ReStore: Reusing Results of MapReduce Jobs

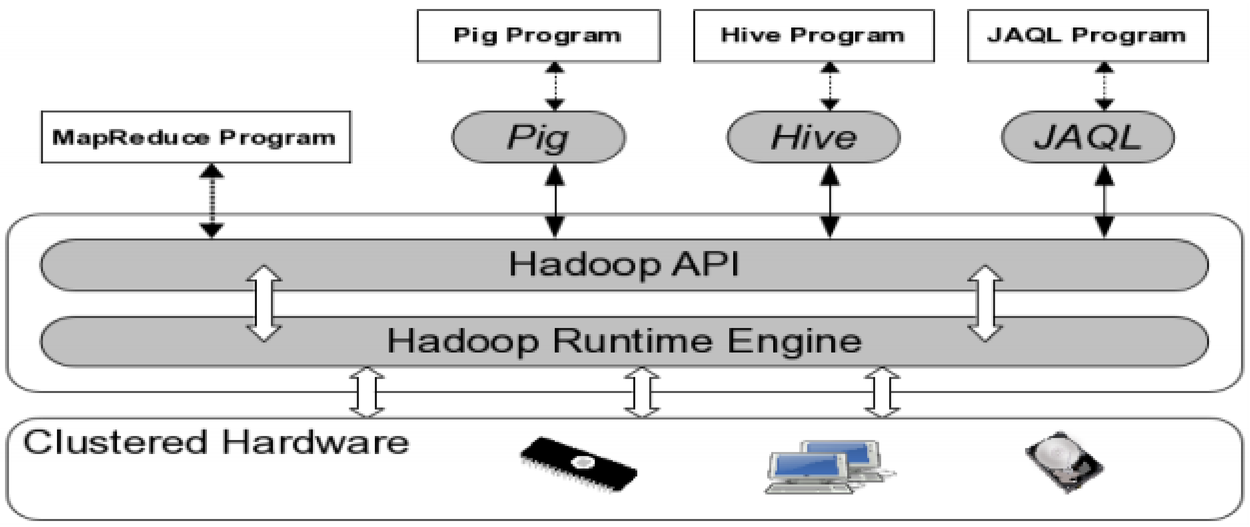
Presented by: Ahmed Elbagoury

Outline

- Background & Motivation
- What is Restore?
- Types of Result Reuse
- System Architecture
- Experiments
- Conclusion
- Discussion

Background

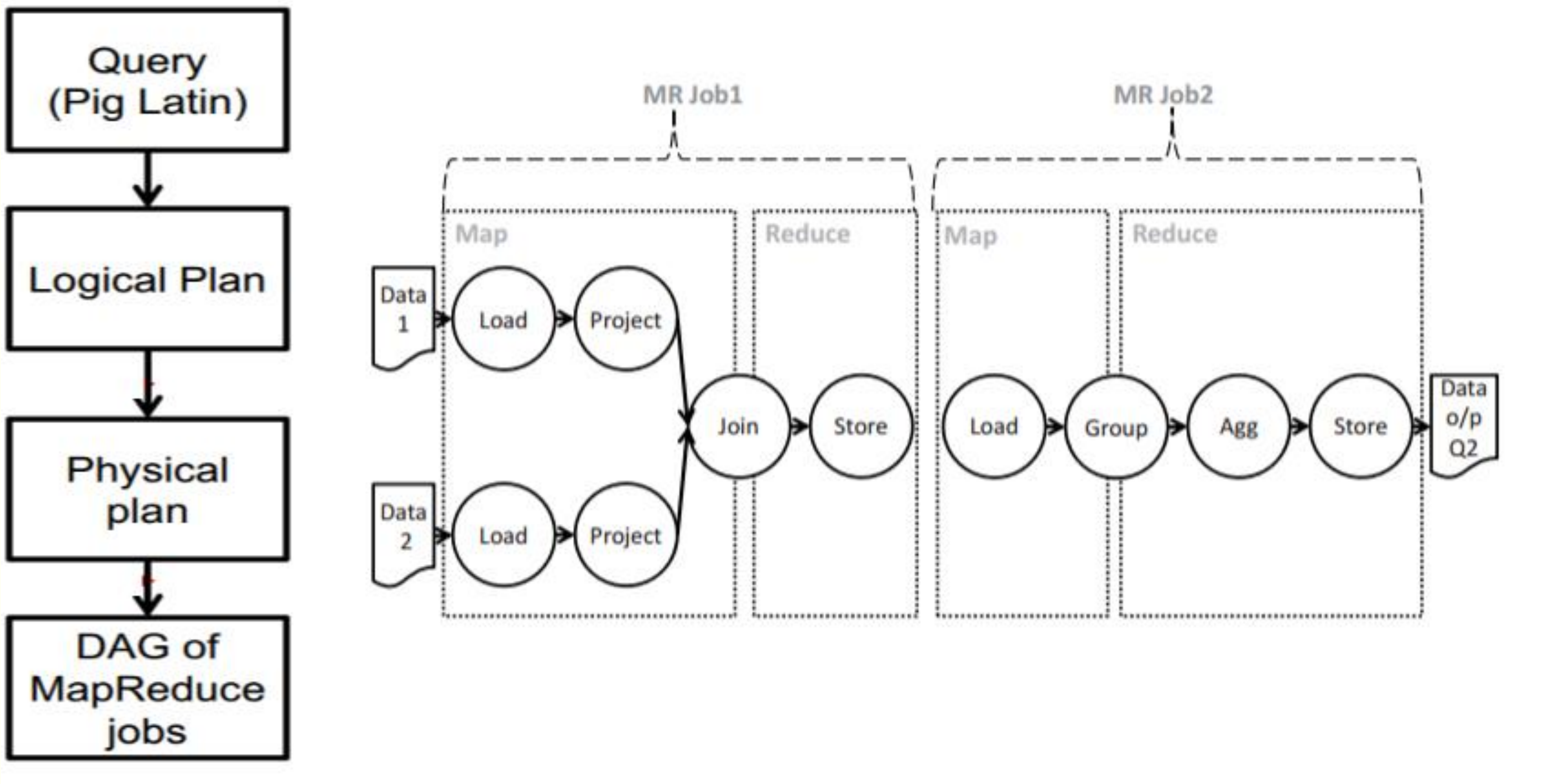
- MapReduce facilitates large scale data analysis
- Users have complex tasks to express as one MapReduce job
- Express complex tasks using high level query languages such as Pig, Hive or Jaql



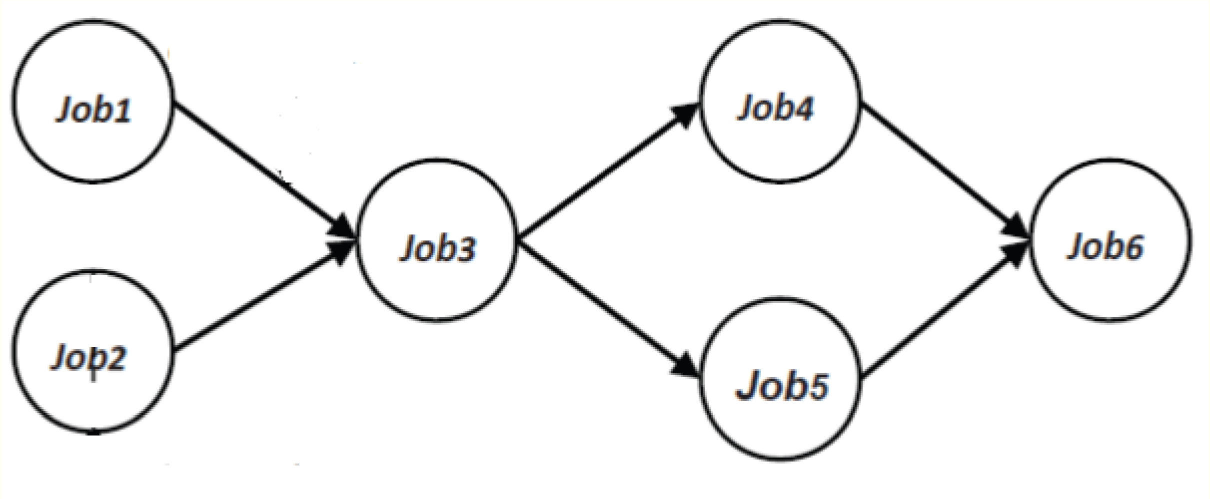
The figure is from "Comparing High Level MapReduce Query Languages" paper

Background

- The compilers of these high-level query languages translate queries into workflows of MapReduce jobs



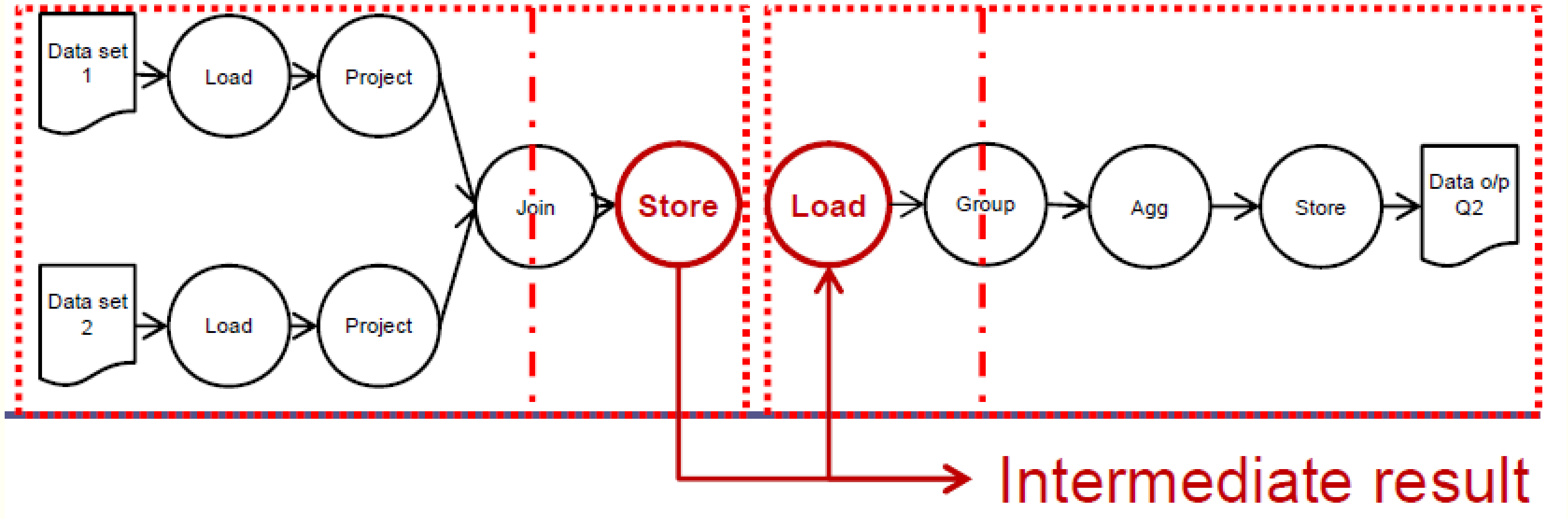
Workflow of MapReduce Jobs



- Each job produces output that is stored in the distributed file system (DFS) used by the MapReduce system
- These intermediate results are used as inputs by subsequent jobs in the workflow
- These intermediate jobs are deleted from the DFS after finishing the workflow

Reusing Intermediate Results

- Saving the intermediate results so that future jobs can use them
- Similar to the materialized views in RDBMS



The figure is from I.Elghandour's Presentation

Outline

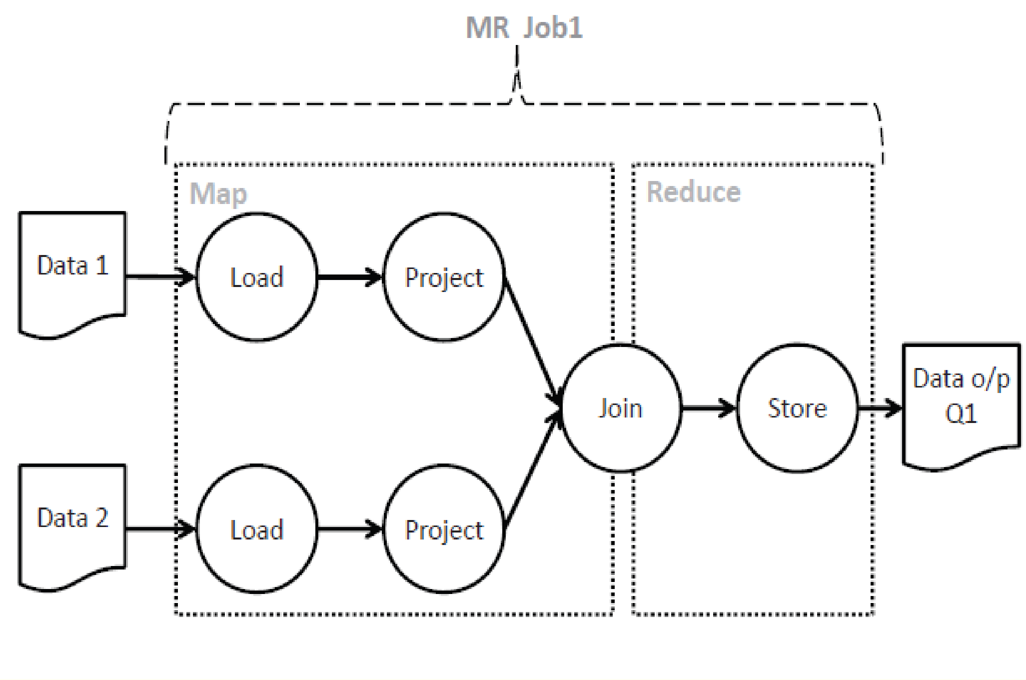
- Background & Motivation
- What is Restore?
- Types of Result Reuse
- System Architecture
- Experiments
- Conclusion
- Discussion

Restore

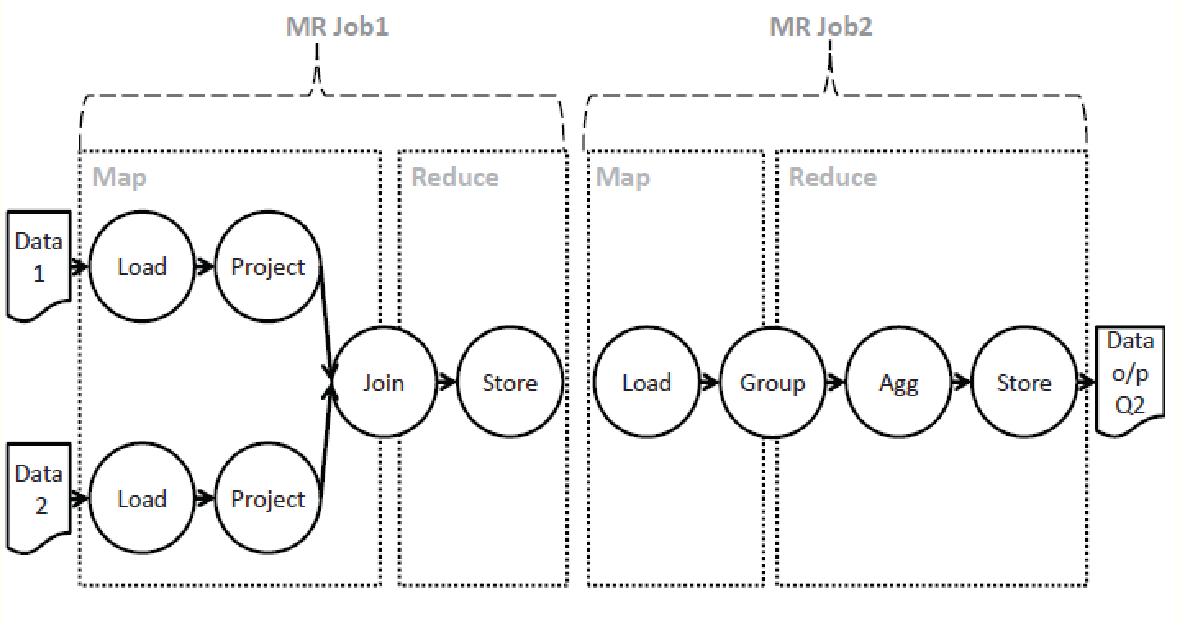
- Restore improves the performance of workflows of MapReduce jobs by
  - Storing the intermediate results of executed workflows and
  - Reusing them in future workflows
- The system is not limited to
  - The queries that are executed concurrently
  - Sharing one operator between multiple queries
- Is sharing results important?
  - Facebook stores the result of any query in its MapReduce cluster for seven days for sharing purposes

Example of Two Queries

Return the estimated revenue for each user viewing web pages

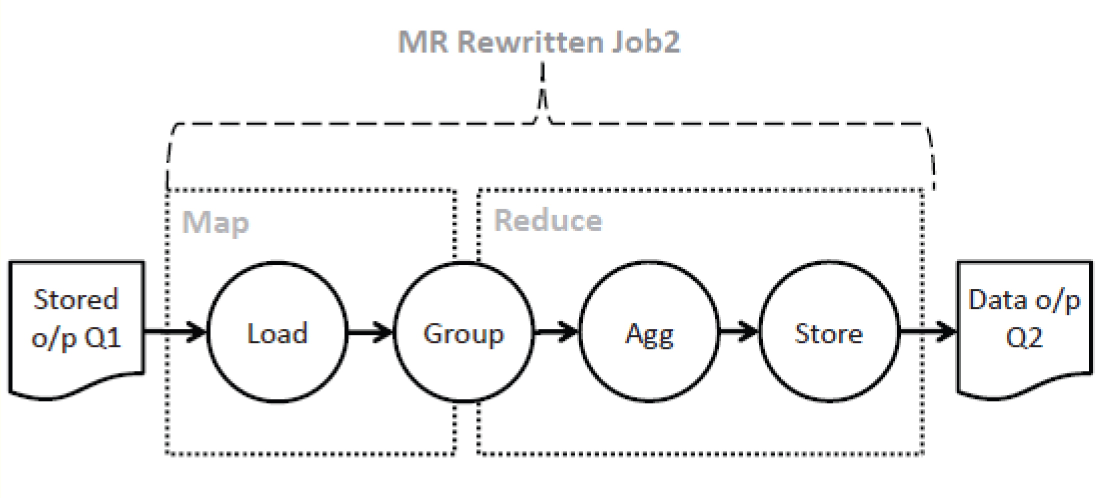


Return the total estimated revenue for each user viewing web pages, grouped by user name



Example of Two Queries

- The MapReduce workflow for query Q2 after rewriting it to reuse the output of query Q1



Outline

- Background & Motivation
- What is Restore
- Types of Result Reuse
- System Architecture
- Experiments
- Conclusion
- Discussion

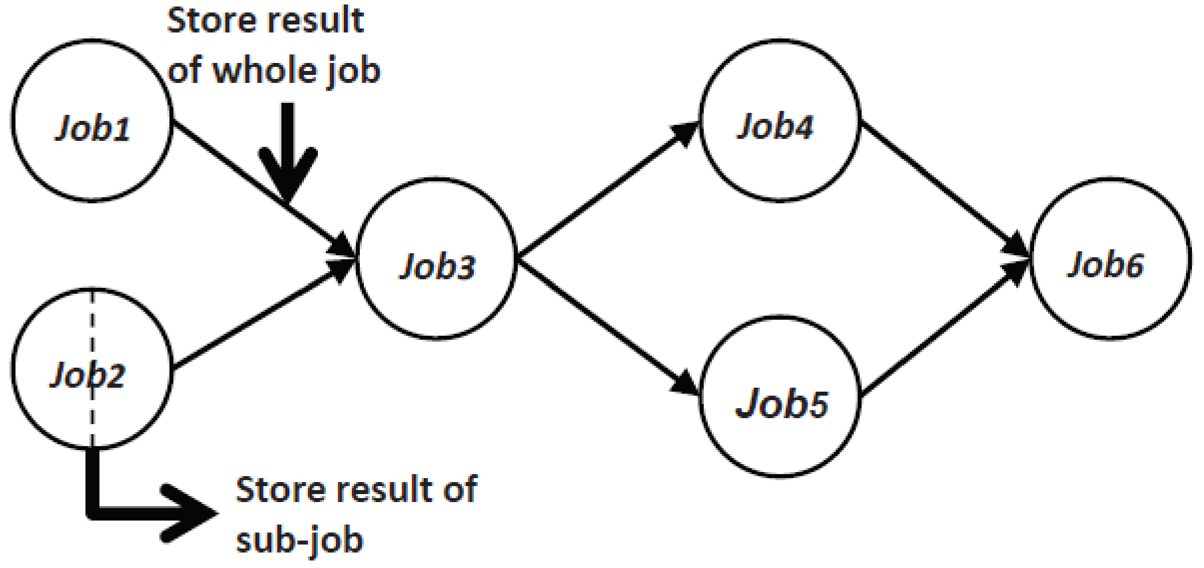
Types of Result Reuse

Time to execute Job_n is

$$T_{total}(Job_n) = ET(Job_n) + \max_{i \in Y} \{T_{total}(Job_i)\}$$

Two types of reuse opportunities

1. Whole job: Reduces $\max_{i \in Y} \{T_{total}(Job_i)\}$
2. Operators in jobs (sub jobs): Reduces $ET(Job_n)$ in future jobs



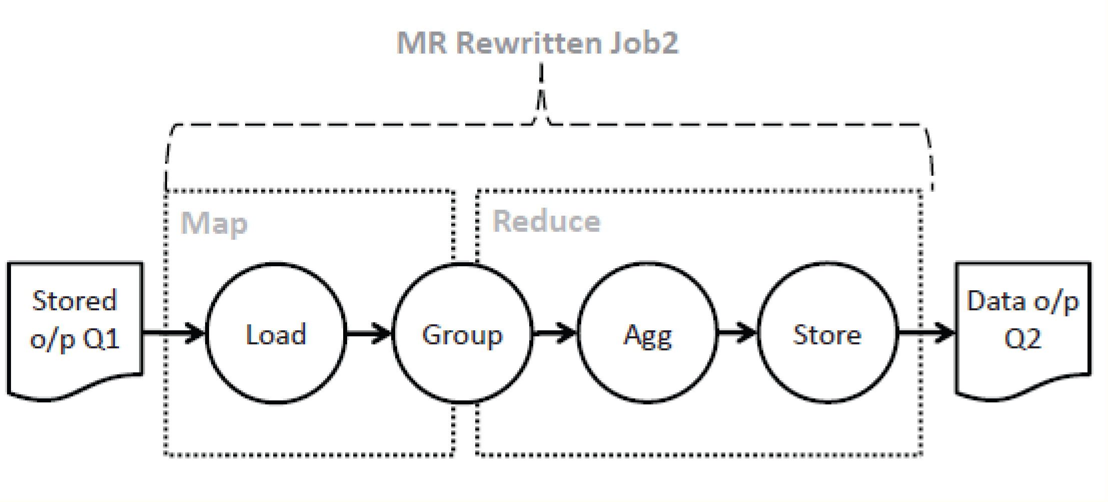
Reusing the Whole Job

$$T_{total}(Job_n) = ET(Job_n) + \max_{i \in Y} \{T_{total}\,(Job_i)\}$$

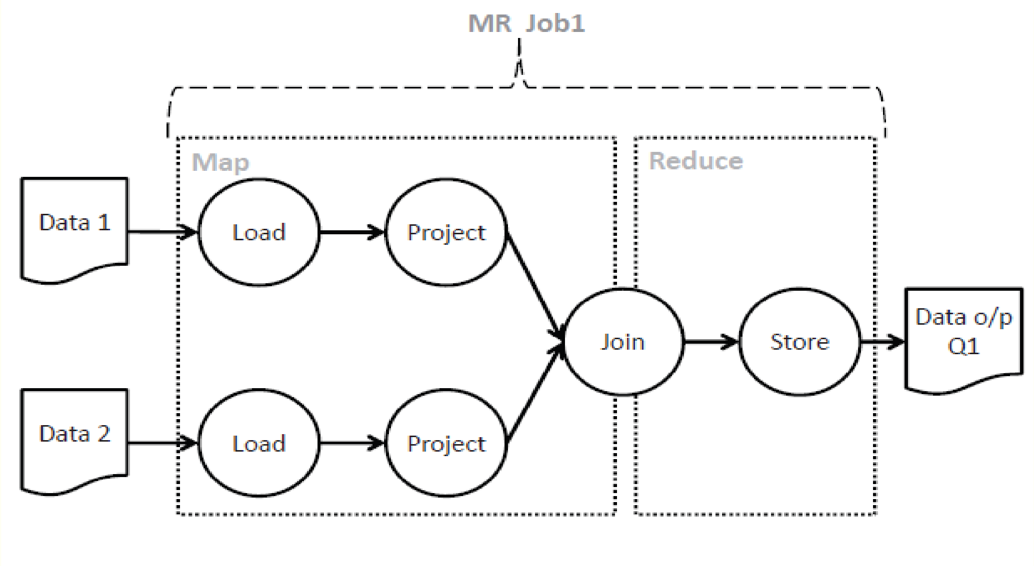
- If the results of all dependent jobs of Job_n are stored in the system, then

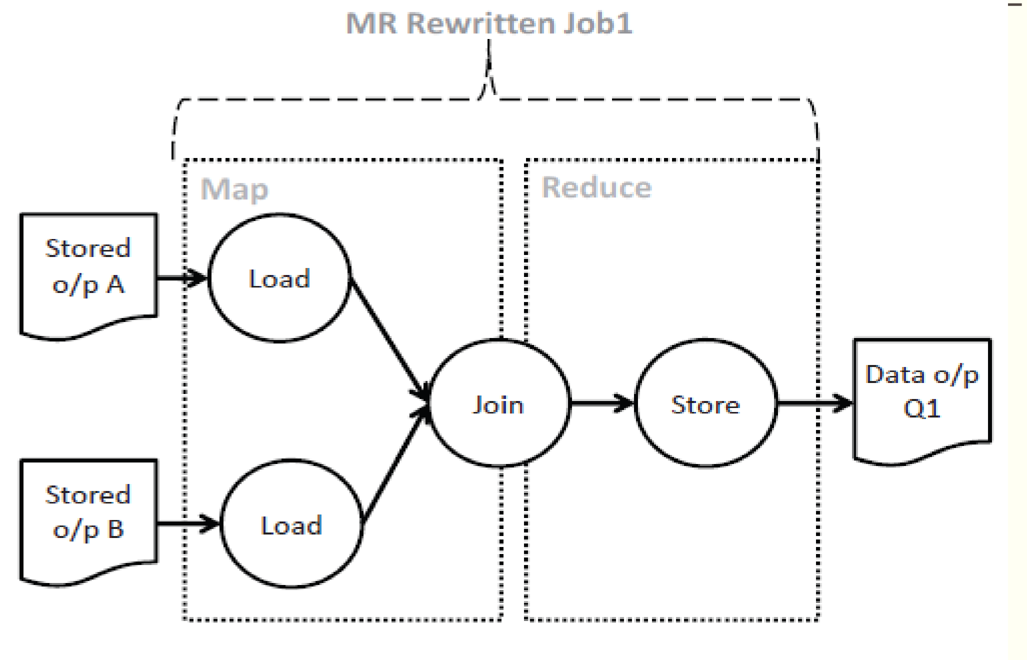
$$T_{total}(Job_n) = ET(Job_n)$$

- If the results of subset $X \subset Y$ of dependent jobs are not stored then
  - $T_{total}(Job_n)$ is reduced only if $\max_{i \in X} \{T_{total}\,(Job_i)\}$ is less than $\max_{i \in Y} \{T_{total}\,(Job_i)\}$



Reusing Sub-job





$$ET(Job_n) = T_{load} + \sum_i ET(OP_i) + T_{sort} + T_{store}$$

We need to answer two questions

1. How to rewrite a MapReduce job using stored intermediate results?
2. How to populate the repository with intermediate results?

Outline

- Background & Motivation
- What is Restore
- Types of Result Reuse
- System Architecture
- Experiments
- Conclusion
- Discussion

System Architecture

Restore has three main components

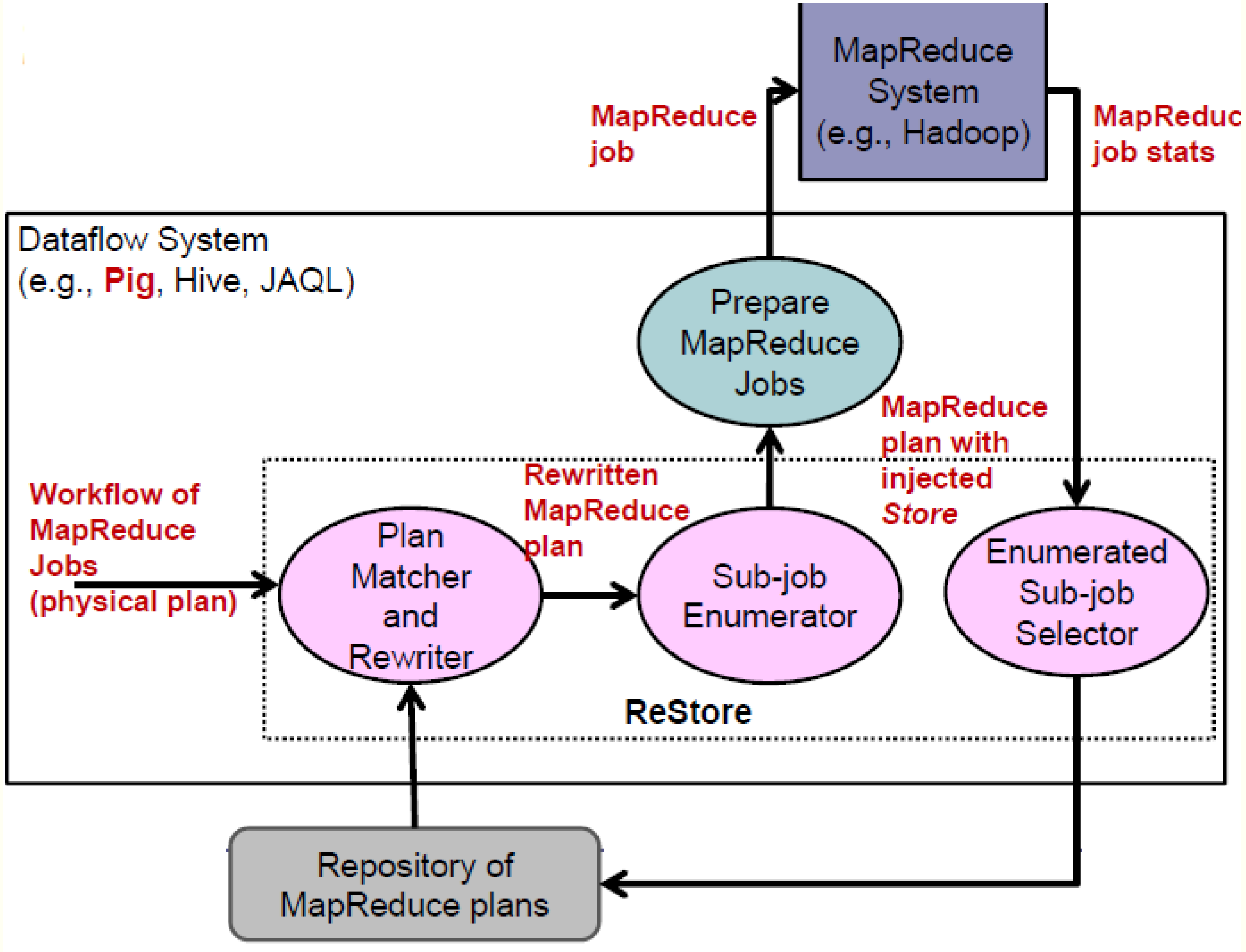
1. Plan Matcher and Rewriter
2. Sub-job Enumerator
3. Enumerated Sub-job Selector

The input is:

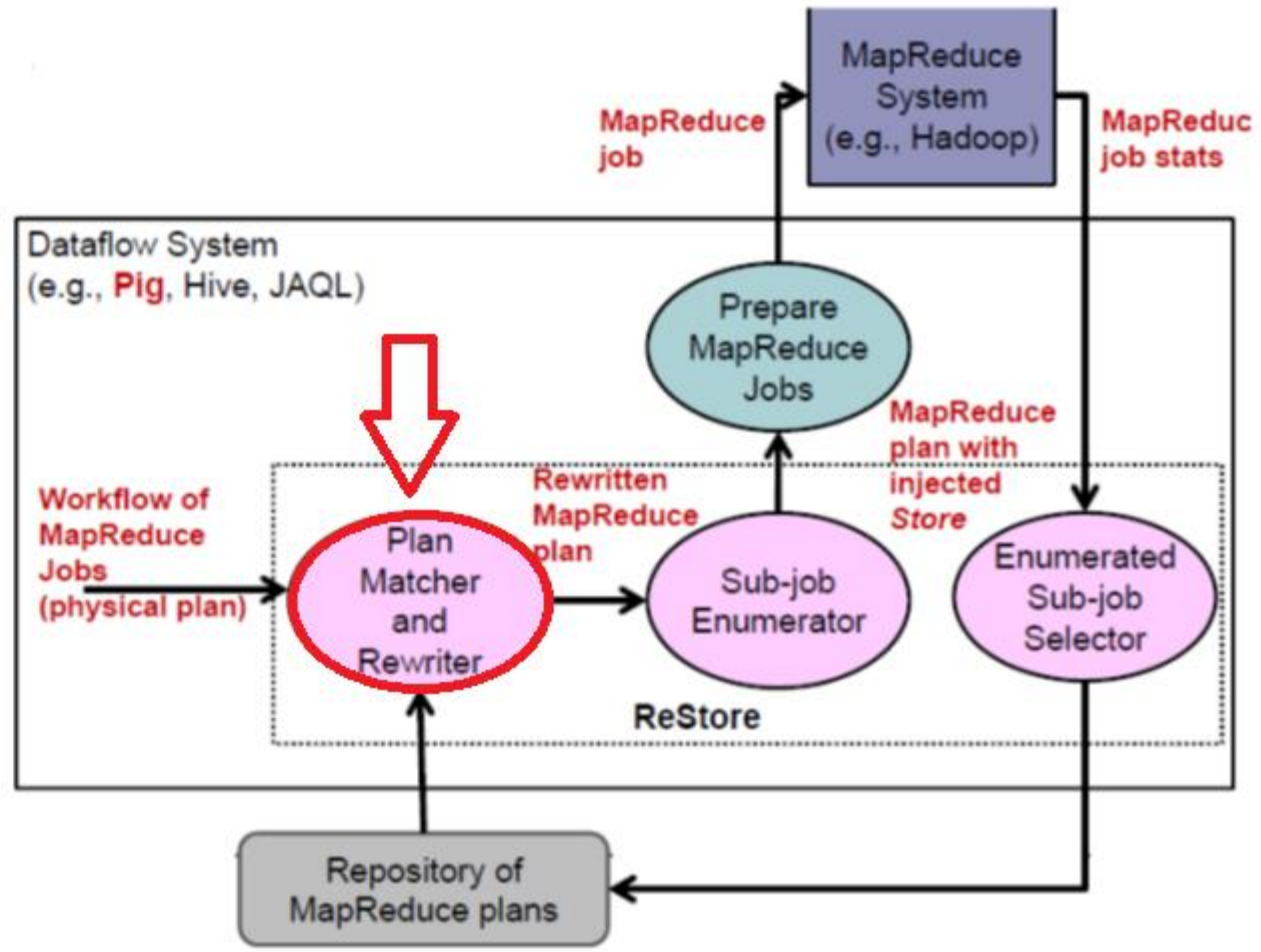
- Workflow of MapReduce jobs

The outputs are

- Modified MapReduce jobs
- Job outputs to store in the DFS



1-Plan Matcher and Rewriter



1-Plan Matcher and Rewriter

- Restore repository contains
  - Outputs of previous MapReduce jobs
  - Physical query execution plans of these jobs
  - Statistics about these MapReduce jobs
- The goal is to find physical plans in the repository that can be used to rewrite the jobs of the input workflow

Matching Algorithm (1)

- Matching and rewriting are performed on the physical plan
  - Matching is simple and robust
  - It is easy to adapt Restore to any dataflow system regardless of the input language
- A physical plan is considered a match if it is contained in the input MapReduce job
- The matching is based on operator equivalence, two operators are equivalent if:
  - If their inputs are pipelined from two equivalent operators or from the same data sets
  - They perform functions that produce the same output data

Matching Algorithm (2)

- Both plans are traversed simultaneously starting from load operators until
  - Mismatching operators are found
  - All the operators of the repository plan have equivalent matches in the input MapReduce plan
- The matched part of the input physical plan is replaced by a load operator that reads the output of the matched plan from the DFS
- More than one plan in the repository can be used to rewrite the input job

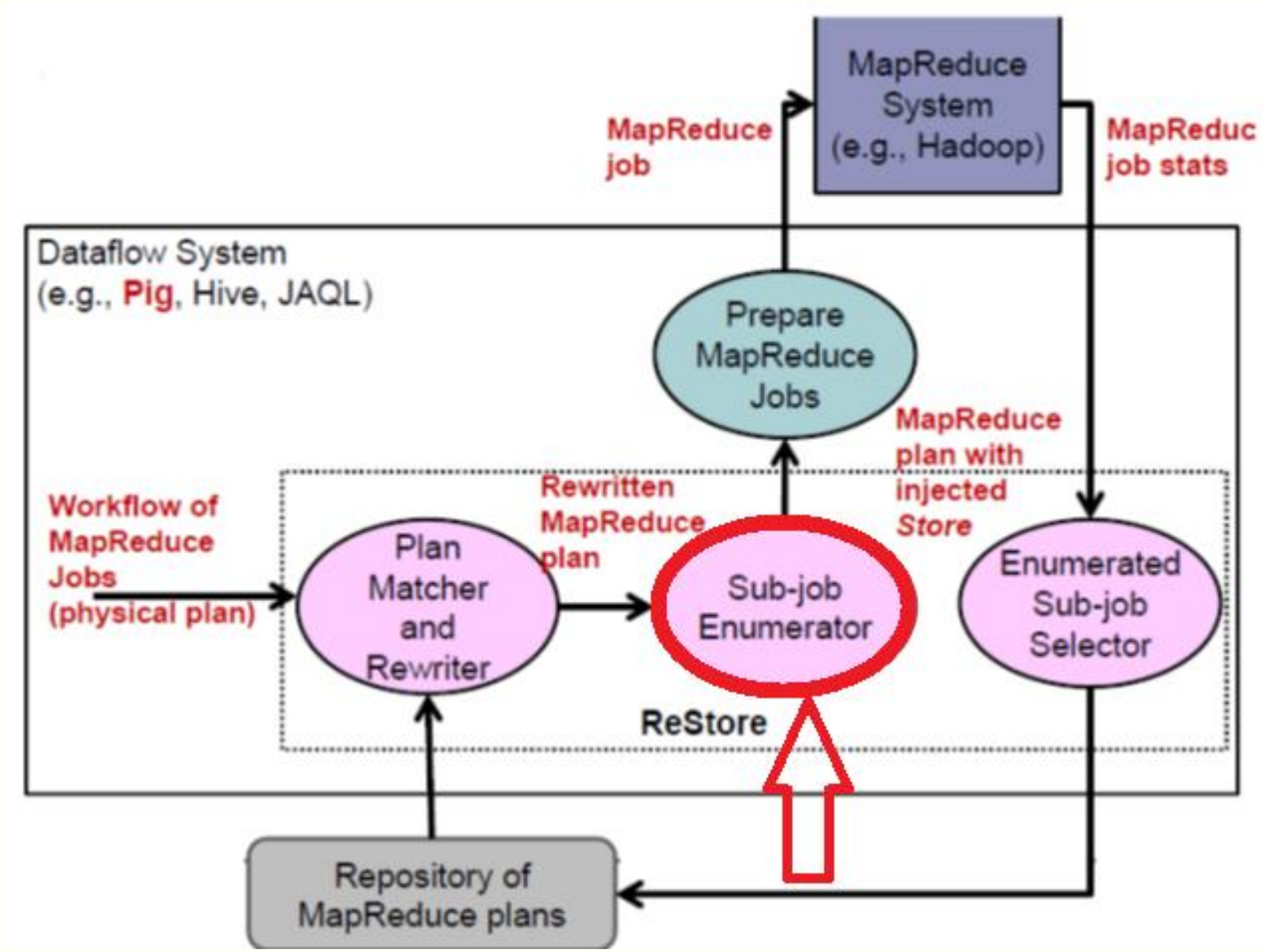
Matching Algorithm (3)

- The first match that Restore finds, is used to rewrite the input MapReduce job
  - The matching becomes more efficient
  - The physical plans in the repository must be ordered
- Ordering physical plans in the repository
- If plan A subsumes plan B (all operators in plan B have equivalent operators in A)
- If neither of A and B subsumes the other:
  - The ratio between the size of the input data and the output data
  - The execution time of the MapReduce job

We need to answer two questions

1. How to rewrite a MapReduce job using stored intermediate results?
2. How to populate the repository with intermediate results?

Generating Candidate Sub-jobs



Generating Candidate Sub-jobs

- The outputs of whole MapReduce jobs and some sub-jobs are saved
- Materializing the outputs of all sub-jobs is infeasible
  - Substantial amount of storage in the DFS
  - It will slow down the execution
- Which sub-jobs should we choose?

Choosing Candidate Sub-jobs

$$ET(Job_n) = T_{load} + \sum_i ET(OP_i) + T_{sort} + T_{store}$$

Good sub-job candidates:

- Operators that reduce the size of their inputs, like: *filter*, *project*
- Expensive operators: *Join* and *Group*

Heuristics for Choosing Candidate Sub-jobs

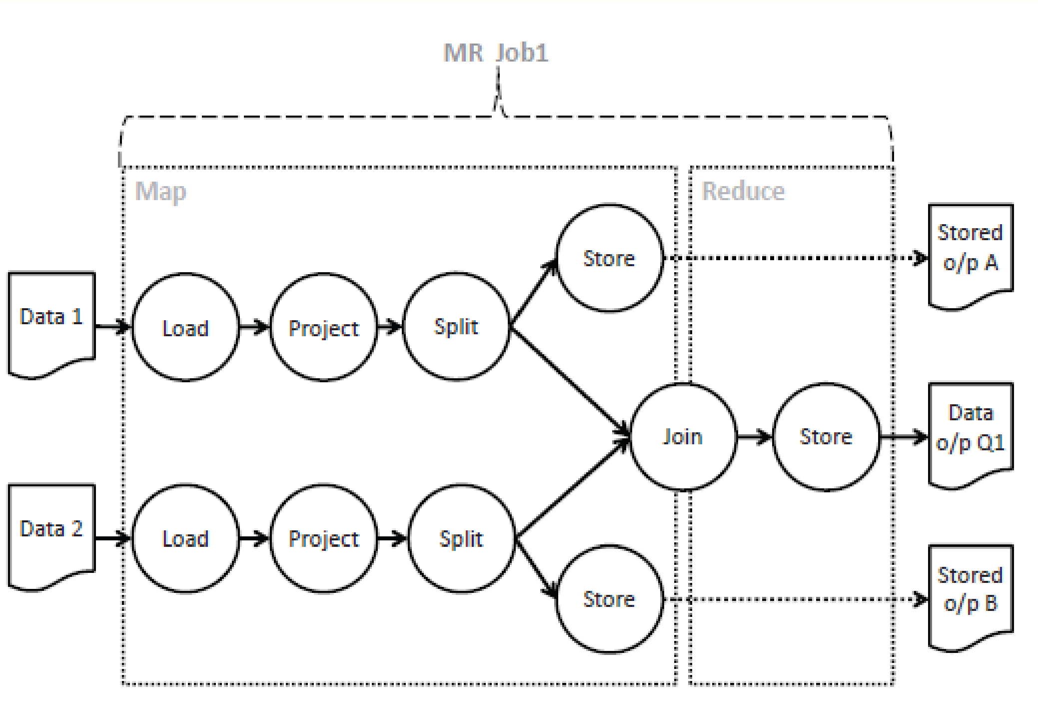
1. Conservative Heuristic
   - Operators that reduce their input size: *project* and *Filter*
2. Aggressive Heuristic
   - Operators that reduce their input size and expensive operators:
     Project, *Filter*, *Join* and *Group*

Conservative heuristic imposes less overhead but creates less reusing opportunities

Choosing Candidate Sub-jobs

For each operator in the physical plan

- Check the used heuristic
- Inject a store operator
  after it (if it is not a store operator)
- Split the flow into two sub-flows

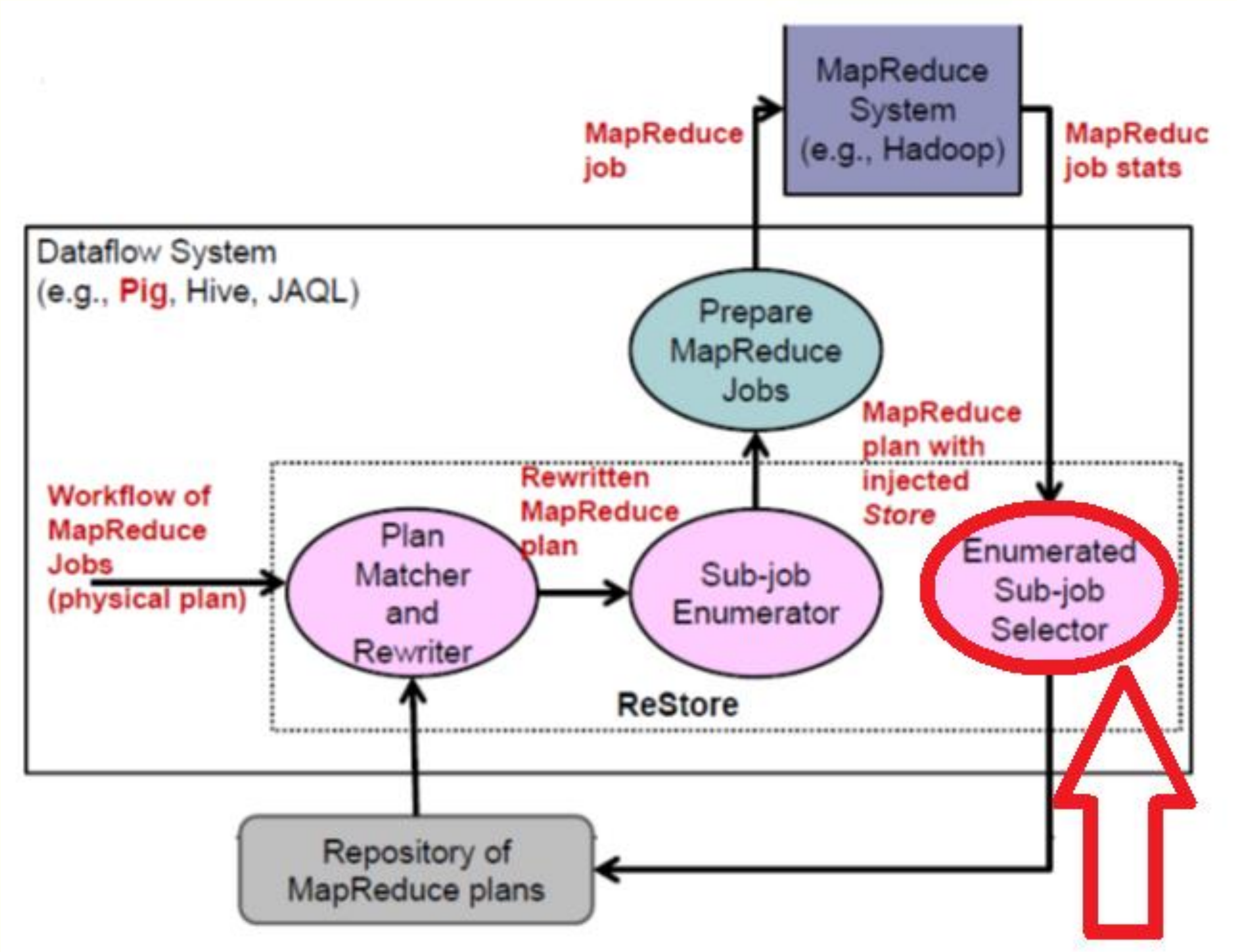


Generated jobs are stored in order the makes the matching efficient

Enumerated Sub-job Selector

- Keeping all generated jobs and sub-jobs is expensive
  - Storage space
  - Too many plans to match in the future workflows
- Decide which outputs to keep
  - The decision is made after executing the workflow
  - Based on the collected statistics

Enumerated Sub-job Selector



Enumerated Sub-job Selector

Keep the output of a job if

- It reduces the execution time when it is used:
  - The size of it is output is less than the size of it is input
  - It reduces the execution time of workflows that use it

$$T_{total}(Job_n) = ET(Job_n) + \max_{i \in Y} \{T_{total}(Job_i)\}$$

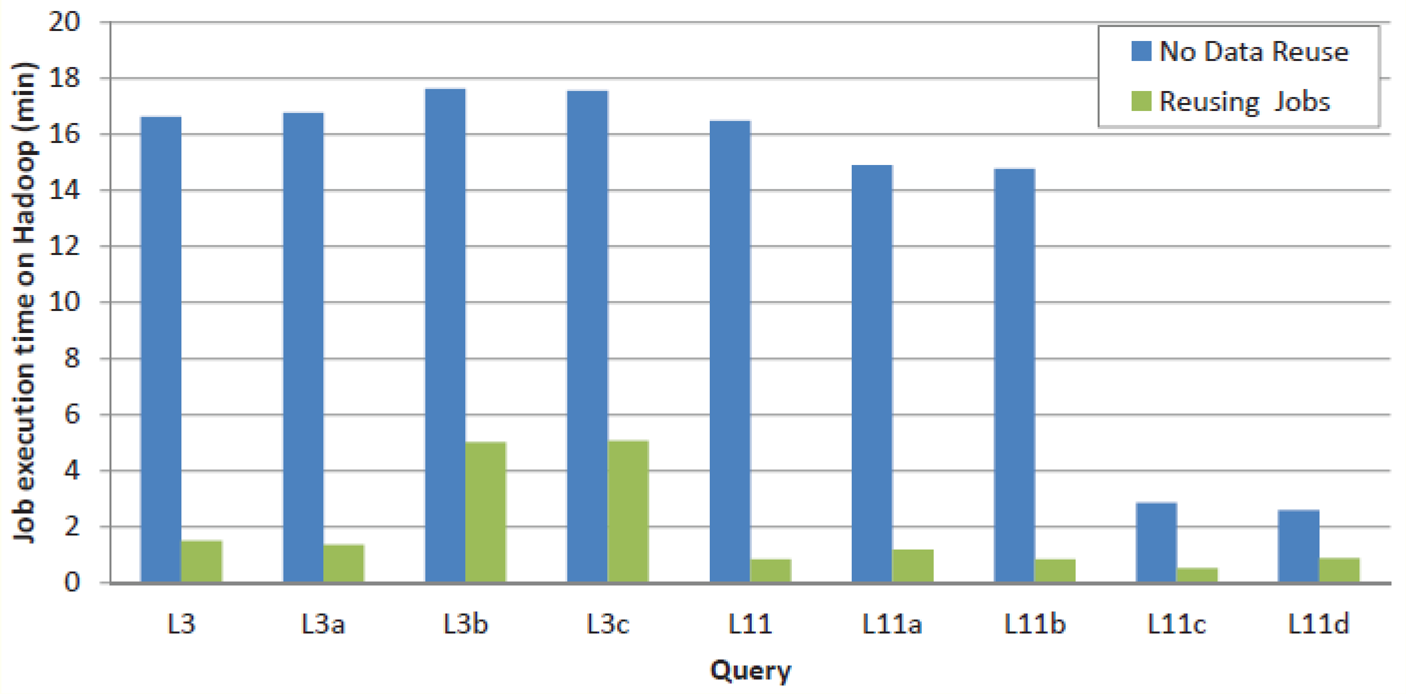
- It is actually used
  - Frequency of usage within time window
  - Deletion or modification of its input

Outline

Experiments

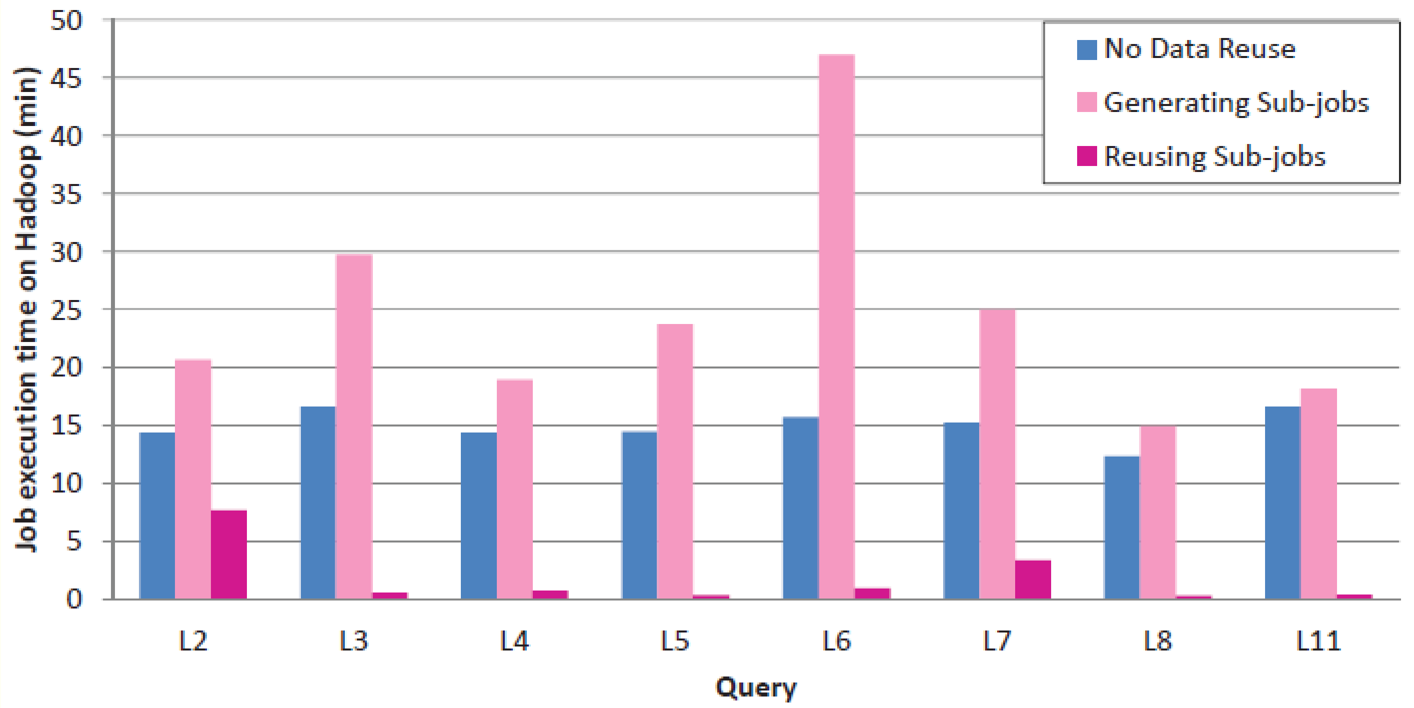
- ReStore implemented as an extension to Pig 0.8
- Experiments run on a cluster of 15 nodes, each with four Dual Core AMD Opteron CPUs, 8GB of memory, and a 65GB SCSI disk
- PigMix benchmark, 150GB data size and 15 GB data size
- Synthetic data with 40GB data size (200 million rows)

Reusing the Output of Whole Jobs



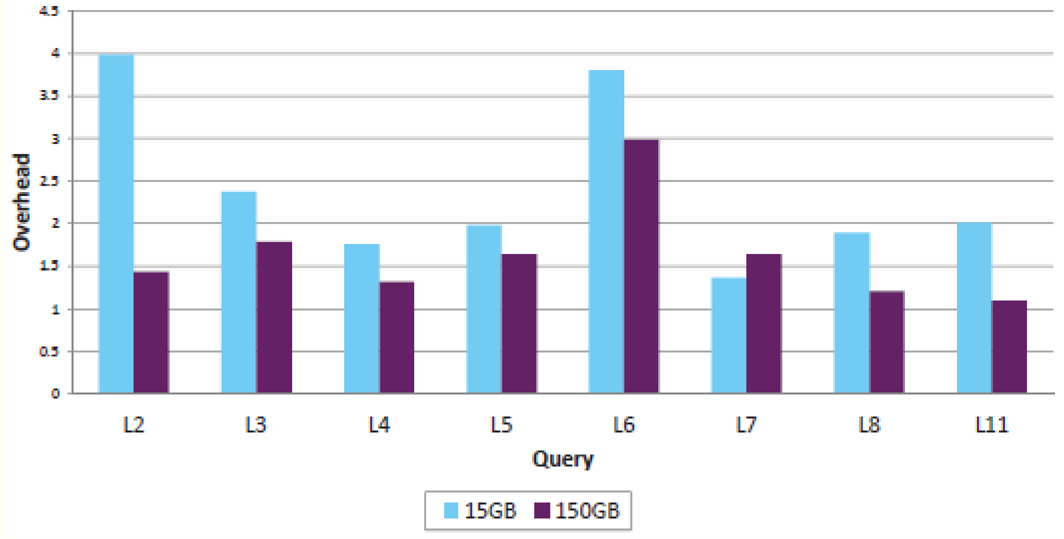
- Assuming all outputs needed for reusing are available
  - Best results the can be achieved
- Speed up is 9.2 with 0% overhead (no extra store operators are inserted)

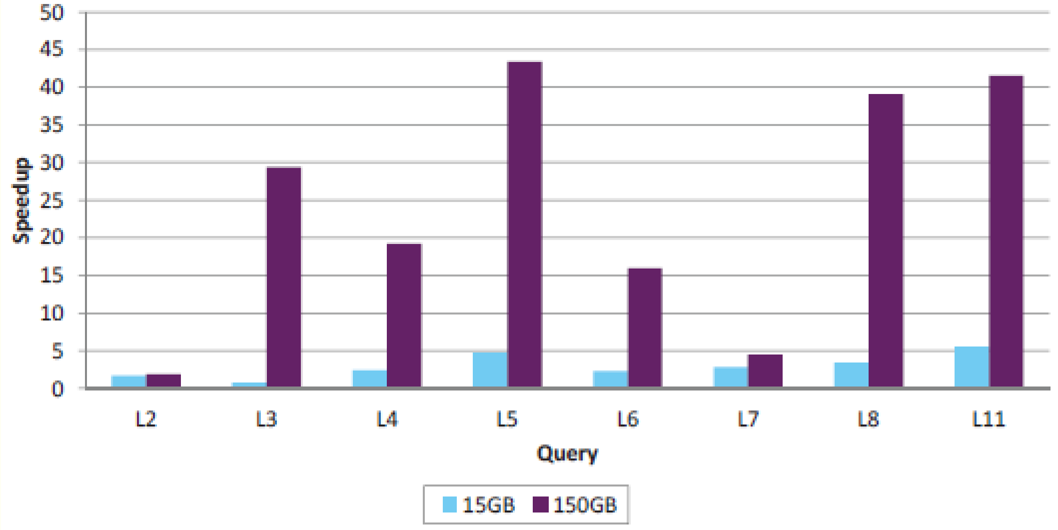
Reusing the Output of sub-Jobs



- Average speedup is 24.4
- Overhead for injecting stores, on average 1.6

Reusing the Output of sub-Jobs



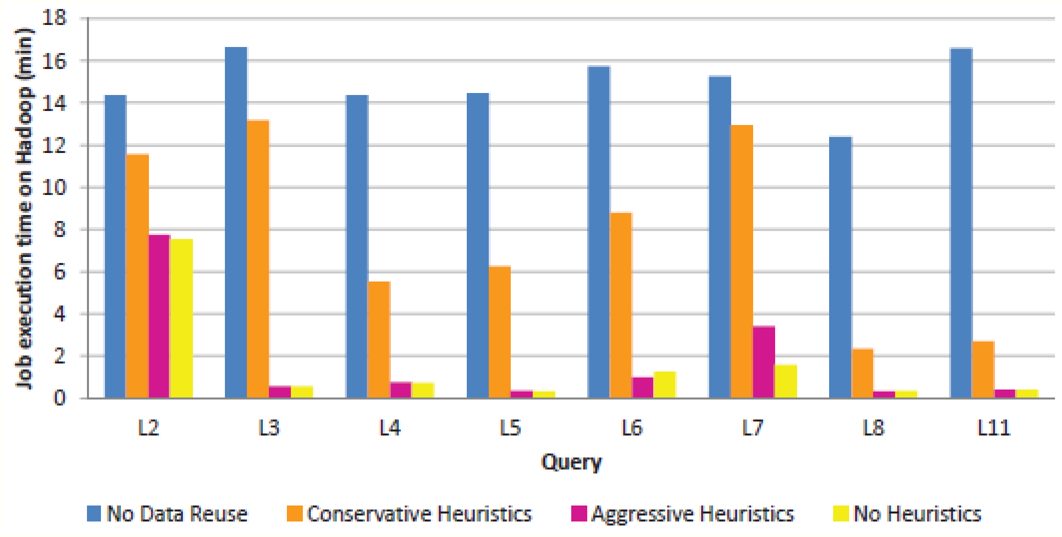


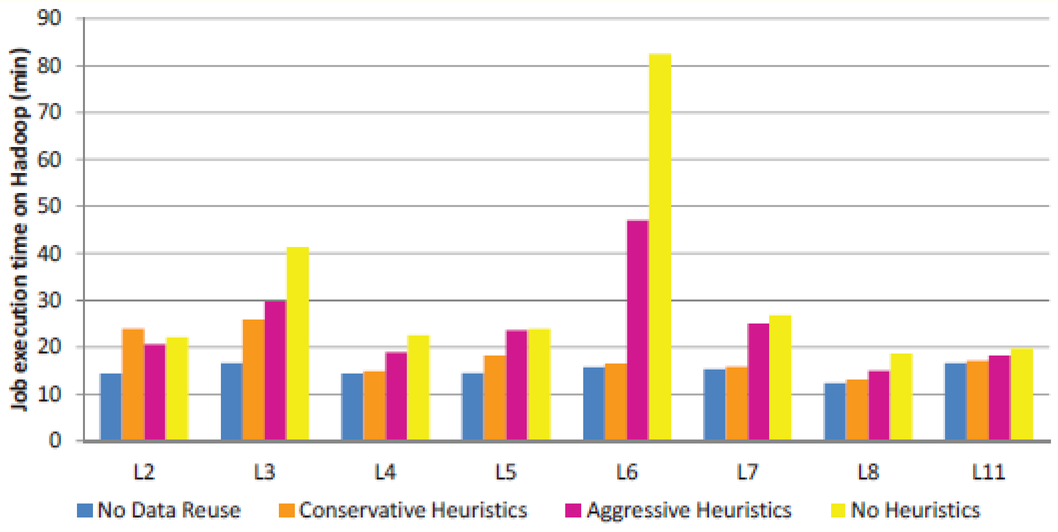
Overhead of 15GB is 2.5
Overhead of 150GB is 1.6

Speedup of 15GB is 3.0
Speedup of 150GB is 24.4

Reusing outputs of sub-jobs is more beneficial for larger data sizes

Comparing the Heuristics

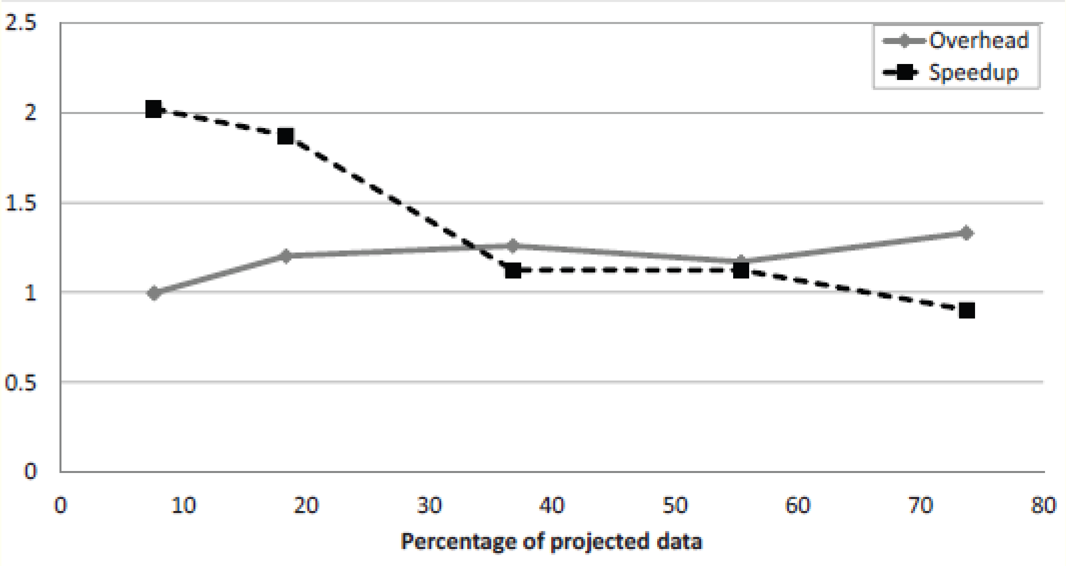


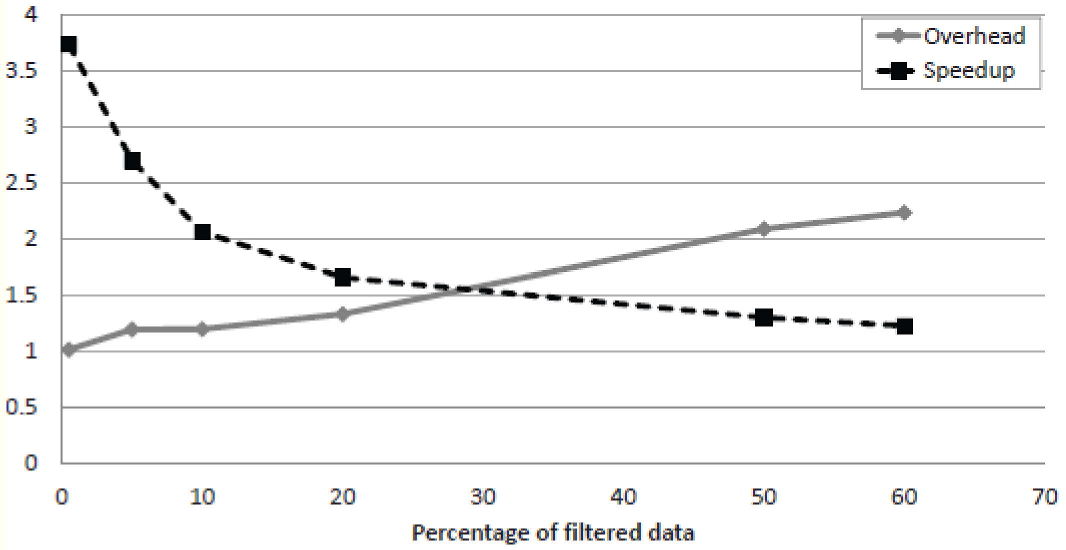


Execution time when using sub-jobs

Execution time with insert operator

Effect of Data Reduction





As the amount of data reduction due to projection or filtering decreases, the overhead increases and the speedup decreases.

Outline

- Background & Motivation
- What is Restore
- Types of Result Reuse
- System Architecture
- Experiments
- Conclusion
- Discussion

Conclusion

- ReStore: a system that reuses intermediate outputs of MapReduce jobs in a workflow to speed up future workflows
- Creates additional reuse opportunities by storing the results of sub-jobs
  - Aggressive vs. conservative heuristic
- Implemented as part of the Pig system
- Significant speedups on the PigMix benchmark

Questions?

Discussion

- Will we need load balancing after injecting store operators?

Discussion

- Will we need load balancing after injecting store operators?
- Sharing between concurrent workflows and keep the output in memory?

Discussion

- Will we need load balancing after injecting store operators?
- Sharing between concurrent workflows and keep the output in memory?
- Can we make better decisions if we know the workload?

Discussion

- Will we need load balancing after injecting store operators?
- Sharing between concurrent workflows and keep the output in memory?
- Can we make better decisions if we know the workload?
- How this can be integrated with pay-as-you-go paradigm?
  - Virtual infinite storage
  - Storage cost
  - Computing cost
  - Can’t make decision after storing
    - Predict running time and storage need
  - Using two level storage and
    - Instead of removing a record it will be moved to the second level storage

Thanks

References

- Elghandour, Iman, and Ashraf Aboulnaga. "ReStore: reusing results of MapReduce jobs." *Proceedings of the VLDB Endowment* 5.6 (2012): 586-597
- Gates, Alan F., et al. "Building a high-level dataflow system on top of Map-Reduce: the Pig experience." *Proceedings of the VLDB Endowment* 2.2 (2009): 1414-1425
- Nguyen, Thi-Van-Anh, et al. "Cost models for view materialization in the cloud."*Proceedings of the 2012 Joint EDBT/ICDT Workshops*. ACM, 2012
- Stewart, Robert J., Phil W. Trinder, and Hans-Wolfgang Loidl. "Comparing high level mapreduce query languages." *Advanced Parallel Processing Technologies*. Springer Berlin Heidelberg, 2011. 58-72.