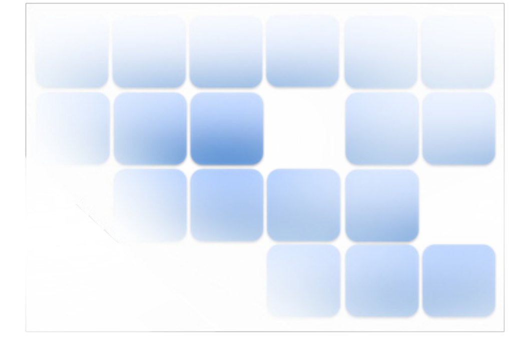
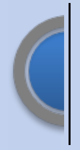
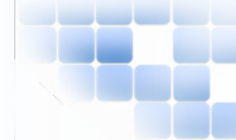


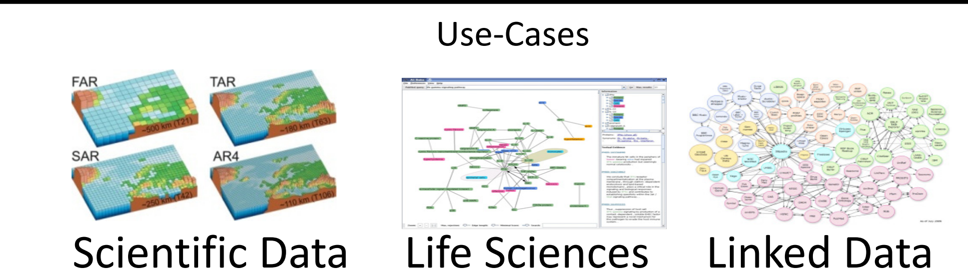
Massively Parallel Analytics beyond Map/Reduce

Stephan Ewen
Fabian Hüske
Odej Kao
Volker Markl
Daniel Warneke

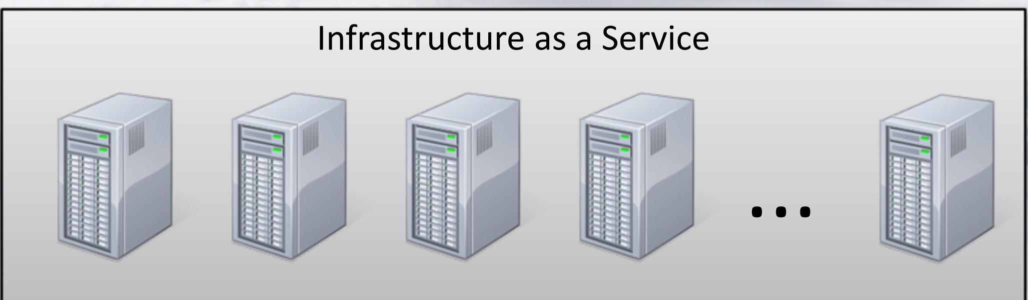


The Stratosphere Project*





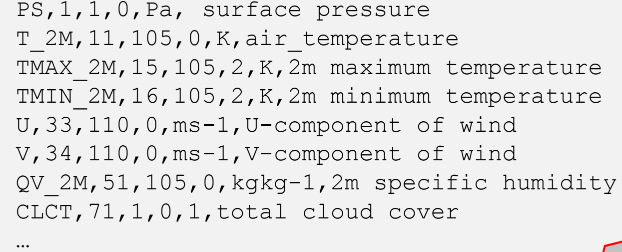
StratoSphere
Above the Clouds
Query Processor



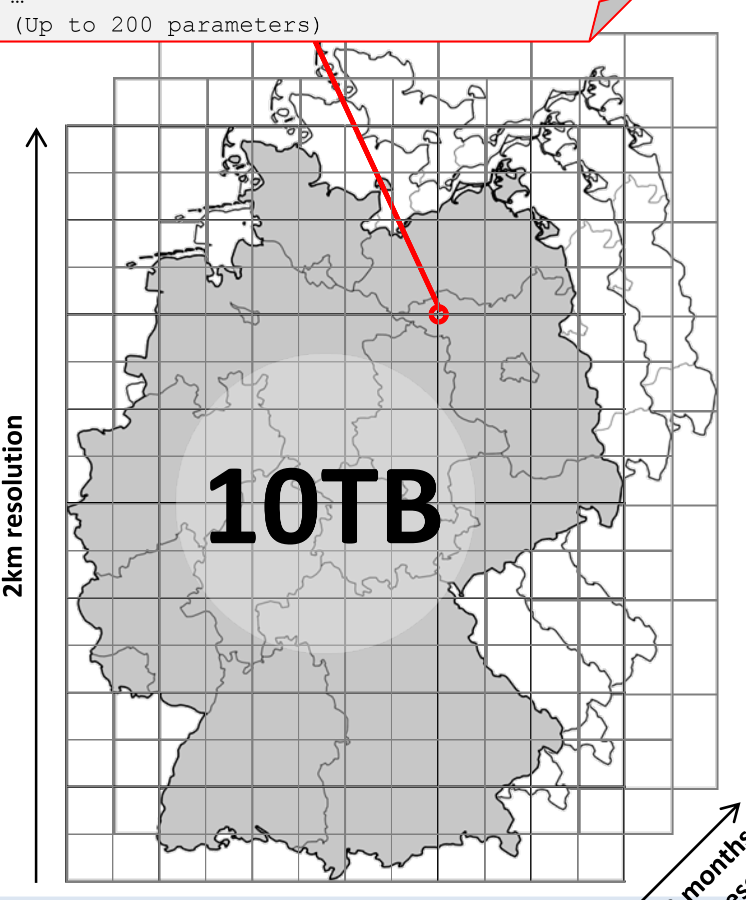
- Explore the power of Cloud computing for complex information management applications
- Database-inspired approach
- Analyze, aggregate, and query
- Textual and (semi-) structured data
- Research and prototype a web-scale data analytics infrastructure

* publically funded joint project with HU Berlin (C. Freytag, U. Leser) and HPI (F. Naumann)

Example: Climate Data Analysis



```
PS,1,1,0,Pa, surface pressure
T_2M,11,105,0,K,air_temperature
TMAX_2M,15,105,2,K,2m maximum temperature
TMIN_2M,16,105,2,K,2m minimum temperature
U,33,110,0,ms-1,U-component of wind
V,34,110,0,ms-1,V-component of wind
QV_2M,51,105,0,kgkg-1,2m specific humidity
CLCT,71,1,0,1,total cloud cover
…
(Up to 200 parameters)
```



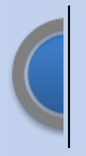
Analysis Tasks on Climate Data Sets

- Validate climate models
- Locate „hot-spots“ in climate models
  - Monsoon
  - Drought
  - Flooding
- Compare climate models
  - Based on different parameter settings

Necessary Data Processing Operations

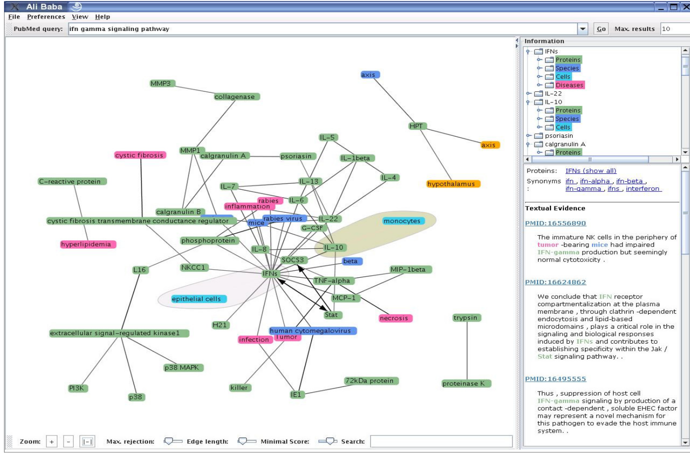
- Filter
- Aggregation (sliding window)
- Join

- Multi-dimensional sliding-window operations
- Geospatial/Temporal joins
- Uncertainty

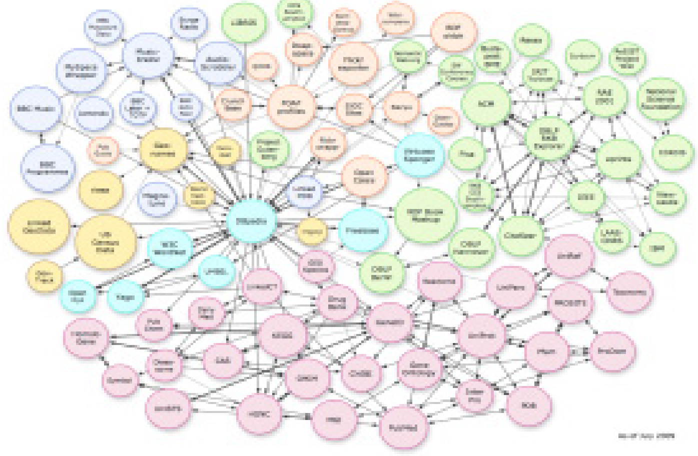


Further Use-Cases

- Text Mining in the biosciences



- Cleansing of linked open data



Outline

- Motivation for Stratosphere
- Architecture of the Stratosphere System
- The PACT Programming Model
- The Nephele Execution Engine
- Parallelizing PACT Programs

TPC-H Aggregation Query using MapReduce

```
SELECT l_orderkey, o_shippriority, sum(l_extendedprice) AS revenue
  FROM orders O, lineitem Li
  WHERE l_orderkey = o_orderkey AND
     o_custkey IN [X] AND o_orderdate > [Y]
  GROUP BY l_orderkey, o_shippriority
```

- Partial Aggregation extendedprice

REDUCE

COMBINE

- Final Aggregation
- Project for output

For Orders:
- Filter & Project O
- Flag with 'O'
- Set key to orderkey

For Lineitems:
- Project Li
- Flag with 'L'
- Set key to orderkey

MAP

- Set key to orderkey, shippriority

REDUCE

- Concatenate 'O'-flagged tuples and L'-flagged tuples

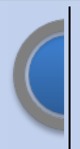
MAP

- Read file from DFS

Input O

Input Li

- Read file from DFS

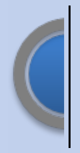


TPC-H Aggregation Query on Hadoop

REDUCE
COMBINE
MAP
REDUCE
MAP
Input O
Input Li

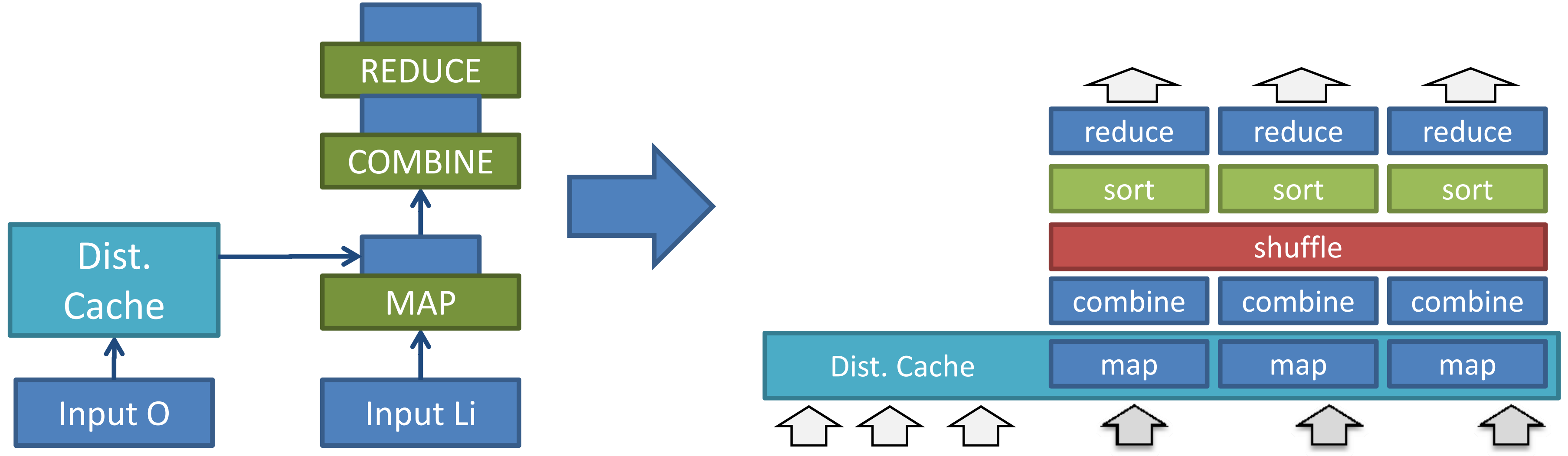
reduce reduce reduce
sort sort sort
shuffle
combine combine combine
map map map
HDFS
reduce reduce reduce
sort sort sort
shuffle
map map map

- Data is shuffled twice
- Intermediate result is written to HDFS

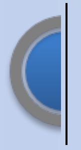


TPC-H Aggregation Query - Alternative

Broadcast strategy using Hadoop's Distributed Cache:



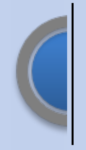
- Only one MapReduce job
  - Data is shuffled once
  - No intermediate result is written to HDFS
  - Efficient if Orders is comparably small
- Hadoop does not know broadcast shipping strategy



Motivation for Stratosphere System

- Complex data processing must be pushed into Map/Reduce
  - Developer must care about parallelization
  - Developer has to know how the execution framework operates
  - Framework does not know what is happening
  - Examples:
    - Tasks with multiple input data sets (join and cross operations)
    - Custom partitioning (range partitioning, window operations)

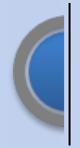
- Static execution strategy
  - Gives fault-tolerance but not necessarily best performance
  - Developer has to hard-code own strategies
    - Broadcast strategy using the distributed cache
  - No automatic optimization can be applied
  - Results of research on parallel databases are neglected



Architecture Overview

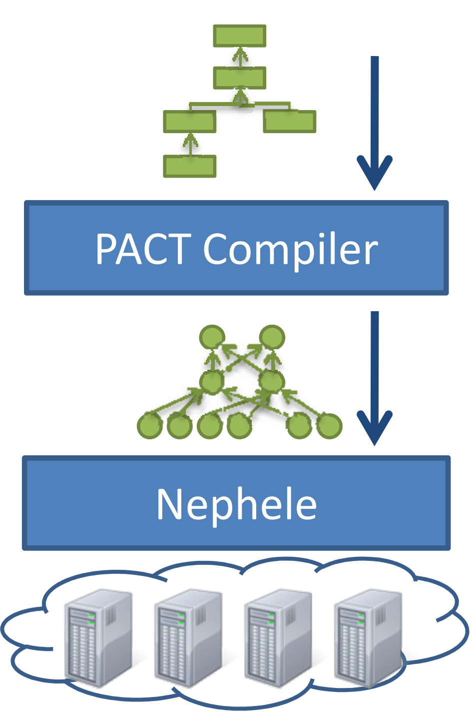
	Hadoop Stack	Dryad Stack	Stratosphere Stack
Higher-Level Language	JAQL, Pig, Hive	Scope, DryadLINQ	JAQL? Pig? Hive?
Parallel Programming Model	Map/Reduce Programming Model		PACT Programming Model
Execution Engine	Hadoop	Dryad	Nephele

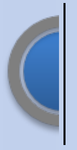
This Talk



Stratosphere in a Nutshell

- PACT Programming Model
  - Parallelization Contract (PACT)
  - Declarative definition of data parallelism
  - Centered around second-order functions
  - Generalization of map/reduce
- Nephele
  - Dryad-style execution engine
  - Evaluates dataflow graphs in parallel
  - Data is read from distributed filesystem
  - Flexible engine for complex jobs
- Stratosphere = Nephele + PACT
  - Compiles PACT programs to Nephele dataflow graphs
  - Combines parallelization abstraction and flexible execution
  - Choice of execution strategies gives optimization potential



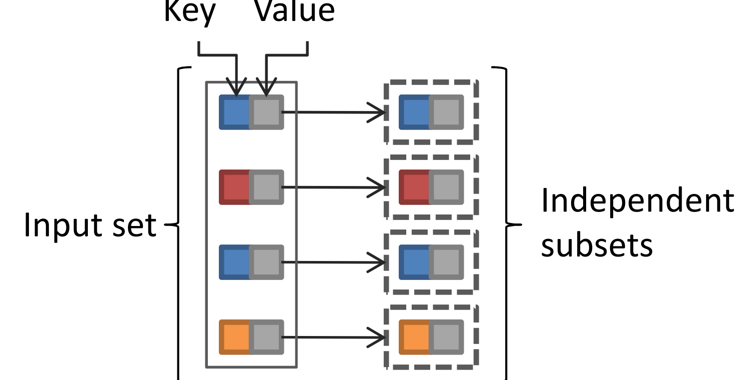


An Intuition for Parallelization Contracts (PACTs)

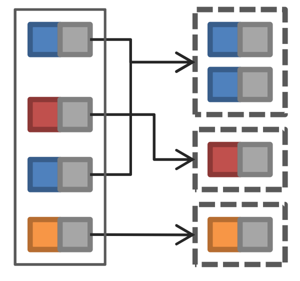
- Map and reduce are second-order functions
  - Call first-order functions (user code)
  - Provide first-order functions with subsets of the input data

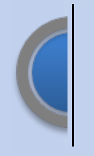
- Map and reduce are PACTs in our context

- Map
  - All pairs are independently processed



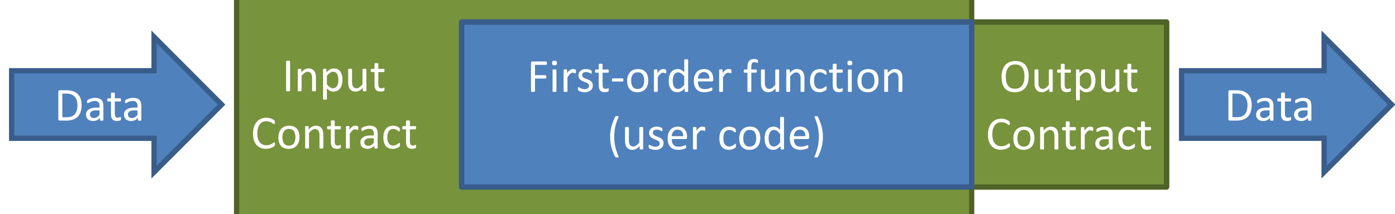
- Reduce
  - Pairs with identical key are grouped
  - Groups are independently processed



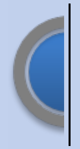


What is a PACT?

- Second-order function that defines properties on the input and output data of its associated first-order function

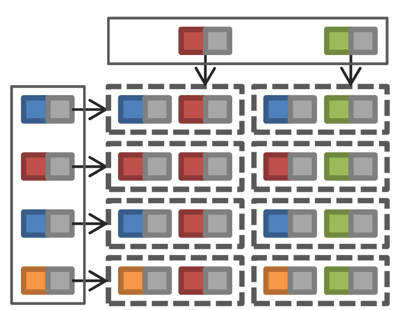


- Input Contract
  - Generates independently processable subsets of data
  - Generalization of map/reduce
  - Enforced by the system
- Output Contract
  - Generic properties that are preserved or produced by the user code
  - Use is optional but enables certain optimizations
  - Guaranteed by the developer
- Key-Value data model



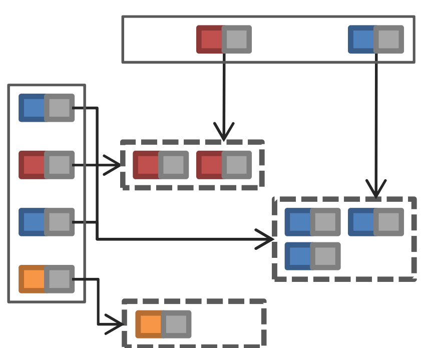
PACTs beyond Map and Reduce

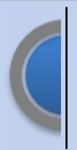
- **Cross**
  - Multiple inputs
  - Cartesian Product of inputs is built
  - All combinations are processed independently



- **Match**
  - Multiple inputs
  - All combinations of pairs with identical key over all inputs are built
  - All combinations are processed independently
  - Contract resembles an equi-join on the key

- **CoGroup**
  - Multiple inputs
  - Pairs with identical key are grouped for each input
  - Groups of all inputs with identical key are processed together





TPC-H Aggregation Query using PACTs

```
SELECT l_orderkey, o_shippriority, sum(l_extendedprice) AS revenue
  FROM orders O, lineitem Li
  WHERE l_orderkey = o_orderkey AND
    o_custkey IN [X] AND o_orderdate > [Y]
  GROUP BY l_orderkey, o_shippriority
```

- Final Aggregate
- Project for output

REDUCE

- Partial Aggregate extendedprice

COMBINE

- Concat O and Li
- Set Key to (orderkey, shippriority)

MATCH

- Filter O
- Project O
- Set key to orderkey

MAP

MAP

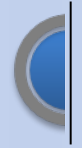
- Project Li
- Set key to orderkey

- Read file from DFS

Input O

Input Li

- Read file from DFS



K-Means Iteration using PACTs

Output Centers

(cid,cpos)

REDUCE

- Compute new center positions from ppos

(cid,ppos)

- Find nearest cluster center
- Set key to cid

REDUCE

(pid,(ppos,cid,d))

- Compute distance d
- Set key to pid

CROSS

(cid,cpos)

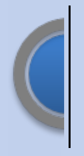
(pid,ppos)

- Read or generate cluster centers

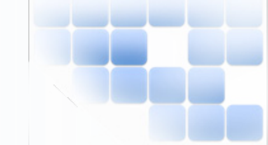
Input Centers

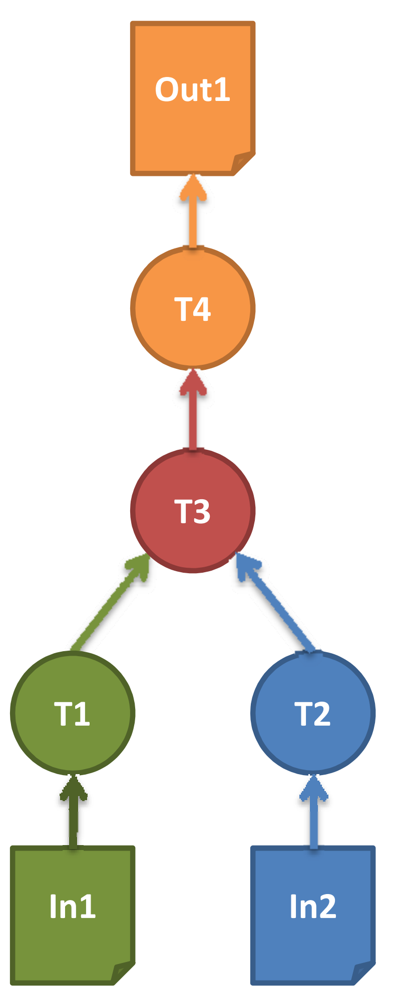
Input Data Points

- Read data points



Nephele Execution Engine





- Evaluates data flow graphs in parallel
- Vertices represent tasks
  - Tasks run user code
- Edges denote communication channels
  - Network, In-Memory, and File Channels
- Rich set of vertex annotations provide fine-grained control over parallelization
  - 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

From PACT Programs to Parallel Data Flows

PACT code
(grouping)

```
function match(Key k, Tuple val1,
               Tuple val2)
-> (Key, Tuple)
{
  Tuple res = val1.concat(val2);
  res.project(...);
  Key k = res.getColumn(1);
  Return (k, res);
}
```

User
Function

```
invoke():
  while (!input2.eof)
    KVPair p = input2.next();
    hash-table.put(p.key, p.value);
  while (!input1.eof)
    KVPair p = input1.next();
    KVPait t = hash-table.get(p.key);
    if (t != null)
      KVPair[] result =
        UF.match(p.key, p.value, t.value);
      output.write(result);
end
```

Nephele code
(communication)

UF1
(map)

UF2
(map)

UF3
(match)

UF4
(reduce)

compile

In-Memory
Channel

V1 V2 V3 V4

Network
Channel

span

V4 V3 V3 V1 V2 V4 V3 V3 V1 V2

PACT Program

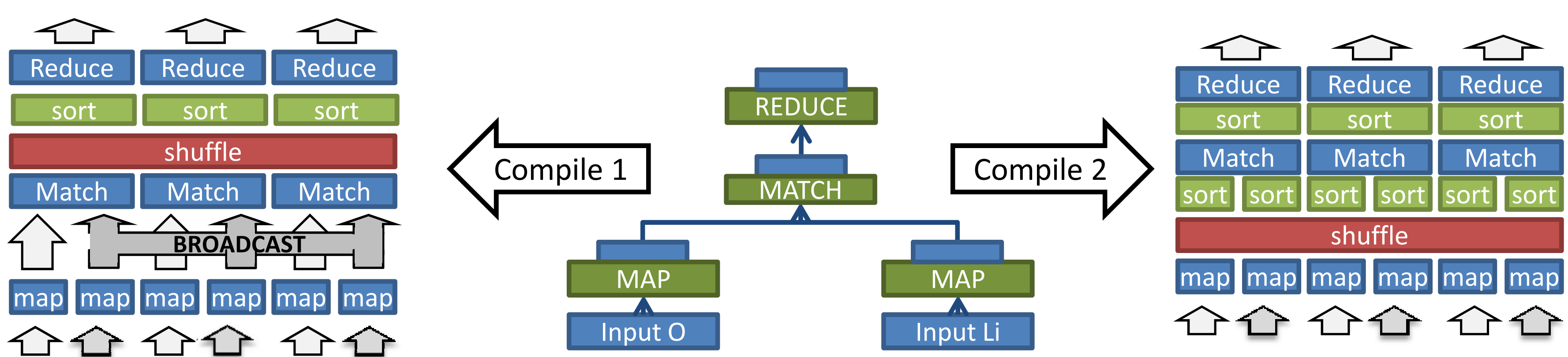
Nephele Schedule

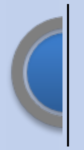
Spanned Data Flow

Optimization Potential

- Optimization of a Single PACT
  - PACTs can be evaluated with multiple execution strategies
  - Data shipping strategies (Repartition / Broadcast / SFR / Ring / …)
  - Local processing strategies (Sorting / HybridHash/ MMHash / …)

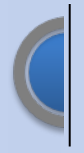
- Optimization across PACTs
  - PACTs sort and partition the data
  - Optimizer considers properties of the data (Sorting / Partitioning)
    - Output contracts give hints
    - Reuse existing properties to obtain better plans





Next Steps on Research Agenda

- Additional Input Contracts
  - Definition of Input Contracts is general
  - Analyze use-cases to derive new requirements
  - Examples: Window Reducer, Fuzzy Matcher
- Flexible Checkpointing & Recovery
  - Find balance between checkpoint-everything and checkpoint-nothing
  - Dynamically manage risk of node failure
- Robust & Adaptive Execution
  - Input data and user functions are not well known
  - Generate plans with adequate worst-case behavior
  - Generate plans that can be easily adapted
  - Manage risk and opportunity



OpenSource and Stratosphere

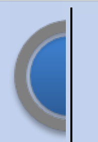
- Stratosphere is built upon OpenSource components
  - HDFS used as distributed filesystem
  - Nephele employs Hadoop IPC Communication Layer
  - Support for Apache Avro serialization framework is planned
- Stratosphere can benefit from Hadoop Ecosystem
  - PACTs are generalization of MapReduce
  - PACT forks of popular Hadoop projects might come up
- Stratosphere going OpenSource?
  - Aiming for release by end of 2010

Summary

- PACT Programming Model
  - Generalizes Map/Reduce
  - Abstracts parallelization of more complex data processing tasks

- PACT Program Execution
  - Optimization of PACT programs
  - Avoiding unnecessary shipping and processing
  - Nephele provides very flexible execution of programs

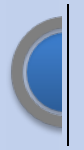
Stratosphere combines
Map/Reduce and parallel database
technology



Questions?

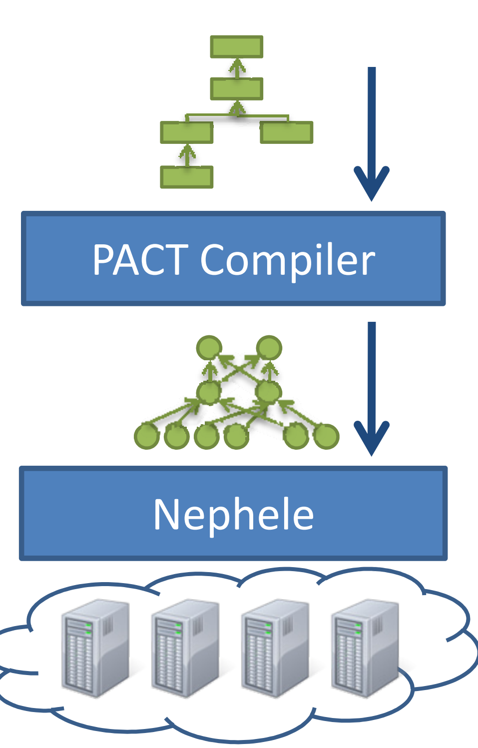
BACKUP

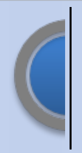
NEPHELE EXECUTION ENGINE



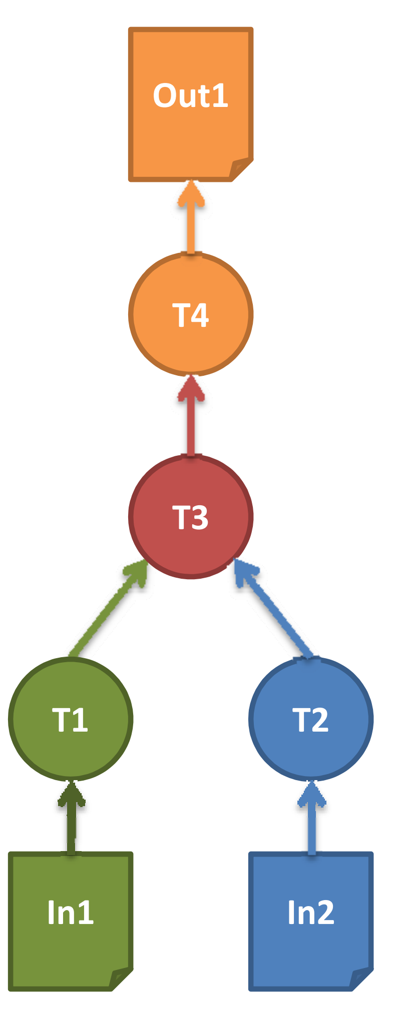
Nephele Execution Engine

- Executes Nephele schedules
  - compiled from PACT programs
- Design goals
  - Exploit scalability/flexibility of clouds
  - Provide predictable performance
  - Efficient execution on 1000+ nodes
  - Introduce flexible fault tolerance mechanisms
- Inherently designed to run on top of an IaaS Cloud
  - Can exploit on-demand resource allocation
  - Heterogeneity through different types of VMs possible
  - Knows Cloud's pricing model

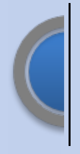




Structure of a Nephele Schedule



- Nephele Schedule is represented as DAG
- Vertices represent tasks
  - Tasks run user code
- Edges denote communication channels
  - Network, In-Memory, and File Channels
- Rich set of vertex annotations provide fine-grained control over parallelization
  - 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



Parallel Execution of a Nephele Schedule

- Nephele transforms schedule to parallel execution graph
  - Vertices are multiplied – Tasks are split up into data-parallel subtasks
  - Edges are added to connect subtasks (following distribution patterns)
- Subtasks are assigned to Nephele workers
  - Nephele ships user code for tasks
  - Nephele manages communication within and across nodes

BACKUP

PARALLELIZING THE EXECUTION

From PACT Programs to Parallel Data Flows

PACT code (grouping)

```
function match(Key k, Tuple val1,
               Tuple val2)
-> (Key, Tuple)
{
  Tuple res = val1.concat(val2);
  res.project(...);
  Key k = res.getColumn(1);
  Return (k, res);
}
```

User Function

```
invoke():
  while (!input2.eof)
    KVPair p = input2.next();
    hash-table.put(p.key, p.value);
  while (!input1.eof)
    KVPair p = input1.next();
    KVPait t = hash-table.get(p.key);
    if (t != null)
      KVPair[] result =
        UF.match(p.key, p.value, t.value);
      output.write(result);
end
```

Nephele code (communication)

UF1 (map)

UF2 (map)

UF3 (match)

UF4 (reduce)

compile

In-Memory Channel

V1 V2 V3 V4

Network Channel

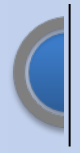
span

V4 V3 V3 V1 V2 V4 V3 V3 V1 V2

PACT Program

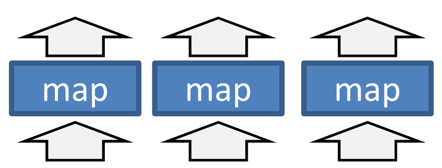
Nephele Schedule

Spanned Data Flow

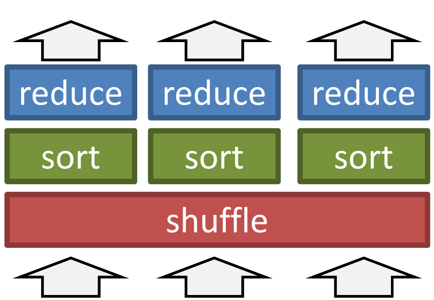


Parallelizing / Executing the Functions

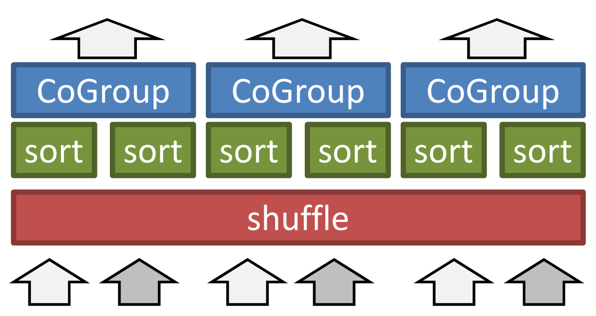
- Parallelizing *Map* is trivial
  - No dependencies between the records

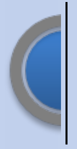


- Parallelizing *Reduce* is known business
  - Input partitioned across all nodes by key
  - Locally group by key via sorting or hashing



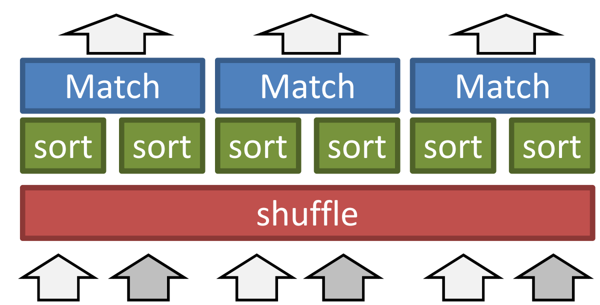
- Parallelizing *CoGroup* is analog to *Reduce*
  - Treat both inputs as in the Reduce function
  - Interleave the streams (zig-zag-fashion)



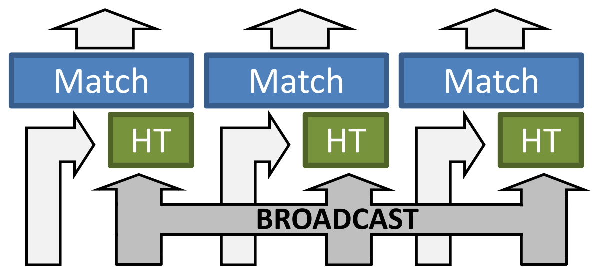


Parallelizing / Executing the Functions *(cont.)*

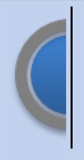
- Parallelizing *Match:*
  - Either partition both sides on the key
  - Or broadcast one side
  - Similar to parallel join optimization in DBMS



- *Match*ing key/value pairs
  - Sort and merge
  - Hash one side
  - Similar to local join optimization in DBMS

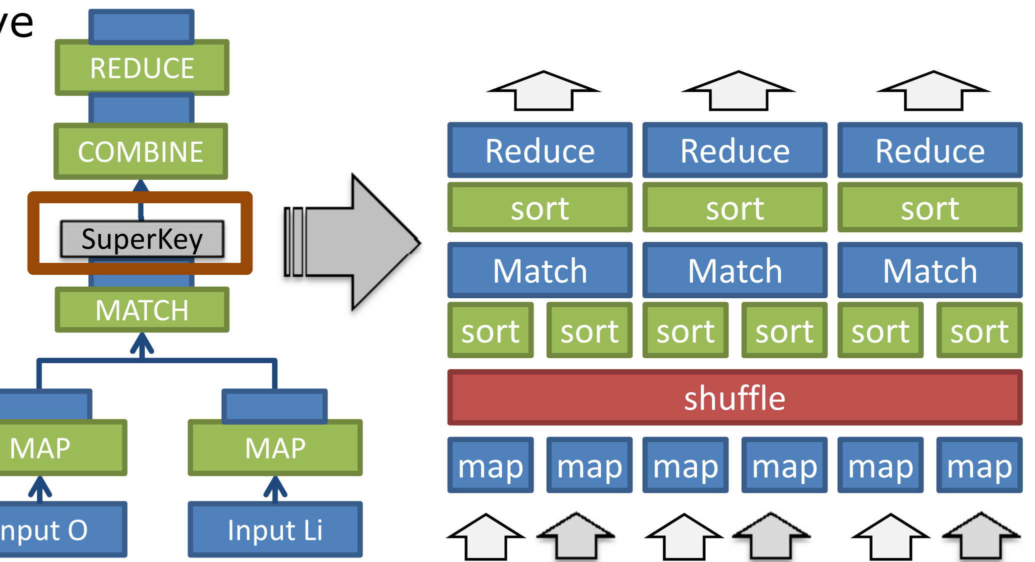


- Parallelizing *Cross* has choices
  - Broadcast one side (asymm.-frag.-replic.)
  - Symmetric-Fragment-Replicate
  - Rings



Optimization Across PACTs

- A PACT's required partition and sort properties can frequently be inferred to be present
  - For example when already established by the parallelization of a preceding PACT
- Global optimization makes different choices than local optimization
  - A locally more expensive choice can establish a partitioning that can be reused
  - Leads to optimization with interesting properties like in DBMS
- Users annotate properties with output contracts

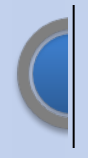


Output Contracts *(examples)*

- **Same-Key**
  - User Function does not alter the key
  - For Multi-Input PACTs specify whose input-key remains

- **Super-Key**
  - Key generated by UF is a super-key of the input key
  - For Multi-Input PACTs specify from which input the key is a super-key

- **Unique-Key**
  - UF produces unique keys



Optimization Algorithm

- Simple bottom up optimizer with top down interesting properties (similar to DBMS)
  - Properties are partitioning and sort order inside partitions
- Top down: Operators describe which properties they benefit from
- Bottom up: Subplan describes which properties it has
  - If a property is interesting, plan is not pruned, even if it is more expensive