

# Towards a Statistical Evaluation of PigLatin Joins

Renato Javier M. Mogrovejo<sup>1</sup>, José Maria Monteiro<sup>2</sup>, Javam C. Machado<sup>2</sup>, Carlos Juliano M. Viana<sup>3</sup>,  
Sérgio Lifschitz<sup>3</sup>

<sup>1</sup> Universidade Catolica San Pablo (Arequipa), Peru  
rmarroquin@ucsp.edu.pe

<sup>2</sup> Universidade Federal do Ceará (Fortaleza), Brasil  
{monteiro,javam}@ufc.br

<sup>3</sup> Pontificia Universidade Católica do Rio de Janeiro, Brasil  
{cviana,sergio}@inf.puc-rio.br

**Abstract.** There are different scalable data management solutions which can take advantage of cloud features making them more attractive for a deployment in such environments. One of the most critical operations in data processing is joining large data sets. This is one of the most expensive and hardest operations to optimize. We are mainly concerned here with join operations expressed in PigLatin, an abstract query language for a high-level platform, called Pig, which creates **MapReduce** programs with Hadoop. In this work we explore statistical methods, namely multiple regression analysis, in order to predict three distinct join types execution times, comparing them with actual running times.

Categories and Subject Descriptors: H.2 [Database Management]: Experimentation

Keywords: Cloud databases, MapReduce, PigLatin, Hadoop, Join Evaluation, Multiple Regression

## 1. INTRODUCTION

An important contribution to large scale data processing has been the **MapReduce** programming model [Dean and Ghemawat 2004]. It was designed to deal with fault-tolerant systems, to enable high-availability and to run on heterogeneous hardware. It has become almost a *de facto* standard for massive data set computations in the cloud. However, the amount of code generated to perform relatively simple data management tasks such as projections and filters may become a problem. For instance, join execution in Hadoop (an open-source platform for the **MapReduce** paradigm) is a very hard task for users even though joins are one of the most critical operations in data processing. Along with known problems such as estimating the output result size and join selectivity, new challenges arise when considering **MapReduce** abstractions e.g. data localization and skewness.

Research projects like Pig [Olston et al. 2008] and Hive [Thusoo et al. 2009] try to make things simpler by creating an abstraction layer for **MapReduce** programming tasks. Hive is a data warehouse that uses Hadoop Distributed File System (HDFS) for storing its data, with a SQL-like language called HiveQL to access the data. Pig works on top of Hadoop framework [hadoop 2010] using a high-level programming language called **PigLatin**. **PigLatin** has some similarities with **SQL** because both perform some similar operations from relational algebra, but they aim at different goals. **PigLatin** expresses transformations as a sequence of steps while **SQL** describes the desired outcome. This feature is considered an advantage because writing a sequence of steps is more natural to developers. This procedural view allows them to write more complex data-flows when compared to plain **MapReduce**, or even trying to express their data-flows with a regular data oriented language.

---

Copyright©2012 Permission to copy without fee all or part of the material printed in JIDM is granted provided that the copies are not made or distributed for commercial advantage, and that notice is given that copying is by permission of the Sociedade Brasileira de Computação.

These high-level programming languages help improving the usability of **MapReduce** by simplifying the use of **MapReduce**'s API and tuning parameters as they provide a query language similar to traditional structured query language (SQL). In spite of the fact that these programming languages abstract and simplify some **MapReduce** complexity, they still need further improvements. For example, cost-optimizers are still work in progress and most of the data is stored with no corresponding metadata, among other extra features of traditional database systems. We explore a statistical approach to find relationships among the known information about our data set and usual data center infrastructures. We evaluate query execution time by using a set of join queries from the TPC-DS database benchmark. We hypothesize that using statistical models we could run "*what-if*" workload scenarios by analyzing an input workload and measuring its performance in order to predict it.

This article aims to investigate the expected efficiency of systems like Pig and Hive, by using a statistical method for modeling join operations' execution time and, then, being able to predict their performances. In our approach, the construction of such a model is done by building up a training data set and performing cross-validation to verify it.

One of the challenges encountered in this research work was the lack of a data set to evaluate specific operations in large scale systems. We used a known database benchmark, TPC-DS, to simulate real data with real data distributions. Nevertheless, queries proposed in TPC-DS are not suitable for systems based on the **MapReduce** programming model. To overcome this issue, we have converted relations described in the TPC-DS into **PigLatin** queries to specifically evaluate join operations.

This article is structured as follows: Section 2 presents the motivation, some background information and related research works. In the following section, we present the Hadoop Joins as they are implemented by the Pig system. In Section 4, the approach and design decisions about our main proposal are explained. We give also some experimental results obtained as well as the insights gathered from them. Finally, Section 5 presents the conclusions of this work and discusses some future work.

## 2. BACKGROUND AND MOTIVATION

In order to take advantage of the *infinite* resource supply offered by the cloud computing paradigm, specific parallel query languages were needed. A very important contribution to the cloud computing research area is the programming model **MapReduce** [Dean and Ghemawat 2004]. It has become a *de facto* standard for massive data set computations in the cloud, a nice abstraction to facilitate distributed coding to programmers. The **MapReduce** framework provides characteristics such as fault tolerance, load balancing and task parallelization in a transparent way.

### 2.1 Hadoop, Pig and PigLatin

There are many implementations of this distributed computing paradigm. One of the most used is the open source framework Hadoop [hadoop 2010], created to support the Nutch [Agarwal et al. 2010] search engine initiative. Later, it was taken by Yahoo!, now Apache, in order to address critical business needs like managing increasing volumes of data. Hadoop is used in many other open source projects due to its ability to process data in a distributed way while using commodity hardware.

To take full advantage of the distributed computation, **MapReduce** uses an underlying distributed file system, **Google File System** (GFS) [Ghemawat et al. 2003], to locally access data. GFS has a master/slave architecture with a large number of working nodes. All the files stored in the GFS are broken up into fixed size chunks in order to have a better performance in I/O operations due to size similarity with sectors in regular file systems. These file chunks are then stored and distributed along with all the available working nodes, or chunk servers, while the metadata associated with all of them is kept in the Master node. There is also an open-source implementation of GFS that works along with the Hadoop framework called **Hadoop Distributed File System** (HDFS) [Nutch 2011].

In the open source world, Pig and Hive are important projects. Particularly, Pig is an abstraction layer of the Hadoop framework that compiles data analysis tasks into **MapReduce** jobs and executes them on Hadoop. Pig uses a language for expressing its dataflow called *PigLatin*. One of *PigLatin*'s biggest advantages is that it enables programmers to express data transformation tasks in few lines. *PigLatin* also differs from SQL in which the former allows expressing transformations as a sequence of steps, and the latter describes desired outcome. This is seen as an advantage because writing a sequence of steps comes natural to developers and allows them to write more complex data flows.

For example, consider the following SQL query that filters the data by applying a user-defined function called 'Clean'. It then counts the number of remaining entries in the log per region.

```
SELECT U.regions, count(*) as total
FROM UsersRegistry U
WHERE Clean(U.query) = 10
GROUP BY U.regions
```

In PigLatin, this example could have been written as:

```
user_registry = LOAD 'UsersRegistry.dat'
                AS (userId, userRegions, userQuery);
users_filtered = FILTER user_registry BY Clean(userQuery);
users_groups = GROUP users_filtered BY userRegions;
output = FOREACH users_groups
           GENERATE $0 AS regions, count($1) as total;
STORE output INTO 'results.txt' USING PigStorage();
```

These works on Pig, Hive and Hadoop show that high level programming languages for programming large-scale parallel computations are going to be more predominant than plain **MapReduce** jobs.

## 2.2 Related Work

Predicting job execution times is a known problem in the distributed systems field. In a very interesting research work ([Smith. 2007]) the construction of a system prediction services is explained. The system was built using an instance-based learning technique to predict job execution times, batch scheduling queue wait times and file transfer times of the *Texas Advanced Computing Center Lonestar System* [University of Texas 2010]. Even though the approach shows relatively high prediction error for the training data (ranging from 37% to 115%), the author discusses that it would be possible to reduce the prediction error by using a larger set of training data. However, it might be possible that the system on which they are working might be simply less predictable than other systems better performances. Nevertheless, the results obtained show a lower prediction error in more structured and well behaved workloads when compared to the work discussed in [Vazhkudai et al. 2002].

On the other hand, estimating query execution time is also a very interesting research topic for the database community due to the fact that this information can help improving the whole system's performance, but can also lead to more proactive decisions of the DBMS (maintaining statistics, choosing better execution plans, etc.). In spite of the similarities of the database world with **MapReduce** environments, they are still different in many aspects, and a very important one is the amount of metadata kept by databases. **MapReduce** environments tend to keep only operational information (i.e. information they need to keep functioning) in order to remain simple, but reliable.

A known scenario with similar features to **MapReduce** environments is the one studied by Multi Database Management Systems (MDBS) [Bright et al. 1994]. For instance, Zhu [Zhu 1993] discusses

the need of good selectivity estimates before query execution so that the global query optimizer in a MDBS can choose a low cost execution plan from many alternatives. The author also presents an integrated method to estimate selectivity in a MDBS based on the discussion. He accomplishes this by applying the techniques proposed by Lipton et al. [Lipton and Naughton. 1995] to an MDBS scenario overcoming difficulties such as not knowing the local structure of a relation, or not being able to modify it. In [Zhu and Larson. 1994], Zhu et al. proposed a query sampling method to derive cost estimation formulas for autonomous local database systems in an MDBS. The main idea of their work is to classify local queries with similar costs, sample each query class, and to use the observed costs of the sample queries and statistically derive local cost estimation formulas for the queries performed on the local databases i.e. using query sampling for estimating local query costs in an MDBS. If the available information were not enough, they would classify queries with similar performance behavior into the same class so that their estimation errors would not be large.

Likewise Olston et al. [Olston et al. 2006] describe the needs in high-level dataflow languages built on top of large-scale parallel dataflow systems. Due to the fact that they provide a faster program development environment and an easier way to maintain the code, they also bring new difficulties and opportunities e.g. automatic optimization and query rewriting. The authors argue about the different possible types of optimizations in Data-Intensive Scalable Computing (DISC) [Bryant 2007] and in database systems. They also point out the differences that make these two contexts similar.

We claim that high-level dataflow systems using the **MapReduce** paradigm could benefit from the use of cost estimates. Those estimates should not have a negative impact on performance e.g. longer execution times, extra complexity. Then, they should be obtained using only known information of the job execution such as cluster capacity, input size, among other known execution parameters.

Therefore, we would like to explore different approaches to extract the relationships between input parameters and measured performance i.e. execution time. Using statistical methods to identify these relationships, we could extrapolate "*what-if*" workload scenarios using the statistical model created as a functional instance of the whole system. Our interest lies specifically on join operations because they are generally more time-consuming, and they are still user-driven in the Pig system.

### 3. JOIN TYPES IN THE PIG FRAMEWORK

The *join* operation tries to match records from two distinct input sets. Join execution is one of the workhorses of data processing and likely to be present on the majority of PigLatin scripts. There are four types of join implemented in the Pig Framework: **hash join**, **fragment-replicate join**, **merge join**, and **skew join**. We will evaluate only the first three because the skew joins depend on data with a skewed distribution. We would have to know *a priori* the underlying data distribution, which is not possible most of the time.

#### 3.1 Hash Join

This is the default join operator in the Pig framework. The main idea of this join is to use keys for each input. Then, when those keys are the same, the two records will be joined and the records that did not fulfill the condition are dropped. Fig 1 illustrates this main idea.

Pig implements this strategy by tagging each record with which input it came from in the map phase, and using the join key as the shuffle key. This type of join needs a reduce phase where all the records with the same value for the join key are collected together, and then it performs a cross product between the records from both inputs. Pig tries to minimize memory usage by making the records of the first input arrive first, cache them, so when the records of the second input arrive, they can be probed against each record from the left side to produce an output record [Gates 2011].

An important feature of the hash join is that when using a multi-way join all the inputs except

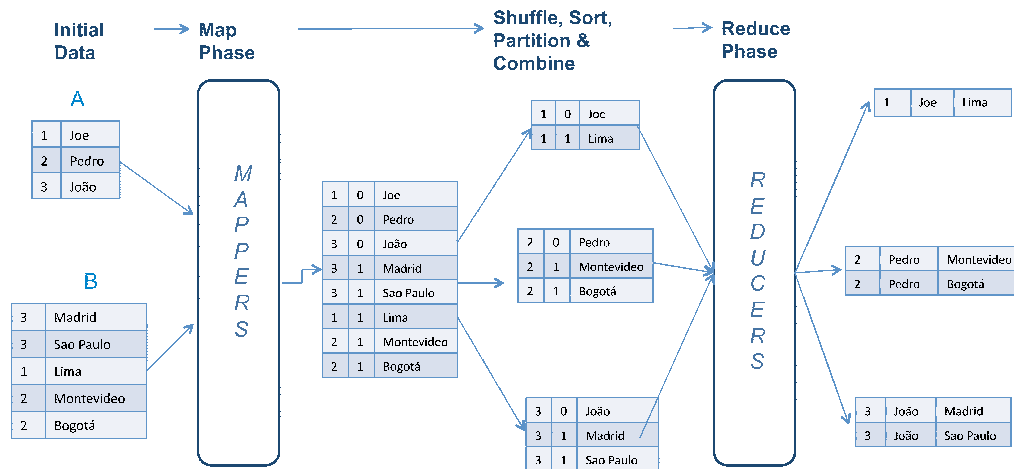


Fig. 1. MapReduce Hash Join

the last one will be held in memory. So, the last input can be streamed through. In this way, it is a good practice to place at last the input with more records per given value of the key. Thus, a better memory usage will be made and the Pig script's performance can be improved.

The code below shows how this type of join is implemented by a PigLatin script. The example shows the sequence of steps to perform a join between two relations, *inventory* and *item*. The first line shows how to load the "*inventory.dat*" file from the "*pigData*" folder, using the '|' character as column delimiter. The second line shows the same loading operation, but for the "*item.dat*" relation. And the third line shows the join operation between both relations.

```
--hash.join.pig
inventory = LOAD 'pigData/inventory.dat' using PigStorage('|') AS
    (inv_date_sk:int, inv_item_sk:int,
    inv_warehouse_sk:int, inv_quantity_on_hand:int);

item = LOAD 'pigData/item.dat' using PigStorage('|') AS
    (i_item_sk:int, i_item_id:chararray, i_rec_start_date:chararray,
    i_rec_end_date:chararray, i_item_desc:chararray);

join_inventory_item = JOIN inventory BY inv_item_sk, item BY i_item_sk;
```

### 3.2 Fragment-Replicated Join

We could run a routine task and perform lookups using a smaller input data. For instance, if we would have to process all the sales for a long period of time, where we would use the item's description rather than the item's code number to identify it. It would be easier to load all of our items into memory and then translate items' codes into their descriptions. This is because we have less items than the total number of sales. We could avoid a reduce phase by sending the smaller relation to every working node and performing the join locally without copying data through the network. This join strategy is called *fragment-replicate join* because we fragment one file (the bigger one, whose fragments will be processed by the mappers) and replicate the smaller one (that will be replicated to all the mappers).

In order to implement this join strategy, Pig takes advantage of a tool the Hadoop framework provides, the *distributed cache*. This tool enables us to send a file to the working nodes, and to pre-load it onto the local disks, so the mappers or reducers can access it if needed. The main advantage of using this tool is not straining HDFS when a small file is needed by many mappers. For instance, if a fragment-replicate join would need 100 mappers, then opening a file stored in HDFS from 100 different nodes at a specific moment will certainly put pressure on the NameNode and on the nodes which have the replicated blocks of that file. Such situations are the motivations for using the distributed cache so no extra pressure is put on the HDFS. Another situation where it is very useful is when multiple mappers are executed on the same working node because all these tasks can share the files in the distributed cache. Therefore, a file has to be copied a fewer times around the network.

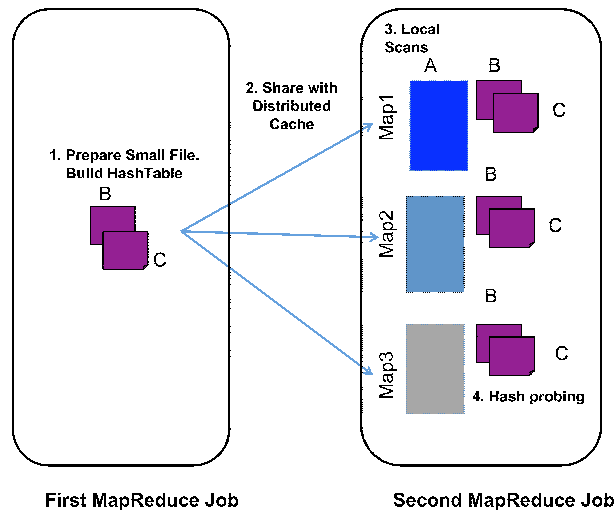


Fig. 2. MapReduce Fragment-Replicated Join

In addition, Pig executes a **MapReduce** job to pre-process the file in order to get it ready for sharing through the distributed cache. If there are other operations such as filtering, then these operations are also part of this initial job, so the file passed to the working nodes is as small as it can be. Therefore, the join operation itself will be done in the second map-reduce job.

The code below shows how this type of join is implemented by a PigLatin script. It is similar to the example presented for the hash join operator, but to tell Pig to use this specific type of join, we have to specify the type of join to use. We accomplish this by adding the "USING 'replicated'" hint.

```
--frag-rep.join.pig
web_sales = LOAD 'pigData/web_sales.dat' using PigStorage('|') AS
    (ws_sold_date_sk:int, ws_sold_time_sk:int,
     ws_ship_date_sk:int, ws_item_sk:int);

item = LOAD 'pigData/item.dat' using PigStorage('|') AS
    (i_item_sk:int, i_item_id:chararray, i_rec_start_date:chararray,
     i_rec_end_date:chararray, i_item_desc:chararray);

join_ws_item = JOIN web_sales BY ws_item_sk, item BY i_item_sk USING 'replicated';
```

When writing this type of join, we must be careful enough to always put the smaller input as the second one. This is because the second input listed will be always loaded into memory. If Pig cannot load it into memory, the join query will fail.

### 3.3 Merge Join

The sort-merge join is a join strategy in traditional relational databases and consists in sorting both inputs on the join key and scan them simultaneously performing the join. Sorting would require a full **MapReduce** job, as Pig's default join does; therefore it does not become the most efficient strategy. In spite of that, if both inputs are already sorted on the join key, then this join strategy could be the most efficient. In this case, both inputs could be opened in the map phase and iterate over them. That is why this strategy is known as merge join, but the sort has to be done before performing the join. The following code shows how this type of join is implemented by a PigLatin script:

```
--merge.join.pig
catalog_sales = LOAD 'pigData/catalog_sales.dat' using PigStorage('|') AS
    (cs_sold_date_sk:int, cs_sold_time_sk:int, cs_ship_date_sk:int,
     cs_bill_customer_sk:int, cs_bill_cdemo_sk:int, cs_bill_hdemo_sk:int);

date_dim = LOAD 'pigData/date_dim.dat' using PigStorage('|') AS
    (d_date_sk:int, d_date_id:chararray, d_date:chararray);

join_date_cs = JOIN date_dim BY d_date_sk, catalog_sales
    BY cs_sold_date_sk USING 'merge';
```

Fig 3 shows the Merge Join job schema execution, which runs a **MapReduce** job to get samples of the second input in order to build an index which will be used as the value of the join key. These samples are the first records of every input split i.e. from each HDFS block, indeed a very fast sampling process. Pig will use another **MapReduce** job which will use the first input (in this case `date_dim`) as its input. When each mapper reads the records in its split, it will take it and look it up in the index built by the previous job. It will look for the entry that is less than the value it read from the first input. Then, it will open the specific split of the second input pointed by the index entry.

Once the correct block of the second input is opened, Pig will start checking for a match of the index. When it finds a match, it gets all the records that matched into memory, and then perform the join. It then gets another record of the first input, and if the key is the same as the last one, then it performs the join. If it is not, it then checks the index to see which the proper file split is to get it and to check again. If it cannot find a match within the second input, it simply advances to another record of the first input and checks again. This type of join only supports currently two way joins and inner joins. It is also more efficient than a hash join because it can be done without a reduce phase.

Our work aims to explore statistical methods to help systems take decisions on which join operator to use, but without user's hints. We will present our approach and practical results in the next section.

## 4. STATISTICAL APPROACH AND PRACTICAL EVALUATION

The Pig system allows the use of four different types of join operators (merge, hash, fragment-replicated, and skew joins). However, to use them the user needs to pass a hint in PigLatin to the optimizer specifying the type of join to be used. This type of communication with the optimizer might be suitable for some experienced users who probably can choose wisely between these operators. However, for new users the constraints and advantages of each type of join operator may not be so

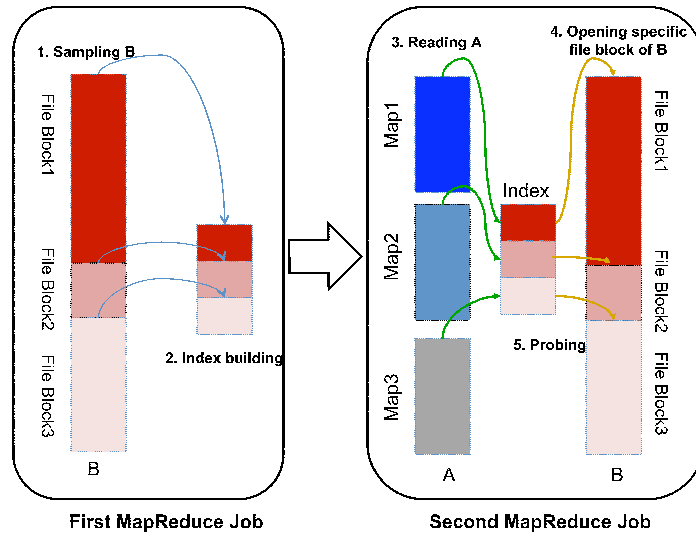


Fig. 3. MapReduce Merge Join

clear. In this way, new users can be mistaken when choosing the correct type of join operator, which can lead to bad performances and misused of computational resources.

One of the main goals of this research work is to investigate possible statistical methods that could fit well for use within our system architecture depicted in Figure 4. We may investigate the parameters involved in creating this statistical model, specifically the design decisions to model and evaluate execution times for the different types of PigLatin join operators. This model would help a rule-based optimizer to choose the most appropriate join operator for each situation.

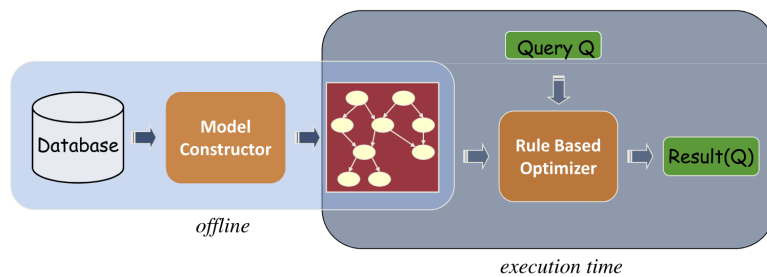


Fig. 4. System Architecture for Our Statistical Model

In that way, we see a possibility of applying studied concepts from the database field into large scale data processing. More specifically, we see a chance in integrating these prediction capabilities into the Pig system [pig 2010] to improve its query execution capabilities. Figure 4 shows how integrating these concepts into the system would be like. The left part of the figure exhibits the model constructor of the system. This component should use only *a priori* information about the input relations and the operations to create a model that characterizes them. Thus, we can use the model to obtain query execution times estimates beforehand. The right part of the figure shows how this *a priori* information can be used by the optimizer in order to make good decisions and create better execution plans.



#### 4.1 Choice of a Data Set

Research has also been done to evaluate different aspects of these non-traditional data stores. Shi et al. [Shi et al. 2010] benchmark five different tasks: data load, grep query, range query, aggregation and fault tolerance. Cooper et al. [Cooper et al. 2010] supply several workloads with different combinations of insert, read, update and scan operations. Nevertheless, all these workloads focus on systems like PNUTS, which provide online read and write access to data. Analytical systems are not mentioned in these research works as they try to evaluate different aspects of cloud data management that make the data sets used specialized for their specific benchmark scenarios.

As there is no data set available for testing MapReduce jobs<sup>1</sup>, we have decided to start with a popular benchmark from the database community. There are several benchmarks used, some specific for transaction processing and others for analytical processing. We have not chosen any of the OLTP benchmarks because their main motivation is to evaluate transactions. Moreover, the MapReduce model has been designed for analytical workloads due to its batch-oriented nature and not for single operations. We have considered then TPC-DS [Council 2010a], which is a decision support workload from the *Transaction Processing Performance Council* [Council 2010b]. Raghunath et al. [Nambiar and Poess 2006] explain that using both synthetic and real world data for designing the TPC-DS has many advantages over benchmarks that use only one type of data. This is because synthetic data sets built using studied distributions such as the Normal or Poisson distributions have many positive points, but they are not well suited for dynamically substituting bind variables. The TPC-DS utilizes traditional synthetic distributions, yielding uniformly distributed values with a Gaussian distribution.

Scaling a data set can be done in two different ways: The number of tuples in the data set is expanded, but the underlying value sets (domains) remain static, or number of tuples remains fixed, but the domains used to generate them are expanded. In the case of TPC-DS, an hybrid approach was chosen, so most table columns employ data set scaling instead of domain scaling, especially fact table columns; Some small tables' columns use domain scaling. As a result of this approach, fact tables scale linearly with the scale factor while dimensions scale sub-linearly.

We decided to use the TPC-DS data because we could have some information about the data set as well knowing how scaling it would affect the underlying data distribution. The data obtained from the TPC-DS were plain text files which we load into our local HDFS to perform our tests.

Due to the fact that the TPC-DS has been designed as a relational database benchmark, queries outlined in [Council 2010a] were meant to evaluate a data warehouse and are not suitable for our experiments. Queries were specifically designed for testing different aspects of DBMS and characterize general purpose queries. Our work aims to characterize a specific (join) operator.

We decided to analyze the database schema proposed by [Nambiar and Poess 2006] for the TPC-DS, extract the relationships between tables and create queries based on such relationships. For example, Figure 5 shows all the relationships between the table *Store\_Sales* and the ones related to it such as some fact tables (*Item*, *Promotion*, etc.) and some dimension tables (*Time\_Dim*, *Date\_Dim*). We generated join queries using such entity's relationships, but keeping in mind the restrictions posed by each type of join. We have created 96 PigLatin queries using a Fragment Replicate Join, 23 PigLatin queries using a Merge Join and 102 queries using a Hash Join. More details in [Mogrovejo 2011].

#### 4.2 Model Construction

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. Statistical techniques, such as multiple regression analysis, aim to find relationships between workload features (e.g. number of computers

<sup>1</sup>although there is the SWIM (<https://github.com/SWIMProjectUCB/SWIM/wiki>) ongoing project.

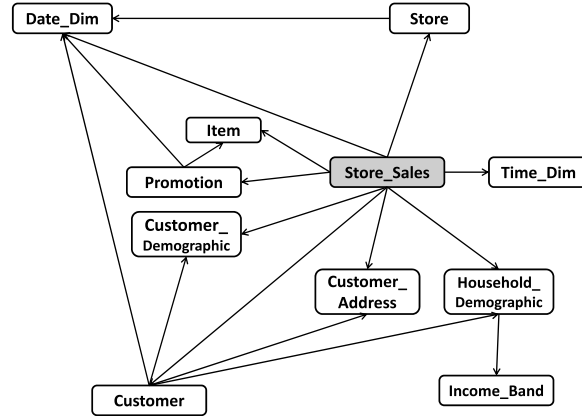


Fig. 5. Relationships from some TPC-DS' tables.

Table I. 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

used, network latency, input size) and performance features (e.g. execution time, I/O metrics). In our case, we have considered those features listed in Table I.

We could have chosen other statistical approaches. Indeed, *clustering techniques*, are common for data analysis in different fields e.g. data mining, pattern recognition, image analysis, and bioinformatics. These techniques consist in assigning a set of observations into subsets, called clusters, based on the similarity of multiple features. This is considered a method of unsupervised learning because it tries to find hidden structures in unlabeled data defining a distance measure between points in the data set. Partition clustering algorithms such as *Kmeans* [MacQueen 1967] are used to identify a set of points that are the nearest ones to a test data point, but they are not suitable for performance modeling because the clustering process would have to be applied on the workload features and on the performance features separately. The similar points with respect of the workload features do not actually reflect the points that cluster together with respect to some other performance features.

Another approach is known as Principal Component Analysis (PCA) and is mostly used for exploratory data analysis, but also for making predictive models. It is the oldest technique for finding relationships in a multivariate data set [Hotelling 1933]. The main idea of PCA is to identify dimensions of maximal variance in a data set and to project raw data onto these dimensions by performing eigenvalue decomposition of the data covariance matrix. The main drawback of using PCA for performance modeling is that the dimensions of maximal variance in workload found do not resemble dimensions that most affect performance. In addition, PCA is not able to correlate workload features with performance ones which makes it not suitable for our needs. Canonical Correlation Analysis (CCA) is a generalization of PCA [Hotelling, 1936]. The main idea is that it evaluates pair-wise data set in order to find dimensions of maximal correlation between the workload features and the performance features. This method and its *kernelized* variant (Kernel Canonical Correlation Analysis) are other statistical machine learning techniques that we find interesting for future work. Kernel Canonical Correlation Analysis (KCCA) [Bach and Jordan 2002] captures the similarity between features using a kernel function, and the correlation analysis is done on pair-wise distances, and not on the

raw data itself. Studying these methods is part of our future work.

Finally, in linear multiple regression analysis, the goal is to predict, knowing the measurements collected on  $N$  subjects, a dependent variable  $Y$  from a set of  $J$  independent variables denoted  $\{X_1, \dots, X_j, \dots, X_J\}$ . In this manner, we can map these independent variables to each workload feature and treat our performance metric as a dependent variable ' $y$ '. The goal of this statistical method is to solve the equation  $a_1X_1 + a_2X_2 + \dots + a_nX_n = y$  for all the coefficients  $a_i$ .

In this research work we have decided to adopt multiple regression analysis because (i) it can make good predictions from multiple predictors (ii) it avoids dependence on a single predictor and (iii) it is based on optimal combination of predictors. Furthermore, our work aims to predict a given performance metric, namely, query execution time. Multiple regression analysis becomes an interesting tool for characterizing join operators because our model also has many different parameters to take into consideration but only one feature to be predicted. In what follows, we will explore multiple regression to study PigLatin join's performances.

#### 4.3 Multiple regression analysis

Our workload features will be used as our independent variables ( $\{X_1, \dots, X_j, \dots, X_J\}$ ), and the query execution time is our dependent variable ( $y$ ). We then use the resulting formulas to predict other queries execution times. An additional parameter that we have added to our model is the join selectivity for each job execution. It enables the estimation of jobs' output with a close relationship to the amount of work to be done. Due to space limitations, the reader may check [Mogrovejo 2011] for further details about it.

One important thing here is that we must be aware of certain problems that can be caused due to our parameters. For example, the parameter representing the number of *Reduce processes* is linearly dependent on the jobs executed which can lead the multiple regression model to 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 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.

This parameter does not vary in our jobs' execution because the number of reducers corresponds to the capacity of our cluster. Thus, we decided not to use the number of reducers in our model construction process because it would cause unstable estimates. The number of map processes is determined by the number of HDFS blocks for 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 for our validating queries, we estimate the number of map processes and use it for constructing our model.

One important challenge being is the lack of metadata in the **MapReduce** computing model, also in its open-source implementation, Hadoop. Systems like Hive [Hive 2010] and Cassandra [Lakshman and Malik. 2010] are trying to incorporate basic statistics into their systems. Once accomplished, these systems will be able to develop efficient cost models for handling big data.

Likewise, Pig framework [pig 2010] lacks of metadata like column definitions or relation's basic statistics. The Pig framework also inherits some drawbacks of HDFS design as the simple metadata kept by the **namenode**. It maintains information about the location of each block and to which file it belongs, guaranteeing that two block files never have the same block ID. In spite of the fact that keeping simple statistics make the system faster, their absence make the system more rigid to perform different operations than the established ones. Therefore, we need to obtain these basic statistics manually in order to estimate the parameters we need for our model (see [Mogrovejo 2011]).

#### 4.4 Experiments and Practical Results

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. The second experiment tries to simulate a real workload where many similar queries are executed in short periods of time. We have decided to create such workload within Pig Framework based on previous observations made by similar research on parallel computer systems. However, due to space limitations, we will discuss only the cross validation experiments in this article. The reader may refer to [Mogrovejo 2011] for the complete set of experiments of this research work.

Cross validation is used for evaluating how the results of a statistical analysis will generalize to an independent data set. In *K-fold cross-validation*, the original sample is randomly partitioned into  $K$  subsamples. Out of the  $K$  subsamples, a single one is retained as the validation data for testing the model and the remaining  $K - 1$  subsamples are used as training data. The process is then repeated  $K$  times (*folds*), with each of the  $K$  subsamples used exactly once as the validation data. The  $K$  results from the folds can be 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. Our model was validated using a 3-fold validation due to the relatively small size of our data set, and to the long running times of our tests.

We describe and discuss next the results obtained from executing cross-validation on our data set of queries. The figures presented 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 by the PigLatin 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 two figures (Figure 6 and Figure 7). Figure 6 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. The error within our join selectivity parameter prevents the model to take full advantage of it. Two jobs took longer than the rest of them because more data were moved through the network.

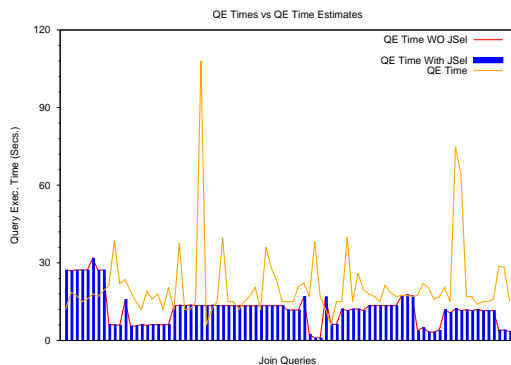


Fig. 6. Query execution times vs time estimates for fragment replicated joins.

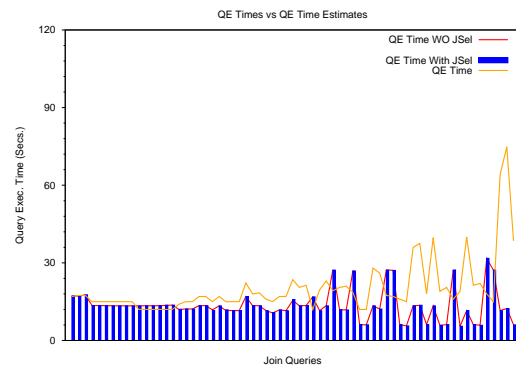


Fig. 7. Fragment replicated joins: execution times vs time estimates for the 80th percentile of the executions.

Figure 7 shows the query execution times from the 80th percentile of the jobs and their execution time estimates. By restricting our results we can omit jobs that suffered from performance problems, execution environment changes and poor linear models. For example, the jobs that in our case suffered from network latency, a key piece for the **MapReduce** framework.

Finally, Figure 8 shows a comparison between the relative error from the two different estimates, either or not using the join selectivity parameter. The error goes from less than 2% up to 205% for all jobs, but for the 80th percentile the error ranges from less than 2% until almost 85%.

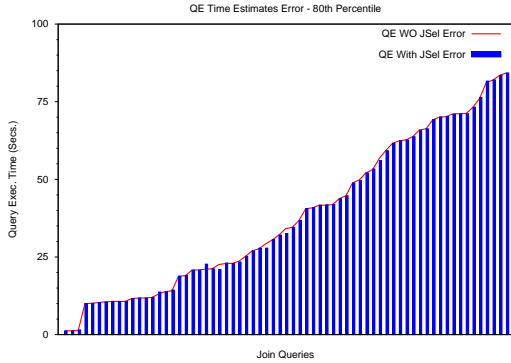


Fig. 8. Fragment replicated joins: error percentage for the 80th percentile of the executions.

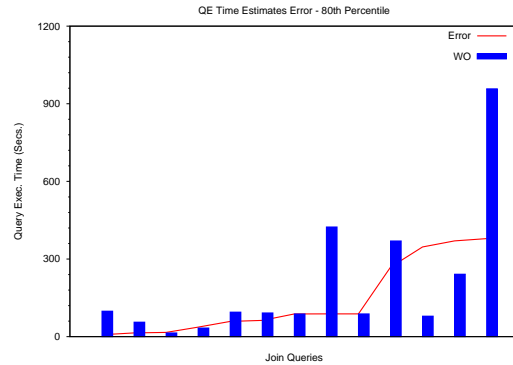


Fig. 9. Merge join: error percentage for the 80th percentile of the executions.

**Merge joins:** We executed the 23 queries created for this specific join operator and used 8 queries for validation. One of the biggest problems for building such queries is the difficulty of creating join operations between ordered data when most data is unordered. This type of join needs both relations to be in ascending order in the join key. Many relations of our data set have the primary key - foreign key relationship, they do share the same key but they are not sorted in the same way on both relations.

Figure 10 shows the differences between the actual 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). The estimates obtained by using the join selectivity parameter did 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 a self-join of the third biggest relation (226MB). Nevertheless, the query was executed in a very fast manner, only 21 seconds to be completed. We hypothesize that this was due to the fact that the column used for joining is sufficiently skewed for the implementation of the merge join operator. The other high point in the figure is also a join involving the third biggest relation. This join execution time was expected due to the size of the input and to the environment variables (e.g. network latency). That is why the predictions do not differ too much from what the time actually took. It differs 7% with the prediction that did not use the join selectivity parameter and 13% with the one that used it.

Figure 11 shows the actual 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 the join selectivity parameter has already the error of the cardinality estimate in addition to the estimate itself.

Figure 9 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. Rather, it samples one of the relations and creates a sparse index on it. Then, it uses this index to access the block directly at probing time. This makes this operator the least memory intensive one because it does not have to load a whole block of data into memory. An interesting thing here is that the model using the join selectivity parameter seems to have a more stable behavior compared to the one that uses the join selectivity parameter. This

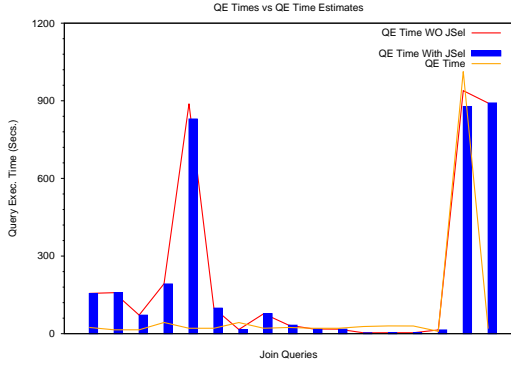


Fig. 10. Query execution times vs time estimates for merge joins.

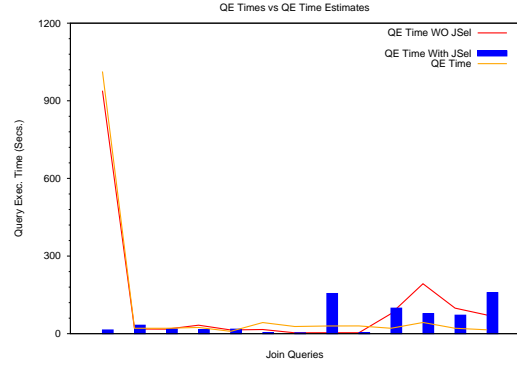


Fig. 11. Merge joins: execution times vs time estimates for the 80th percentile of the executions.

is also due to the error accumulated into the join selectivity parameter, which ends up not being so useful for our discussed 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 figures comparing the actual execution time against the estimated execution time using the models built. Figure 12 shows the difference between the real execution time (*QE Time*) and the prediction values either using the join selectivity parameter (*QE With JSelectivity*) or not using it (*QE WO JSelectivity*) for building the model. One important observation about it is that the actual execution times were really fast compared to the estimated times. We think 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. Even though we decide to keep only the 80th percentile of all the queries, Figure 13 does not change substantially. The real execution times are lower when compared to the estimated execution times.

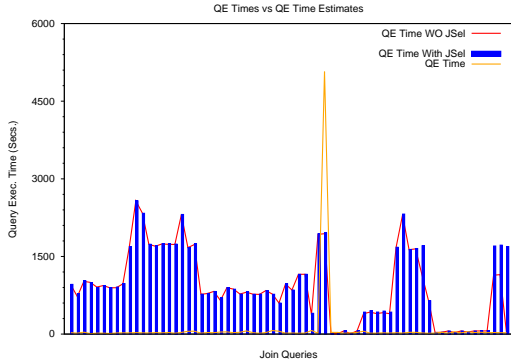


Fig. 12. Query execution times vs time estimates for hash joins.

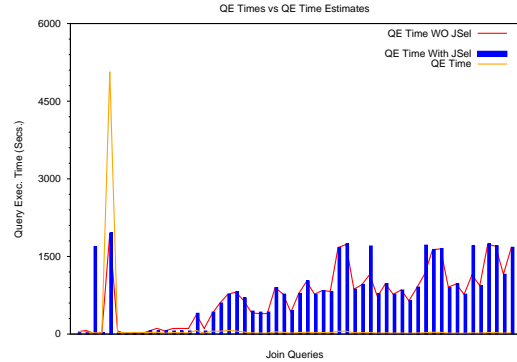


Fig. 13. Hash join: query execution times vs time estimates for the 80th percentile of the executions.

Therefore, we were expecting high errors in our estimates. Figure 14 compares the estimate's error when using or not join selectivity as a parameter. The difference between both estimates is that the join selectivity parameter causes the query execution time to vary along the plot. The error considering this parameter is probably too high and our model cannot take advantage.

This huge difference between our estimates and the actual values motivated us to search for an explanation. That is the reason why we decided to look into the actual jobs executions. In Figure 15 we show the variance between the different query executions of the cross-fold validation. We recorded

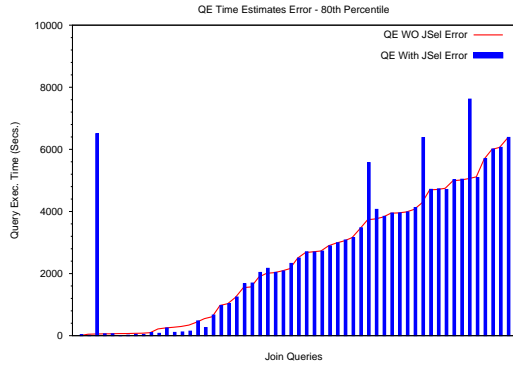


Fig. 14. Hash join: error percentage for the 80th percentile of the executions.

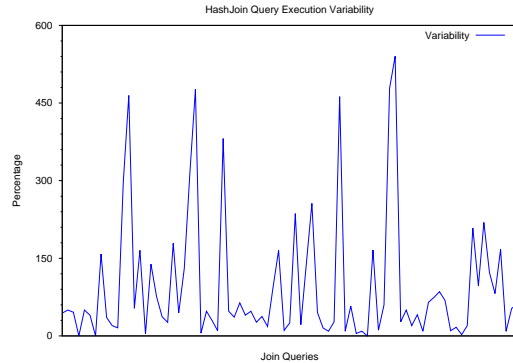


Fig. 15. Hash join query execution variability.

each execution time from each run and then compared them against the other query executions. As we can see, the job execution time varies in hundreds of orders of magnitude from each different execution. This means 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. This join operator is the less stable compared to other operators because it heavily depends on main memory which can vary along query execution.

## 5. CONCLUSIONS

PigLatin needs to receive user hints in order to enable its compiler to choose correctly between the different types of join it has implemented. Thereby, creating a statistical model to characterize the different types of joins in order to estimate its execution time beforehand would mean that the most efficient join operator could be chosen automatically.

The main contributions of this article are (i) a statistical model for query execution time prediction based on multiple regression applied to join queries in the Pig Framework and (ii) the construction of a data set for PigLatin join queries with a higher number of queries for each type of join, consequently, a more realistic evaluation.

We perceived through our experiments that we could expand the number of queries to be executed as well as their variety. In this work we decided to use the standard queries and data model originally defined by TPC-DS as it is an accepted database community benchmark. This has led us to obtain a relatively small volume of join operations' data that resulted in an overfitted model. Hence, we are working on building a bigger data set and, at the same time, studying some different approaches to avoid over fitting of our regression model (e.g., shrinkage methods). We also think about studying the TPC-WS - not OLAP - as a data set. In addition, we believe that exploring other statistical methods are very appealing to the approach presented here. Kernel methods such as those used in [Ganapathi 2009] show great advantages compared to regular statistical methods. Locally weighted learning [Kavulya et al. 2010] and instance based learning [Smith. 2007] seem also suitable.

## REFERENCES

- AGARWAL, S., BORTHAKUR, D., AND STOICA, I. Snapshots in hadoop distributed file system. Technical report, EECS Department, University of California, Berkeley, November, 2010.
- BACH, F. R. AND JORDAN, M. I. Kernel independent component analysis. *Journal of Machine Learning Research* 3 (1): 1–48, 2002.

- BRIGHT, M. W., HURSON, A. R., AND PAKZAD, S. Automated resolution of semantic heterogeneity in multidatabases. *ACM Trans. Database Syst.* 19 (1): 212–253, 1994.
- BRYANT, R. E. Data-intensive Supercomputing: The Case for DISC. Technical report, Carnegie Mellon, School of Computer Science, 2007.
- COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*. SoCC '10. ACM, New York, NY, USA, pp. 143–154, 2010.
- COUNCIL, T. P. P. TPC-DS: A new decision support workload. <http://www.tpc.org/tpcds/>, 2010a.
- COUNCIL, T. P. P. Transaction processing performance council. <http://www.tpc.org/>, 2010b.
- DEAN, J. AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. Vol. 6. Berkeley, CA, USA. USENIX Association, pp. 10–10, 2004.
- GANAPATHI, A. S. *Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning*. Ph.D. thesis, EECS Department, University of California, Berkeley, USA., 2009.
- GATES, A. Programming Pig. <http://ofps.oreilly.com/titles/9781449302641/index.html>, 2011.
- GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA, ACM, pp. 29–43, 2003.
- HADOOP, A. Apache hadoop - The Apache Software Foundation. <http://hadoop.apache.org/>, 2010.
- HIVE. Hive - The Apache Software Foundation. <http://hive.apache.org/>, 2010.
- HOTELLING, H. Analysis of a complex of statistical variables into principal components. *J. Educ. Psych.* vol. 24, 1933.
- HOTELLING., H. Relations Between Two Sets of Variates. *Biometrika* vol. 28, pp. 321–377, 1936.
- KAVULYA, S., TAN, J., GANDHI, R., AND NARASIMHAN., P. An analysis of traces from a production mapreduce cluster. *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010.
- LAKSHMAN, A. AND MALIK., P. Apache cassandra. <http://cassandra.apache.org/>, 2010.
- LIPTON, R. J. AND NAUGHTON., J. F. Query size estimation by adaptive sampling. In *Selected papers of the 9th annual ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. Vol. 6. Orlando, FL, USS. Academic Press, Inc, pp. 18–25, 1995.
- MACQUEEN, J. B. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press, pp. 281–297, 1967.
- MOGROVEJO, R. J. *Exploratory Statistical Analysis of MapReduce Joins*. Ph.D. thesis, Dept. Informatics, PUC-Rio, Brazil, 2011.
- NAMBIAR, R. O. AND POESS, M. The making of tpc-ds. In *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, pp. 1049–1058, 2006.
- NUTCH. Nutch - The Apache Software Foundation. <http://nutch.apache.org/>, 2011.
- OLSTON, C., REED, B., SILBERSTEIN, A., AND SRIVASTAVA., U. Automatic optimization of parallel dataflow programs. *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2006.
- OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD 08*, 2008.
- PIG, A. Apache pig - The Apache Software Foundation. <http://pig.apache.org/>, 2010.
- SHI, Y., MENG, X., ZHAO, J., HU, X., LIU, B., AND WANG, H. Benchmarking cloud-based data management systems. In *Proceedings of the second international workshop on Cloud data management*. CloudDB '10. ACM, New York, NY, USA, pp. 47–54, 2010.
- SMITH., W. Prediction services for distributed computing. *Parallel and Distributed Processing Symposium. IPDPS 2007. IEEE International*, 2007.
- THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: a warehousing solution over a map-reduce framework. *Proceedings of Proc. VLDB Endow.*, 2009.
- UNIVERSITY OF TEXAS. Texas Advanced Computing Center Lonestar System. <http://www.tacc.utexas.edu/resources/hpc/#lonestar>, 2010.
- VAZHKUDAI, S., SCHOPF, J. M., , AND FOSTER, I. T. Predicting the performance of wide area data transfers. *Proceedings of the 16th International Parallel and Distributed Processing Symposium, IPDPS*, 2002.
- ZHU, Q. An integrated method for estimating selectivities in a multidatabase system. *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: distributed computing - Volume 2*, 1993.
- ZHU, Q. AND LARSON., P.-A. A query sampling method of estimating local cost parameters in a multidatabase system. *Proceedings of the Tenth International Conference on Data Engineering*, 1994.