# IV
# Statistical Modelling of MapReduce Joins

In this chapter, we will also explain each component used while constructing our statistical model such as:

- The construction of the dataset used.
- The use of the statistical technique chosen.
- The parameters used for the statistical modelling.
- The procurement of a special parameter that is the join selectivity for each query.

We will also show the results gathered at each stage of the model construction. In section IV.1, we will explain some of the main characteristics of the `TPC-DS` which led us to choose it as our dataset in our experiments. We will also describe how we adapted `TPC-DS` data and queries for our experiment. Then in section IV.2, we will explain different statistical techniques used for estimating execution times for computational jobs. Section IV.3 explains the join selectivity concept and the algorithm used to obtain it. In addition, we describe the problems and challenges found while obtaining the join selectivity parameter. The experimental setup is detailed in the section IV.4. The results and explanation about them are presented in section IV.5 of this chapter. Possible improvements for the results are discussed at the end of this chapter.

## IV.1 Data Set

Most companies trying to manage big data are web companies that have enough resources to handle big data and develop tools for it. Information obtained by them is part of their assets, and for most of those companies such information is part of their business core. On the other hand, working with big data tools implies the need of big data to actually test them. This is a problem because there are not any real datasets publicly available for academia. Therefore, we decided to use a benchmark from the database community for

PUC-Rio - Certificação Digital Nº 0912867/CA

Table IV.1: TPC-DS data scaling [22].

| Table Name | Avg.Size | 1GB | 100GB | 1TB | 10TB | 100TB |
|---|---|---|---|---|---|---|
| call_center | 305 | 0.00174 | 0.00872 | 0.012216 | 0.0157 | 0.01745 |
| catalog_page | 139 | 1.55334 | 2.70423 | 3.97682 | 5.30242 | 6.62803 |
| catalog_returns | 166 | 22.807 | 2280.35 | 21371.26 | 227971.55 | 2279643.18 |
| catalog_sales | 226 | 310.69 | 31035.74 | 290961.14 | 3103630.08 | 31036298.03 |
| customer | 132 | 12.58 | 251.77 | 1510.62 | 8182.52 | 12588.50 |
| customer_address | 110 | 5.2452 | 104.90 | 629.42 | 3409.38 | 5245.20 |
| customer_demographics | 42 | 76.936 | 76.936 | 76.936 | 76.936 | 76.936 |
| date_dim | 141 | 9.8227 | 9.82275 | 9.8227 | 9.8227 | 9.8227 |
| household_demographics | 21 | 0.14419 | 0.14419 | 0.14419 | 0.14419 | 0.14419 |
| income_band | 16 | 0.000305 | 0.000305 | 0.000305 | 0.000305 | 0.000305 |
| inventory | 16 | 179.214 | 6093.29 | 11200.90 | 20012.28 | 29988.67 |
| item | 281 | 4.8236 | 54.6684 | 80.394 | 107.72 | 134.527 |
| promotions | 124 | 0.03547 | 0.11825 | 0.17738 | 0.23651 | 0.29563 |
| reason | 38 | 0.00126 | 0.00199 | 0.00235 | 0.00253 | 0.00271 |
| ship_mode | 56 | 0.001068 | 0.001068 | 0.001068 | 0.001068 | 0.001068 |
| store | 263 | 0.00300 | 0.10082 | 0.25131 | 0.37622 | 0.47705 |
| store_returns | 134 | 36.74 | 3679.79 | 34504.94 | 368032.002 | 3680388.50 |
| store_sales | 164 | 450.50 | 45043.47 | 422284.61 | 4504383.98 | 45043669.47 |
| time_dim | 59 | 4.86145 | 4.86145 | 4.86145 | 4.86145 | 4.86145 |
| warehouse | 117 | 0.00055 | 0.00167 | 0.00223 | 0.00278 | 0.00334 |
| web_page | 96 | 0.00549 | 0.18676 | 0.27465 | 0.36639 | 0.45812 |
| web_returns | 162 | 11.087 | 1112.005 | 10428.09 | 111239.73 | 1112341.69 |
| web_sales | 226 | 155.049 | 15518.45 | 145483.53 | 1551810.94 | 15518260.53 |
| web_site | 292 | 0.00835 | 0.00668 | 0.01503 | 0.02172 | 0.02673 |

testing `MapReduce` jobs as there is no specific dataset for it. We consider TPC-DS [22] which is a decision support workload being evaluated by the *Transaction Processing Performance Council* [23].

Raghunath et al. [88] explain that using both synthetic and real world data for designing the TPC-DS has many advantages over other database benchmarks that use only one type of data. This is because synthetic data sets built using studied distributions such as the Normal or the Poisson distributions has many positive points, but they are not well suited for dynamically substituting bind variables. The `TPC-DS` utilizes traditional synthetic distributions, yielding uniformly distributed integers or word selections with a Gaussian distribution.

Scaling a data set can be done in two different ways: Scaling the number of tuples while the underlying value sets (domains) remain static, or number of tuples remain fixed expanding the domains used to generate. In the case of `TPC-DS`, an hybrid approach was chosen, so that most table columns employ dataset scaling instead of domain scaling, specially fact table columns; Some small tables' columns use domain scaling. Therefore, fact tables scale linearly with the scale factor while dimensions scale sub-linearly.

Table IV.1 shows how data from `TPC-DS` scales in size while varying the scale factor. It can be noticed that there are relations such as household_demographics, income_band, date_dim, and others that do not grow in number of tuples because changing its size would not be consistent with a data warehouse application where dimension tables remain the same along the time.

We have decided to consider the `TPC-DS` data because we could have some information about the dataset as well knowing how scaling it would

PUC-Rio - Certificação Digital Nº 0912867/CA

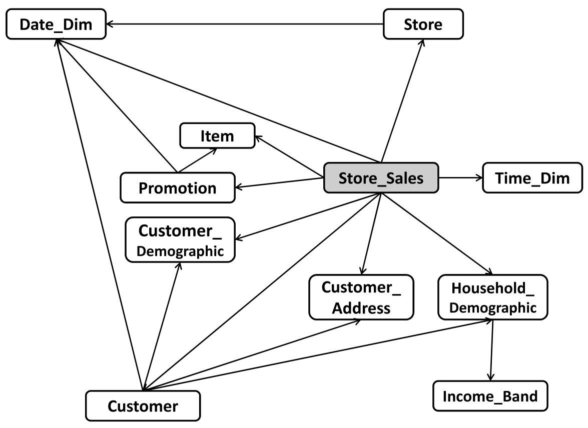

Figure IV.1: Relationships from some `TPC-DS`' tables.

affect the underlying data distribution. The data generated using `TPC-DS` was plain text files, which we have loaded into our local `HDFS` to perform our tests. As we load the data into `HDFS`, we will call the records inside the files 'tuples' because they come from a relational benchmark, and not necessarily because they are actually databse tuples.

It should be noted that the `TPC-DS` has been designed as a decision support application to benchmark relational databases, the queries outlined in [22] were suitable to evaluate a data warehouse application, but they are not suitable for our experiments. This is because the queries for the benchmark were specifically designed for testing different aspects of DBMS e.g. performance, query optimizations, reporting capabilities, among others. But this dissertation aims to characterize a specific relational operator and not to general purpose queries.

In this way, we have decided to analyze the database schema proposed for the `TPC-DS` [88], extract relationships between tables and create queries based on these. For example, Figure IV.1 shows all the relationships for table *Store_Sales*, some fact tables (*Item, Promotion,* etc.) and dimension tables (*Time_Dim, Date_Dim*). We have generated join queries using these tables, but keeping in mind the restrictions posed by each type of join e.g. merge join operator needs both relations physically sorted by the join attribute.

The goal of this work is to model join queries performance and not common queries. That is the reason why we created only join queries that represent all relationships between tables. We have created 96 Pig-Latin queries using a Fragment Replicate Join. In addition to that, we created 23 Pig-Latin queries using a Merge Join, and 102 queries using a Hash Join. Appendix VI shows examples of each type of query.

PUC-Rio - Certificação Digital Nº 0912867/CA

Table IV.2: Workload features.

| **Feature** | **Type** | **Multiple Regression** |
|---|---|---|
| Job name | Categorical | Not used |
| Number of Map processes | Numeric | Used |
| Number of Reduce processes | Numeric | Not Used |
| Map input bytes | Numeric | Used |
| Map input records | Numeric | Used |
| Join Selectivity | Numeric | Used |

## IV.2 Model Construction

Recalling multiple regression description made in chapter III, our workload features will be used as our independent variables ($\{X_1,..,X_j,..,X_J\}$), and the query execution time will be used as our dependent variable ($y$). We then use the resulting formulas to predict other and future queries' execution times.

To build our model we used the workload features described in Table IV.2. The parameters used to build the multiple regression analysis are chosen having in mind that only *a priori* knowledge about the job execution must be used. There are other parameters that whether are known only at job execution time or only after the job has finished executing e.g. "*Combiner spilled bytes*", "*Reduce output bytes*". These parameters will not be used because we want to use only parameters that can be known beforehand.

An important parameter we added to the model is join selectivity for each job execution. Being able to estimate the jobs' output has an important impact on the model since this parameter would describe both the quantity of elements to be written on the distributed file system and the amount of work that would be done. In section IV.3 we will describe the join selectivity algorithm used, its advantages and the difficulties for calculating it.

We must be aware of certain problems that can be caused due to our parameters shown in the table IV.2. There are two parameters that we did not use. The first one is the *job name*. This parameter was not be used because its value does not affect job execution. The other parameter that is not considered is the number of *Reduce processes*. This is due to the fact that it becomes linearly dependent in the number of jobs executed. This linear dependency can lead the multiple regression model to make unstable estimates. The number of reducers to be used is dependent on what the output will be used for, the reduce capacity of the cluster, the amount of data needing to be reduced, and the time needed to perform the reduce operation. In our case, this parameter refers to the number of reducers that will be used in the join operation. All join

PUC-Rio - Certificação Digital Nº 0912867/CA

PUC-Rio - Certificação Digital Nº 0912867/CA

algorithms used in the Pig Framework try to take advantage of all resources available in the cluster and to avoid the network overhead of copying data from mappers to reducers.

In this manner, this parameter does not vary in our jobs' execution because we use the reduce capacity of our cluster as the number of reducers. Therefore, we decided not to use the number of reducers in our model construction process because it would cause unstable estimates. Nevertheless, it could be calculated following [45] recommendations. It states that the number of reducers should be:

*0.95 or 1.75 * (nodes * mapred.tasktracker.tasks.maximum)*

This formula relates the maximum number of tasks with the number of reducers that a single compute node can support at the same time. The first two possible parameters have different purposes. The parameter '0.95' is usually used to improve launching time of jobs by starting map output transfer immediately after the map operation has finished. The latter, '1.75', is used in order to exploit faster nodes because it will force the nodes that finish their tasks to launch a second round of reducers as soon as they finished executing the first ones. So this latter parameter helps improving load balancing. The *"nodes"* parameter represents the number of nodes available in the cluster and the *"mapred.tasktracker.tasks.maximum"* parameter represents the maximum number of tasks that can be supported in a single compute node.

The number of map processes is determined by the number of HDFS blocks in the input files, and can be computed by dividing the number of bytes to be processed by the size of HDFS blocks. To predict the execution time of our validating queries, we have to estimate the number of map processes and then use these values for constructing our model.

In the following section, we will explain how the join selectivity parameter is obtained, the design decisions taken in order to implement it and also the challenges to compute it.

## IV.3 Join Selectivity

In traditional database management systems, optimizers use cost models to generate more efficient query execution plans, so the optimizer can choose the least expensive plan from a set of alternatives. A cost model usually includes features such as the query itself, some database statistics and description of computational resources such as CPU speed, cost of sequential access and random disk operations, size of memory buffer, among others [94].

For estimating the query cost, the optimizer calculates the total resource consumption which is an aggregation of the costs involving operations using CPU, disk and network resources. Therefore, the DBMS needs to maintain some basic statistics about the data in its catalog such as:

– Number of tuples in each relation.

– Number of key values.

– Number of distinct values.

– Minimum and maximum values.

– Histograms or complementary structures describing the distribution of different values for that column.

A substantial problem for the optimizer is estimating the selectivity of join query predicates. The selectivity of query predicates may be seen as the filter factor of a predicate $C$, when applied to a certain relation $R$, gives as a result a fraction of $R$'s tuples that satisfy $C$.

$$\frac{\sigma_C|R|}{|R|} \tag{1}$$

Join selectivity can be estimated using basic statistics about the column used in such predicate. This can be done using histograms on the columns involved by predicate $C$, or using sampling methods on those columns. If there was no additional information some basic formulas can be used to estimate selectivity assuming the uniformity and independence of the underlying data [94].

One important challenge being studied and worked on is the lack of metadata in the MapReduce computing model, and also in its open-source implementation, Hadoop. Systems like Hive [34], Cassandra [76] are trying to incorporate basic statistics into their systems. Once they have accomplished this, these systems will be able to develop appropiate cost models for handling big data. Likewise, the Pig framework [31] lacks of metadata such as column definitions, or relation's basic statistics because it works on top of the Hadoop framework. Pig works along with the HDFS in order to take advantage of important features such as data replication and fault-tolerance, but it also inherits the drawbacks of HDFS design e.g. network latency and lack of schema.

One important HDFS drawback is the simple metadata kept by the `Namenode`. The metadata consists in information about the location of each block file and to which file it belongs, guaranteeing that no two block files have the same Block ID. In spite of the fact that keeping simple statistics make the

PUC-Rio - Certificação Digital Nº 0912867/CA

system faster, the lack of them make the system more rigid to perform different operations than the established ones.

In this way, we need to obtain these basic statistics in order to estimate the parameters we need for our regression model. The following sections will describe the way we obtain statistics that enable the estimation of join selectivity. We may use this estimate as a parameter in our multiple regression model.

## (a) Getting Basic Statistics

One of the main challenges is to get accurate statistics about the relations without incurring in too much overhead. To compute join selectivity, we need to calculate the average length and cardinality of each relation, and the number of distinct values from each sample. For example, to obtain the cardinality of each of the participant relations of a join operator, we have to consider the average length of the records and the total file size. This is directly related to the following cardinality definition:

$$|A| = \frac{RelationSize}{AverageTupleSize} \tag{2}$$

In order to determine the average length of each relation without having to read both relations completely, we decided to sample $n$ tuples from the first $L$ tuples of each of the $k$ file blocks, and then determine their average length. We have run several tests on sampling for determining the cardinality of the relations, varying the amounts of tuples sampled, but always accessing all the file blocks in order to get tuples from all over the relation.

Figure IV.2 shows graphically how this process has been made. It shows that from each file block, we would randomly choose a certain number of tuples from the first $L$ elements. $L$ would be 1000, and 1500. We do not choose the first elements because we would not be taking a representative sample from the underlying data. We divide the number of tuples to be taken, $n$, with the number of blocks, $k$. So we get the same amount of tuples from each file block.

If we have $n$ tuples that will be obtained from $k$ blocks, trying to set a range too wide for choosing tuples can become expensive very fast, and that would perish performance on the whole model construction process. For example, if we want to sample 10 000 tuples checking 1 000 tuples in each block using a 1GB file, then we would have to iterate at least $10^6$ times. This means that even sampling a small number of tuples from a small range we would have to iterate many times to obtain our samples. That is why we decided to use a

PUC-Rio - Certificação Digital Nº 0912867/CA

PUC-Rio - Certificação Digital Nº 0912867/CA

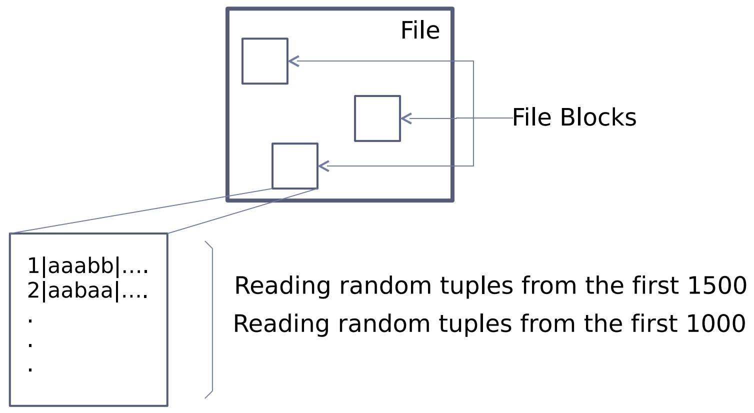
File
File Blocks
1|aaabb|....
2|aabaa|....
Reading random tuples from the first 1500
Reading random tuples from the first 1000

Figure IV.2: Sampling file blocks.

lower number of sampled tuples for testing, but varying the range from where tuples were being chosen.

The I/O cost remains the same compared to previous executions because we use every block to get a uniform sample from the whole file. So the cost difference is by using a wider range from where the samples will be taken i.e. using a larger or smaller $L$. The cardinality estimate does considerably vary when we use a wider range and when take more random tuples i.e. when we randomize the tuples being read in a better way. That is why we tested with larger values of $L$.

We tested cardinality estimation for two different ranges, '$L$' = 1 000 and for '$L$' = 1 500, but varying the number of elements sampled from each block of the different files. Figure IV.3 shows the average error for '$L$' = 1 000 while varying the elements being sampled. The chart shows that if we sample fewer tuples, we will get a higher estimation error, but after increasing the number of tuples samples over a hundred, the error stabilizes. Other important observations about the chart are the two peaks for 2 000 and for 10 000 tuples. These two error peaks are due to the fact that we had added two test relations of 2 000 and 10 000 respectively, and construct them with a single column of numbers. Numbers ranging from 1 to 2 000 and to 10 000. These two test relations do not belong to the originial TPC-DS. The fact of just containing a range of numbers made our sampling strategy not very effective for this case because sometimes random sampling can get tuples of lower numbers (numbers with less digits) from each block, or sometimes it can get only tuples with high number (numbers with more digits) from another block.

PUC-Rio - Certificação Digital Nº 0912867/CA

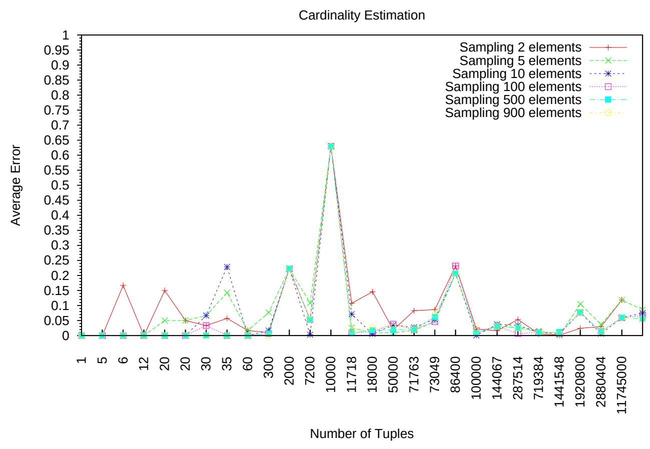

Figure IV.3: Cardinality sampling using $L = 1000$.

This would mean that estimating the cardinality for a relation can be done accurately getting more sample tuples, but for columns with highly variable lenght might not be such a good idea. In this particular case, the average length of numerical tuples varies too much across the records. For example, if we have an index file composed by the numbers from 1 to 1 million, and the random samples are all lower than 1'000, then we may think that all our records have length 3, but they can actually have double the size of the sampled ones. On the other hand, varying the range from where samples are taken improves cardinality estimate because it helps getting a better randomization of the sampling process.

Figure IV.4, shows the average error for $L = 1500$, but samples with more than a hundred tuples because as explained before the average error stabilizes. In this chart, we can observe that the average error is a lot less than in Figure IV.3 because its maximum average error does not gets to 25% whereas for '$L$' = 1000 its maximum average error goes over 60%. This corroborates the idea that with a wider range to help the randomization process, the cardinality can be estimated more accurately.

Figure IV.5 shows the variance of the average error for estimating cardinality for different relations for ranges from the first 1000 tuples and then for a range of the first 1500 tuples. As we can notice, the higher error is when we sample fewer elements from the specified range $L$. However, when we increase the number of elements being sampled from $L$, we are able to randomize the sampling in a better way. This better randomization helps providing more accurate cardinality estimates.

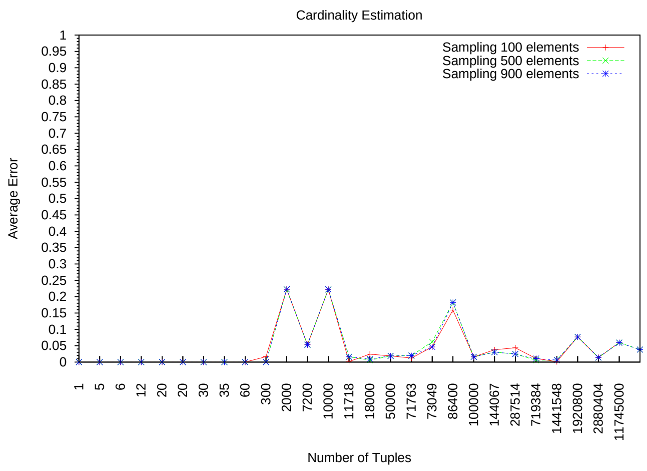

Figure IV.4: Cardinality sampling using $L = 1500$.

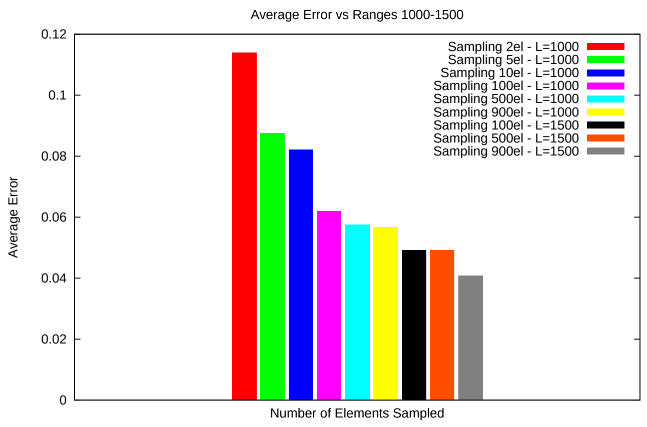

Figure IV.5: Average error for sampled ranges ($L$ being 1000 and 1500).

PUC-Rio - Certificação Digital Nº 0912867/CA

### (b) The join selectivity parameter

Once having the cardinality estimate, the join selectivity parameter can be estimated. We have to keep in mind that the simple statistics gotten so far already have a margin of error explained in the previous section. It is worth recall that we consider only binary joins for simplicity because the Merge join algorithm of the Pig framework can only be applied to two relations at the time. Furthermore, by considering Pig binary joins we also make the results of each type of join operator comparable among them.

In traditional relational database systems join selectivity is usually estimated by using simple statistics about the data, or by using additional data structures such as equi-depth or equi-width histograms maintained by the DBMS. This is done so accurate estimates of value occurrences cardinality can be obtained without actually having to read the entire relations. In addition to this, some DBMS keep compressed histograms to keep exact counts for the most frequently occurring values, or just regular histograms to estimate the frequency of less frequent values. Histograms are heavily used in DBMS to estimate predicate selectivity, but the problem with them is that they have to be built and maintained by the DBMS. If simple statistics about the data were available, the join selectivity could be calculated as follows [68]:

$$R \bowtie_{R\ .a=S.b} S = min(\frac{1}{distinct(R.a)}, \frac{1}{distinct(S.b)}) \tag{3}$$

In the HDFS system which is designed to maintain even less metadata, building and maintaining such structures is still an open research problem [52, 50, 49, 57].

We decided to use sampling methods to estimate join selectivity. We do this because reading the entire relations to generate, or to maintain exact statistics about them would represent totally different problems than the one this work aims for. The main idea is to sample a small subset of tuples from a large relation to measure the number of sample tuples that satisfy a specific predicate, then, to extrapolate such measurement to the whole relation. In this way, a sampling-based method can be favorable for estimating the selectivity for join queries by avoiding repetitive table scans.

There are two types of sampling-based methods for estimating selectivity studied in relational database systems: Sequential sampling technique [44, 79, 80] or a double sampling technique [55]. The former is characterized by how sample gathering is done and by its stopping condition. It obtains sample units one at a time, but checking the outcome of each sample in order to determine whether an additional sample unit is to be taken or not. Its stopping criterion is based on setting a lower bound on the required sample size for a given error constraint.

The latter performs sampling in two stages. In the first stage, it gathers a few sample units in order to determine some previous information about the data like mean and variance. Then, based on this information, it computes the needed sample size to guarantee that the estimate meets the precision requirement with a certain confidence level. In the second stage, some other sample units are taken for the final estimate to be computed. One of the drawbacks of the double sampling technique is that there are not guidelines to determine the amount of sampling to be done in the first stage which is fundamental to the estimation accuracy. This is why we decided to use a sequential sampling method for estimating join selectivity for our model.

PUC-Rio - Certificação Digital Nº 0912867/CA

Lipton et al. [80] also use a sequential sampling method for their join selectivity estimation strategy. For instance, if we consider the join query $Q$ : $t_{R_1 \bowtie R_2}$ and the Cartesian product $R_1$ X $R_2$ as their data population. The authors' method considers each tuple $r$ from $R_1$ as a sample unit which is $S_r = t_{\mathcal{X}|\mathcal{X}}$ *is a concatenation of r and a tuple in* $R_2\}$ *in the Cartesian product* $R_1$ X $R_2$. Lipton et al. call $\bar{S}_r$ the tuples in $S_r$ that satisfy predicate $F$. In that way, the authors draw a tuple $r$ from $R_1$ (i.e. a sample unit $S_r$ from $R_1$ $x$ $R_2$) and evaluate the qualified tuples $\bar{S}_r$ in $S_r$ until a certain amount of tuples obtained are enough to meet the stopping criterion.

The authors use two stopping conditions, one for the cases in which a fair amount of elements belonging to $S_r$ have a non-empty $t_{\bar{S}_r}$ i.e. enough elements from the Cartesian product satisfy predicate $F$. And the other stopping condition is when just a few $S_r$ have non-empty $t_{\bar{S}_r}$ i.e. too many elements have been sampled and there are not too many matching tuples.

The work proposed by Lipton et al. was modeled for a centralized DBMS where many access methods (indexes, views, among others) can be used to facilitate sampling tuples from $R_1$ and to avoid having to scan the entire relation. This scenario is totally incompatible with our problem scenario where such access methods do not exist yet. A scenario which shares some similar features with ours is the one studied by Multi Database Management Systems (MDBS) [14] as previously discussed.

In this manner, Zhu [112] extends the work done by Lipton et al. by applying the techniques proposed by them to a MDBS scenario overcoming difficulties such as not knowing the local structure of a relation, or not being able to modify it. The author explains that the lower bound of complexity of Lipton and Naughton's method in an MDBS is $N*(|R_1|+|R_2|)$ because every time a sample tuple from $R_1$ has to be drawn the whole table would probably be fully scanned. Likewise, to find the tuples in $R_2$ that match a sample tuple in $R_1$, the whole table $R_2$ is usually scanned. Therefore, Zhu improved the join selectivity strategy by using systematic sampling to draw sample tuples from $R_1$. Systematic sampling works with a relation $R$ with cardinality $N$ whose tuples can be accessed in ascending/descending order on the join attribute(s) of $R$; decide on the size n of the sample relation; to produce the sample relation, select a tuple at random from the first $k = \lceil \frac{N}{n} \rceil$ tuples of $R$ and every $k$th tuple thereafter [46].

Furthermore, Zhu extends [79, 80] in two ways. First, by using a buffer to hold column values of a number of samples tuples from $R_1$. And second, by finding the tuples from $R_2$ that match with a certain number of sample tuples from $R_1$ in all scans of the table $R_2$.

PUC-Rio - Certificação Digital Nº 0912867/CA

The algorithm to calculate the join selectivity used by [112] is shown below. For better understanding, let us consider the join query $t_{R_1 \bowtie_f R_2}$, where f is $R_1.a = R_2.b$, $B$ represents a buffer that can hold $m$ values of $R_1.a$, assuming $m < |R_1|$, and $K = \lceil \frac{|R_1|}{m} \rceil$. And for the stopping condition we denote $k_1$ as a value associated with a given confidence level, *total_match* is the number of qualified tuples accumulated so far, *total_sample* is the number of samples made, $b$ is the maximum size of all $r$ and $e$ is the desired limit of a relative error.

```
Zhu's Join Selectivity Algorithm
Pseudocode
        total_match = 0;
        total_sample = 0;
        while ( total_match $t_{<k_1 \cdot b \cdot (1+e)/e^2}$ ) ) {
            systematic_sampling();
            count the matching tuples from $R_2$ with the sample tuples from $R_1$;
            total_match = total_match + x;
            total_sample = total_sample + m;
        }
        selectivity_estimate = total_match / (total_sample * $|R_1|$);
```

```
Systematic Sampling Process
Pseudocode
        systematic_sampling(){
            choose a random number $\gamma$ between 1 and K;
            obtain all $R_1$.a from $R_1$;
            if K = $|R_1|$/m, hold in buffer B the values of $|R_1|$.a for the $\gamma$th tuple,
            the ($\gamma$+K)th tuple, ...m and the ($\gamma$+(m-1)*K)th tuple retrieved from $R_1$;
            if K > $|R_1|$/m, hold in buffer B the values of $R_1$.a for the $\gamma$th tuple, ... ,
            the ($\gamma$+(m-2)*K)th tuple, and a tuple randomly chosen among
            ($\gamma$+(m-1)*K)th ...$|R_1|$th tuples retrieved from $R_1$;
        }
```

The step of counting the matches from $R_2$ with the sampled tuples from $R_1$ is implemented as a SQL query in Zhu method. His query finds the number of the tuples in $R_2$ that match at least one of the m sample tuples from $R_1$ during one scan of $R_2$. This is because their method focuses on MDBS that will work with possibly heterogeneous DBMS, but that they will all share the same query language. In our case, we cannot assume this due to the fact that

PUC-Rio - Certificação Digital Nº 0912867/CA

our relations are stored in the HDFS which does not possess any type of easy and fast access to the data e.g. indexes, views, query language, etc.

We performed cluster sampling [56] on $R_2$ to maintain a certain amount of data in memory to enable the counting of the matching tuples between $R_2$ samples and the ones from $R_1$. We do this due to the fact that $R_2$ might be too big to perform a full scan as proposed by Zhu where for each sampled tuple from $R_1$ a query obtaining the matches would be done on $R_2$. We decided to use "*cluster sampling*" to obtain $R_2$ samples because when a disk page is brought into memory, all tuples on the page are sampled incurring in less I/O costs. In addition to this, cluster sampling is more efficient because it avoids moving file blocks across the network while sampling. However, it also has disadvantages such as making the samples no longer independent which could make the join selectivity process incur in extra errors. Later, we will explain the parameters used in order to obtain a reasonable estimate without having to read too many tuples from relation $R_2$.

Another difference between the MDBS scenario and ours is that due to the fact that the files on which we work are stored on a distributed file system, the sampling process we used is different. In [112], the author performed systematic sampling over the relations, but in our situation, this could not be done in the same way because we are dealing with HDFS files. That is the reason why we decided to perform systematic sampling on each file block (a "*file split*" in the HDFS). We had to take care of not sampling a single block more than the rest of blocks. In order to sample the number of instances uniformly from each split, we divided the total number of instances to be sampled by the number of file splits, and obtain the same number of samples from each file split in a systematic way.

As for the stopping criteria used, we use the same stopping condition proposed by Zhu [112]. In such stopping condition, $k_1$ varies depending on the confidence level and if the central limit approximation applies or not [80]. We also assume this because the *central limit approximation* (CLT) states conditions under which the mean of a sufficiently large number of independent random variables will be approximately normally distributed i.e. the shape of the sampling distribution is approximately normal. Then, we let the maximum number of matches,$b$, be a percentage of the worst case which means it would be a percentage of the Cartesian product between the relations. For the results presented below, we set it as 0.00001 percent because of main memory limitations, but we are aware that this value should probably be set automatically by taking into account resources available in the computer. For example, if we have two big relations about 100 million records each, the

PUC-Rio - Certificação Digital Nº 0912867/CA

Cartesian product between them would be around $10^{16}$, and if each record is only 10Bytes, then we would need more than 80PB to store such worst case. That is why we chose such a small percent of the worst case.

At the same time as discussed in [80] there are situations in which a large portion of the total query size is due to a small portion of the samples i.e. the partitions sizes of the query are highly skewed. This occurs in our case due to the sampling method used for obtaining $R_2$ tuples, cluster sampling. Lipton and Naughton also proposed a solution for these situations which is to guarantee the error will be at most a fixed fraction of the worst-case size by setting a `sanity bound`: $total\_sample\ t_{>k_2 \cdot e^2}$ [80]. Here, $t_{k_2 \cdot e^2}$ is the number of samples taken, and $k_2$ also depends on the desired confidence level.

Table IV.3 shows the values for the parameters used in each test run. We run three different tests varying the default error from 0.1 to 0.9999. The parameter "*SAMPLE_RATIO*" is based on a sampling ratio due to the fact that we perform cluster sampling on $R_2$ instead of the query done in [112], and also because we let this relation, $R_2$, always be the bigger one. Therefore, we cannot sample the same amount of records from $R_2$ as we would from $R_1$ because 60% of $R_2$ could mean 100% or maybe even more of $R_1$. This is the reason why we use a sampling ratio between the two relations. In this way, if we decide to sample 30% of the smaller relation, $R_1$, this would mean that we will get 15% of the amount of records of the bigger relation, $R_2$. For example, if the smaller one has 10'000 records, and we decide to sample 20% of it, this means that we will get 2'000 records. But if the biggest relation has 1'000'000 records, and we sample the same percentage as in the smallest one, 20%, then we would get 20'000 records. Due to resource limitations and to keep our method efficient, we always tried to sample as few tuples as possible.

For this reason, we used a sampling ratio to define this relationship between the sampling percentages of these relations. Continuing with our example, if the sampling ratio is 0.5, then we would only sample 10% of $R_2$, i.e. 10'000 records, which is still a bigger quantity compared to the amount of samples taken from $R_1$, but not too big as to be unmanageable. We decided this due to resource limitations and due to the fact that we need to be able to create the model without incurring in high costs for reading too many elements of the relations. However, it would be interesting to test larger sampling parameters with more computational resources; this is another aspect we would like to investigate on future work. The "*DEFAULT_ERROR*" parameter is used as the desired limit of a relative error. Varying this parameter lets us experiment with the amount of samples needed to be taken to obtain good estimates. We varied this parameter due to the observations made by [80] in which $e$ was

PUC-Rio - Certificação Digital Nº 0912867/CA

Table IV.3: Parameters Used for Join Selectivity Sampling.

| | **QZ JS 0.1** | **QZ JS 0.5** | **QZ JS 0.9999** |
|---|---|---|---|
| **SAMPLE_RATIO** | 0.25 | 0.5 | 0.5 |
| **DEFAULT_CONFIDENCE_LEVEL** | 0.99 | 0.99 | 0.99 |
| **DEFAULT_SAMPLE_PERCENTAGE** | 0.6 | 0.5 | 0.5 |
| **MAX_PERCEN_NUM_MATCHES** | 0.00001 | 0.00001 | 0.00001 |
| **DEFAULT_ERROR** | 0.1 | 0.5 | 0.9999 |

set high to avoid "**sanity escapes**" i.e. sample more tuples in order to get a better estimate but taking care of not sampling too many of them.

The figure IV.6 shows a chart comparing the three runs varying the *"DEFAULT_ERROR"* parameter from 0.1 until 0.9999, and also the error of executing the simple selectivity count explained at the beginning of the section. As we can see, the simple count (called *Raw* in the figure) presents a higher variability than obtaining the join selectivity with the same strategy as [112]. Although this *raw count* shows some good estimates when joining small data, it becomes unstable when dealing with bigger data. On the other hand, the method proposed by [112] also presents high errors but it remains predictable. The relative error of the Zhu's method ranges from 19% until 99%. We attribute these errors to the lack of simple statistics such as cardinality, or average record length.

The chart also shows that with a higher default error, the estimates improve. This is due to the fact that with a higher error, "*sanity escapes*" are avoided i.e. we will sample more records but taking care of not sampling too many of them. These observations are similar to the ones found by Lipton and Naughton [80].

## IV.4 Experimental setup

We used the queries designed for each specific type of join to create two different set of experiments. The first experiment consists in performing cross validation of the model generated for each type of join. Cross validation is used for evaluating how the results of a statistical analysis will generalize to an independent dataset. In *K-fold cross-validation*, the original sample is randomly partitioned into *K* subsamples. Of the *K* subsamples, a single subsample is retained as the validation data for testing the model, and the remaining *K - 1* subsamples are used as training data. The cross-validation process is then repeated *K* times (*the folds*), with each of the *K* subsamples

PUC-Rio - Certificação Digital Nº 0912867/CA

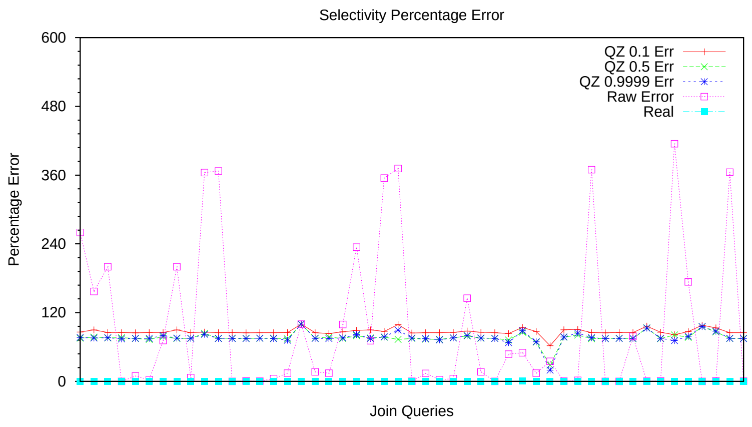

Figure IV.6: Selectivity Percentage Error varying according to table IV.3.

used exactly once as the validation data. The $K$ results from the folds then can be averaged (or otherwise combined) to produce a single estimation. The advantage of this method over repeated random sub-sampling is that all observations are used for both training and validation, and each observation is used for validation exactly once.

In our approach, our model was validated using a 3-fold validation due to the small size of our dataset, and to the long running times of our tests. In the next following sections we describe and discuss the results obtained from executing cross-validation on our data set of queries. Figure IV.7 represents this process graphically. We can see that the initial dataset is divided into three different parts, and then each part will be used for validation at least once during the process. The other two parts will be used to train the model.

The second experiment tries to simulate a real workload where many similar queries are executed in short periods of time. We decided to create such workload based on previous observations made by similar research on parallel computer systems. An important work is the one done by Downey et al. [27] and Li et al. [78]. In such works, the authors argue about the uniformity of users' actions and their predictability in short periods of time. This behaviour allows better predictions to be made which could help improving the whole system. Based on these observations, we created three workloads, one for each type of join implemented on the Pig framework.

We divided each group of queries in six subgroups with the same number of queries. Then, we chose a subgroup to be replicated five times, two other groups to be replicated three times, and we did not replicate the other three

PUC-Rio - Certificação Digital Nº 0912867/CA

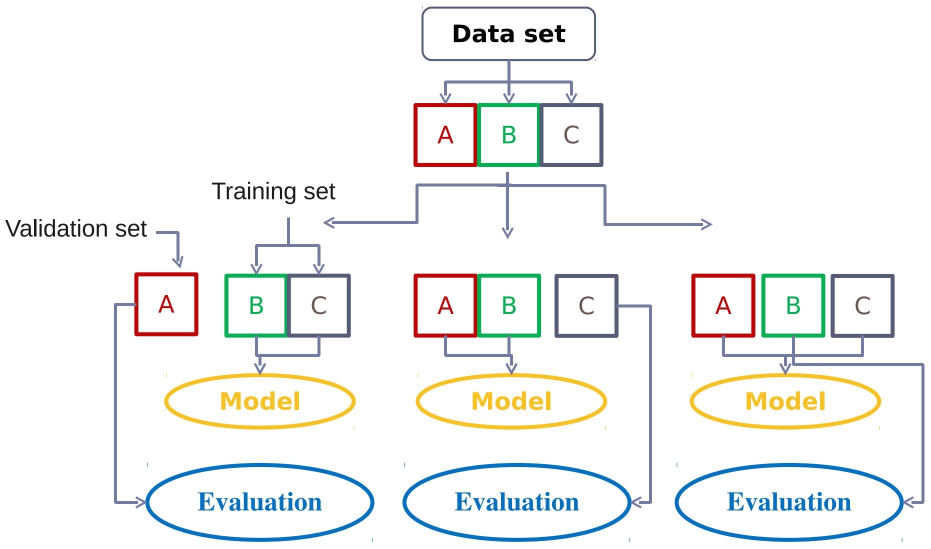

Figure IV.7: Graphical Three-Fold validation.

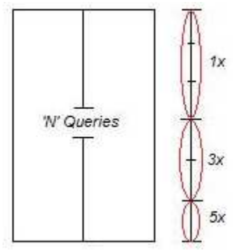

Figure IV.8: Workload structure.

PUC-Rio - Certificação Digital Nº 0912867/CA

subgroups. In this way, we created a workload simulating a specific group of join queries being the most commonly used, another group being less common, and a last group of queries being the ones rarely executed. Figure IV.8 shows the structure of such workloads.

As a result of this, we obtained 238 queries for the hash join operator, 232 queries for the fragment replicate join, and 41 for the merge join. In addition to this, we chose systematically 10% of queries from the original group in order to obtain a subset of queries to validate the model created from each workload.

As far as the hardware configuration used, we employed three computers with 4GB of RAM each running Ubuntu 10.4. We mounted a small Hadoop cluster on them using Hadoop 0.20.2, and obtained almost 1.5 TB of storage from these commodity machines. For executing the join queries, we used Apache Pig framework version 0.7 which we installed as a complement for our Hadoop cluster.

The data used is up to 3.5GB because we used `Scale 1` for the TPC-DS which means the data is up to a gigabyte. Furthermore, we used the default

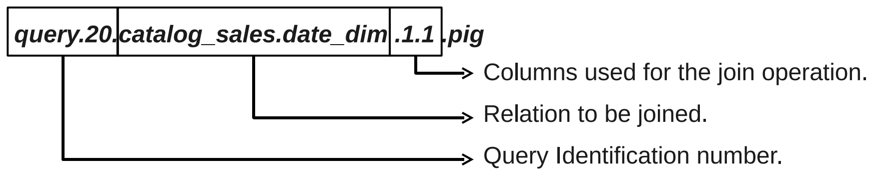
query.20.
catalog_sales.date_dim
.1.1
.pig
Columns used for the join operation.
Relation to be joined.
Query Identification number.

Figure IV.9: Query naming convention.

replication level of the HDFS, 3 which means that for each gigabyte used, we would need three gigabytes for storing it. That explains why our data set is about 3GB.

Another problem faced while building our query set was how to keep track of the queries once they were executed. In order to accomplish this task, we decided to name each query with the relations' names and with the columns used in the join operation. We also did this to be able to estimate the join selectivity for each query. Figure IV.9 shows a query name and what each part of it represents.

In addition to this, we executed the model construction process two different times. The first time we generate the model, we did not use the join selectivity parameter, and the second time we did use it. This was done in order to be able to compare both models, but also to see the relevance of the join selectivity parameter for the model. In the next section, we will discuss the results obtained for each type of join on each workload.

## IV.5 Results

A problem we noticed while running our tests was the number of reduce processes used. This is because this feature does not vary along query execution due to the fact that the number of reducers used depends on cluster's capacity if not set specifically to a lower number. The use of this feature makes all jobs linearly dependent which in turn causes the model construction process to fail because linear regression in the presence of collinearity can produce unstable estimates. That is why we decided not to use the number of reducers as part of our model as we work with the maximum number of reducers available. In the following sections, we will describe the results obtained, and discuss the problems and insights got from the experiments.

PUC-Rio - Certificação Digital Nº 0912867/CA

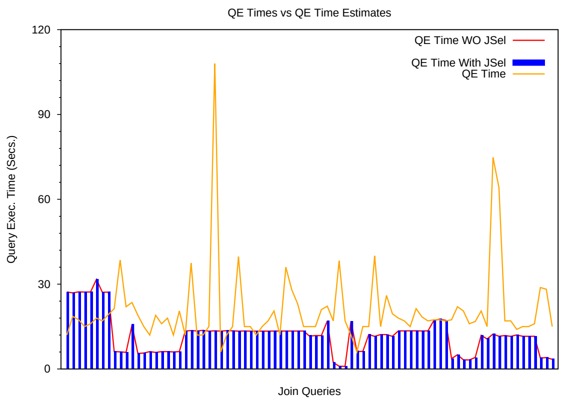

Figure IV.10: Query execution times against time estimates for fragment replicated joins.

## (a) Cross Validation

The diagrams presented in this section show the difference between the real query execution times against the estimated query execution time for the 3-Fold validation process. We performed such validation on three types of join operators implemented in the Pig-Latin Framework.

### Fragment Replicated Joins

We executed 96 queries of the fragment-replicated joins but using 32 queries for validation on each fold. We plot the results in the two charts below. The former IV.10 shows the query execution times (*QE Time*) and the estimates obtained using the join selectivity parameter (*QE Time With JSel*), and not using the join selectivity parameter (*QE Time WO JSel*). A first observation is that the variation between the estimates gotten using or not the join selectivity parameter could almost pass unnoticed. This is because as explained before, the error within our join selectivity parameter prevents the model to take full advantage of it. There are two jobs that took longer than the rest of them which are the jobs that involved moving more data through the network. These jobs had to replicate relatively big files to the other nodes where the join operation would occur. The files replicated were 9.9MB and 13MB respectively.

The chart IV.11 shows the query execution times from the 80th percentile of the jobs and their execution time estimates because by restricting our results to it we can omit the jobs which suffered from performance problems, execution environment changes, and poor linear models. For example, the jobs that in our case suffered from network latency which is a key piece for the MapReduce

PUC-Rio - Certificação Digital Nº 0912867/CA

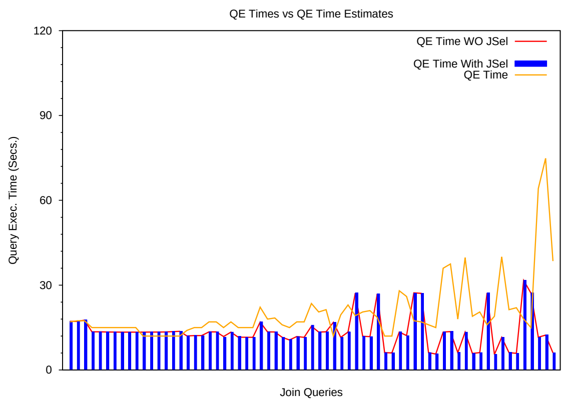

Figure IV.11: Query execution times against time estimates for the 80th percentil of the executions.

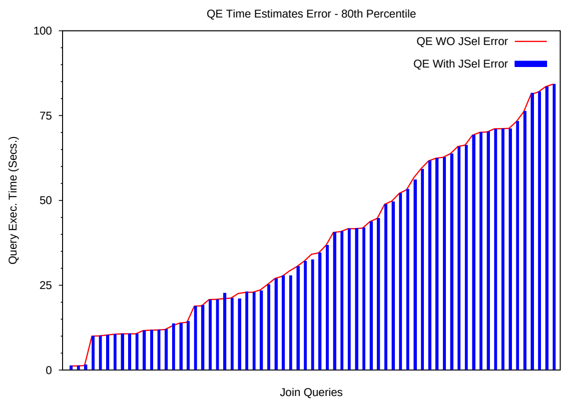

Figure IV.12: Error percentage for the 80th percentil of the executions.

PUC-Rio - Certificação Digital Nº 0912867/CA

framework.

Finally, the chart IV.12 shows a comparison between the relative error from the two different estimates, using the join selectivity parameter and not using it. The error goes from less than 2% until 205% for all jobs, but for the 80th percentile the error ranges from less than 2% until almost 85%.

## Merge Joins

We executed the 23 queries created of this specific join operator, and used 8 queries for validation. One of the biggest problems for creating such queries is the difficulty of creating join operation between ordered data when most of it is not. This type of join needs both relations to be ordered ascending in the join key. As many of the relations of our data set have the primary key - foreign key relationship, they do share the same key, but they are not sorted

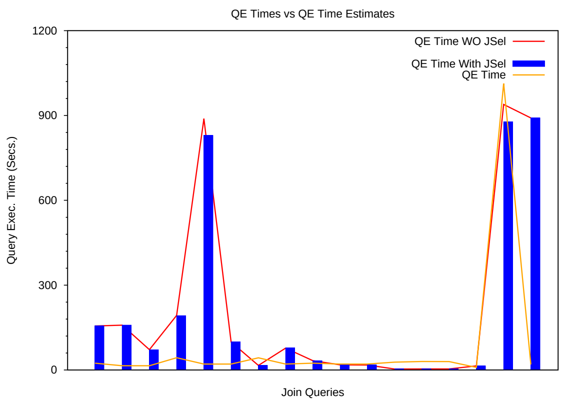

Figure IV.13: Query execution times against time estimates for merge joins.

on the same way on both relations.

The diagram IV.13 shows the differences between the real query execution times (*QE Time*) against the estimates obtained using the model generated with and without the join selectivity parameter (*QE Time With JSel*, and *QE Time WO JSel* respectively). Similarly to the Fragment Replicate joins experiment, the estimates obtained by using the join selectivity parameter do not differ considerably from one to the other. There were two estimates in which the model predicted long execution times. The first one is due to the fact that a self-join of the third biggest relation (226MB) was performed, and in spite of that, the query was executed in a very fast manner, it took only 21 seconds to be completed. We hypothesize that this fast execution was due to the fact that the column used for joining is sufficiently skewed for the implementation of the merge join operator take advantage of it to perform better than in the rest of the situations. The other high point in the chart is also a join of the third biggest relation with a relation of 10MB. This joins execution time is expected due to the size of the input and to the environment variables (number of computers, network latency, etc.), and that is why the predictions made on it do not defer too much from what it actually took. It differs 7% with the prediction which did not used the join selectivity parameter, and 13% with the one which used the join selectivity parameter.

The figure IV.14 shows the real query execution times against the estimated values for the 80th percentile of the queries. In this case our error ranges from 7% to 378% for the model not using the join selectivity operator, and for the model using the join selectivity operator its error ranges from less than 14% to 423%. This difference in the error from both models is due to the fact that join selectivity parameter has already the error of the cardinality

PUC-Rio - Certificação Digital Nº 0912867/CA

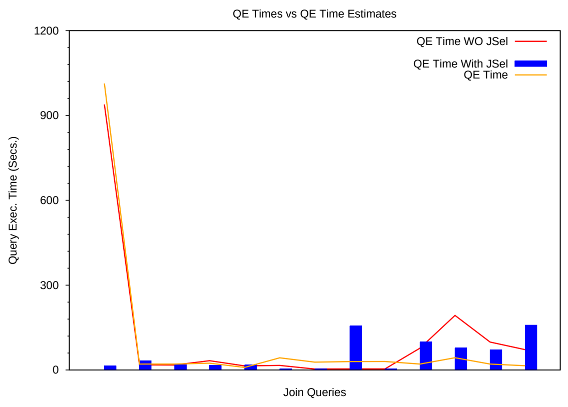

Figure IV.14: Query execution times against time estimates for the 80th percentil of merge join executions.

estimate in addition to the its estimate itself.

The graphic IV.15 shows the percentage error obtained for the 80th percentile of the queries using both models (applying the join selectivity parameter (*Error With JSel*), and not applying it (*Error WO JSel*)). The highest points in the graphic are due to the fact that the merge join operator takes advantage of not having to read all of the big relations, but instead it samples one of the relations and creates a sparse index on it. And then, it uses this index to access the block directly at probing time. This makes this operator the least memory intensive algorithm because it does not have to load a whole block of data into memory. An interesting thing about the graphic is that the error that the model which uses the join selectivity parameter seems to have a more constant behavior compared to the one which uses the join selectivity parameter. We hypothesize that this is also due to the error accumulated into the join selectivity parameter which ends up not being so useful for building the model.

### Hash Joins

We executed 102 queries of the hash join operator using 34 queries for validation. Similarly to the other join operators, we will describe the charts comparing the actual execution time against the estimated execution time using the models built. The chart IV.16 shows the difference between the real execution time (*QE Time*), and the prediction values using the join selectivity parameter (*QE With JSelectivity*) and not using it (*QE WO JSelectivity*) for building the model. One important observation about it is that the actual execution times where really fast compared to the estimated times. We think

PUC-Rio - Certificação Digital Nº 0912867/CA

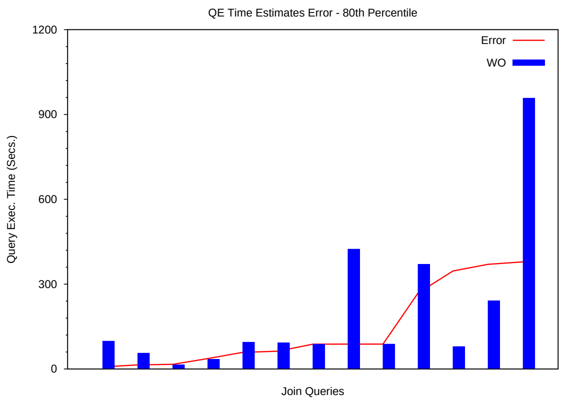

Figure IV.15: Error percentage for the 80th percentil of the merge join executions.

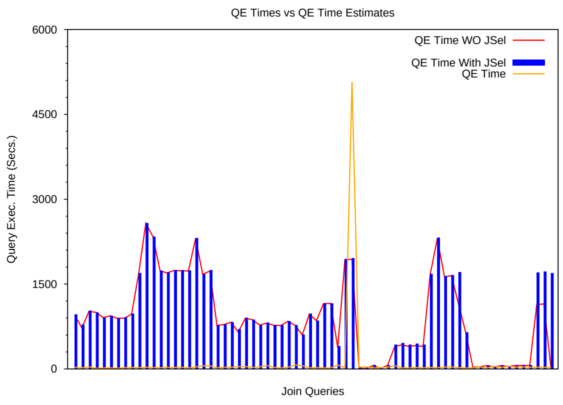

Figure IV.16: Query execution times against time estimates for hash joins.

PUC-Rio - Certificação Digital Nº 0912867/CA

this is due to the fact that this operator relies heavily on the hardware i.e. on main memory for maintaining and probing tuples from one relation against the other.

And even though we decide to keep only the 80th percentile of all the queries, the diagram IV.17 does not change substantially. The real execution times are still very fast compared to the estimated execution times.

Therefore, we expected high errors in our estimates. The graph IV.18 compares the estimate's error for the cases of using or not using join selectivity as a parameter of the model. The difference between both estimates is that the join selectivity parameter causes the query execution time to vary along the plot. This is because the error within such extra parameter is probably too high for the model to take full advantage of it.

This huge difference between our estimates and the real values motivated

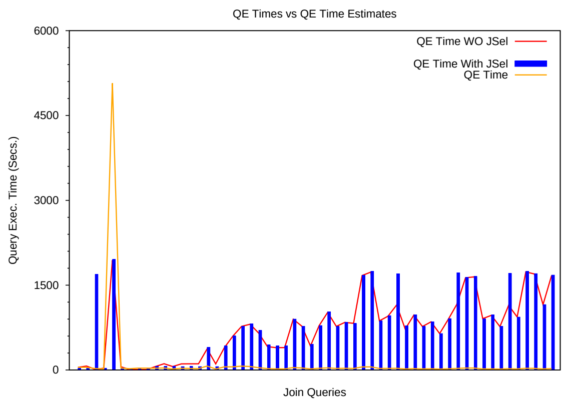

Figure IV.17: Query execution times against time estimates for the 80th percentil of hash join executions.

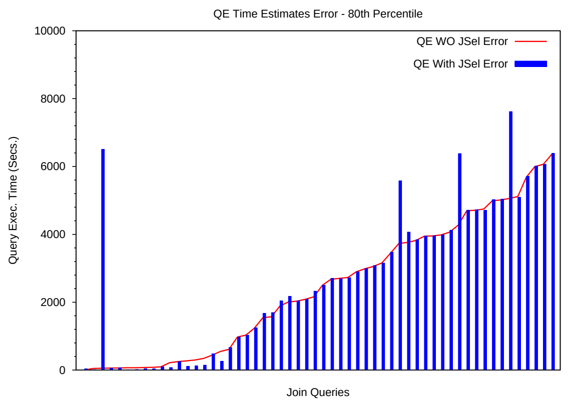

Figure IV.18: Error percentage for the 80th percentil of hash join executions.

PUC-Rio - Certificação Digital Nº 0912867/CA

us to search for an explanation. That is the reason why we decided to look into the actual jobs executions. In the picture IV.19, we show the variance between the different query executions of the cross-fold validation. We recorded each execution time from each run and then compared them against the other query executions. As we can see in the picture, the job execution time varies in hundreds of orders of magnitude from each different execution. This implies that this join operator heavily depends on how the cluster is being used in the specific moment when the hash join is executed. It varies because this join operator always loads into memory the relation from the left, and the other one is streamed through. Besides that, the Pig framework expands four times the size of data when loading it from disk into memory i.e. if it loads a 1KB file, the Pig framework will need 4KB of memory for it in memory [39]. This join operator is the less stable compared to other operators because it heavily

PUC-Rio - Certificação Digital Nº 0912867/CA

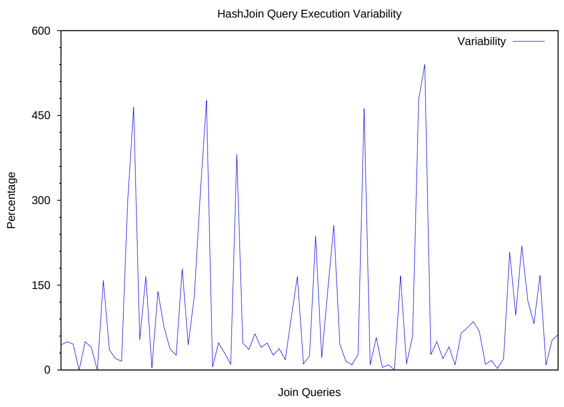

Figure IV.19: Hash join query execution variability.

depends on main memory which can vary along query execution.

This job variability shows why the model built from the job execution times are not able to characterize this particular operator. We might need a bigger dataset to be able to model this operator successfully. Another interesting way to test this would be by using a more robust execution environment i.e. using a cluster which supports loading the relations into memory without degrading considerably its performance. In the next section, we will describe the results obtained by simulating a workload from the queries created.

## (b) Workload Simulation

We decided to create and extra set of experiments by simulating a workload from the queries created. We did this by replicating the queries in different proportions in order to resemble query usage of the system. The construction of such workload was explained in the section IV.4. In this section, we will discuss the results obtained and the problems encountered while attempting to predict query execution times.

### Fragment Replicated Joins

We executed 232 queries using the fragment replicated joins. Seventeen of them simulated to be the most used ones then they were replicated five times in the workload; thirty-four of them simulated to be the second most used ones as they were replicated three times, and fifty-one queries were not replicated in order to simulate the queries rarely executed. The chart IV.20 shows the query execution times (*QE Time*) against the estimates made using

PUC-Rio - Certificação Digital Nº 0912867/CA

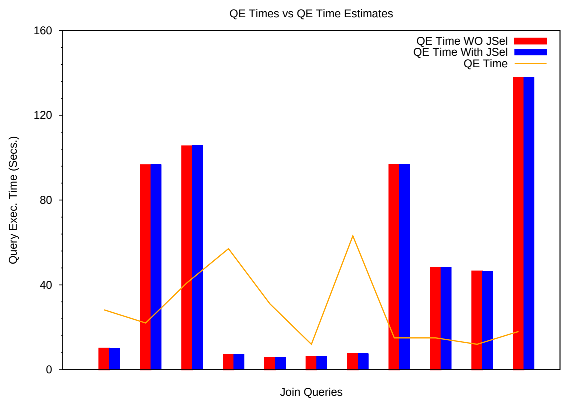

Figure IV.20: Query execution times against time estimates for the simulation workload of fragment-replicated joins.

the join selectivity parameter and not using it (*QE Time With JSel* and *QE Time WO JSel* respectively).

We can notice the differences between the real values and the estimates are really high. The relative error ranges from 48% to over 660%. We hypothesized that this high errors are due to poor linear models, and to the size of the query set constructed i.e. the number and variety of the queries executed which probably led to overfit the model.

### Merge Joins

We executed 41 queries using merge joins. Three of them simulated to be the most used ones then they were replicated five times in the workload; six of them simulated to be the second most used ones as they were replicated three times, and eight queries were not replicated in order to simulate the queries rarely executed. The chart IV.21 shows the query execution times (*QE Time*) against the estimates made using the join selectivity parameter and not using it (*QE Time With JSel* and *QE Time WO JSel* respectively).

Similarly to the fragment replicate join, the estimates done for the merge join operator also presented really high relative errors that range from 164% to almost 9'000%. We attribute these high errors to the even smaller quantity of queries used. The main problem using this join operator is the preconditions it poses on the data on which it will work e.g. both input relations having to be physically ordered by the join key in ascending order. This fact obstructs the construction of a bigger dataset for building the model.

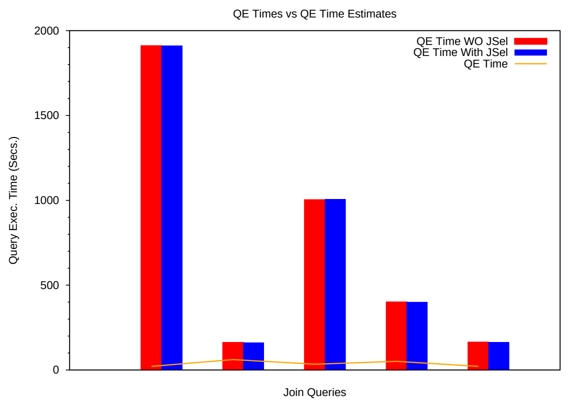

Figure IV.21: Query execution times against time estimates for the simulation workload of merge joins.

## Hash Joins

We executed 238 queries using the hash join operator. Seventeen of them simulated to be the most used ones then they were replicated five times in the workload; thirty-four of them simulated to be the second most used ones and they were replicated three times, and finally, fifty-one queries were not replicated in order to simulate the queries rarely executed. The chart IV.22 shows the query execution times (*QE Time*) against the estimates made using the join selectivity parameter and not using it (*QE Time With JSel* and *QE Time WO JSel* respectively).

The simulated workload for the hash join operator provides similar results to the ones obtained by the other two types of join operators. The relative error ranges from 23% to over 1500%. We consider that these high errors are due to the job execution variability, and to the same problems encountered while performing this experiment with the other join types. There is only one job query execution estimate which seems more accurate than the rest of them. This job performs a join between a relatively big relation (141MB) and a small one (8.6KB), and this query belongs to the group of queries that were not replicated within the workload. We consider this relatively accurate estimate as prove that the model is overfitted because the model seems to be exaggerating minor variations in the data.

We noticed that this last experiment results do not led us to any conclusive assumptions about the workloads used. This is due to a common problem with statistical methods known as overfitting the model. Overfitting usually occurs when a model has too many parameters compared to the number of observations, and an overfitted method usually performs poorly at predicting

PUC-Rio - Certificação Digital Nº 0912867/CA

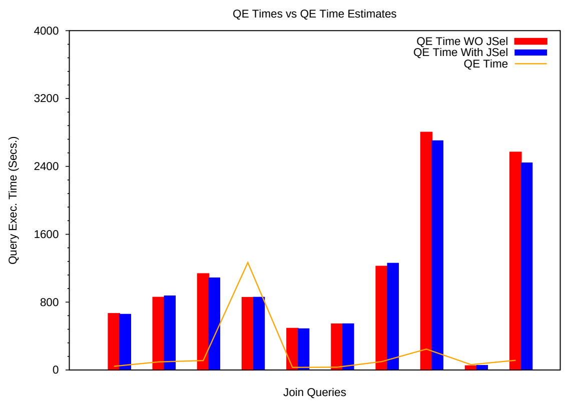

Figure IV.22: Query execution times against time estimates for the simulation workload of hash joins.

values, and can also misrepresent small fluctuation within the data.

A solution to this problem is to create a richer dataset i.e. a larger quantity of queries, and also with more diverse data distributions. Another solution to such a problem is to use shrinkage methods [95] which are intended to reduce the overfitting problem by making more modest predictions, but which are closer to the average.

PUC-Rio - Certificação Digital Nº 0912867/CA