AIM3 – Scalable Data Analysis and Data Mining

04 – Stratosphere

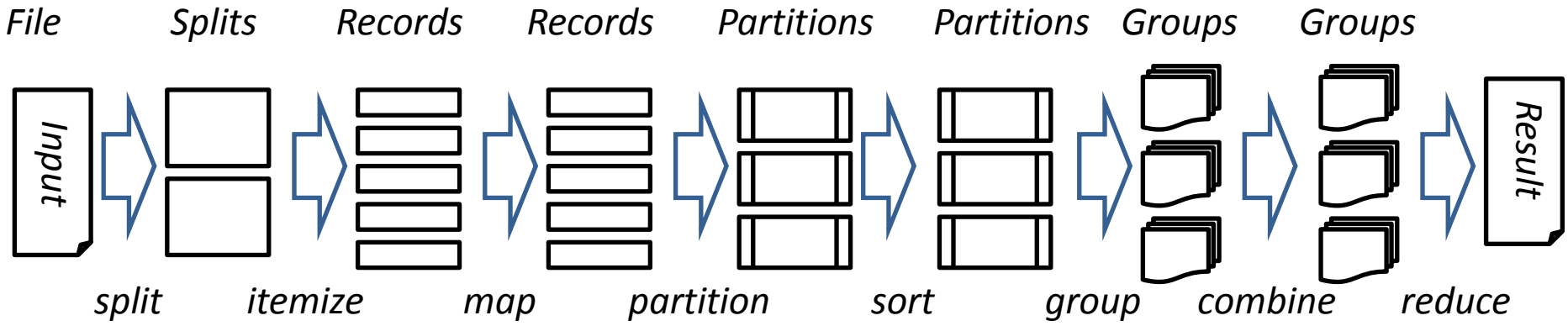
Sebastian Schelter, Christoph Boden, Volker Markl

Fachgebiet Datenbanksysteme und Informationsmanagement
Technische Universität Berlin

http://www.dima.tu-berlin.de/

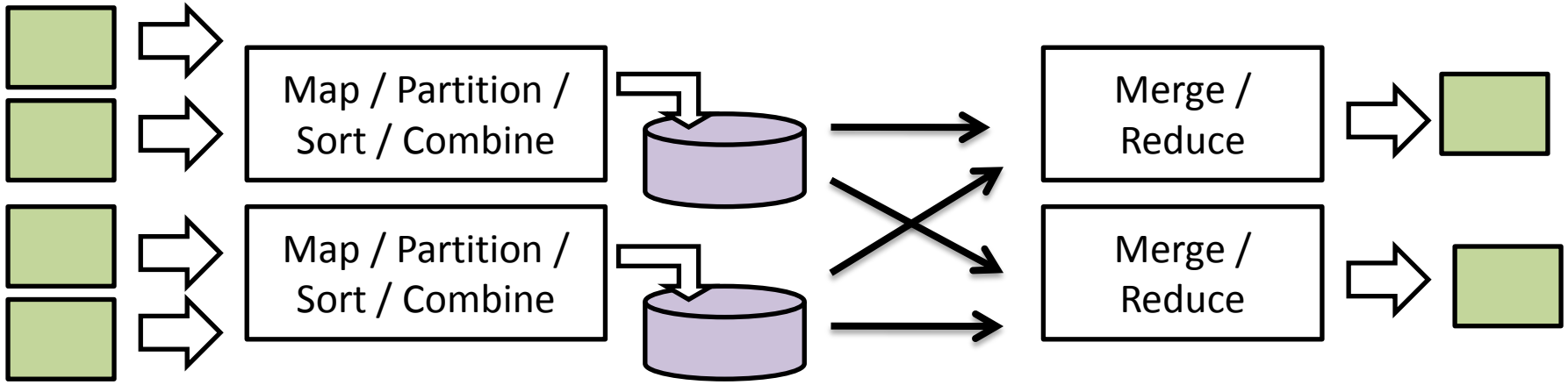
Let's talk MapReduce first, though…

- MapReduce is a powerful abstraction
  - Express problems are pairs of functions *Map* and *Reduce*
  - In practice, more functions are available (and required to use)
- A typical MapReduce system (like Hadoop) implements a processing pipeline
  - Here the view from the programmers perspective



MapReduce execution

- Hadoop's runtime implements a static strategy based on a distributed sort based on partitioning

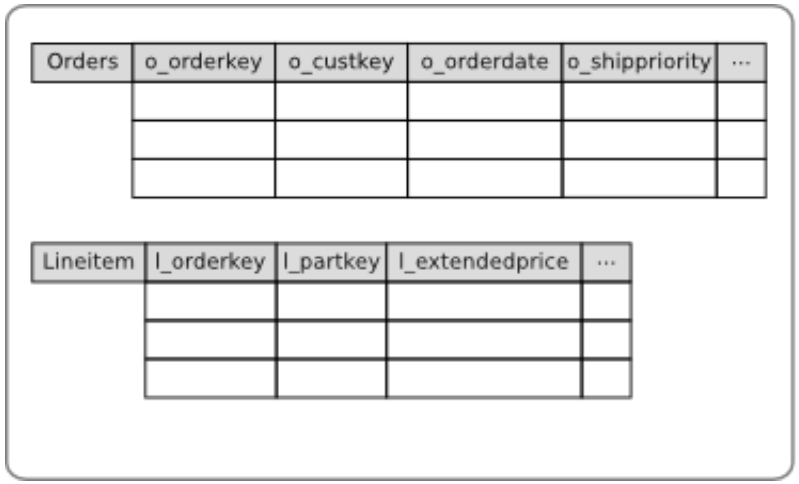


Input Splits

- What about problems that do not fit sorting or partitioning best?
- What about other techniques for data processing?

Example Task: A simple analytical Query

- Given a schema with two tables
- Simple query that joins them and computes an aggregate

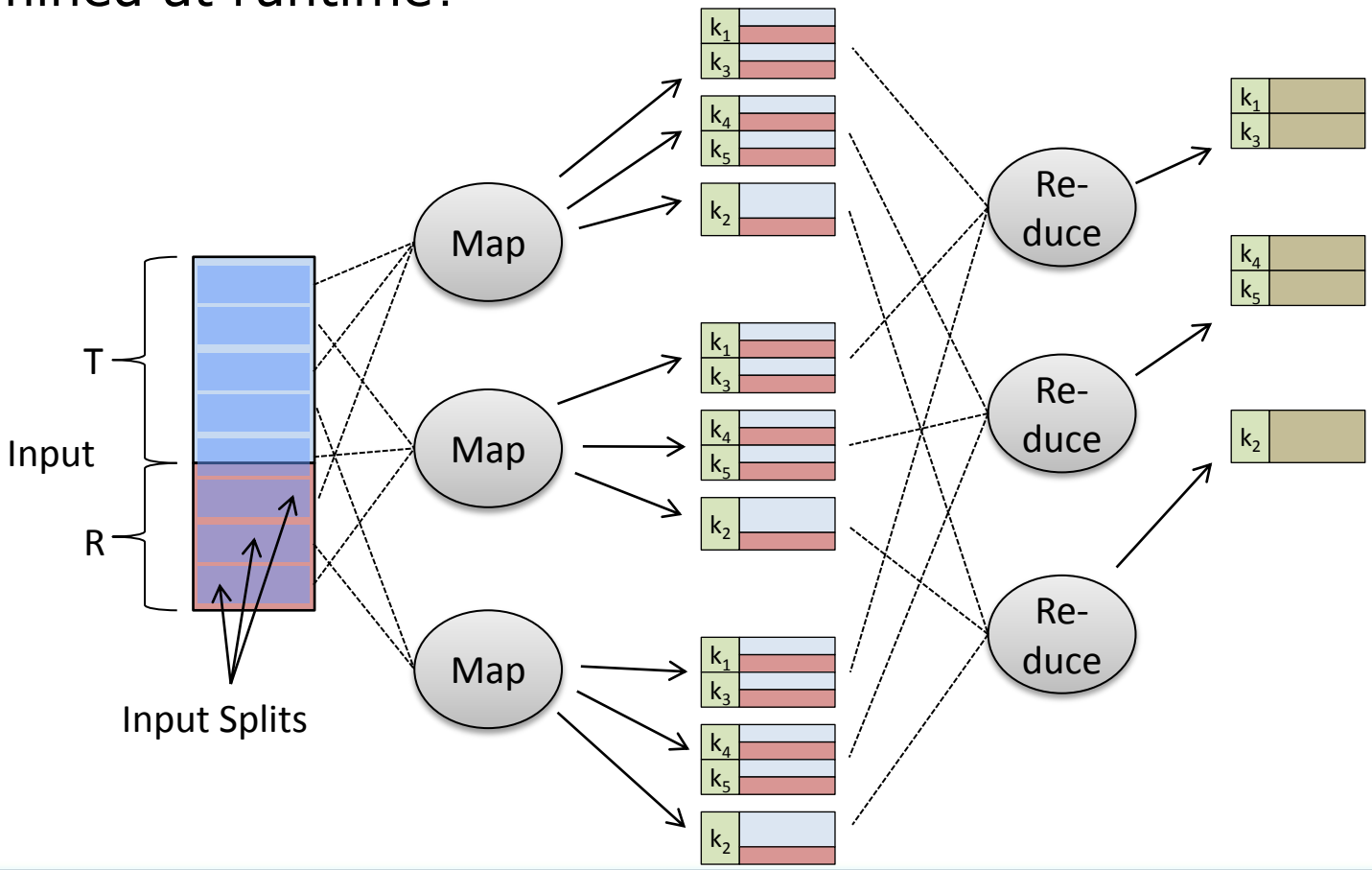



```
SELECT l_orderkey, o_shippriority,
    sum(l_extendedprice) as revenue

FROM orders, lineitem

WHERE l_orderkey = o_orderkey
    AND o_orderstatus = "X"
    AND YEAR(o_orderdate) = Y
    AND o_orderpriority LIKE "Z%"

GROUP BY l_orderkey, o_shippriority;
```

- How would one express that in MapReduce?
  - Different possibilities?
- How many jobs are required?

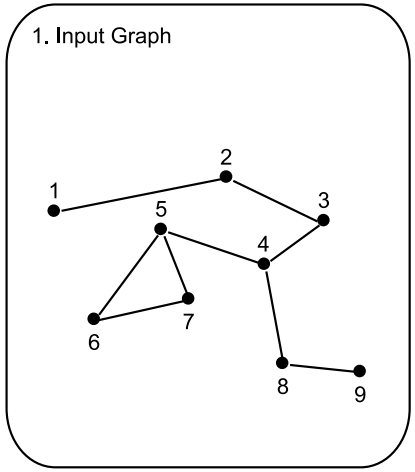
Typical Join Implementation in M/R

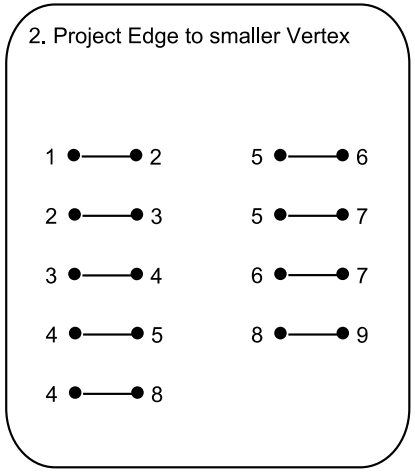
- Is that natural to program?
- What if one side is much smaller?
- What if the size is hard to be estimated and can only be determined at runtime?

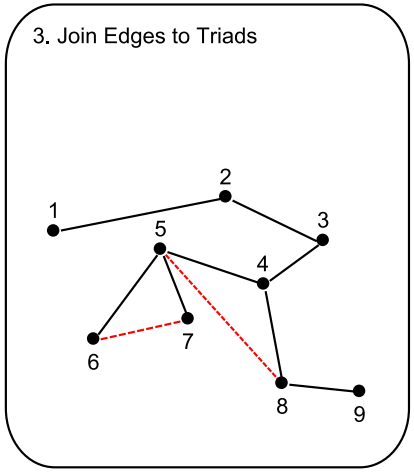


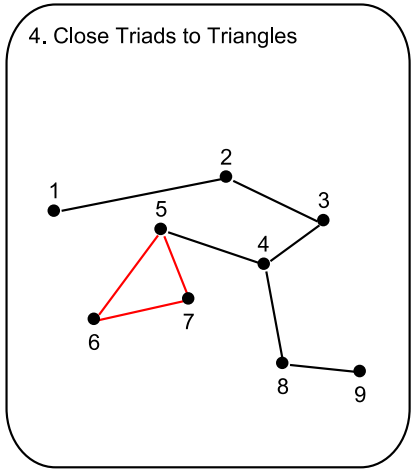
One more example: Triangle Enumeration

- Enumerating Triangles in a social graph is a frequent preprocessing step
- Basic procedure:





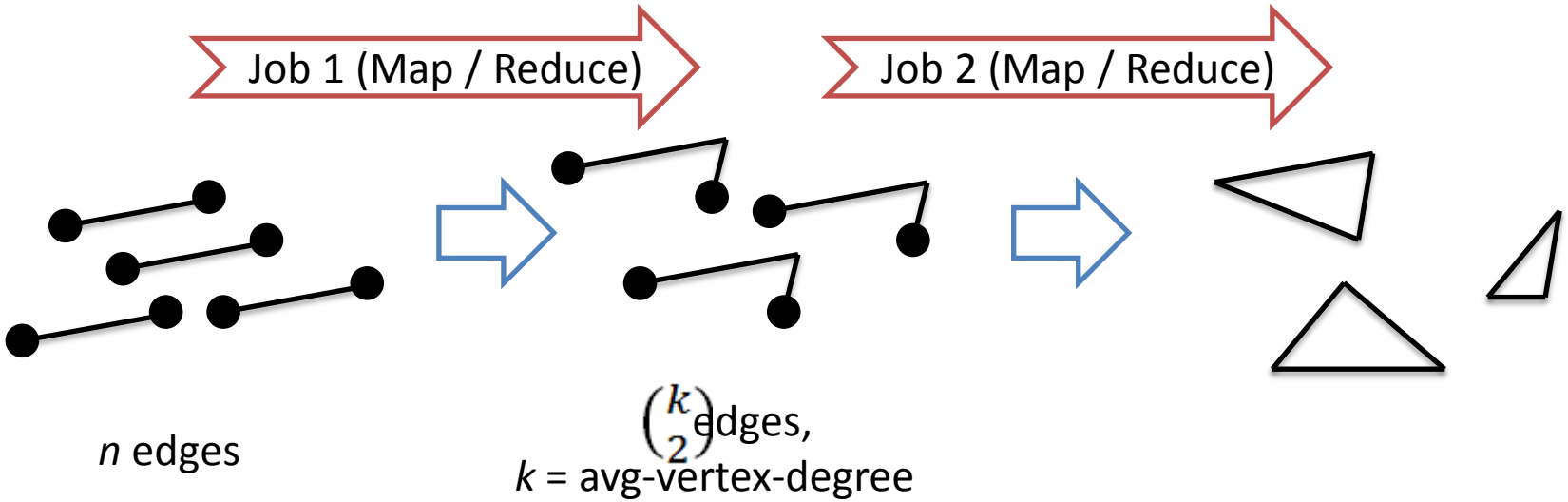




- Representation in MapReduce?

Triangle Enumeration continued…

- The plan is basically to subsequent joins
  - First one is a self joins among the edges
  - Second one joins open triads to edged



- What are the intermediate result sizes?
- How would the "join" be executed most efficiently?
- Is it a good idea to store the intermediate result in a DFS?

What can be improved?

Some points can be easily observed

1) Break the static execution pipeline
   - Support more options beside partitioning and sorting

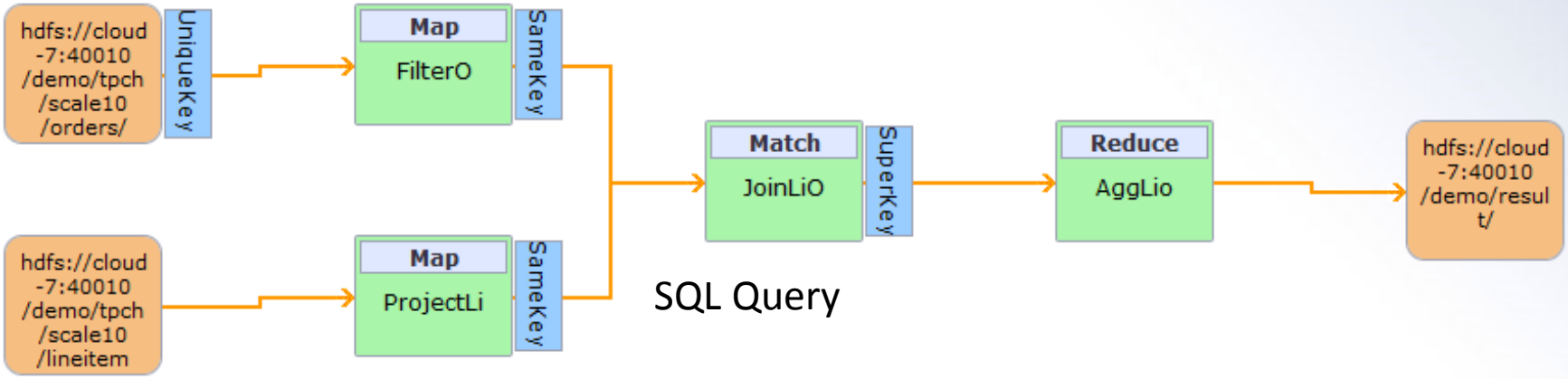
2) Encapsulate the parallelization requirement to "*match items*" with its semantics

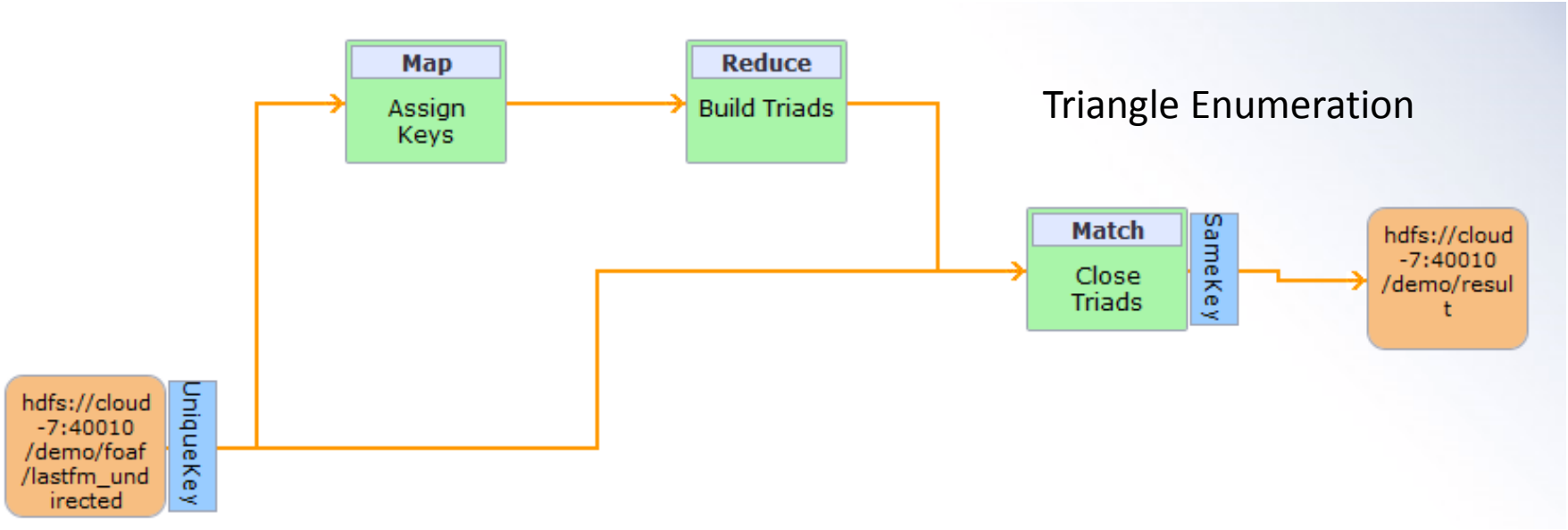
3) Support composed data-flows larger than the two-stage MapReduce pipeline
   - Don't write everything, but only where it is very valuable for recovery

Extending the MapReduce Idea…

THE STRATOSPHERE APPROACH

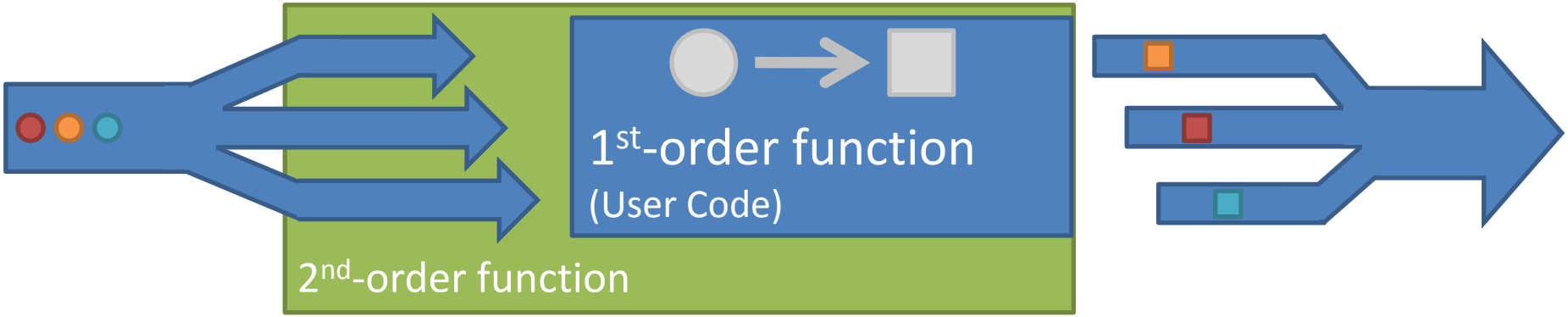
Examples in Stratosphere





Data Parallelism

- PACT is a generalization and extension of MapReduce
  - PACT inherits many concepts of MapReduce
- Both are inspired by functional programming
  - Fundamental concept of programming model are 2nd-order functions
  - User writes 1st-order functions (user functions)
  - User code can be arbitrarily complex
  - 2nd-order function calls 1st-order function with independent data subsets
  - No common state should be held between calls of user function



The Second-Order-Functions

- Define dependencies between the records that must be obeyed when splitting them into subsets
  - Cp: Required partition properties

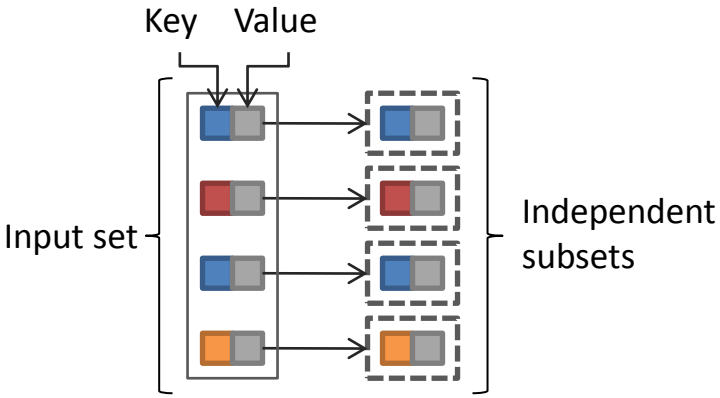
Map:

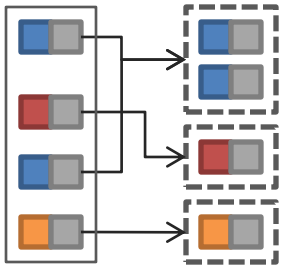
• All pairs are independently processed

Reduce:

• Pairs with identical key are grouped

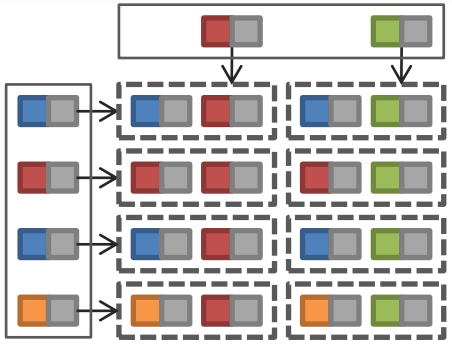
• Groups are independently processed



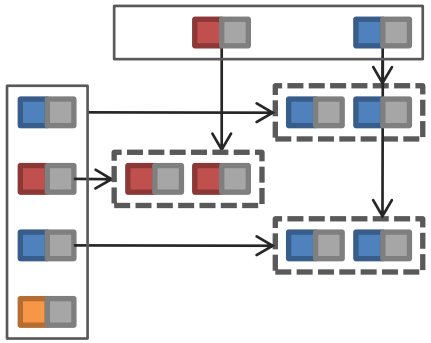


Input Contracts beyond Map and Reduce

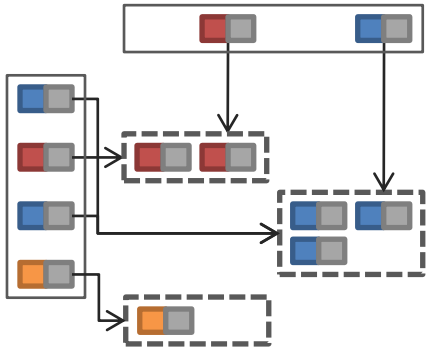
- Cross
  - Builds a Cartesian Product
  - Elements of CP are independently processed



- Match
  - Performs an equi-join on the key
  - Join candidates are independently processed

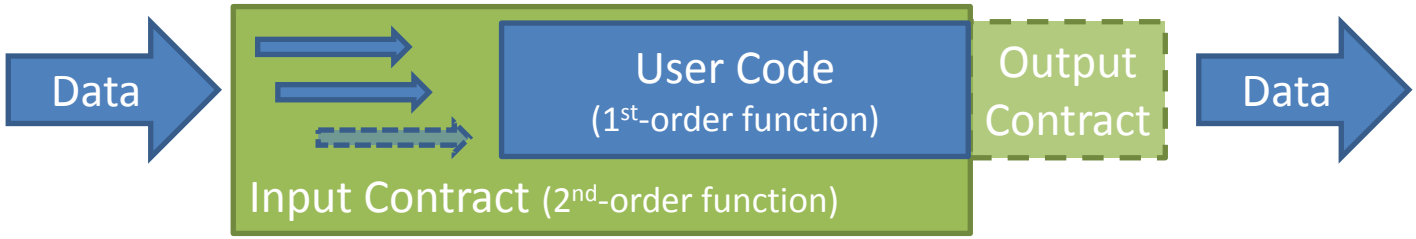


- CoGroup
  - Groups each input on key
  - Groups with identical keys are processed together



PACT Programming Model

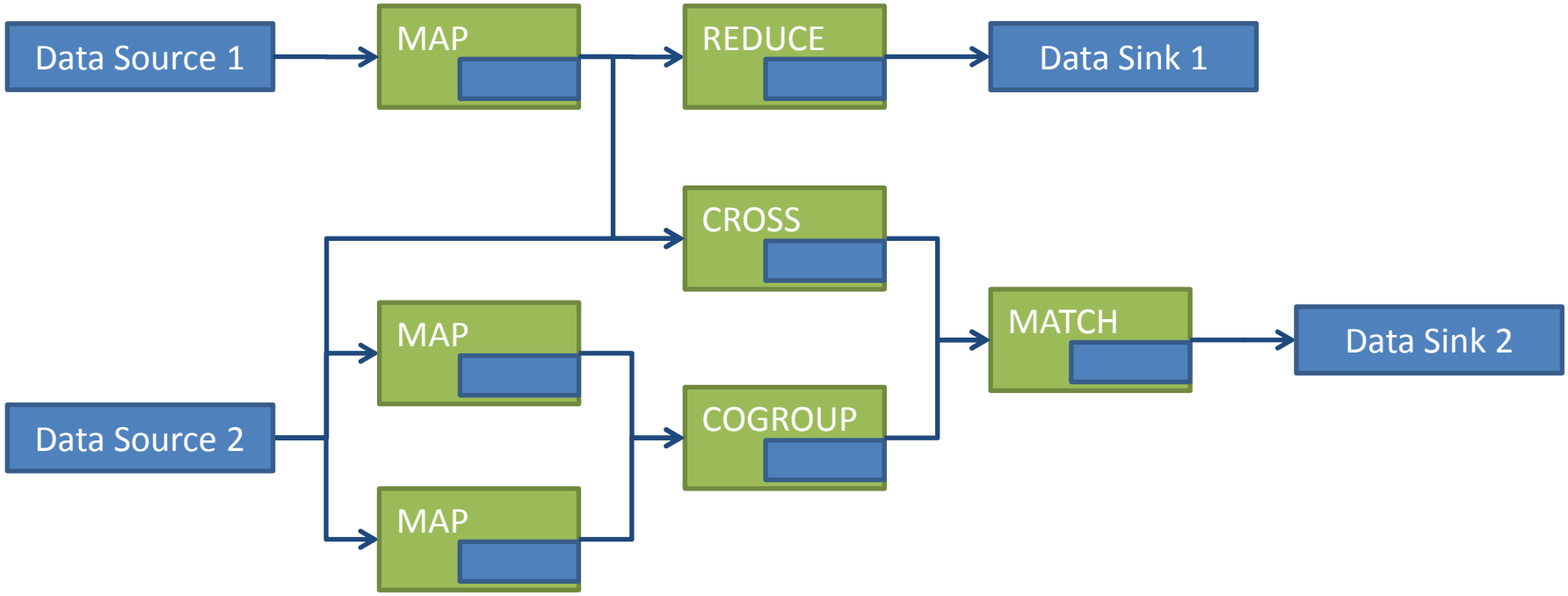
- Generalization and Extension of MapReduce
- Based on Parallelization Contracts (PACTs)



- Input Contract
  - 2nd-order function; generalization of Map and Reduce
  - Generates independently processable subsets of data
- User Code
  - 1st-order function
  - For each subset independently called
- Output Contract
  - Describes properties of the output of the 1st-order function
  - Optional but enables certain optimizations

PACT Programming Model

- PACT Programs are data flow graphs
  - Data comes from sources and flows to sinks
  - PACTs process data in-between sources and sinks
  - Multiple sources and sinks allowed
  - Arbitrary complex directed acyclic data flows can be composed



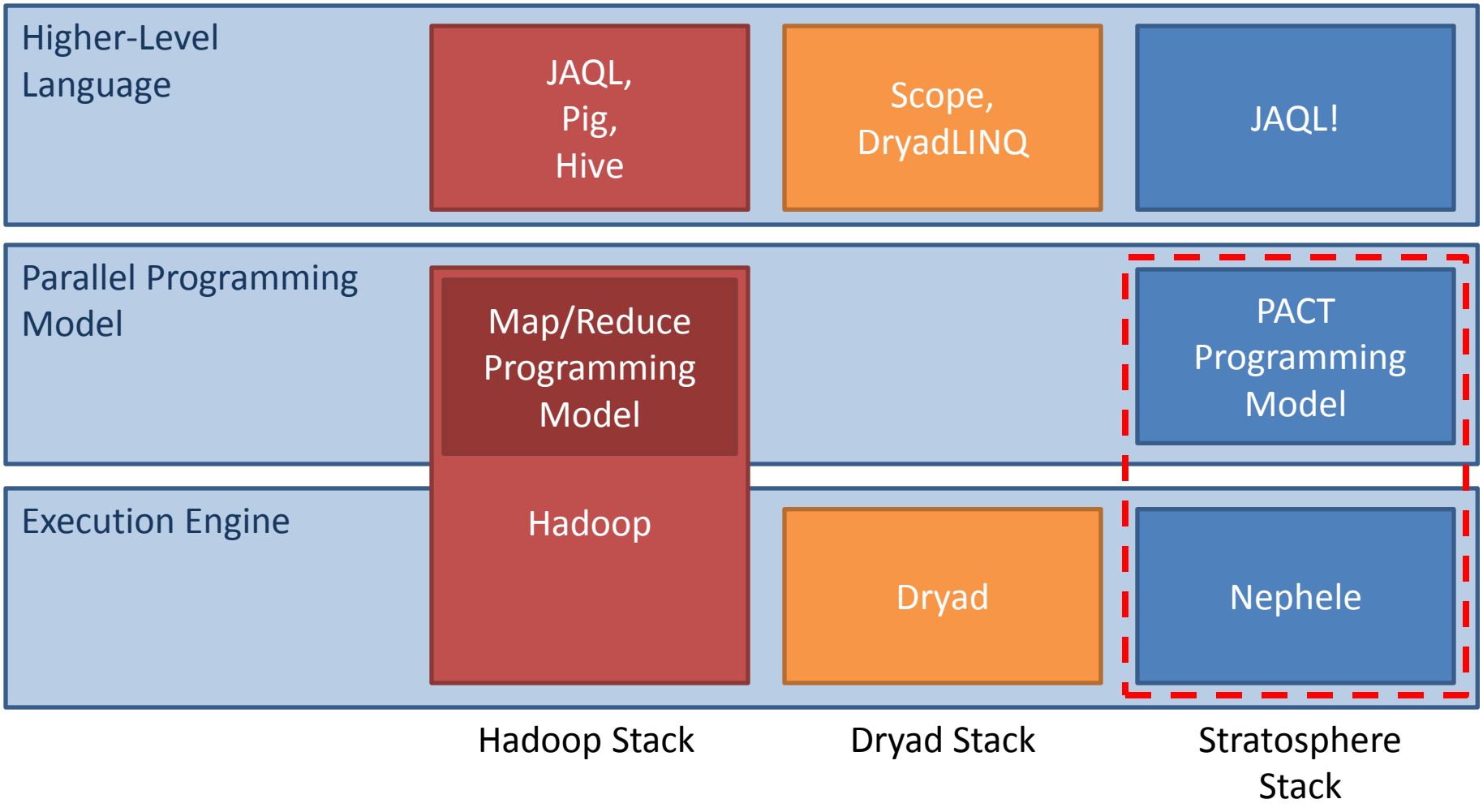
Output Contracts

- Same-Key
  - User Function does not alter the key

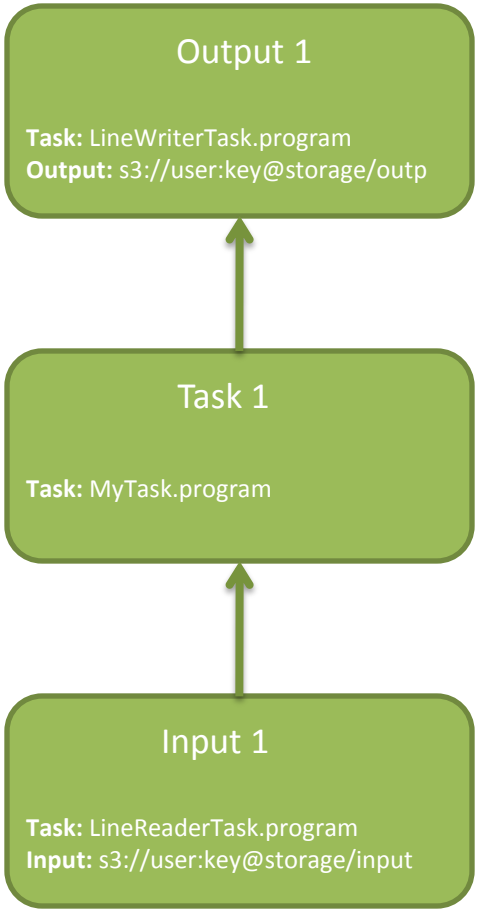
- Super-Key
  - Key generated by UF is a super-key of the input key

- Unique-Key
  - Data source or UF produces unique keys

Architecture Overview

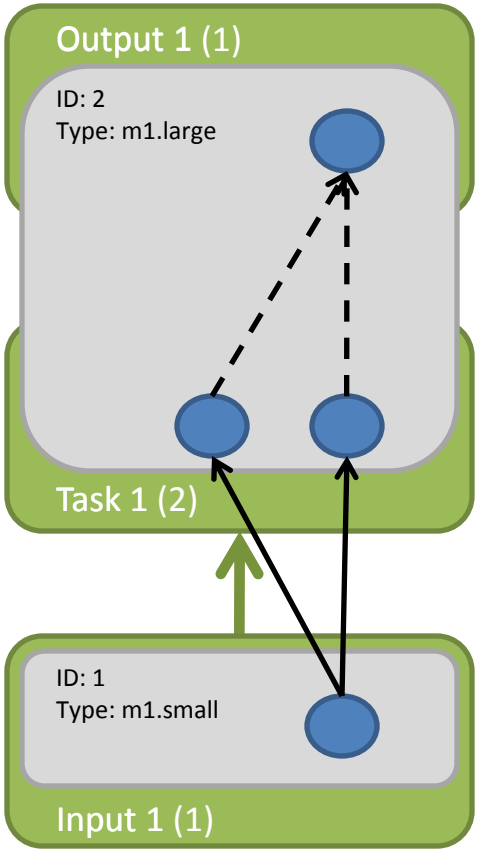


Structure of a Nephele Schedule



- Nephele Schedule is represented as DAG
  - Vertices represent tasks
  - Edges denote communication channels
- Mandatory information for each vertex
  - Task program
  - Input/output data location (I/O vertices only)
- Optional information for each vertex
  - Number of subtasks (degree of parallelism)
  - Number of subtasks per virtual machine
  - Type of virtual machine (#CPU cores, RAM…)
  - Channel types
  - Sharing virtual machines among tasks

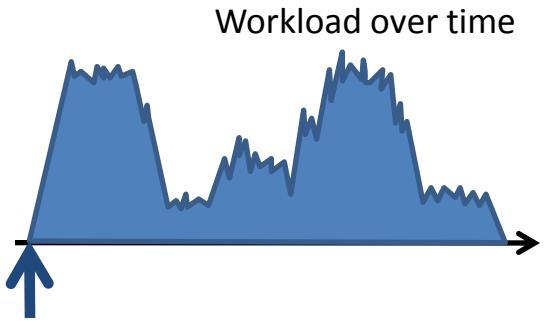
Internal Schedule Representation

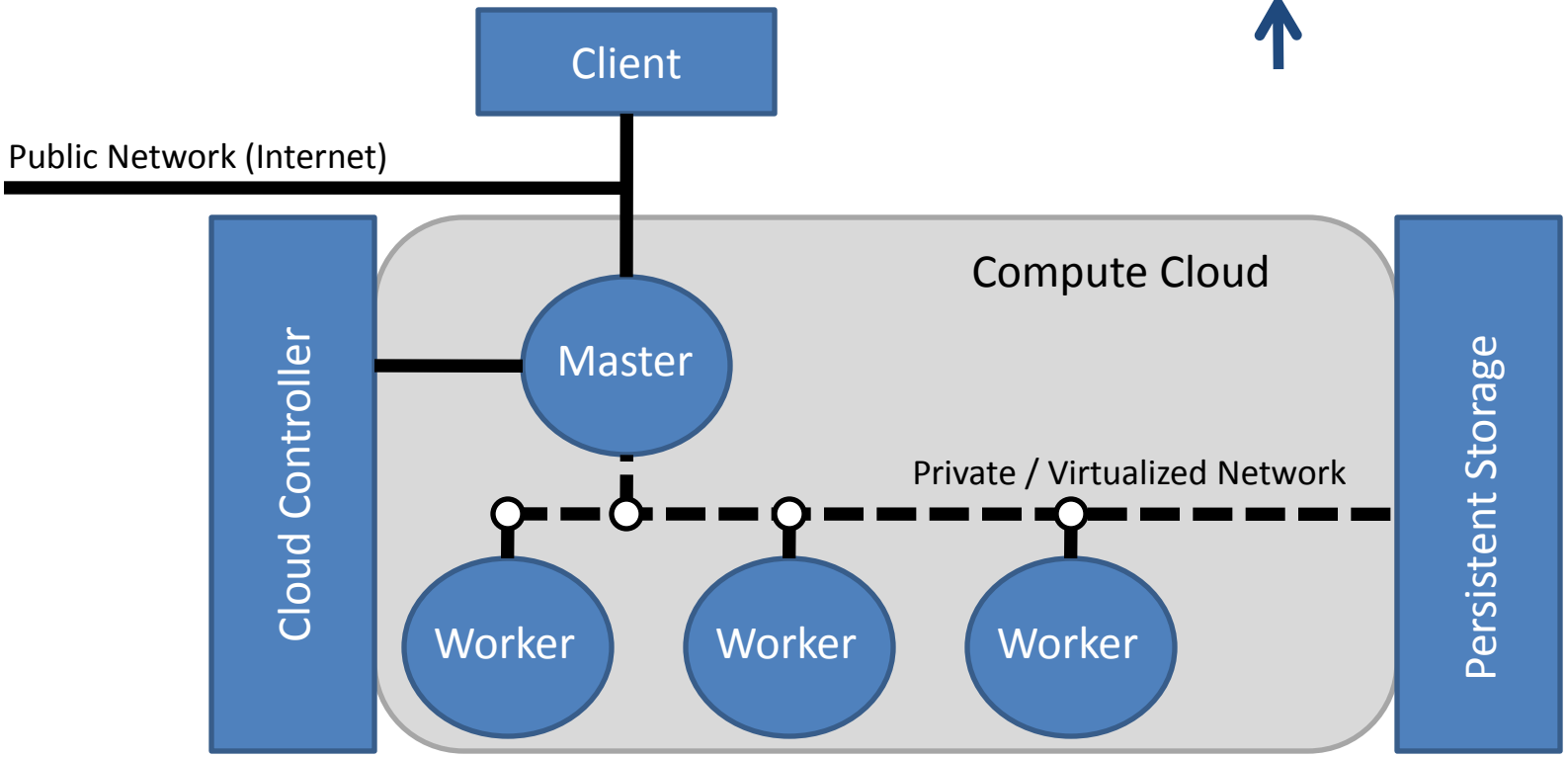


- Nephele schedule is converted into internal representation
- Explicit parallelization
  - Parallelization range (mpl) derived from PACT
  - Wiring of subtasks derived from PACT
- Explicit assignment to virtual machines
  - Specified by ID and type
  - Type refers to hardware profile

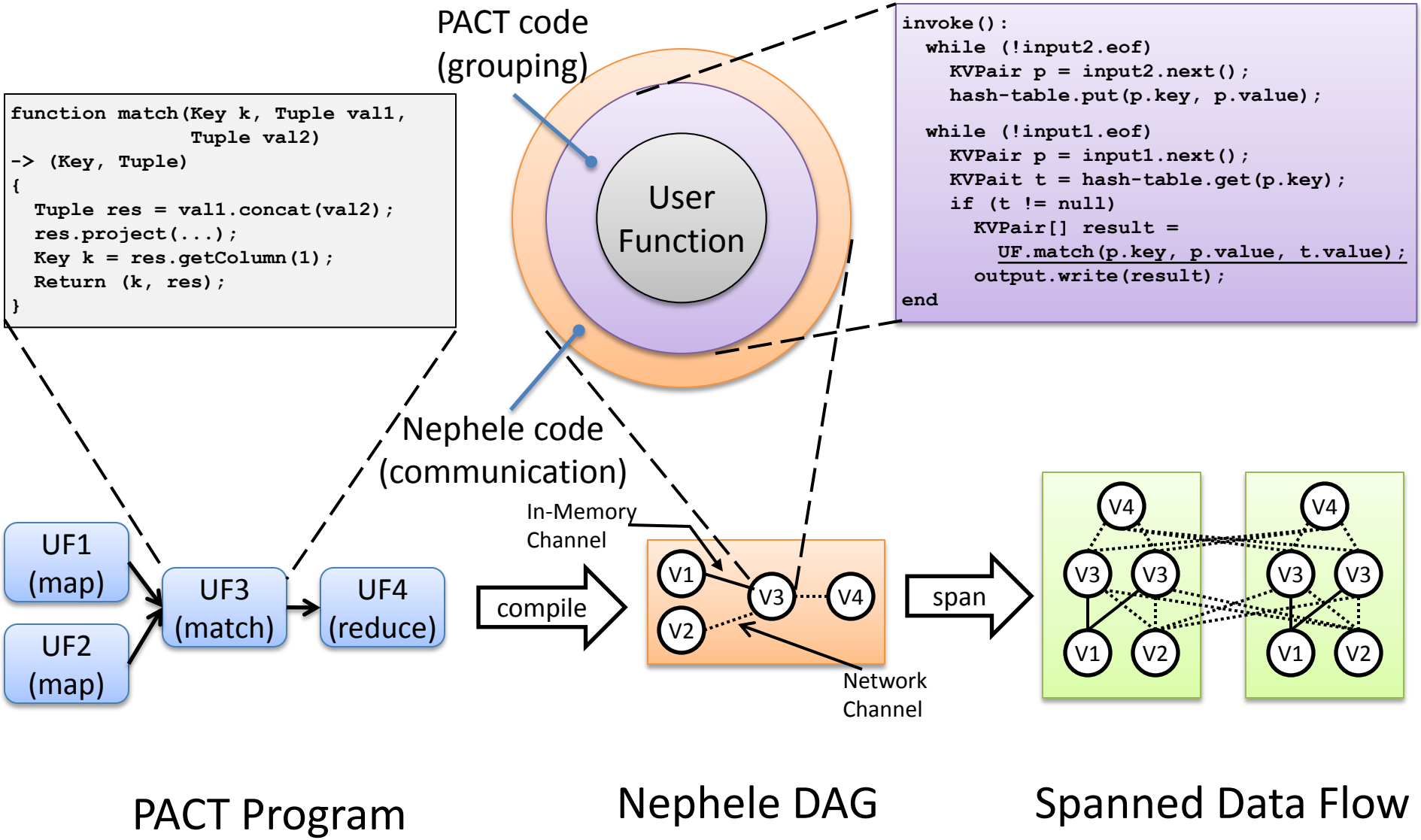
Nephele Architecture

- Standard master worker pattern
- Workers can be allocated on demand





From PACT Programs to Data Flows



Optimizing PACT Programs

- For certain PACTs, several distribution patterns exist that fulfill the contract
  - Choice of best one is up to the system
- Created properties (like a partitioning) may be reused for later operators
  - Need a way to find out whether they still hold after the user code
  - Output contracts are a simple way to specify that
  - Example output contracts: Same-Key, Super-Key, Unique-Key
- Using these properties, optimization across multiple PACTs is possible
  - Simple System-R/Volcano style optimizer approach possible

Download and try Stratosphere

Get a binary or clone the source at http://stratosphere.eu

Website provides

- A lot of user documentation
- Several illustrated examples
- Architectural details
- Guide how get started with the code, if you want to extend the system

Nice Features

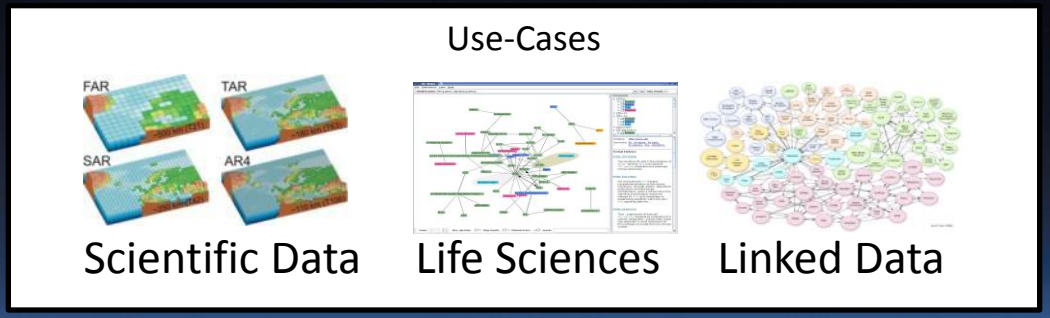
Execute jobs via

- Command line client
- Web GUI with plan visualization

Different modes

- A local mode that starts a small-scale version everything in a single JVM (no reconfiguration needed)
- A Cluster-Mode that uses a pool of machines
  - Machines may be heterogeneous. Matching hardware profiles allows to use them optimally together
- A cloud mode that automatically allocates as many machines as needed from a cloud-controller

The Stratosphere Project*



- Explore the power of Cloud computing for complex information management applications
- Database-inspired approach
- Analyze, aggregate, and query
- Textual and (semi-) structured data
- Research and prototype a web-scale data analytics infrastructure

* FOR 1306: DFG funded collaborative project among TU Berlin (Markl, Kao), HU Berlin (Freytag, Leser) and HPI Potsdam(Naumann)