Reusing Results in Big Data Frameworks

HUI SHANG

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:186

KTH Information and
Communication Technology

KTH Royal Institute of Technology

Dept. of Software and Computer Systems

Degree project in Distributed Systems

Reusing Results in Big Data Frameworks

Author:	Hui Shang
Supervisors:	Vasiliki Kalavri
Examiner:	Associate Prof. Vladimir Vlassov, KTH, Sweden

Abstract

Big Data analysis has been a very hot and active research during the past few years. It is getting hard to efficiently execute data analysis task with traditional data warehouse solutions. Parallel processing platforms and parallel dataflow systems running on top of them are increasingly popular. They have greatly improved the throughput of data analysis tasks. The trade-off is the consumption of more computation resources. Tens or hundreds of nodes run together to execute one task. However, it might still take hours or even days to complete a task. It is very important to improve resource utilization and computation efficiency. According to research conducted by Microsoft, there exists around 30% of common sub-computations in usual workloads. Computation redundancy is a waste of time and resources.
Apache Pig is a parallel dataflow system runs on top of Apache Hadoop, which is a parallel processing platform. Pig/Hadoop is one of the most popular combinations used to do large scale data processing. This thesis project proposed a framework which materializes and reuses previous computation results to avoid computation redundancy on top of Pig/Hadoop. The idea came from the materialized view technique in Relational Databases. Computation outputs were selected and stored in the Hadoop File System due to their large size. The execution statistics of the outputs were stored in MySQL Cluster. The framework used a plan matcher and rewriter component to find the maximally shared common-computation with the query from MySQL Cluster, and rewrite the query with the materialized outputs. The framework was evaluated with the TPC-H Benchmark. The results showed that execution time had been significantly reduced by avoiding redundant computation. By reusing sub-computations, the query execution time was reduced by 65% on average; while it only took around 30 ˜ 45 seconds when reuse whole computations. Besides, the results showed that the overhead is only around 25% on average.

Referat

Big Data analys har vart ett attraktivt samt aktivt forskningsområde under de senaste åren. Det börjar bli svårt att effektivt exekvera dataanalytiska arbetsuppgifter med traditionella lösningar via datalager. Parallella process plattformar samt parallella dataflödes system blir allt mer populära för att bryta trenden. De har tillsammans ökat genomströmningen av analytiska uppgifter drastiskt. Detta sker dock på en kostnad av konsumtion av beräkningsresurser. Fastän allt mellan 10 och 100 noder arbetar tillsammans för att exekvera en uppgift, så kan det fortfarande ta dagar för att slutföra denna uppgift. Det är av stor vikt att förbättra denna resursförbrukning samt att öka beräkningseffektiviteten. Enligt uppgifter från Microsoft så finns det kring 30% gemensamma subkomponenter i en typisk arbetsuppgift. Detta kan leda till slöseri av både tid och resurser.

Apache Pig är ett parallellt dataflödes system som körs på Apache Hadoop, som är en parallell prossecerings plattform. Pig och Hadoop är tillsammans ett av de populäraste kombinationer var det gäller storskalig data processering. Detta examensarbete föreslår en ram som tar fram och återanvänder uträkningar för att förhindra redundans med Pig/Hadoop. Denna idé kom från materialized view technique i Relaterade Databaser. Beräkningsresultat utvaldes och sparades i Hadoop File System på grund av deras stora storlek. Exekveringsstatistiken av resultaten sparades i sin tur i MySQL Cluster. Ramen använde sig utav en plan matcher och rewriter component för att hitta den minsta gemensamma nämnaren för beräkningarna med query från MySQL Cluster, varav ramen modifierar denna beroende på resultat. Ramen evaluerades med TPC-H Benchmark. Som resultat så visar det sig att exekveringstiden minskade drastiskt genom att undvika onödiga uträkningar. Genom att återanvända gemensamma beräkningar såminskade query-tiden i medel med 65%, varav det enbart tog 30 till 45 sekunder när hela beräkningen återanvändes. Resultaten visade trots allt att overhead tiden i medel enbart var 25%.

Acknowledgment

I am deeply thankful to my supervisor, Vasiliki Kalavri, who gives me great guidance and help throughout this work.
I would also like to give my sincere gratitude to my examiner Vladimir Vlassov, who gives precious feedback and advices.
I also want to express my thanks to my seniors, Ying Liu and Vamis Xhagjika, who give valuable ideas and help me a lot.
I give my special thanks to all my friends from EMDC, especially Vaidas Brundza, who discusses with me the technical ideas and gives a lot of suggestions.
I also want to thank Ahmad Al-Shishtawy, who help me to access the cluster.
Last but not least, I would like to thank my family. They give me immense encouragement and support, with which I could overcome difficulties and persist my work.
Thank you!

Stockholm, 04. July 2013

Hui Shang

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

1.1 Motivation

The amount of information that businesses and organizations can store and analyze is rapidly increasing. There is a growing need for efficient processing of very large volumes of data, larger than ever before, which is frequently referred to as Big Data. It is getting increasingly popular to use parallel processing systems for large scale data processing. There are a lot of such systems, combined with Parallel Dataflow Programs(PDP) running on top of them: MapReduce/Sawzall [18, 38], Dryad/DryadLinQ [28, 44], Hadoop/Pig [2, 21] and Cosmos/SCOPE [17]. MapReduce is the first parallel programming framework and it was introduced by Google, which processes a huge amount of data every day. Due to the explosion of data, traditional data warehouse solutions could not complete the data analysis task efficiently and economically, and this has driven the emergence of MapReduce. Similarly, many other companies such as Yahoo and Microsoft also started the research for parallel processing systems, to deal with large datasets. These systems have greatly improved the throughput of data analysis tasks, comparing to traditional solutions.
However, the parallel processing systems also consume huge amount of resources, as tens or hundreds of nodes are used together to execute one task. To improve computation efficiency and resource utilization, it is necessary to optimize the computation tasks. Optimization is always an important topic in data processing systems. Since the cost for task execution and data storage is very expensive, considerable savings can be obtained by optimization. According to the research of Microsoft [23, 26, 27, 39], there exists around 30% computation redundancies in the workflow. Avoiding computation redundancy can save a lot of time and money.
This thesis project aims to build a framework, which can efficiently detect and avoid redundant computations. Relational databases had faced the same problem in the past and there had been a lot of studies. One of the most important techniques is the materialized view technique, which saves and reuses the result of previous computations. The goal of this thesis project is to study the materialized view technique and and explore the possibilities of applying it to the Big Data environment.

1.2 Contribution

In this thesis, we present a framework that can materialize and reuse previous computation results. We studied the materialized view technique in relational databases and related research in the Big Data area. We identified the difficulties and opportunities of adopting the materialized view technique in Big Data environment. We developed a simple framework that could reuse previous results for computation efficiently. We studied Apache Pig, which is a popular parallel dataflow system, and built our framework on top of it. We ran Pig with our framework on Hadoop, which is a popular parallel processing platform, for evaluation. We used MySQL cluster to store the execution statistics for the materialized results.
We show that the framework can efficiently detect reuse opportunities, and reuse previous results for computation. In general, the overhead of materialization is small, on average around 25% of the execution time without the framework. By reusing materialized results, query execution time has been significantly reduced. By reusing results of sub-computations, query execution time is on average around 35% of the time without the framework; and by reusing results of whole computations, the query execution time is around 30 ˜ 45 seconds, regardless of the query.

1.3 Structure of the Thesis

The rest of the thesis is structured as following: In Chapter 2, we introduce systems related to our work. In Chapter 3, we present the materialized view technique in relational databases. We show materialized view techniques from three aspects: view design, view maintenance and view exploitation. Chapter 4 gives the implementation details of the thesis work, including the framework architecture and how each component works. Chapter 5 is the evaluation of the framework. It discusses the experimental setup and results. In Chapter 6, We present related research in Big Data environment, based on different parallel processing platforms. In Chapter 7 we conclude our work and discuss future directions.

Chapter 2
Background

In this Chapter, we introduce the systems related to our work. They are: Hadoop - the parallel processing system, Pig - the parallel dataflow program, and MySQL cluster - the storage engine.

2.1 Apache Hadoop

Hadoop [2, 29] is a framework which supports running data-intensive applications on a large cluster of commodity nodes. It has two major components: the MapReduce programming model and the Hadoop Distributed File Sytem(HDFS). Hadoop is mainly used as distributed computing platform and storage engine.

2.1.1 MapReduce

Hadoop MapReduce [5] framework has implemented the Google MapReduce model [18]. MapReduce has made it easier to process large amounts of data on thousands of nodes in parallel. As the name indicates, MapReduce includes two phases: map and reduce. Tasks performing a map function are called mappers and tasks performing reduce function are called reducers. The MapReduce execution model is shown in Figure 2.1 [21].

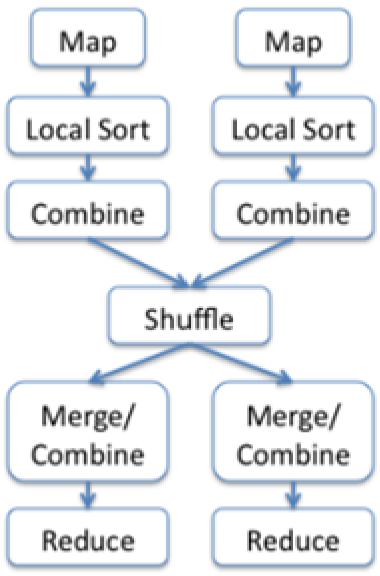


Figure 2.1: MapReduce Execution Stages [21].

The map function takes the input key-value pairs and produces a new set of key-value pairs.
After mapping, the map outputs are sorted. Combiners can be optionally specified after sorting the data. The combiners can aggregate the key-value pairs and reduce the data size to be transferred over network. Combiners often perform similar work to reducers.
After sorting and combining, the data would be partitioned and transferred to each reducer. This phase is called shuffle. While receiving data, the reducer groups the data by keys, this is called sort/merge/combine.
The last step is reduce. The reduce function aggregates the set of values for a key, which would usually result in a smaller set of values.
Hadoop supports writing MapReduce applications in languages other than Java, by using the hadoop streaming utility. However, sometimes it is difficult to express a task as MapReduce jobs, thus high-level query languages such as Pig [21, 37], Hive [42] or Jaql [15] are developed to make MapReduce programming easier.

2.1.2 Hadoop Distributed File System(HDFS)

HDFS [4, 40] is designed to run on large cluster of commodity nodes, and it is highly fault-tolerant. HDFS consists of one Namenode and several Datanodes. It works in master-slave fashion.
The Namenode is the master that controls namespace operations and access to the files. The namespace contains file metadata, such as filename, replication factor,

access permissions, modification and access times. It also contains mapping from file name to the file's physical location on Datanodes. Namespace operations are similar to operations with other file systems, such as open, close, remove and rename.
The Datanodes are slaves and they are used to store data files. The files are split into blocks and each block is replicated according to the replication policy (default replication factor is three). Each block consists of two parts: the data itself and the block's metadata such as checksum and timestamp.
The current latest stable Hadoop version is 1.0.4, and it only has a single Namenode. Thus Namenode is the single point of failure.

2.2 Apache Pig

Pig [1, 20, 21] is a dataflow system running on top of Hadoop. It offers SQL-like language, Pig Latin [37], to express the data flow and transfer the dataflow to map reduce jobs. Besides, it supports user defined functions(UDFs) which can currently be implemented in Java, Python, JavaScript and Ruby [12]. Though it is developed to run on top of Hadoop, it can also be extended to run on top of other systems [30]. Pig allows three execution modes: interactive mode, batch mode and embedded mode. In interactive mode, the user issues commands through an interactive 'Grunt' shell. And only when a 'Store' command is given, Pig would execute all the commands. In batch mode, the user runs a pre-written query script, which typically ends with 'Store'. Embedded mode allows user to embed Pig query in java or other programs. Dynamic construction and control over the commands are enabled under this mode.

2.2.1 System Overview

Pig provides an execution engine to do parallel processing of data on top of Hadoop. It can transform the Pig Script to an execution plan that can be passed to Hadoop for execution, as shown in Figure 2.2. In general there are three steps : (1) Build a logical plan; (2) Translate the logical plan to physical plan; (3) Convert the physical plan to MapReduce plan. Comparing with using MapReduce directly, there are several advantages to use Pig instead [20]. First, Pig Latin is SQL-like language and it contains all standard data-processing operations. While some operations, such as join, are not provided by MapReduce. Second, Pig Script is easier to understand , it defines how to process data step by step. Third, the fact that MapReduce have no data type definitions makes it hard to check coding errors.

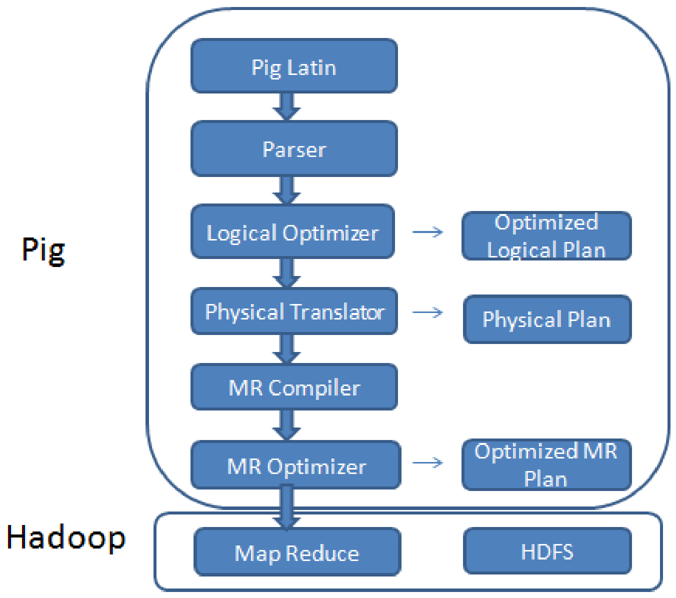


Figure 2.2: Pig Compilation and Execution Stages.

2.2.2 Pig Latin

Pig Latin [10, 20, 37] is the dataflow language provided by Apache Pig. It describes how to read data, process data and store the results.
Pig Latin is designed to be SQL-like so that people familiar with SQL can get started with it easily. It includes some common operations with SQL, as shown in Table 2.1.

SQL	Pig Latin
Select ... From ...	Filter ... By ...
Project	ForEach ... Generate ...
Join	Join
Group By	Group By
Order By	Order By
Union	Union
Select Distinct	Distinct
Cross Join	Cross

Table 2.1: Common operations for Pig Latin and SQL

Despite the similarities, there are many differences between Pig Latin and SQL [20]. First, SQL is used to describe the problems to answer while Pig Latin is used to describe how to answer the problems step by step. Second, each SQL query script can only answer one question while each Pig Latin script allows answer multiple questions. Third, SQL can only process data with pre-defined schema while Pig can process data that is relational, nested or unstructured.

2.2.3 Compilation to MapReduce

This section presents the process of translating a Pig Script to a MapReduce plan. The transformation process is shown in Figure 2.3 [21].

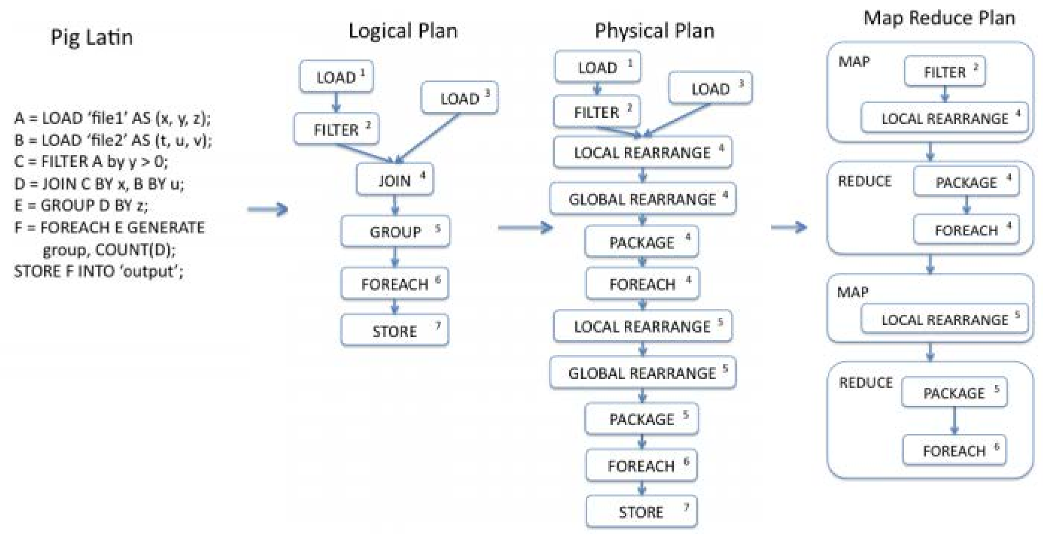


Figure 2.3: Translating Pig Script to MapReduce Plan [21].

The first step is to build a logical plan. The parser firstly performs a series of checks towards the program, including syntactic correctness, data type and schema inference. Then the parser maps the statements with logical operators, and builds a directed acyclic graph(DAG). This is the canonical logical plan corresponding with the Pig program.
The second step is to convert the logical plan to a physical plan. The logical plan built by the parser first goes through the logical optimizer for optimization. The logical optimization details is discussed in Section 2.2.4.1 and we don't show it here. After optimization, each logical operator in the optimized logical plan is mapped to one or more physical operators by the physical translator. As can be seen in Figure 2.2, logical 'Load', 'Filter', 'ForEach' and 'Store' correspond to physical 'Load', 'Filter', 'ForEach' and 'Store' respectively; while logical 'Join' and 'Group' are mapped to multiple physical operators.

The third step is to construct the MapReduce plan from the physical plan. The MapReduce compiler transverses the physical plan, looks for operators(Local Rearrange, Global Rearrange and Package) that indicates a map job or reduce job, and places the physical operators into one or more MapReduce jobs. This intermediate plan is then passed to the MapReduce optimizer and the final MapReduce plan is produced.
After all the above steps, a query is transformed to MapReduce plan consisting of one or more MapReduce jobs. Each MapReduce job would be converted to a Hadoop job and pass to Hadoop for execution.

2.2.4 Pig Optimization

Apache Pig performs optimization at two levels: logical optimization and MapReduce optimization.

2.2.4.1 Logical Optimization

Logical optimization is an extensively used technique. Many batch processing systems such as Pig [21], Hive [42], Drill [25] and Dremel [34] have a logical plan optimizer. Pig's logical optimizer is rule based. Instead of exhaustively searching the optimal logical plan, Pig optimizes the plan by transforming the plan with matched rules. Pig has defined a list of RuleSets. Each RuleSet is a set of rules that can be applied together. Each rule contains a pattern of sub-plan, a matcher to find a matched pattern in the logical plan, as well as a transformer to transform the logical plan. Some rules are defined to be mandatory and they must be applied. For other rules, Pig allows users to disable them and remove the unwanted ones from the RuleSets. The optimizer would run the rules in each RuleSet repeatedly until none of them can find a match or the maximum iterations(default 500) has been reached.
Pig's logical optimizer has a very rich set of optimization rules. Some typical optimization logics in relational databases are also used by Pig, such as early filter and early projection. Except for rules to optimize singe program, Pig also contains rules for multi-query optimization. We show some representative rules here.
FilterAboveForeach : This optimization rule implements the early filter logic. It pushes 'Filter' operators above 'Foreach' operators. The filter operation can eliminate not needed records and reduce data size. Thus, filter should be put in an earlier position.
ColumnMapKeyPrune : This rule implements the early projection logic. It removes the unwanted columns and map keys for each record. The project operation retains a subset of fields for each record and reduces the data size. Therefore, project should be done as early as possible.
ImplicitSplitInserter : This rule is used for multi-query optimization. In a multi-query Pig script, 'Split' is the only operator that can have multiple outputs.

This rule inserts an implicit 'Split' operator after each non-split operator with multiple outputs, and the outputs are made to be outputs of the 'Split' operator. Without the 'Split' operator, the computation for each output would be executed independently; with the 'Split' Operator,the shared computations would be executed for only once.

2.2.4.2 MapReduce Optimization

MapReduce optimization applies to the MapReduce workflows. Currently Pig only performs one optimization at this layer. It checks whether combiners could be added. Using combiners aggressively has two benefits: first, it can reduce the size of data to be shuffled and merged by reducer; second, it tends to give a more uniform distribution of the amount of values associated with a key.
Pig breaks distributive and algebraic aggregation functions(such as AVERAGE) into three steps. Initial: generate key-value pairs; Intermediate: combine multiple key-value pairs into a single pair; Final: combine multiple key-value pairs by applying the function and take the result. The three steps correspond to map, combine and reduce respectively. This enables more chances to use the combiners.
However, there are more optimization possibilities. There is much research on MapReduce workflow optimization. For example, Stubby [33] is a cost-based and transformation-based MapReduce optimizer that can optimize the workflow of MapReduce jobs. It can be integrated with many languages including Pig [21] and Hive [42].

2.3 MySQL Cluster

MySQL Cluster [6] is a distributed in-memory database system. It consists of a cluster of nodes that share nothing with each other. MySQL Cluster technology integrates MySQL server with Network DataBase(NDB), an in-memory clustered storage engine.

2.3.1 System Overview

A MySQL Cluster has three components: NDB management nodes, data nodes and SQL nodes. It has provided APIs to access the MySQL Cluster. The architecture of MySQL cluster is shown in Figure 2.4 [7].
We introduce the three types of nodes briefly.
Management Node: The Management Node manages the configuration of the other nodes. It is also in charge of starting and stopping data nodes, running backup and so forth. To run MySQl cluster, the management node must be started first.
Data Node: The Data Node stores data. MySQL 5.0 and earlier versions store data

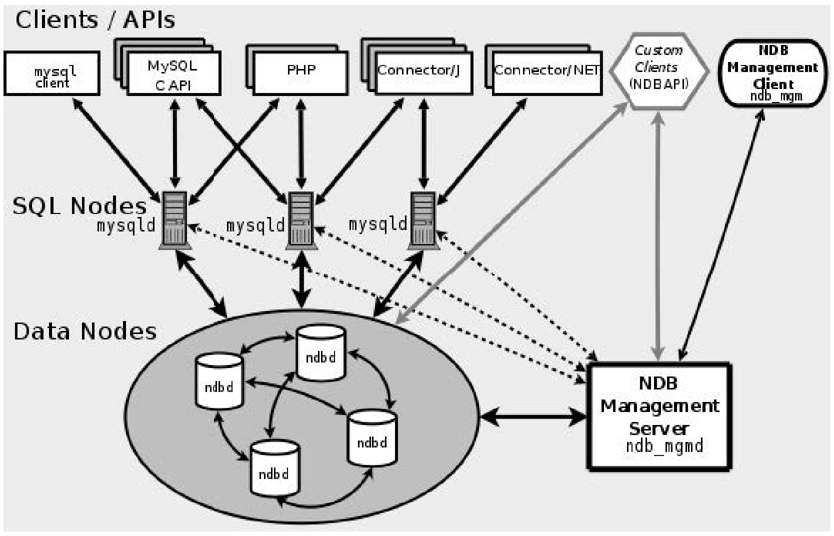


Figure 2.4: MySQL Cluster Architecture [7].

completely in memory. Later versions allows some data to be stored on disk.To provide redundancy and high availability, a MySQL Cluster should have at least two replicas.
SQL Node: The SQL Node is a traditional MySQL server which uses the NDBCluster storage engine. It provides an API to access data in the cluster.

2.3.2 MySQL Cluster Connector for Java

MySQL provides a collection of Java APIs for the java applications to access MySQL cluster [8], as shown in Figure 2.4. There are four main Java Connectors to connect java applications with MySQL Cluster : Java Database Connectivity(JDBC) and mysqld, Java Persistence API(JPA) and JDBC, ClusterJ, and ClusterJPA.
JDBC can send user's SQL statements to MySQL server and return results; JPA needs to use JDBC to connect to MySQL Server; ClusterJ can access NDBCLUSTER directly using JNI bridge which is included in the library 'ndbclient'; ClusterJPA uses either JDBC or ClusterJ to access MySQL Cluster.
Among them, ClusterJ can be used independently and it is not bound to SQL statements. It has some restrictions comparing with ClusterJPA, but its simplicity makes it a good choice as long as it can satisfy the needs of applications.

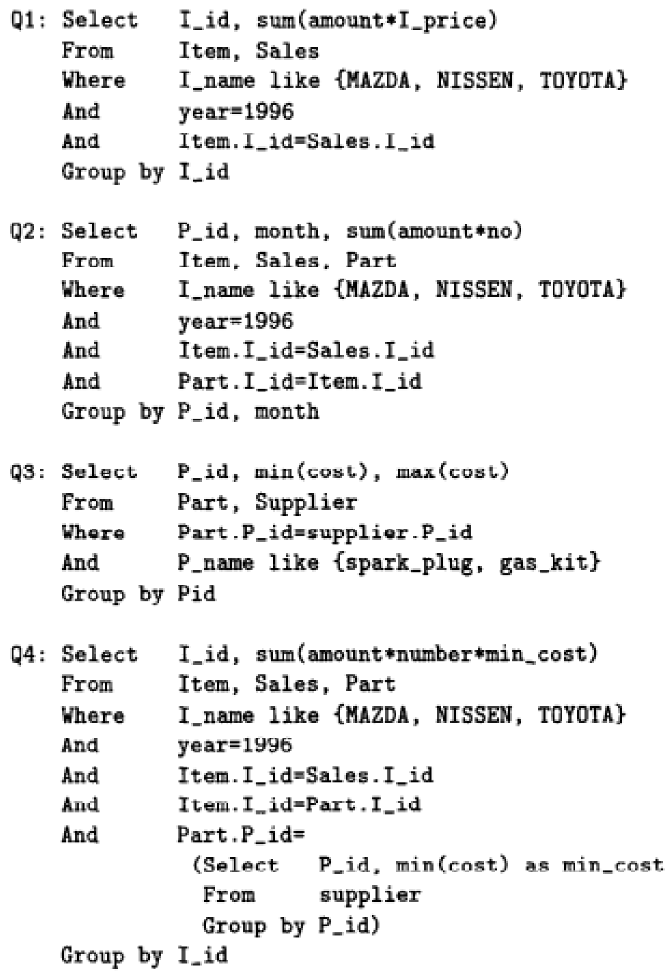
3 Chapter 3
The Materialized View Technique in Relational Databases

Views are relations derived from base relations. They are distinguished from base relations and final results. Materialization means storing the views in the database. Materialized view technique is motivated by four kinds of applications [31]: query optimization, maintaining physical data independence, data integration, and others. It can perform a number of roles according to the applications. We only present materialized view technique in query optimization. Each view is the result of a sub-query. Queries can be computed from materialized views instead of base relations, by reusing results of common sub-queries. The materialized view technique targets three problems : View Design, View Maintenance and View Exploitation. We discuss them separately in the following sections.

3.1 View Design

View Design determines which views should be materialized. On one hand, there is limited space to store the views; on the other hand, the searching cost to find a related view is proportional to the amount of materialized views. Thus, it is not practical to materialize all the views. They must be selected before materialization. View Design has two steps: View Enumeration and View Selection.
View Enumeration : The purpose of View Enumeration is to reduce the number of candidate views to be considered by the selection phase. It filters out the non-related views. This phase is optional.
View Selection : View selection is based on cost-benefit model. Since relational databases use cost-based query optimizer, View Selection can be integrated with the query optimizer easily. The view is beneficial if : (1) It is expensive to compute; and (2)It can be reused by other queries. The cost comes from two aspects: (1)view construction - the overhead to select, create and store the views; and (2)view maintenance - the overhead to keep the views updated. Views will be selected by the query optimizer based on their benefit and cost.

We show some representative view design strategies in the following.
The first strategy selects candidate views from derived relations and evaluates them with cost-benefit algorithm. Representative work is [43].
In [43], the view design technique constructs a multiple view processing plan(MVPP) for candidate view enumeration. Multiple queries sharing common sub-queries would be merged into a single plan called MVPP, as shown in Figure 3.1 [43] and Figure 3.2 [43].



```
Q1: Select   I_id, sum(amount*I_price)
    From     Item, Sales
    Where    I_name like {MAZDA, NISSEN, TOYOTA}
    And      year=1996
    And      Item.I_id=Sales.I_id
    Group by I_id

Q2: Select   P_id, month, sum(amount*no)
    From     Item, Sales, Part
    Where    I_name like {MAZDA, NISSEN, TOYOTA}
    And      year=1996
    And      Item.I_id=Sales.I_id
    And      Part.I_id=Item.I_id
    Group by P_id, month

Q3: Select   P_id, min(cost), max(cost)
    From     Part, Supplier
    Where    Part.P_id=supplier.P_id
    And      P_name like {spark_plug, gas_kit}
    Group by Pid

Q4: Select   I_id, sum(amount*number*min_cost)
    From     Item, Sales, Part
    Where    I_name like {MAZDA, NISSEN, TOYOTA}
    And      year=1996
    And      Item.I_id=Sales.I_id
    And      Item.I_id=Part.I_id
    And      Part.P_id=
              (Select   P_id, min(cost) as min_cost
               From     supplier
               Group by P_id)
    Group by I_id
```

Figure 3.1: Example Queries [43].

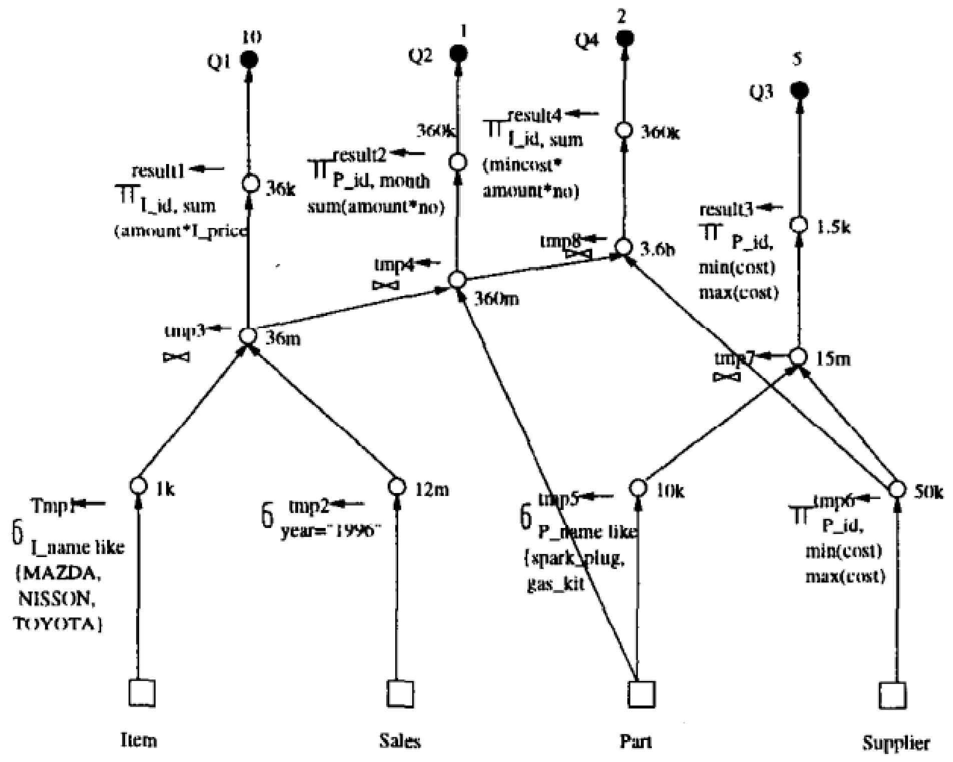

Q1
10
Q2
1
Q4
2
Q3
5
result1
Π I_id, sum (amount*I_price)
36k
result2
Π P_id, month sum(amount*no)
360k
result4
Π I_id, sum (mincost* amount*no)
360k
result3
Π P_id, min(cost) max(cost)
1.5k
tmp8
3.6h
tmp4
360m
tmp3
36m
tmp7
15m
Tmp1
σ I_name like {MAZDA, NISSON, TOYOTA}
1k
tmp2
σ year="1996"
12m
tmp5
σ P_name like {spark_plug, gas_kit}
10k
tmp6
Π P_id, min(cost) max(cost)
50k
Item
Sales
Part
Supplier

Figure 3.2: MVPP of the Example Queries [43].

Different combinations of the queries would result in a set of MVPPs. Thus in View Enumeration phase, the best MVPP would be selected. And in the View Selection phase, views would be selected from the best MVPP.
We first introduce the materialized view design algorithm(HAmvd), which is used in both MVPP Selection and View Selection. HAmvd contains the following steps: first, define a cost-benefit model to evaluate the relations in the execution plan; second, compute the benefit and cost for each relation; third, compute the total benefit and cost of relations in the MVPP and select the one with best benefit gains; fourth, select views in the best MVPP based on their benefit gains.
With the knowledge of HAmvd algorithm, we introduce the two approaches used to get the best MVPP. First is the HAmvpp algorithm. It uses the optimized query plans to generate all possible MVPPs. Then it runs the HAmvd algorithm to get the best MVPP. The second way is to model the MVPP selection problem as 0-1 integer programming(IP) problem, which has been well studied and is easy to solve. This approach would first build a IP problem model to select a subset of join plan trees which can be used to answer all queries and give the minimum total execution cost; then it solves the IP problem to get a MVPP plan formed by the selected join plan

trees.
To conclude : (1) HAmvpp only uses optimized plans to enumerate MVPPs, while MVPP considers all possible plans; (2) HAmvpp may miss the best MVPP; (2) Given n queries, the complexity of HAmvpp is O(n) while that of IP approach is O(2 ˆ n).
The second strategy generates candidate views by merging sharable sub-expressions and evaluates them with a set of cost-based heuristics, represented by [4].
The view design technique in [45] is based on sharable sub-expressions. The definition of sharable sub-expression provided by SQL is : (virtual) views and common table expressions using the WITH clause.
In View Enumeration phase, the query optimizer detects sharable expressions and uses them to generate Covering SubExpressions(CSEs). The query optimizer uses 'table signature'(an abstract of the query that can be used to identify the query) to detect sharable expressions. When the query optimizer receives a batch of queries, it firstly processes them in the normal way; then it calculates the table signature for each expression, looks up the CSE manager to detect sharable expressions and register the new table signatures. After detecting sharable expressions, a set of join-compatible expressions are divided to the same group. Candidate CSEs are generated for each group using greedy algorithm.
In View Selection phase, the query optimizer selects CSEs to materialize. The CSEs should be more beneficial than using its sources separately and should not be contained by other CSEs.
The third strategy both selects candidate views from derived relations and generate views in View Enumeration phase. The candidate views would be evaluated by cost-based metrics. This strategy is represented by [14].
In View Enumeration phase, a candidate selection module is used to shrink the number of candidate views. First, it selects the most interesting base relations. Second, it selects a set of views as candidates for each query associated with the interesting base relations. Third, it applies the 'MergeViewPair' algorithm to create merged views using views selected from the second step. The generated views should be able to answer the queries that can be answered by either of its parent views, which are the views used to create the merged views.
In View Selection phase, both the views selected in the second step and the merged views generated in the third step would be passed to the query optimizer. They would be selected based on cost estimation.

3.2 View Maintenance

View Maintenance refers to the update of materialized views corresponding to the change of base relations. Specifically, when operations such as Insert, Update or Delete are performed on the base relations, the materialized views would get dirty and they must either be updated or garbage collected.
In [24], the view maintenance problems and techniques are discussed. It classifies

the view maintenance problem from four different aspects: information dimension, modification dimension, language dimension, and instance dimension. It also presents several maintenance algorithms that have been proposed in literatures. Which view maintenance algorithms should be used depends on the amount of information available, the modification type, and the characteristics of the language.
We would not show them here, since most parallel processing platforms assume the input data to be append only. Update materialized results is not the issue. Instead, a Garbage Collector is needed to detect and delete the obsolete results, as presented in Chapter 5 and Chapter 6.

3.3 View Exploitation

View Exploitation describes how to efficiently use materialized views for query optimization. View Exploitation includes two phases : View Matching and Query Rewriting. View Matching is to find the related views that can be used for answering queries. Query Rewriting has two kinds : equivalent rewriting and maximally contained rewriting [31]. The former generates an equivalent expression to the original query and gives an accurate answer; while the later would only provide a maximal answer. In this thesis, rewriting refers to equivalent rewriting. It means generating new equivalent expressions with the related views.
We present the most representative match and rewrite algorithms separately.
Representative View Matching Algorithm : The criteria for related view selection is defined in [31]. There are three conditions : (1)the view's base relation set should contain that of the query; (2)for the common attributes, the join and selection predicates of the view must be equivalent or weaker than that of the query; (3)the view should not project out attributes needed by the query. View Matching algorithm in [22] follows this criteria. First, it uses an index structure to filter the irrelevant views, based on base relation set. The views are irrelevant to the queries if their source table sets are less than or different from that of the queries. Second, it checks whether the materialized view could be used for query rewriting with three tests. They are : equijoin subsumption test, range subsumption test, and residual subsumption test. They check if select, join, project and other predicates of the view fulfill the criteria specified in [31].
Representative Query Rewriting Algorithm : In the Query Rewriting phase, a new query equivalent to the query is constructed using the matched views. Given a query, for each matched view returned by the View Matching algorithm, compensating predicates are computed and added. All the rewritten queries are then passed to the query optimizer, and the best one is selected on cost-benefit basis.

Chapter 4
Implementation

We implemented a reuse framework on top of Pig. The framework makes use of previous computation results to avoid computation redundancy and speed up query execution. In this Chapter, we present the implementation details. We introduce the framework design and how each component works and cooperates.

4.1 System Design

In this Section, we explain our design choice and present an overall view of the framework.
The framework is built on top of the Pig/Hadoop, as shown in Figure 4.1.
We explain the design choices in the following.
Pig/Hadoop We choose Pig/Hadoop combination since it is one of the most popular platforms for large scale data processing. Furthermore, both Pig and Hadoop are open source.
MySQL Cluster We choose MySQL Cluster to store the computation's hash and execution statistics. It has fast read/write throughput since it is in-memory database; it provides high availability, scalability and fault tolerance since it is distributed system. Besides, in MySQL Cluster, data are stored in MySQL databases. Thus, we can make use of SQL queries to monitor and operate on the tables.
ClusterJ We need to connect Pig with MySQL Cluster. MySQL Cluster has provided many Java Connectors, as described in Section 2.3.2. And we choose ClusterJ. ClusterJ is simple to use and can well satisfy our needS.
HDFS We store the intermediate results in HDFS, since we expect the intermediate results might be too large to store in memory.
Garbage Collector When implementing the Garbage Collector, we take advantage of MySQL Cluster database and ClusterJ. ClusterJ provides an interface 'Query-Builder' which can send filter queries to MySQL Cluster database. MySQL database would process the queries to filter obsolete records and return the results. We can then remove their corresponding outputs from HDFS and delete them in MySQL Cluster database.

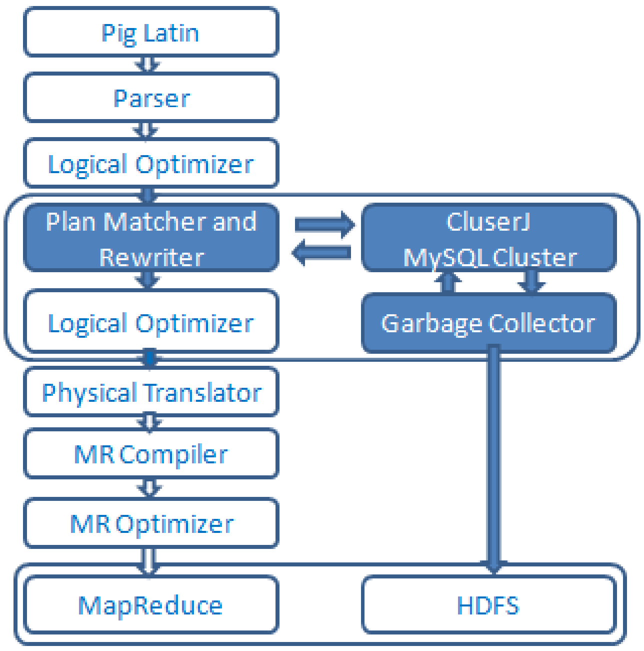


Figure 4.1: Framework Architecture.

The framework consists of four components: Plan Matcher and Rewriter, Logical Optimizer, MySQL cluster, and Garbage Collector.
Plan Matcher and Rewriter is between Logical Optimizer and Physical Translator. Its input is an optimized logical plan and its output is the rewritten logical plan. It saves the fingerprint(a fingerprint of a plan is the hash of the plan) and execution statistics of a plan in MySQL Cluster. Also, it looks up MySQL cluster for a match. Logical Optimizer is used to optimize the rewritten logical plan and its output would be given to Physical Translator.
The Garbage Collector is independent of Pig. It only needs to contact MySQL Cluster and the file system. It gets the obsolete records from MySQL Cluster, and deletes relevant data in both MySQL Cluster and the file system.

4.2 Plan Matcher and Rewriter

In this Section, we introduce the Plan Matcher and Rewriter.
Assume the input is logical plan A. We first give the definition of match and rewrite.
Match : If the fingerprint of plan A or its sub-plans exists in MySQL Cluster and the output exists in HDFS, it is a match.

Rewrite : Generate new logical plan by adding/removing operators from A.
The Plan Matcher and Rewriter is in charge of the following tasks: (1)Choose a set of sub-plans from A to materialize; (2)Calculate the fingerprints of A and A's sub-plans; (3)Store the fingerprint and execution statistics of a plan in MySQL Cluster if no match is found;Rewrite the query if there is a match. We explain each of them in separate subsections.

4.2.1 Sub-plan Selection Criteria

It would be expensive to materialize the outputs of all sub-plans. And also, different sub-plans have different cost and different possibilities to reoccur. Thus, it is necessary to select the most beneficial sub-plans. The sub-plans should fulfill three criteria: (1) The data size of the output is reduced; (2) The computation is expensive; (3)It could be reused by other queries.
In Pig, there are two commonly used operators which can reduce the data size: 'Filter' and 'ForEach'. 'Filter' is often in the very front of the logical plan. Also, the filter conditions tend to change in different queries. 'ForEach' is used together with 'Generate'. It may perform project operation, by pruning unwanted columns. It can also be used for embedded queries, which makes it complicated to deal with.
'CoGroup' and 'Join' are two commonly used operators which are known to be expensive. As shown in Figure 2.3, 'Filter' and 'Group'(a special type of 'CoGroup') would be translated to three physical operators: local rearrange, global rearrange and package. Local rearrange corresponds to the end of map task and package corresponds to the beginning of reduce step. Hadoop would store the intermediate results between map jobs and reduce jobs and between different MapReduce jobs to a temporary path on disk. And the temporary path would be deleted after getting the final results. So there is an I/O overhead of 'Join' and 'CoGroup' to store the intermediate result to disk even if we don't store their outputs. Then if we store the outputs of 'Join' or 'CoGroup', the introduced overhead should not be significant.
We assume I/O overhead is dominant. We define a cost-benefit model to choose operators of which the output should be saved:

$$C = \frac{\sharp\ common\ computations\ covered}{Size\ of\ intermediate\ results\ to\ materialize}$$

The selection strategy should introduce the least I/O overhead while covering as more reuse opportunities.
Based on the above analysis, 'CoGroup' and 'Join' are the best choices whose output should be materialized. Furthermore, there is no solution that is the most suitable for all the workloads or applications. It's better to combine the selection strategy with specific workload. We show more details in Section 5.3.

4.2.2 Fingerprint Computation

A query can be uniquely determined by its input and its logical plan. Input is the data to be processed and logical plan represents which computations would be performed on the data. We assume the input files can be uniquely determined by their filenames or paths. Also we assume filenames don't change(if the user changes the filename, then it becomes a new file). We first calculate the fingerprint for logical plan, then we calculate the fingerprint for input files and use their concatenation to identify a query.

4.2.2.1 Logical Plan Composition

The logical plan is a directed acyclic graph composed by operators and directed edges. There are two kinds of operators used to express a logical plan: relational operators and expression operators. Relational operators extend the LogicalRelationalOperator class and expression operators extend the LogicalExpressionOperator class. Relational operators include 'Load', 'Filter', 'Join', 'Store' and so on. Expression operators include 'Constant', 'And', 'Or', 'Multiply' and so on. Both relational and expression operators can be represented in string format by calling the toString() method. The toString() method has been overridden and it contains all useful information of an operator, such as name and schema.
There are also two kinds of logical plans which either extend the LogicalPlan class or LogicalExpression class. The former is composed by relational operators and the later is composed by expression operators. An instance of LogicalExpression can only be inner-plan of relational operators.
The logical plan A is an instance of the LogicalPlan class. It is composed by relational operators. Some relational operators contain inner-plans, such as 'Filter', 'Join', 'Group', and 'Generate'. The inner plan can be either instance of LogicalPlan or LogicalExpression. For example, the inner-plan of relational operator 'ForEach' is an instance of LogicalPlan, and the inner-plan of relational operator 'Filter' is an instance of LogicalExpression.
We can conclude that, a logical plan is composed by operators, and since all operators can be represented by a string, the logical plan can also be represented by a string.

4.2.2.2 Logical Plan Fingerprint Computation

To compute the fingerprint of logical plan A, we do the following steps: (1)Use depth first transverse algorithm to get a list of ordered operators in plan A, as shown in Figure 4.2; (2)Concatenate the string representations of the operators with a StringBuilder; (3) Calculate the hash of the string in the StringBuilder by calling hashcode() method. Oracle has defined the hash computation for a String object as: s[0]*31^ (n-1)+s[1]*31^ (n-2)+...+s[n-1] , s [i] represents the ith character of the string, n indicates the length of the string, and ^ represents exponentiation [3].

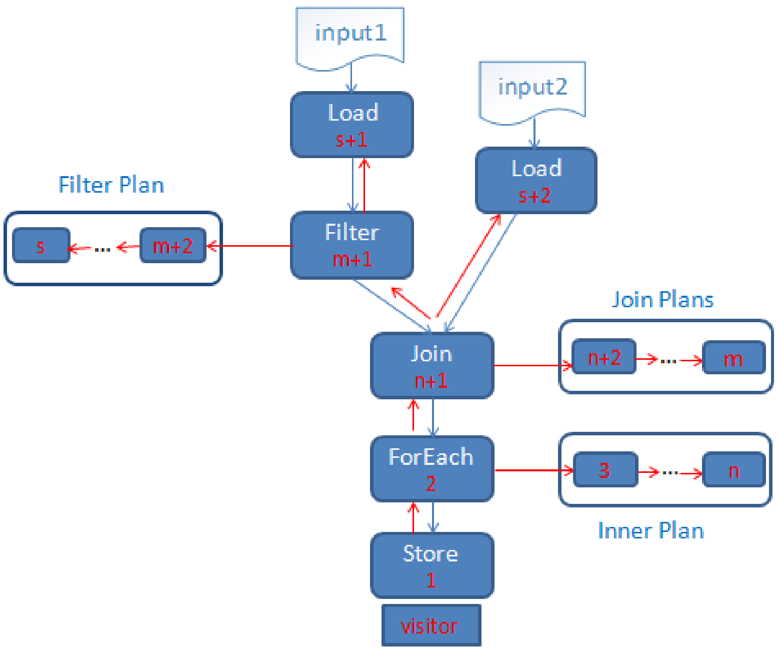


Figure 4.2: Transverse the Logical Plan A to get its operators.

Apache Pig's getSignature() method in LogicalPlan class has followed the above steps to calculate the hash of a logical plan. However, it is only used to set the property value of the plan's fingerprint. We extend the getSignature() method so that we not only get the fingerprint of plan A, but also get the fingerprint of A's sub-plans during the transverse.
First, we do depth first transversal of logical plan A starting from leaves(A leaf is an operator with no successors).
Each operator has a StringBuilder. During the transversal, each operator gets its string format, and appends it to the StringBuilder. If the operator contains inner-plans, it would transverse the inner-plans, and append the string representation of its inner-plans to its StringBuilder. The operator then recursively gets the content of its predecessor's string builder, and appends it to its own StringBuilder. Thus the content in each operator's StringBuilder can represent a sub-plan, and this operator is the sub-plan's leaf. Figure 4.3 shows the sub-plans of logical plan A. A has four sub-plans, sub-plan 1(plan A), sub-plan 2, sub-plan 3 and sub-plan 4.

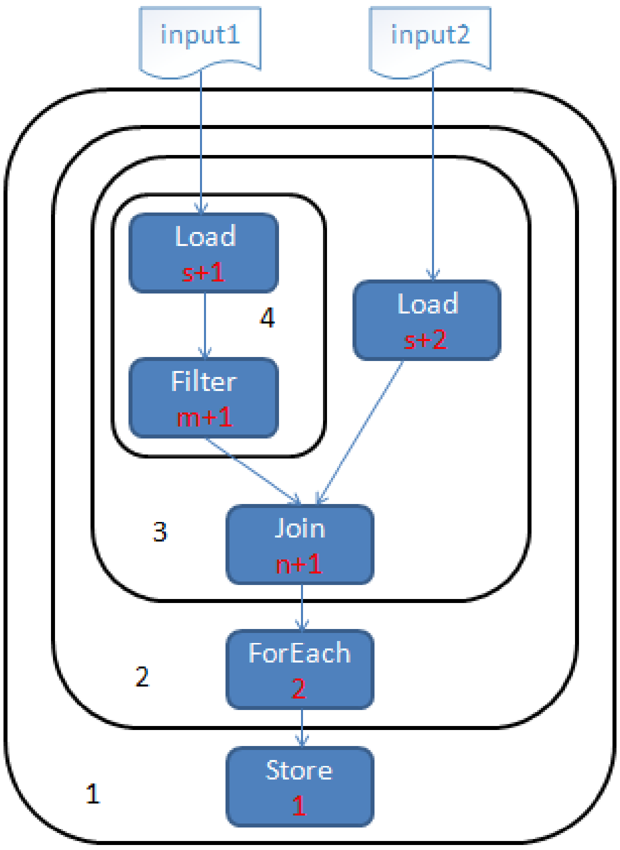


Figure 4.3: Sub-plans of plan A.

The mapping from an operator to the corresponding sub-plan is shown in Table 4.1. We cast the StringBuilder of each operator to a string, and call the hashcode() method to calculate the hash of the string. Then we get the hash of the corresponding sub-plan. We set the hash for each operator before it returns the content in its StringBuilder to its successor. Thus, the hash of each operator represents the hash of its corresponding sub-plan.

Second, we compute and match the fingerprint in good order. We define good order as following : Plan C contains plan D if C contains all the operators and edges in D. Good order means, if sub-plan C contains sub-plan D, then the hash of sub-plan C is always computed and matched before sub-plan D.

As shown in Figure 4.3, logical plan A has four sub-plans. Sub-plan 1(plan A) contains sub-plan 2, sub-plan 2 contains sub-plan 3, and sub-plan 3 contains sub-plan 4. The algorithm must first compute and match the hash of the sub-plan 1, then sub-plan 2, 3 and 4 sequentially. This order is consistent with the benefit of the sub-plans.

To perform match and rewrite in good order, we create a list (List<Operator>)

Owner of the String-Builder	Operators with string in StringBuilder (in order)	Corresponding sub-plan in plan A
Store^1	Store^1, ForEach^2, Join^{n+1}, Filter^{m+1}, Load^{s+1}, Load^{s+2}	sub-plan 1 (A)
ForEach^2	ForEach^2, Join^{n+1}, Filter^{m+1}, Load^{s+1}, Load^{s+2}	sub-plan 2
Join^{n+1}	Join^{n+1}, Filter^{m+1}, Load^{s+1}, Load^{s+2}	sub-plan 3
Filter^{m+1}	Filter^{m+1}, Load^{s+1}, Load^{s+2}	sub-plan 4
Load^{s+1}	Load^{s+1}	N/A
Load^{s+2}	Load^{s+2}	N/A

Table 4.1: Mapping from Operators to Sub-plans

to keep the relational operators of the plan in depth first order. The operators in the list are in the same order as being visited during the transversal. The Plan Matcher and Rewriter gets this list, and iterates over operators in the list. During the iteration, the Plan Matcher and Rewriter gets the hash of each operator(or the operators defined by sub-plan selection strategy) and looks for a match in MySQL Cluster database.
Third, we modify the toString() method of each operator by removing unrelated information, for example 'alias'. Alias is the name of relations or column fields. In this way, the hash of "A= load 'input' as (x,y,z) " is equal to the hash of "B=load 'input' as (a,b,c)".
By following the above steps, we can compute and match the sub-plans of logical plan A in good order.
The last step, we attach the information of input to the logical plan.
We get the filenames from 'Load' operators. During the transversal, we record the input filenames in a list for each sub-plan, and they are in the same order as being visited during transversal. We use a StringBuilder to concatenate the input filenames. Then we cast the StringBuilder to a string and calculate the fingerprint of the string. The final fingerprint for the query is the concatenation of the logical plan's fingerprint and the input's fingerprint.

4.2.3 The Match and Rewrite Algorithm

4.2.3.1 Execution of the Match and Rewrite Algorithm

We have implemented the Match and Rewrite Algorithm to reuse previous computation results. The execution diagram is shown in Figure 4.4. The input is an optimized logical plan, and the output is the rewritten plan.

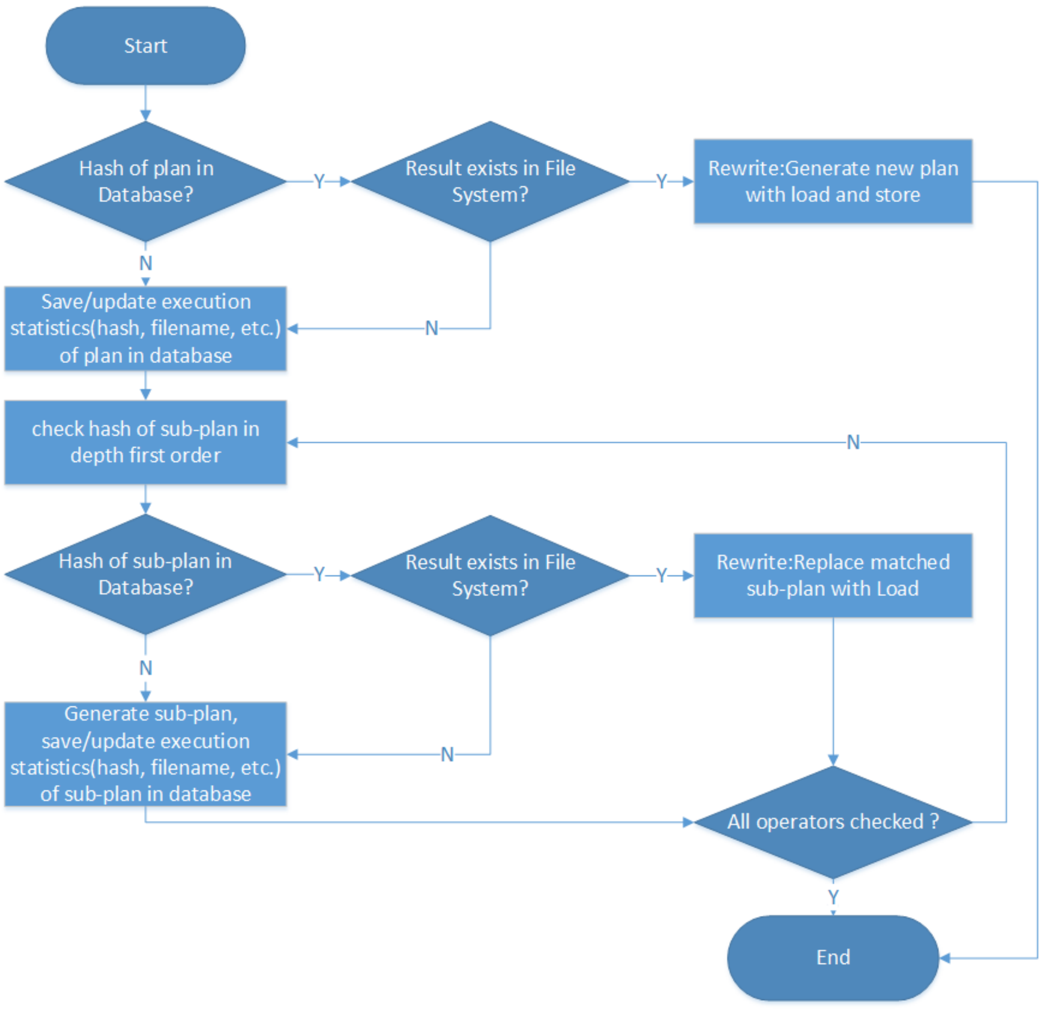

Start
Hash of plan in Database?
Y
Result exists in File System?
Y
Rewrite:Generate new plan with load and store
N
Save/update execution statistics(hash, filename, etc.) of plan in database
N
check hash of sub-plan in depth first order
N
Hash of sub-plan in Database?
Y
Result exists in File System?
Y
Rewrite:Replace matched sub-plan with Load
N
Generate sub-plan, save/update execution statistics(hash, filename, etc.) of sub-plan in database
N
All operators checked ?
Y
End

Figure 4.4: The execution diagram of Match and Rewrite Algorithm.

Assume the input is logical plan A.We use an example to illustrate the Match and Rewrite process, as shown in Figure 4.5.

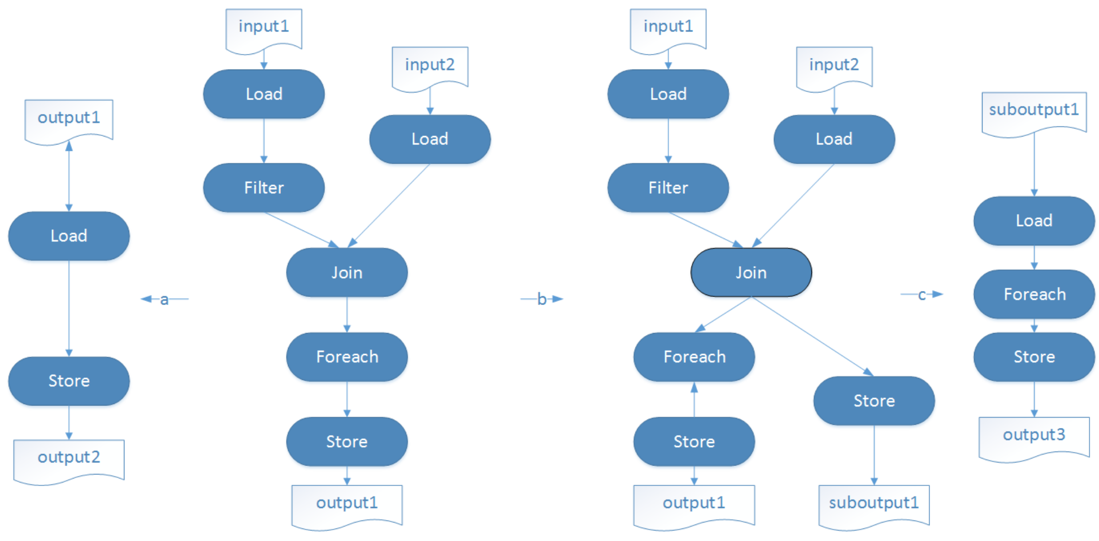


Figure 4.5: Match and Rewrite a Logical Plan.

First, if logical plan A finds a match in MySQL cluster database, and the result exists in file system(in our case, HDFS), we generate a new plan with 'Load' and 'Store' operator, as shown in Figure 4.5-a.
Second, if the plan failed to find a match in MySQL Cluster database or the result 'output1' does not exist in the file system, the Plan Matcher and Rewriter checks if the sub-plans of plan A can find a match in MySQL Cluster database. As discussed in Section 4.2.2.2, we compute and match the hash of sub-plans in depth first order. The Plan Matcher and Rewriter gets the operator list of the plan, and the operators are in depth first order. The hash of each operator corresponds to a sub-plan. The Plan Matcher and Rewriter then iterates over the list to check if a match can be found in MySQL Cluster database.
If the sub-plan failed to find a match or the output does not exist, the Plan Matcher and Rewriter would generate sub-plans by inserting 'Store' operator, as shown in Figure 4.5-b. If a match is found and the result exists in file system, the plan matcher and rewriter replaces the matched sub-plan with a 'Load' operator, as shown in Figure 4.5-c. It then skips all the operators that should be removed from the plan and continues to check other operators, until reaching the end of operator list.

4.2.3.2 Restrictions of the Match and Rewrite Algorithm

Currently, the Match and Rewrite Algorithm only deals with execution plan with one leaf(leaf is the operator with no successors). For logical plans with multiple leaves, we have a few issues with limited time to explore. One of them is the graph disconnection problem. Consider the following case: the input of the Plan Matcher and Rewriter is logical plan A1 which has two leaves, as shown in Figure 4.6.

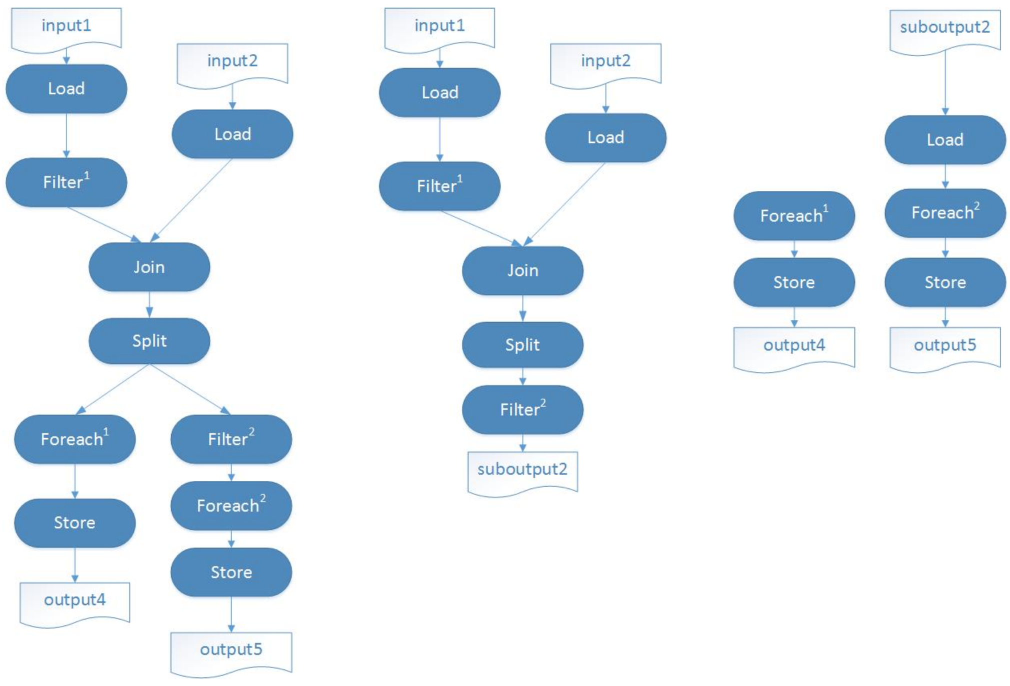


Figure 4.6: The Graph Disconnection Problem.

There is no match for plan A1, but there is a match for the sub-plan with a output 'suboutput2'. Then if we replace the matched sub-plan in A1 with 'Load', the graph becomes disconnected.
It would be complicated to deal with this case. Furthermore, the logical plan of a query which contains lots of embedded queries can be very complex. Of course, there might also exist reuse opportunities in these queries, for example, if we consider only the part of the plan without multiple outputs. However, this indicates the reuse opportunities to be limited. Besides, we have limited time to conduct tests for possible complex queries. Thus we leave the work for logical plan with multiple leaves for future work. And currently, if the input plan has multiple leaves, the match and rewrite algorithm would do nothing and just return the input plan A1 as its output.

4.2.4 Saving the output to MySQL Cluster with ClusterJ

The MySQL Cluster contains one management node, one sql node and two data nodes. We followed the guide on MySQL Cluster website [9] to install and configure the cluster. We would not present the details here. We use ClusterJ to connect Pig with MySQL Cluster.

4.2.4.1 Map Java Object with Table in MySQL Cluster database

ClusterJ provides a mapping from Java objects to the view of data in MySQL cluster. The Java object can be represented by an annotated interface. The annotated interface maps to the table in MySQL Cluster database. Each property in the interface maps to a column in the table. All property names are the same as column names by default. Each property has getter and setter methods. ClusterJ cannot create the table automatically thus the table must be created first unless it already exists in MySQL cluster.
We connect to the SQL node in MySQL cluster and create the table 'repository' with four columns: hashcode - hash of the plan; filename - output path of the plan; frequency - the reuse frequency; last_access - last time to access the record.
Correspondingly, we created an annotated interface Repository.java in Pig. It has four properties: hashcode(primary key and default first index), filename, frequency, and last_access.

4.2.4.2 Connect Java application with MySQL Cluster

ClusterJ uses the SessionFactory interface to get a session instance, which represents the connection from Java application to MySQL cluster. The session interface contains a set of methods. We can use session to create a instance of the annotated interface, which maps to a row in the table of MySQL database. We can also use session to perform set, get, update and delete operations on the instances. [8] shows in detail how to configure SessionFactory, how to get an session and how to use session to operate on the table of MySQL Cluster database. We are not going to repeat here. We will show the overhead of MySQL Cluster read/write in Chapter 5.

4.3 Logical Optimizer

The logical optimizer optimizes the rewritten logical plan. It's necessary for the following reasons. First, 'Store' operators are inserted to generate sub-plans. Since 'Split' is the only operator that is allowed to have multiple outputs, we must apply the rule of 'ImplicitSplitInserter' to insert a 'Split' operator. Second, the matched sub-plan is replaced by a 'Load' operator, which might not be able to recognize the type of the input data. We must of the rule of 'TypeCastInserter' to translate the data type. There might be other cases which require optimization. Thus we used Pig's optimizer.
Since the logical plan has been optimized before the Match and Rewrite phase, we expect the overhead of optimization to be small. We present the optimization overhead in Chapter 5.

4.4 Garbage Collection

If the amount of records in the MySQL database keeps increasing, the overhead to search for matched plan would increase and the available disk space would decrease. We use a garbage collector to detect and delete the obsolete materialized results. Obsolete means they haven't been accessed for a long time or they have a less reuse frequency.
We implement the garbage collector as a separate component. It only needs to contact the MySQL Cluster and the file system. The Garbage Collector can be initiated by the user. The Garbage Collection process includes three steps: filter out the obsolete records; delete corresponding outputs in the file system; delete records in MySQL database. We use the QueryBuilder interface in ClusterJ to filter obsolete records in MySQL database. Then we delete both the records in MySQL database and the corresponding outputs in the file system.
Garbage Collection is based on reuse frequency and last access time. Since the record that is reused frequently tends to have a more recent access time than the less reused ones, our garbage collector is only based on the last access time threshold, which is a user-defined configuration parameter. For example, if we set the threshold to be 5 days, then the records that have not been accessed within 5 days from the time we run the garbage collector would be detected and deleted. In this case, the materialized results that have never been reused would also be deleted, since their last access value is 'NULL'. If necessary, the reuse frequency threshold could also be specified by user.

Chapter 5 Evaluation

5.1 Cluster Configuration

To evaluate our framework, we configured one Hadoop cluster and one MySQL cluster.
Virtual Machines We have used 20 Ubuntu Linux virtual machines from SICS(Swedish Institute of Computer Scienc) in total. On each node, the Ubuntu version is 11.10 and The Java(TM) SE Runtime Environment version is 1.7.0.
Hadoop We used Hadoop version 1.0.4. The Hadoop cluster has one Namenode and fifteen Datanodes. Hadoop Namenode has 16 GB of RAM , 8 cores and 160 GB disk; each Hadoop Datanode has 4GB of RAM, 2 cores and 40 GB disk. We configured Hadoop cluster with the following parameters:
• 15 mappers and 9 reducers in maximum
• block size of 64MB
Pig For convenience, we installed Pig on the Hadoop Namenode. We used Pig version 0.11.1.
MySQL Cluster We used MySQL cluster version 7.2.12. MySQL Cluster has one manage node, one SQL node and two data nodes. Each node has 4GB of RAM, 2 cores and 40GB disk. We use the default parameters to configure memory for data storage(80 M) and index storage(18 M).

5.2 Test Data and Test Query Sets

In this section, we present the test data and workloads. We have used TPC-H Benchmark [11] for our evaluation. We introduce the TPC-H Benchmark and show how to use to generate data and test workloads.

5.2.1 The TPC-H Benchmark

The TPC-H Benchmark [41] represents the activity of selling or distributing a product worldwide for any industry. The data model contains eight data sets in total:

part, partsupp, orders, nation, lineitem, supplier, region, and custom. It consists of twenty-two SQL queries which are used to analyze complex business problems. Jie etc. [32] has rewritten the twenty-two SQL queries in Pig Latin and made them run on Pig. We have used their work.
We choose TPC-H Benchmark to do evaluation for the following reasons:
(1)The queries are highly complex and involve multiple datasets;
(2)The queries are different from each other, and their logical plans vary from each other;
(3)The queries contain substitution parameters which change across query execution.
The first two characteristics guarantee the complexity and variability of the test workloads. The third characteristic is very important, since the substitution parameters determine whether there exist reuse opportunities and where the reuse opportunities lie. We illustrate this point with an example. In the example, we use Query 20 from TPC-H Benchmark.
Example : Potential Part Promotion Query (Q20).
The Potential Part Promotion Query identifies suppliers in a particular nation having selected parts that may be candidates for a promotional offer. The logical plan of the query is shown in Figure 5.1. We marked the substitution parameters with red color.

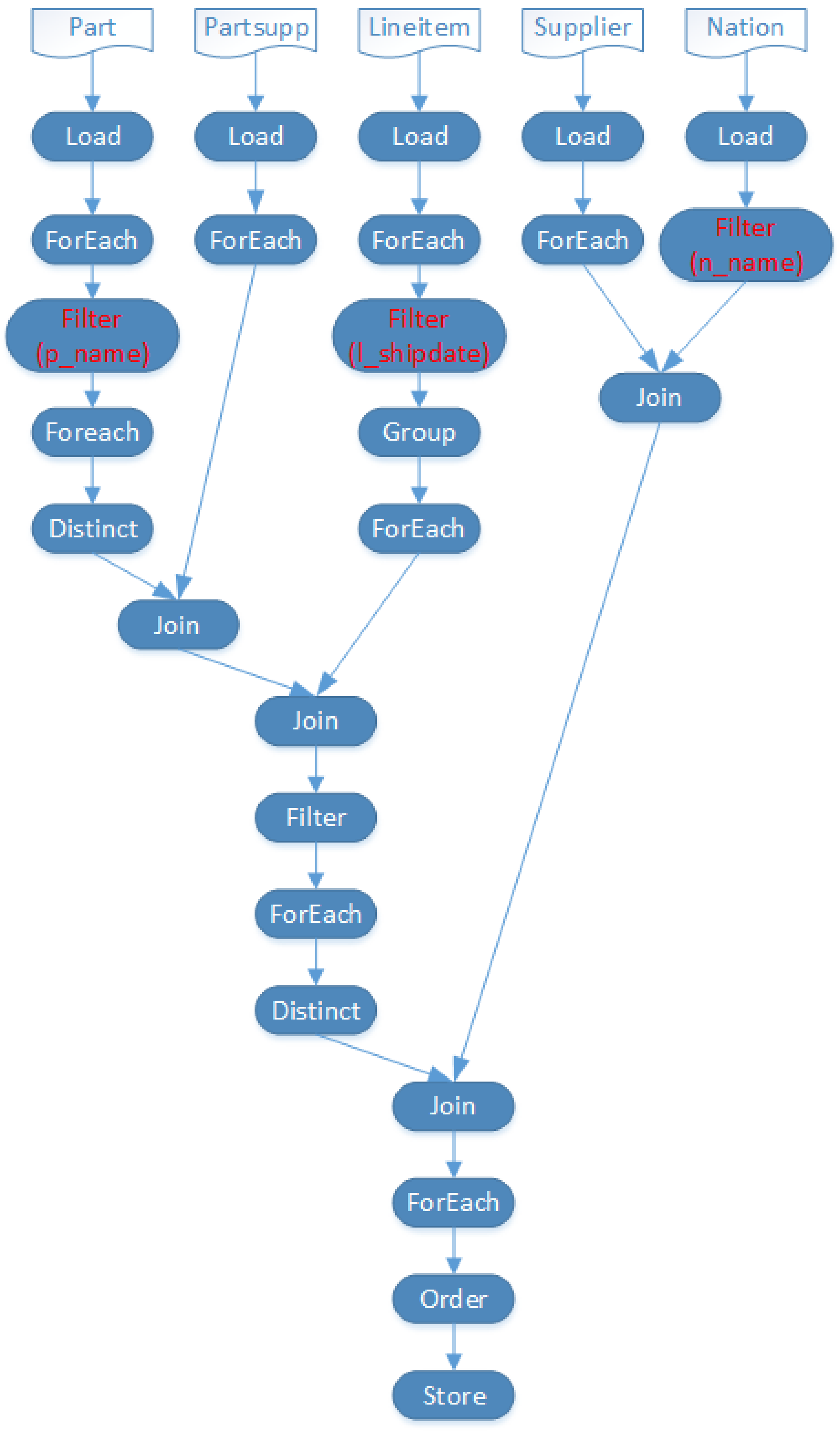


Figure 5.1: logical plan of TPC-H Query 20.

As shown in Figure 5.1, there are three substitution parameters: p_name, l_shipdate and n_name. P_name can be selected from a list of 92 strings representing color; l_shipdate is the date to ship product, it is on January first and the year can be selected from 1993...1997; n_name can be selected from the name list of 25 countries. If we change some parameters(for example: n_name) and keep the rest unchanged, then the output of unchanged part can be reused.

5.2.2 Generate data and query sets

We generate the data sets with the DBGEN tools of TPC-H Benchmark. We have generated 107 GB data in total. There are eight data sets, as mentioned in Section 5.2.1.
As introduced in Section 5.2.1, the twenty-two queries are different from each other, thus there are no sharing opportunities between them. However, each query is not used to answer a single problem, but a group of problems with different substitution parameters, as we illustrated by Query 20. Therefore, we created some new queries by changing substitution parameters.
Finally, we selected twenty queries in total for evaluation, with fourteen from the original TPC-H Benchmark queries and six new created queries.

5.3 Sub-query Selection Strategy

We always store the execution statistics of final results in MySQL Cluster. For sub-query generation, we discussed the selection criteria in Section 4.2.1. In general, 'Join' and 'CoGroup' are the best choices and their output should be stored; while 'Filter' and 'ForEach' could also be considered if necessary. We analyzed the queries in TPC-H Benchmark. They are very complex and usually contain more than twenty operators, and most of them fulfill the selection criteria, as shown in Figure 5.1. It is necessary to further reduce the amount of intermediate results to materialize while covering as more common sub-queries.
We have three findings:
First, the 'Filter' operator is used at very early stage and it is where the modifications occur. If there is only one filter operation in the plan, and the filter condition changes, then there is no reuse opportunities for other queries. In this case, it is better not to store any intermediate results or only store the output of the most expensive operations.
Second, some queries have more than one substitution parameters, such as Query 20. If the filter condition of a 'Filter' operator stays unchanged, then its output can be reused. However, we noticed that there usually exists one or more expensive operators ('Join' or 'CoGroup') which use the output of 'Filter' as input. If the output of 'Filter' can be reused, then the output of its expensive successors can also be reused. Thus it is not the best choice to reuse the output of 'Filter' directly.

Storing the output of the expensive operators is enough.
Third, if one operation in the plan changes, all the operations rely on this operation have to be recomputed. For logical plan with multiple inputs, if one input branch changes, then the final 'Join' of all the inputs must be recomputed. Thus, if the plan has multiple inputs, we only materialize the intermediate results on each input branch before the final 'Join' operator. The TPC-H queries usually contain multiple inputs, such as Query 20. Starting from the 'Store' operator, Match and Rewrite would not be performed until reaching the beaches.
Based on the above analysis, we choose to store the output of 'Join' and 'CoGroup' operators only. And if the plan has multiple inputs, we would not store intermediate results between the final 'Join' Operator and the ' Store' operator.

5.4 Results

In this section, we present the results for the framework evaluation. As stated in Section 5.2.2, we have twenty queries in total for the evaluation. The first fourteen queries are from the TPC-H Benchmark, and the last six queries are created by changing substitution parameters of TPC-H queries. We run them on Pig without and with our framework. For each test, we run the query sets for five times, and calculate the average. If the variation is big, we run more times and make sure the errors are small enough. Finally, the error range is between 3% ˜ 8%.
We present the overhead and benefit of materializing and reusing intermediate results, the performance of our algorithm and MySQL Cluster.

5.4.1 Reusing the Outputs of Sub-jobs

5.4.1.1 Overhead of the first Fourteen Queries

We first run the fourteen queries from TPC-H Benchmark, to enable the reuse property. We show the total overhead of storing intermediate results in Figure 5.2. It includes the overhead of Match and Rewrite, optimization, storing execution statistics to MySQL Cluster, and storing results to disk. We assume I/O overhead is dominant. The numbers on x axis are in correspondence with the query numbers in TPC-H Benchmark.

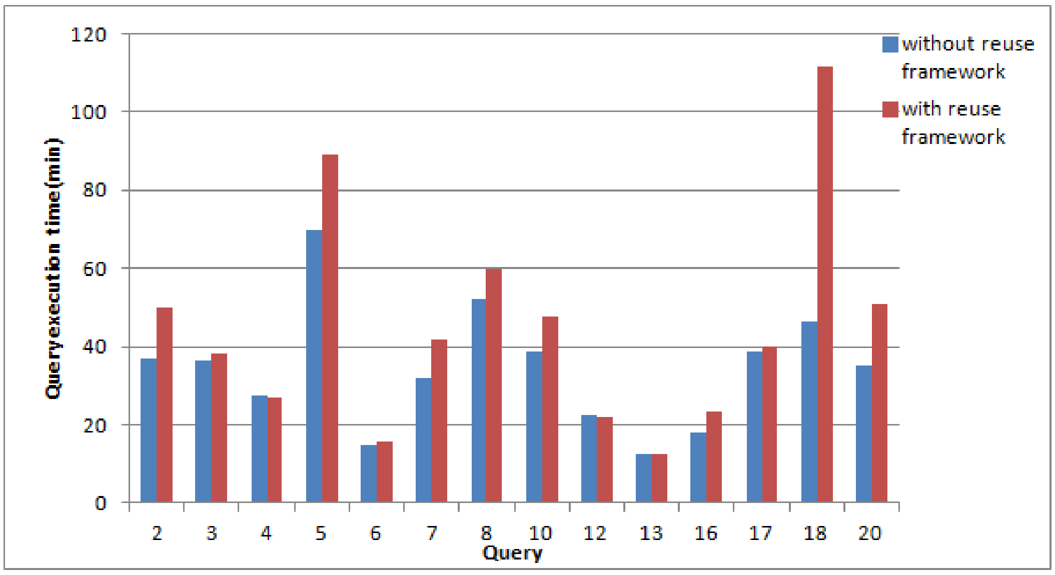


Figure 5.2: The Overhead with Reuse Framework (Empty Database).

With the reuse framework, three cases are observed from Figure 5.2.
First : There is no overhead or negligible overhead(for Query 4, 6, 12, 13 and 17). In this case, the size of materialized intermediate results must be 0 or very small.
Second : Very small overhead, from 5% to 25%(for Query 2, 3, 5, 7, 8, 10, 18 and 20). This is the expected case. The queries produce some useful intermediate results with small overheads.
Third : Very large overhead(for Query 18), around 120%. This can happen when the size of intermediate results is very large while it fulfills the selection criteria. First, we expect this case to be not so frequent, since usually there would be a filter or project operation before 'Join' and 'CoGroup'; Second, since the results are stored for only once but can be reused multiple times, the overhead might be worthy.

5.4.1.2 Benefit of Reusing Sub-jobs

After materializing the outputs of the first fourteen queries, we run the last six queries which share common sub-queries with them. The result is shown in Figure 5.3. We show in the braces from which queries they are created from and they share common sub-queries with.

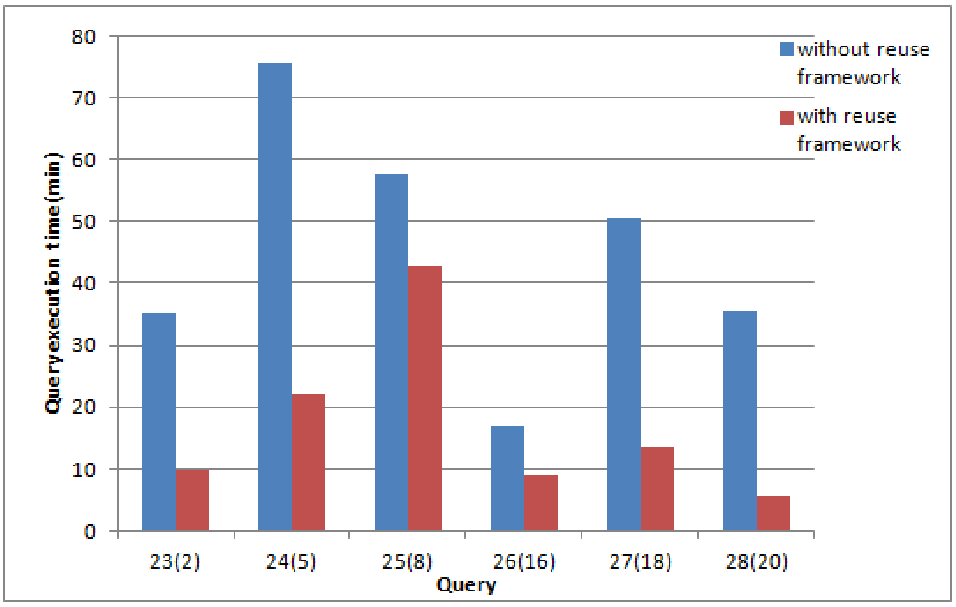


Figure 5.3: The Benefit with Reuse Framework(Database with first 14 Queries).

As can be seen, except for Query 25, which has the lowest speed up of 30 %, the other queries have a speed up between 50% ˜ 80%. The reason that Query 25 has a lower speed up is we didn't change the substitution parameters for the I/O dominant part of the query. Besides, while these queries are reusing previous computation results, they might also have intermediate results to materialize, though this cannot be observed from Figure 5.3.

5.4.1.3 The Size of Intermediate Results

We assumed I/O overhead is dominant, thus we show the size of materialized intermediate results in Figure 5.4 to study its effect on the query execution time and disk usage.

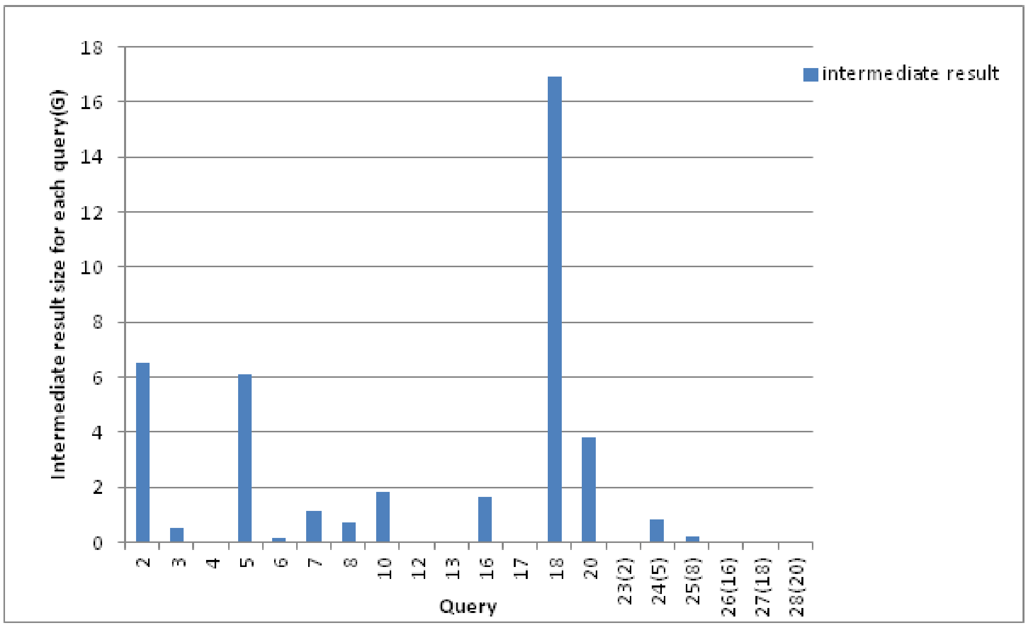


Figure 5.4: The size of intermediate results.

We have three findings.
First, comparing Figure 5.2 with Figure 5.4, it shows that the size of intermediate results is in consistent with the overhead. After examining the queries with little or no intermediate results to materialize, we find that they fulfill the first case of our selection strategy. They don't have sharing opportunities with the other queries. Thus, we have avoided storing useless intermediate results.
Second, comparing Figure 5.3 with Figure 5.4, it shows that the materialized results of Query 18 has speed up the execution of Query 27 significantly(around 85%), though it introduces a big overhead. The materialization is worthy since it would probably be reused for many times. Besides, it shows that the speedup is good even if materialization overhead is introduced at the same time(for Query 24 and 25).
Besides, we calculated the disk usage of intermediate results for all the queries, around 40 GB. Comparing with the total input, which is 107 GB, the extra disk usage is acceptable.

5.4.2 Reuse Outputs of Whole jobs

After we have run all the twenty queries and stored the relevant data in both MySQL Cluster and the file system, we run them again to see the benefit of reusing output of whole jobs. The results is very good and the execution time is around 30 ~ 45 seconds. We show the speedup of queries, which is the execution time without reuse framework divided by the execution time with reuse framework, in Figure 5.5.

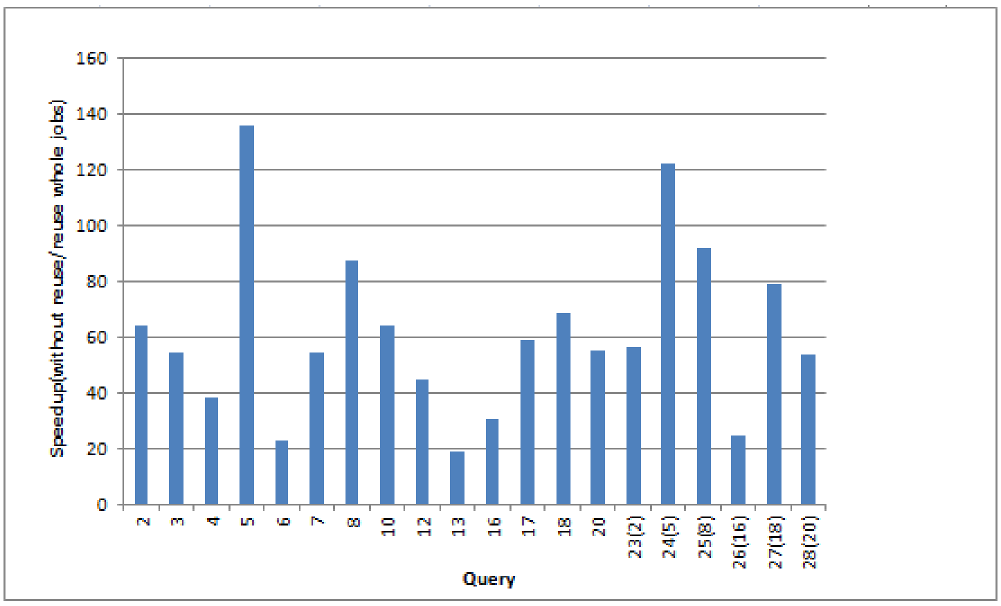


Figure 5.5: Speedup when reusing whole jobs.

The speedup between execution time without reuse framework and reuse whole jobs is from 20 to 140 approximately. It depends on the base execution time, the size of the final results and the execution time variance.
The total execution time of the twenty queries is around 753 minutes without reuse benchmark; while the total execution time of reuse whole jobs is around 13 minutes. It is not very often to run the same query again, but it almost cost noting to store the final results and its execution statistics. In case the same query is submitted, the execution would just take several seconds instead of minutes or hours.

5.4.3 The non-I/O overheads

The non-I/O overheads are introduced by Plan Matcher and Rewriter and access to MySQL Cluster. They are shown in Table 5.1.

Operation	Time(ms)
Compute fingerprint	10.48
Create session to access MySQL cluster	780.56(62.6%), 5797.84(37.4%)
Check existence of matched plan	83.57
Save a new record to MySQL database	2.17
Read a record from MySQL database	20.03
Match and rewrite(no reuse)	138.32
Match and rewrite(reuse sub-jobs)	186.79
Match and rewrite(reuse whole jobs)	148.92
Optimize(no reuse)	9.79
Optimize(reuse sub-jobs)	6.59
Optimize(reuse whole jobs)	1.18

Table 5.1: The non-I/O overheads of the Framework.

Table 5.1 shows that the non-I/O overheads are negligible comparing with I/O overheads.
Create session to access MySQL cluster takes the longest time. We collected around 200 values of session creation time. 62.6% of the values are around 780 ms; for the rest, it takes around 6 seconds to create a session, which indicates the network connection might be bad.
The Match and Rewrite Algorithm takes less than 200 milliseconds. It includes computing fingerprint, checking the existence of matched plan and MySQL Cluster read/write operations. Check existence of matched plan takes the longest time, around 83 milliseconds.
Optimize the rewritten plan takes very little time, less than 10 milliseconds.

6 Chapter 6

Related Work

In this Chapter, we present research on optimization in the Big Data area, based on different parallel processing systems. In general, there are two categorizations: concurrent work sharing and non-concurrent work sharing [36]. Concurrent work sharing is similar to multi-query optimization in relational databases. It tries to find sharable parts among multiple queries and maximize the sharing benefit. Concurrent sharing includes: shared scan, shared computation, shared shuffling and so forth. Non-concurrent work sharing resembles the materialized view technique. It materializes the intermediate and final computation results, and uses them to answer queries.

6.1 Optimization based on MapReduce

MapReduce is one of the most popular parallel processing platforms. It is introduced in Chapter 2.1.1. There are several works on optimizing MapReduce jobs.
MRShare [35] is a **concurrent sharing** framework. It assumes I/O cost is dominant. Based on this assumption, it considers the following sharing opportunities : Sharing Scans, Sharing Map Output and Sharing Map Functions.
Sharing Scans : If the input of two map tasks are the same, the input data would be scanned for only once. Scan is considered be dominant part of the I/O overhead. In [13], sharing scans of large data files is discussed in detail.
Sharing Map Output : If the output of two mapping tasks have overlapping key-values pairs, they would be tagged by both origins and they would be sorted and transfered for only once.
Sharing Map Functions : If both the map functions and the input of two map tasks are identified to be the same, the map function would be executed for only once. It is also possible to share parts of map functions.
Sharing Scans and Sharing Map Output reduces the I/O overhead; Sharing Map Functions is similar to multi-query optimization and avoids computation redundancy for a batch of queries executed at the same time.
Restore [19] is a **non-concurrent sharing** system built on top of Pig. It uses

materialized results for query optimization. The idea is similar to our framework but the implementation is quite different.
Restore consists of four components: sub-job enumerator, matcher and rewriter, the repository and the sub-job selector. The sub-job enumerator selects sub-jobs to materialize and the matcher and rewriter performs match and rewrite. In our implementation, the Plan Matcher and Rewriter plays the role of both the two components. The repository is used to store execution statistics, and it plays the same role to MySQL Cluster in our framework. The sub-job selector performs Garbage Collection.
The major differences between Restore and our framework are as following.
First : As claimed in the paper, Restore works at physical layer; while our framework is build on top of logical layer.
Second : Restore stores and matches the plan object; while we use fingerprints to identify and match a plan.
Third : It didn't mention what is used as its repository, and the garbage collection is not implemented; while we used MySQL Cluster to store execution statistics and we take advantage of it to do Garbage Collection.
Incoop [16] is a **non-concurrent sharing** framework that can reuse results for incremental computation. It assumes the file system is append-only. Two main techniques are used : Detect input data changes and materialization.
Detect input data changes : Incoop uses Incremental Hadoop Distributed File System(Inc-HDFS) to detect changes of input data. It uses content-based chunks instead of fixed-size chunks to locate files. The boundary of the chunk is determined by the content of the file.
Materialization: It includes incremental map phase and incremental reduce phase, and both the outputs of map tasks and reduce tasks are materialized. It uses hash to identify the chunks.
In incremental map phase: (1) the changed chunks and unchanged chunks are identified by Inc-HDFS; (2) the map results of the unchanged chunks are fetched from the file system and unchanged chunks are processed by mapper.
Incremental map can avoid re-mapping the same input. However, small changes in map results may cause re-execution of the entire reduce task. Thus, incremental reduce not only stores the final results, but also stores sub-computation results by introducing a contraction phase.
In the contraction phase, the input of a reduce job is split into chunks and they are processed by the combiners. The results of the combiners are materialized. Thus in incremental reduce phase, the combination results of unchanged chunks can be fetched from file system and the changed chunks would be processed by combiners.
Incoop and our framework implement materialization from different aspects. In our framework, we target at identify the computations sharing same input; while Incoop targets at identify the inputs sharing same computations. It would be good to include both in the framework. We discussed the possibilities and challenges of implementing incremental computation in Section 7.2.

6.2 Optimization based on Dryad/DryadLINQ

Dryad/DryadLINQ [28, 44] is a parallel processing system developed by Microsoft. Dryad [28] is a distributed execution engine to process data-parallel applications and DryadLINQ [44] is a high-level language. A Dryad application is expressed as a directed acyclic graph where each vertex is a program and each edge represent a data channel. DryadLINQ would translate the queries into plans that can be executed by Dryad. It processes a query with the following steps: (1)Translate the query into a logical plan;(2)Transform logical plan to physical plan; (3)Encapsulate physical plan to Dryad execution graph.
Microsoft has conducted a series of research on query optimization. In [26], two kinds of redundant computations are identified: input data scans and common sub-query computations. Redundant scanning contributes to around 33% of the total I/O; while 30% of the sub-queries are detected to have a match(have same input and common computation). The computation redundancy are also detected in [23, 27, 39]. To solve this problem, Microsoft developed the DryadInc system [39], the Nectar system [23] and the Comet System [27] for query optimization. They are all built upon the Dryad/DryadLINQ System. And they assume the file system is append-only.
DryadInc [39] is a **non-concurrent sharing** framework for incremental computation. It consists of two components: the rerun logic and the cache server. The rerun logic plays the role of materialize, match and rewrite. The cache server provides an interface to read and write the execution statistics.
The rerun logic uses two heuristics to do incremental computation: identical computation(IDE) and mergeable computation(MER). Both IDE and MER use fingerprints(the MD5 checksum of content) to identify programs.
IDE: It saves the mapping from computation fingerprints to computation results in the cache server, and uses them to answer queries. The problem is sub-computation selection.
MER: For a function F and input I, it caches the result of F(I). If delta (Δ) is appended to I, to compute F(I+Δ), MER identifies Δ and detects whether F(I) exists in the cache server. If F(I) exists, it computes F(Δ) and merges F(I) with F(Δ), which is the result for F(I+Δ). The difficulty of MER lies in the merge function.
DryadInc doesn't present the implementation details. Rather, it gives two directions for materialization: with the same input, computations can be identified; with the same computations, input can be identified. As we discussed in Section 6.1, our framework implemented the first option, and Incoop implemented the second option. However, MER and Incoop are different since they work at different layers. MER deals with a plan while Incoop works on top of the execution engine.
Nectar [23] is a **non-concurrent sharing** framework. It deals with both computation efficiency problem and storage utilization problem.
Computation management: It uses a program rewriter and a cache server to avoid redundant computations, similar to DryadInc.
Data Management: It uses a program store and a data store to manage data. The

program store contains all the executed programs and the data store contains all the computation results. When the Garbage Collector detects that there are computation results that have not been accessed for a long time, it will replace these computation results with the programs that create them.
Nectar is similar to our framework. There are two major differences.
First, the sub-job selection is based on the history request statistics and real-time execution statics of the sub-job. If they exceed the predefined thresholds, the sub-job would be materialized.
Second, when there are multiple matches, it uses cost estimator to select the best alternative. However, it is not very clear how the cost estimator works.
Comet [27] enables **both concurrent sharing and non-concurrent sharing**. It performs three kinds of optimization: query normalization, logical optimization and physical optimization.
In query normalization phase, the single query(S-query) is split into sub-queries(SS-query) which can be reused by other SS-queries. Besides, each SS-query is tagged with a timestamp, so that the system can identify a set of SS-queries from different S-queries that can be executed together. They are constructed as a jumbo-query.
In logical optimization phase, common sub-queries in a jumbo query are detected and they would be executed for only once. To expose more common expressions, operators can be reordered. The results of jumbo queries can also be reused by other jumbo-queries. Besides, Comet considers co-location when replicate materialized results, in order to reduce network traffic.
In physical optimization phase, shared scan and shared shuffling are performed. Shared scan can reduce local data transfer and shared shuffling can reduce network data transfer. In the current design of DryadLINQ, shared scan is enabled by introducing a Dryad vertex, which would bring I/O cost. Thus, shared scan is not always beneficial.
Comet is the most comprehensive framework. It includes almost all the ideas of reusing in the other works. However it doesn't show the implementation in detail.

6.3 Comparison between Optimization in Relational Databases and Big Data area

We showed materialized view technique in Relational Databases in Chapter 3. In this section we compare the optimization techniques in Relational Databases and in Big Data area.

6.3.1 Non-concurrent optimization

Both systems materialize computation results and use them to answer queries.
Relational Databases not only select candidate views from derived relations, but also generate candidate views. Besides, the View Selection is based on cost-benefit model. In Big Data area, candidates are only selected from the outputs of sub-jobs. And it is almost impossible to use a reliable cost model to evaluate query plans [26]. This is because the system often knows little about the data being processed; a query could use custom functions and user defined functions with unknown performance characteristics; the query could be complicated and the computation would involve multiple distributed steps.
Both systems have the match phase and the rewrite phase. In Relational Databases, match means the view contains no less information than needed by the query; rewrite means generate a new expression which is equivalent to the original query. This can be done by adding compensate predicates to prune excess data. All the matched views are used to generate new expressions and the best one is selected by the query optimizer. In Big Data area, currently match and rewrite can only be performed when the materialized results contains exactly the same information as needed by the query or part of the query.

6.3.2 Concurrent Optimization

Both systems use multi-query optimization technique. It takes advantage of common sub-queries. It detects queries sharing common expressions and merge them into one optimized global plan, so that the common parts would be executed for only once.
However Concurrent Sharing in Big Data environment is more complex. First, shared computation is not always beneficial. It might cause extra I/O cost. For instance, Pig would insert a 'Split' operator after the shared computation and the outputs would be dumped to disk temporarily. Second, optimizations can be performed at more levels, for example: shared scan and shared shuffle. They can reduce the I/O cost, but they might also introduce I/O overhead. In Comet [27], shared scan would cause extra I/O cost.
Thus, concurrent sharing optimization in Big Data environment is more complex and requires more studies.

7

Chapter 7

Conclusions

Parallel data processing platforms are increasingly popular in the past few years. They have significantly improved the throughput of Big Data analysis task by employing more computation resources. Computation efficiency and resource utilization muse be optimized to save both time and money. A noticeable problem in all data analysis tasks is computation redundancy. It comes along with the workloads, regardless of the data processing tools. The goal of our work is to build a framework to avoid computation redundancy. In Relational Databases, one of the most important techniques to solve this problem is the materialized view technique. We studied this technique and applied the idea of materialization in the Big Data environment, that is, store the intermediate and final computation results and use them to answer queries.
We implemented a framework which can reuse previous computation results on top of Pig/Hadoop. It consists of four components: the Plan Matcher and Rewriter, Logical Optimizer, MySQL cluster and Garbage Collector. The Plan Matcher and Rewriter is used to select results to materialize, store execution statistics in MySQL cluster and rewrite queries with matched sub-computations. It is the most essential component of the framework. It resembles the View Selection and View Exploitation phases in Relational Databases, but the implementation is quite different due to the difference of execution models. The Logical Optimizer optimizes the rewritten plan and it is necessary for Apache Pig. The Garbage Collector manages the materialized results. It detects and deletes obsolete data from both MySQL cluster and the file system. It is much simpler comparing to View Maintenance in Relational Databases, due to the append-only property of the data storage system.
We used the TPC-H Benchmark to evaluate the framework. The results shows that our framework has significantly reduced query execution time by reusing previous results. By reusing sub-computations, the query execution time is reduced by 30% to 75%; while it only takes around 40 seconds when reuse whole computations, regardless of the query. Besides, it also showed that the overhead of materialization is small, around 25% of the execution time without materialization. And the non-I/O overheads are negligible comparing with I/O overhead. The Match and Rewrite algorithm is efficient and takes less than 200 milliseconds; MySQL cluster access

and read/write take the longest time, but just around 6 seconds in the worse case. Thus, it is important to have a good sub-job selection strategy to reduce the I/O overhead.

7.1 Open Issues

There are some issues which remain unexplored due to time restrictions.
First, as we illustrated in Section 2.2 , 'Split' is the only operator which is allowed to have multiple outputs. This operator would dump its output to a temporary path on the disk, before continuing the jobs after it. After we insert a 'Store' operator, Pig would insert an 'Split' operator before it. We are not clear how much I/O overhead would be produced by these two operators. As long as we know, one possibility is that Pig would copy the result of 'Split' in the temporary path to the output path of 'Store', and delete this temporary path after completing the query execution. We assume this is the case, if Pig can do a 'move' operation instead of 'copy', it would be much faster.
Second, we think it is not necessary to give intermediate results a high replication factor. They are usually much larger compared to the final results, even after careful selection. Thus, storing them on disk with a high replication factor would consume a lot of disk and time. Though the useless outputs would be finally deleted by the Garbage Collector, they consume the disk so fast that there is a risk to use up the disk quickly. We have actually met this problem. Besides, it takes a lot of time to replicate a large dataset in distributed file system. If we can reduce the replication factor of the outputs' paths before materialization, without changing the default replication factor, the overhead should be reduced. Currently, changing the replication factor for a specific path only has an effect when the path already exists.

7.2 Future Work

The optimization for parallel processing platforms is not limited to this research.
First, as mentioned in Section 4.2.3, it is nice to figure out as more reuse opportunities for logical plans with multiple outputs.
Second, it is good to implement incremental computation. Since the identifications of the input and the execution plan are separated, the main challenge is to detect the change of input. The two problems to solve are how to detect the changed part and unchanged part of the input and how to merge the computation results of the two parts.
Third, the framework is build upon logical layer and it is possible to extend this work to other parallel dataflow systems with logical layer, such as Hive. For Plan Mather and Rewriter, there are three issues: intermediate results selection, match,

and rewrite. First is the intermediate results selection. The general selection criteria discussed in Section 4.2.2 also fits the other systems and the operators supported by the system must be learned. Second is how to do match. The algorithm to compute fingerprint of Pig's query can also be used for logical plans of other systems, as long as they are directed acyclic graphs. Third is the problem of rewrite. There are two issues. One is that the plan object might have a different construction. It needs to be studied so that we can operate on the plan with right methods. The other one is to learn how the system read input and write output, then we know how to read and write intermediate results. As for the optimization, we need to know how the logical optimizer of the system works, but this is not a big problem. If optimization is needed, we can use the system's optimizer directly, since optimization takes less than 10 milliseconds as shown by our results.

Bibliography

[1] Apache Pig. `http://pig.apache.org/`. Last accessed: June 25,2013.

[2] Hadoop. `http://hadoop.apache.org/docs/r1.0.4/index.html`. Last accessed: June 25,2013.

[3] Hashcode. `http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#hashCode()`. Last accessed: June 25,2013.

[4] HDFS Architecture Guide. `http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html`. Last accessed: June 25, 2013.

[5] MapReduce Tutorial. `http://hadoop.apache.org/docs/r1.0.4/mapred_tutorial.html`. Last accessed: June 25,2013.

[6] MySQL Cluster. `http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster.html`. Last accessed: June 25,2013.

[7] Mysql Cluster Architecutre. `http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster-overview.html`. Last accessed: June 25,2013.

[8] MySQL Cluster Connector for Java. `http://dev.mysql.com/doc/ndbapi/en/mccj.html`. Last accessed: June 25,2013.

[9] MySQL Cluster Installation. `http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-installation.html/`. Last accessed: June 25,2013.

[10] Pig Latin Basics. `http://pig.apache.org/docs/r0.10.0/basic.html`. Last accessed: June 25,2013.

[11] TPC-H Benchmark. `http://www.tpc.org/tpch/`. Last accessed: June 25,2013.

[12] User Defined Functions. `http://pig.apache.org/docs/r0.10.0/udf.html`. Last accessed: June 25,2013.

[13] Parag Agrawal, Daniel Kifer, and Christopher Olston. Scheduling shared scans of large data files. *Proceedings of the VLDB Endowment*, 1(1):958–969, 2008.

[14] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 496–505, 2000.

[15] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

[16] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A Acar, and Rafael Pasquin. Incoop: MapReduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 7. ACM, 2011.

[17] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[19] Iman Elghandour and Ashraf Aboulnaga. Restore:Reusing results of mapreduce jobs. *Proceedings of the VLDB Endowment*, 5(6):586–597, 2012.

[20] Alan Gates. *Programming Pig*. O'Reilly Media, 2011.

[21] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[22] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: a practical, scalable solution. In *ACM SIGMOD Record*, volume 30, pages 331–342. ACM, 2001.

[23] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–8. USENIX Association, 2010.

[24] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin*, 18(2):3–18, 1995.

[25] Michael Hausenblas and Jacques Nadeau. Apache Drill: Interactive ad-hoc analysis at scale. *Big Data*, 2013.

[26] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Wei Lin, Bing Su, Hongyi Wang, and Lidong Zhou. Wave computing in the cloud. In *Proceedings of the 12th conference on Hot topics in operating systems*, pages 5–5. USENIX Association, 2009.

[27] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 63–74. ACM, 2010.

[28] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[29] Joey Jablonski. Introduction to Hadoop. `http://i.dell.com/sites/content/business/solutions/whitepapers/en/Documents/hadoop-introduction.pdf`, 2011.

[30] Vasiliki Kalavri. Integrating Pig and Stratosphere. Master's thesis, KTH, 2012.

[31] Alon Levy. Answering queries using views:A survey. *VLDB Journal*, 10(4):270–294, 2001.

[32] Jie Li, Koichi Ishida, Muzhi Zhao, Ralf Diestelkaemper, Xuan Wang, and Yin Lin. Running TPC-H on Pig.

[33] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *Proceedings of the VLDB Endowment*, 5(11):1196–1207, 2012.

[34] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[35] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. MRShare: sharing across multiple queries in mapreduce. *Proceedings of the VLDB Endowment*, 3(1-2):494–505, 2010.

[36] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, pages 267–273, 2008.

[37] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[38] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data:Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[39] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. DryadInc: Reusing work in large-scale computations. In *USENIX workshop on Hot Topics in Cloud Computing*, 2009.

[40] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[41] Standard Specification and Transaction Processing Performance Council TPC. TPC Benchmark TM H. Last accessed: June 25,2013.

[42] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[43] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 136–145. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1997.

[44] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14, 2008.

[45] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 533–544. ACM, 2007.

I hereby certify that I have written this thesis independently and have only used the specified sources and resources indicated in the bibliography.

Hui Shang, 04. July 2013

. . . .

My Name

TRITA-ICT-EX-2013:186