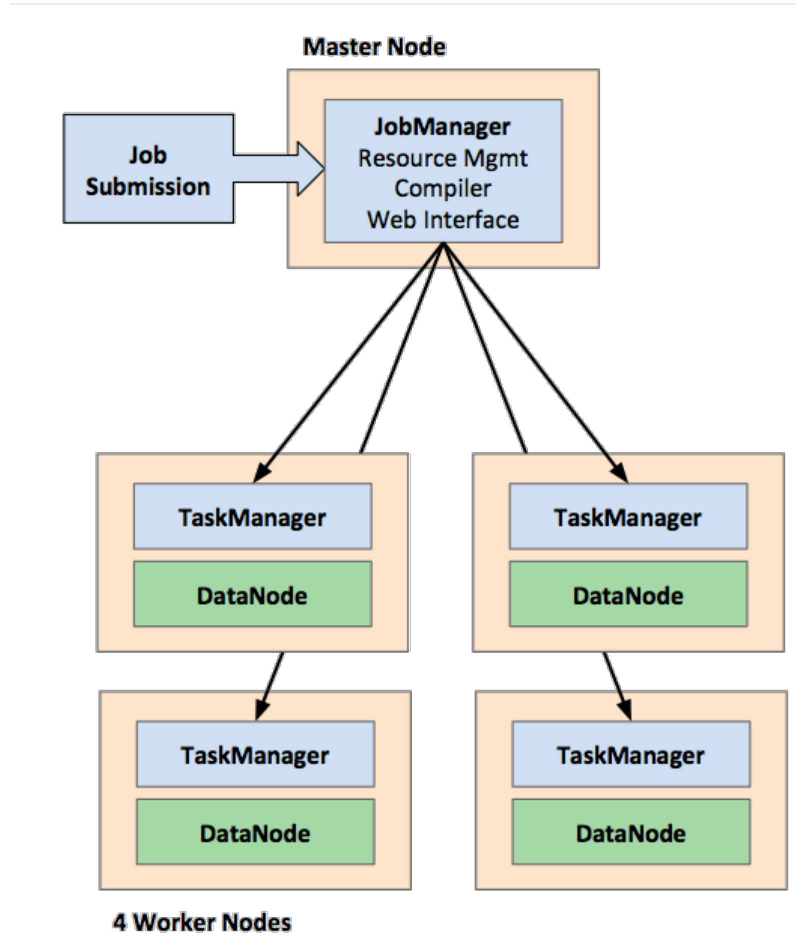
Stratosphere runtime



- Hybrid MapReduce and MPP database runtime
- Streaming engine
  - Low-latency queries
- Stateful multi-pass algorithms
  - Very efficient for ML/graphs
- Heavily in-memory
  - Fast on modern machines
- Out-of-core gracefully
  - Scales beyond main memory

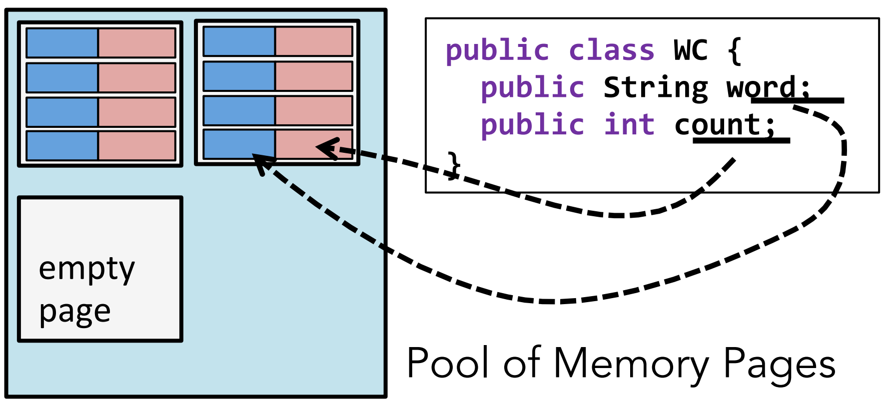
Distributed architecture

- Master (Job Manager) handles job submission, scheduling, and metadata
- Workers (Task Managers) execute operations
- Data can be streamed between nodes
- All operators start in-memory and gradually go out-of-core



Memory management

Big Data looks tiny from Stratosphere



Pool of Memory Pages

- Works on pages of bytes
- Maps objects transparently to these pages
- Full control over memory, out-of-core enabled
- Algorithms work on binary representation
- Address individual fields (not deserialize whole object)

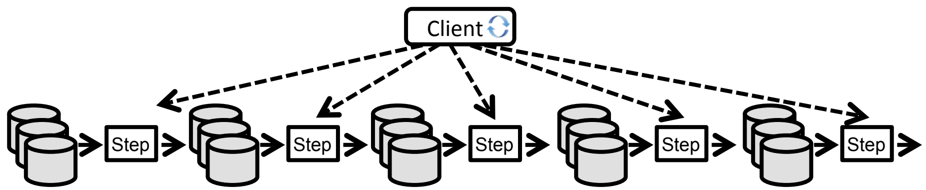
Spark

Distributed Collection

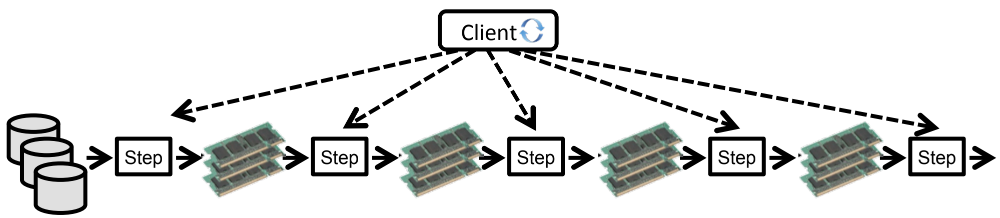
List[WC]

- Collections of objects
- General-purpose serializer (Java / Kryo)
- Limited control over memory & less efficient spilling

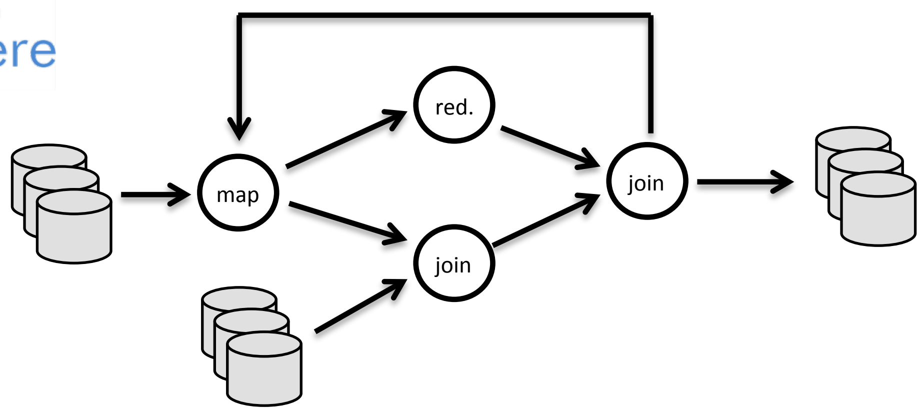
Built-in vs. driver-based looping



Loop outside the system, in driver program



Iterative program looks like many independent jobs

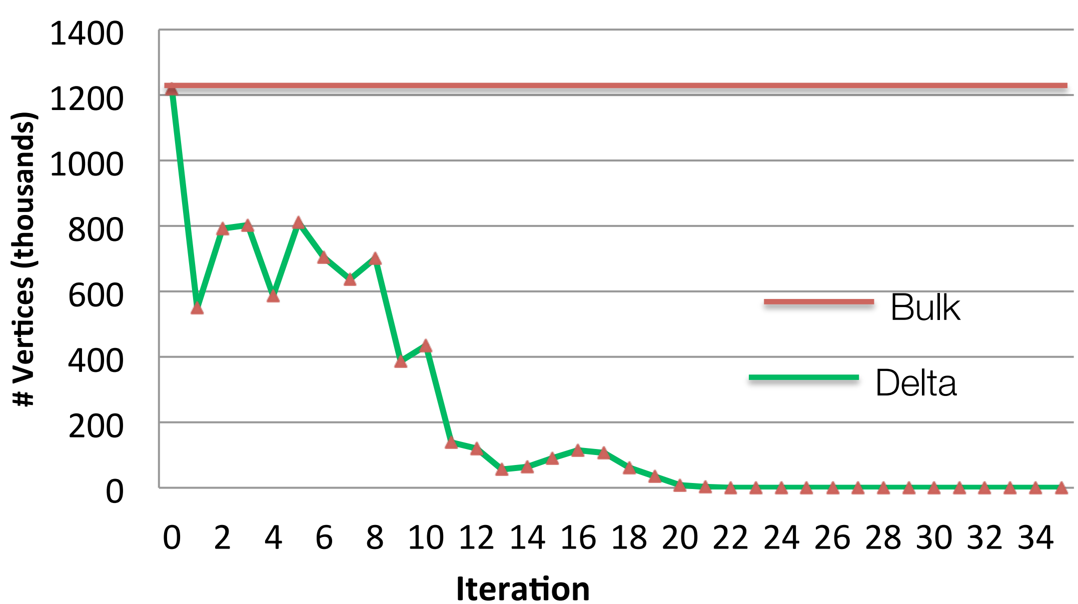


Dataflows with feedback edges

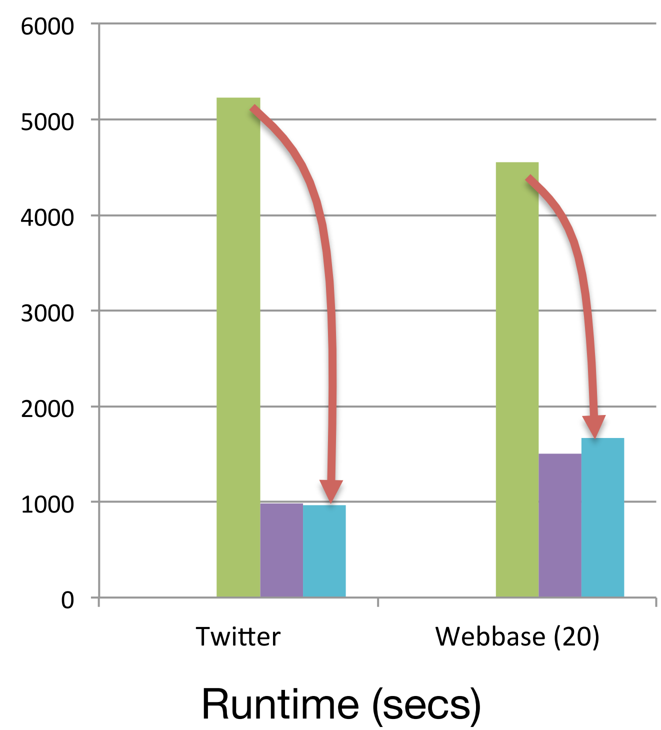
System is iteration-aware, can optimize the job

Delta iterations

Cover typical use cases of Pregel-like systems with comparable performance in a generic platform and developer API.



Computations performed in each iteration for connected communities of a social graph



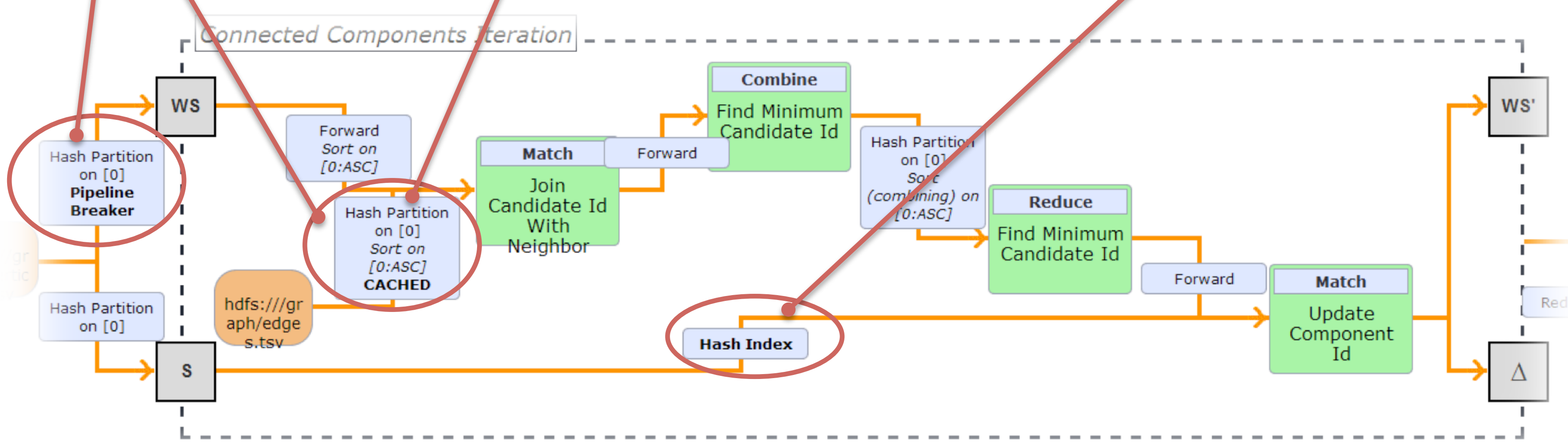
*Note: Runtime experiment uses larger graph

Optimizing iterative programs

Pushing work
„out of the loop“

Caching Loop-invariant Data

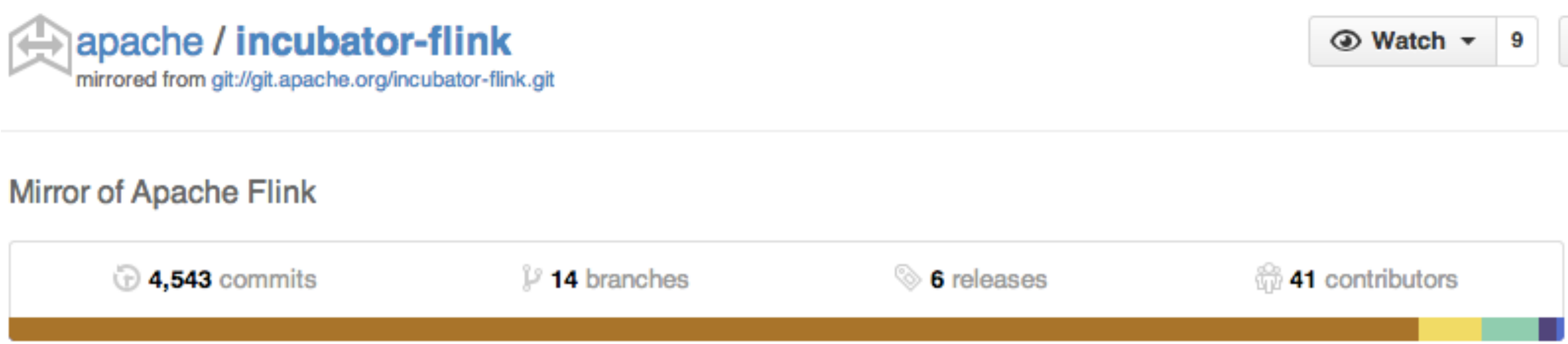
Maintain state as index



Project Structure and Roadmap

Project stats

- Last major release (v0.5) the result of work of 26 contributors from 12 Universities and companies
- Mentoring organization in Google Summer of Code 2014
- Used by ResearchGate, evaluated by Spotify, Deutsche Telekom



Project structure

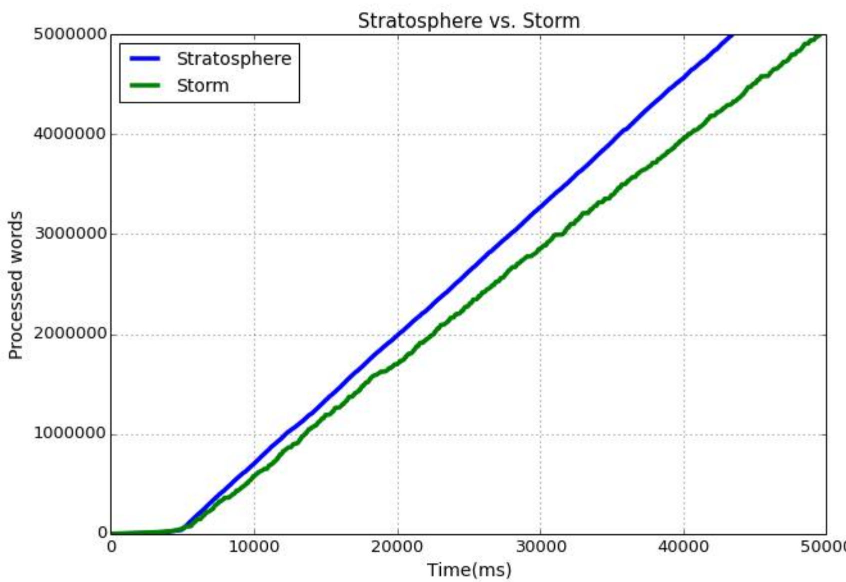
- Flink is an Apache Incubator "podling"
- Democratic
  - Committers and mentors have binding votes
  - No organizational ownership
- Open
  - All discussions are public
  - Contributions are very welcome
  - A goal of incubation is to add committers and enlarge the community

Features in the works

- Mid-query fault tolerance
- Interactive shells
- Python API
- Stratosphere streaming
- Hadoop MapReduce compatibility
- Mahout frontend
- Stratosphere on Tez
- …

Stratosphere Streaming

```
public class StreamedWordCount {

  public static void main(String[] args) {
    StreamExecutionEnvironment env =
      new StreamExecutionEnvironment();

    DataStream<Tuple2<String, Integer>> stream = env
      .readTextFile(”path/to/file”)
      .flatMap(new WordCountSplitter())
      .partitionBy(0)
      .map(new WordCountCounter())
      .addSink(new WordCountSink());

    env.execute();
  }
}
```



Single core performance

How to get involved

- Developers and users: report bugs, contribute patches
  - Very active and friendly community
- Companies: use the system in-house
  - Contact us for a joint project/partnership
- Students: use Stratosphere in your thesis

Big Data looks tiny from Stratosphere

stratosphere.eu

github.com/apache/incubator-flink

@stratosphere_eu

Hands-on

Stratosphere Development

Stratosphere Development

- Map-Reduce Basics
- Hands-on: Counting Words on Massive Corpora
- Hands-on: Parallel K-Means Clustering

K-Means Clustering

- Given *k*, the *k-means* algorithm consists of four steps:
  - Select initial centroids at random.
  - Assign each object to the cluster with the nearest centroid.
  - Compute each centroid as the mean of the objects assigned to it.
  - Repeat previous 2 steps until no change.

K-Means Clustering (cont.)