Imperial College London

Department of Computing

Individual Project: Final Report

Query Analyzer for Apache Pig

Author:
Robert Yau Zhou
00734205
(robert.zhou12@imperial.ac.uk)

Supervisor:
Dr Peter McBrien
Second Marker:
Dr Thomas Heinis

June 13, 2016

Abstract

Database Query Optimisation has been researched extensively, commercialised Relational Database Management Systems uses Cost Based Query Optimisation frequently. At the same time, Big Data tools such as Apache Pig do not perform these kind of Cost Based Optimisation, and it is up to the users to decide the best way their queries are executed. Since both Pig are SQL are Relational Algebra like, it is interesting to know whether the approaches in traditional Query Optimisation can be used with Pig well. In this project, I have explored the cost model, one important component of Cost Based Query Optimisation, that assign a cost to each alternative query plan. Using the amount of data transferred within the network required to execute a Pig query as the cost, I have designed and developed two cost models using existing theories in Query Optimisation as well my own ideas. One model estimates cost using the sizes of tables in dataset, and the other model would make use of distributions of individual attributes of tables in dataset. Both models are performing well in terms of ranking queries based on their costs. This project can serve as a good foundation of further researches in Pig query analysis and optimisation.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr Peter McBrien, for his patience, guidance and valuable suggestions throughout the project. His extensive knowledge in the subject of database as well as his enthusiasm made the completion of this project a joyful experience and this project cannot be completed without his help.

I would also like to thank my second marker, Dr Thomas Heinis, for his useful advice and feedback on my project.

I would also like to thank my friend, Bryan Liu, for his help and advice throughout this project.

Additionally, I would like to thank all my family members for their love and support throughout my life, including the duration of this project when love and support are needed.

Contents

Chapter 1

Introduction

Big Data is the word that you will hear everywhere nowadays, and everyone seems eager to make use of it one way or another. Retail sector companies analyse customer behaviours from the enormous amount of data they have collected to improve their business strategies, scientists and engineers at CERN consume 30 petabyte of data annually [1], about 3600 GB per hour, and even politicians uses Big Data to win themselves elections [2].

With the sheer amount of data that is there to be cracked open, traditional Relational Database Management Systems (RDBMS) are not up to the job in most cases. The scalability of RDBMS suffers from the fact that they often offer ACID (Atomicity, Consistency, Isolation, Durability) properties, and large amount of data would considerably worsen the performance of RDBMS, also the fact that data has to be loaded into the database, and allow all processing to be done before analysis can be performed[3] also adds unnecessary overheads. Furthermore, ACID properties are sometime not required, such as analyses of website logs. Usually the user wants to quickly go through all of the logs and see if anything interesting can be found, such as correlations in customer behaviours. In these applications, the losses of some files are tolerable.

Google released their paper on Map Reduce [4] a decade ago, and this provide a scalable model which utilises large clusters of commodity hardware to process a large amount of data. This model take cares of the parallelisation, fault-tolerance, data distribution and load balancing [4] and all the programmers need to do is to convert their program into map jobs and reduce jobs, and therefore made analyses of large amount of data easier. Hadoop is an widely used open source library which employs a similar approach to Google's Map Reduce which allows scalable distributed processing for large dataset and some notable users include Adobe, eBay, IBM and Yahoo! [5].

Hadoop and MapReduce solve the problem of difficulties on processing large amount of data in a distributed environment, but certain features are missing, two of them are particularly relevant to the thesis of this project namely the use of relational-algebra-like language and automatic query optimisations. When using Hadoop users were required to provide map functions and reduce functions in Java which would be submitted to the clusters and executed, frequently used database operation such as projection, and selection must to be implemented in Java. This adds some unnecessary redundancy and repetitiveness when working on Hadoop as programmers may constantly have to write map tasks that are only two lines to achieve projections. Programmers also loss the luxury of only having to consider the functional description of their code and let the query optimiser to come up with a query plan as they do when working with SQL in RDBMS, this mean efforts must be put in to make sure their query are efficiently and they do not waste valuable network resources by reading in unnecessary fields or passing unused rows around the cluster.

Pig is a platform which allows programmer to write relational algebra with Pig Latin and execute these programs by compiling them to equivalent Map Reduce jobs in Hadoop [6]. Pig solves the problems we described partially. The fact that Pig Latin implements a subset of relational algebra and is therefore highly declarative means programmer can write code similar to SQL and enjoy the scalability of Hadoop at the same time. What Pig does not do is query optimisation similar to the ones offered by most RDBMS, while Pig does perform some optimisations, it is nowhere near as sophisticated as what RDBMS can offer, and therefore much work is place on the shoulders of programmers.

1.1 Objectives

The aim of this project is to explore the possibilities query optimisations in Pig. Since this is a relatively new area, there are many possibilities as well as difficulties. There are many problems that needs to be worked on and perfecting on all aspect of query optimisation is well beyond the scope of the project.

There are two important parts of query optimisations, an accurate cost model and good query plan enumeration algorithms and both topics are difficult and can be studied in depth, and choosing a correct scope for the project is therefore important.

One important factor that we must take into consideration is the fact that a good cost model must come before the actual optimisation algorithms. An algorithm can be as good as you want, but if the things that you are optimising actually bears no relation to the real world application, the optimisation will not work well. For example, for a RDBMS, usually the system would try to minimise the usage of I/O and CPU time, and there are usually a cost model which will estimate the cost of query plans, assuming calculating this estimated cost is cheaper than running the queries, it would make sense to find a query plan with the lowest estimated cost. The problem would arise when the estimation is significantly inaccurate such that it bear little resemblance to how costly the queries are in the real world, we would be wasting our time trying to find the query with the lowest estimated cost. Putting it in the context of this project, we need to have an acceptable cost model before we explore algorithms that optimise a query agains the costs in our model. Therefore the main objective would be to design a good cost model for Pig queries that serve as the basis of any cost based optimisation algorithms.

In this project I would focus on the development of some working models for estimating cost of executing Pig queries, based on existing theories in query optimisations as well as my own solutions to problems that were not present in traditional RDBMS. I would try two different approaches of estimating the cost of Pig queries, one that only take the sizes of the input tables into account, and another that also considers the distributions of values in those tables, and assess how well they work.

To assess how well the models work, I would measure how well they can rank different queries in terms of their cost instead of how accurate and precise the models can calculate the exact resources needed to complete those queries. This is because if the model is adapted to be used in cost base optimisation, we want the model to help us pick the least costly query out of all of the alternatives, and as long as the relative ranking is correct, the model is working well for the intended purpose.

Chapter 2

Background

In this chapter, we are going to have an overview of background knowledge that is important for this project. During the discussions of these background knowledge, I would try to relate these to the project if possible, talk about what each point meant from the point of developing a good cost model, and what we should pay attention to during the development.

2.1 Hadoop

2.1.1 MapReduce

MapReduce is a programming model for processing and generating large dataset [4]. In MapReduce, programmers specify their program in a map function, which process a set of key/value pair and generate another set of intermediate key/value pair, and a reduce function, which merge all the intermediate values with the same keys.

One example is to find the most read news article of a website, and the map reduce can be structured as

Input: logs of web page requests
Map: url $\rightarrow$ list of (url, 1)
Reduce: (url, [int]) $\rightarrow$ (url, count)

What this says is that in the map phase, each entry of logs will be processed, and produce a tuple of the url and the number `1`, meaning it has been accessed once. After the map phase, all the `1`'s of an url will be put into a list, and in the reduce phase, the list will be counted so that we know how many times a url has been visited.

This simple paradigm allows programs to be parallelised automatically and be executed on a large cluster of commodity machines. Low level details such as data partitioning, scheduling, communication and failure handling are taken care of by the run-time of the library [4].

While these setups might incur some overhead, the main advantage of this programming model is scalability. The same task might have better speed running on a single machine using code specific for the task, but when the data gets large and many machines is needed to perform these computation in parallel, a programmer will then have to spend many time setting up the machines using libraries such as Condor[7] and manually split up the tasks into different jobs. Which is time consuming for the programmer, compare to only having to worry about the functional side of the program and all parallelisation problems to be taken care of by MapReduce.

2.1.2 Hadoop's MapReduce

There can be many different implementation of MapReduce, and the right choice depend on the environment [4]. Since we are going to study how Pig query to be optimised, it is important to understand the design of MapReduce of Hadoop, so that we can know what can or cannot be optimised under the architecture and how.

The MapReduce framework has a Master/Slave architecture where there is a ResourceManager on a node acting as the master and a NodeManager on each slave nodes [8]. The Resource Manage is responsible for allocating resources monitoring tasks, scheduling tasks and providing status to the client. There is also an ApplicationMaster for each job, and it asks resources from the ResourceManager and work with the NodeManager to execute the tasks.

During a MapReduce job, the input data is usually split into independent partition, determined by the block size, each partition is then processed by a map task running on a cluster node independent and parallel to other map tasks [9]. The framework would take care of sheduling, monitoring, detecting failure tasks and re-executes the failed tasks. After the map tasks are completed, shuffles" are performed where relevant data are fetched from the map nodes, then the data are sorted so that each output from the mappers with the same key are grouped together. Then the reducers provided by the user would be executed.

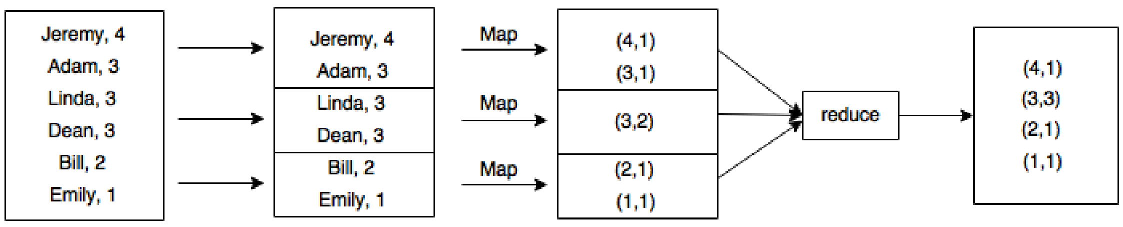

Figure 2.1: MapReduce

In figure 2.1, the input file is a list of students and the year group they belong to. We are trying to find out how many students there are per year. As we can see, the input is splited into some partitions, and each would be processed by a map task. The map tasks procude tuples with number of students they have seen each year, and the reducer combines the results and output the list of number students at each year.

To achieve smaller usage of network, and take advantage of locality, tasks are usually scheduled on nodes where data are stored, and only schedule them to nodes where data have to be copied from somewhere else if all nodes are unavailable. This is achieved by making the HDFS and MapReduce run on the same nodes, and this minimise the use of the scarce network bandwidth which is critical for the performance.

2.1.3 HDFS

The Hadoop Distributed Filesystem is a distributed file system that works with Hadoop [10]. It is designed to run on commodity hardware. The significance of HDFS is that it is highly fault-tolerant, high throughput, scalable for large files and built on the write-once-read-many access model for files.

The following assumptions are made when designing HDFS [10]:

2.1.3.1 Hardware Failure:

Hadoop is design to be used to distributed environment with commodity hardwares where failures are highly likely, if this is not automatically dealt with, a file system with thousands of machine will probably not able to run at all. This is due to the increase probability of failure of at least one machine grows quickly when the number of machine increases. Assuming failures happens in one particular day for one machine is one in thousand and each machine is independent to another, with merely 100 machines the probability of at least one failure occurred in one day is about 10

2.1.3.2 Large Data Sets:

HDFS is designed to support large data sets, which means it must be able to store files that are terabytes in size.

2.1.3.3 Streaming Data Access:

Since the Hadoop is a tool to aid data analysis, and interactive use by users are not expected but the assumed pattern is write-once-read-many. After the dataset is loaded many analysis will be performed on them with little modifications, therefore the system would be made to have a high throughout instead of low latency, and therefore not all POSIX requirements are met to allow steaming access to the data in the file system.

2.1.3.4 Design of HDFS

The HDFS employs a mater/slave architecture where a NameNode is the master server which manage file system namespace and regulate file access by clients, it also stores all the metadata and the the file system tree. There would also be many DataNodes, the slaves, which manage storage in each node, and store and retrieved data on request of the NameNode or Clients, and keep the NameNode updated on the blocks that they are storing.

A block in HDFS is similar to a block in normal file systems, acting as unit of storages. The difference is that a block is typically 128 MB in HDFS oppose to a few KB in the traditional file systems. This brings the benefit of less seeks would be performed and more time could be used on file transfer. Administrators should find a balance in block size as a block size too big mean too few takes would be performed in parallel, because the Map phases usually operates on single blocks.

HDFS achieve availability and fault-tolerance by data replication, and each file can be specified to have a certain number of replicas. When the number is three, HDFS will place two at the same rack and one at another rack, which ensure both availability and network traffic efficiency, as nodes in the same rack is likely to have a better connection between them.

You might have realised there is something wrong with the design of HDFS which might have compromised the availability. The fact that there is only one master node, the NameNode, means if it fails the system will fail, this is a single point of failure. There are a few ways to tackle this issue, one is to have a "secondary NameNode", which, despite the name, is not a name node, but a node which record snap shots of the NameNode, and in the event of failure human interaction is needed and data since last checkpoint will be lost. Another way is to have a standby node which will can be automatically substituted in the event of failure, however this require extra setups.

2.2 Measuring Cost

Now that we have some understanding of how Hadoop and MapReduce works, we should decide on what constitute "cost" in this project. In other words, what should we use to represent the cost of each query? In centralised RDBMS, we usually use the number of pages on the disk were read and written[11], and by optimising the I/O, we could optimise the execution time, but is this what we should be adopting in this case?

Since Pig operates on Hadoop, a library for distributed data analytic, we would typically have cluster set up in a server room with a number of machines, and to communicate, each node needs a connection to each other. The implication is that we can easily buy more machines, so we can easily have more CPU time, more data being read at the same time, but the cost of upgrading the connections to allow each node to communicate with each other directly growth in n^2. Because for n nodes we need $(n-1)+(n-2)+...+1$ connections. In most setting, nodes do not connect to each other, and we would expect inter-rack communication to be slow as each node in a rack shares the switch.

This means network is the most scarce resources, and our focus should be on reducing the amount of data being sent through the network. In the context of Pig, we aim at reducing the data sent from map nodes to reduce nodes, as Hadoop would they to ensure as little data is transferred in map phase as possible[9].

Another important feature of Hadoop is that very often users are not expecting a quick result as they are analysing a huge amount of data, and often many jobs are run at the same time or the machines are shared in a certain way. This means reducing the network usage of a Pig job would have a good overall impact on other tasks in the system.

When it comes to optimising Pig queries, we should optimise them so that as little network is used as possible and therefore we shall use network usage as our sole performance indicator. Since each map jobs are usually performed where the data is stored, we would consider the data sent from map nodes to reduce nodes as the cost of a query.

2.3 Pig

Pig is a platform which for analysing large dataset, it uses the language Pig Latin to express the data analysis programs, and convert the program to Hadoop's MapReduce jobs, and execute on Hadoop architecture [6].

Pig Latin is a high level language which includes many traditional data operations such as join, sort and filter, as well as allow user to use their own user defined functions (UDFs) for processing data. MapReduce, despite having some great advantages we have discussed in previous sections, has its own short coming. In addition to asking programmer to write all of the low level operations, another big problem is that join operation is very difficult to perform, but joins are used commonly when different data sets need to be combined.

It would be a good time to have a look at what Pig Latin looks like. Lets say if you are going through your database, and you want to find out about the toys sales revenue from each country, a typical SQL query would look like

```
SELECT country, SUM(price)
FROM sales_log
WHERE category = "toy"
GROUP BY country
```

An equivalent Pig Latin program would look like

```
toy_sales = FILTER sales_log BY category MATCHES toy
groups = GROUP toy_sales BY country
output = FOREACH groups GENERATE country, SUM(toy_sales.price)
```

As long as you have some experience with SQL, it is easy to tell that the FILTER and FOREACH in Pig Latin is equivalent to WHERE and SELECT in SQL respectively.

2.3.1 Pig Latin

We will now discuss the Pig Language [12], this part of the report would focus the discussion on how each operators in Pig would be used in MapReduce and its costs as our goal is to optimise the Pig Latin query and not to teach people how to write Pig programs.

2.3.1.1 LOAD

The LOAD operator load data from HDFS into the memory to be used by other part of the program, you can used using to specify the loading function to load data in different format and as to specify the schemas. Since data must be loaded in order to be processed, not much optimisation can be done here. Some optimisation such as not loading unused columns are already implemented by Pig, and therefore we will not focus on the LOAD operator.

2.3.1.2 STORE

The STORE operator allows user to save the processed data to HDFS. User can use using to specify the function and into to specify the path. Since the user should have the final say on what to be written on the disk, and we will not look into how to optimise STORE.

2.3.1.3 DUMP

DUMP is similar to STORE but it print out result to the screen instead of writing them to files. This can be useful for some ad hoc queries. Just like STORE we will not consider DUMP for optimisation.

2.3.1.4 FOREACH GENERATE

```
FOREACH student GENERATE id, mark;
FOREACH trades GENERATE high - low as range;
```

FOREACH is similar to SELECT in SQL and selection in Relational Algebra. It takes a data set, go through each record, and apply the expressions to generate new records. values in each column can be referred by the name or by position such as $1. It also works with complex data types such as map and bags, and the semantic would change when processing a bag.

Usually FOREACH would reduce the amount of data, and therefore we want this to be done as early as possible, so that less data will be sent through the network.

2.3.1.5 FILTER BY

```
FILTER student BY score >= 70;
```

`FILTER` is similar to `WHERE` in `SQL` and selection in Relational Algebra. It takes a data set, and apply the logical predicate after `BY` to every record. The example above returns all student scored at least 70. If the predicate is evaluated to true for a particular record, then the record is will be passed down the pipeline. It has the same semantic when dealing with null, and is (not) null should be used.

Just like `FOREACH`, `FILTER` reduces the amount of data and therefore should be executed before data is sent through the network.

In the cost model, `FILTER` is particularly important as it determine the tables that subsequent operators have to process, and whether we can correctly estimate how many rows in the data would pass the condition would directly affect the accuracy of the model.

To demonstrate this, we can look at an example that shows that depending on the conditions written in the queries or the selectivity of the attributes, the resulting tables' sizes can vary significantly.

Lets look at the three different queries below:

```
men = FILTER student BY gender == 'm';
adult = FILTER student BY age >= 18;
awarded = FILTER student BY score >= 95;
```

For the `men` table, assuming the university has equal ratio of male and female students (which is not true for Imperial!), we would expect it to have 50% of the rows of the `student` table. On the other hand, assuming the university consist of mainly adult students, which is a fair assumption to make, we would expect the `adult` table to have more than 90% of rows from the `student` table. Since not many student scores over 95, we would expect the `awarded` table to contain only 1% of the rows from `student`.

if these table are passed down to perform further analysis, we would expect to have a big difference in cost, and being able to distinguish them is very important if I were to develop an accurate model, the tasks gets more difficult when the conditions of `FILTER` becomes more complex. How to predict the `FILTER` well would be discussed in further chapters including the use of selectivity, cardinality and histograms.

2.3.1.6 GROUP, COGROUP, CUBE

```
years = GROUP student BY year;
best = FOREACH years GENERATE group, MAX(score)
```

A `GROUP` statement, being identical to `COGROUP`, in Pig is similar to `GROUP BY` in `SQL` but is also, at the same time, fundamentally different. The example above gives you the best score from student in each year group, but `GROUP BY` in `SQL` must immediately be consumed by aggregate functions, like what we have done in the example, but this is not the case in Pig. What Pig returns after GROUP BY is a set of keys and a set of bags associate with the each key. Since Pig allow complex type such as bag to be a value in a record, there is no difference between the relation "years" above and others. GROUP ALL is a special case where the produce will be one record with key all and the value as a bag containing all records, this forces things to be done sequentially.

A CUBE is similar to COGROUP, only that the group keys are all combinations of the group attributes.

```
years = CUBE student BY (year, age);
```

For example, in the above query, table student will be grouped on (year, age), (year,), (, age) and (,).

In MapReduce, a GROUP BY is usually done in the Reduce phase, which means if the pipeline is already in Reduce phase, GROUP BY will pass through a new Mao, Shuffle and Reduce phases. This is due to the fact that grouping collects all records with the same key, which is exactly what you get at the start of a reduce phase.

Since you can have skewness in data sets, that is amount of values are not uniformly distributed to all the keys, and some reduce job might take much longer to finish than others. In some cases this can be helped by using combiners, which can be understood as do part of the reduce phase in before data is passed to the reduce nodes.

Another problem would be the key chosen to group on. If there are only a few possible value, then the number of rows after grouping is likely to stay the same when data increase in size, in the example above, a university typically have several year groups, no matter how many students we have, we will have still have that many rows. If, however, we are grouping on url address for a big online forum where new pages are generated every seconds, then the number of rows is likely to grow with the number of records. The implication of Hadoop is that if the number of rows is small, we would need to have some Reduce tasks that handles a huge amount of data for a single key and if the number of rows is too large, then it will be easier to balance the load with different reduce nodes.

If we look at the example of counting url clicks, the map phase produce many (url, 1) and the at reduce phase the input would be (url, [1,1,1,,1]). This can be improved if the reduced can also be performed at each map node, so the map node would produce many (url, x) where x is the number of (url, 1) seen in data provided to this map node, and then reduce nodes would have input which looks like (url, [109, 75,, 12]).

Pigs operator and Pigs user defined functions (UDFs) would use combiner when possible, and the users UDFs can use combiner by implementing the Algebraic Interface.

In terms of estimating the cost of GROUP, an interesting observation is that, intuitively, the amount of data being transferred on a GROUP statement should be easily calculated. Regardless of what you are grouping on, the amount of data being sent through the network should be roughly the same, i.e. the whole table. The only difference would be the overhead and the values of attributes that were being grouped on. What is important is that what comes out of these operators would make a huge difference to the following statements.

If the GROUP was performed on the gender attributes, we are usually going to have two or three rows as the result, and if a aggregating function such as COUNT or MAX is used, we should expect to have a very small table, and hence any future cost involving this table would be very low. but if we GROUP on the id attribute, we should expect to have the same number of rows as the original tables and any resulting tables from aggregating functions is likely to be very large.

Another complication for designing a cost model in Pig is that results of `GROUP` in Pig is treated the same as any other table whereas in SQL the relation must be preceded by an aggregation function such as SUM. This adds another layer of complication because if the sum table is not proceeded by aggregating functions, then it will be more difficult to estimate the cost of following operations as we would end up working with a bag data type.

2.3.1.7 ORDER BY

```
ord = ORDER student BY score;
```

ORDER BY is the same as ORDER BY in SQL which sorts the data according to columns specified after gym and ASC and DESC can be specified at the end to choose ascending order or descending order respectively.

Similar to GROUP BY, ORDER BY happens in the reduce phase, but the skewness issue is automatically optimised by Pig as it would send a small sample through to test the ORDER statement, and balance the load on each reduce nodes.

As ORDER BY always takes place in reduce phases, we also want to make sure that as little data is sorted as possible, considerations similar to GROUP by can be taken on order by, so that we project/select early and maybe combine a few ORDER BY together.

2.3.1.8 DISTINCT

DISTINCT remove duplicated records, and contrary to SQL, DISTINCT in Pig can only be applied to records, not individual fields. It forces a join phase and combiner is automatically used. Similar to ORDER BY and GROUP BY, we should make sure as little data is transferred as possible.

2.3.1.9 JOIN

```
j = JOIN student BY id, locker BY student_id;
```

JOIN is the same as JOIN in SQL, where columns from data sets are indicated and records with the same values in those columns will be joined together. Pig also support outer joins (left, right and full).

For left join, Pig must have schema for the right side to fill in nulls when no records are matched, the same with right and full join which schema for left side and both sides are needed respectively. Records with null values for the keys will not match with anything.

In MapReduce, the map phase would be used to mark each record which input it same from, and the join keys will become the shuffle key (where data is grouped). Cross product between the records from both inputs with the same key is produced in the reduce phase. Pig also make sure data on the left side arrive first and they would be kept in memory, then each right record is crossed with all of the left records in memory as they arrive.

To improve the performance of joins, one should put the record with more records per value to the right side and thus reduce memory usage and improve performance. Just like all the operators that forces a reduce phase, the timing and amount of data pass through to JOIN is important to be considered when doing optimisation.

Pig also provide many other implementation of JOIN, and we should access the which one to be the best at which situation and recommend to the user when possible.

Since Join is a very important tool in data analysis, and a large part of Cost base Optimisation is Join Ordering, it is very important to accurately calculate the sizes of output of joins, and accurately calculate it's sizes. The distribution of join keys of the tables being joined is very important to accurately determine the size of the table after the join. If both table has only one value of the join key, then we are basically performing a Cartesian product of the two table, which is very costly, on the other hand, if table `X` and `Y` are joining on attribute `a`, and we know that `a` is a foreign key to table `X`, then the output would have the same number of records as `Y`.

The above two are the cases where we know exactly how many records we are going to have, and in many cases we are going to have something in between, in this case we would have to estimate what the output would look like, often with the help of histograms.

We also need to work out the cost of each join operation accurately, and Pig provides a few different algorithms for `JOIN`. The default behaviour of `JOIN` in Pig is that the map phase to annotate each record with it's input, then it shuffles on the join key, and a cross product is made on records with the same values on the join keys from all of the tables in the join [13]. Apart from the default one, one can use `replicated` where a small table is sent to all machines in the Map phase and therefore no Reduce phase is needed, in this case the cost would be much smaller as no Reduce phase is performed, and loading of the small tabel can be taken care of by optimisations on the Hadoop platform such as loading the replicated input into Hadoops distributed cache [13] or maybe performing many different Map tasks on the same physical machines. if we have seen replicated and have verified the right table is small enough, our model might potentially regard the cost of this operation as 0 as we are only considering the bandwidth cost of reduce phases.

There are many other join algorithms such as `skewed` and `merge` that accommodates to different situations, a good model need to be able to clearly distinguish and recognise the cost of each algorithms and therefore evaluate different join plans. Skewed join samples the data first and tries to balance the load between reducers, intuitively it does not seem to reduce the bandwidth usage and is therefore not too much of a concern in our model. If the data is sorted already, one can use `merge` where is uses a MapReduce job to sample the input and require a second Map phase to complete the join, which is virtually no Reduce phase. If this is used out model can also potentially regard this as 0 cost.

2.3.1.10 LIMIT

LIMIT makes sure the number of records in result is bounded by the number specified by the user, and ordering is generally not guaranteed. LIMIT also forces a reduce but the limit is also applied by the combiner.

Pig does not stop reading input when the limit is reached, but other optimisations can be performed such as doing limit early.

2.3.1.11 SAMPLE

SAMPLE gets a will get a percentage of the records in a data set according to the value specified by user. SAMPLE A 0.1 is the same as FILTER A BY random() ¡= 0.1 and can therefore be optimised the same as as FILTER.

2.3.1.12 PARALLEL

PARALLEL specifies the number of reducer to have and should only be used on operators that force reduce phases. It is set for each operator only and does not affect subsequent operators. The value of PARALLEL can also be a variable and provided at run rime. Determine this number can be very important for achieving good performances. When PARALLEL is not set, Pig will usually set one reducer for 1GB of data.

2.4 SQL query optimisations

In SQL, programmer suppose to only care about the functional side of their code, making sure their code behave correctly and hand the efficiency to the query optimiser. Query optimisation is very important for traditional SQL. Consider simple query such as

```
SELECT name, grade
FROM student, department
WHERE student.departmentId = department.id  and student.mark > 80
```

In centralised RDBMS, the cost is usually calculated by the number of disk reads required for each query. A bad query plan will create the cross product by reading one page of student at a time, go though all pages for dept for each page of student records. Then writes the cross product to disk, re-read the pages and filter by the requirement. A better plan would to filter the mark of students first, then proceed with a join algorithm.

The same principle can be applied to Pig. If a programmer writes

```
c = JOIN population BY driving_license_id, car_ownership BY owner_id;
d = FILTER c BY age >= 20 AND age <= 24;
```

This gives all the records of car owners who are 20 year old. This query, if executed as written will be very efficient. Assumes that 10% of the population are age $20 - 24$, which means we reduce the network usage for the population table by 90% if we filter first. This kind of trivial optimisation is already performed by Pig. However, there are cases where Pig do not automatically optimise, those areas are going to be the main focus of this project.

When an SQL query is sent to be executed, the query would be parsed and represented as a tree, which each operator would be represented as relational algebra. For example, the following query would be represented as the diagram in figure.

```
SELECT name, grade
FROM student, department
WHERE student.departmentId = department.id  and student.mark > 80
```

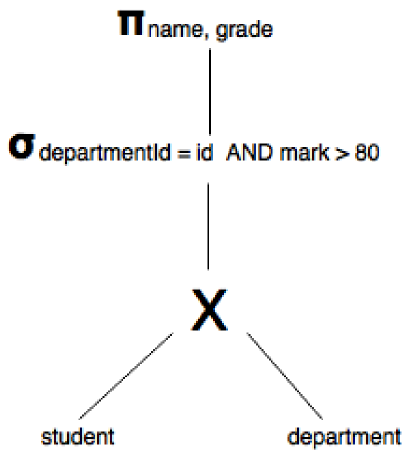

Figure 2.2: Parse Tree of an SQL Query

Pig also uses structure similar to these trees, and details would be discussed in later chapters.

After the tree like the one in figure 2.2 is constructed, the optimiser would try to evaluate the cost different query plans with equivalent results, then select the one with the lowest estimated cost and send to the execution engine. The three important components are equivalent results, different query plans and evaluating the cost.

Query optimiser usually try to generate many alternative plans, and try to find the best plan out of all of the candidates using a suitable cost model. There could be planing using different join algorithms or complete restructure of the tree. One problem is that we must make sure all the plans generated are actually equivalent. How do one make sure the different query plans you consider yield the same result? After all, your query can be as fast as you like, but if the result is wrong, the optimised query plan is useless. As mentioned earlier, the query is represented in relational algebra, which means there are mathematical laws that can be proven to show equivalence in two query, and as long as the alternative queries can be shown to be equivalent in relational algebra, we would have confidence that they are equivalent.

One example of the laws is, if both R and S have attributes C[11], then

$$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie \sigma_C(S)$$

There are many other rules that generates equivalent quires, but it will not be discussed further as the focus of this project is not query plan enumeration but cost model design.

2.5 Mathematical Tools

During this project, to perform analysis and evaluate the performance of the models, we would need some mathematical tools to help us in different situations. While assuming all readers are familiar with basic mathematics and statistics such as the meaning of uniform distribution, average and sum, some less well-known formulae to the general public would also be used. I will discuss these formulae so that reader who are not familiar with them can follow the content of the report easily.

2.5.1 Spearman's Rank Correlation Coefficient

After models have been designed and implemented, we would need to evaluate the model and see how well it works in estimating the cost of queries. As discussed before, we would like the model to be able to differentiate the cost of different plans, hence we want the model to be able to rank different queries accurately.

To measure how two ranking systems correlates with each other, we can use the Spearman's Rank Correlation Coefficient. Assuming there is no repeating values, a value of 1 means two ranking systems have exactly the same ranks for every members, and -1 means on is the reverse of the other.

Assuming we have two lists of ranks, X and Y, where X_i and Y_i represent the rank of cost of query i in `TPC-H`, and the formula for Spearman's Rank Correlation Coefficient is as follow:

$$r = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$$

d_i is the difference of rank of query i, and $d_i = X_i - Y_i$.

The advantage would be that we can have a single value that indicates the performance of a particular model, and thus allow us to easily compare against different models or different performance of the same model after modifications. The disadvantage would be that it encapsulate too much information and therefore further analysis would also be done when we are look at the results of each model.

2.5.2 Sigmoid Function

When we are doing estimations, we sometimes want to have an output value between the upper bound and lower bound. If we, however, want the output value to scale with another variable, `t`, so that the output grows as t grows, while at the same time we do not want our output to go above the upper bound, we cannot just simply take the average.

The sigmoid function, or the logistic function, written below is a function that has the properties we want.

$$S(t) = \frac{1}{1+e^{-t}}$$

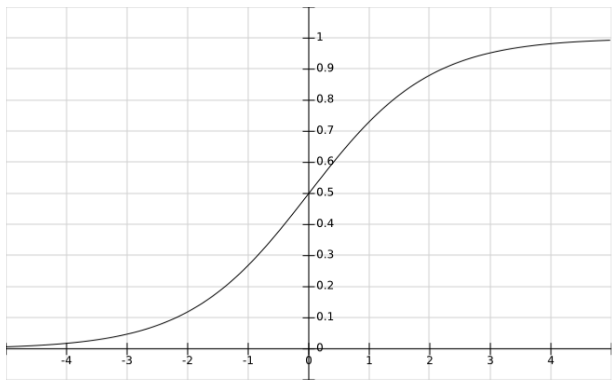

Figure 2.3: Plot of S(t). Image Source: http://fooplot.com

If we look at the plot of the sigmoid function, we can see that the output is trapped between 0 and 1. This corresponds to the upper bound and lower bound of the output value. As the input, `t`, grows, the output also grows and never exceed the upper bound. In our project, we can transform the function multiplying the function to the range of output and add the lower bound, so that the output range will be exactly the upper bound and lower bound that we required.

$$S(t) = \min + \frac{1}{1+e^{-t}} \times (max - min)$$

If we want to scale the function so that the amount needed for t to affect the output grows as the range grows, we should have

$$S(t) = \min + \frac{1}{1+e^{-t/(max-min)}} \times (max - min)$$

The reason why we required this in our project is that we would need to estimate the number of unique tuples in the tables, and it is difficult if we only know about the distributions of each attribute in the tuple.

For example, we might want to know how many unique tuple we have in tables with attributes `(age, gender, grade)`. Lets assume that we have 4 different `ages`, 2 different `genders` and 4 different `grades` in the table, the total combinations we can have is $4 \times 2 \times 4 = 32$. Assuming we have only 10 students, that makes the number of unique tuples to be bounded by 4 and 10, that is, we cannot have lower than 4 unique tuples as we already have 4 unique ages, and we cannot have more than 10 because 10 is all we have.

This give us the lower bound and upper bound, and it is natural to assume that the more combinations we have, the more likely we would have more unique tuples, and therefore the estimation we give must grow as the number of combination grows, and this make the difference between the number of combinations and the upper bound, $32 - 10$, our input variable.

Using the formula we have above, we have:

$$\begin{aligned}
S(t) &= 4 + \frac{1}{1 + e^{-(32-10)/(10-4)}} \times (10 - 4) \\
S(t) &= 4 + \frac{1}{1 + e^{-22/6}} \times 6 \\
S(t) &\approx 9.85 \\
S(t) &\approx 10
\end{aligned}$$

Therefore in this case we would estimate that the number of unique tuple is 10.

2.5.3 Histogram

Histogram is a representation of distribution. This is used in one of the model to represent the distributions of values in individual attributes in tables. Inside a histogram, we have `bins` and `frequencies`. `Bin` represents some values, and `frequencies` shows the amount of values that belong to a particular bin. In the scope of this project, for numerical values, each bin represent a range of values, and for string values, each bin represents a particular string.

Numeric Histogram	
Bin	Frequency
0+	0
20+	10
40+	50
60+	300
80-100	100

String Histogram	
Bin	Frequency
"Alice"	2
"Bob"	10
"Charles"	8
"David"	6
"Ed"	9

Table 2.1: Examples of different types of histogram used in this project

If we look at table 2.1, we can see in the Numeric Histogram that the `bin` with ranges 0+ (any value i that $0 \leq i < 20$) has a `frequency` of 0, which mean no values in the attribute falls into this range. The `frequency` of `bin` 40+ (any value i that $40 \leq i < 60$) has a frequency of 50, which means in the attribute, 50 values falls into this range.

If we look at the String Histogram in table 2.1, there are 10 people whose name are exactly "Bob" in a given table.

Chapter 3

Project Set-ups

3.1 Hadoop Cluster

To be able to run and test queries, we need to have a Hadoop cluster at our disposal. This would allow us to test alternative queries and measure running time and cost through the logs of Pig jobs.

To set up the cluster, I have created nine virtual machines through CSG's Cloud Stack service[14][15], one as the master and eight as slaves. The virtual machines may not run as fast as typical server machines, as it is sharing resources with other virtual machines hosted in the environment. This would potentially affect the running time of Pig queries, however this might not be a problem of our project as we are only measuring the amount of data transferred between nodes, which is not affected by the speed of CPU at each nodes, therefore it would not affect our measurement of Pig quries' cost.

3.2 Local Machine

At my local machine, I have written programs that allows Pig queries to be run remotely in the Hadoop via `Java`'s method call, as well as programs that can generate Pig scripts from templates given schemas and other informations. This allows tests to be carried out easily from my development environment without using `SSH`.

The Java programs that runs Pig script is done using `Linux`'s system call, and would need Hadoop's URL to be set up in the local operating system. The logs and text output of each Pig jobs can be stored separately, and allow future analysis after queries have been executed. To be able to submit Pig queries to the cluster, I would also need to connect to Imperial College's network, i.e. via Imperial College's VPN[16].

3.3 Benchmark

To be able to measure how well my models are working, I would need to have some sample queries that I can measure the cost then compare against estimations produced by my models.

I can download some sample data from the internet and write some sample queries on those data, but this would often not be representative of what queries a typical user would want to run. I can alternatively find some ”real world” data and sample queries, which might gives me a more typical query that people do actually run, but this approach has limitation that the data is usually fixed or is difficult to adjusted in size to suit my need. As discussed in later chapter, the ability of having a variable dataset size is very useful when evaluating the performance of our models.

One good solution to this problem is that I can make use of common benchmark database such as `TPC-H` [17]. `TPC-H` contains a set of non-trivial queries that would involves using`GROUP`, `JOIN` and many other Pig's features, and it has a good mixture of costly, time-consuming quires as well as some quires that are light-weight. This provides the advantage that I can benchmark against a commonly used benchmark database and see how my model performs on complex queries.

`TPC-H` also comes with a database generator that allows it's dataset to be scaled so that it can have any sizes [17]. This allows me to test my model in many different settings and might provide insight into problems that could not be detected when we are running against a fixed size dataset.

I have generated `TPC-H` dataset in the size of 10 MB, 100 MB, 1 GB AND 10 GB, and all are uploaded to HDFS to be used by pig queries later on.

While `TPC-H` provides `SQL` codes that represent it's queries, Pig code is not provided. To make sure that I cannot tune the Pig queries specifically to make it runs well with my model, I would be best to make use of Pig version of `TPC-H` queries written by others.

Fortunately this is done by Jie Li from Horton Work, and the code is published on [18]. Since readers might want to see the code when those queries are referred to in later chapters, I have included these codes by Jie Li in the Appendix. During this project, I would use these queries as a benchmark, I would measure the cost of running these queries through my Hadoop cluster, and make comparisons to my estimations for those queries on the specific dataset.

Chapter 4

Parsing and Analysis the Pig Script

To develop models that can analyse Pig scripts, it is important that we can parse the Pig scripts and allow our programs to understand them. There are two ways we can build a parser, I can write a parser from scratch, which will give us much more flexibility and control over the data structures and we would not have dependencies on external projects, or, as Pig is an open-sourced project, I can make use the parsers in the Pig project itself, in this case I will not have to implement the details of Pig Latin language, but face the problem that there might be limitations in what I can do with the existing architectures.

After researching through the Pig's Source code, it turns out that Pig codes provide a very good and general architecture where I can extend to suit my own need, namely the pig programs are parsed to a tree like structure and visitor classes and walker classes are already defined. Since the tools provided by Pig is very general and flexible, clearly the benefit of extending the Pig project far outweigh the cost and I am going to modify the source code to suit my own need should any limitation arises.

4.1 The Overall Picture

To obtain the parsed tree, we would need to invoke method `registerScript(InputStream in, Map<String,String> params)` in `PigServer`, and the PigServer class would perform parameters substitution and parsing, at this stage, the `LogicalPlan` is generated.

The `LogicalPlan` is a tree of `LogicalOperator` [19] such as `LOLoad` which represent the load statment of a Pig Latin program, and inside these operators, there are associated informations such as the path to load for a `LOLoad` operator or the expression tree for a `LOFilter` operator.

When a physical plan is needed, the `ExecutionEngine` would compile the `LogicalPlan` into a `PhysicalPlan` and when a MapReduce plan is needed the laucher would covert the `PhysicalPlan` to a `MROperPlan` which consists of descriptions of several MapReduce Jobs.

The Pig project contains visitor classes already defined for all of these plans and I can therefore extend those classes to implement my cost models which requires trversing the parse tree of a pig program. Walker classes which guarantees the order of nodes being visited in the tree is already provided in the Pig project, which makes analysis easier. For example, when analysing the cost of a particular operator of a script, it would be essential to have already analysed the operators that the current operator depends on, when estimating the cost of a join statement, it would be important to know the sizes of the two tables that is being joined. the `DependencyOrderWalker` classes would make sure that the operators are visited in the order of dependency and therefore take away the unnecessary complexity when analysing the plans..

4.2 Choice of Plans to work with

Since there are three different kind of plans generated for each Pig query, namely `LogicalPlan`, texttttPhysicalPlan and `MROperPlan`, it would be important to pick one of the plan to work with. Naturally, `LogicalPlan` would be the closest to what we have written in Pig Latin, MapReduce would be the closest to what actually get executed on the Hadoop Cluster. If we have a look at this simple Pig Latin code

```
join1 = JOIN orders BY o_custkey, customer BY c_custkey;
```

PigServer would produce the following Logical and MapReduce plans(with some details omitted).

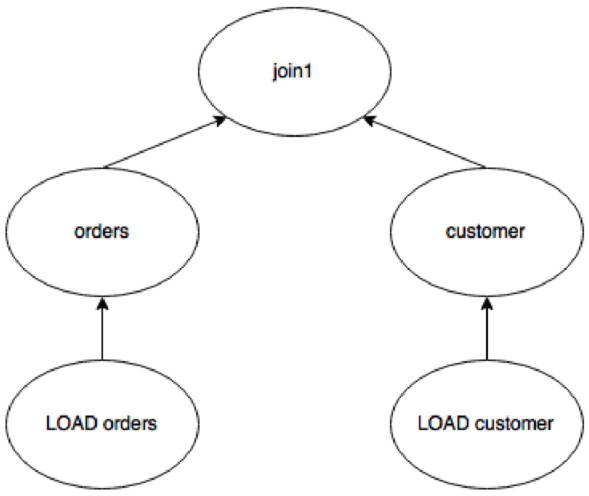

Figure 4.1: Logical plan generated for the above query

We can see that logical plan consists of two `LOAD`'s, and one `JOIN`, and in the MapReduce plan, we can see that there are also two `LOAD`s, but the there is a `Local Rearrange` after each `LOAD` and there is a `Package` at the start of the Reduce stage. We can see that it is not immediately obvious what the MapReduce plan is trying to do, at least not as easy to understand as the `LogicalPlan`.

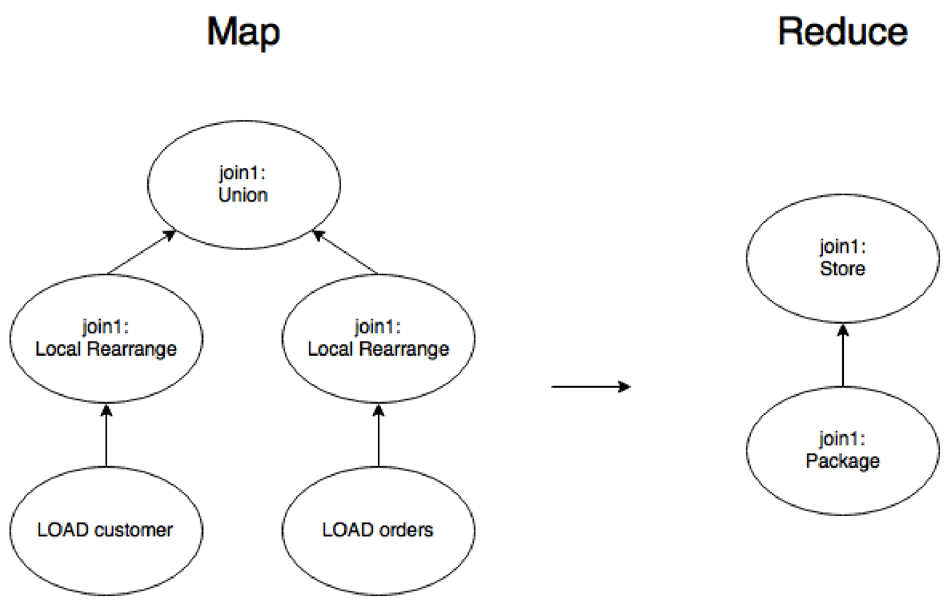

Figure 4.2: MapReduce plan generated for the above query

I have made the decision to focus on the logical plan and use it as the basis of cost analysis because I feel that it gives a closer resemblance to relational algebra, and thus it might be easier to understand the semantic and apply cost estimation techniques which are well established in SQL queries.

4.3 Data Structure of the LogicalPlan

Now that we have chosen `LogicalPlan` as the Pig's query plan representation in our models, we should understand what how this data structure works and what information it provides, at a high level, so that we know what we can use for analysis.

4.3.1 The LogicalPlan

The `LogicalPlan` is a directed graph of the Pig's `Logical Operators`, with the nodes being each operators in the Pig query plan and the edges being the dependencies (data-flow) between two operators. For example, in figure 4.1, the arrows from `LOAD`s to `JOIN` shows that the `JOIN` depends on the two `LOAD`s and data flows from the `LOAD`s to the `JOIN`.

In addition to all of it's functionalities which aids the parsing of of the pig query, the plan can give us informations about the dependencies and hence we need to enquire it when we need dependencies informations. Usually the plan would start with a `LOAD` operator and ends with a `STORE` operators.

Visitor class for the `LogicalPlan` is also provided in the Pig project and this allows easy implementation of any class that requires traversing the plan, and for our cost models, we are going to make use of the visitor and hence we can leave out many unnecessary difficulties not related to the optimisation of Pig query.

4.3.2 The Schema

The operators which falls into a `LogcialPlan` are `LogicalRelationalOperators`, and they all contains a `LogicalSchema` which is basically a collection of `LogicalFieldSchema`, which is a class that records the schema informations of individual attributes of tables. It includes informations such as the `alias`, `type` and `id` of an attribute. The most important pieces of information here are the `type` and `id`. we have to make different interpretation and calculations according to the type of an attribute, and in the context of this project, for example, estimations about the `bag` type can not be calculated as accurately as the `integer` type. The `id` is also a very important piece information, if the an attribute is the same after an operators, such as being in the result of `SELECT` without alteration, the column is going to have the same id, so this would give us valuable information regarding whether we can have a known accurate distribution of a column.

The assumption in this project is that schema information would be provided for the queries we optimised, which is a reasonable requirement and would properly limit the scope of the project to work with queries without having to put type inferences into account.

4.3.3 LOLoad

LOLoad is the entry point of Pig queries, it provides the path where the input files are loaded. This is very important as it allows the program to automatically collect information about the input without user having to manually provide them. Users need to make sure the path in the Pig script and the URL to the HDFS site are correct, which, if not, the Pig script will not run, and the program can send requests to collect information about the input files either via either HDFS commands in Java or pre written Pig scripts. And this is exactly what we have done when implementing out models and will be discussed in following sections.

4.3.4 LOFilter

As discussed in previous chapters, `FILTER` can change the size of it's output, and the percentage changed can vary from 0% to 100% of the original output depending on the condition. The `LOFilter` class provides a `LogicalExpressionPlan`, an expression tree, to represent the select condition. The `LogicalExpressionPlan` has a similar structure to the `LogicalPlan` and consist of operators such as `add`, `subtract`, `and`, `or`, `larger than` and etc. We can make use of this plan and evaluate the condition ourself and make estimation of the selectivity of the condition.

Similar to the `LogicalPlan`, a visitor class is also provided and it should allow easier implementation of any evaluation of the `LogicalExpressionPlan`.

4.3.5 LOCogroup

For grouping, the most essential piece of information is what attributes we are grouping on, as it decides the number of records we are going to get. The `LOCogroup` operator has a map of `LogicalExpressionPlan` which maps from input table to plans. Each `LogicalExpressionPlan` provides information on what attributes are being used for grouping via the `ProjectExpression` class which each contains the id and type of the attribute for grouping.

Since `LOCogroup` permits grouping multiple tables together, having the map data-structure here facilitate a general implementation of analysis that accomondates any number of input tables.

4.3.6 LOJoin

Similar to `LOCogroup`, `LOJoin` operator also provides information on the attributes being join, and can therefore be used in analysis of the operator. They information is provided in the same way that a map from input table to `LogicalExpressionPlan` and each attribute is contained in a `ProjectExpression` object.

4.3.7 LOLimit

The `LOLimit` operator provides the number which the table is limited to, thus we can use it for our analysis.

4.3.8 The Walkers

Although not part of the `LogicalPlan`, the walkers of the visitor class is there to ensure a specific traversal order, and depending of what we are doing with the plans, we might need different traversal orders.

For cost estimations, we usually need informations about predecessors of operators in order to perform analysis. If we were to determine the size of a `JOIN` operator, we would definitely need to know about the sizes of the input tables, and would probably be better off having more information such as the distribution of each attributes of each input tables. This means we need the traversal to be done in the order of data dependency. In this case we can use the `DependencyOrderWalker` provided in the Pig project and ensure that we have all of the information needed when performing analysis on operators.

If we were to analysis the conditions of `FILTER`, it is a different story. As we know from evaluating any mathematical formulae, we need to solve from bottom-up, as each operators would be expecting values from it's child-nodes. In this case we can use the `DepthFirstWalker` which would be suitable in this situation as it would ensure all child-nodes to be traversed before visiting the parent nodes.

Chapter 5

The Simple Cost Model

5.1 Design Philosophy

At the start, we would like to develop a simple model, a model which estimates cost of queries with very limited informations, and therefore require little overhead and assistance from the users. This model is likely to be inaccurate due to the fact that it does not have information such as cardinality, selectivity and distribution of values in attributes of input tables, but it can be very useful in the situation when those information are not available or are very expensive to collect. In some RDBMS, if the user do not explicitly provide these extra informations or command the RDBMS to collect these informations, the RDBMS would in fact have to rely on assumptions made without looking at the actual data.[11]

We have decided to require two pieces of information of input tables, the schema for the the input tables and the sizes of the tables. We would expect programmers who want their pig script to be optimised to provide schema details, and the sizes information is very cheap to obtain and is actually very important to estimating the cost of a query. A good example would be a Cartesian product is viable if one or both of the tables are very small, i.e. less than 10 rows, and having table's sizes would be a very important indication of whether a query is viable in terms of cost.

From the information we are given, we are not going to look at the details of each operators, i.e. we will not analyse the condition for `FILTER` or look at what attributes are used as join keys. Instead, we would do experiment to find out the general behaviours of certain operators or to use analysis and established theories to make assumptions about the behaviour of operators, then we would implement the model using assumption made from those analysis or experiments.

To analyse the pig script, we have also decided to look at each operators in the script sequentially, such that the model would assign a cost to each operator depending on the type of operators and sizes of it's inputs, and the total cost would be the sum of all individual costs. This would ignore the fact that `Combiner` can be used to reduce the cost of certain operators, and thus some operators are not as expensive as estimated.

As discussed before, we would use the bandwidth being used in reduce phase, i.e. from map nodes to reduce nodes, as the cost of each operators.

5.2 Specification

This model must be able to take in any Pig scripts and informations about the sizes of the input and produce an estimation of cost in terms of the amount of data transferred from map nodes to reduce nodes when the script is executed on a Hadoop Cluster. It must be able to provide the answer quickly, i.e. less than a few seconds for each query.

5.3 Design

According to this specification, we would need a few components in this program:

1. An object which can read in a Pig script and produce a data structure that can be analysed
2. An object which retrieve size informations about input tables from HDFS
3. An object which can perform the analysis based on data provided by the other two objects

I have also decided to use Java as there are libraries to work with both Pig and HDFS in Java, and it is a compatible language that can be used on almost all platforms.

5.4 Experiments

Before we actually implement the model, it is important to confirm some of our hypothesis, so that we can produce a model with more confidence.

5.4.1 Obtaining the cost of Pig query

Since we want are using the bandwidth between map nodes and reduce nodes as the cost, this information can be obtained by parsing the log files of Pig jobs. By matching lines like these in the log file using regular expression, we can obtain the cost of each Pig job easily.

```
2016-05-24 15:03:51,518 [localfetcher#1] INFO  org.apache.hadoop.mapreduce
.task.reduce.InMemoryMapOutput - Read 1582367 bytes from map-output for
attempt_local1615850287_0001_m_000000_0
```

5.4.2 Number of attributes in GROUP and CUBE

One interesting fact that we should know about is the whether the cost increases if we are grouping or cubing on different number of attributes. The intuition of grouping would be that the cost should stay the roughly the same regardless of the number of attributes it is grouping on, because apart from the newly formed `group` column which represents the group keys, the results will contain one copy of the original input table that has been split into many different buckets, and therefore the size should not change much. Where as for `CUBE`, a tuple is going to be matched with multiple group keys. If a tuple is matched with `(female, 100)`, it will also match with `(female,)`, `(,100)` and `(,)`, and presumably these copies of tuples all have to be sent through the network from map nodes due to they are associating with a different key in MapReduce, so we should expect an 2^n increase from respected groups with the same group attributes, where n is the number of attributes being grouped on.

I have conducted an experiment which I wrote quires that `GROUP` and `CUBE` on different attributes, and the I measure the cost of these queries.

Query	CUBE	GROUP	ratio
PART_brand_type	9.74×10^7	2.76×10^7	3.53
PART_size_type	1.01×10^8	2.76×10^7	3.66
PART_brand_type_size	1.94×10^8	2.76×10^7	7.03
ORDER_custkey_orderstatus	7.22×10^8	1.86×10^8	3.89
ORDER_orderpriority_orderstatus	7.01×10^8	1.85×10^8	3.78
ORDER_custkey_orderstatus_orderpriority	1.38×10^9	1.85×10^8	7.45

Table 5.1: cost (byte) of GROUP and CUBE running on 1GB of TPC-H dataset

Result of the experiment can be found in 5.1, the capital words in the first column represent tables in the `TPC-H` dataset and words in lower case are the attributes being grouped on. According to this result, previous analysis is likely to be correct. We can clearly see that when grouping the `PART` table, the cost is identical and varies less than 1% in the case of the `ORDER` table. We can also see when the cost of `CUBE` is roughly 2^n the cost of `GROUP` on the same attributes, where n is the number of attributes, the difference is likely to do with the overhead, and the extra column generated as the shuffle key. As we can see from the table `ORDER`, being the larger table, has a closer ratio to 2^n possibly due to the overhead is relatively smaller.

5.4.3 GROUP with multiple input

`GROUP` and `COGROUP`, being identical operators, allows multiple tables to be grouped at the same time, and it would be important to know whether this have any implication on the cost, i.e. would this reduce the cost compare to grouping multiple tables individually. If we think about the algorithm, which each tuple would be grouped with it's group key, and and sent to reduce nodes to combine, it is likely that the cost would be linearly correlated to the cost of grouping them individually as all tuples from all input tables have to be sent to reduce nodes once. If this is the case, estimating the cost of GROUP can be done by simply summing the cost of grouping on individual tables.

TABLE_key	Cost (byte)
CUSTOMER_c_custkey	2.67×10^6
CUSTOMER_c_nationkey	2.68×10^6
NATION_n_nationkey	2.50×10^3
ORDERS_o_custkey	1.83×10^7
PART_p_brand	2.67×10^6
PART_p_brand_p_type	2.73×10^6

Table 5.2: Cost of grouping individual tables on 100MB TPC-H

TABLE_attributes	Cost (byte)	Sum of Individual Groups
NATION_n_nationkey_CUSTOMER_c_nationkey	2.69×10^6	2.69×10^6
PART_PART_p_brand	5.34×10^6	5.34×10^6
PART_PART_p_brand_p_type	5.45×10^6	5.45×10^6
PART_PART_PART_p_brand_p_type	8.18×10^6	8.18×10^6
CUSTOMER_c_custkey_ORDERS_o_custkey	2.10×10^7	2.10×10^7

Table 5.3: Compassion of COGROUPs and individual GROUPs on 100MB TPC-H

Amazingly, the cost, which is the bytes from map node to reduce nodes in the context of this project, of grouping multiple tables is identical to the sum of the cost of individual tables as suggests in table 5.2 and table 5.3. This suggests that, at the time of the experiments are conducted, grouping multiple tables together in Pig is implemented as they were grouped separately. Although this can be changed in future version of Pig, we can safely assume the cost of grouping multiple table is the sum of individual tables, as our earlier analysis suggests any optimisation would still result in similar results.

This result also suggests that `COGROUP` should always be preferred over grouping separately then join as we are effectively omitting a join stage, which would incur a separate reduce phase.

5.4.4 Growth of the queries cost

Another important information that we are interested to know is that how the cost of the queries grow when the sizes of input increases. Intuitively, a query that involves only `SELECT`, `FILTER`, `GROUP` and other operators that does not involve operation resembling cross product should have their cost increases linearly or less with the input size, because these operators can be implemented via algorithms with linear or constant space. For queries involving `JOIN`, the cost might be exponential or less depending on the nature of the join. If the joins are on attribute with only one value, then we are effectively performing Cartesian product between the input tables. Although the cost of a single `JOIN` is likely to be the same regardless of what we are joining on, because cross product is done in the reduce phase hence the extra tuples do not come from map nodes, bigger result from `JOIN` is going to increase the cost of operators which depend on that output and hence increase the cost of the overall query.

I have conducted this experiment by generating different copies of `TPC-H` data set each with size 10 MB, 100 MB, 1 GB and 10 GB, run all 22 of the TPC-H queries with these different input sizes, the I calculate the growth in cost of each query and what we have in table 5.4 is the result. A value of 10 would represent perfect linear growth of cost of query.

TPC-H queries	1	2	3	4	5	6
Cost growth	5.54	10.02	9.99	10.01	9.84	5.56
TPC-H queries	7	8	9	10	11	12
Cost growth	9.99	10.40	10.23	9.94	10.08	10.00
TPC-H queries	13	14	15	16	17	18
Cost growth	10.20	10.01	17.85	10.04	9.99	10.04
TPC-H queries	19	20	21	22		
Cost growth	9.99	11.26	10.20	9.99		

Table 5.4: Average growth of cost of each TPC-H queries

As we can see from table 5.4, most of the queries do grow linearly with values very close to 10 with the exceptions of `Q1`, `Q6` AND `Q15`.

For `Q1`, the growth of cost of dataset size size 10 MB, 100 MB, 1 GB and 10 GB are 854 bytes, 884 bytes, 5096 bytes and 50112 bytes receptively, and for `Q6`, the growth are 23 bytes, 23 bytes, 138 bytes and 1334 bytes respectively. We can see that their costs do not vary much from 10MB and 100MB, since `TPC-H` recommend minimum size of 1 GB[20], it is entirely possible that the 10 MB and 100 MB are not distributed the same way as dataset with sizes above 1 GB. Since the 1 GB and 10 GB dataset does show a growth factor of 10, we can reasonably conclude that the growth of both queries are linear.

```
flineitem = filter lineitem by l_shipdate >= '1996-01-01'
    and l_shipdate < '1996-04-01';

sumlineitem = foreach flineitem generate l_suppkey,
    l_extendedprice * (1 - l_discount) as value;

glineitem = group sumlineitem by l_suppkey;

revenue = foreach glineitem generate group as supplier_no,
    SUM($1.value) as total_revenue;

...
```

For `Q15`, if we look at the snippet above, we can see that there is a group and aggregate onto some filtered results from `lineitem`. We can see that the aggregation function is `SUM` which can be optimised by the `Combiner`. Since the number of unique `l_suppkey` are scaled with the `Scale Factor` of the dataset[20], we would have more `l_suppkey` at a larger dataset, and therefore more unique groups at each map nodes, therefore less can be optimised by the `Combiner` as the dataset grows.

5.5 Implementations

In this section, we are going discuss the overall picture of the implementation of this model, without going into details about the actual programs, we will explore the design choices we made and overall algorithms.

5.5.1 Parsing the script

For parsing a Pig script, as we have discussed in previous chapters, this is already implemented in the Pig project. The only limitation here is that the `LogicalPlan` class that we are going to use is not exposed by the `PigServer` class, luckily the project itself is open sourced and I had added a few lines of code in the `PigServer` class to expose the `LogicalPlan` inside. This might become a problem for distributing my project as I would have to provide my own version of Pig if others would like to use my model, but it is not a problem within the scope of this project which is to develop a working cost model for Pig.

5.5.2 Retrieving sizes' information from HDFS

This can be easily achieved through `FileStatus` provided by Hadoop library and by calling the `status.getLen()` method we can retrieve the size of a particular file in byte. All we need from the user would be the URL to the HDFS site.

5.5.3 Overall work-flow of the Model

During the analysis, the `LogicalPlan` will be traversed in the order of data dependency, and therefore I have made a `OperatorMetadata` class which holds information about each operators, namely their output size and the cost of the operator. The size information would be used by other operators which depends on it and the cost is used when calculating the total cost of the Pig query.

As each operator is visited, the cost would be calculated based on the sizes of the predecessors, and cost would be stored and when all of the operators are analysed, the total cost would be calculated by summing the costs of individual operators.

5.5.4 LOLoad

In this stage, we collect the sizes information about the table being loaded, by connecting to HDFS and enquire about the file path contain in the `LOLoad` operator, and store this as the output size of this operator.

One of the potential complication of using the file size as the table size is that the actual table size is usually smaller in memory, one reason could be that the `TPC-H` data set input files are strings of characters, and all data types, including numbers and dates are stored this way. When those data are loaded into memory, or bring sent through the network, they are usually smaller as they are represented using more efficient data types. A 10 digit integer require 10×2 bytes to be represented in characters but only need 4 bytes with Java's `int`.

However, as the model only need to be able to distinguish the relative difference in cost, i.e. be able to rank different query plans according to the cost, this approach should not be a problem if used consistently throughout all queries analysed by this model.

5.5.5 LOFilter

For this operator, we must decide on how much of the table would remain after the filter conditions has been applied. To acquire an accurate prediction, one should look at the specific condition and the distribution of values of the input table. But in this simple model we have decided to not have information about distribution of values in the input table.

One way forward is to look at the filter conditions and make assumptions about the distribution via the attributes type and name. For example, we might be able to say we always assume uniform distribution, and if the attribute name is "gender" or "birthday" we can kind of have an idea of the range of possible values can be hold in the column.

Implementing this estimation means that we need to have a table of attribute aliases and some assumed distribution of the values. These assumptions are very likely to be wrong and the construction of such table is complex, on top of that, we might not be able to achieve a significant improvement in accuracy of estimating the reduction rate of a query over a constant reduction. Apart from the obvious problem of large number of aliases to consider, and many different form of the same alias, lets consider we have a way to work out whether an attribute is "birthday", to have a default distribution, we can lookup the age distribution of world's population, but then the age distribution of students at a school would be very different from the age distribution of members of a country club, and they would both be very different from the world's population's age distribution. For most of the attribute we just cannot have any meaningful assumptions, such as the `comment` attribute which frequently appear on many tables in `TPC-H` dataset.

Another approach we can take is to assume a constant reduction rate, while this is not going to be very accurate, this approach is recommended in [11] for estimating the output of `FILTER`. This approach allows fast estimation of `FILTER` and it is also a monotonic function, which would be the case if the same condition of `FILTER` are applied onto tables with the same distribution but larger size.

In this case, I have adopted the second approach and half the size of the input table whenever a `FILTER` is applied, and the cost is assumed to be 0 as no reduce phase is required to complete the `FILTER` in Pig.

5.5.6 LOStore

This is the operator that we do not need to pay much attention to as data arrive at this stage would be the same regardless of intermediate processing, otherwise we would have incorrect optimisation.

5.5.7 LOForEach

`LOForEach` has no reduce phase, therefore we can set the cost of it as 0. To estimate the output size of `LOForEach`, we should notice that the number of tuples (rows) in the output would always be the same as it is a Projection from the table. Since we can have the schema of the input table and output table, the approach I have chosen is to approximate the average size of a tuple by looking at the type of each attributes. After calculating the size of tuples of input and tuples of output, we can estimate the `shrink ratio` by calculating $\frac{\text{size of input tuple}}{\text{size of output tuple}}$ and multiply by the input size.

Type	Size (bits)
BOOLEAN	1
BYTE	8
INTEGER	32
LONG	64
FLOAT	32
DOUBLE	64
DATETIME	24 * 8
BYTEARRAY	128 * 8
CHARARRAY	256 * 8
MAP	4096 * 8
TUPLE	64 * 2
BAG	1024 * 16 * 8

Table 5.5: Estimation of type size

To estimate the tuple size, I have chosen to assign a size to each data type as shown in table 5.5, for many types such as `BOOLEAN` and `DOUBLE`, their sizes are fixed and this is likely to be accurate, whereas for type like `BAG`, this is going to be inaccurate. Although this approach is not going to work very well with these dynamic data-types, it does serve the purpose for providing a quick and easy estimation of output size.

5.5.8 LOJoin

For `LOJoin`, we need to decide both the output size and the cost of the operation. As discussed earlier, it is relatively easy to determine the cost of joins, i.e. both tables have to sorted by the join key and sent to reduce node according to the sort keys. so we can estimate the cost of joins as the sum of the table sizes.

Estimating the output size of the `JOIN` is much more difficult to do. Usually, the simplest estimation formula requires the cardinality of the join keys [11], which in this case is not available. We of course can consider the approach which was discussed in the analysis of `LOFilter`, namely to assume some distribution according to types and aliases, and we would just face the same problems.

In this case, I have chosen the same approach as I have done in `LOFilter`, which is to assume a constant factor of 2 multiplied by the sum of sizes of input tables. While it is possible for have a smaller table after join, in many cases we join because we normalised the table to accommodate one-to-many or many-to-many relationships, and usually the size increases in this case as those matched keys produce Cartesian Products.

5.5.9 LOCogroup, LOCube

To estimate the output size and the cost of `LOCogroup` is very easy, as it is linearly related to the input size, a good estimation can simply be the input size. This is because regardless of what attribute is used for grouping, each tuple table will need to be sent to the reduce nodes once and the output is going to have all tuples in the input table (now stored in bags). The same goes to `LOCube`, since for each attribute that we group on, a tuple is going to be matched 2^n times, where n is the number of attributes we are grouping on, and each would have to be sent to the reduce nodes, hence both the cost and the size is going to be 2^n of the corresponding join.These claims are confirmed by previous experiments.

What we do not have is the number of tuples of the output, as it often determine the size of the aggregates that follows the group. What we have effectively done is that we have set a fixed ratio of size reduction of aggregation by giving a fixed size to bags and other types that it can be aggregated to, as shown in table 5.5.

5.5.10 LOSplit, LOSplitOutput

`SPLIT` allows multiple filters to be applied to a table at the same time and produced multiple tables, under the hood, the input to `SPLIT` will be the input to `LOSplit`, but `LOSplit` do not contain any conditions, it serves as an input to `LOSplitOutput`, and `LOSplitOutput` is basically an `LOFilter` and contains the filter conditions. Therefore each `LOSplit` has it's input size as it's output size and 0 cost, where as the `LOSplitOutput` is implemented the same way as `LOFilter`.

5.5.11 LOUnion

Since the only information we are keeping throughout the analysis is the size, `LOUnion` can simply be calculated by adding the sizes of all of the input tables, and it would have a cost of 0 as it does not require a reduce phase.

5.5.12 LOSort

`LOSort` does not change the size or the distribution of the input, as all it does is to re-order the table in a certain way, therefore we can us the input size as the output size. We can also set the cost of the operator as the input size as all tuples have to be sent through the cluster to reduce nodes.

5.5.13 LORank

`LORank` is very similar to `Sort` as a sort is required to rank a table and can be implemented almost the same way, with only two differences. One is that sometimes a sort is not necessary if no attributes are specified, in this case pig will simply prepends a sequential value to each tuple[21], and no reduce phase is required, and therefore has a 0 cost. Another difference is that a new column with `int` type is always created to store the ranks and we can use table 5.5 to estimate the increase in average tuple size and generate an estimation of change in table size.

5.5.14 LODistinct

For distinct, a reduce phase is required, although some duplicates might be removed by the `Combiner` before the reduce phase, I would still assume the whole table would be sent through as it would be difficult to determine how much work would be done by combiners, and whether duplicate records to exists in the same file split that can be picked up by the combiner.

In terms of output size, I have chosen 0.9 as the constant reduction factor of `LODistinct`.

5.5.15 LOLimit

For `LOLimit`, we know the number of tuples that it is going to produce, so we need to know the number of tuples in the input table to work out the reduction ratio. Although we do not have this information, we can estimate it by dividing the input size by the average tuple size using table 5.5.

$$\text{no. of input records} = \frac{\text{size of table}}{\text{size of a record}}$$
$$\text{size reduction ratio} = \frac{\text{no. of output records}}{\text{no. of input records}}$$

The cost of `LOLimit` would be 0 as it does not require a reduce phase.

5.5.16 LOCross

The row count (number of rows) in the output of `LOCross` is the product of number of rows of the input tables, which we can estimated using the method mentioned in the analysis of

`LOLimit`. After the output row count is known, we multiply this by the estimated output schema size using table 5.5 to get the estimated table size.

The cost of the cross is not immediately obvious, as it depends on the implementation. Pig implements `CROSS` by generating a synthetic join key, replicating rows, and then joins[11], and the cost is going to increase as the parralell factor increases as input record are duplicated by the factor of square root of parallel level[11].

Therefore the cost of cross would be

$$\sqrt{\text{parallel level}} \times \sum \text{sizes of input tables}$$

5.6 Evaluation

To evaluate the effectiveness of the model, we would first run all `TPC-H` queries in our Hadoop cluster with data-set size of 1GB, then we would measure it's cost by parsing it's log files. We would also let our model to run on these queries, and we would collect the estimated cost of each. After we have the real costs and estimated costs, we will rank each queries according to the cost then use Spearman's Rank Correlation Coefficient to score the performance of the model's ability to rank the model by the cost and the results are presented in table 5.6.

Query	Estimated Cost	Estimated rank	Measured Cost	Measured Rank	d	d^2
Q1	2.37×10^7	11	5.10×10^3	2	9	81
Q2	4.55×10^7	14	1.98×10^8	17	-3	9
Q3	3.55×10^7	13	1.18×10^8	14	-1	1
Q4	1.01×10^7	8	4.87×10^7	10	-2	4
Q5	6.30×10^7	18	2.66×10^8	20	-2	4
Q6	5.80×10^5	2	1.38×10^2	1	1	1
Q7	8.84×10^7	19	1.16×10^8	13	6	36
Q8	5.51×10^7	16	1.85×10^8	16	0	0
Q9	1.66×10^8	22	3.81×10^8	22	0	0
Q10	5.41×10^7	15	9.57×10^7	12	3	9
Q11	1.34×10^6	3	2.29×10^7	5	-2	4
Q12	9.73×10^7	20	3.49×10^7	9	11	121
Q13	5.04×10^5	1	2.54×10^7	6	-5	25
Q14	1.10×10^7	9	2.89×10^7	8	1	1
Q15	1.79×10^6	4	2.01×10^6	3	1	1
Q16	5.59×10^6	7	2.74×10^7	7	0	0
Q17	4.67×10^6	6	1.74×10^8	15	-9	81
Q18	2.35×10^7	10	2.00×10^8	18	-8	64
Q19	5.55×10^7	17	3.69×10^8	21	-4	16
Q20	2.40×10^7	12	4.92×10^7	11	1	1
Q21	1.09×10^8	21	2.36×10^8	19	2	4
Q22	3.15×10^6	5	1.72×10^7	4	1	1
					coefficient	0.74

Table 5.6: Result of Simple Model, cost in bytes

As we can see from table 5.6, this model has achieved a coefficient of 0.74, which is a indication of good performance of this model, it clearly shows that the Simple Model's ranking has a good correlation to the actual ranking of cost of queries, and can give useful insight how how costly a particular query plan would be when doing optimisations.

However, there are a few queries that appears to have diverted away from the actual rank considerably, they are Q1 ,Q12, Q17 and Q18, and it is important for use to analysis why this is the case. Q1 and Q12 are over estimated, such that their cost rankings are too high, whereas Q17 and Q18 had been underestimated as their cost rankings are too low.

We will now look at each quires individually to see what is causing our model to underperform when analysing these queries.

5.6.1 Q1

```
SubLineItems = FILTER LineItems BY shipdate <= '1998-09-02';
StatusGroup = GROUP SubLine BY (returnflag, linestatus);
PriceSummary = FOREACH StatusGroup GENERATE ...
SortedSummary = ORDER PriceSummary BY returnflag, linestatus;
```

If we take a look at the snippet of Q1 above, we can see that there is a FOREACH immediately after GROUP, and since we are grouping on (returnflag, linestatus), two attributes with very few distinct values, the cost of the ORDER is likely to be very low. But this is not enough to explain the cost being in thousands of byte. LineItems is a very large table, with more 700 MB in size, and the FILTER condition still output a majority of rows in the table. On possible reason would be that many of the aggregations after groupong are done in the Combiner stage, where a reduce is performed on the map nodes before data being sent to reduce nodes. Since the cardinality of group keys is very small, each map nodes would aggregate them and only output hundreds of bytes of data as there are only a few rows in from each map nodes.

For more detailed code of any TPC-H queries, please refer to the Appendix.

5.6.2 Q12

```
flineitem = FILTER lineitem BY l_shipmode MATCHES 'MAIL|SHIP' AND
                   l_commitdate < l_receiptdate AND l_shipdate < l_commitdate AND
                   l_receiptdate >= '1994-01-01' AND l_receiptdate < '1995-01-01';

l1 = JOIN flineitem BY l_orderkey, orders by o_orderkey;
sell1 = FOREACH l1 GENERATE l_shipmode, o_orderpriority;
grpResult = GROUP sell1 BY l_shipmode;
sumResult = FOREACH grpResult ..
sortResult = ORDER sumResult BY group;
```

For Q12, we can see that the initial FILTER is using a very specific condition and is likely to filter out many rows and have a mall number of rows in the output. Since we are assuming the table size to be cut in half, the output size is obviously too large, thus we have over estimated the query.

5.6.3 Q17

```
...
COG1 = COGROUP part by p_partkey, lineitem by l_partkey;
COG1 = filter COG1 by COUNT(part) > 0;
COG2 = FOREACH COG1 GENERATE COUNT(part) as count_part,
FLATTEN(lineitem), 0.2 * AVG(lineitem.l_quantity) as l_avg;
...
```

There are two possible reason that leads to the underestimation of this query. As we can see from the snippet above `COG1` is the product of `COGROUP` of `part` and `lineitem`. `lineitem` itself is a very large table. `COG2` is the result of applying `FLATTEN` to a bag, which is going to produce a cross product between all elements in the bad and other attributes in the `GENERATE`.

Another factor is that the `FILTER` after `GROUP` is based on the number of element in the bag, which is not something you can do in the `Combiner` stage. If `Combiner` was active in many other queries but not this one, and our model do not take combiner into account, we would estimate this query just as costly as some other, but in fact others might have been optimised by the `Combiner` and therefore this query would have a much higher cost ranking than we have anticipated.

5.6.4 Q18

```
...
COG = cogroup lineitem by l_orderkey, orders by o_orderkey;
COG = filter COG by SUM(lineitem.l_quantity) > 300;
...
```

`Q18` is also very similar to Q17, such that the `FILTER` after `GROUP` can only be applied once all results has been known, and is therefore not optimised by `Combiner`.

5.6.5 Discussion

In conclusion, the Simple Model is working well given the information it has, it clearly has the ability to distinguish queries based on their cost in most cases. Although many assumptions are made, the ranking of quires it has analysed has a strong correlation with the real rankings and it could sometime be used in cost based query plan optimisation when information are limited.

If we look at the four misbehaving queries, we can see that three of the mistakes could be caused by not anticipating the `Combiner`, a mechanism in Pig, and only one is inaccurate because we our assumption is too unrealistic. This might suggest when considering Pig and Hadoop, we might need to pay more attention to it's own design instead of only the traditional database optimisation theories as I have done.

One thing that can definitely be improved on is the number of bytes we have estimated. Although we only aim at getting the ranking right, so that we can use it in query plan selection, it would still be good if we can have a correct idea of how much resources will be spent on each query, and might give users some useful feedback if they have some very limited resources.

Chapter 6

The Histogram Cost Model

6.1 Design Philosophy

Now that we have developed a very basic model, which emphasis on speed and relies on many wild assumptions, it is time to move onto a more sophisticated model, that emphasis accuracy and does less ”rule of thumbs” but rely more on statistics and maths. In this model, we are going to try to estimate cost and output sizes based on available statistics, such as the distribution of values in attributes and number of rows in each intermediate tables. We are also going to make use of many of the information provided by the `LogicalPlan`, such as evaluating the filter conditions or look at specific join keys so that we can have a complete understanding of each queries.

Having some statistics allow us to sometime make very accurate calculations, knowing how many distinct value (cardinality) there is in the group key, we can know exactly how many columns would be produced after the aggregations, and know how many times a particular value occurs on both side of the `JOIN` allows us to know the exact size of the output of a `JOIN`.

In this model, we would try to calculate estimations base on all available statistics. If we have enough statistics to produce an almost exact output, we would do it that way, if there is not enough information to determine the output size and cost exactly, we would make estimations based on all available statistics and some reasonable assumptions.

Another important consideration is that we still want to rely as little user input as possible, which is why we have automate the statistics collections where possible.

6.2 Specification

This model must be able to take in any Pig scripts and informations such as distribution of values (which include number of rows, cardinality, selectivity, minimum, maximum and a histogram) of attributes of input tables and the sizes of input tables, to determine the cost of the script in terms of the amount of data transferred from map nodes to reduce nodes when the script is executed on a Hadoop Cluster. And it needs to be accurate in a way that can accurately identify the relative cost difference between different Pig queries.

6.3 Design

According to this specification, we would need a few components in this program:

1. An object which can read in a Pig script and produce a data structure that can be analysed
2. An object which retrieves relevant statistics of input tables from HDFS
3. An object which can perform the analysis based on data provided by the other two objects

Clearly the first component from the Simple Model can be reused, and we only need to implement the other two components.

6.4 Implementations

6.4.1 Collecting Statistics

For each attribute, I would aim at collecting a histogram that represents the distribution of the values in the column. I categories different types into two categories: Numeral and String. Any attributes with numeral types such as `int` or `double` would be classified as Numeral and all other types will be classifies as String. I chose String as the general type for a few reasons. One is that all data re originally stored as String and they can perhaps be converted into other types if required. Another reason is that values such as dates and time can be represented by string well because we can perform comparison of time using their string values if they are formatted properly.

There are two different approaches to gathering the histograms. For a numeric one, I can save space by having ranges as the the `bins` and count all values in the range into that bin. Inside the bin, the numbers are assumed to be uniformly distributed. This approach means we will not have the exact distribution for numerical values, but considering there are 2^{64} different values a `long` typed variable can hold, we would have to make sacrifices in accuracy in order to have a reasonable complexity. The min and max values of each attribute would be collected first, and depending on the range, a reasonable `number of bins` would be chosen (capped at 500). Each bin would have an equal size so the range at each bin would be $\frac{max-min+1}{\text{number of bins}}$.

For string histograms, there would be a choice, we either have a histogram that have a frequency of each individual strings (a complete distribution) or we do not have a histogram at all. We would not collect a histogram if the selectivity, meaning the ratio of unique values to number of rows, is high and at the same time the number of tuples in the table is also high. This is because a high selectivity means for each unique strings there is usually only one occurrence in the table, in this case having a histogram is the same as copying the entire column locally, and is very inefficient. Since the analysis involves constantly manipulating many histograms, and after waiting for results of analysis for more than half an hour and realising most of the time is spent on dealing with the large histograms, I have imposed this restriction to make the analysis more efficient. The reason why a ranged bin is not chosen for string is that it is not meaningful to have ranges in string, using lexical graphical order, there is an infinite amount of strings that can be fit into a bin, simply by adding more characters at the end. Due to this fact, the probability of any string occurring in any bucket is 0, and it is therefore almost impossible to be used in estimate how many keys would join.

To collect statistics about these data, we need access to those files stored in HDFS. It would be impractical to load the data to locally as Hadoop is design for processing large dataset that can reach the magnitude of Terabytes. Instead, we can make use of Pig. Since during `LOAD` (and `FOREACH`), we would be given information such as the path to the tables, the schema of the tables and aliases of each columns that we are interested in, we can simply make some template pig scripts to help us collect these statistics. This way, not only we are not putting huge stress onto the user's machine, we are also making use of infrastructure that is already in placed for large dataset analysis. Obviously, we need to recognised that collecting these statistics is sometime just as expensive as the queries that we are trying to optimised, and we would need to store these statistics into a cache for future use to avoid loading the histograms multiple times.

To achieve the collection of statistics, I have implemented a few pig script templates for collecting `maximum`, `minimum`, `row count`, `cardinality` and `histograms` for specific attributes in specific tables. When required, the file path, schema and required column would be sent to the statistic collection object, and if cache is not present, it would construct a temporary Pig script and submit the Pig job. The results would be sent back and stored in cache, and would be parsed and used in further analysis.

6.4.2 PigColumnMetadata

In order for the statistics we have collected to be used throughout the analysis, we need a data-structure that can store the data and provide easy manipulation. I have implemented the `PigColumnMetadata` class to accommodate this need. Apart from storing the metadata (statistics) about each columns, this class also implements many methods that would allow easy creation of new information about tables used at intermediate stage of the analysis, and they would be discussed below.

6.4.2.1 Equalities and Inequalities

It is very common in database queries that you would want to filter by some equalities or inequalities.For example, if you want a table that gives you all of the student whose scores are higher than or equal to 40, or you might want a table showing all of the students with fee status equal to "International". When this kind of queries occurs, I would like to slice the histograms so that future queries can also have accurate distribution of data.

For example, if we have a distribution of the `score` attribute, after the condition of >= 40 is applied, we should pass on a distribution similar to what we have on the right of table 6.1, so that the analysis of any future operators that depends on this `FILTER` will have this updated distribution to work with.

Histogram before FILTER	
Score	Frequency
0+	0
20+	10
40+	50
60+	300
80+	100

Histogram after FILTER	
Score	Frequency
40+	50
60+	300
80+	100

Table 6.1: Table illustrating effect of FILTER to a histogram

6.4.2.2 Uniformly Reduce

What happens to other attributes that is not in the `FILTER` condition? If we do not change and maintain the distribution of values of other attributes in the table, later operators would have a set of histograms that are uneven, such that each have a different number of rows, and is difficult to make sense of it.

Therefore after each operation to histogram for one attribute, it is necessary that we increase or decrease the row count of other attributes accordingly. The assumption we are going to make here is that each column would be independent to each other, and what it allows us to do is to keep the same distribution of other attributes and just reduce the frequencies of each bin accordingly.

There are a few ways we can perform the reduction (expansion), one is to calculate the ratio of reduction, and multiply the ratio to the frequencies at each bin. This is, however, problematic when the number is small. Since the type of frequency is integer, after multiplication, we can round, ceil or floor the result. Consider the case when all frequencies are 1, multiplying by the same ratio (that is smaller than 1) would result in either all frequencies being 0 or 1 depending how you covert the floating-point number to integer. And this means we either have a column with 0 rows or column with 10 rows, which are both wrong.

Another way that we can do it is to do some "dart throwing", where we randomly pick one bucket at a time (probability proportional to the size of frequency in the bucket), and decrease (increase) the frequency by one, until the the number of rows (total frequency) is the same as required. This is also problematic, mainly due to performances concern. For a "throw", we need to go through every buckets to pick the correct bucket, and we need to do the it again for all other "throws" because the probability changes after each throw. The iterations we need is:

$$\text{number of attributes} \times \text{number of rows to change} \times \text{number of bins}$$

If we are dealing with a table with millions of records, dozens of attributes, and we are cutting the table by half, you can see how much time we would be wasted on ”dart throwing”.

The solution I have thought of is a variation of the first approach. Instead of always rounding the floating point number, I would decide to ceil or floor the number with a probability equal to how the big decimal part of the number is.

$$decimal = x - \lfloor x \rfloor$$

$$x = \begin{cases} \lfloor x \rfloor & \text{if } r \geq decimal \\ \lceil x \rceil & \text{otherwise} \end{cases}, 0 \leq r \leq 1$$

In the above formulae, r is a random number generated between 0 and 1, the larger decimal part is, the higher the probability it would be rounded up, and vice versa. This way, we would in average achieve the correct amount of frequency and handle the edge cases of having small frequencies in some bins well.

6.4.2.3 Union, Intersect and Complement

Very often we would deal with a combination of boolean conditions joined together by `AND`, `OR` and `NOT`, therefore, we need to be able to combine the two independently filtered histograms, and estimate what would the distribution be when all conditions are applied.

For `Union`, I would match each bin on both side, and take the larger frequency of the same bin as demonstrated in table 6.2, and for Intersect, I would take the smaller value for each bin.

Before Union	
Score	Frequency
0+	0
20+	5
40+	10
60+	30
80+	10

Before Union	
Score	Frequency
0+	0
20+	10
40+	2
60+	40
80+	5

After Union	
Score	Frequency
0+	0
20+	10
40+	10
60+	40
80+	10

Table 6.2: Tables demonstrating the effect of Union

For `Complement`, I would take a histogram that was not altered by any operations as the universe and the histogram that is under a NOT operator and do subtraction on individual bins as demonstrated in table 6.3

6.4.3 Overall work-flow of the Model

The histogram information would be collected during the `FOREACH` immediately after `LOAD`, where Pig would assign schema information to the tables. When those informa-

Universe	
Score	Frequency
0+	0
20+	10
40+	50
60+	300
80+	100

Before Complement	
Score	Frequency
0+	0
20+	10
40+	2
60+	40
80+	5

After Complement	
Score	Frequency
0+	0
20+	0
40+	48
60+	260
80+	95

Table 6.3: Tables demonstrating the effect of Complement

tion are collected and stored, it would be stored in a map which map from attribute's id to the `PigColumnMetadata`. Every subsequent operators would make use of these `PigColumnMetadata` generated by previous operators and produce a set of `PigColumnMetadata` of their own which represents the distribution of attributes in it's output. Apart from calculating the distribution of output and sizes of output, each operators would also calculate and store the bandwidth cost. At the end, when all operators have been analysed, the total cost is calculated by summing the individual bandwidth cost of each operators.

6.4.4 LOLoad, LOForEach

The difference in this model about `LOLoad` and `LOForEach` is that we need to collect the relevant statistics at this stage. In Pig, if schema information is provided, an extra `FOREACH` stage would be added and the output would be the same with shcema information added. Therefore the `LOLoad` would remain mostly unchanged, and if the `LOForEach`'s predecessor is an `LOLoad`, then we would initiate the statistics collection process.

For the general case of `LOForEach`, we would need to look at the output schema, and copy over any histograms of existing attributes. For newly generated attributes, a new histogram is not calculated due to it's complexity. It can be the result of any user defined functions, and it is not possible in project to estimate them.

6.4.5 LOFilter

We need to evaluate the `filterPlan` inside the `LOFilter` in order to have an accurate estimation of the output. As discussed before, the `filterPlan` is an expression tree that represent some boolean values after evaluation.

To limit the scope of the project, I have imposed some restrictions in what for of conditions I can evaluate. For example condition such as

$$score + weight == 3 \times \text{house number}$$

would be legal in Pig, but to analyse it with a set of histograms and produce an estimated distribution after this condition is applied required mathematics way beyond the scope of this project. Although a possible extension to the project would be to find ways to approximate arbitrary conditions.

In this project, I would only analyse inequality/equality conditions that has one attribute on one side and a constant on the other. To analyse the `filterPlan`, we also have a similar structure to the `LogcalPlan`, such that we would be given an initial set of histograms, and each `LogicalExpressionOperators` within the `filterPlan` would make use of distributions from previous operators, and produce it's own estimation of distribution. There is also a separate map storing the literal values being used in the conditions. The `LogicalExpressionOperators` that we need to analyse are:

6.4.5.1 ProjectExpression

This indicates what attribute is concerned in the next operators, there is no need to modify any distribution so the output would be the same as input.

6.4.5.2 Binary Boolean Expressions

These operators includes `NotExpression`, `AndExpression` and `OrExpression`, as discussed previously, each would have two input histograms, and we would need to estimate the output distributions by combining the two input histograms. For `AND` conditions, we perform an intersect between the two input histograms, and union for `OR`. For `NOT`, we would perform a Complement against the unaltered histogram.

6.4.5.3 Equalities and Inequalities

These operators includes GreaterThan, LessThan, LessThan, Equal, Negative and etc. We would first confirm which side is the constant and which side it the `ProjectExpression`. After which we can use the built in methods in `PigColumnMetadata` to partition the histogram of the attribute in the condition, and reduce the histograms of other attributes uniformly.

6.4.5.4 Arithmetic Expressions

For these expressions, I would require both side to be a numeric value, and once the calculation is done, it would be stored in a map to be used in equality/inequality conditions.

6.4.5.5 RegexExpression

This represents the `MATCHES` condition in Pig. If we have a histogram for the strings, we would perform `Java`'s string matching to see what strings in the distribution matches, and produce a histogram with only those string that matches the regular expression.

6.4.5.6 Others

For any other operators, such as `UserFuncExpression`, I have decided to leave them out due to the scope of the project. For these operators, and cases that is not considered in other operators, such as inequality between two attributes, they would all reduce the input histograms by a fixed reduction ratio which would allow the analysis to be performed without actually implementing them.

6.4.5.7 Output

After the analysis of `filterPlan` is done, we would be given the output histograms of the `LOFilter` operator, and therefore it would be used as the output of `LOFilter`.

6.4.6 LOJoin

For `LOJoin`, we will first work out how the join keys would be joined together, i.e. how many would not be joined, how may are joined more than once and therefore we would have a mini Cartesian Product. Once the distribution of the join key are worked out, we would calculate how the number of rows has changed regarding both input tables, and uniformly expand(reduce) all other attributes.

For numerical histograms, we would assume uniform distribution in each bins to estimate how many would match. The formula we would use is

$$F_i^{new} = \frac{F_i^{left} \times F_i^{right}}{cardinality}$$

where F_i represents frequency at bin i, and cardinality would be the bin's range. This might be inaccurate if the value store is floating point numbers which means we might assumes a small cardinality and overestimated the frequency.

Before JOIN	
Score	Frequency
0+	20
20+	100
40+	0

Before JOIN	
Score	Frequency
0+	20
20+	40
40+	0

After JOIN	
Score	Frequency
0+	20
20+	200
40+	0

Table 6.4: Tables showing changes to join keys attributes

A problem is that the bin might not be aligned perfectly, therefore we need to regroup the histogram and make sure both side of the join has the same bin fragments. This can be achieved by assuming uniform distribution in each bin and cut a bin into smaller bins to match the bin of the other histogram.

For string attributes, it is much easier to match because we know exactly what values from both sides would matched, and we can therefore get a very accurate row counts from these joins.

If one side do not have histogram, then we cannot estimate each bin individually, and we would estimate the output row count of the entire attribute using the following formula [11]

$$F_{new} = \frac{F_{left} \times F_{right}}{max(Card_{left}, Card_{right})}$$

where F is the row count of tables and `Card` is the cardinality. Once the output row count is computed, all histograms would be reduced(expanded) by the ratio of F_{new}/F_{left} and F_{new}/F_{right}

For attributes that cardinality information is not available, we would assume it to be half of the row count.

In terms of the cost of Joins, Pig provides a few algorithms that effectively eliminates the reduce phase, if those are used, we would set the cost to 0, otherwise, we would estimate the cost by the sum of input sizes.

6.4.7 LODistinct

`LODistinct` only works on the whole tuple, so that a tuple would only be considered duplicate is every attribute has the same values as another tuple. It is difficult to calculate how many rows will remain after the operation because we do not have information about correlation between the columns, and if we start considering all combinations of all columns, the possible combinations would soon become astronomical, and might be way more than the number of rows that we have.

Since we cannot usually estimate how many unique tuples we are going to have directly, we can try using upper and lower bounds to get some idea. The lower bound would be the maximum cardinality among all attributes, because we must have at least that many unique rows sine one of the column has that many unique values. The upper bound is either the row count of the whole table, or the product of cardinality of all attributes, whichever is lower.

We can just pick the middle between the lower bound and the upper bound, but then we are not taking into account how large the number of possible combinations is, and it is reasonable to assume that the more possible combinations you can gave, the more likely you would have more distinct tuples in the output. We would need a function that would give us a value that gets closer to the upper bound when the number of possible combinations are larger.

We can make used of a `sigmoid function` as discussed in previous chapters.

$$S(t) = \frac{1}{1 + e^{-t}}$$

This function has the good property that the output will between 0 and 1 regardless of the input, and it will be closer to 1 as the input get larger, similar to the property that we need for output's cardinality approach the row count as the number of possible combinations gets larger. We can adapt the formula as:

$$range = rowCount - min$$
$$cardinality = min + \frac{1}{1 + e^{\frac{rowCount - combinations}{range}}} \times range$$

As we can see, the cardinality will be somewhere between `min` and `min` + `range`, and the larger the combinations is, the closer we get to `min` + `range`. The reason why we need to divide `rowCount` − `combinations` by `range` is that we need to scale the input to sigmoid function rather than using the absolute difference, otherwise a small difference between `combinations` and `rowCount` would make the output extreamely close to `rowCount`. This function also has nice property that when combinations is the same as `rowCount`, we get `min` + 0.5 × `range`, effectively taking the average.

The cost, as discussed previous chapters, is going to be the size of the input table as we are assuming many duplicate eliminations are performed at reduce nodes.

6.4.8 LOCogroup

For grouping, it is essential to work out the number of rows that would be in the output table, and this is determined the number of distinct values in the group keys. If we are grouping on one attribute, it is easy as we (usually) have a the information about attribute's cardinality. It is more difficult when we are grouping on multiple attributes, as it is difficult to estimate the cardinality of tuples of multiple attributes when we do not have information about the correlations between columns.

Notice this is the exact same problem we are facing with `LODistinct`, and we can approach it the same way. In the case of multiple attributes are used for grouping, we use the sigmoid function to approximate how many rows are actually unique.

One problem that we would face is that, although we can calculate how many rows we would have after the group, it is very difficult to keep track of the information any further. For the data being grouped, they are all stored in the `bag` data type, and since they are not simple values, it is very difficult to have a histogram for the distributions of `bags`. the same goes to the group key when we are grouping on multiple attributes, it would be difficult to have a histogram for tuples since we do not know the the actual distribution. This would cause information loss after grouping, and this might prevent accurate estimation in subsequent calculations.

6.4.9 LOCube

To estimate the number of rows in the output of `LOCube`, we need to consider the output of all combinations of attributes. For examples, if we have three attributes `(name, age,` gender), then the input will also be grouped by (, age, gender), (name, , gender), (name, age,), (name, ,), (, age,), (, gender) and (). This is a power-set of the set of attributes that we are grouping on, and it would have 2^n combinations. What we need to do is to generate a power-set of of these group attributes, and for each set in the power-set, we perform the same cardinality estimation using the sigmoid function like what we have done with group. Then we can sum up the cardinality of each combinations and use it as the row count of the output.

6.4.10 LOSplit, LOSplitOutput

Since LOSplit is just a place holder for LOSplitOutput, this operator will copy over all of the histograms for each input attributes. Sometimes the id of attributes would change at this stage, therefore we need to match the output attributes and input attributes by their alias.

LOSplitOutput is basically the same as LOFilter, therefore I have reused most of the implementation of LOFilter. Th only differences is that LOSplitOutput needs to change the uid of output attributes, and a map is provided.

6.4.11 LOUnion

Since Pig allows union to be performed on tables that do not share the same schemas, and would cause any subsequent operators to to have a schema, we will not be considering cases like these in out model, as attributes that contains multiple type cannot be reasonably analysed using out histogram methods.

Assuming the schema is the same, then we would merge the histogram information on each attributes. If both attributes have histograms, their bins would be split as we have done for LOJoin and the frequency at each new bin would be the sum frequencies at the corresponding bin of the input attributes.

6.4.12 LOSort

For LOSort, we do not need to do much about the output's distributions, as they would stay the same. The extra information that came from sorting is that we now know for sure whether some attributes in the output tables are ordered. This would have implication on LOLimit and Merge Join.

To accommodate this, there is a field in PigColumnMetadata that indicate whether this attribute is sorted, and it has three status: NONE, ASC and DESC which represents not sorted, sorted in ascending order and sorted in descending order respectively.

Apart from changing the orderedStatus of the PigColumnMetadata, all other data such as the histograms and cardinalities would remain unchanged and be passes onto subsequent operators.

6.4.13 LORank

As discussed in previous chapters, `LORank` would sort the table according to the attributes specified, and when none is specified, each row would be numbered by their original order. If those attributes are specified, it would reused the code from `LOSort`, and then it would add an extra histogram, where the type is always int, with minimum equal 1 and maximum equal to the row count of the input table. The cardinality equals to the cardinality of the attributes being grouped on and the histogram would be a uniformly distributed one.

6.4.14 LOLimit

In limit, we need to retrieve the number of limit, `n`. Then we would go through each attributes' histogram, if the attribute is not ordered, we would uniformly reduce it using the ratio `n/input rowCount`. If the attribute is ordered, we would pick out the top/bottom most `n` frequencies in the input histograms and store them into the ouput histograms depends on how the attribute is ordered.

6.4.15 LOCross

For `LOCross`, we need to multiply the rowCount of all input tables, and use it as the rowCount of the output table. Then we need to calculate the ratio of expansion on each table i using formula

$$ratio_i = rowCount_{output}/rowCount_i$$

then we would uniformly expand each attributes in each table by this ratio, and this would be the estimation of the output of `LOCross`.

6.5 Evaluation

6.5.1 FILTER

To see how effective the new algorithm can estimate size change for `FILTER`, I have conducted an experiment. I have written a few simple quires that consist of one `FILTER` operator, after which a `GROUP` if applied to the table produced by the `FILTER` so that I can measure the size of the result. I used `GROUP` because earlier experiment shows that the cost of a `GROUP` operator is almost about the size of the table. I also performed `GROUP` on the tables before they were passed to `FILTER`, and divide them to calculated the ratio of changes in table sizes after `FILTER`.

Then, I record how my model have estimated the ratio of size reduction, and the results are shown in table 6.5. All of the code for each test queries can be found in the Appendix.

The percentage shows how big the output is compare to the input, and 100% mean no changes at all and 0% means no rows has passed the filter conditions.

Query	Real Change	Estimated Change	Difference
filter1	100.00%	67.20%	32.80%
filter2	0.09%	1.35%	-1.26%
filter3	0.09%	0.40%	-0.30%
filter4	99.61%	99.60%	0.01%
filter5	90.04%	86.81%	3.24%
filter6	50.01%	33.59%	16.42%
filterNation	5.05%	4.00%	1.05%

Table 6.5: Percentage of output table as the input of FILTER operator

As we can see, most queries are estimated accurately, and the only one that is way off is `filter1`, which is a combinations of OR conditions, which results in 100% of tuples passing the condition, but since we evaluate each boolean operators individually, and assume all other attributes are reduced uniformly, it is difficult to correctly predict that all tuples would pass the conditions.

This is a problem in the design of the model as the assumption of independence between attributes are very unrealistic, but otherwise the complexity of estimation would be much larger than what we currently have in this model. Depending on what kind of quires are written more, this problem may or may not seriously affect the accuracy of this model. If most filter conditions are simple, then this would be a reasonable assumption to make, and it might not worth the resources to develop a more sophisticated model.

6.5.2 GROUP

To find out how well the `GROUP` estimation method works, I have conducted a small experiment. I have written a few queries that group on different tables with different attributes, after aggregation (`COUNT` in this case) I then `DUMP` the output and calculate the number of rows in the output.

These are then compared against the estimated row numbers after aggregation in my model, and the results are shows in table 6.6.

Query	Actual row	Estimated row	% Difference
CUSTOMER_all_ORDERS_all	1	1	0.00%
CUSTOMER_c_custkey_ORDERS_o_custkey	15000	15000	0.00%
ORDERS_o_custkey_o_orderpriority	46230	50000	-8.15%
PART_p_type_p_brand	3733	3750	-0.46%
NATION_all	1	1	0.00%
ORDERS_o_custkey	10000	10003	-0.03%
ORDERS_o_orderpriority	5	5	0.00%
PART_p_partkey	20000	20000	0.00%
PART_p_type	150	148	1.33%

Table 6.6: Number of rows in the output of `GROUP` and aggregation

In table 6.6, the first column shows the table we are grouping on in capital, and the keys of each table in lower case. As we can see, most of the estimation are very accurate, but it starts to to less accurate when we are dealing with more than one group key, which is expected as we are using the sigmoid functions to estimate the output row count.

One thing that we can do is that we can tweak the scaling of sigmoid function over time according how well it has predicted the number of output rows, if the queries is needed to be run multiple times.

6.5.3 JOIN

I have performed a similar experiment for the `JOIN` operator to test how well the JOIN analysis algorithms work. This is also done by counting the number or rows in the output of some queries that involves only one JOIN operator, then those numbers are compared against the output row count that has been estimated by the model.

Query	Actual row	Estimated row	% Diff
PART_PARTSUPP_SUPPLIER	8.00×10^4	1.09×10^5	-35.84%
SUPPLIER_NATION_REGION	1.00×10^3	1.31×10^3	-31.00%
PART_PARTSUPP_SUPPLIER_NATION	8.00×10^4	1.14×10^5	-42.41%
PART_PARTSUPP_SUPPLIER_NATION_REGION	8.00×10^4	1.42×10^5	-78.04%
nil_PART_PARTSUPP	0.00×10^0	2.00×10^4	N/A
non_key_PART_SUPPLIER	9.64×10^2	9.49×10^2	1.56%
non_key_PARTSUPP_NATION	1.60×10^3	2.02×10^3	-26.50%
non_foreign_PART_NATION	2.40×10^1	2.40×10^1	0.00%
non_foreign_PART_SUPPLIER	1.00×10^3	1.00×10^3	-0.20%
ORDERS_CUSTOMER	1.50×10^5	1.55×10^5	-3.11%
PART_LINEITEM	6.01×10^5	6.15×10^5	-2.33%
PART_PARTSUPP	8.00×10^4	8.18×10^4	-2.31%
PARTSUPP_LINEITEM	2.40×10^6	2.46×10^6	-2.36%
PARTSUPP_LINEITEM_PART	2.40×10^6	2.52×10^6	-4.96%
SUPPLIER_NATION	1.00×10^3	1.05×10^3	-4.80%

Table 6.7: number of rows in the output of `JOIN`

From the table above, the top four queries are joins that involves different foreign keys, i.e. `part` joins with `partsupp` using `partkey`, `partsupp` to `supplier` using `suppkey` and etc. The queries with prefix `non_key` means they were joined not using keys of wither table, and `non_foreign` means they were join on keys of each table but those keys are not in a foreign key relationship and the one with `nil` prefix is one that bears no relationship at all and know to produce 0 output. All other queries are join on foreign keys.

As we can see, when joining two tables on foreign keys, the model is working very well, it also work well when we are joining two tables on non-foreign keys. The reason why it starts to diverge when joining on more than two tables is that we are assuming the distribution of attributes other than the join keys to be the same after the join, but this is probably not the case in the real-world.

Take the first query in the table as an example, when we joined both `part` and `partsupp`on `partkey`, we are keeping the distribution of `suppkey` in the result of `JOIN` the same, and then the subsequent `JOIN` with `supplier` would be based on a wrong distribution of `suppkey`.

6.5.4 Row Count

Although estimating the row count of final output table not the most important performance measure in this project, it would be interesting to know how well the model can estimate the number of rows in the output table of each query.

I have counted the number of rows each `TPC-H` queries would project and compared them against the row count estimated by the model as shown in table 6.8.

Query	Estimated Rows	Real Rows	Diff
Q1	6	5	1
Q2	8019	101	7918
Q3	78697	11	78686
Q4	1	6	-5
Q5	9	6	3
Q6	1	2	-1
Q7	1	5	-4
Q8	976	3	973
Q9	69708794	176	69708618
Q10	117264	21	117243
Q11	16002	1049	14953
Q12	7	3	4
Q13	12500	43	12457
Q14	1	2	-1
Q15	4996	2	4994
Q16	187500	18315	169185
Q17	1	2	-1
Q18	2	58	-56
Q19	1	2	-1
Q20	1	187	-186
Q21	1	101	-100
Q22	175274	8	175266

Table 6.8: Comparison of the number of rows in the output between estimated and actual

As we can see from table 6.8, about half of the estimations are quite accurate, with a difference form actual row count being lower than 5, this shows that for these queries, the model can correctly understand most of the operators and filter conditions. However, many other estimations are off by a huge margin, these are mainly caused by UDFs in filter conditions or newly generated attributes.

In the Histogram Model, there is no distribution information available for attributes that is generated during the execution of the script, for example, if we look at the snippet from Q3 below

```
...
sell1 = foreach l1 generate l_orderkey,
l_extendedprice*(1-l_discount) as volume
...
```

the new column volume is formed by arithmetic combination of l_extendedprice and l_discount. In this model, I have not implemented functions to estimate the distributions of these attributes, as the mathematics is beyond the scope of this project. For these columns, as well as any columns that is selected using User Defined Functions, are approximated by taking a fixed size reduction. Which is why the final row count is very far from the actual number.

6.5.5 Overall

Again, we are going to look at our benchmark, how well the model can rank the cost of TPCH queries, we are going to do the same measurement as we have done for the Simple Model, we would measure the bytes being sent from map nodes to reduce nodes during the execution of each TPC-H quires and rank them according to the number of bytes costs, and compare that against the ranking of cost estimated by the model, and the result is shown in table 6.9.

We can see from table 6.9 that the Histogram Model is performing well and is better than the Simple Model. The Spearman's Rank Correlation Coefficient has improved from 0.74 in the Simple Model to 0.78. We can also see that there are only three big differences in ranking this time, i.e. Q1, Q12 and Q16. Q1 and Q12 are already the outliers in the Simple Model, and the difference is that Q12 has been underestimated in the Histogram Model instead of being overestimated.

We will now have a closer look at each queries and discuss why they are not working so well under out Histogram Model.

6.5.5.1 Q1

```
SubLineItems = FILTER LineItems BY shipdate <= '1998-09-02';
StatusGroup = GROUP SubLine BY (returnflag, linestatus);
PriceSummary = FOREACH StatusGroup GENERATE ...
SortedSummary = ORDER PriceSummary BY returnflag, linestatus;
```

The reason why Q1 is over estimated is the same as in the Simple Mode, because we have not anticipated the optimisation of the Combiner, which in this particular case had reduced the costs by order of magnitudes.

Query	Est. Cost	Est. Rank	Measured Cost	Measured Rank	d	dˆ2
Q1	5.35×10^8	14	5.10×10^3	2	12	144
Q2	2.72×10^8	11	9.57×10^7	12	-1	1
Q3	1.20×10^8	5	2.29×10^7	5	0	0
Q4	2.60×10^8	10	3.49×10^7	9	1	1
Q5	3.19×10^7	3	2.54×10^7	6	-3	9
Q6	4.87×10^8	13	2.89×10^7	8	5	25
Q7	1.58×10^7	2	2.01×10^6	3	-1	1
Q8	2.24×10^8	8	2.74×10^7	7	1	1
Q9	7.60×10^8	16	1.74×10^8	15	1	1
Q10	9.56×10^8	20	2.00×10^8	18	2	4
Q11	7.84×10^8	17	3.69×10^8	21	-4	16
Q12	2.33×10^8	9	1.98×10^8	17	-8	64
Q13	3.94×10^8	12	4.92×10^7	11	1	1
Q14	8.44×10^8	18	2.36×10^8	19	-1	1
Q15	1.86×10^8	7	1.72×10^7	4	3	9
Q16	1.45×10^8	6	1.18×10^8	14	-8	64
Q17	4.15×10^7	4	4.87×10^7	10	-6	36
Q18	1.66×10^9	21	2.66×10^8	20	1	1
Q19	6.24×10^6	1	1.38×10^2	1	0	0
Q20	6.21×10^8	15	1.16×10^8	13	2	4
Q21	9.03×10^8	19	1.85×10^8	16	3	9
Q22	7.98×10^0	22	3.81×10^8	22	0	0
					coefficient	0.78

Table 6.9: Result of Histogram Model, cost in byte

6.5.5.2 Q12

```
flineitem = FILTER lineitem BY l_shipmode MATCHES 'MAIL|SHIP' AND
                   l_commitdate < l_receiptdate AND l_shipdate < l_commitdate AND
                   l_receiptdate >= '1994-01-01' AND l_receiptdate < '1995-01-01';

l1 = JOIN flineitem BY .......
```

For `Q12`, which is underestimated in this model, we can see that a very specific filter condition is applied right at the start of the query. One property of the Histogram Model is that earlier operators tends to be more accurate, because this is when we have the most accurate information. Through the analysis, more assumptions would be made, and the distribution information we have would tends to get more and more inaccurate and hence the estimations would be inaccurate. Since not many other queries has such a specific filter at the start, later filter tends to be approximates less accurately, and therefore this is underestimated compare to other queries where `FILTERs` occurring later in the script caused the reduction rate to be less extreme.

6.5.5.3 Q16

```
fsupplier = FILTER supplier BY (NOT s_comment MATCHES '.*Customer.*Complaints.*');
.....
```

Similar to Q12, Q16 also has a very specific filter condition at the start. Also, contrary to many other FILTER conditions, this one is properly handled by our model, such that it does not contain UDF and is the comparison between a column and a literal. Since this can be estimated reasonably well, it might causes this query to be underestimated compare to others.

6.6 Discussion

Although this model only work slightly better than the Simple Model, there are potentials for this model to work very well. We have seen that most of our concepts are correct, this can be shown in those small experiments on FILTER, JOIN and GROUP, and the main factor that affect the accuracy is that there are too many limitations set in this model. Many FILTER conditions are actually done with UDF, which is ignored in this model. The information loss after applying the GROUP operator means our model cannot properly asses subsequent queries well. And the assumption of individual attributes being independent in tables also hurt the accuracy especial when the query is long.

Since all of the operators are analysed using statistics, an important feature of this model is that one improvement in one operator has the potential of improving the accuracy of this model by a large margin. This is because analysis of each operator depends heavily on distributions generated by previous operators, and therefore if some distributions from previous operators are more accurate, the subsequent operators can be improves as well. If we compare to the Simple Model, even if we have made the analysis of one operator much more accurate, other operators are still relying on these unrealistic assumptions which makes it difficult to improve.

This allows the extension and improvement of the model to be made easily, which, on top of it's already good performance, can serve as a good basis of any future more sophisticated plans.

In conclusion, this model has adopted a more scientific approach, and relied on statistics of the input table, although many features in Pig is still not supported, the model is still working well under these restrictions. If some improvement can be added, such as recognition of some UDF's, then the accuracy of the model should be improved.

Chapter 7

Applying the Histogram Model to Bigger Data

7.1 Motivation

As we have discussed in the introductory chapters, Pig and Hadoop are tools that are designed to work with large dataset, and therefore any cost model of Pig should in principle be able to work well with large datasets. Throughout this project, we have been testing my models with modest dataset sizes, mostly 100 MB and 1 GB due to the limitation of disk and computing power available, and we have seen that the models are showing promising results. We can reasonably assumes that the Histogram Model can work just as well when the distribution informations about the large dataset are given, since most of the estimations made in that model are based on the distributions of values within the dataset. However, a big challenge we would face is that, as mentioned previously, the queries of collecting the distributions of values are not cheap.

```
strip = FOREACH $table GENERATE $col;
dist = DISTINCT strip;
card = FOREACH (GROUP dist ALL) GENERATE MIN(dist.$col),
MAX(dist.$col), COUNT(dist);
DUMP card;
```

For a typical attribute, I would first find out it's minimum, maximum and cardinality, which is implemented by selecting the distinct values in that particular attribute them perform a group and aggregate to work out the three values, as shown in the script above. This itself involves two reduce phase, since the `GROUP` can be easily optimised by the `Combiner` since they are all algebraic function, we can say all of the data in the attribute has been sent through the cluster once.

```
strip = FOREACH $table GENERATE $col;
bucket_vals = foreach strip generate
```

```
$min + (($max- $min) / $buckets)*(((int)$col- $min) * $buckets / ($max- $min + 1))
as bucket, $col;
group_by_bucket = group bucket_vals by bucket;
bin_id = foreach group_by_bucket generate group, COUNT(bucket_vals.$col);
DUMP bin_id;
```

After these informations are generated, the model would then proceed to collect the histogram, which is done by annotating each row by their bin and then group on the bin and count the rows in each bin, as shown in the script above. This is again another reduce phase and in total all the values in each attribute are sent through the cluster twice.

Above analysis shows us how expensive these operations are, although in this project I have implemented a cache, that might not be very helpful if the data being analyses are different every times. If we think about a typical use case of Hadoop which is web log processing, it is likely that we have slightly different distribution of data every day, otherwise why bother analysing everyday.

One approach that can be used for solving this problem is that we can try to collect the distribution about a fraction of the data and try to make estimations with those sample distribution instead of the actual distribution. This would allow the model to work with relevant distributions and at the same time keep down the cost of pre-processing.

7.2 Implementation

To test whether the Histogram Model can work well in this setting, I have done the following.

1. Generate a 10 GB `TPC-H` dataset and a 1 GB `TPC-H` dataset and upload both to HDFS.
2. Collect metadata about the 1GB dataset.
3. Modify the code so that when a distribution is loaded in Java, it will be scaled up by 10. (Multiply each frequencies by 10 and in the case of numeric histograms, extend the range of bins by 10).
4. Analysis each `TPC-H` queries with the scaled distributions.

7.3 Results

To test how well the model work in this setting, we let our model to run on all of the `TPC-H` queries, and we would collect the estimated cost of each. After we have the real costs and estimated costs generated with the scaled distributions, we will rank each queries according to the cost then use Spearman's Rank Correlation Coefficient to score the performance of the model's ability to rank the model by the cost and the results are presented in table 7.1.

Query	Est. Cost	Est. Rank	Measured Cost	Measured Rank	d	d^2
Q1	5.35×10^9	12	5.01×10^4	2	10	100
Q2	4.76×10^9	11	1.99×10^9	17	-6	36
Q3	1.45×10^9	6	1.18×10^9	14	-8	64
Q4	4.21×10^8	4	4.87×10^8	10	-6	36
Q5	1.46×10^{10}	21	2.66×10^9	20	1	1
Q6	1.40×10^7	1	1.33×10^3	1	0	0
Q7	6.35×10^9	14	1.16×10^9	13	1	1
Q8	9.05×10^9	19	1.93×10^9	16	3	9
Q9	8.84×10^{18}	22	3.92×10^9	22	0	0
Q10	8.34×10^9	17	9.72×10^8	12	5	25
Q11	1.27×10^9	5	2.32×10^8	5	0	0
Q12	2.60×10^9	9	3.49×10^8	9	0	0
Q13	3.19×10^8	3	2.54×10^8	6	-3	9
Q14	5.94×10^9	13	2.89×10^8	8	5	25
Q15	1.66×10^8	2	4.72×10^7	3	-1	1
Q16	2.24×10^9	8	2.75×10^8	7	1	1
Q17	7.60×10^9	15	1.74×10^9	15	0	0
Q18	9.56×10^9	20	2.01×10^9	18	2	4
Q19	7.84×10^9	16	3.69×10^9	21	-5	25
Q20	3.94×10^9	10	5.46×10^8	11	-1	1
Q21	8.44×10^9	18	2.44×10^9	19	-1	1
Q22	1.88×10^9	7	1.72×10^8	4	3	9
					coefficient	0.80

Table 7.1: Result of Histogram Model using the data of a 1 GB dataset to estimate query cost of a 10 GB dataset.

As we can see from the result shown in table 7.1, this model is working well. The Spearman's Rank Correlation Coefficient in this case is actually higher than when we estimate a cost against a 1 GB dataset with actual distribution collected on those data.

Is this evidence that the model actually works better when using small sample distributions to estimate the cost of queries on larger dataset? I would argue this is not the case. One possible explanation is that according to `TPC-H` specification[20], many attributes in the table are randomly generated with a uniform distribution, and this matches the assumptions in our model. Therefore when we scale the distributions by 10, it is still providing an accurate representation. To cause the model to work better with 10 GB, it could be due to the fact that the more data is generated randomly, the closer the data is to the intended distribution. If we are generating 100 numbers in the range of 1 to 100, it is very likely that not all number is generated exactly once, if we generate 50,000 rows between 1 an 100, we are very likely to get a frequency around 500 for each bucket form 1 to 100. Since the columns in the 10 GB dataset is likely to be closer to a uniform distributed than those of 1 GB dataset, out model would match the dataset more.

Since we are still having a relatively accurate distribution, and the real distribution in the 10 GB dataset is closer to what we have assumed in the model, it is not too much a surprise our model will be working better in this setting.

Although this scenario is perhaps unique to testing on benchmarks, we can still deduce useful conclusions from it. This experiment demonstrates that if the sample distribution is representative of the actual dataset, the model, should be performing close to the level of having an set of real distributions, and therefore the model would be useful when it comes to estimating cost of queries that analyse large dataset, provided that a good sampling technique is used.

Chapter 8

Future works

8.1 Type sizes

Type	Size (bits)
BOOLEAN	1
BYTE	8
INTEGER	32
LONG	64
FLOAT	32
DOUBLE	64
DATETIME	24 * 8
BYTEARRAY	128 * 8
CHARARRAY	256 * 8
MAP	4096 * 8
TUPLE	64 * 2
BAG	1024 * 16 * 8

Table 8.1: Estimation of type size

This is a table that is heavily used in the Simple Model, for estimating the size of a particular tuple according to it's schema, and it's accuracy is therefore vital to the performance of the Simple Model. Many assumptions are made here, a part from the dynamic types, such as `CHARARRAY` and `BAG`, have not been able to verify whether the size information for the primitive types are correct.

For example, what Java types are actually used for BOOLEAN, and whether the serialisation process actually convert a into 1 bit.

Another problem related to this is that I have been using the file size as the size of input tables, and this very often leads to the actual cost it has estimated to be less helpful as the size in memory and through the network might be very different, i.e. a more efficient types are used.

If some work can be done on working out hoe Pig does the serialisation, and the ratio input file and their in-memory and serialised size would be a great improvement to the project.

8.2 Histogram of bags and tuples

In the Histogram Model, I have decided not to keep track of any information about the bags, because the distribution of bags is hard to represent with a traditional histogram. Perhaps some future projects can focus on designing ways to estimate the distributions within each bags and to keep track of information about the bags that can accommodate the aggregations after grouping.

This might be achieved by some recursive data structures that can represent distribution of arbitrary types.

8.3 UDF

A feature of Pig is that user can make use of `User Defined Functions` in Pig. These functions are written in Java and can provide many functionalities that is not easily implemented in Pig. During this project, it has been ignored to keep the scope of the project in check, but it is still very important to be able to work with these `UDF` so that we can analyse all the details in a script.

Although it would not be possible to understand all `UDFs`, perhaps a project can be done on implementing the analysis a set of `UDFs` that is commonly used in Pig, such as`MAX`, and test how much more accurate the model would have become if those `UDFs` can be analysed.

8.4 Combiner and MapReduce Plan

`Combiner` can optimise the cost of Pig queries a lot as demonstrated in previous chapters, and is currently not considered in the model. If a model can be developed on MapReduce plan, that includes information about Combiners, this might have the potential of improving the accuracy of the estimations.

Chapter 9

Conclusions

I have started the project as a initial attempt to bring cost based optimisation into Pig by developing cost models that estimate Pig queries cost in terms of bandwidth used. Throughout the project we have explored two very different approached in doing cost analysis, one through assumptions that do not depend on the structure of the input at all, and another that looks at distributions of individual attributes of input tables.

As expected, both methods has shown promises and are working well in terms of the test and benchmark we have set, and the more sophisticated Histogram model does work better than the Simple Model when it comes to ranking queries by their costs.

While the performance of individual models are evaluated in previous chapters, we now need to have a look at both models together and compare their strength and weaknesses. We have to realise that the improvement of the Histogram Model over the Simple model is only modest. With the cost of running an expensive query to gather all the information needed for the query to run, one might want to ask the question of whether it is worth using the Histogram model at all.

The answer is that it depends. If we were going to develop an commercial cost based query optimisation engine base on a cost model, I would argue that the approach similar to the Histogram Model is definitely needed. The model developed in this project is not performing to it's potential as it cannot understand all forms of queries permitted by Pig. Once features such as keeping track of the information in bags after `GROUP` and predicting the effect of `Combiner` are added, the Histogram Model would work much better.

On the other hand, if what we need i a quick indication of what that cost of our query is going to be, or which alternative ways to write the query is better, we can make use of the quick Simple Model as a guidance that helps Pig developments.

In conclusion, this project has built a solid foundation of Pig query cost model, It has looked at many options when facing different problems in cost estimation and has demonstrate the strength and weaknesses in many different approaches. This project is a good first step into building an automatic Pig query optimisation system, and better models can be easily build based on the two models developed during this project if they can tackle come of the know weaknesses.

Bibliography

[1] CERN. Computing: Experiments at CERN generate colossal amounts of data. The Data Centre stores it, and sends it around the world for analysis. http://home.cern/about/computing, 2016. [Online; accessed 28-01-2016].

[2] Andrew Lampitt. The real story of how big data analytics helped Obama win. http://www.infoworld.com/article/2613587/big-data/the-real-story-of - how-big-data-analytics-helped-obama-win.html, 2013. [Online; accessed 28-01-2016].

[3] Alan Gates. Comparing Pig Latin and SQL for Constructing Data Processing Pipelines. https://developer.yahoo.com/blogs/hadoop/comparing-pig-latin-sql-constructing-data-processing-pipelines-444.html, 2010. [Online; accessed 28-05-2016].

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[5] The Apache Software Foundation. PoweredBy. http://wiki.apache.org/hadoop/PoweredBy, 2016. [Online; accessed 28-01-2016].

[6] The Apache Software Foundation. Welcome to Apache Pig! https://pig.apache.org, 2015. [Online; accessed 28-01-2016].

[7] Computer Sciences Department at University of Wisconsin-Madison. HTCondor Team. Computing with HTCondor. https://research.cs.wisc.edu/htcondor/index.html, 2016. [Online; accessed 28-01-2016].

[8] The Apache Software Foundation. Apache Hadoop YARN. https://pig.apache.org, 2016. [Online; accessed 28-01-2016].

[9] The Apache Software Foundation. What Is Apache Hadoop? https://hadoop.apache.org, 2016. [Online; accessed 28-01-2016].

[10] Dhruba Borthakur. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2013. [Online; accessed 28-01-2016].

[11] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 9780131873254.

[12] 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*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376726. URL `http://doi.acm.org/10.1145/1376616.1376726`.

[13] Alan Gates. *Programming Pig*. O'Reilly Media, Inc., 1st edition, 2011. ISBN 1449302645, 9781449302641.

[14] The Apache Software Foundation. Cloud Stack. `https://cloudstack.apache.org`, 2016. [Online; accessed 10-06-2016].

[15] Imperial College Computing Support Group, Computing. CSG Cloud. `https://www.doc.ic.ac.uk/csg/services/cloud`, 2016. [Online; accessed 10-06-2016].

[16] Imperial College ICT. Imperial College VPN. `https://www.imperial.ac.uk/admin-services/ict/self-service/connect-communicate/remote-access/method/set-up-vpn/`, 2016. [Online; accessed 10-06-2016].

[17] tpc.org. TPC. `http://www.tpc.org/default.asp`, 2016. [Online; accessed 28-01-2016].

[18] Jie Li. PIG-2397: Running TPC-H Benchmark on Pig. `https://issues.apache.org/jira/browse/PIG-2397`, 2011. [Online; accessed 10-06-2016].

[19] The Apache Software Foundation. PigExecutionModel. `https://wiki.apache.org/pig/PigExecutionModel`, 2009. [Online; accessed 22-05-2016].

[20] Transaction Processing Performance Council. TPC BENCHMARKTM H, Standard Specification. `http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v2.17.1.pdf`, 2014. [Online; accessed 28-05-2016].

[21] The Apache Software Foundation. Pig Latin Basics. `http://pig.apache.org/docs/r0.12.0/basic.html`, 2013. [Online; accessed 24-05-2016].

Chapter 10

Appendix

10.1 TPC-H in Pig

All of the Pig code below are written by Jie Li during his time at Horton Work, details can be found at https://issues.apache.org/jira/browse/PIG-2397. Only input I have is to modify the input and output to suit the need of my project, and I will list these code for referencing purpose only.

10.1.1 Q1

```
LineItems = LOAD '$input/lineitem.tbl' USING PigStorage('|') AS
      ( orderkey: int,
    partkey: int,
    suppkey: int,
    linenumber: int,
    quantity: double,
    extendedprice: double,
    discount: double,
    tax: double,
    returnflag: chararray,
    linestatus: chararray,
    shipdate: chararray,
    commitdate: chararray,
    receiptdate: chararray,
    shipinstruct: chararray,
    shipmode: chararray,
    comment: chararray);
SubLineItems = FILTER LineItems BY shipdate <= '1998-09-02';

SubLine = FOREACH SubLineItems GENERATE returnflag, linestatus,
   quantity,
extendedprice , extendedprice*(1-discount) AS disc_price ,
 extendedprice*(1-discount)*(1+tax) AS charge , discount;

StatusGroup = GROUP SubLine BY (returnflag , linestatus);
PriceSummary = FOREACH StatusGroup GENERATE group.returnflag AS
    returnflag ,
group.linestatus AS linestatus , SUM(SubLine.quantity) AS sum_qty
    ,
SUM(SubLine.extendedprice) AS sum_base_price , SUM(SubLine.
    disc_price) a
s sum_disc_price , SUM(SubLine.charge) as sum_charge , AVG(SubLine
    .quantity)
as avg_qty , AVG(SubLine.extendedprice) as avg_price , AVG(SubLine
    .discount)
as avg_disc , COUNT(SubLine) as count_order;
SortedSummary = ORDER PriceSummary BY returnflag , linestatus;
-- STORE SortedSummary INTO '$output/Q1out';
DUMP SortedSummary;
```

10.1.2 Q2

```
Part = load '$input/part.tbl' USING PigStorage('|') as (
    p_partkey:long ,
 p_name:chararray , p_mfgr:chararray , p_brand:chararray , p_type:
     chararray ,
 p_size:long , p_container:chararray , p_retailprice:double ,
     p_comment:chararray);

Supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
    s_suppkey:long ,
s_name:chararray , s_address:chararray , s_nationkey:int , s_phone:
    chararray ,
s_acctbal:double , s_comment:chararray);

Partsupp = load '$input/partsupp.tbl' USING PigStorage('|') as (
    ps_partkey:long ,
 ps_suppkey:long , ps_availqty:long , ps_supplycost:double ,
 ps_comment:chararray);

Nation = load '$input/nation.tbl' USING PigStorage('|') as (
    n_nationkey:int ,
 n_name:chararray , n_regionkey:int , n_comment:chararray);

Region = load '$input/region.tbl' USING PigStorage('|') as (
    r_regionkey:int ,
r_name:chararray , r_comment:chararray);
```

```
FRegion = filter Region by r_name == 'EUROPE';
FR_N = join FRegion BY r_regionkey, Nation BY n_regionkey;
FR_N_S = join FR_N BY n_nationkey, Supplier BY s_nationkey;
FR_N_S_PS = join FR_N_S BY s_suppkey, Partsupp BY ps_suppkey;

FPart = filter Part by p_size == 15 and p_type matches '.*BRASS
    ';
FR_N_S_PS_FP = join FR_N_S_PS by ps_partkey, FPart by p_partkey;

G1 = group FR_N_S_PS_FP by ps_partkey;
Min = FOREACH G1 GENERATE flatten(FR_N_S_PS_FP),
MIN(FR_N_S_PS_FP.ps_supplycost) as min_ps_supplycost;
MinCost = filter Min by ps_supplycost == min_ps_supplycost;

RawResults = foreach MinCost generate s_acctbal,
          s_name,
          n_name,
          p_partkey,
          p_mfgr,
          s_address,
          s_phone,
          s_comment;
SortedMinimumCostSupplier = ORDER RawResults BY s_acctbal DESC,
    n_name, s_name, p_partkey;

HundredMinimumCostSupplier = LIMIT SortedMinimumCostSupplier
    100;

DUMP HundredMinimumCostSupplier;
```

10.1.3 Q3

```
customer = load '$input/customer.tbl' USING PigStorage('|') as (
    c_custkey:long,
c_name:chararray, c_address:chararray, c_nationkey:int, c_phone:
    chararray,
c_acctbal:double, c_mktsegment:chararray, c_comment:chararray);

orders = load '$input/orders.tbl' USING PigStorage('|') as (
    o_orderkey:long,
o_custkey:long, o_orderstatus:chararray, o_totalprice:double,
o_orderdate:chararray, o_orderpriority:chararray, o_clerk:
    chararray,
o_shippriority:long, o_comment:chararray);
```

```
lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
    l_orderkey:long,
l_partkey:long, l_suppkey:long, l_linenumber:long, l_quantity:
    double,
l_extendedprice:double, l_discount:double, l_tax:double,
    l_returnflag:chararray,
l_linestatus:chararray, l_shipdate:chararray, l_commitdate:
    chararray,
 l_receiptdate:chararray,l_shippingstruct:chararray, l_shipmode:
     chararray,
  l_comment:chararray);


fcustomer = filter customer by c_mktsegment == 'BUILDING';
forders = filter orders by o_orderdate < '1995-03-15';
flineitem = filter lineitem by l_shipdate > '1995-03-15';

o1 = join forders by o_custkey, fcustomer by c_custkey;
selo1 = foreach o1 generate o_orderkey, o_orderdate,
    o_shippriority;

l1 = join flineitem by l_orderkey, selo1 by o_orderkey;
sell1 = foreach l1 generate l_orderkey, l_extendedprice*(1-
    l_discount) as volume,
o_orderdate, o_shippriority;

grResult = group sell1 by (l_orderkey, o_orderdate,
    o_shippriority);
sumResult = foreach grResult generate flatten(group), SUM(sell1.
    volume) as revenue;
sortResult = order sumResult by revenue desc, o_orderdate;
limitResult = limit sortResult 10;

DUMP limitResult;
```

10.1.4 Q4

```
Orders = LOAD '$input/orders.tbl' USING PigStorage('|') AS (
    o_orderkey: int,
     o_custkey: int,
     o_orderstatus: chararray,
     o_totalprice: double,
     o_orderdate: chararray,
     o_orderpriority: chararray,
     o_clerk: chararray,
     o_shippriority: int,
    o_comment: chararray);
LineItem = LOAD '$input/lineitem.tbl' USING PigStorage('|') AS (
    l_orderkey: int,
    l_partkey: int,
    l_suppkey: int,
    l_linenumber: int,
    l_quantity: double,
    l_extendedprice: double,
    l_discount: double,
    l_tax: double,
    l_returnflag: chararray,
    l_linestatus: chararray,
    l_shipdate: chararray,
    l_commitdate: chararray,
    l_receiptdate: chararray,
    l_shipinstruct: chararray,
    l_shipmode: chararray,
    l_comment: chararray);
DateFilter = FILTER Orders BY o_orderdate >= '1993-07-01'
and o_orderdate < '1993-10-01';
CommitDateFilter0 = FILTER LineItem BY l_commitdate <
   l_receiptdate;
CommitDateFilter = FOREACH CommitDateFilter0 generate l_orderkey
   ;

COG1 = COGROUP DateFilter BY o_orderkey, CommitDateFilter BY
   l_orderkey;
Fil1 = FILTER COG1 by COUNT(CommitDateFilter) > 0;

OrdersCount = FOREACH Fil1 GENERATE FLATTEN(DateFilter);
PriorityGroup = GROUP OrdersCount BY o_orderpriority;
PriorityChecking = FOREACH PriorityGroup GENERATE group, COUNT(
   OrdersCount) AS order_count;

SortedPriorityChecking = ORDER PriorityChecking BY group;
DUMP SortedPriorityChecking;
```

10.1.5 Q5

```
customer = load '$input/customer.tbl' USING PigStorage('|') as (
   c_custkey:long,c_name:chararray, c_address:chararray,
   c_nationkey:int, c_phone:chararray, c_acctbal:double,
   c_mktsegment:chararray, c_comment:chararray);

orders = load '$input/orders.tbl' USING PigStorage('|') as (
   o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
    o_totalprice:double, o_orderdate:chararray, o_orderpriority:
    chararray, o_clerk:chararray, o_shippriority:long, o_comment:
    chararray);
verbatim
lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
    l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
    :long, l_quantity:double, l_extendedprice:double, l_discount:
    double, l_tax:double, l_returnflag:chararray, l_linestatus:
    chararray, l_shipdate:chararray, l_commitdate:chararray,
    l_receiptdate:chararray,l_shippingstruct:chararray,
    l_shipmode:chararray, l_comment:chararray);

supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
    s_suppkey:long, s_name:chararray, s_address:chararray,
    s_nationkey:int, s_phone:chararray, s_acctbal:double,
    s_comment:chararray);

nation = load '$input/nation.tbl' USING PigStorage('|') as (
    n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment
    :chararray);

region = load '$input/region.tbl' USING PigStorage('|') as (
    r_regionkey:int, r_name:chararray, r_comment:chararray);

fregion = filter region by r_name == 'ASIA';
forders = filter orders by o_orderdate < '1995-01-01' and
    o_orderdate >= '1994-01-01';

n1 = join nation by n_regionkey, fregion by r_regionkey;
seln1 = foreach n1 generate n_name, n_nationkey;

s1 = join supplier by s_nationkey, seln1 by n_nationkey;
sels1 = foreach s1 generate n_name, s_suppkey, s_nationkey;

l1 = join lineitem by l_suppkey, sels1 by s_suppkey;
sell1 = foreach l1 generate n_name, l_extendedprice, l_discount,
     l_orderkey, s_nationkey;

o1 = join forders by o_orderkey, sell1 by l_orderkey;
selo1 = foreach o1 generate n_name, l_extendedprice, l_discount,
     s_nationkey, o_custkey;

c1 = join customer by (c_custkey, c_nationkey), selo1 by (
    o_custkey, s_nationkey);
selc1 = foreach c1 generate n_name, l_extendedprice * (1 -
    l_discount) as eachvalue;
```

```
grResult = group selc1 by n_name;
sumResult = foreach grResult generate flatten(group), SUM(selc1.
   eachvalue) as revenue;
sortResult = order sumResult by revenue desc;

DUMP sortResult;
```

10.1.6 Q6

```
lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
     l_orderkey: int,
     l_partkey: int,
     l_suppkey: int,
     l_linenumber: int,
     l_quantity: double,
     l_extendedprice: double,
     l_discount: double,
     l_tax: double,
     l_returnflag: chararray,
     l_linestatus: chararray,
     l_shipdate: chararray,
     l_commitdate: chararray,
     l_receiptdate: chararray,
     l_shipinstruct: chararray,
     l_shipmode: chararray,
     l_comment: chararray);

flineitem = FILTER lineitem BY l_shipdate >= '1994-01-01' AND
   l_shipdate < '1995-01-01' AND l_discount >= 0.05  AND
   l_discount <= 0.07 AND l_quantity < 24;

saving = FOREACH flineitem GENERATE l_extendedprice * l_discount
   ;
grpResult = GROUP saving ALL;
sumResult = FOREACH grpResult GENERATE SUM(saving);

DUMP sumResult;
```

10.1.7 Q7

```
supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
   s_suppkey:long, s_name:chararray, s_address:chararray,
   s_nationkey:int, s_phone:chararray, s_acctbal:double,
   s_comment:chararray);
```

```
lineitem0 = load '$input/lineitem.tbl' USING PigStorage('|') as
   (l_orderkey:long, l_partkey:long, l_suppkey:long,
   l_linenumber:long, l_quantity:double, l_extendedprice:double,
    l_discount:double, l_tax:double, l_returnflag:chararray,
   l_linestatus:chararray, l_shipdate:chararray, l_commitdate:
   chararray, l_receiptdate:chararray,l_shippingstruct:chararray
   , l_shipmode:chararray, l_comment:chararray);

orders = load '$input/orders.tbl' USING PigStorage('|') as (
   o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
   o_totalprice:double, o_orderdate:chararray, o_orderpriority:
   chararray, o_clerk:chararray, o_shippriority:long, o_comment:
   chararray);

customer = load '$input/customer.tbl' USING PigStorage('|') as (
   c_custkey:long,c_name:chararray, c_address:chararray,
   c_nationkey:int, c_phone:chararray, c_acctbal:double,
   c_mktsegment:chararray, c_comment:chararray);


nation10 = load '$input/nation.tbl' USING PigStorage('|') as (
   n_nationkey:int, n_name1:chararray, n_regionkey:int,
   n_comment:chararray);

nation20 = load '$input/nation.tbl' USING PigStorage('|') as (
   n_nationkey:int, n_name2:chararray, n_regionkey:int,
   n_comment:chararray);

nation1 = filter nation10 by n_name1=='FRANCE' or n_name1=='
   GERMANY';
nation2 = filter nation20 by n_name2=='FRANCE' or n_name2=='
   GERMANY';

lineitem = filter lineitem0 by l_shipdate >= '1995-01-01' and
   l_shipdate <= '1996-12-31';

supplier_nation1 = join supplier by s_nationkey, nation1 by
   n_nationkey USING 'replicated';

liteitem_supplier_nation1 = join supplier_nation1 by s_suppkey,
   lineitem by l_suppkey; -- big relaion on right.

customer_nation2 = join customer by c_nationkey, nation2 by
   n_nationkey USING 'replicated';

orders_customer_nation2 = join customer_nation2 by c_custkey,
   orders by o_custkey; -- big relaion on right.
```

```
final_join = join orders_customer_nation2 by o_orderkey,
   liteitem_supplier_nation1 by l_orderkey; -- big relaion on
   right.

filtered_final_join = filter final_join by
(n_name1=='FRANCE' and n_name2=='GERMANY') or
(n_name1=='GERMANY' and n_name2=='FRANCE');

shipping = foreach filtered_final_join GENERATE
  n_name1 as supp_nation,
  n_name2 as cust_nation,
  SUBSTRING(l_shipdate, 0, 4) as l_year, l_extendedprice * (1 -
     l_discount) as volume;

grouped_shipping = group shipping by (supp_nation, cust_nation,
   l_year);
aggregated_shipping = foreach grouped_shipping GENERATE FLATTEN(
   group), SUM($1.volume) as revenue;

ordered_shipping = order aggregated_shipping by group::
   supp_nation, group::cust_nation, group::l_year;
DUMP ordered_shipping;
```

10.1.8 Q8

```
customer = load '$input/customer.tbl' USING PigStorage('|') as (
   c_custkey:long,c_name:chararray, c_address:chararray,
   c_nationkey:int, c_phone:chararray, c_acctbal:double,
   c_mktsegment:chararray, c_comment:chararray);

orders = load '$input/orders.tbl' USING PigStorage('|') as (
   o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
   o_totalprice:double, o_orderdate:chararray, o_orderpriority:
   chararray, o_clerk:chararray, o_shippriority:long, o_comment:
   chararray);

lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
   l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
   :long, l_quantity:double, l_extendedprice:double, l_discount:
   double, l_tax:double, l_returnflag:chararray, l_linestatus:
   chararray, l_shipdate:chararray, l_commitdate:chararray,
   l_receiptdate:chararray,l_shippingstruct:chararray,
   l_shipmode:chararray, l_comment:chararray);
```

```
supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
   s_suppkey:long, s_name:chararray, s_address:chararray,
   s_nationkey:int, s_phone:chararray, s_acctbal:double,
   s_comment:chararray);

nation = load '$input/nation.tbl' USING PigStorage('|') as (
   n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment
   :chararray);

region = load '$input/region.tbl' USING PigStorage('|') as (
   r_regionkey:int, r_name:chararray, r_comment:chararray);

part = load '$input/part.tbl' USING PigStorage('|') as (
   p_partkey:long, p_name:chararray, p_mfgr:chararray, p_brand:
   chararray, p_type:chararray, p_size:long, p_container:
   chararray, p_retailprice:double, p_comment:chararray);
fregion = filter region by r_name == 'AMERICA';
forders = filter orders by o_orderdate <= '1996-12-31' and
   o_orderdate >= '1995-01-01';
fpart = filter part by p_type == 'ECONOMY ANODIZED STEEL';

n1 = join nation by n_regionkey, fregion by r_regionkey USING '
   replicated';
seln1 = foreach n1 generate n_nationkey;

c1 = join customer by c_nationkey, seln1 by n_nationkey USING '
   replicated';
selc1 = foreach c1 generate c_custkey;

o1 = join forders by o_custkey, selc1 by c_custkey;
selo1 = foreach o1 generate o_orderkey, o_orderdate;

n2 = join supplier by s_nationkey, nation by n_nationkey USING '
   replicated';

lineitem = foreach lineitem generate l_partkey,l_suppkey,
   l_orderkey,l_extendedprice * (1 - l_discount) as volume;
p1 = join fpart by p_partkey, lineitem by l_partkey;

s1 = join n2 by s_suppkey, p1 by l_suppkey;

l1 = join o1 by o_orderkey, s1 by l_orderkey;

sels1 = foreach l1 generate SUBSTRING(o_orderdate,0,4) as o_year
   , volume, (n_name == 'BRAZIL'? volume : 0) as case_volume;
grResult = GROUP sels1 by o_year;
```

```
sumResult = foreach grResult generate group, SUM($1.case_volume)
    / SUM($1.volume);
sortResult = order sumResult by group;

DUMP sortResult;
```

10.1.9 Q9

```
customer = load '$input/customer.tbl' USING PigStorage('|') as (
   c_custkey:long,c_name:chararray, c_address:chararray,
   c_nationkey:int, c_phone:chararray, c_acctbal:double,
   c_mktsegment:chararray, c_comment:chararray);

orders = load '$input/orders.tbl' USING PigStorage('|') as (
   o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
   o_totalprice:double, o_orderdate:chararray, o_orderpriority:
   chararray, o_clerk:chararray, o_shippriority:long, o_comment:
   chararray);

lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
   l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
   :long, l_quantity:double, l_extendedprice:double, l_discount:
   double, l_tax:double, l_returnflag:chararray, l_linestatus:
   chararray, l_shipdate:chararray, l_commitdate:chararray,
   l_receiptdate:chararray,l_shippingstruct:chararray,
   l_shipmode:chararray, l_comment:chararray);

supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
   s_suppkey:long, s_name:chararray, s_address:chararray,
   s_nationkey:int, s_phone:chararray, s_acctbal:double,
   s_comment:chararray);

nation = load '$input/nation.tbl' USING PigStorage('|') as (
   n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment
   :chararray);

region = load '$input/region.tbl' USING PigStorage('|') as (
   r_regionkey:int, r_name:chararray, r_comment:chararray);

partsupp = load '$input/partsupp.tbl' USING PigStorage('|') as (
   ps_partkey:long, ps_suppkey:long, ps_availqty:long,
   ps_supplycost:double, ps_comment:chararray);
```

```
part = load '$input/part.tbl' USING PigStorage('|') as (
    p_partkey:long, p_name:chararray, p_mfgr:chararray, p_brand:
    chararray, p_type:chararray, p_size:long, p_container:
    chararray, p_retailprice:double, p_comment:chararray);

fpart = filter part by REGEX_EXTRACT(p_name,'(green)', 1) != '';

j1 = join fpart by p_partkey, lineitem by l_partkey; -- '
    replicated' failed again in 100GB.

j2 = join supplier by s_nationkey, nation by n_nationkey USING '
    replicated';

j3 = join j2 by s_suppkey, j1 by l_suppkey;  -- 'replicated'
    failed in 100GB.

j4 = join partsupp by (ps_suppkey, ps_partkey), j3 by (l_suppkey
    , l_partkey);

j5 = join j4 by l_orderkey, orders by o_orderkey;

selo1 = foreach j5 generate n_name as nation_name, SUBSTRING(
    o_orderdate, 0, 4) as o_year, (l_extendedprice * (1 -
    l_discount) - ps_supplycost * l_quantity) as amount;

grResult = GROUP selo1 by (nation_name, o_year);

sumResult = foreach grResult generate flatten(group), SUM(selo1.
    amount) as sum_profit;
sortResult = order sumResult by nation_name, o_year desc;

DUMP sortResult;
```

10.1.10 Q10

```
customer = load '$input/customer.tbl' USING PigStorage('|') as (
    c_custkey:long,c_name:chararray, c_address:chararray,
    c_nationkey:int, c_phone:chararray, c_acctbal:double,
    c_mktsegment:chararray, c_comment:chararray);

orders = load '$input/orders.tbl' USING PigStorage('|') as (
    o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
    o_totalprice:double, o_orderdate:chararray, o_orderpriority:
    chararray, o_clerk:chararray, o_shippriority:long, o_comment:
    chararray);
```

```
lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
   l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
   :long, l_quantity:double, l_extendedprice:double, l_discount:
   double, l_tax:double, l_returnflag:chararray, l_linestatus:
   chararray, l_shipdate:chararray, l_commitdate:chararray,
   l_receiptdate:chararray,l_shippingstruct:chararray,
   l_shipmode:chararray, l_comment:chararray);

supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
   s_suppkey:long, s_name:chararray, s_address:chararray,
   s_nationkey:int, s_phone:chararray, s_acctbal:double,
   s_comment:chararray);

nation = load '$input/nation.tbl' USING PigStorage('|') as (
   n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment
   :chararray);

region = load '$input/region.tbl' USING PigStorage('|') as (
   r_regionkey:int, r_name:chararray, r_comment:chararray);

partsupp = load '$input/partsupp.tbl' USING PigStorage('|') as (
   ps_partkey:long, ps_suppkey:long, ps_availqty:long,
   ps_supplycost:double, ps_comment:chararray);

part = load '$input/part.tbl' USING PigStorage('|') as (
   p_partkey:long, p_name:chararray, p_mfgr:chararray, p_brand:
   chararray, p_type:chararray, p_size:long, p_container:
   chararray, p_retailprice:double, p_comment:chararray);


forders = filter orders by o_orderdate < '1994-01-01' and
   o_orderdate >= '1993-10-01';
flineitem = filter lineitem by l_returnflag == 'R';

c1 = join customer by c_custkey, forders by o_custkey;
selc1 = foreach c1 generate c_custkey, c_name, c_acctbal,
   c_address, c_phone, c_comment, c_nationkey, o_orderkey;

o1 = join nation by n_nationkey, selc1 by c_nationkey;
selo1 = foreach o1 generate o_orderkey, c_custkey, c_name,
   c_address, c_phone, c_acctbal, c_comment, n_name;

l1 = join flineitem by l_orderkey, selo1 by o_orderkey;
sell1 = foreach l1 generate c_custkey, c_name, l_extendedprice *
    (1 - l_discount) as volume, c_acctbal, n_name, c_address,
   c_phone, c_comment;
```

```
grResult = GROUP sell1 by (c_custkey, c_name, c_acctbal, c_phone
    , n_name, c_address, c_comment);

sumResult = foreach grResult generate group.c_custkey, group.
    c_name, SUM(sell1.volume) as revenue, group.c_acctbal, group.
    n_name, group.c_address, group.c_phone, group.c_comment;
sortResult = order sumResult by revenue desc;
limitResult = limit sortResult 20;

-- store limitResult into '$output/Q10out' USING PigStorage('|')
    ;
DUMP limitResult;
```

10.1.11 Q11

```
partsupp = load '$input/partsupp.tbl' USING PigStorage('|') as (
    ps_partkey:long, ps_suppkey:long, ps_availqty:long,
    ps_supplycost:double, ps_comment:chararray);

supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
    s_suppkey:long, s_name:chararray, s_address:chararray,
    s_nationkey:int, s_phone:chararray, s_acctbal:double,
    s_comment:chararray);

nation = load '$input/nation.tbl' USING PigStorage('|') as (
    n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment
    :chararray);

fnation = filter nation by n_name == 'GERMANY';

j1 = join fnation by n_nationkey, supplier by s_nationkey;

selj1 = foreach j1 generate s_suppkey;

j2 = join partsupp by ps_suppkey, selj1 by s_suppkey;

selj2 = foreach j2 generate ps_partkey, (ps_supplycost *
    ps_availqty) as val;

grResult = group selj2 all;

sumResult = foreach grResult generate SUM($1.val) as totalSum;
```

```
    above inside, below outside)
```

```
outerGrResult = group selj2 by ps_partkey;

outerSumResult = foreach outerGrResult generate group, SUM($1.
    val) as outSum;

outerHaving = filter outerSumResult by outSum > sumResult.
    totalSum * 0.0001;

ord = order outerHaving by outSum desc;

DUMP ord;
```

10.1.12 Q12

```
orders = load '$input/orders.tbl' USING PigStorage('|') as (
    o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
    o_totalprice:double, o_orderdate:chararray, o_orderpriority:
    chararray, o_clerk:chararray, o_shippriority:long, o_comment:
    chararray);

lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
    l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
    :long, l_quantity:double, l_extendedprice:double, l_discount:
    double, l_tax:double, l_returnflag:chararray, l_linestatus:
    chararray, l_shipdate:chararray, l_commitdate:chararray,
    l_receiptdate:chararray,l_shippingstruct:chararray,
    l_shipmode:chararray, l_comment:chararray);

flineitem = FILTER lineitem BY l_shipmode MATCHES 'MAIL|SHIP'
   AND
                    l_commitdate < l_receiptdate AND l_shipdate <
                        l_commitdate AND
                    l_receiptdate >= '1994-01-01' AND
                       l_receiptdate < '1995-01-01';

l1 = JOIN flineitem BY l_orderkey, orders by o_orderkey;
sell1 = FOREACH l1 GENERATE l_shipmode, o_orderpriority;

grpResult = GROUP sell1 BY l_shipmode;
sumResult = FOREACH grpResult{
    urgItems = FILTER sell1 BY o_orderpriority MATCHES '1-URGENT
        ' or o_orderpriority MATCHES '2-HIGH';
    GENERATE group, COUNT(urgItems), COUNT(sell1) - COUNT (
        urgItems);
};
sortResult = ORDER sumResult BY group;
```

```
DUMP sortResult;
```

10.1.13 Q13

```
customer = load '$input/customer.tbl' USING PigStorage('|') as (
    c_custkey:long,c_name:chararray, c_address:chararray,
    c_nationkey:int, c_phone:chararray, c_acctbal:double,
    c_mktsegment:chararray, c_comment:chararray);

orders = load '$input/orders.tbl' USING PigStorage('|') as (
    o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
    o_totalprice:double, o_orderdate:chararray, o_orderpriority:
    chararray, o_clerk:chararray, o_shippriority:long, o_comment:
    chararray);

forders = FILTER orders by (NOT o_comment MATCHES '.*special.*
    requests.*');

porders = FOREACH forders GENERATE o_custkey, o_orderkey;

pcustomer = FOREACH customer GENERATE c_custkey;

COG1 = COGROUP pcustomer by c_custkey, porders by o_custkey;
COG2 = filter COG1 by COUNT(pcustomer) > 0; -- left out join,
    ensure left side non-empty
COG = FOREACH COG2 GENERATE group as c_custkey, COUNT(porders.
    o_orderkey) as c_count;

groupResult = GROUP COG BY c_count;

countResult = FOREACH groupResult GENERATE group as c_count,
   COUNT(COG) as custdist;

orderResult = ORDER countResult by custdist DESC, c_count DESC;

DUMP orderResult;
```

10.1.14 Q14

```
lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
    l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
    :long, l_quantity:double, l_extendedprice:double, l_discount:
    double, l_tax:double, l_returnflag:chararray, l_linestatus:
    chararray, l_shipdate:chararray, l_commitdate:chararray,
    l_receiptdate:chararray,l_shippingstruct:chararray,
    l_shipmode:chararray, l_comment:chararray);

part = load '$input/part.tbl' USING PigStorage('|') as (
    p_partkey:long, p_name:chararray, p_mfgr:chararray, p_brand:
    chararray, p_type:chararray, p_size:long, p_container:
    chararray, p_retailprice:double, p_comment:chararray);

filtered_lineitem = filter lineitem by l_shipdate >=
    '1995-09-01' and l_shipdate < '1995-10-01';
lineitem2 = foreach filtered_lineitem generate l_partkey,
    l_extendedprice * (1 - l_discount) as l_value;

lineitem_part = join lineitem2 by l_partkey, part by p_partkey;
lineitem_part_grouped = group lineitem_part ALL;
sum_all = foreach lineitem_part_grouped generate SUM(
    lineitem_part.l_value);

f_lineitem_part = filter lineitem_part by SUBSTRING(p_type, 0,
    5)=='PROMO';
f_lineitem_part_group = group f_lineitem_part ALL;
sum_filter = foreach f_lineitem_part_group generate SUM(
    f_lineitem_part.l_value);

promo_revenue = foreach sum_all generate 100*sum_filter.$0/
    sum_all.$0;

DUMP promo_revenue;
```

10.1.15 Q15

```
lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
    l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
    :long, l_quantity:double, l_extendedprice:double, l_discount:
    double, l_tax:double, l_returnflag:chararray, l_linestatus:
    chararray, l_shipdate:chararray, l_commitdate:chararray,
    l_receiptdate:chararray,l_shippingstruct:chararray,
    l_shipmode:chararray, l_comment:chararray);

supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
    s_suppkey:long, s_name:chararray, s_address:chararray,
    s_nationkey:int, s_phone:chararray, s_acctbal:double,
    s_comment:chararray);

flineitem = filter lineitem by l_shipdate >= '1996-01-01' and
    l_shipdate < '1996-04-01';
```

```
sumlineitem = foreach flineitem generate l_suppkey,
    l_extendedprice * (1 - l_discount) as value;

glineitem = group sumlineitem by l_suppkey;

revenue = foreach glineitem generate group as supplier_no, SUM(
    $1.value) as total_revenue;

grevenue = group revenue all;

max_revenue = foreach grevenue generate MAX($1.total_revenue);

top_revenue = filter revenue by total_revenue == max_revenue.$0;

j1 = join supplier by s_suppkey, top_revenue by supplier_no;

sel = foreach j1 generate s_suppkey, s_name, s_address, s_phone,
     total_revenue;

ord = order sel by s_suppkey desc;
DUMP ord;
```

10.1.16 Q16

```
/*
SELECT P_BRAND, P_TYPE, P_SIZE, COUNT(DISTINCT PS_SUPPKEY) AS
    SUPPLIER_CNT
FROM PARTSUPP, PART
WHERE P_PARTKEY = PS_PARTKEY AND P_BRAND <> 'Brand#45' AND
    P_TYPE NOT LIKE 'MEDIUM POLISHED%%'
AND P_SIZE IN (49, 14, 23, 45, 19, 3, 36, 9) AND PS_SUPPKEY NOT
    IN (SELECT S_SUPPKEY FROM SUPPLIER
 WHERE S_COMMENT LIKE '%%Customer%%Complaints%%')
GROUP BY P_BRAND, P_TYPE, P_SIZE
ORDER BY SUPPLIER_CNT DESC, P_BRAND, P_TYPE, P_SIZE
*/
-- parts = LOAD '$input/part.tbl' USING PigStorage('|') AS
-- (p_partkey, p_name, p_mfgr, p_brand: chararray, p_type:
    chararray, p_size, p_container, p_retailprice, p_comment);

-- partsupp = LOAD '$input/partsupp.tbl' USING PigStorage('|')
    AS (ps_partkey: int, ps_suppkey: int, ps_availqty,
    ps_supplycost:double, ps_comment);
```

```
-- supplier = LOAD '$input/supplier.tbl' USING PigStorage('|')
   AS (s_suppkey: int, s_name, s_address, s_nationkey, s_phone,
   s_acctbal, s_comment: chararray);

-- fsupplier = FILTER supplier BY (NOT s_comment MATCHES '.*
   Customer.*Complaints.*');
-- fs1 = FOREACH fsupplier GENERATE s_suppkey;

-- pss = JOIN partsupp BY ps_suppkey, fs1 BY s_suppkey;

-- fpartsupp = FOREACH pss GENERATE partsupp::ps_partkey as
   ps_partkey, partsupp::ps_suppkey as ps_suppkey;

-- fparts = FILTER parts BY
-- (p_brand != 'Brand#45' AND
--  NOT (p_type MATCHES 'MEDIUM POLISHED.*') AND
--   p_size MATCHES '49|14|23|45|19|3|36|9');

-- pparts = FOREACH fparts GENERATE p_partkey, p_brand, p_type,
   p_size;

-- p1 = JOIN pparts BY p_partkey, fpartsupp by ps_partkey;
-- grResult = GROUP p1 BY (p_brand, p_type, p_size);
-- countResult = FOREACH grResult
-- {
--    dkeys = DISTINCT p1.ps_suppkey;
--    GENERATE group.p_brand as p_brand, group.p_type as p_type,
   group.p_size as p_size, COUNT(dkeys) as supplier_cnt;
-- }
-- orderResult = ORDER countResult BY supplier_cnt DESC, p_brand
   , p_type, p_size;

-- DUMP orderResult;

RUN /home/robert/Dropbox/university/4th/Project/src/hadoopRun/
   pig/tpch/loadTPCH.pig;


fsupplier = FILTER supplier BY (NOT s_comment MATCHES '.*
   Customer.*Complaints.*');
fs1 = FOREACH fsupplier GENERATE s_suppkey;

pss = JOIN partsupp BY ps_suppkey, fs1 BY s_suppkey;

fpartsupp = FOREACH pss GENERATE partsupp::ps_partkey as
   ps_partkey, partsupp::ps_suppkey as ps_suppkey;
```

```
fparts = FILTER part BY
(p_brand != 'Brand#45'
  AND NOT (p_type MATCHES 'MEDIUM POLISHED.*')
  AND (chararray) p_size MATCHES '49|14|23|45|19|3|36|9');

pparts = FOREACH fparts GENERATE p_partkey, p_brand, p_type,
    p_size;

p1 = JOIN pparts BY p_partkey, fpartsupp by ps_partkey;
grResult = GROUP p1 BY (p_brand, p_type, p_size);
countResult = FOREACH grResult
{
  dkeys = DISTINCT p1.ps_suppkey;
  GENERATE group.p_brand as p_brand, group.p_type as p_type,
      group.p_size as p_size, COUNT(dkeys) as supplier_cnt;
}
orderResult = ORDER countResult BY supplier_cnt DESC, p_brand,
    p_type, p_size;

-- store orderResult into '$output/Q16out' USING PigStorage('|')
    ;
DUMP orderResult;
```

10.1.17 Q17

```
lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
    l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
    :long, l_quantity:double, l_extendedprice:double, l_discount:
    double, l_tax:double, l_returnflag:chararray, l_linestatus:
    chararray, l_shipdate:chararray, l_commitdate:chararray,
    l_receiptdate:chararray,l_shippingstruct:chararray,
    l_shipmode:chararray, l_comment:chararray);

part = load '$input/part.tbl' USING PigStorage('|') as (
    p_partkey:long, p_name:chararray, p_mfgr:chararray, p_brand:
    chararray, p_type:chararray, p_size:long, p_container:
    chararray, p_retailprice:double, p_comment:chararray);

lineitem = foreach lineitem generate l_partkey, l_quantity,
    l_extendedprice ;
part = FILTER part BY p_brand == 'Brand#23' AND p_container == '
   MED BOX';
part = foreach part generate p_partkey;

COG1 = COGROUP part by p_partkey, lineitem by l_partkey;
COG1 = filter COG1 by COUNT(part) > 0;
```

```
COG2 = FOREACH COG1 GENERATE COUNT(part) as count_part, FLATTEN(
    lineitem), 0.2 * AVG(lineitem.l_quantity) as l_avg;

COG3 = filter COG2 by l_quantity < l_avg;
COG = foreach COG3 generate (l_extendedprice * count_part) as
    l_sum;

G1 = group COG ALL;

result = foreach G1 generate SUM(COG.l_sum)/7.0;

DUMP result;
```

10.1.18 Q18

```


lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
    l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
    :long, l_quantity:double, l_extendedprice:double, l_discount:
    double, l_tax:double, l_returnflag:chararray, l_linestatus:
    chararray, l_shipdate:chararray, l_commitdate:chararray,
    l_receiptdate:chararray,l_shippingstruct:chararray,
    l_shipmode:chararray, l_comment:chararray);

customer = load '$input/customer.tbl' USING PigStorage('|') as (
    c_custkey:long,c_name:chararray, c_address:chararray,
    c_nationkey:int, c_phone:chararray, c_acctbal:double,
    c_mktsegment:chararray, c_comment:chararray);

orders = load '$input/orders.tbl' USING PigStorage('|') as (
    o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
    o_totalprice:double, o_orderdate:chararray, o_orderpriority:
    chararray, o_clerk:chararray, o_shippriority:long, o_comment:
    chararray);

lineitem = foreach lineitem generate l_orderkey, l_quantity;
orders = foreach orders generate o_orderkey,o_orderdate,
    o_totalprice, o_custkey;
COG = cogroup lineitem by l_orderkey, orders by o_orderkey;

COG = filter COG by SUM(lineitem.l_quantity) > 300;

lineitem_orders = foreach COG generate SUM(lineitem.l_quantity)
    as l_quantity_sum, flatten(orders); -- orders has only one
    tuple per bag!
```

```
lineitem_orders_customer = join customer by c_custkey ,
   lineitem_orders by o_custkey;

lineitem_orders_customer_group = group lineitem_orders_customer
   by (c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice)
   ;
result = foreach lineitem_orders_customer_group generate group.
   c_name as c_name, group.c_custkey as c_custkey, group.
   o_orderkey as o_orderkey, group.o_orderdate as o_orderdate,
   group.o_totalprice as o_totalprice, SUM(
   lineitem_orders_customer.l_quantity_sum) as l_quantity_sum;

out = order result by o_totalprice desc, o_orderdate;

DUMP out;
```

10.1.19 Q19

```
lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
   l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
   :long, l_quantity:double, l_extendedprice:double, l_discount:
   double, l_tax:double, l_returnflag:chararray, l_linestatus:
   chararray, l_shipdate:chararray, l_commitdate:chararray,
   l_receiptdate:chararray,l_shipinstruct:chararray, l_shipmode:
   chararray, l_comment:chararray);

part = load '$input/part.tbl' USING PigStorage('|') as (
   p_partkey:long, p_name:chararray, p_mfgr:chararray, p_brand:
   chararray, p_type:chararray, p_size:long, p_container:
   chararray, p_retailprice:double, p_comment:chararray);

lpart = JOIN lineitem BY l_partkey, part by p_partkey;

fltResult = FILTER lpart BY
  (
    p_brand == 'Brand#12'
  and p_container matches 'SM CASE|SM BOX|SM PACK|SM PKG'
  and l_quantity >= 1 and l_quantity <= 11
  and p_size >= 1 and p_size <= 5
  and l_shipmode matches 'AIR|AIR REG'
  and l_shipinstruct == 'DELIVER IN PERSON'
  )
  or
  (
    p_brand == 'Brand#23'
  and p_container matches 'MED BAG|MED BOX|MED PKG|MED PACK'
  and l_quantity >= 10 and l_quantity <= 20
  and p_size >= 1 and p_size <= 10
  and l_shipmode matches 'AIR|AIR REG'
  and l_shipinstruct == 'DELIVER IN PERSON'
  )
  or
  (
  p_brand == 'Brand#34'
  and p_container matches 'LG CASE|LG BOX|LG PACK|LG PKG'
  and l_quantity >= 20 and l_quantity <= 30
  and p_size >= 1 and p_size <= 15
  and l_shipmode matches 'AIR|AIR REG'
  and l_shipinstruct == 'DELIVER IN PERSON'
  );
volume = FOREACH fltResult GENERATE l_extendedprice * (1 -
   l_discount);
grpResult = GROUP volume ALL;
revenue = FOREACH grpResult GENERATE SUM(volume);

DUMP revenue;
```

10.1.20 Q20

```


part = load '$input/part.tbl' USING PigStorage('|') as (
   p_partkey:long, p_name:chararray, p_mfgr:chararray, p_brand:
   chararray, p_type:chararray, p_size:long, p_container:
   chararray, p_retailprice:double, p_comment:chararray);
part1 = foreach part generate p_partkey, p_name;
part2 = filter part1 by SUBSTRING(p_name, 0, 6)=='forest';
part3 = foreach part2 generate p_partkey;
part4 = distinct part3;

lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
   l_orderkey, l_partkey:long, l_suppkey:long, l_linenumber,
   l_quantity:double, l_extendedprice, l_discount, l_tax,
   l_returnflag, l_linestatus, l_shipdate:chararray,
   l_commitdate, l_receiptdate,l_shippingstruct, l_shipmode,
   l_comment);
lineitem1 = foreach lineitem generate l_partkey, l_suppkey,
   l_shipdate, l_quantity;
```

```
lineitem2 = filter lineitem1 by l_shipdate >= '1994-01-01' and
   l_shipdate < '1995-01-01';
lineitem3 = group lineitem2 by (l_partkey, l_suppkey);
lineitem4 = foreach lineitem3 generate FLATTEN(group), SUM(
   lineitem2.l_quantity) * 0.5 as sum;

partsupp = load '$input/partsupp.tbl' USING PigStorage('|') as (
   ps_partkey:long, ps_suppkey:long, ps_availqty:long,
   ps_supplycost, ps_comment);
partsupp1 = foreach partsupp generate ps_suppkey, ps_partkey,
   ps_availqty;
ps_p = join part4 by p_partkey, partsupp1 by ps_partkey;
l_ps_p = join ps_p by (ps_partkey, ps_suppkey), lineitem4 by (
   l_partkey, l_suppkey);
l_ps_p2 = filter l_ps_p by ps_availqty > sum;
ps_suppkey_prj = foreach l_ps_p2 generate ps_suppkey;
ps_suppkey_distinct = distinct ps_suppkey_prj;

nation = load '$input/nation.tbl' USING PigStorage('|') as (
   n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment
   :chararray);
nation1 = filter nation by n_name == 'CANADA';

supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
   s_suppkey:long, s_name:chararray, s_address:chararray,
   s_nationkey:int, s_phone, s_acctbal, s_comment);
supplier1 = foreach supplier generate s_suppkey,s_name,
   s_nationkey,s_address;
s_n = join supplier by s_nationkey, nation1 by n_nationkey;
s_n_ps = join s_n by s_suppkey, ps_suppkey_distinct by
   ps_suppkey;

res_prj = foreach s_n_ps generate s_name,s_address;
res = order res_prj by s_name;

DUMP res;
```

10.1.21 Q21

```
orders = load '$input/orders.tbl' USING PigStorage('|') as (
   o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
   o_totalprice:double, o_orderdate:chararray, o_orderpriority:
   chararray, o_clerk:chararray, o_shippriority:long, o_comment:
   chararray);
```

```
lineitem = load '$input/lineitem.tbl' USING PigStorage('|') as (
    l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber
    :long, l_quantity:double, l_extendedprice:double, l_discount:
    double, l_tax:double, l_returnflag:chararray, l_linestatus:
    chararray, l_shipdate:chararray, l_commitdate:chararray,
    l_receiptdate:chararray,l_shippingstruct:chararray,
    l_shipmode:chararray, l_comment:chararray);

supplier = load '$input/supplier.tbl' USING PigStorage('|') as (
    s_suppkey:long, s_name:chararray, s_address:chararray,
    s_nationkey:int, s_phone:chararray, s_acctbal:double,
    s_comment:chararray);

nation = load '$input/nation.tbl' USING PigStorage('|') as (
    n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment
    :chararray);

______________________________

lineitem = foreach lineitem generate l_suppkey,l_orderkey,
    l_receiptdate,l_commitdate;
orders = foreach orders generate o_orderkey, o_orderstatus;
supplier = foreach supplier generate s_suppkey, s_nationkey,
    s_name;

gl = group lineitem by l_orderkey;

L2 = filter gl by COUNT(org.apache.pig.builtin.Distinct(lineitem
    .l_suppkey))>1;

fL2 = foreach L2{
        t1 = filter lineitem by l_receiptdate > l_commitdate;
        generate group, t1;
}
fL3 = filter fL2 by COUNT(org.apache.pig.builtin.Distinct($1.
    l_suppkey)) == 1;

L3 = foreach fL3 generate flatten($1);

___________________________


fn = filter nation by n_name == 'SAUDI ARABIA';
fn_s = join supplier by s_nationkey, fn by n_nationkey USING '
    replicated';

fn_s_L3 = join L3 by l_suppkey, fn_s by s_suppkey;
```

```
fo = filter orders by o_orderstatus == 'F';
fn_s_L3_fo = join fn_s_L3 by l_orderkey, fo by o_orderkey;

gres = group fn_s_L3_fo by s_name;
sres = foreach gres generate group as s_name, COUNT($1) as
   numwait;
ores = order sres by numwait desc, s_name;
lres = limit ores 100;

DUMP lres;
```

10.1.22 Q22

```
customer = load '$input/customer.tbl' USING PigStorage('|') as (
   c_custkey:long,c_name:chararray, c_address:chararray,
   c_nationkey:int, c_phone:chararray, c_acctbal:double,
   c_mktsegment:chararray, c_comment:chararray);
orders = load '$input/orders.tbl' USING PigStorage('|') as (
   o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
   o_totalprice:double, o_orderdate:chararray, o_orderpriority:
   chararray, o_clerk:chararray, o_shippriority:long, o_comment:
   chararray);

customer_filter = filter customer by c_acctbal>0.00 and
   SUBSTRING(c_phone, 0, 2) MATCHES '13|31|23|29|30|18|17';
customer_filter_group = group customer_filter all;
avg_customer_filter = foreach customer_filter_group generate AVG
   (customer_filter.c_acctbal) as avg_c_acctbal;

customer_sec_filter = filter customer by c_acctbal >
   avg_customer_filter.avg_c_acctbal and SUBSTRING(c_phone, 0,
   2) MATCHES '13|31|23|29|30|18|17';
customer_orders_left = join customer_sec_filter by c_custkey
   left, orders by o_custkey;

customer_trd_filter = filter customer_orders_left by o_custkey
   is null;
customer_rows = foreach customer_trd_filter generate SUBSTRING(
   c_phone, 0, 2) as cntrycode, c_acctbal;

customer_result_group = group customer_rows by cntrycode;
customer_result = foreach customer_result_group generate group,
   COUNT(customer_rows) as numcust, SUM(customer_rows.c_acctbal)
    as totacctbal;
```

```
customer_result_inorder = order customer_result by group;

DUMP customer_result_inorder;
```

10.2 Filter Queries

These are the queries that I used for testing the ratio of reduction after the conditions has been applied.

10.2.1 filter1

```
filsupp = FILTER supplier BY (s_suppkey < 25 * 20) OR
(s_suppkey <= 65-20*3 AND s_suppkey >= 8*9-70) OR
(s_name < 'abc');
group1 = GROUP filsupp all;
DUMP group1;
```

10.2.2 filter2

```
fil = FILTER part BY p_brand == 'Brand#23' AND
p_container == 'MED BOX';
group1 = GROUP fil all;
DUMP group1;
```

10.2.3 filter3

```
fil = filter partsupp by ps_suppkey == 5;
group1 = GROUP fil  all;
DUMP group1;
```

10.2.4 filter4

```
fil = filter partsupp by ps_suppkey >= 5;
group1 = GROUP fil  ALL;
DUMP group1;
```

10.2.5 filter5

```
fil = filter partsupp by ps_suppkey > 100;
group1 = GROUP fil all;
DUMP group1;
```

10.2.6 filter6

```
fil = filter partsupp by ps_suppkey > 500;
group1 = GROUP fil all;
DUMP group1;
```

10.2.7 filterNation

```
fil = filter nation by n_name == 'CANADA';
group1 = GROUP fil all;
DUMP group1;
```