Universidade de Lisboa

Faculdade de Ciências

Departamento de Informática

Detection of Outliers and Outliers Clustering on Large Datasets with Distributed Computing

Rui Manuel Aleixo Pais

MESTRADO EM INFORMÁTICA

2011

UNIVERSIDADE DE LISBOA

Faculdade de Ciências

Departamento de Informática

Detection of Outliers and Outliers Clustering on Large Datasets with Distributed Computing

RUI MANUEL ALEIXO PAIS

DISSERTATION

Project oriented by Prof. Doctor Paulo Urbano (Universidade de Lisboa)
and Prof. Doctor Chunming Rong (Universitetet i Stavanger)

MESTRADO EM INFORMÁTICA

2011

Acknowledgements

Special thanks are given to my supervisors in Lisbon and Stavanger, Prof. Paulo Urbano and Prof. Chunming Rong, who accepted supervise this master thesis project in two different universities, on two countries so far away, for the orientation and suggestions, patience on my writing process and precious corrections.

I'm very grateful to the PhD candidate Rui Máximo Esteves, for his help on my first steps on Hadoop and in the use of the cluster of the Department of Electrical and Computer Engineering at University of Stavanger.

Many thanks are also due to Ana Paula Matos at the International Relations Office of University of Lisbon and Bente Dale at the International Relations Office of the University of Stavanger, who helped to make the bureaucratic process of a co-orientated thesis in both universities as smooth as possible.

And finally, but also primarily, to all my friends from the international community of students, researchers and workers in Stavanger, for the discussions and support – they showed me the knowledge of a broader and vaster world within the bounds of the same humanity.

Abstract

Outlier detection is a data analysis related problem, of great importance in diverse science fields and with many applications. Without a definitive formal definition and holding several other designations – deviations, anomalies, exceptions, noise, atypical data, – outliers are, succinctly, the samples in a dataset that, for some reason, are different from the rest of the set. It can be of interest to either remove them, as a filtering process to smoothing data, or collect them as new dataset holding additional information potentially relevant. Its importance can be seen from the broad range of applications, like fraud or intrusion detection, specialized pattern recognition, data filtering, scientific data mining, medical diagnosis, etc.

Although an old problem, with roots in Statistics, the outlier detection problem has become more pertinent then ever and yet further difficult to deal with. Better and more ubiquitous ways of data acquisition and storage capacities increasing constantly, made the size of datasets grow considerably in recent years, along with its number and its availability. Larger volumes of data becomes harder to explore and filter, while simultaneously data treatment and analysis emerges as more demanded and fundamental in today's life.

Distributed computing is a computer science paradigm to distribute hard, complex problems across several independent machines, connected on a network. A problem is break down in more simple sub-problems, that are solved simultaneous by the autonomous machines, and all resultant sub-solutions collected and put together into a final solution. Distributed computing provides a solution for the limitations in the hardware scaling, both economical and physical, by building up computational capacity, as needed, with the addition of new machines, not necessarily new or advanced models, but any commodity hardware.

This work presents several distributed computing algorithms to outlier detection, starting from a distributed version of an existent algorithm, CURIO[9], and introducing a series of optimizations and variants that leads to a new method, Curio3XD, that allows to resolve both the common issues typical of this problem, the constraints imposed by the size and the dimensionality of the datasets. The final version, and its variant, is applicable for any volume of data, by scaling the hardware in the distributed computing, and to high dimensionality datasets, by moving the original exponential dependency on the dimension to a dependency, quadratic, on the local density of data, easily tunable with an algorithm parameter, the precision.

Intermediate versions are presented for the sake of clarification of the process that took to the final method, and as an alternative approach, possibly useful with very sparse datasets.

For a distributed computing environment with full support for the distributed system and the underlying hardware infrastructure, it was chosen Apache Hadoop[23] as a platform for developing, implementation and testing, due to its power and flexibility, and yet relatively easy usability. This constitutes an open-source solution, well studied and documented, employed by several major companies, with an excellent applicability to both clouds and local clusters.

The different algorithms and variants were developed within the MapReduce programing model, and implemented in the Hadoop framework, which supports that model. MapReduce was conceived to permit the deployment of distributed computing applications in a simple, developer-oriented way, with main focus on the programmatic solutions of the problems, and leaving the underneath distributed network control and maintenance absolutely transparent. The developed implementations are included in appendix.

Results of tests, with an adapted real world dataset, showed very good performances of the referred algorithms' final versions, with excellent scalability on both size and dimensionality of data, as previewed theoretically. Performance tests with the precision parameter and comparative tests between all variants developed are also presented and discussed.

Keywords: Outlier Detection; MapReduce; Hadoop; Distributed Computing

Resumo

Detecção de outliers é um problema relativo à análise de dados, de grande importância em diversos campos científicos. Sem um definição formal definitiva e possuindo diversas outras designações – desvios, anomalias, exceções, ruído, dados atípicos, – outliers são, sucintamente, as amostras num conjunto de dados que, por alguma razão, são diferentes do resto do dados. Pode ser de interesse quer a sua remoção, como um processo de filtragem para uma suavização dos dados, quer para a recolecção num novo conjunto de dados constituindo informação adicional potencialmente relevante. A sua importância pode ser notada no diversificado espectro de aplicações, como sejam a detecção de fraudes ou intrusos, reconhecimento especializado de padrões, filtragem de dados, prospecção de dados científicos, diagnósticos médicos, etc.

Apesar de se tratar de um problema antigo, com origem na Estatística, a detecção de outliers tem-se tornado mais pertinente que nunca e contudo mais difícil de lidar. Melhor e mais ubíquas formas de aquisição de dados e capacidades de armazenamento em constante crescimento, fizeram as bases de dados crescer consideravelmente nos últimos anos, em conjunto com o aumento do seu número e disponibilidade. Um maior volume de dados torna-se mais difícil de explorar e filtrar, e simultaneamente o tratamento e análise de dados emerge como um processo mais necessário e fundamental nos dias de hoje.

A computação distribuída é um paradigma das ciências da computação para distribuir problemas complexos e difíceis por diferentes máquinas independentes, ligadas em rede. Os problemas são divididos em problemas menores, mais simples, que são resolvidos simultaneamente pelas várias máquinas autónomas, e todas as sub-soluções resultantes coligidas e combinadas para obter uma solução final. A computação distribuída fornece uma solução para as limitações, físicas e económicas, no escalamento de equipamento, pela incremento de capacidade computacional, conforme a necessidade, com a adição de novas máquinas , não necessariamente modelos novos ou avançados, mas quaisquer equipamento à disposição.

Este trabalho apresenta diversos algoritmos em computação distribuída para detecção de outliers, tendo como ponto de partida uma versão distribuída de um algoritmo existente, CURIO[9], e introduzindo uma série de optimizações e variantes que levam a um novo método, Curio3XD, que permite resolver ambos os problemas típicos comuns a este tipo de problemas, relacionados com o tamanho e com a dimensionalidade dos conjuntos de dados. Essa versão final, ou a sua variante, é aplicável a qualquer volume de dados, por escalamento de equipamento na computação distribuída, e a conjuntos de qualquer dimensão, pela remoção da dependência exponencial original na dimensão, substituindo-a por uma dependência, quadrática, na densidade local dos dados, facilmente controlável por um parâmetro do algoritmo, a precisão.

As versões intermédias são apresentadas pela clarificação do processo que levou ao método final, e como uma abordagem alternativa, potencialmente útil com conjuntos de dados muito esparsos.

Para um ambiente de computação distribuída com suporte completo a um sistema distribuído e uma infraestrutura de hardware adjacente, foi escolhido o Apache Hadoop[23] como plataforma para desenvolvimento, implementação e teste, devido às suas potencialidades e flexibilidade, e sendo contudo de relativo uso fácil. Este constitui um solução open-source, bem estudada e documentada, empregue por diversas grandes empresas, com uma excelente aplicabilidade quer em cloud como em clusters locais.

Os diferentes algoritmos e variantes foram desenvolvidos no modelo programático MapReduce, e implementados no quadro do Apache Hadoop, que suporta esse modelo e oferece a capacidade de um fácil desenvolvimento em cloud e grandes clusters.

Resultados dos testes, com um conjunto de dados real adaptado, mostrou um muito bom desempenho das versões finais dos referidos algoritmos, com uma excelente escalabilidade em ambas as variáveis tamanho e dimensionalidade dos dados, conforme previsto teoricamente. Testes de desempenho com a precisão e testes comparativos entre todas as variantes desenvolvidas são também apresentados e discutidos.

Table of contents

List of tables

List of figures

1. Introduction

1.1 Outliers Detection.

We live in a world surrounded and ruled by information. Continuously, huge amounts of data are collected, stored and made available in a multiplicity of forms.

Acquiring, archiving, processing, analyzing and evaluate data are everyday tasks; and with it, science fields like Data Mining, Knowledge Discovery in Databases or Machine Learning, have appeared and received a great increase of attention and development in recent years. There is a definitive and growing need to, efficiently, retrieve information from existing data and, furthermore, extract new implicit information or create predictive models from it.

Processing and analyzing data also requires filtering mechanisms; data size may need to be reduced, by relevance or other criteria, deviations or atypical patterns detected – either for removal or recollection as new data, cleaning of errors that can come from of all kind of sources, etc.

Without a definitive formal definition and with several designations[24], - atypical data, deviations, anomalies, exceptions, noise, etc., - outliers, as it will be more frequently used in this work, can be described as "observation (or subset of observations) which appears to be inconsistent with the remainder of that set of data", following the most referred definition, proposed by Barnett & Lewis, 1994, in their classical work, Outliers in Statistical Data[7]. More specialized definitions can be employed, according to the need and the purpose of their detection. Chandola et al.[10] present a classification of outliers types according to some possible definitions.

Although outliers occurrence is expected to be low by definition, its importance arises precisely from the fact that being rare and out of the intrinsic pattern of the rest of the data, they potentially contains non-trivial and valuable information. It may be of interest to remove them from the original set, as a cleaning or reducer filter, for more efficiency or precision of results; or collect them as a new dataset, formed only with those irregularities. It is an important, some times critical, problem in several domains to analyze and get information from the exceptional and atypical cases.

Examples of applications are numerous and applicable on a wide range of fields:

- Errors filtering on databases, due to sensors failures, incorrect measurements or human mistakes on data entering and validating.

- Fraud and criminal activities detection, like irregular uses on network banking, credit cards and e-commerce.

- Intrusion and activity monitoring of all kind of network servers accesses.

- Analyzing and mining very large datasets of scientific data, that tends to growth at high rate with the constant advance of new technologies and equipment, with the most recent examples of sub-atomic particles data from CERN or astronomy data from Hubble space telescope and the large arrays of radio telescopes.

- Pattern recognition, to help data analysis and risk-analysis; as an example with geological data to oil, gas or minerals prospering, resources discover optimization, detection of hidden military facilities, etc.

- Medical diagnoses, where outliers can help to classify or even predict unhealthy states on patient's data.

- Satellite image processing, meteorology and geophysics data analysis, where detection of atypical phenomena can be researched for modeling and forecast of unpredictable natural catastrophes like earthquakes, volcano eruptions, tornados, etc.

Hodge & Austin[24], Papadimitriou et al.[35], or Otey et al.[34] provide extensive and detailed lists of examples. Hodge & Austin[24] or Chandola et al.[10] present good surveys on outlier detection problem and classification of its methods and applications

1.1.1 Common Methods and Issues.

From the early days of pure statistical approaches, several brunches of methodologies and a myriad of algorithms has been developed. Those can be classified in two main categories[9], the predictive and the direct.

The predictive methods apply techniques from Machine Learning to create a model for outlier data. They require training sets of labeled data – implying that such samples are available, – and use supervised neural networks, decision trees or support vector machines, among other techniques, that try to classify objects from the dataset with the learned model[9][34]. Although usually simpler than direct methods, predictive methods impose the existence of classified instances and rely, during the training phase, on many readings of the training set for a good, accurate model, with the risk of becoming too slow for practical uses.

The direct methods includes the statistical methods, also called distribution-based due to the data fitting to some standard distribution model. These models classify as outliers the samples outside the distribution[7][35], requiring some knowledge of the data in order to fit the samples to a distribution model, and are generically limited[6][35] to datasets with few dimensions (defined as the number of attributes in a dataset). Direct methods also comprise the more recent methods like clustering, depth-based, distance-based or density-based[9][35][6].

Clustering methods partition the data into groups, usually trying to maximize intra-group similarity and intergroup-dissimilarity. Outlier detection is done on a posterior process, after the clustering being concluded, or occasionally by specialization of the clustering algorithm, in order to include an outlier identification procedure[9].

Depth-based methods consider instances as points in a k-dimensional space, and classify them according to a defined depth measure. Outliers will be the objects in the outer layers, with smaller depths[35][6]. Methods on this category are known to be inefficient with large values of dimension k, due to its requirement of k-d convex hulls calculations, with a lower bound complexity of $\Omega\big(N^{k/2}\big)$ for N instances[6], that made them efficient for at most $k \leqslant 3$.

Distance-based or proximity approach, compute distances between objects and tries to identify those more distant from the others. First presented by Knorr & Ng[26][27], generalizes most of the concepts of distribution-based and depth-based algorithms, but with a better computational complexity, acceptable even for higher values of dimensionality. A data object is classified as outlier if at least a fraction of the other objects are at a given distance from it. This method prove to be of generally good results in terms of performance, but with classifying problems when dataset has both dense and sparse regions[35]. Several extensions of the original DB-outliers algorithm[27][37] have been proposed, trying to reducing the quadratic complexity due to the pairwise comparison[9], namely by pruning the original dataset[3] or introducing partitioning of the data space[43].

Density-based approach, originally proposed by Breunig et al, with the seminal algorithm LOF[6], apply techniques from density-based clustering algorithms. Objects are assigned with a *local outlier factor*, LOF, which depends on the local density of their neighborhood, and the ones with higher LOF are classified as outliers. The distance to a given number, *MinPts*, of nearest neighbors, defines the neighborhood. This method enables the discovery of local outliers, objects atypical relatively to their local neighborhood. Several variants and extensions have been proposed that try to simplify the issue of select a good number of *MinPts* nearest neighbors points, a non-trivial problem, or introduce several optimizations on the process of outlier evaluation (see also chapter 2, Related Work).

Regardless the several methods developed, not a single approach collect consensus or solves all problems. Each one has its own limitations, and some basic issues are common to all of them, since they are related with the characteristics of datasets itself.

The majority of problems with outlier detection relates directly with the two main properties of all datasets: size and dimension, the number of observations and the number of attributes, respectively. Scalability with size is becoming more and more critical, due to the growth of today's standards of collecting and storage data. Dimensionality, on the other side, has been a curse for most of the techniques employed and algorithms proposed. Most authors refer to at most 10 dimensions as a limit to functionality of their algorithms, in some cases even less, as in the statistical approaches. The scalability of most algorithms is usually exponential with the dimension and quadratic with size[34]. But real data, nowadays, goes up to giga-, tera-, or even peta-bytes in size[36], in some cases with continuous real-time updates, and, so common on Machine Learning applications field, the number of dimensions can be far beyond the limit mentioned, when isn't always possible or acceptable to search and eliminate redundant dimensions. Those are problems that still need to be addressed.

1.2 Distributed Computing.

Distributed computing is a computer science paradigm where data and computation are split across several autonomous computers, interacting with each other in a network, having each one its own non-shared memory and operating system[8]. A computational task is performed upon this system of multiple machines to achieve a common goal. It aims to improve hard computational processes, by breaking the problems into smaller sub-problems and distribute them by the several computers or nodes of the network. Each node becomes responsible for the computations of its sub-problem, with these partial solutions to be collected and combined into a final result. Simultaneously, distributed systems are also able to deal with cases where the amount of data would make it hard or impossible to treat within a single machine, due to memory and storage limitations.

There are several kinds of hardware infrastructures. Clusters, distributed systems on distinct networks, grids, supercomputers, clouds, with different specifications and requirements. The definitions for distributed computing on those different configurations are not definitive or distinct, overlapping on several points[22][8]. But they all share however the characteristics and purposes of the above description for distributed computing: process more data and compute harder problems, in the most reliable and efficient possible way.

Cloud computing, of those above, is perhaps the one that recently is raising more attention. The easiness of maintenance and replication of systems, the cheapest alternative for most situations, since it requires only acquisition of services to basic hardware requisites, usually a simple console or browser on an already existent computer, make it extremely attractive, both for research or business. Services like Amazon Elastic Compute Cloud (EC2)[16] propose simplicity, high efficiency and scalability, with interaction through simple web interfaces, allowing the clients to hire the specific desired computing capacity for their problems, on different hardware configurations, with the easiness of dynamic adjustments as needed.

From the developer's perspective, some aspects are still of some concern. Low-level details on computation parallelization, synchronizing and continuity of processes, interaction between nodes, mainly between master (the controller of tasks) and slaver nodes, distribution and replication of data, faults control, etc., are too complex to be deal in same level as the computing problems.

Dean & Ghemawat introduced, for Google, the MapReduce programming model[15]. Implementations of this model, like the one used on daily basis on Google or the Open Source framework from Apache, Hadoop, can process massive quantities of data in a highly parallelized way, for any number of machines. The underlying framework takes care of all the details concerning distribution of the input data, scheduling of processes, network computers communication and faults. The developer needs only to concentrate on transcribe their problems into the MapReduce model; essentially specifying a map function, that processes a pair (*key,value*) and a reduce function that processes lists of the intermediate values associated to each key. Programs written within this model are automatically parallelize by the underlying framework and can be executed on a large number of machines[15].

1.3 Motivation and Objectives.

While the number of new methods and new algorithms to outlier detection continues to grow, recent new technologies can also play an important role on that field.

Support for parallel and distributed computing, specially interesting in the cases of intensive computing processes and massive data management situations, are now available and freely distributed on Internet. It is in this context that frameworks like Apache Hadoop for distributed computing can constitute an interesting option on a search for an effective approach, that can take care of both situations of problematic storage of huge datasets and the heavily computational requirements associated to the computing processes of data analysis and outlier search.

This work proposes to develop a distributed version of one of the existent methods on outlier detection, and implement that version on the MapReduce framework of Apache Hadoop[23] platform, exploring the potentialities of the distributing computing and the MapReduce model characteristics.

The basis for this research was CURIO, one of the most recent density-based algorithms by Ceglar et al.[9], chosen due to its simplicity and clearness. That algorithm executes only two readings of the dataset and avoids explicit distances calculations by partitioning the data space and use the cell's granularity and the proximity of neighbors partitions as an implicit distance metric[9].

This work proposes to adapt that algorithm to the MapReduce model, explore possible variants and analyze its performance and scalability with the datasets size and number of machines. It is an important issue to address, if the distributed computing may relieve, and under which conditions, the intensive computational cost implicit on the exponential nature of the nearest neighbors search and, as such, provide a solution for the dimensionality problem, or if other approaches would be desirable or even mandatory.

1.4 Contributions of this Work.

The contributions of this work are:

- Development of a distributed computing adaptation of the density-based algorithm CURIO[9] by Ceglar et al., designated CurioD, within the MapReduce data model. Development of some variants, CurioXD, CurioXXD/CurioXXcD and CurioXRRD, looking for improved performance by exploiting the distributed approach, the MapReduce model and the Hadoop framework potentialities.

- A new method is presented, Curio3XD, with a variant, Curio32XD. This distributed algorithm employs several of the optimizations developed on the previous contribution and, exploring the restructuration of data imposed by the MapReduce model, leads to a way of moving the exponential growth on the datasets dimension to a much less severe quadratic dependency on the number of identified partitions assigned in the partitioning step. That number can yet be controlled by a tunable parameter and it not suffers significant changes with the growth of the dataset size, making it suitable for both cases of big and dynamically updated datasets.

- Detailed implementations of all algorithms and variants were made within Hadoop's MapReduce framework, with tests on the variables and parameters of interest for all the algorithms and the distributed technologies, on a real-world dataset, the KDD Cup 1999 Dataset. Tests were executed and results compared, among different volumes and dimensions of data in clusters of different sizes, as also diferent values of precision and tolerance.

The algorithms implemented extract lists of occurrences that are atypical, isolated from main blocks of the rest of data, the outliers according to the definition and premises given above. This work does not try to inspect or prove the quality of the outlier detection, or investigate the nature of the detected outlier occurrences. The purpose of the algorithms development was to, in an useful and efficient way, detect situations of irregularities and suspected outliers to the model implicit on the data, that, in real data processing, would require further analysis, outside the scope of this dissertation, based on nature and specific requisites of the dataset and work in question. The focus, on this work, was kept on optimization, execution speed and good scalability of the proposed methods.

1.5 Structure of this document.

This document is divided as follows. Chapter 2 presents the related work with the different methods of outlier detection, density-based and CURIO algorithm specifically and distributed computing on outlier detection. Chapter 3 introduces the theory beyond the chosen algorithm and density-based approach on outlier detection problem. It also develops the distributed computing topic and the MapReduce data programing. Chapter 4 develops the proposed method, the design of the distributed version and MapReduce version of CURIO and the development of the consequent variants, including a new final version of this approach. Chapter 5 presents the methodology employed to test the performance and the scalability with the variables of interest, the description of the datasets tested and characteristics of the equipment used, along with the experimental results obtained. Chapter 6 analyzes and discusses those results and the performance of the several implementations. At last, on Chapter 7, the conclusions are presented and references for future work are proposed. Details of the implementation on the distributed environment, cloud and cluster settings, Hadoop installation and configuration, are left to the Appendixes A, B and C. All code developed is listed on Appendix D.

2. Related Work

2.1 Previous Work on Outlier Detection.

The older approaches, statistical models like the *discordancy tests* from Barnett & Lewis[7], tried to construct probability distribution models from data, where the outliers were the instances with low probability. Several updates for these methods had emerged, some already after the year 2000[9], but yet with the original limitation of only applicable to low-dimensions and with a poor scalability.

Clustering methods date from the 90's decade. They are usually not conceived with the outlier detection as main purpose, but as a specialization or an extension. Some of these methods consider the outlier detection in a way fully separated of the clustering process, allowing the exploration of possibilities and improvements. Some of the main algorithms, DBSCAN[32], CLARANS[19] or BIRCH[43], has been used as base for more complex versions, integrated with other general methods of clustering like k-means and wavelet transforms[9], to separate clusters from atypical, isolated samples.

In distance-based approach, one of the first algorithm proposed, DB-Outliers, by Knorr & Ng[26]. This algorithm discretizes the data space into hypercells with linear complexity on the number of instances and exponential complexity on the dimension. Ramaswamy et al.[37] propose an extension of DB-Outliers using the BIRCH algorithm[43] to produce a ranked list of potential outliers. Shekhar et al. [38] proposed a variant that analyze the neighborhood topologically instead of using distances. Finally, Bay & Schwabacher proposed an algorithm[3] based on the randomization of data with a pruning of the search space. This is the state-of-art in distance-based techniques, still with a theoretical quadratic time, but with report from their authors as running in practice close to linear time.

The LOF algorithm, proposed by Breunig et al.[6] in 2000, is a seminal work in the outlier detection field and in the so-called density-based methods. Objects are assigned with a *Local Outlier Factor*, LOF, which depends on the local density of their neighborhood, with a complexity of $O(N \log N)$ and have also the novelty of detect anomalies of instances in relation to its neighborhood, the local outliers. Several variants and extensions have been proposed; trying to optimize and solve existent issues like the selection a good number of nearest neighbors' points, a non-trivial problem.

Among others, GridLOF[12] by Chiu & Fu (2003), introduce a direct optimization for LOF, by providing a preliminary step with a quantizing of the data domain and avoiding dense cells from further consideration on later steps. Similarly, FastOut[11] by Chaudhary et al. (2002) quantizes the data domain into regions, but of equal density, and objects are classify as outliers if the density ratio exceeds a specified threshold. LOCI and aLOCI[35], proposed by Papadimitriou et al. (2002), evaluate instances by defining a local correlation designated by the authors as *Local Correlation Integral*, LOCI, which leads to a highly effective method for both outlier detection and micro-clustering, the detection groups within the outlier instances. The variant aLOCI is an approximated algorithm, which its authors claim to provide fast, highly accurate outlier detection.

More recently, SNIF[41] by Tao et al. (2006) or CURIO[9] by Ceglar et al. (2007) are also density-based methods that pursue further in these previous line of optimizations. SNIF propose a method designated *prioritised flushing*, to allow accomplish the outlier detection on any arbitrarily large dataset, in three readings. Priorities are defined, based on the likelihood that an instance will be, or not, an outlier in relation to just a few neighbors, reducing, reducing vastly the number of objects in the outlier evaluation process.

CURIO, quoting directly its authors, "optimizes the analysis of disk resident datasets by avoiding object-level comparison and enabling analysis in only two sequential scans"[9]. In this work, as a distributed computing implementation and development of that algorithm, we follow the original CURIO proposal as the starting point for this research. The algorithm and its analysis will be presented in detail in section 3.3, of chapter 3 Theory.

2.2 Related Work on Distributed Algorithms.

There is not much literature available on distributed computing algorithms for outlier detection. Most algorithms distributed algorithms are conceived for dealing with either the problem of distributed data (by several physical locations, not necessarily close geographically) or a very specific domain or type of data, like sensors network or collaborative intrusion detection[34].

A promising distributed method can be found in the work of Sarker & Kitagawa[40], from 2005. The paper presents a distributed algorithm based on the computation of distances between an element on the dataset to its k^{th} nearest neighbors. The authors also refer their work as the first proposal, to the best of their knowledge, of a distributed algorithm for this purpose. But their work is purely theoretical and, although Sarker & Kitagawa refer an on going research on the algorithm implementation, no work come yet after this first paper, with details on the development, practical results and comparison with other approaches.

Other related works on distributed environments are the ones by Otey et al.[34] from 2006, Zhou et al.[44], from 2008, and Koufakou & Georgiopoulos[28], from 2009. Those don't develop specific distributed computing methods for outlier detection, but deal with the problem of databases that are, *per se*, distributed across several nodes on a network.

Those algorithms execute their computations over a training set on a node level, in parallel since all nodes contain exclusively non-shared data, to build a local model, and a second task on a master node collects the results and computes a global model of the data that is then applied to the all dataset to be processed. This implies several differences relative to the work developed in this dissertation. All low level work related to the distributed computation must be done by the implementation of the algorithm, the start of the computations on the local nodes, the verification of finished tasks and collection of results.

One interesting and most important feature of the method presented by Otey et al.[34] is the possibility of deal with all kind of attributes, numerical and categorical, although not directly related with the distributed nature of the method. Categorical attributes and numerical attributes are isolated and treated with different techniques. For the first type its used a technique named frequent itemset mining and for the numerical type the calculation of their covariance matrix[34]. An anomaly score function is then computed from that pre-analysis of both distinct attribute spaces, using the concept of *links*, that is the similarity between items by their attributes. Instances with infrequent categorical sets or continuous attributes differing from the covariance are more likely to be outliers.

The computational load being realized simultaneously by the different nodes results in a speedup of the calculations involved on both processes, and a reducing in memory and computing requirements[34]. This method is linear with the size of data, and quadratic in the number of numeric attributes is, less optimally, exponential on the number of categorical attributes. Being an extremely effective method in terms of execution times, scalability, plus its ability of deal with mixed type of attributes and the extension for a distributed approach, make this method one of the most general-purposes methods and the state-of-art in outlier detection.

The other two methods referred[44][28] follow similar procedures, by modeling local models on partial training datasets distributed by several nodes and a global model from the processing of those local models to be applied to the all data, but using different assumptions and techniques (distance-based). They try to improve the performance and the accuracy of Otey's method, which may encounter problems when classifying instances with irregularities on both categorical and numerical attributes or with high-dimensionality, sparse datasets[28]. Koufakou & Giorgiopoulos present[28] direct comparisons between their method and the one by Otey et al.[34], with excellent results in terms of performance, equally good scalability with the size of data and number of nodes, and a great accuracy even in the case of very sparse data in the mixed attributes case.

The more directly related work with this one, is the method for outlier detection on categorical datasets using MapReduce, by Koufakou et al.[29]. That method implements a distributed computing version of a technique, developed previously by those authors[29], named *Attribute Value Frequency*, AVF, in the MapReduce programming paradigm. AVF assigns a score to each instance, using the frequency of each unique attribute, and performing one single reading of the dataset. The method is easily parallelizable and its implementation on the MapReduce model was called MR-AVF.

Dealing explicitly and exclusively with categorical data, constitutes the oppose situation of the algorithms proposed in this work, that allocate the instances in a discretized space, considering a numerical conversion of those attributes to allow the association with the coordinates reference of the CURIO method.

A last reference is Kang et al.[25], from 2008, that uses Hadoop to implement an algorithm that can deal with the hard task of computing diameter of huge graphs, namely petabyte-sized graphs. Such algorithm can be used to detect outliers, as is suggested by their authors, but that's not the purpose of the method, that however still stands as a promising suggestion for future researches. As far as it was possible to ascertain on the available information, the work presented in this dissertation is the first implementation of a full-distributed algorithm for outlier detection on the MapReduce programming model.

As far as it was possible to ascertain on the available information, the work presented in this dissertation is the first implementation of a full-distributed algorithm for outlier detection on the MapReduce programming model.

3. Theory

3.1 Distributed Computing.

Physical constrains impose conditioning and limitations on the scaling of hardware. Memory and processors capacity can't grow indefinitely, and nowadays they beginning to reach a power limit[13]. Estimates point to an expectation of 10 to 20 more years for the present rate of doubling the density of circuits every generation for silicon based circuits[13]. While computational problems becomes harder and the volumes of data to process bigger, computing capacity don't grow at same rate and ultimately any hardware, any processor hits the end of its possible evolution. Economically, trying to keep with the escalating issues is also problematic, since latest state-of-art hardware is always expensive and the old equipment, still functional, accumulates.

A solution for this issue is scaling the number of hardware elements, connecting several machines, not necessarily of the most advanced type, on some network architecture, and using the resultant total resources of those distributed systems, while at same time having the commodity of reusing existent equipment. Or rely on available services from external sources by Internet, with almost unlimited data capacity, and without any direct maintenance needed.

Distributed computing is a general paradigm of computer science which tries to solve large, heavy computational problems by break them into smaller sub-problems and assign them to many computers in a network. The computation runs in parallel on those computers, in order to get smaller execution times than would be obtained if such computations where executed on a single machine, sequentially. The solutions of the small sub-problems are combined in a way to get the solution of the initial main problem.

Through time several computing paradigms for large computer structures appeared, grid computing, cluster computing, supercomputers and more recently, and getting more attention every day passing, cloud computing. With several overlap definitions[22], techniques, but same goal: process more data, compute harder problems over it, reliably and efficiently. Foster et al.[22] and White[42] give comparative reviews on this different approaches and technologies beyond, with detailed comparison specially focused on grid and cloud computing.

Memory and storage issues can be solved by adding more hardware, disks or memory cards, but distribute a heavy computation across several processors don't share the same linearity or easiness. The technical problems of add more space is easy to solve, due to the static nature of data storage, addressed mainly by the operating system and the file system underneath which can mount several disks in a transparent way for the user or use solid-state memory by request. The breakage of a problem in several sub-problems is not trivial, since most real hard problems don't exhibit such linear nature that allows simple subdivisions. Another set of problems, of a very different type, comes from the more technical background to be keep track, like network traffic and load distribution, control of initialization, ending and flow of processes, data or hardware failure, etc.

It's in this context that paradigms like MapReduce and implementation frameworks like Hadoop were developed, allowing the developers to concentrate on solutions for they applications, in a simple methodology for distributed computing, and relying on the framework to take care of all the hard technical details of the distributed system and keep the implementation fault-tolerant.

3.1.1 The MapReduce Programming model.

MapReduce is a programming data model proposed by Dean and Ghemawat[15] from Google, Inc., conceived for distributed computing. It was designed for parallel processing of large amounts of data, efficiently, by distribute the computation across several machines in a network of computers. Implementations of MapReduce should take care of all details related with data splitting, scheduling of processes, failures handling and management of all inter-machines network communications. The users can then focus exclusively on the program conceptualization to solve their specific data problems, without requiring time and effort on those non-directly related issues.

One of the machines in the network is defined as master, the starter and controller of the MapReduce job, or sequence of jobs, and all underlying processes. All the other machines involved in the MapReduce process will be designated as slaves (the workers on Dean's terminology[15]). The master will take care of the data splitting across all nodes, and scheduling and verification of the processes until all the MapReduce jobs, running in the nodes, finish and the output files are saved, normally on the master machine.

The model is inspired on Lisp and some similar functional languages. It relies on two main functions, a Map and a Reduce. MapReduce frameworks should implement those functions in a way abstract enough, that the clients needs only to develop the code to fulfill those functions according to their programming goals.

Each machine will run its own copy of the MapReduce application, with multiple MapReduce processes simultaneously. A MapReduce process is constituted by two tasks that will run, in all the slaves of the cluster, the implementations of the Map and the Reduce functions. The number of map and reduce tasks is defined by the user, usually based on the number of available resources (number of nodes, number of processors). The processes executing map tasks are designated mappers and the ones executing reduce tasks are designated reducers.

The Map is defined by the function written by the developer and will receive a pair (key, value), usually a data related reference as key and the data itself as value, and generate pairs of keys and values, from the received data, according to the developer specification. The output holding those pairs is saved locally, on the slave's disks, receiving the master the information that the task is finished and the locations of the output on the network.

The Reduce, also a user-written function, will act on the intermediary output produced by the Maps. The reducers will access the Map outputs from all slaves, sort them by key, and process the list of values associated to each key. Since different maps can output values with the same key, and values from each key must be merged and processed together in same reducer, each reducer will need to access the outputs on all slaves. The reducers will generate lists of values to be saved on the master, as final output.

In both tasks, the processes of mapping values with keys and reading the outputs from the slaves are done in parallel on the machines of the cluster. The underneath handling of the data, processes scheduling and parallel computations is completely transparent to the users; all that is mandatory to the developer is to specify the code for the two functions according to the model which requires the following arguments and data flow[15][30]:

$$\begin{array}{rcl} \text{Map } (key_1,\ value_1) & \rightarrow & \text{list}[(key_2,\ value_2)] \\ \text{Reduce } (key_2,\ \text{list}[value_2]) & \rightarrow & \text{list}[(\{key_3,\} value_3)] \end{array}$$

It is important to note that key_1 and key_2 may not be related or belonging to the same data domain, depending on the user's developed functions. Usually, key_1 is just a reference to a data file, a filename, a line number or the offset position on the file for some portion of data; key_2 can take also any kind of values, either identical or some transformation of key_1 or $value_1$, – anything that match the logic of the Map implementation for the problem in hands. The same applies to $value_1$ and $value_2$. But key_2 and the values on the list of $value_2$ in Reduce arguments must necessarily be of the same type as the ones in the output of the Map.

In the final output, a key_3 is optional, and the Reduce may emit just a list of values. For each key_2 processed, the Reduce can produce a single value, several values or none, as the function output $value_3$.

In the complete process, MapReduce is an application that transforms a set of pairs (*key, value*) into a set of values. The model is simple, yet powerful and flexible; is it also possible, desirable in some problems, to build more complex structures with some of the tasks repeated and combined, as several Maps and one Reduce or a Map and several Reduces, or any kind of combination to achieve some specific result.

To exemplify a typical MapReduce job, next is presented one of the most common MapReduce applications, the word counting:

Algorithm MapReduce Word Counting

```
mapper (file, content):
    for each word in content
        emit (word, 1)

reducer (word, values):
    sum = 0
    for each value in values
        sum = sum + value
    emit (word, sum)
```

The algorithm is quite simple. The Map function receives a reference to the file, reads the contents and parses each word founded emitting a pair (word, 1). Note that in the example, the reference "file" is just a key, required by the programming model, and never really used (the files IO control is done by the underlying framework). The reduce function receives all associations (key, list[values]) for each word – the reducer key in this example, – and just sums all the values in the associated list, emitting the total counting by word.

As an example, consider the following tongue-twister: "the worst word in the world is the word world". The Word Count algorithm would process this set of words as:

Map (somekey, "`the worst word in the world is the word world`")

↓

[(`the`,1); (`worst`,1); (`word`,1); (`in`,1); (`the`,1); (`world`,1); (`is`,1); (`the`,1); (`word`,1); (`world`,1)]

↓

{ (`the`, [1,1,1]); (`worst`, [1]); (`word`, [1,1]); (`in`, [1]); (`world`, [1,1]); (`is`, [1]) }

⇊⇊

Reduce (key, list[ ])

↓

[(`the` 3); (`worst` 1); (`word` 2); (`in` 1); (`world` 2); (`is` 1)]

This section was focused on the programmatic side of the MapReduce model, where the user put all his/her efforts implementing the required code for the abstract Map and Reduce functions. Next section examines the general MapReduce process in more detail, focusing on the framework design, on the communication and data flow over the network.

3.1.2 MapReduce Data Flow.

With a Map and a Reduce functions implemented by the user, and a simple program, to be executed on the master, that applies any given configuration and starts the jobs, – the framework will take care of the files IO, network nodes communication and controlling of processes, beyond the scene, in a way completely transparent to the user.

Each slave node runs its own copy of the MapReduce application. The instances are able to execute several map tasks and several reduce tasks, usually in less number and defined by the user according to the specifications of the framework.

The different stages of a MapReduce process can be schematized as:

Input Data > split > **Map** > Intermediary Output > sort + merge > **Reduce** > Output Result

The input data is split into smaller files, passed to the slaves, to be processed by the mappers. In parallel each Map function generates lists of pairs that will be saved, as intermediary output, on the local disk of the node where they were generated. The reducers will read the intermediary output, sort it and merge the values into lists associated to each key. The Reduce function will be called once per key, iterating over the list of values. The final output, with the results, will be saved on the master machine.

The complete data flow of a MapReduce process is shown in the figure 1:

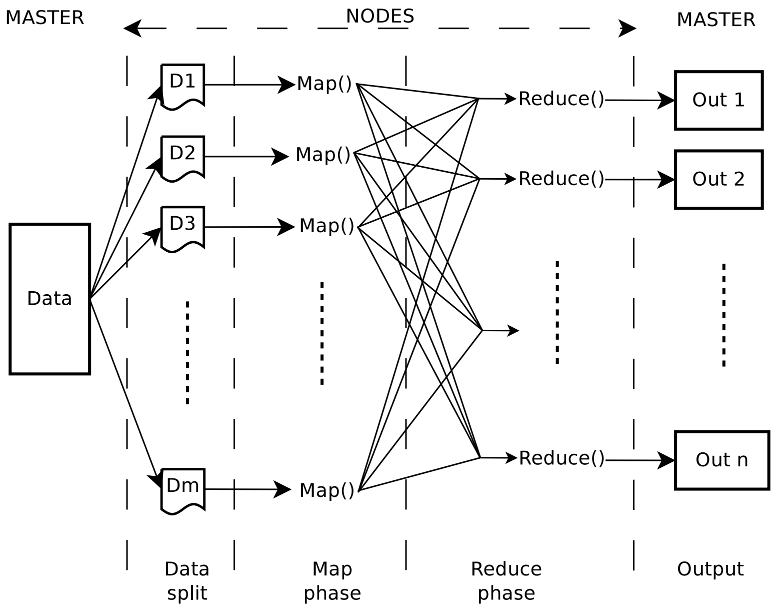


Figure 1. Data flow of a MapReduce process.

The overall view of the process can be detailed as:

- The master splits the data across the slaves and MapReduce instances are initialized on each of them.

- The master selects idle slaves, with no tasks running, and assigns them map or reduce tasks.

- The mappers reads the contents of the split file and after applying the Map function, generate the intermediate pairs (key, value) to a list saved locally.

- The Master is informed about the status of the finished processes and the locations of the saved output. It will forward these locations to the reducers.

- The reducers use remote procedures calls to read the data from the received locations. The data is sorted by the intermediate keys, so that all values associated to a unique key are grouped together and processed by the same reducer.

- The Reduce function takes each key and its associated list of values, and iterates over it.

- The output of the Reduce function is saved to a final output file, specific for that particular reducer, on the master node.

- When all Map and Reduce tasks are finished, the main program on master can terminate or proceeded with any other instructions from the user code.

One important refinement introduced to this model is the use of partitions to hold the output from the mappers. Data will be partitioned on each node by a same number of partitions as defined reducers; partitions will be reduce-specific named subdirectories. A Partitioner function, of the intermediate key, associates each pair to one of the partitions, in a way that all values for a given key are always saved to the same partition.

The partitioning can be a customized function, provided by the user, but a default should be available from the MapReduce implementation, generally a hash function (Dean & Ghemawat[15], for example, use *hash(key)* ***mod*** R, with R = number of Reduces, as default partitioner function), which tends to produce well balanced partitions. The reducers will then retrieve exclusively the data from the partition, on each node, that is associated with, reducing the network traffic and the computational demand in the processing and merging of the Maps output.

The Partitioner not only ensures that smaller output files will be assigned to each reducer, but also that each key and associated values will be saved to same partition on each mapper, fetched and processed by a single, corresponding reducer. Otherwise the result of the Reduce wouldn't be correct.

While this model may appear relatively complex, eventually inefficient, if compared to more sequential processes, it can be applied to significantly large datasets, with data storage and computational effort distributed across any number of commodity machines, and computations being taken in completely parallel mode without any extra intervention from the part of the users. This approach achieves basically linear speed-up with the number of core processors[13][42].

One potential weakness of the model lies on the need of the reducers to do remote procedures calls on the other nodes on the network. That creates an intense flux of communications across the network, aggravated by the fact that the Map step can often produce great volumes of data that, being transferred between machines, can eventually lead to overflows of network bandwidth. To minimize this issue, the MapReduce model includes an optional function, the Combiner, an abstract function, similar to the Reduce function, which must also be implemented by the user according to his/her problem requirements.

The Combiner runs on every node where mapper tasks have been performed. It receives the data emitted by the Map on a given node, and performs a partial merge of this data, reducing its size, allowing for a minimization of data volume saved and reducing the total bandwidth required by the reducers to retrieve it.

The Combiner can be considered as a "mini-reduce" process, which runs locally, on each node, after a Map execution, one or several times, recurrently, depending on the framework implementation and configuration. Typically its code is similar or, in some cases, identical to the one on the Reduce function. Identical Combiner and Reduce functions are possible when the function is both commutative and associative[42], otherwise specialized Combiner function must be written by the user, if that feature is to be used. Note that running a Combiner requires an extra computational effort and a possible extended runtime on the mapper side, so it can be seen as exchanging bandwidth usage by computational resources. Its use may not always result in an optimization or speed up of the processes.

The previous example of word counting is an excellent case where a combiner could be of great help. For big documents the map will produce a large quantity of pairs (word, 1), some with a high occurrence of repetitions. A Combiner would merge such pairs with identical key/word, executing the sums locally on each node, and save them also locally. The Reducer would retrieve a much smaller volume of data, more easily transferred over the network, and had just to proceed on the final sum for each key/word from all Map+Combiner of all nodes.

As an example, consider that some document contains 10000 occurrences of the word "the", evenly distributed through the text, and that there were 10 Maps running. Instead of huge lists of about 1000 pairs ("the", 1) by Map, to be retrieved by the reducers, the Combiner would merge those lists on single pairs ("the", 1000), saved in very small files, and only 10 sums of such pairs would be executed by the Reduce.

A MapReduce implementation will also require, besides Map and Reduce abstractions, a means of connecting the processes executed on the Map and Reduce phases. The usual option is a dedicated distributed file system, although other options are possible. On the original Dean and Ghemawat proposal[?] this is done via GFS, Google File System, a distributed file system develop internally by Google Inc.

Several implementations of MapReduce frameworks exist, such as the proprietary Google's original version[15], the Apache's open-source solution Hadoop[?] or the recent and yet experimental GGL-MapReduce[18], that uses streaming for all communications, eliminating the overheads of performing the communications via a file system. In this work it was used the Apache Hadoop implementation which will be presented in the next section.

3.2 The Apache Hadoop Framework.

Hadoop is an open-source framework, for reliable and scalable distributed computing on large clusters of commodity computers, specially oriented to large datasets, computing-intensive processing, which is being developed and supported by Apache Foundation[23]. It includes, among others, the following main projects[23]:

- HDFS: A distributed file system that provides high throughput access to application data.
- MapReduce: A software framework for distributed processing of large data sets on computer clusters.
- Hadoop Common: The common utilities that support the other Hadoop subprojects.

Hadoop and related projects are written in Java, but not restricted to that language. Is possible to write MapReduce applications to run under Hadoop, via Hadoop Streaming, in Python, Ruby or any language that can read standard input and write to standard output[42] or through the C++ specific interface, named Hadoop Pipes[42].

Hadoop have an abstract definition of filesystem, build upon abstract Java classes, with several implementations[42]. Of those, the more commons are the types Local and HDFS, but others such as HFTP, HSFTP, HAR, KFS or S3 are currently available for specialized purposes (see White[42] or Hadoop documentation[23] for details). The Local filesystem is used mainly for testing and debugging purposes, under a single-client machine. The HDFS filesystem is designed to work efficiently with the MapReduce model.

Under Hadoop's MapReduce framework, data is copied to HDFS filesystem and the MapReduce implementation is then used to combine, in an optimal way, the data from the several nodes and distributing the computation across the several processors of the machines of the cluster. In short, the HDFS provides a reliable shared data storage while the MapReduce provides the analysis capabilities for the data processing.

The HDFS filesystem is a highly fault-tolerance distributed file system similar to Google File System (GFS), and it was design to be deployed on low-cost commodity hardware, providing high throughput access to application data, specially on situations of very large data sets[23]. Its most salient features are:

– can handle very large datasets, up to terabytes and petabytes of data,

– streaming data access, HDFS is designed around the idea that the most efficient data processing pattern is write-one/read-many-times, the typical usage of datasets,

– runs on commodity computers (commonly available hardware from multiple vendors), not requiring expensive or high reliable hardware, since reliability is ensured by redundancy of copied data,

- can be interacted by shell-like commands directly from Hadoop line command,
- highly configurable, with a default configuration suited for most situations.

There are two types of operating modes on a master/slaves configuration. A HDFS cluster has those two types of nodes: a namenode, the master, and a number of datanodes, the workers or slaves.

The namenode manages de filesystem namespace, maintaining the filesystem tree and its files and directories metadata, persistently stored, from itself and from the several datanodes. Datanodes store and retrieves blocks of data, reporting back to namenode, periodically, with updated lists of data blocks stored.

The filesystem is accessed from a client with a POSIX-like filesystem interface, which made the access completely transparent to the users. The user code doesn't need to know or specify anything related with the namenode or datanodes to be functional.

For a deeper overview of the HDFS filesystem, usage and detailed specifications, White's Hadoop The Definitive Guide[42] or Hadoop documentation[23] are the ultimate source of information. The next section details the MapReduce implementation in Hadoop.

3.2.1 The MapReduce Implementation in Hadoop.

A user-written MapReduce application does not run directly over the distributed filesystem, but through the Hadoop MapReduce program. The executable location and any other needed parameters should be passed to Hadoop, via command line, as flags.

Within Hadoop a MapReduce process is called job. In its simplest use a MapReduce job can be executed with a singe line of code, evoking the function: JobClient.runJob(Conf). But usually the user wants more control of both how the MapReduce process evolves and how the cluster and network is used.

Hadoop includes also some other Java applications, coordinated by the main Hadoop program in conjugation with the user-written MapReduce application (a job Jar, if written in Java). When Hadoop runs a job, at least four independent entities are always present[42]:

- The client, - the main Java application that submits the MapReduce jobs.
- The jobtracker, - a Java application that coordinates the job execution.
- The tasktrackers, - are Java applications that runs the tasks, Maps or Reduces, that the job has been slit into.
- The distributed filesystem, usually HDFS, used for sharing files between the other entities.

Hadoop, running in all machines, will keep a jobtracker and several tasktrackers running and in contact across the cluster, interacting with the underlying filesystem for share data and configuration files. It will then start the MapReduce program.

The scheme of a MapReduce job and the interaction of those entities is presented on figure 2:

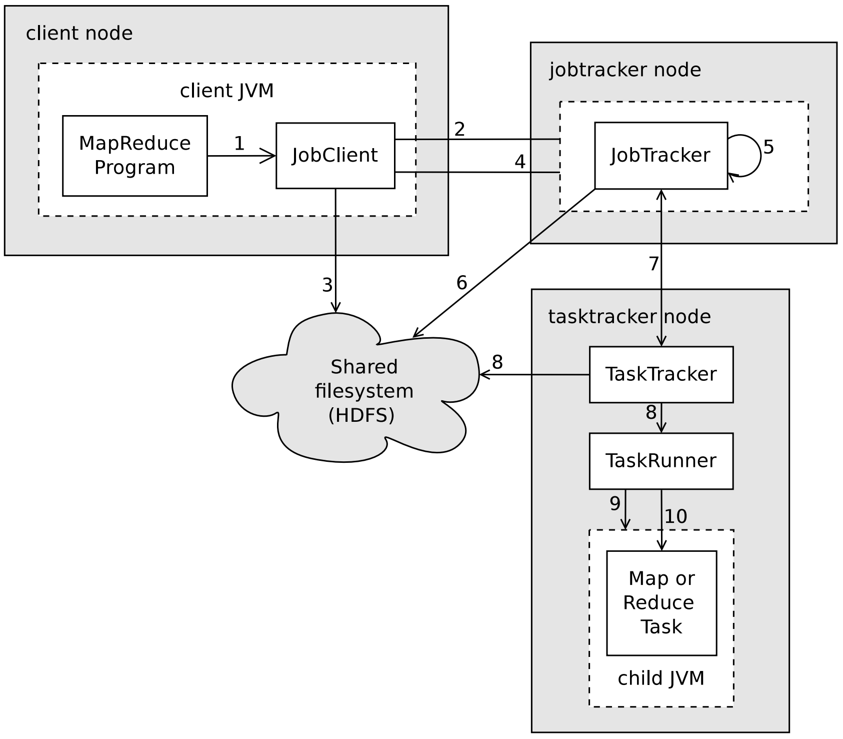


Figure 2. A MapReduce job in Hadoop

1. The MapReduce program calls the runJob() method on its main function. It will create a new instance of a JobClient object, which is responsible for the initialization, validation of requirements and submission of the MapReduce job.

2. 2. The Job Client gets a job ID from the JobTracker. It then verifies if both input and output directories are specified and are valid, by computing the input split files on input directories and ensuring that output directory does not exist. If the JobClient cannot compute the input splits or output directory already exists the job is not submitted and an error is throw to the MapReduce program.

3. If validation results are correct the JobClient copies from the shared filesystem the resources needed, such as configuration files, input splits and job Jar. The job Jar is copied with a high replication factor in order to have several copies across the cluster available for the tasktrackers.

4. The JobClient informs the jobtracker that the job is ready for execution, calling submitJob() method on JobTracker.

5. The JobTracker initializes the job, putting it in a queue line which execution is con-

trolled by the internal scheduler of Hadoop.

6. The JobTracker retrieves the input splits from the shared filesystem, creating a Map task for each existent split. The number of Reduce tasks will be defined by a configurable property tuned by the user in order to optimize the processing capacity and is usually defined proportionally to the number of nodes or total number of processors on the cluster.

7. The tasktrackers run on a loop that periodically sends calls (designated heartbeats calls on the Hadoop jargon[42]) to the JobTracker. The communication between JobTracker and TaskTrackers ensures that TaskTrackers are alive and forward assignments of new tasks from the JobTracker to idle TaskTrackers.

8. A TaskTracker that had been assigned a task, copies the job Jar from the shared filesystem, creates a working directory for the task and starts an instance of TaskRunner to run the task.

9. The TaskRunner launches a new Java Virtual Machine for each task assigned. It is possible however reuse de JVM by changing the default option for the property `mapred.job.jvm.num.tasks` (defaults to 1) via configuration file.

10. The child JVM runs the assigned Map or Reduce task. Note that bugs on the user written Map or Reduce functions that may crash or hang the child JVM will not affect the tasktrackers.

The full MapReduce job process is fairly complex and this scheme summarizes the main steps, properties and entities involved. The deeper inwards of the all process, including the job scheduler, task assignments process and failure handling, briefly mentioned here, are quite technique and fall outside of the scope of this thesis. They are covered in extend detail on Hadoop official documentation[23] and White's Hadoop The Definitive Guide[42].

3.2.2 The Map and the Reduce under Hadoop.

Hadoop Java implementation of MapReduce programming model is done with Java abstract classes (since Hadoop version 0.20; on previous, deprecated versions by Java interfaces), named Mapper and Reducer classes. The user must implement his/her own classes for Mapper and Reducer by extending the abstract ones.

The MapReduce model demands that the Reduces receives its input sorted buy keys. The process of transference of the Maps outputs to the reducers, sorting and merging values lists, denominated by some authors as shuffle[1], is one of the most delicate and important aspects of MapReduce, and one of the potential bottlenecks of any MapReduce implementation. The

1. In some contexts the term shuffle is used to designate only the transference part of the process[42].

details of that process in the Hadoop framework and its refinements will be given next.

The Map phase in Hadoop is optimized by the use of memory buffering, data partitioning and pre-sorting (cf. figure 3).

A circular memory buffer is used to collect the data produced by the Maps and limited limited by a certain threshold size. The buffer size is controlled by a configurable property, `io.sort.mb`, with 100 MB set by default, and the threshold size is controlled by the property `io.sort.spill.percent`, with 0.80 by default. When the threshold is reached, the buffer contents starts, in a background thread, to be written to the disk in job-specific directories. The written, or spill process is done in a round-robin way, with the Maps continuing to write to buffer at same time and from there to disk, being the Maps temporary blocked anytime the buffer reaches its threshold limit size.

In memory, data is divided into partitions, corresponding to the reducers that they will be send to, by the background thread, and in each partition a sort by key is made and a Combiner applied, if one as been specified. Only after the partitioning, sort and any Combiner defined has been executed, the data is written to disk.

In Hadoop, the Maps outputs are made available to the reducers on the network over HTTP protocol.

The Reduce phase is constitute by three steps, the copy, the sort/merge and finally the Reduce itself. Each reducer will generate several threads to fetch the outputs from the Maps in parallel, by default 5, but configurable from the property `mapred.reduce.parallel.copies`.

The reducers start to fetch the outputs as soon as a mapper finishes, to optimize the copy time, since they usually finish in different times. Each reducer needs to get the output for its particular partition. The outputs are copied directly to the reducer memory, and saved to disk in a way similar to the writing of the Maps outputs in the mappers. When the buffer memory is full they are temporarily saved on disk, and a background thread starts to merge them, before all outputs are fetched, again to minimize the total time of the copy process.

During the merge, the sorted order of the data is preserved, so in Hadoop implementation one can say – more properly, – that the sort process is done on the mapper side and on reducer size the process is essentially a merge by (already) sorted keys.

The merge is done in rounds on a given number of outputs, with a default merge factor of 10, controlled by the io.sort.factor property. The last round of the merging is not merged to a single file on disk, but is immediately directed to the Reduce, saving time and an IO operation on disk.

On the Reduce step, Hadoop implementation follows the MapReduce model quite strictly, with the Reduce function being called once for each key, as prescribed by the model. The Reduce function, in Hadoop, always emits as output a pair, or pairs, of key and value, but one of these elements can be emitted null if the user wants a plain list as a final result.

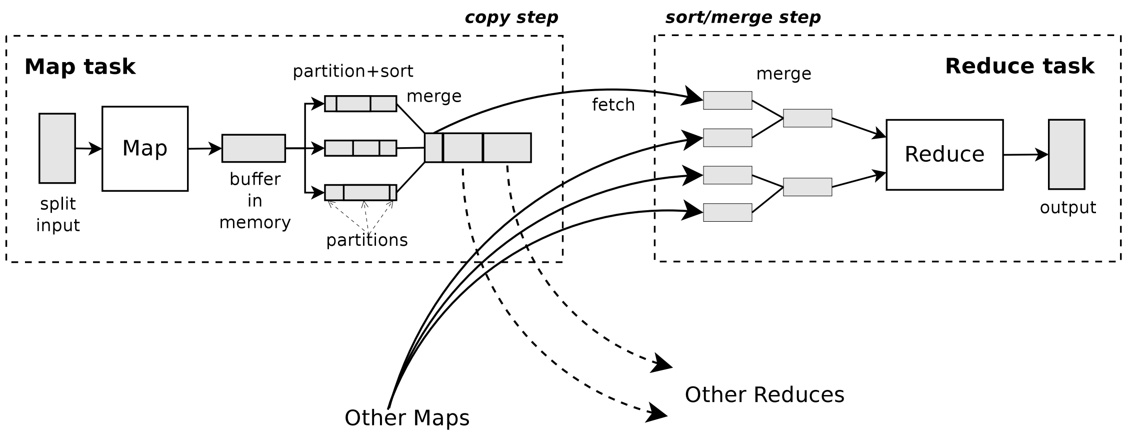


Figure 3. Data flow of a MapReduce in Hadoop.

Figure 3 shows the details of the data flow of a MapReduce process in Hadoop, with the steps mentioned previously signalized. Note that a Reduce task fetches data from only one partition on each Map, and that data of other partitions data will be directed to Reduce tasks on their respective reducer.

The final output of the Reduce is written directly to the distributed filesystem, typically HDFS, ultimately taking advantage of the fault-tolerant filesystem to ensure the safety of the final results, possibly also in big size files and result of many hours of computation, too precious to be lost due to some disk or network failure.

The user will need to retrieve the results from the distributed system, via the Hadoop line command or using some included tools, developed to easy access to the output results that in some cases spreads across several files and in Hadoop internal binary formats.

A table with the most important configuration properties of Hadoop, their default values and configured values used for this work is given in the Appendix A, Apache Hadoop Installation and Configuration, page 99.

3.3 Outlier Detection.

The CURIO algorithm, a density-based method proposed by Ceglar et al.[9], was chosen as a starting point and base for a distributed computing implementation, to be developed within the MapReduce programing model, of an outlier detection method.

A density-based method was the preferred choice due to the ability of these methods in discover local outliers[9], the atypical instances relatively to their local neighborhood[6], with generic good performances and results well known in literature. The CURIO algorithm was chosen due to its intuitively approach and simplicity of data process visualization, allowing a relatively simple and easy implementation on the MapReduce data model.

The several distributed algorithms developed for this work will be presented on section 4. In the next two sections the CURIO algorithm will be described and detailed, followed by an explanation of the MapReduce data model and its Hadoop framework implementation.

3.3.1 CURIO Algorithm.

The CURIO algorithm[9] is a density-based method, applicable to datasets constituted of numeric attributes, which employs partitioning of the data space domain. The data space is discretized into partitions, also called cells, to which the instances are indexed. Cell density is defined, in this context, by the quantity of instances allocated to a cell.

It's important to emphasize that the term partition, in this context, doesn't refer nor is related with the concept of partition on the MapReduce model, and that the two concepts must be kept clearly separated. In relation to the dataset partitioning, the terms partition and cell will be used indistinguishibly.

Dissimilarity evaluation is usually done by definition of a distance, being more dissimilar the objects with bigger distances from the more common ones. But with the data allocated to the cells on the partitioned space, the atypical instances can be detected without the need to explicitly define or calculate distances. The definition of outlier as an object dissimilar from the majority of the others objects in the set results necessarily – in terms of cells allocation, – that outliers are allocated in relatively sparse, low density cells (cf. figure 4). Those cells, with a very few number of instances, will be designated, for commodity, as potential outlier cells.

The fact that an outlier must belong to a low density cell, is a necessary but not a sufficient condition. If the instances on the potential outlier cells were classified immediately as outliers, a great number of false positives would be generated, mainly due to situations of low density cells on the frontiers of dense cells regions. To eliminate this problem, a validation of potential outliers cells as a second step is required.

A possible solution is to take a nearest neighbors search, and defining a validation criterion based on neighborhood density. Nearest neighbors cells are those at a one-cell distance of the one being checked; that is, the 8 contiguous cells in the 2D dimension case, 26 cells in 3D dimension, etc. Is reasonable consider that outliers – the atypical, dissimilar instances, – are naturally isolated from the others instances of the set, and so, a good rule can be the validation of those sparse cells which neighborhood have also an equally relative low density. The false positives low density cells on frontiers of clustered regions wouldn't then be accepted, since they neighborhood would include cells high density populated.

In CURIO algorithm, the validation criterion was therefore defined as the total density of nearest neighborhood being identically low as the cell density. If such criterion is achieved the potential cell is validated positive and its instances classified as outliers.

Figure 4 shows an example of a 2D dimension space partitions, with two different values of granularity.

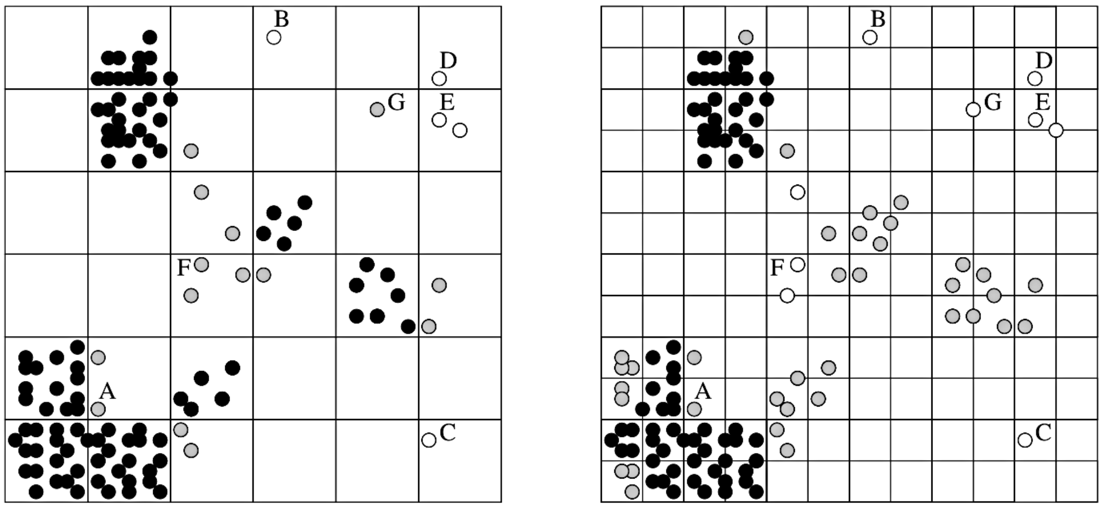


Figure 4. Example of partitions on a 2D dimension space with different granularities (precision).

The different tones of objects represents the algorithm classification, with samples in grey being the potential outliers and in white the ones to be classified as outliers, with possible different classifications according to the granularity choice. Note as instances on cells F and G are validated as outliers when the granularity is duplicated, due to the lowering number of instances as nearest neighbors, while instances on the edges of dense clusters, like cell A, still remains as potential outliers.

To control the granularity of partitioning, the number of cells defined on the data space, a parameter need to be defined. CURIO denominates that parameter as precision. The density threshold, used for cell classification as potential outliers and outlier validation, will be the other control parameter, and will be denominated tolerance.

The details on the implementation of the partitioning and validation steps are described in the next section.

3.3.2 CURIO Algorithm: Partitioning and Validating.

As stated before, the CURIO algorithm consists of 2 stages; a quantization step, where basically the samples space is discretized in cells and the number of instances in each cell counted, and a second step, where the cells identified as potential candidates are validated.

Quantization will be done upon definition of a parameter, the precision, P. Precision controls the granularity of the partitions layout. The partitioning of space and the allocation of instances to the cells is done on the same step, in a single sequential reading of the dataset, using a function that define the cells index from the instance itself.

The index is constructed from the binary representation of the values of each attribute of the instance, by concatenation of P most significant bits. The number of partitions will depend on the chosen value of P and will be, for k dimensions, $(2^P)^k = 2^{P*k}$. Since cell indexes are defined from the existent instances, empty cells will not have indexes generated and will not be listed by the algorithm, making the effective number of partitions on memory smaller than 2^{P*k}.

Each index will unequivocally identify a cell. While partitioning occurs, a counter is incremented for each new instance mapped to an index. The updated counts per index are the values to be kept associated to each index (a hashtable is suggested as an efficient structure by Ceglar et al.[9], according to their performance tests).

To clarify the indexing scheme consider, as an example, the following points of a 3D dimension space: { (2, 9, 8); (9, 14, 17); (10, 15, 4) }. The binary representation of each value by dimension and respective cell indexes by concatenations of the P most significant bytes, according to the given value of precision, is shown on table 1:

	dim1	*dim2*	*dim3*	cell index:	$P=2$	$P=3$
$(2,9,8)$	0010	1001	1000	$\rightarrow$	00 10 10	001 100 100
$(9,14,17)$	1001	1110	0111	$\rightarrow$	10 11 01	100 111 011
$(10,15,4)$	1010	1111	0100	$\rightarrow$	10 11 01	101 111 010

Table 1. Cell indexing examples for 3 samples of a 3D space, for 2 values of precision: $P=2$ and $P=3$.

Note as the second and third point are assign to the same or a different partition, according to the precision value, P, due to the fact that the bigger value of P defines smaller partitions, in higher number, upon the same space.

In validation step, cells are classified as potential outliers cells if the number of instances counted is equal or less than a certain tolerance value T, also to be defined by the user.

To validate the potential outliers a nearest neighbor search is executed. For each potential outlier cell, all neighbor cells are checked and their counts are summed. If the total sum of counts of the neighborhood is also less or equal than the tolerance, according to the validation rule proposed in previous subsection, the cell is classified as an outlier cell, and the instances assigned to it will be classified as outliers. To accelerate the process, as soon as the sum of neighbor's counts exceeds the tolerance, a case of false positive, the neighbors checking can be terminated.

The two conditions required for being an outlier instance are, in resume:

$$\text{instance} \in \text{cell} : \text{cell}_{\text{count}} \leqslant T \quad \wedge \quad \sum_{i}^{\text{neighbors}} \text{count}(\text{cell}_i) \leqslant T\,,$$

no explicit distance or other measure of dissimilarity is further used.

The CURIO algorithm can be summarized as:

Algorithm 1. CURIO

```
                    Step 1. Partitioning.

Define parameter precision: $P$.
// Discretize and count instances by cell:
For each instance in Dataset:
    index = getIndex(instance, $P$)
    Cells(index) = Cells(index) + 1

                    Step 2. Validation.

Define parameter tolerance: $T$.
// Verify potencial outliers cells:
For each instance in Dataset:
    index = getIndex(instance, $P$)
    If Cells(index) $\leqslant T$:
        // Validate potencial outliers cells:
        If Sum(NearestNeighbors$_{\text{Cells}}$) $\leqslant T$:
            Set instance as outlier
```

For more details on possible implementations of CURIO algorithm, the authors[9] give detailed information on the choice of the best structure to hold the partitions information and pseudocode to the validation step.

3.2.3 Complexity Analysis.

In total the full algorithm executes two readings of the dataset, have therefore a complexity of $O(2N)$, where N is the total number of instances. That cost, although linear and comparatively good relatively to other competitor algorithms, usually with at least $O(3N)$[9], can yet be too heavy on really huge datasets.

On the second step, the algorithm undertakes a nearest neighbors search, that is exponential on dimension k, $O(C\,3^k)$, with C being the number of potential outliers cells. In order to the neighbors search still allowed the algorithm efficiency, the number of potential cells and, in particular, the dimension must be small. The authors mention yet the dynamic termination, ending of the neighbors search as soon as the validation criterion been overpassed, as a way of reducing the exponential growth, resulting in practice that the C factor will be reduced to the number of validated potential cells, expected to be much lesser than the number of potential outlier cells[9]. However that reduction will depend on the number of validated outlier cells, that will varies from dataset to dataset, impossible to estimate a priori, and the exponential escalating with of dimension will always be present.

The authors results shows[9] a reasonable efficiency for dimension $k < 10 \sim k < 12$, above which the execution times becomes too long. That limit for the dimension is not quite satisfactory, with very common situations of datasets with much higher numbers of attributes, and consequently outside the abilities of CURIO.

In terms of memory usage, the number of partitions will depend on the precision chosen and the number of dimensions of the dataset, $O(2^{Pk})$. Since the cells indexes are built directly from the existent instances, only cells that actually contain occurrences are registered. That means, in practice, that empty cells are never indexed nor listed, and the total number of partitions kept in memory will be smaller – in most cases much more smaller, – than 2^{Pk}, therefore also depending on the density and distribution of data.

3.2.4 Final Comments on CURIO Algorithm.

As it was mentioned before, the CURIO algorithm processes datasets with numerical attributes exclusively, due to need of a quantifiable data space where partitions can be defined. However the algorithm may still be applicable, after a careful conversion of categorical attributes to numeric, since assign distinct numeric values to each category by dimension will imply the assignment of the instances to different partitions, grouping them by similarity.

The method employed by CURIO for partition indexing uses binary representations of each attribute. The index function should be able to take care of any base conversion needed, avoiding the imposition of processing exclusively binary datasets or the requirement of a full dataset conversion, this last one not always convenient due to memory and disk storage problems with the size of data. Implementations of code can easily be made with specialized features for base independency and conversion to binary as needed.

The function used for indexing on the distributed versions to be discussed later, is the same as the original non-distributed. In the developed code for this work, the indexing function accepts data in the most common decimal representation, executing binary conversions, after a rescale, to ensure a sufficient number of significant bits (e.g. normalized data can be scaled to 0-1000, or any other scale chosen by the user) to be selected by the precision parameter.

The original article[9] does not refers any heuristics or includes any reasoning for a choice of the control parameters values. To solve this shortcoming some knowledge of the data will be required and some heuristics should be outlined to determine good starting values for that choice. In most cases some preliminary experiments, opportunely over just a small sample of the dataset, may be necessary to achieve good results.

The minimum and maximum value should be known or determined prior to the definition of precision and the partitioning. This will define the limits of the data space upon which the partitions are outlined, meaning, in terms of binary representation, the total number of bits to use and the maximum of precision possible to assign.

As an example, if 0 and 100 are, respectively, the minimum and the maximum values expected in any dimension of some dataset, in binary the values will be spread between 0000000 and 1100100, with 7 significant bits. That would be the maximum possible value for the precision, or, more precisely, an integer in the range $2 < P < 7$ (being the extremes too coarse or too dense to be of use).

The ideal choice of precision will ultimately depend of the nature of data and its underlying distribution. No precise rule can be derived and some preliminary experiments should always be done for each dataset. As general guideline: a too low precision will create a partitioning too coarse and outliers may be not founded, a value too high can create situation with false positives, due to the excessive fine grain of the cells network, besides the growing run times and memory resources required.

The tolerance will also require some fine-tuning or some knowledge of the data structure. A good starting point is to define tolerance as a small percentage of the total number of instances on the dataset. In that way, consider the algorithm as a filter for a specified quantity of occurrences that are most dissimilar of others. This value can then be adjusted if the outliers output contains too much or too few outlier instances relatively to either some eventual estimation or to the total number of outliers, or some statistical analysis may be applied on the deviation of those classified outlier instances, if an extra rigor on results is a critical factor. However, such treatments are beyond the scope of this work (see Barnett & Lewis[7] for a more statistical analysis of detected outliers).

4. Proposed Methods

This chapter presents, in detail, the elaboration of a distributed version of CURIO, employing the MapReduce programming model. Several variants are derived next, introducing different types of optimizations, either in terms of conceptualization as in exploration of the MapReduce specific characteristics.

The first three variants presented, explore some optimizations related with the modeling of data by the maps, with the last of those having its own set of specialized combiner functions for their jobs. The forth variant introduces a new feature, indexed counts, that with conjugation with chained reduces allow to reduce considerably the number of values emitted by the map on validation step.

In the end of the chapter a new distributed algorithm is proposed, which although based on CURIO and using some of the techniques developed on the previous distributed adaptations, employs a different approach for the Nearest Neighbors search on the validation step, that release it from its exponential complexity on dimension, resulting on a massively reduction of redundant values outputted and correspondingly expected reduction of execution times.

While the proposal of this thesis is oriented for work and test on the Hadoop framework, the design and development of the algorithms for a distributed version and new method were kept completely abstract, within the MapReduce model, and can been deployed inside any other framework that implements that data model. The development in the Hadoop platform, code specifications and details for that framework are provided in the Appendix C, Implementation under Hadoop.

4.1 CurioD, a MapReduce adaptation of CURIO.

A MapReduce version of CURIO will require at least two distinct jobs, one for each step of the original algorithm, and a third one for the extraction of the discovered outliers from the original dataset (all the algorithms developed generate as output a new dataset containing the outliers instances). The MapReduce algorithm developed for this work is named CurioD.

The first step of CURIO is basically a partitioning of the dataset space into cells and a count of instances allocated per cell. This can be implemented as a simple variant of the canonical word count application of the MapReduce model, presented before on section 3, Theory, and will constitute the first job of this adaptation.

In the Map task, the function receives each instance on the dataset and emits a pair with the index of the partition to which the instance belongs and a constant value, 1, indicating one instance associated to that index. The partition index is defined by the same function as CURIO, from the instance attributes converted to binary and P most significant bits from each dimension concatenated, with parameter P, the precision, defined by the user.

The Reduce task just counts the number of elements, all equal to 1, in the list of values for each index received, emitting a pair $\langle \text{index}, \text{count}() \rangle$[2].

The data flow of the first job, the partitioning, can be schematized as:

$$\{\,\text{instance}\,\} \overset{\text{Map}}{\to} \left\langle \,\text{index}(\text{instance}, P)\,, 1 \right\rangle \overset{\text{Red.}}{\to} \langle\,\text{index}\,, \text{count}()\,\rangle$$

2. In this work, as convention, the received MapReduce pairs will be noted as $(\,,\,)$ and the emitted pairs as $\langle\,,\,\rangle$.

The next job will validate the potential outliers resultant of the counting on this job. That's not trivial in the MapReduce formalization and will require a careful and detailed description to make the validation process clear.

A potential outlier cell is defined as a partition with the total of instances count equal or less than a parameter T, the tolerance, also to be defined initially by the user. In CURIO, validation of outliers is made by summing all the instance counts in the nearest neighbor cells and requiring that the total sum also be less than or equal to the Tolerance.

Since for each cell to be checked the information relative to the other cells is needed again, to locate their neighbors and respective instance counts, this task is neither direct nor trivial, particularly if the implicit data reading recurrence must be avoided.

For each partition to be validated, the Map will have access to the information relative to that and only that partition. The instance counts of other partitions will not be accessible. The MapReduce model, due to its conceptualization in two steps and its non-sequential nature, impose the solution of these problems as a two break down procedures – to be described next, – with the data being mapped on first step in a way that allow the operations to be successfully accomplished on the second step.

The Map function receives, for the partition being verified, the respective pair $\langle \text{index}, \text{count} \rangle$, which will be the only information available on each call of the Map function. However, due to the way cell indexes were defined, it is possible to easily build the list of indexes of its nearest neighbors. The instance counts of the neighbors can't be directly accessed, but the neighborhood relation is symmetrical, and it can be considered instead, the fact that the neighbors have, reciprocally, the present cell on processing as neighbor, with the known count; telling each neighbor that have this count value, by emit its own count for each neighbor index.

If the Map emits the count value received for every possible neighbor index, even if it refers to cells without instances in the dataset, they will be sorted and merged by index in a list, during the reduce step, and the Reduce function will receive the correct list of counts from the neighbors of each partition. The total number of instances in the neighborhood of each cell can then be easily calculated by iterating in the list, performing the sum of its contents.

To exemplify the counting , consider the set of pairs with indexes I*i* and they corresponding count, C*i*:

$$\{ (\text{I1, C1}); \; (\text{I2, C2}); \; (\text{I4,C4}); \; ... \} ,$$

assuming that a partition with index I*i* have as nearest neighbors I(*i-1*) and I(*i+1*), and that indexes referring to empty cells are not listed, like the case of I3 in the above set. The MapReduce will be:

Map:

$$\begin{aligned}
(\text{ I1, C1 }) &\rightarrow \langle \text{I2}, \text{C1} \rangle \\
(\text{ I2, C2 }) &\rightarrow \langle \text{I1}, \text{C2} \rangle \\
&\rightarrow \langle \text{I3}, \text{C2} \rangle \\
(\text{ I4, C4 }) &\rightarrow \langle \text{I3}, \text{C4} \rangle \\
&\rightarrow \langle \text{I5}, \text{C4} \rangle \\
&\;\;\cdots
\end{aligned}$$

Reduce:
$$
\begin{aligned}
(\text{ I1, [C2] }) &\rightarrow \langle \text{I1}, \text{C2} \rangle \\
(\text{ I2, [C1] }) &\rightarrow \langle \text{I2}, \text{C1} \rangle \\
(\text{ I3, [C2, C4] }) &\rightarrow \langle \text{I3}, \text{C2} + \text{C4} \rangle \\
&\ldots
\end{aligned}
$$

and the resulting output will contain the sum of the neighbor's counts of each cell.

Two problems need yet to be solved. One will be the identification, on reduce phase, of the potential outliers cells. A non-potential outlier cell, those with a count of instances larger than the Tolerance, must be declassified in the Reduce step of the validation, and the cell not considered neither on the neighbors' counts sums nor in any further emission to the output. This impose that the reduce must also receive the instance counts of the cell itself. But then, a way to distinguish those counts from the counts of the neighbors must be considered.

A second problem arises when the Reduce have to deal with empty partitions, i.e. with no instances from the dataset. It was necessary, in the map step, to emit counts to all possible nearest neighbors cells, even if they were empty, originating also for those a list of neighbors' counts in the Reduce step (see example above, for index I3). However, as the sum of the neighbors' counts of the empty cells is not needed, since obviously no outlier instances are located there, for efficiency purposes it shouldn't be carried out. Those cases must be identified, in the Reduce step, and not considered in the final output.

One simple solution for both issues can be accomplished by emitting the negative of the count for the cell itself (instead of the real value of its instances count). If the negative of the count is used, it is easy to distinguish neighbors counts from the instances count on the cell, and still keep access to the value of the instances count, by getting the negative again. The inexistence of a negative value indicates an index referring an empty cell, making easy to deal with the second issue.

In the previous example this solution would result in:

Map:
$$
\begin{aligned}
(\text{ I1, C1 }) &\rightarrow \langle \text{I1}, -\text{C1} \rangle \\
&\rightarrow \langle \text{I2}, \text{C1} \rangle \\
(\text{ I2, C2 }) &\rightarrow \langle \text{I2}, -\text{C2} \rangle \\
&\rightarrow \langle \text{I1}, \text{C2} \rangle \\
&\rightarrow \langle \text{I3}, \text{C2} \rangle \\
&\ldots
\end{aligned}
$$

Reduce:
$$
\begin{aligned}
(\text{ I1, [-C1, C2] }) &\rightarrow \langle \text{I1}, \text{C2} \rangle \\
(\text{ I2, [-C2, C1, C3] }) &\rightarrow \langle \text{I2}, \text{C1} + \text{C3} \rangle \\
(\text{ I3, [C2, C4] }) &- \quad \textit{empty cell, no emission} \\
&\ldots
\end{aligned}
$$

Therefore, a negative value on the counts list indicates to the Reduce function that the index represents a partition containing instances, but its count will not be added to the sum of the neighbors counts. Instead it will be used to check if the cell is a potential outlier cell, by comparing its negative against the Tolerance, and the Reduce terminates in case of a count bigger than it (since the neighborhood validation is useless for non potential outliers).

To clarify the MapReduce version of the procedure a simple and concrete example is given bellow. Consider that in following table are the instances count per cell, on a 1D dimensional dataset, from some hypothetical first job:

0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
3	1	4		6		7	9		1		3	8		5

Table 2. Cells instances count example.

The first row contains the index of the cells and the second row the instances count. The first cell, with index 0001, has 3 instances and the neighborhood (cell 0010) have only one cell. The second cell, with index 0010, has only 1 instance and the neighborhood (cells 0001 and 0011) has a total of $3+4=7$ instances, and so on.

Defining, for the example purpose, the Tolerance as $T=5$, the MapReduce solution presented will produce:

MAP

(0001, 3) $\rightarrow \langle 0001, -3^* \rangle$
$\rightarrow \langle 0010, 3 \rangle$
(0010, 1) $\rightarrow \langle 0010, -1^* \rangle$
$\rightarrow \langle 0001, 1 \rangle$
$\rightarrow \langle 0011, 1 \rangle$
. . .
(1101, 8) $\rightarrow \langle 1101, -8^* \rangle$
$\rightarrow \langle 1100, 8 \rangle$
$\rightarrow \langle 1110, 8 \rangle$
(1111, 5) $\rightarrow \langle 1111, -5^* \rangle$
$\rightarrow \langle 1110, 5 \rangle$

* always emit the negative of the count for the received cell index.

REDUCE

(0001, [-3, 1]) $\rightarrow \langle 0001, 1 \rangle$
(0010, [-1, 3, 4]) – $(3+4>T \Rightarrow \textit{no emission})$

(0011, [-4, 1]) $\rightarrow \langle 0011, 1 \rangle$
. . .
(1101, [-8, 3]) – $(|-8|>T \Rightarrow \textit{no emission})$

(1110, [2, 5])** – (*empty cell*)

(1111, [-5]) $\rightarrow \langle 1111, 0 \rangle$

** no negative value is present in the list, the index refers to a cell with no instances, it is unnecessary to emit the sum of the neighbors counts.

Example 1. Count of instances on Nearest Neighbors cells with MapReduce.

Note that cells like 0100 or 1110 are empty, and the Map will not produce any output; but the index will appear on the list of some neighbor cells and will be emitted, generating entries for the Reduce without any negative count. When the Reduce find such cases it will skip the unneeded sum and will not emit any pair, as expected for an empty cell. The Reduce also skips the index 0010, since the neighbors' sum is higher than the tolerance, an invalidated potential outlier, and the index 1101 that not refers to a potential outlier.

The above procedure generates, on the Reduce step, the total sum of instances count on neighbors' cells, for each partition. To get a list of partitions containing exclusively outliers, it is necessary to impose the outliers validation condition, of both neighbors counts and cell count being less than or equal to the Tolerance, to filter the indexes emitted. The cells count is obtained, of course, by taking again the negative of the received negative value.

The data flow of the second job, validation of potential outliers, can be schematized as:

$$\{(\text{index}, \text{count})\} \;\rightarrow\; \{\langle\,\text{index}, -\text{count}\,\rangle, \langle\,\text{indexNeighbor0}, \text{count}\,\rangle, \ldots.\} \;\;\rightarrow^{*}\; \langle\,\text{index}, \text{Sum(counts)}\,\rangle$$

*emits only if a negative value is present on the received list, |-count| $\leqslant T$ and Sum(counts) $\leqslant T$.

The output of the second job will contain a list of all partitions indexes with outliers. A third job will then be needed in order to extract an explicit list of outlier's instances, the ones that are located on the partitions of that list.

This is also a simple MapReduce job, a comparison of lists. The configuration needs both locations, from the dataset and the output of the second job - the list of outlier partition's indexes. The Map function will have as entries now, data from both inputs, which are from different types, instances and indexes. They will have to be filtered according to their type, and the indexes recalculated again for the instances coming from the dataset.

For each entry received, the Map emits, if it comes from the dataset, a pair of ⟨index, instances⟩ or, if it comes from the output of the second job, a pair with ⟨index, *null*⟩. The token *null* value will be used as an indicator of an outlier partition index.

Each Reduce function receives an index and the list of all instances associated to that index, besides the token *null* in the case of the index corresponds to an outlier partition. If a *null* is in the list, the Reduce will emit the instances contained in the list, i.e., all the instances of that partition (outliers); otherwise the index corresponds to a non-outlier partition and the Reduce won't emit in that case. The result will be a list of outlier instances as output.

The following example uses the data from the previous example to illustrate the working of the third job. The final result of the second job, for a Tolerance $T = 5$, was 4 outlier cells, with indexes 0001, 0011, 1010 and 1111, with 13 outliers out of a total of N instances. Note that a *null* value can be originated from any of the map outputs on the cluster and take any position on the value lists.

MAP

$$\begin{aligned}
(\ \text{inst}_1\ ) &\;\rightarrow\; \langle\ \textit{indexOf}(\text{inst}_1), \text{inst}_1\rangle \\
(\ \text{inst}_2\ ) &\;\rightarrow\; \langle\ \textit{indexOf}(\text{inst}_2), \text{inst}_2\rangle \\
(\ \text{inst}_3\ ) &\;\rightarrow\; \langle\ \textit{indexOf}(\text{inst}_3), \text{inst}_3\rangle \\
&\;\;\cdots \\
(\ \text{inst}_N\ ) &\;\rightarrow\; \langle\ \textit{indexOf}(\text{inst}_N), \text{inst}_N\rangle \\
(\ 0001,\ 3\ ) &\;\rightarrow\; \langle\,0001, \boldsymbol{null}\rangle \\
(\ 0011,\ 4\ ) &\;\rightarrow\; \langle\,0011, \boldsymbol{null}\rangle \\
(\ 1010,\ 1\ ) &\;\rightarrow\; \langle\,1010, \boldsymbol{null}\rangle \\
(\ 1111,\ 5\ ) &\;\rightarrow\; \langle\,1111, \boldsymbol{null}\rangle
\end{aligned}$$

REDUCE

(0001, [inst, inst, ***null***, inst]) $\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

(0010, [inst])* – (*no emission*)

(0011, [inst, ***null***, inst, inst, inst]) $\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

. . .

(1010, [inst, ***null***]) $\rightarrow \langle$ inst $\rangle$

(1100, [inst, inst, inst])* – (*no emission*)

(1101, [inst, inst, ... , inst])** – (*no emission*)

(1111, [inst, ... , ***null***, inst]) $\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

$\rightarrow \langle$ inst $\rangle$

* for non-potential outlier instances the Reduce emits no value.

** the lists for non-outlier cells can have any number of instances, according to the case.

Example 2. Filtering the outlier instances on the third job of CurioD.

The outlier partitions will have, by definition, at most T instances. But the non-outliers partitions lists on the Reduce function can have any number of instances, and eventually a very large number in the case of big datasets, from the cells highly populated. Recall that the *null* values can come from any map output and therefore its position on the values list cannot be known for sure. The search for a *null* entry may imply a full check on the lists received, being the search obviously more costly and time consuming as bigger the list is.

So an optimization was introduced here. The list check is terminated and the function exits without any further action, if the number of values checked becomes higher than the Tolerance parameter, denoting an obvious non-outliers cell.

The data flow of the third job, listing of outliers, can be schematized as:

$$\{\ \{\text{instances}\},\ \{(\text{index, sum})\}\ \} \rightarrow \{\ \langle \text{index}, \text{instance}\rangle,\ \langle \text{index}, \textit{null}\rangle\ \} \rightarrow^{*} \langle \text{instance}\rangle$$

*emits only if a *null* value is present on the received list and the list size is less than or equal T.

With all three jobs together, the CurioD full algorithm is given bellow, Algorithm 2. CurioD listing:

Algorithm 2. CurioD, a MapReduce version of CURIO.

```
// Job1, Partitioning:
  $S$ = { instances }
  // Map
    for each instance in $S$
        index = calculateIndex(instance)

          emit < index, 1 >

  // Reduce
    receive { (index, list[Values]) }
    for each index in { (index, list[Values]) }
        count = sum(list[Values])
        emit < index, count >


// Job2, Validating Potential Outliers:
  receive { (index, count) }
  // Map
    for each index in { (index, count) }
        emit <index, − count >
        for each indexNeighbor in neighborhoodOf(index)
            emit <indexNeighbor, count >

  // Reduce
    receive { (index, list[counts]) }
    for each index in { (index, list[counts]) }
        cellCount = 0
        for each count in list[counts]
            while sumNeighbs <= Tolerance and cellCount < Tolerance
                if count < 0
                    cellCount = -count
                else
                    sumNeighbs = sumNeighbs + count
        if  0 < cellCount $\leqslant$ Tolerance and sumNeighbs $\leqslant$ Tolerance
            emit <index, sumNeighbs >


// Job3, Listing Outliers:
  receive U = { instances } ∪ { (index, total) }
  // Map
    for each entry in U
        if entry is instance
            emit < calculateIndex(entry), entry >
        else
            emit <index, null >

  // Reduce
    receive { (index, list[Values]) }
    for each index in { (index, list[Values]) }
        if size(list[Values]) $\leqslant$ Tolerance and list[Values] contains null
            for each outlier in list[Values]
                emit < outlier >
```

The complete data flow during the execution of CurioD algorithm is shown in Figure 5.

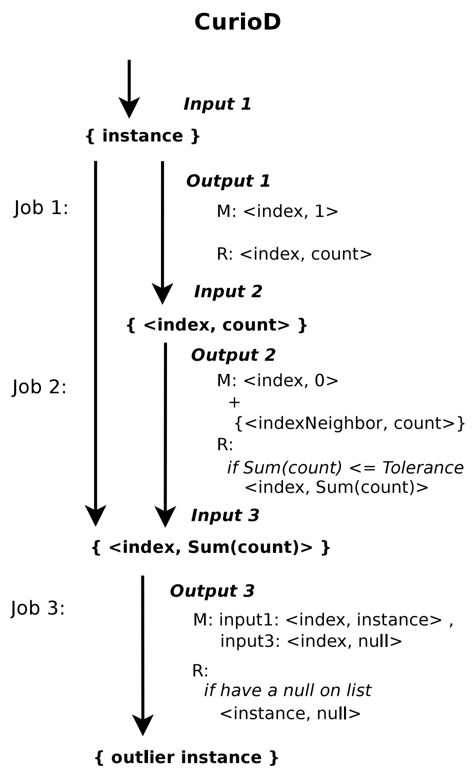


Figure 5. Data flow of CurioD algorithm.

4.1.1 Complexity Analysis. Possible optimizations.

It's useful to introduce the following notation. Let:

N be total number of instances in the dataset (the size),

k, the number of attributes in the dataset (the dimension),

c, the number of identified partitions, those containing instances from dataset (no information about empty partitions is generated during the indexing on first step), and

n the number of partitions with validated outliers.

Note that will be always $n < c < N$ and, generally, $n \ll N$. It's also important to note that, although c, the number of cells, depend on the instances of the dataset, adding more instances to a dataset will not necessary imply that c will also get bigger, so the growing rate of c is much smaller than than the one of N.

The CurioD algorithm has 3 jobs with 2 tasks each. The basic six tasks executed are: read all dataset sequentially, go through the list of identified partitions performing sums, generate all nearest neighbors of each partition, sum on list of nearest neighbors, read of both dataset and last output and, finally, search in the list of partitions indexes for the outlier cells and emit a maximum of T outlier instances for each.

The complexity analysis per task can be resumed on the following table:

Job 1	Map:	$O(N)$
	Red:	$O(c)$
Job 2	Map:	$O(c3^k)$
	Red:	$O(T3^k)^*$
Job 3	Map:	$O(N+n)$
	Red:	$O(c+n\mathrm{T})$

Table 3. Complexity by task on CurioD.

* The sum on neighbors counts will be aborted and the emission skipped if total reaches a value higher than the tolerance T.

On the first job, the data partitioning, the first task reads once the all dataset, generating indexes for the located partitions, essentially a binary conversion, for all N instances in the dataset. The second task checks all generated c indexes and sums the constants 1 on each list received. The complexity of this job will be of $O(N)+O(c) \sim O(N)$.

The algorithms developed on this work only consider the nearest neighbors, meaning the ones in the immediate neighborhood, at 1 cell distance, D, for a given cell. The number of neighbors of a partition on a k-dimensional space is $(2D+1)^k$, that is 3^k possible nearest neighbors for each cell.

On the second job, the neighbors validation, the third and forth tasks will, respectively, generate a list of all possible nearest neighbors for each one of the c partitions and sum the counts emitted on all generated indexes. Each cell will have 3^k neighbors' indexes, so the all job will have an exponential complexity of $O(c3^k)+O(T3^k) \sim O(c3^k)$.

The third job will read two inputs, the N instances of the dataset and the n discovered partitions with outliers, on the first task, the map. On the second task, the reduce receives the c indexes with the list of values containing all the instances and the n null entries signalizing the index as representative of an outlier cell. For each c indexes it must search the associated list of instances, but the check is terminated as soon T instances are counted, because that's the highest possible count for an outlier cell. The search will be at most $O(c\mathrm{T})$. It will then emit, for the n outlier cells, the outlier instances, again with a maximum T, by definition of outlier cell, implying a complexity of $O(n\mathrm{T})$. The complexity of this job will be $O(N+n+c\,\mathrm{T}+n\,\mathrm{T}) \sim O(N+c)$.

The full procedure of CurioD algorithm will have a complexity of $O(N+c3^k+N+c)$, that is, simplifying, $O(2N+c3^k)$, since the number of partitions c will be smaller than the total number of instances N, and less dominant with the growth of the datasets size.

If the dataset have a small number of attributes the algorithm behavior will be dominated by the linear parcel with the number of samples, $O(2N)$, from the two readings of the data. This is the case where distribution computing can be of help on very large or growing databases. But scalability with dimension, k, may impose a serious problem, where distributing computing may not be enough or the appropriate solution. This problem will be explored on detail on section 4.4.

The MapReduce adaptation also makes impossible the optimization shortcut, dynamic termination, suggested for CURIO by their authors[9].

This MapReduce version requires the emission of the counts to every nearest neighbor, for each partition listed. The process of instances counting on the neighbors of a cell is a two tasks job, a map and a reduce from the model, and at any point of the map task there is information about the others neighbors. All cases, potential and non-potential outliers cells, must be considered on the nearest neighbors' count emission. If a non-potential outlier cell did not emit its count, then potential cells on its neighborhood, without such information, would be validated as true outliers cells, resulting on a large number of false positives.

For that reason, the emission of the count of a partition to any possible neighbor is mandatory, making impossible any shortcut on the expensive emission to the neighbors' indexes. With c partitions having each a possible 3^k neighbor indexes, the map will have at least a complexity $O(c3^k)$. Such optimization can however be implemented on Reduce phase, keeping the total job in the order of $O(3^k)$.

Besides the gain expected from the distributed computation by several computers, it is possible to minimize the constant of the complexity on $O(2N)$, by eliminating the need of a second read of the dataset searching for the identified outliers instances. In the case of the MapReduce adaptation this optimization will also eliminates the burden of a full third job, the repeated reading of the dataset and the re-writing of it, implied on the output of the map task. Even considering that this re-write of the all data (plus the writing of the indexes for each instance) is distributed by all nodes, it may become, besides overloading, impossible to accomplish in the case of very large datasets, due to space limitations. Suppressing the third job will also simplify code and maintenance, and reduce the work executed by the software framework on job initializations, configuration readings, network checks, etc.

This optimization is introduced in the next variants of CurioD and is developed on next section.

4.2 Optimized Variants of CurioD.

4.2.1 CurioXD and CurioXXD.

In order to avoid the last search job on the distributed algorithm described in the previous section a new variant, denominated CurioXD, was made. This variant uses, instead of simple count of instances on partitions, lists of references to those instances in the dataset.

The references, to be detailed bellow, allow a direct read of the instances on the dataset, without the need of a search. The procedure can, that way, still be able to verify the number of instances in a partition by count the number of references in the list, but, more important, access the instances directly from the dataset when they classified as outliers, what makes the aforementioned third job unneeded. Also, the artificial emission of the negative values for the count of instances in the cells on the Map of the second job is removed; it's easy to differentiate the instances count of the cell itself, in form of lists of references, from the counts of neighbors, that are just plain numbers. The detailed explanation of CurioXD is given next.

On the first job, the partitioning of the data space, the Map emits not a simple constant 1, but a reference to the instance location on the dataset. That reference can come either from an indexing system, if such is available (for example, some datasets have as 1$^{\text{st}}$ attribute an *id* unique to each instance), or, more generally, the offset position of the start of the instance line in the case of plain ASCII files. The Reduce step needs only to concatenate the list of such references, received already gathered by index by the MapReduce mechanism.

The Map function of the validation step emits the same list of instances for each partition index, and the count of the instances on the list for each nearest neighbor index. There is no need to the artificial use of the negative counts, as on CurioD, since the partitions have the information about their instances always present, which distinguish them from the counts of neighbors that are just a sum of counts, a single number.

On the Reduce task of the validation step, instead of the emission of the partitions indexes that contain outliers, the list of they offset positions will be available and the reducer can access the data files in fast random access mode and emit the instances itself without further searches or extra jobs.

For example, consider the previous example presented in table 2, on CurioD section. In the second job, the first index on the Map, 0001, would have now not a plain count, 3 in the example given, (0001, 3), but a list with the references for the 3 instances:

$$(\ 0001,\ \{\,\mathrm{pos}_1;\mathrm{pos}_2;\mathrm{pos}_3;\,\}\ ).$$

The Map would emit the same list for the index of the cell itself (instead of the negative of the instances count) and the usual plain count, the number of references on the list, to the neighbor indexes:

MAP

$$(\ 0001,\ \{\,\mathrm{pos}_1;\mathrm{pos}_2;\mathrm{pos}_3;\,\}\ )\ \rightarrow\ \langle\,0001,\{\,\mathrm{pos}_1;\mathrm{pos}_2;\mathrm{pos}_3;\,\}\,\rangle$$
$$\rightarrow\ \langle\,0010, 3\,\rangle$$

The reduce receives, for that index, the combination of its own emission, with the references list, and the emissions from the neighbors, the single 0010 in this case, with a plain count. The existence of a list indicates to the Reduce function that is not an empty cell, the size of the list respects the validation rule ($\leqslant T$) and the sum of neighbors counts is also less than T, so the cell is classified as outlier and all instances at the positions referred in the list are read from the dataset and emitted immediately:

REDUCE

$$\begin{aligned}(\; 0001, \; [\{ \, \text{pos}_1; \text{pos}_2; \text{pos}_3; \}, \, 1] \;) \quad & \rightarrow \langle \, \textit{instanceAtLocation}(\text{pos}_1) \, \rangle \\ & \rightarrow \langle \, \textit{instanceAtLocation}(\text{pos}_2) \, \rangle \\ & \rightarrow \langle \, \textit{instanceAtLocation}(\text{pos}_3) \, \rangle\end{aligned}$$

The framework will have an extra load saving and fetching larger outputs (instead of lists of 1's there will be lists of long integers). The instances counts must be replaced by checks on the size of the instances offsets lists, introducing an extra processing effort, since the list are carried from mappers to reducers via text files, they will need to be saved as String concatenations and later split to lists again. This overhead on the algorithm should be rewarding if the dataset size is very large, due the single sequential reading and the total number of jobs being reduced to two.

The data flow of CurioXD is presented in Figure 6:

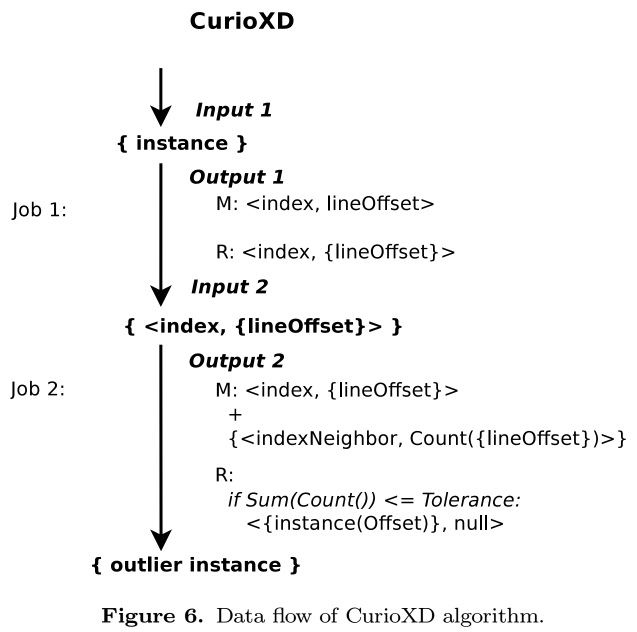


Figure 6. Data flow of CurioXD algorithm.

The algorithm for the CurioXD version is resumed on Algorithm 3. listing:

Algorithm 3. CurioXD.

```
// Job1, Partitioning:
  $S = \{$ instances $\}$
  // Map
    for each instance in $S$
         index = $calculateIndex$(instance)

        value = offset(instance)
            emit < index, value >

   // Reduce
    receive { (index, list[Values]) }
    for each index in { (index, list[Values]) }
        Offsets = concatenate(list[Values])
            emit < index, Offsets >

// Job2, Validating Potential Outliers:
  receive { (index, Offsets) }
  // Map
    for each index in { (index, Offsets) }
        emit <index, Offsets >
        counts = $count$(Offsets)
        for each indexNeighbor in $neighborhoodOf$(index)
            emit <indexNeighbor, counts >

  // Reduce
    receive { (index, {Offsets, list[counts]}) }
    for each index in { (index, {Offsets, list[counts]}) }
      if $0 < count$(Offsets) $\leqslant$ Tolerance
          sumNeighbs = $sum$(list[counts])
          if sumNeighbs $\leqslant$ Tolerance
              for each offset in $split$(Offsets)
                  outlier = $readInstanceAt$(offset)
                      emit < outlier >
```

At this point it is possible to consider yet a further optimization, eventually worthwhile for really large datasets.

Recall that the list of offsets, to be concatenated by the first Reduce during the partitioning, will only be needed for the detected outliers partitions on second Reduce, after the nearest neighbor's validation. It will be used for access the instances from the fewer partitions that would be classified as outliers, and never used otherwise. Consequently, carrying all those lists, specially large precisely for the unrequired non-potential partitions, which will constitute the vast majority for the most common cases, will only create big outputs that will take more space, more time to fetch and read, and extra computational cost in count the number of its instances later on.

At the time of the concatenation of the offsets list it is, evidently, impossible to know which partitions will be classified as outliers, but it's easy to know the ones that are potential outliers. Since the number of potential outliers partitions will still be lesser than the total number of partitions indexed, emitting just those, will constitute a good compromise solution. This extra-optimized version was named CurioXXD.

In CurioXXD, during the concatenation of the offsets list, a counter is incremented at every new instance offset in the partition being processed by the Reduce function, on partitioning step. When the count overpasses the value of the Tolerance – meaning a non-potential outlier cell, – the concatenation is skipped, and the function will continue only with the update of the counter. After all offsets values been considered, the function emits either the simple count or, only for the potential outliers, the full concatenated list of offsets.

On the second step, while the neighbor's validation is being processed, the Map and Reduce functions just need to check if they receive counts or lists, and only in the Map the offset lists of the potential outliers have yet to be counted, to be emitted to the nearest neighbors.

The data flow for this optimization variant, CurioXXD, is represented on figure 7:

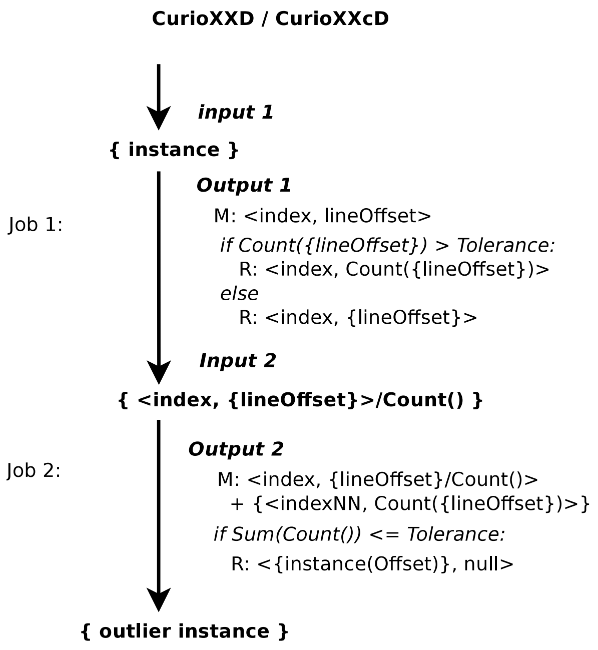

CurioXXD / CurioXXcD
input 1
{ instance }
Job 1:
Output 1
M: <index, lineOffset>
if Count({lineOffset}) > Tolerance:
R: <index, Count({lineOffset})>
else
R: <index, {lineOffset}>
Input 2
{ <index, {lineOffset}>/Count() }
Job 2:
Output 2
M: <index, {lineOffset}/Count()> + {<indexNN, Count({lineOffset})>}
if Sum(Count()) <= Tolerance:
R: <{instance(Offset)}, null>
{ outlier instance }

Figure 7. Data flow of CurioXXD algorithm.

The algorithm, now with the new changes, is presented on listing Algorithm 4. CurioXXD:

Algorithm 4. CurioXXD.

```
// Job1, Partitioning:
 $S$ = { instances }
 // Map
  for each instance in $S$
     index = calculateIndex(instance)

    fileOffset = offset(instance)
      emit < index, fileOffset >

 // Reduce
  receive { (index, list[fileOffset]) }
  for each index in { (index, list[fileOffset]) }
    count = sizeOf(list[fileOffset])
    if count $\leqslant$ Tolerance
      Offsets = concatenate(list[fileOffset]) with separator
      emit < index, Offsets >
    else
      emit < index, count >

// Job2, Validating Potential Outliers:
 receive { (index, value) }
 // Map
  for each index in { (index, value) }
    emit <index, value >
    if value is aList
      counts = count(value)
    else
      counts = value
    for each indexNeighbor in neighborhoodOf(index)
      emit <indexNeighbor, counts >

 // Reduce
  receive { (index, {Offsets, list[counts]}) }
  for each index in { (index, {Offsets, list[counts]}) }
   if 0 < count(Offsets) $\leqslant$ Tolerance
     sumNeighbs = sum(list[counts])
     if sumNeighbs $\leqslant$ Tolerance
       for each offset in split(Offsets)
         outlier = readInstanceAt(offset)
           emit < outlier >
```

Returning to the example presented in table 2, on CurioD Algorithm and with a defined Tolerance of $T = 3$, the result of the first job for the CurioXXD version would now be:

0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
{ ; ; ; }	{ ; }	4		6		7	9		{ ; }		{ ; ; ; }	8		5

and the second job of CurioXXD would produce:

MAP

$$\begin{array}{rl}
(\ 0001,\ \{\,;\,;\,;\,\}\ )^{*} & \rightarrow \langle\, 0001, \{\,;\,;\,;\,\}\,\rangle \\
 & \rightarrow \langle\, 0010, 3\,\rangle \\
(\ 0010,\ \{\,;\,\}\ )^{*} & \rightarrow \langle\, 0010, \{\,;\,\}\,\rangle \\
 & \rightarrow \langle\, 0001, 1\,\rangle \\
 & \rightarrow \langle\, 0011, 1\,\rangle \\
(\ 0011,\ 4\ )^{**} & \rightarrow \langle\, 0011, 4\,\rangle \\
 & \rightarrow \langle\, 0010, 4\,\rangle \\
 & \rightarrow \langle\, 0100, 4\,\rangle \\
 & \quad \ldots \\
(\ 1101,\ 8\ ) & \rightarrow \langle\, 1101, 8\,\rangle \\
 & \rightarrow \langle\, 1110, 8\,\rangle \\
 & \rightarrow \langle\, 1100, 8\,\rangle \\
(\ 1111,\ 5\ ) & \rightarrow \langle\, 1111, 5\,\rangle \\
 & \rightarrow \langle\, 1110, 5\,\rangle
\end{array}$$

* for potential outliers cells the map receives and emits the set of instance offsets.
** for non-potential outlier cells emit only the indexed counts for each index.

REDUCE

$$\begin{array}{rcl}
(\ 0001,\ [\{\,;\,;\,;\,\},\ 1]\ ) & \rightarrow & \langle \mathit{instancesAt}(\{\,;\,;\,;\,\})\rangle \\
(\ 0010,\ [\{\,;\,\},\ 3,\ 4]\ ) & - & (3+4>T \Rightarrow \textit{no emission}) \\
(\ 0011,\ [4,\ 1]\ ) & - & (\textit{no list} \Rightarrow \textit{no emission}) \\
 & \ldots & \\
(\ 1101,\ [8,\ 3]\ ) & - & (\textit{no list} \Rightarrow \textit{no emission}) \\
(\ 1110,\ [8,\ 5]\ ) & - & (\textit{no list} \Rightarrow \textit{no emission}) \\
(\ 1111,\ [5]\ ) & - & (\textit{no list} \Rightarrow \textit{no emission})
\end{array}$$

Example 3. The MapReduce process of the second job of CurioXXD.

As a final note on these two last variants, they will generate exactly the same result, as final output, as the plain CurioD, which is a new dataset containing only the detected outliers instances. The result is obtained with a slight extra computational effort, from the manipulation of the references lists and counting of its elements, but with one less MapReduce job and a reduction of one full reading of the original dataset, replaced by a smaller reading of the n validated partitions with outliers, meaning a reducing of complexity of$O(2N)$ to $O(N+n)$.

4.2.2 Specialized Combiners: CurioXXcD.

With the possibilities and features of the MapReduce model in mind, the use of Combiner functions was considered next as a potential optimization for the distributed versions of CURIO. Combiners essentially minimize the volume of data being written to local disks to the Maps outputs and fetched across the network by the reducers. That may provide an optimization proportionally efficient with the size of dataset, and thus relevant for the first job, but of particular interest for the second job, where the map will emit an exponential high number of pairs by partition, arising from their nearest neighbors validation.

In order to use Combiners, in this case, specialized functions need to be developed, since the Reduce functions of the MapReduce jobs proposed so far don't possess a distributive nature that allows a simple re-use as Combiners.

First, it is necessary to consider that a Combiner can be called recurrently, so the function must be prepared to act both on the output of maps and its own output. In the case of CurioXXD first job, the partitioning step, the output of the Map will contain a list of file offsets; but the Combiner for that task can output, and consequently also receive, the string concatenations of the offsets list or just a single number, the plain count of those offsets, if their quantity is greater than the Tolerance.

$$\begin{tikzcd}
\text{Map} \arrow[r] & \{\,(\,\text{index}, \text{offset})\,\} \arrow[dl] \\
\text{Combine} \arrow[r, shift left] & \{\,(\text{index}, \text{OffsetsList}), (\text{index}, \text{count})\,\} \arrow[l, hook', shift left] \arrow[dl] \\
\text{Reduce} \arrow[r] & \{\,(\text{instances})\,\}
\end{tikzcd}$$

The Combiner will concatenate the offsets for each index, bound them with a separator character, in single strings. However each mapper can't guarantee that had access to all offsets of a cell. Naturally, instances associated to an index can be sent to any mapper in the network. That means that the Combiner will produce offset lists that may be just partial lists, which will be only completely merged on the reducer to where, by key, they will be sent.

Thus, additionally, precautions must be taken in order to ensure that the offsets entries are correctly separated and bounded on the list concatenation. If it happens a repetition or an omission of the character defined to separate the offsets, either their counts will be inaccurate or several offsets can be merged in a single incorrect one on the string conversion. In practice, that means that the Combiner must ensure that any concatenated string will end with the separator character and that the Reduce will make plain concatenations without further insertion of separators.

$$\{\,(\,\text{index}, \text{offset})\,\} \;\longrightarrow\; \text{Combine} \underset{\hookleftarrow}{\longrightarrow} \;\{\;(\text{index}, \text{`offset1;offset2;...'})\;\}$$

The algorithm for the first job, with its own Combiner, must be re-written from the previous CurioXXD, as:

Algorithm 5A. CurioXXcD, partitioning step with Combiner.

```
// Job1, Partitioning:
 $S$ = { instances }
 // Map
      ( no changes )


  // Combine
   **receive** { (index, values) }
   **for each** index **in** { (index, values) }
      **if** values *isList*
         count = *sizeOf*(values)
         **if** count $\leqslant$ Tolerance
            Offsets = *concatenate*(values) **with** separator
            **emit** < index, Offsets >
         **else**
            **emit** < index, count >
      **else**
         **emit** < index, count >


 // Reduce
   **receive** { (index, values) }
   **for each** index **in** { (index, values) }
      **if** values *isList*
         count = *sizeOf*(values)
         **if** count $\leqslant$ Tolerance
            Offsets = *concatenate*(values)
            **emit** < index, Offsets >
         **else**
            **emit** < index, count >
      **else**
         **emit** < index, count >
```

In the second job, the concatenations of offsets are no longer in the process and basically the task consists in performing sums. Additions are associative and commutative and can easily be accomplished with Combiners identical to the Reduce function. The only reason for a customized Combiner function is the fact that the reducer does not emit the total sum when it finishes, but use that sum to evaluate the validation condition and emit a list of instances, after read them from the dataset.

The Combiner for the second job will need only to execute the sums on the partial lists of the neighbor's counts, passing that simplified output along with the concatenated offsets list.

The Reduce will be exactly the same function as in the case of no Combiner, since it will just sum all values received, no matter if they come from the count of a single partition or if some sums had already been performed by the Combiners in the mappers machines.

The algorithm for the second job must be written, based also on CurioXXD variant, as:

Algorithm 5B. CurioXXcD, validation step with Combiner.

```
// Job2, Validation:
 $S = \{$ instances $\}$
 // Map
      ( no changes )

 // Combine
   **receive** { (index, {Offsets, list[counts]}) }
   **for each** index **in** { (index, {Offsets, list[counts]}) }
      **emit** < index, count >
      sumNeighbs = *sum*(list[counts])
      **emit** < index, sumNeighbs >


 // Reduce
      ( no changes )
```

Conceptually, the CurioXXcD version is exactly the same as the previous CurioXXD, but deploying its own specialized versions of Combiner functions for each of its jobs. The first combiner performs sum of instances count and concatenation of available offsets lists, locally on the mappers, and the second combiner performs the sums of the partial neighbors' counts on mappers, before the reducers had fetched them for the final validation. Comparative tests between this version and the previous, will mainly verify the impact in terms of performance of the combiners' implementation and use, and its behavior with the size and number of dimensions. That will be essentially a MapReduce feature to test, not directly related with the nature of the algorithm.

4.3 Chain MapReduce tasks. CurioXRRD.

The optimized versions developed so far, CurioXD and CurioXXD/CurioXXcD, reduce to two the number of jobs and require a single sequential read of all dataset. They still need to emit the instances count for the nearest neighbors of all partitions, keeping a complexity of $O(c3^k)$. The next variant tries to reduce the factor of that exponential complexity.

Let p be the number of potential outlier partitions. Using the notation introduced early in this section, being n the number of outliers instances, c the number of partitions detected and N the total number of instances, is

$$n \leqslant p < c \leqslant N,$$

where the identities occurs only on extremely exceptional and rare cases. Generically, p must be much smaller than c, the number of indexed partitions, according to the definition of potential outlier cell, the ones on the highly sparse regions of the dataset space.

If the process emits only counts for the neighbors of the potential outliers cells, the ones that will in fact need that validation, the exponential dependency on the dimension can be smoothed, reduced to $O(p3^k)$.

The main idea is to decrease, in the second phase of the algorithm, the information emitted, doing it only to the cases that will need it later on the process, that is, when the total of instances in the neighborhood of a potential outlier partition will be compared to the Tolerance, to validate that partition. This means that only potential outliers neighbors will in fact need to receive the counts. But at the time of the emission to the neighbors, the Map, processing a partition, don't know which of its neighbors is a potential outlier; the only related information available will be, whether or not the partition itself is a potential outlier.

Then it will be needed that only potential outliers partitions send to their neighbors their indexes, letting them know they are neighbors, giving access to that information later on, during the Reduce step. And since the potential outliers will also need the counts of the neighbors, an optimal solution would be to send both pieces of information at the same time.

So, in this version is introduced this new concept, indexed counts, to implement the above approach. It will map more information – the counts and the indexes, – to the indexes being processed and emitted by the Map function, reducing not only the emission of counts of the nearest neighbors cells, but also making the emitting of counts for the other cells in much less quantity. More information on the indexed counts will be given bellow.

Note that for the potential outliers neighbors is important to emit also their list of references, in case they'll be positively validated, and for the others just the count associated to its own index, for the validation checks ahead, since that partition will not be classified as outlier cell.

After this step, all the neighbors of potential outliers partitions know which are the neighbors that is relevant to send information. On the other hand, the neighbor partitions had already sent their counts. It is only needed that the non-outliers partitions sent the respective counts to their neighbors that are potential outliers (and only to those!), in order to the validation by the summing of nearest neighbors counts could be completed. At that time it will be possible to identify the potential outliers neighbors, because they are the ones that had sent indexed counts, and they indexes are contained on it. The counts unneeded are discarded and the indexes are used to get the list of the neighbors that effectively are potential outliers.

In resume, it is not reduced the information emissions exclusively to potential outliers, because non-potential outliers must also inform they neighbor about they counts, but for those, the emissions were decreased to just the potential outliers neighbors. Considering that the number of potential outliers is in general much smaller than the others, there will be a substantial reduction on the emissions of counts during the Map step.

The Reduce function for deal with such variant can't be a simple task, like the ones proposed before. The data received from the Map will need to be iterated and processed in two distinct steps. At it was exposed; there must be an emission for the potential outliers neighbors, and then for the others. To avoid an extra full job, this version will use a more complex combination of MapReduce tasks, a chain of Reduces after a Map, M|R+R, and will be designated CurioXRRD.

For the first phase of the CURIO process, the partitioning of data space, nothing new or additional is required. Any of the partitionings jobs proposed before can be used, with special focus on the last ones, due to its optimized performance. For the version to be developed here, it will be used the partitioning from CurioXXD/CurioXXcD.

The second step of the algorithm uses indexed counts on the map step to feed the Reduce function with extra information. Indexed counts are associations of a cell count with the cell index, to be implemented as an array or a concatenation of two strings.

As an example, a Map receiving a pair (i, count) will emit, for the neighbor cells, pairs with the indexed count: $\langle$neighborIndex, $\text{count}_{\text{index}}\rangle$. The information transmitted is: not only the neighbor partition has this neighbor with that count but also that its index is the one associated.

In CurioXRRD, the Map function will emit, for the case of the potential outlier cells, the received list of offsets and the indexed counts, with the counts of the offsets in the list, for all of its neighbor cells. The other cases, the non-potential outliers cells, in a much higher quantity, will emit just one single indexed count for each index. No nearest neighbors list needs to be created, in that case, and no more pairs are emitted.

Two reduces are then used in a pipeline, with the output of the first reduce feeding the input of the second one.

The first Reduce will emit directly the pairs with associations of index and lists of offsets, coming from potential outliers cells, previously filtered by the Map. Lists of indexed counts will constitute the other cases. The Reduce, search on those lists, for the entry with identical index of the index key, fix the associated count on that entry and re-emit all the other indexes in the list together with that count value. All the rest of count values on the index count list are dropped and will not be used.

To exemplify, consider the following example of a possible pair received by the first Reduce:

$$(\,0011, [7_{\,0110}\,; 1_{1000}\,; 3_{\,0101}\,; \mathbf{9_{\,0011}}]\,)$$

the indexed count with identical index is $9_{\,0011}$, hence the count will be fixed as 9, and the Reduce emits 3 pairs with that value, for the other indexes entries:

$$<0110, 9>, <1000, 9>, <0101, 9>.$$

The second Reduce receives, by index, the lists of offsets and the lists of counts, as input. It can then sum those counts and, if less then or equal than the Tolerance, use the offsets list to read and emit the instances, in an identical way as the previous optimized variants (like CurioXD and CurioXXD).

The CurioXRRD algorithm isn't of simple understanding nor easy data flow visualization, due to the non-sequential nature of the MapReduce processes and the chain of Map|Reduce+Reduce.

To make the algorithm clearer, a full example is detailed next. Consider the table 2 on previous example from 4.1, CurioD, and that the Tolerance was defined in this example as $T=3$:

0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
{;;;}	{;}	4		6		7	9		{;}		{;;;}	8		5

The CurioXRRD method exposed before will produce:

MAP

$$\begin{array}{rcl}
(\ 0001,\ \{\,;\,;\,;\}\ )^{*} & \rightarrow & \langle\, 0001, \{\,;\,;\,;\}\,\rangle \\
 & \rightarrow & \langle\, 0010, 3_{\,0001}\,\rangle \\
(\ 0010,\ \{\,;\}\ )^{*} & \rightarrow & \langle\, 0010, \{\,;\}\,\rangle \\
 & \rightarrow & \langle\, 0001, 1_{\,0010}\,\rangle \\
 & \rightarrow & \langle\, 0011, 1_{\,0010}\,\rangle \\
(\ 0011,\ 4\ ) & \rightarrow & \langle\, 0011, 4_{\,0011}\,\rangle^{**} \\
 & \cdots & \\
(\ 1100,\ \{\,;\,;\,;\}\ )^{*} & \rightarrow & \langle\, 1100, \{\,;\,;\,;\}\,\rangle \\
 & \rightarrow & \langle\, 1011, 3_{\,1100}\,\rangle \\
 & \rightarrow & \langle\, 1101, 3_{\,1100}\,\rangle \\
(\ 1101,\ 8\ ) & \rightarrow & \langle\, 1101, 8_{\,1101}\,\rangle \\
(\ 1111,\ 5\ ) & \rightarrow & \langle\, 1111, 5_{\,1111}\,\rangle
\end{array}$$

* for potential outliers cells the map receives and emits the set of instance offsets.
** for non-potential outlier cells emit only the indexed counts for each index.

REDUCE 1

$$\begin{array}{rcl}
(\ 0001,\ [\{\,;\,;\,;\},\ 1_{0010}]\ ) & \rightarrow & \langle\, 0001, \{\,;\,;\,;\}\,\rangle \\
 & \rightarrow & \langle\, 0010, 3\,\rangle \\
(\ 0010,\ [\{\,;\},3_{0001}]\ ) & \rightarrow & \langle\, 0010, \{\,;\}\,\rangle \\
 & \rightarrow & \langle\, 0001, 1\,\rangle \\
(\ 0011,\ [1_{0010}\,, 4_{0011}]\ ) & \rightarrow & \langle\, 0010, 4\rangle \\
(\ 0101,\ [\,5_{0101}]\ ) & - & \\
 & \cdots & \\
(\ 1100,\ [\{\,;\,;\,;\},2_{1101}]\ ) & \rightarrow & \langle\, 1100, \{\,;\,;\,;\}\,\rangle \\
 & \rightarrow & \langle\, 1101, 3\,\rangle \\
(\ 1101,\ [3_{1100}, 8_{1101}]\ ) & \rightarrow & \langle\, 1100, 8\,\rangle \\
(\ 1111,\ 5_{1111}\ ) & \rightarrow & —
\end{array}$$

REDUCE 2

$$\begin{array}{rcl}
(\ 0001,\ [\{\,;\,;\,;\},\ 1]\ ) & \rightarrow & \langle\,\{\,;\,;\,;\}\,\rangle \\
(\ 0010,\ [\{\,;\},\ 3,\ 4]\ ) & - & (\,3+4 \geqslant T\,) \\
(\ 0011,\ [1,\ 4]\ ) & - & (\textit{not outlier}) \\
 & \cdots & \\
(\ 1100,\ [\{\,;\,;\,;\},\ 8]\ ) & - & (\,8 > T\,) \\
(\ 1101,\ [8,\ 3]\ ) & - & (\textit{not outlier})
\end{array}$$

Example 4. The M|R+R on the second job of CurioXRRD.

The full CurioXRRD algorithm is resumed in the following listing, Algorithm 6.:

Algorithm 6. CurioXRRD Algorithm.

```
// Job1, Partitioning:

        ( identical to CurioXXD/CurioXXcD )


// Job2, Validation:
 // Map
  receive { (index, value) }
    if value isList    // Potential Outlier
      emit < index, value >
      indexedCount = {sizeOf(value), index}
      for each neighbor in neighborhoodOf(index)
        emit < index, indexedCount >
    else
      indexedCount = {value, index}
      emit < index, indexedCount >


 // Reduce1
  receive { (index, list[values]) }
  for each entry in list[values])
    if entry isListOfReferences
      emit < index, entry >
      cellCount = sizeOf(entry)
    else
      if indexOf(entry) == index
        cellCount = valueOf(entry)
      else
        indexArray.add(indexOf(entry))
  if cellCount > 0
    for each index in indexArray
      emit < index, cellCount >


 // Reduce2
  receive { (index, list[values]) }
  for each value in list[values])
    if value isListOfReferences
      Offsets = value
    else
      sumNeighbs = sumNeighbs + value
  if Offsets is not empty and sumNeighbs $\leqslant$ T
    for each offset in Offsets
      outlier = readDatasetAt(offset)
      emit < outlier >
```

4.4 After mapping data, a new approach, Curio3XD.

The above algorithms and variants are linear with size, $O(2N)$, and will exhibit linear behavior with the increase of instances in the dataset. But scalability with dimension, k, may impose a serious problem, where distributing computing may not be enough or the appropriate solution. If the number of attributes is high, and that is common in most realistic situations, the exponential behavior, $O(3^k)$, will be predominant over the linear part. Ceglar et al.[9] mentions $k = 10$ as a maximum for usability on their tests (a non-distributed application).

In this distributed adaption, preliminary experiments has shown values of the order of $k \cong 15$ as reasonable for use with the variants proposed above. Datasets of higher dimensions resulted on execution times too long and jeopardize the use of this approach. In case of Hadoop MapReduce framework, the problem is aggravated by the memory usage during the mapping tasks on the second jobs, before being saved on local disks, creating erratic issues with the JVM allocated memory, and requiring quite big memory allocation as safe margin to prevent possible overflow issues; eventually quantities not so easily available.

The problem resultant from the exponential dependency on dimension arises in the Map of the second job, from the emission of counts to all nearest neighbor's indexes, for each identified partition. On versions prior to CurioXRRD this emissions are done by all the c identified partitions, and generate an excessively large output, made mainly with superfluous partitions references, that later on the process will be discarded.

In the CurioXRRD variant, the p potential outliers partitions are the ones that exclusively emit counts for their nearest neighbors' indexes. But because it still needs no emit to all possible 3^k neighbor indexes, that results in just a minimization of the exponential growth constant, from $O(c3^k)$ to $O(p3^k)$, being in general p much smaller than c. Still, the dependency on k will pose problems to high-dimensional datasets.

One possible way to work around this problem is to consider the emission not to all neighbors' indexes of a given partition, but only to the indexes of partitions previously identified as not empty, the ones with allocated instances. In that way, for each partition to be validated, the procedure will require a check in the list of identified partition indexes, and verify which one corresponds to a neighbor. That's a quadratic operation, for each one of the c partitions there are a number of c verifications to be done, and the complexity will be $O(c^2)$.

A list of partitions with allocated instances is the final result of the first job, since the partitioning and counting of instances step only registers the indexes for non-empty partitions. But the MapReduce model doesn't allow easy recurrences, particularly inside a task of a job, either with the input on the Map phase or the list associated with the keys on the Reduce phase, any of them intended for a single reading. The proposed solution requires a second access to the partitions list or, more intricate, a second access to that list any time a partition is being validated.

This issue can be approached in two different ways, recurring to either memory or space and computational work. The Curio3XD and its variant Curio32XD implement, respectively, those two tactics.

The partitioning of dataset will not require any difference from the preceding versions. As first job, the same implementation of partitioning previously developed can be used, with or without the combiner.

On the second job, for the potential outliers validation, the Map will need, before any index being evaluated, to access the output of the first job, typically saved on the master node. All indexes will be read into a global array or any convenient data structure, available to all entries received by the Map. Note that the reader function on each mapper will read the indexes of all detected partitions by the first job, not only those assigned to that node by the data split.

For each index to be validated, the Map will don't have anymore to emit a full list of all possible nearest neighbors. The new index list of detected partitions will be checked and only existent neighbors will be emitted with the count of the cell. The check on that list employs a function to verify if the listed indexes are neighbors of the index received by the Map. That is a much simpler and lighter function than the one used for generate 3^k nearest neighbors. Moreover, the output of the Map will be significantly smaller, considering that for the most common cases, data is not uniformly distributed and a great quantity of partitions is expected not to have instances allocated, cells on the dataset space, and will not appear in the list.

Since essentially the Map just emits now a significantly smaller number of values for each index, avoiding superfluous, unneeded cases, but without introduce changes in the structure of the MapReduce implementation of the neighbors validation, the Reduce function will not have anything new, just a simple reuse of the Reduce function from CurioXXD/CurioXXcD. Obviously the same comment applies to the Combiner function.

The Reduce receive a list of values containing counts of instances in the neighbors cells and the list of offsets for the potential outliers partitions. If the list of offsets is present, then the sum of counts is performed and if less than or equal the Tolerance, the partition is classified as outlier and the instances referred by the offsets list are emitted directly by the Reduce as final output.

The data flow and algorithm scheme of Curio3XD are represented in figure 8 and in the following listing, respectively:

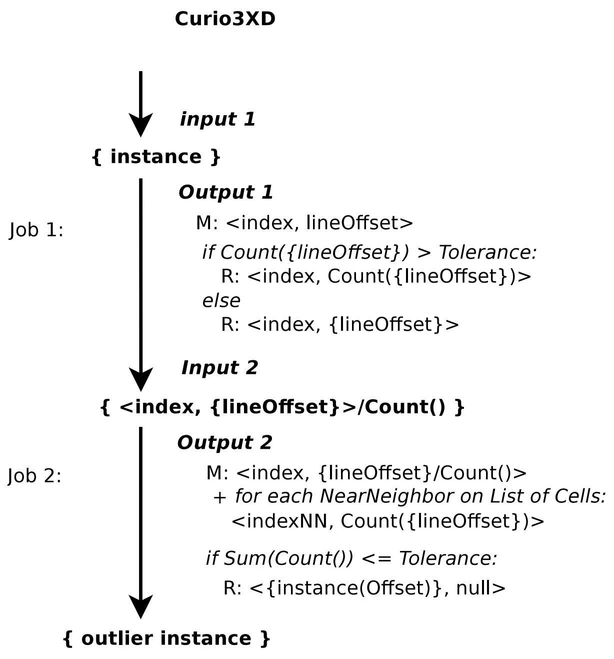

Curio3XD
input 1
{ instance }
Job 1:
Output 1
M: <index, lineOffset>
if Count({lineOffset}) > Tolerance:
R: <index, Count({lineOffset})>
else
R: <index, {lineOffset}>
Input 2
{ <index, {lineOffset}>/Count() }
Job 2:
Output 2
M: <index, {lineOffset}/Count()>
+ for each NearNeighbor on List of Cells:
<indexNN, Count({lineOffset})>
if Sum(Count()) <= Tolerance:
R: <{instance(Offset)}, null>
{ outlier instance }

Figure 8. Data flow of Curio3XD algorithm.

Algorithm 7. Curio3XD

```
// Job1, Partitioning:
 ( *identical to preceding variants* )


// Job2, Validating Potential Outliers:
 listOfIndexes = readCellsIndexes(pathToJob1Output)
 receive { (index, value) }
 // Map
   for each index in { (index, value) }
       emit <index, value>
       if value is aList
           counts = count(value)
       else
           counts = value
       for each cellIndex in listOfIndexes
           if cellIndex  in isNeighborOf(index)
               emit <cellIndex, counts>

 // Reduce
   receive { (index, {Offsets, list[counts]}) }
   for each index in { (index, {Offsets, list[counts]}) }
     if  $0 < count(\text{Offsets}) \leqslant \text{Tolerance}$
         sumNeighbs = sum(list[counts])
         if  $\text{sumNeighbs} \leqslant \text{Tolerance}$
             for each offset in split(Offsets)
                 outlier = readInstanceAt(offset)
                     emit < outlier >
```

Each mapper will receive a split of the output from the first job. The output will contain c entries of the detected partitions with allocated instances. For a defined number of M mappers, the split for each one will have c/M entries. And each mapper will read and keep in memory the list of the total c partitions detected. So, for each of the c/M indexes, a verification on the c partition indexes list is to be made. That represents a complexity of $O(c \times c/M) \sim O(c^2)$.

4.4.1 Curio32XD, Distributed versus On Memory.

On Curio32XD, the listing of the neighbors' partitions indexes for each detected partition is done in an extra MapReduce job, receiving the final job the list of the neighbors for each index. The neighbors index listing is done in distributed mode, but the output is saved in kept in disk, being delivered to the partitions in validation on the Map only a small list of its own neighbors' indexes.

The listing of neighbors' indexes is done for all cells at that stage. For each partition i identified as neighbor of another j, both informations, i in relation to j and j to i is saved, taking advantage of the symmetry of the neighborhood relationship. And since a partition is not neighbor of itself, not being considered in the process, the complexity is reduced from $O(c^2)$ to $O((c^2 - c)/2)$. This process will be described in detail next.

The new second job will now execute a listing of neighbors per partition identified on the partitioning step. The Map receives all the pairs with the indexes and the count of instances, and emits an irrelevant identical key, to aggregate the indexes in a same list, with the index as value: <"k", index>.

The Reduce will receive the list of all indexes under the key "k", and iterate on that list, validate each value if is neighbor of the previous ones received, and saving the received indexes and the temporary accepted neighbors on a data structure like a table or a map, according to:

```
$\mathrm{I}_0 \rightarrow$ table.$addKey(\mathrm{I}_0)$
**for each** $\mathrm{I}_i$ **in** {partitions indexes}
  **for each** $\mathrm{I}_j$ **in** table
    **if** $\mathrm{I}_i$ $isNeighborsOf(\mathrm{I}_j)$
      table.$add(\mathrm{I}_i, \mathrm{I}_j)$
      table.$add(\mathrm{I}_j, \mathrm{I}_i)$.
```

And then emits the validated neighbors indexes in the table, for each index.

The validation of potential outliers is now done on a third job, which takes the output of both job1 and job2 as input. The procedure is similar to the equivalent on Curio3XD, but the Map will also receive for each index, besides the count, the list with all indexes of its neighbors' partitions. The emission of the counts for the neighbors is done by direct access of the neighbors' indexes on that list.

The Curio32XD algorithm is resumed in the following listing:

Algorithm 8. Curio32XD

```
// Job1, Partitioning:
  ( identical to preceding variants )


// Job2, Listing Nearest Neighbors:
  receive { (index, value) }
  // Map
    for each index in { (index, value) }
        emit < "k", index >

  // Reduce
    receive { (index, {key, list[indexes]}) }
    for each index in list[indexes]
        tableNN.addKey(index)
        for each key in tableNN.keys
            if key isNeighbors(index)        // false if key = index
                tableNN.add(index, key)
                tableNN.add(key, index)      // by symmetry
    for each index in tableNN.keys
        emit < index, tableNN.Values(index) >


// Job3, Validating Potential Outliers:
  receive { (index, values) }
  // Map
    for each index in { (index, values) }
        emit <index, values >
        if value is aListOfReferences
            counts = count(values)
        else
            counts = value
        listOfIndexes = readNeighborIndexes(index)
        for each cellIndex in listOfIndexes
            emit <cellIndex, counts >

  // Reduce
    receive { (index, {Offsets, list[counts]}) }
    for each index in { (index, {Offsets, list[counts]}) }
      if $0 < count$(Offsets) $\leqslant$ Tolerance
          sumNeighbs = sum(list[counts])
          if sumNeighbs $\leqslant$ Tolerance
              for each offset in split(Offsets)
                  outlier = readInstanceAt(offset)
                      emit < outlier >
```

The data flow of the Curio32XD algorithm is represented in figure 9, bellow:

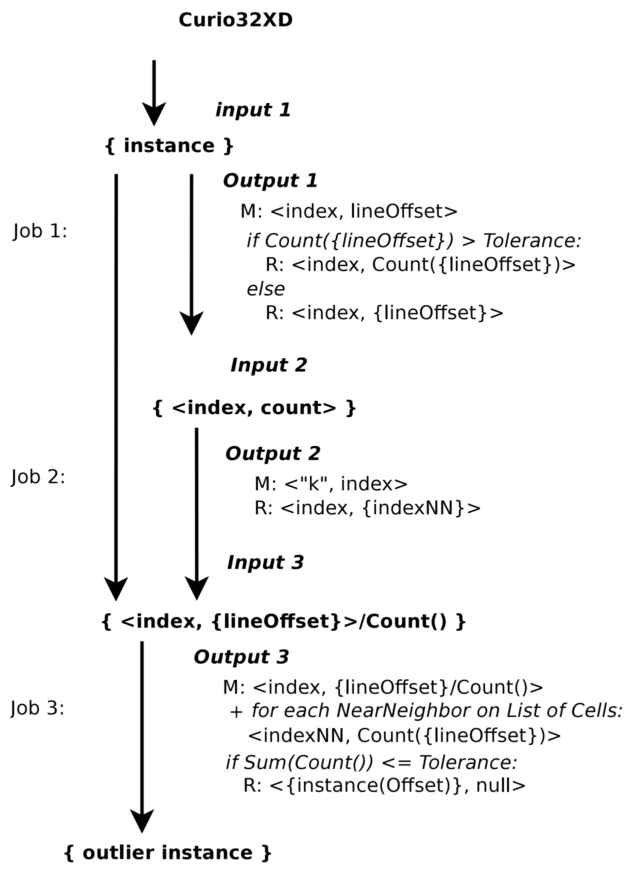

Curio32XD
input 1
{ instance }
Output 1
M: <index, lineOffset>
if Count({lineOffset}) > Tolerance:
R: <index, Count({lineOffset})>
else
R: <index, {lineOffset}>
Job 1:
Input 2
{ <index, count> }
Output 2
M: <"k", index>
R: <index, {indexNN}>
Job 2:
Input 3
{ <index, {lineOffset}>/Count() }
Output 3
M: <index, {lineOffset}/Count()>
+ for each NearNeighbor on List of Cells:
<indexNN, Count({lineOffset})>
if Sum(Count()) <= Tolerance:
R: <{instance(Offset)}, null>
Job 3:
{ outlier instance }

Figure 9. Data flow of the Curio32XD algorithm.

Due to the implementation of the listing of neighbors' indexes in a separated MapReduce job, this variant is expected to produce only better results than the previous variant for very large datasets and eventually the overload of the three jobs won't be compensated for smaller datasets.

An pratical example will generate identical outputs for both Curio3XD and Curio32XD. Using again the example processed on the anterior variants and keep the Tolerance with the previous value of $T = 3$,

0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
$\{;;;\}$	$\{;\}$	4		6		7	9		$\{;\}$		$\{;;;\}$	8		5

the second job of Curio3XD/Curio32XD would produce:

MAP

$$\begin{array}{rcl}
(\ 0001,\ \{\,;\,;\,;\,\}\ )^{*} & \rightarrow & \langle\,0001, \{\,;\,;\,;\,\}\,\rangle \\
 & \rightarrow & \langle\,0010, 3\,\rangle \\
(\ 0010,\ \{\,;\,\}\ )^{*} & \rightarrow & \langle\,0010, \{\,;\,\}\,\rangle \\
 & \rightarrow & \langle\,0001, 1\,\rangle \\
 & \rightarrow & \langle\,0011, 1\,\rangle \\
(\ 0011,\ 4\ ) & \rightarrow & \langle\,0011, 4\,\rangle \\
 & \rightarrow & \langle\,0010, 4\,\rangle^{**} \\
 & \ldots & \\
(\ 1101,\ 8\ ) & \rightarrow & \langle\,1101, 8\,\rangle \\
 & \rightarrow & \langle\,1110, 8\,\rangle \\
(\ 1111,\ 5\ ) & \rightarrow & \langle\,1111, 5\,\rangle^{***}
\end{array}$$

* for potential outliers cells the map receives and emits the set of instance offsets.
** emit only for the neighbor partitions that have instances allocated.
*** emit only the cell counts for instances without neighbors.

REDUCE

$$\begin{array}{rcll}
(\ 0001,\ [\{\,;\,;\,;\,\},\ 1]\ ) & \rightarrow & \langle\,\{\,;\,;\,;\,\}\,\rangle & \\
(\ 0010,\ [\{\,;\,\},\ 3,\ 4]\ ) & - & & (3+4>T \Rightarrow \textit{no emission}) \\
(\ 0011,\ [4,\ 1]\ ) & - & & (\textit{no list} \Rightarrow \textit{no emission}) \\
(\ 0101,\ [6]\ ) & - & & (\textit{no list} \Rightarrow \textit{no emission}) \\
 & \ldots & & \\
(\ 1010,\ [\{\,;\,\}]\ ) & \rightarrow & \langle\,\{\,;\,\}\,\rangle & \\
(\ 1100,\ [\{\,;\,;\,;\,\},\ 8]\ ) & - & & (\ 8>T \Rightarrow \textit{no emission}) \\
(\ 1101,\ [8,\ 3]\ ) & - & & (\textit{no list} \Rightarrow \textit{no emission}) \\
(\ 1111,\ [5]\ ) & - & & (\textit{no list} \Rightarrow \textit{no emission})
\end{array}$$

Example 5. Curio3XD/Curio32XD example.

It was presented, in this chapter, several versions and some variants for a distributed computing adaptation of the CURIO method, for outlier detection with large datasets.

CurioD, the first proposed versions is a direct implementation of CURIO algorithm in the MapReduce programing model. CurioXD and CurioXXD variant introduce several potential optimizations at expense of some computational and disk space resources. CurioXXcD propose yet another variant implementing combiners, an extension contemplated by the MapReduce model. CurioXRRD version tries to reduce the volume data saved in temporary outputs of the MapReduce jobs, minimizing I/O operations and optimizing some of the methods in terms of superfluous computations and related outcome. Finally, Curio3XD and Curio3XD propose the recourse to memory, saving already mapped information that tries to overcome the weakness of this approach when it comes to high-dimensionality datasets.

Appendix C contains the details of algorithms implementation, in terms of code, specific solutions for the most important functions and different aspects of the use of Hadoop's MapReduce implementation and libraries.

The Java code for the implementations of the algorithms in that framework is listed on Appendix D.

Comparative performance tests will serve to investigate the best options for a choice among the optimized variants, as also the behavior and scalability with the volume of data and the dimensionality of the datasets. The methodology for the tests executed, dataset and equipment used is presented in the next chapter, along with the results obtained.

5. Experimental Results

5.1 Debugging and Validation of Results

For debugging purposes and validation of results on the implementations of the distributed computing algorithms, it was used the following approach. The distributed versions were applied to small, well-known datasets, after those datasets had been previously analyzed by an implementation of the standard CURIO algorithm. It was considered an adequate validation criterion for the distributed versions code, the detection of the same number and the classification of the same instances as outliers for all distributed algorithms as by the original non-distributed algorithm.

The datasets used for this purpose were: Iris, wisconsin-breast-cancer and an altered version of this last one, with the addition of some random instances, carefully defined to be far distant from the pre-existents. Those datasets allow a quick detection and direct visualization of their outliers, which make them suitable to this type of testing.

Iris is known for hold a small number of instances hard to classify or associate in clusters. The tests executed attempt to locate and identify them, as they constitute, by definition, the outliers of that dataset. Tests were executed on distributed and non-distributed modes, for several combinations of the parameters Precision and Tolerance, with the results checked by visual inspection of 3-D visualizations of the dataset and detected outliers. The results were consistent between the distributed and non-distributed versions and visualization showed the detected outlier instances as the isolated and most distant from the dense groups. The two variants of the wisconsin-breast-cancer dataset, processed by both distributed and non-distributed versions, also achieved results consistent between the two modes, being the obvious atypical instances, artificially inserted on the rest of the data, detected identically by all algorithm variants.

Naturally, for real big datasets, with high dimensionality, such visualization is not simple or even possible, and the validation of the detected outliers would require more sophisticated methods. However such validation processes fall outside the scope of this thesis, and the above-described comparison was considered enough for the correctness of code.

Those were the initial tests, executed yet during the code development stage. Results are not presented, since they are essentially similarity comparisons of results, with little interest in terms of performance.

Being the purpose of this work to test the performance of the algorithms with big datasets, a direct comparison of a distributed and the non-distributed version naturally ends up, at certain volumes of data, to be unaffordable to the last one, in terms of execution times and memory management. For the final testing it was used, exclusively with the distributed computing versions, a larger dataset, the KDDCup 1999 Intrusion Detection Data, used by Ceglar at al.[9] in the original CURIO and other reference papers[3][34][28].

In order to obtain a reasonable and controllable volume of data, with different sizes and different number of dimensions, the original dataset was transformed by replication of data and by extraction of selected number of attributes. Its description and variations are described next, followed by a description of the cluster and software used, prior to the methodology employed in the final testing.

5.2 Dataset and its Variants.

The tests of the algorithms implemented in this work were made using the KDDCup 1999 Network Intrusion Detection DataSet [39], the dataset proposed for the third International Knowledge Discovery and Data Mining Tools Competition.

The 1998 DARPA Intrusion Detection Evaluation Program was conducted by the MIT Lincoln Labs, with the purpose of to survey and evaluate research in networks intrusion detection. A standard set of data to be audited, which included a wide variety of intrusions simulated in a military network environment, was provided.

Data were gathered from nine weeks of raw TCP dump data from a typical US air force LAN. During the use of the LAN, several attacks were performed on it. Per record, extra features were derived, based on domain knowledge about network attacks[39]. The 1999 KDD contest used a version of this dataset. The details or knowledge on the nature of the data are not, although, relevant for this work, only the size, in number of instances, and the number of attributes are relevant for the tests proposed.

The dataset presented to contest[39], contained in the repository file kddcup.data, have the following characteristics:

- 4898431 records/instances ($\approx$ *5 millions*).
- 41 attributes + 1 label *(unused in tests)*.
- 743 Mb size.

An application specifically conceived for that purpose, GetColumns (see Appendix D, page 181 for details), was used to extract a given number of attributes from a dataset, creating new sets of data with the pretended number of dimensions. Eight distinct datasets, with dimensionality of 5 to 40 attributes, in steps of 5 attributes, were generated and used in several of these tests. The same number of instances, of approximated $5{\times}10^6$, was kept across all those variants of the dataset.

In order to investigate the situation of a growing volume of data, tests were made using with the 15 attributes dataset file replicated several times, as need. The choice for the 15D dataset file was made after some preliminary tests, where was concluded being that size the reasonable value for the dimension in terms of run times for all the algorithms. Higher dimensions created both huge execution times for some of the algorithms and a requirement of memory beyond the capacity available on the hardware used.

Table 4 presents the relation between the number of the 15 attributes data file replicas, the size of data in gigabytes, and the :

Number of 15D file replicated	**1**	**3**	**6**	**9**	**12**
Total size of data (Gb.)	0.5	1.5	3	4.5	6
Total number of instances (approx.)	5×10^6	$15{\times}10^6$	$30{\times}10^6$	$45{\times}10^6$	$60{\times}10^6$

Table 4. Relation between number of files replicated, number of instances and total size of data.

The resultant files containing all variants of the dataset were copied, for the final tests, into the HDFS filesystem of a Hadoop installation on the equipment described next.

5.3 Equipment and Software Used.

The debugging phase and preliminary tests were made using Amazon Elastic Compute Cloud, known as EC2, running customized Amazon Machine Images (AMI), with a pristine Apache Hadoop installation, version 0.20.2, over a Cloudera Linux, CDH3[14]. The details of the AMI creation and use under Amazon EC2 are given in the Appendix B.

For the final tests, the distributed system was implemented in the cluster of the Department of Electrical and Computer Engineering at University of Stavanger. 11 machines, with 6-core processors AMD Phenom II X6 1090T, and 8 Gb of memory RAM, running Ubuntu Server Linux as operating system, constitute that cluster.

The Hadoop installation was also a 0.20.2 version, obtained directly from Apache's site, and following instructions from M. Noll [33]. A detailed description of the Hadoop installation and configuration is given in Appendix A. Customized configuration properties are resumed on table 14, on page 99, of Appendix A.

During the testing period the cluster was always used exclusively running the tests, with only a minimal number of operating system processes on background.

In all situations the Java version installed was Java 6.

5.4 Objectives and Methodology.

With the proposed algorithms implemented in the Hadoop's framework (for details see Appendix C), it is of major interest to investigate the following main points:

- How the number of nodes in a cluster affects performance, and how the different algorithms scale with that number?
- How the different algorithms scale with the volume of data?
- How well distribute computing handles the scaling of execution times with the dimensionality of data and which of the proposed versions remain applicable to high dimensions?
- How the algorithms dependent on the number of partitions detected behave with changes on the precision parameter, that defines the partitions properties and consequently they number?

For this purpose, each of the above questions requires a specific testing environment where the different parameters are held fixed and the variable on study may vary in a controlled and consistent way. This was achieved with the dedicated cluster descibed above, running a Hadoop platform exclusively for the tests execution.

The main variables of interest to investigate in the performance tests and the values used were:

• number of machines on distributed computing (size of cluster):	1, 3, 5, 7, 9, 11
• size of the dataset (volume of data, in millions of instances):	5, 15, 30, 45, 60
• dimension of the dataset (number of attributes):	5, 10, 15, 20, 25, 30, 35, 40
• precision parameter (size and density of partitions):	3, 4, 5, 6, 7, 8, 9

Four different series of tests were made, one for each of those variables, keeping in each test all the other parameters constant. The results are presented in the following sections of this chapter.

5.5 Results

5.5.1 Tests with the Number of Nodes.

For the investigation of the algorithms' scalability with the cluster size, ie the number of constituting nodes, the algorithms were executed over a relatively large dataset variant, with a dimension of **15 attributes**, replicated 3 times, in a total of **15 millions of instances** (1.5Gb of data).

Testes were performed in:

- Single-node configuration on one of machines of the described cluster.
- Multi-node configurations with the cluster configured successively for 3, 5, 7, 9 and 11 nodes.

Each algorithm was executed 3 times (total execution times were very close, with variations under the 2% of total times) and the average time is showed in table 5. Runtimes were taken from the output of the implemented code, specifically written to produce the result discriminated per job executed. Times, in seconds, are specified for the two tasks of the CURIO procedure, partitioning of the data space and validation of potential outliers.

The parameters precision and tolerance were kept fixed for all configuration as:

$$P = 8 \ \text{ and } \ T = 50.$$

	nodes:	**1**	**3**	**5**	**7**	**9**	**11**
CurioD	*part.*	190.9	179.4	172.5	175.6	173.6	169.2
	valid.	13063.8	4850.3	3446.4	2477.3	3176.2	1837.2
CurioXD	*part.*	200.6	188.4	190.4	179.8	182.8	180.8
	valid.	13130.1	5238.5	3594.2	2627.0	2787.1	1891.6
CurioXXD	*part.*	194.8	179.3	177.4	174.8	171.1	172.5
	valid.	12850.1	4618.1	3088.9	2283.5	2978.8	1569.8
CurioXXcD	*part.*	162.0	149.5	153.6	146.6	144.3	143.0
	valid.	12850.1	4618.1	3088.9	2283.5	2978.8	1569.8
CurioXRRD	*part.*	161.8	145.5	149.6	142.4	145.0	145.3
	valid.	12841.7	4621.4	3252.3	2258.5	2995.5	1631.5
Curio3XD	*part.*	159.0	153.4	151.0	147.1	146.8	146.0
	valid.	30.4	30.4	30.4	30.5	30.6	30.6
Curio32XD	*part.*	161.3	148.9	147.6	144.7	147.4	147.3
	valid.	57.5	57.9	58.0	58.1	59.3	58.6

Table 5. Average run times (sec.) by task, partitioning (white) and validation (grey), with the number of nodes.

Note that CurioXXcD, CurioXRRD, Curio3XD and Curio32XD variants all use the same implementation of the partitioner for the first job.

The average times for the total run times of all jobs, in seconds, for each algorithm is resumed on table 6:

nodes:	**1**	**3**	**5**	**7**	**9**	**11**
CurioD	13254.7	5029.7	3618.8	2652.9	3349.8	2006.4
CurioXD	13330.7	5426.9	3784.6	2806.8	2983.7	2072.4
CurioXXD	13044.9	4797.4	3266.3	2458.2	3149.9	1742.3
CurioXXcD	13003.5	4766.9	3401.9	2400.8	3140.5	1776.8
CurioXRRD	7345.5	2715.1	2144.0	1711.9	1852.9	1478.2
Curio3XD	189.3	183.7	181.4	177.6	177.4	176.6
Curio32XD	218.7	206.8	205.6	202.8	206.7	206.0

Table 6. Average total run times (sec.) of the algorithms with the number of nodes.

The value of variance on tests was always under 2%, in most cases inferior to 1%, without much fluctuations and so individual error margins are not listed.

In total four different partitioners were implemented. A base version included in CurioD, an adapted version for CurioXD called PartitionerD, and an improved third one, called PartitionerXD, with the possibility of running with or without a MapReduce combiner. The algorithm CurioXXD was tested with this late version without the combiner, and all the others, CurioXXcD, CurioXXRD and the two Curio3*XD, used the version with combiner.

The results on the table 7 are the time averages, in seconds, of the partitioner only run times:

nodes:	**1**	**3**	**5**	**7**	**9**	**11**
CurioD Partitioner	190.9	179.4	172.5	175.6	173.6	169.2
PartitionerD (CurioXD)	200.6	188.4	190.4	179.8	182.8	180.8
PartitionerXD no combiner	194.8	179.3	177.4	174.8	171.1	172.5
PartitionerXD with combiner	161.0	149.3	150.4	145.2	145.9	145.4

Table 7. Average run times (sec.) of first task, the partitioning, with the number of nodes.

5.5.2 Tests with the Size of Dataset.

For the scalability tests with the volume of data, the same dataset with **15 dimensions** was used. In order to reproduce a growing volume of data, that dataset was replicated in successive runs to all algorithms, a first series without a replication, fallowed by replications of 3, 6, 9 and 12 times the data file, corresponding to, respectively:

- 5, 15, 30, 45 and 60 millions of instances.

Table 8 presents the average execution times, in seconds, for the two stages of all algorithms implemented. Tests were done in a cluster with a fixed number of **9 nodes**.

Precision was set to $P = 8$ and the tolerance value was $T = 50$.

	Size: #instances (millions):	**5**	**15**	**30**	**45**	**60**
CurioD	*part.*	155.0	175.6	197.7	237.4	282.6
	valid.	3115.3	3182.8	3162.8	3215.4	3237.9
CurioXD	*part.*	158.8	201.1	306.1	342.5	410.9
	valid.	2824.6	2799.6	2449.4	2509.5	2469.4
CurioXXD	*part.*	153.3	171.9	202.6	232.8	287.9
	valid.	2927.2	2974.7	2927.5	2912.2	2922.1
CurioXXcD	*part.*	140.7	144.8	152.4	149.4	158.8
	valid.	2979.7	3009.7	2992.2	2967.2	2947.2
CurioXRRD	*part.*	148.8	143.8	147.8	151.8	160.4
	valid.	2172.2	1712.0	1114.2	976.6	991.6
Curio3XD	*part.*	136.2	148.5	156.0	152.1	159.8
	valid.	30.5	30.6	30.6	30.6	30.6
Curio32XD	*part.*	135.4	147.4	146.9	147.0	156.3
	valid.	59.2	59.3	58.7	58.8	58.2

Table 8. Average run times (sec.) by task, partitioning (white) and validation (grey), with size of dataset.

The average total execution times of both tasks, in seconds, for each algorithm are shown in table 9:

Size (Gb.):	**0.5**	**1.5**	**3**	**4.5**	**6**
CurioD	3270.3	3358.4	3360.5	3452.8	3520.5
CurioXD	2983.4	3000.7	2755.5	2852.0	2880.3
CurioXXD	3080.5	3146.6	3130.1	3145.0	3210.0
CurioXXcD	3120.4	3154.5	3144.6	3116.6	3106.0
CurioXRRD	2321.0	1855.8	1262.0	1128.4	1152.0
Curio3XD	166.7	179.1	186.6	182.7	190.4
Curio32XD	194.6	206.7	205.5	205.8	214.5

Table 9. Average total execution times (sec.) with size of dataset.

Table 10 lists the average run times for each partitioner implemented, with the growing size of the dataset:

size(Gb.):	**0.5**	**1.5**	**3**	**4.5**	**6**
CurioD Partitioner	155.0	175.6	197.7	237.4	282.6
PartitionerD (CurioXD)	158.8	201.1	306.1	342.5	410.9
PartitionerXD no combiner	153.3	171.9	202.6	232.8	287.9
PartitionerXD with combiner	140.3	146.1	150.8	150.1	158.8

Table 10. Average run times (sec.) of algorithms first task, partitioning, with the number of nodes.

Again, the PartionerXD without combiner was used on CurioXXD and with combiner on CurioXXcD, CurioXRRD, Curio32XD and Curio32XD.

5.5.3 Tests with the Dimension of Dataset.

Tests with dimensionality where only executed for the Curio3XD and Curio32XD algorithms, the ones that shown to be able to deal, without problems, with high dimension datasets. For the other algorithms, the limit of 15 attributes as a reasonable dimension (and even in that case with considerable high execution time), didn't show enough interest to implement a series of tests exploring lower dimensions, already known to have exponential dependence on that variable.

Add or remove dimensions from a dataset naturally implies, as a change on the structure of the data, different potential outlier cells and outlier instances. That results on explicit differences on execution times not directly related with the change of dimension but due to the different number of potential outlier cells and nearest neighbors to be verified. The order of time variance of those changes however is not significant compared to the one introduced by the growth of dimension, but in order to minimize that effect, dimensions has been varied by relatively large steps, 5 dimension per test:

- 5, 10, 15, 20, 25, 30, 35 and 40 attributes,

all containing 30 millions of instances (resultant from the replication of the original data files).

Tests were made on a **9 nodes** cluster; Each run was repeated 10 times to more reliable averages and the estimated error indicated. The results are shown in table 11, average times are in seconds.

Algorithm parameters were kept, as previously, $P=8$ and $T=2000$:

attributes:	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**
Curio3XD	109.0	147.6	180.3	179.2	158.4	151.5	142.5	143.1
std.error	4.3	7.2	7.0	12.1	9.1	10.8	7.3	3.6
Curio32XD	136.8	175.5	212.1	210.1	185.5	178.1	171.3	171.1
std.error	6.0	8.0	12.1	12.1	6.0	5.0	9.1	4.2

Table 11. Average run times (sec.) with increasing dimensions (#attributes) of dataset.

5.5.4 Tests with the Precision Parameter.

The average run times for different values of the parameter P, the precision, ranging from 3 to 9, for the Curio3XD and its variant Curio32XD, are presented in the table 12. The times, in seconds, are averages of 10 runs for each algorithm, on a cluster with **9 nodes**.

The dataset used was the full KDD dataset, with **40 dimensions**, repeated 6 times to increase the data size to 3Gb, in a total of **30 millions of instances**.

In order to generate usefull results across all values of precision, having always some outlier cells and without falling in the opposite situation where their number would create memory problems during the map phase, the value of tolerance had to be raisen for this specific test.

Tests were done with two different values of tolerance, $T = 2000$ and $T = 5000$.

	precision P:	**3**	**4**	**5**	**6**	**7**	**8**	**9**
T=2000	Curio3XD	143.5	146.3	147.1	145.0	148.8	156.4	238.4
	std.error	2.0	3.5	1.4	4.5	5.4	2.7	21.8
	Curio32XD	170.7	172.6	180.8	171.7	178.1	184.2	258.7
	std.error	2.6	3.5	23.5	1.1	3.4	3.0	20.3
T=5000	Curio3XD	144.5	143.4	151.7	147.0	151.2	176.2	235.8
	std.error	1.4	2.2	14.7	2.7	3.8	17.5	18.9
	Curio32XD	168.8	182.0	171.5	175.0	191.0	204.8	272.9
	std.error	2.8	22.8	2.2	2.0	32.1	30.9	31.2

Table 12. Average run times (sec.) with parameter P, precision, for two values of tolerance, T.

These results will be analyzed and discussed in the next chapter.

6. Discussion

The tests executed for this work intended to investigate the behavior of the algorithms and variants proposed with several factors, namely the size of dataset, the dimensionality or number of data attributes and the CURIO parameters, precision and tolerance. The results were collected after several runs, on several configurations of clusters described in 5.3.

The orientation for this investigation followed three main lines:

- The comportment of the algorithms in clusters of different sizes, from the single node case to multi-node scenarios with several machines.

- The performance in speed and scalability with the volume of data in a growing size dataset.

- The performance in speed and scalability with the dimensionality of the dataset and effects of the defined precision.

In the absence of a definitive methodology to choose good values for the CURIO parameters, the values used were the ones obtained, after some trial and error search, during the preliminary tests with the following basic heuristics.

Precision will depend on the scale of the data space, limited by the minimum and maximum value of value son dataset. For the purpose of this work, and assuming that those values may not be known a priori by the data analyst, a distributed application was developed, also using Hadoop and the MapReduce model (see Appendix D), to efficiently determine such values even for big datasets.

The maximum and minimum of the datasets were previously determined and the values found used in every test, in order to avoid unnecessary calculations on each run. The maximum, identical to all variants of the dataset, permitted precisions values up to 16. In this tests, with the exception of the last series where the precision was the variable considered, was always used the value $P = 8$.

The tolerance basically defines the density of potential outliers cells and can be set as a fraction of total data volume, by some estimation of the percentage of outliers expected or the desired outlier density to be filtered. The value used for most tests was $T = 50$, correspondent to 0.001% of the total of instances in the dataset without replications. For the study of the variation with the precision was used the value of $T = 2000$ correspondent to 0.004% of the instances number and in the last test was also considered a tolerance of 5000 (0.01% of the instances number), for comparison with the previous value.

6.1 Performance with the Size of the Cluster.

The first point to investigate was the performance of the several algorithms implemented with a growing number of nodes in the cluster. Primarily we wanted to verify if the reduction in execution times resultant from the distributed computing is always improved and, at least, at an approximated rate, implying otherwise the existence of an optimal number of nodes to execute these specific algorithms, as pointed by Esteves, Pais and Rong[20].

A first observation is the complete different order of magnitude of the Curio3XD and Curio32XD algorithms, generically 20 times faster than the average of the others, requiring different scales to a meaningful representation, as can be seen on figures 10 and 11, with the average execution times for the several number of nodes tested (see table 6 on section 5).

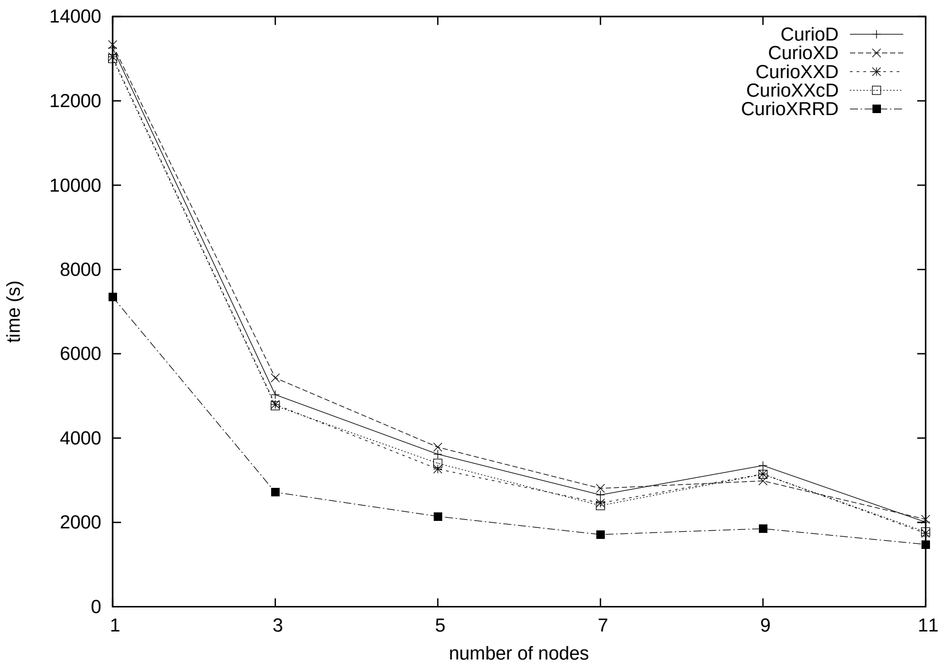


Figure 10. Average execution times with the number of nodes.

The previous time scale wouldn't allow to see details on the two Curio3XD variants, due to their comparatively fast run times. Figure 11, next, shows the average execution times of those two algorithms:

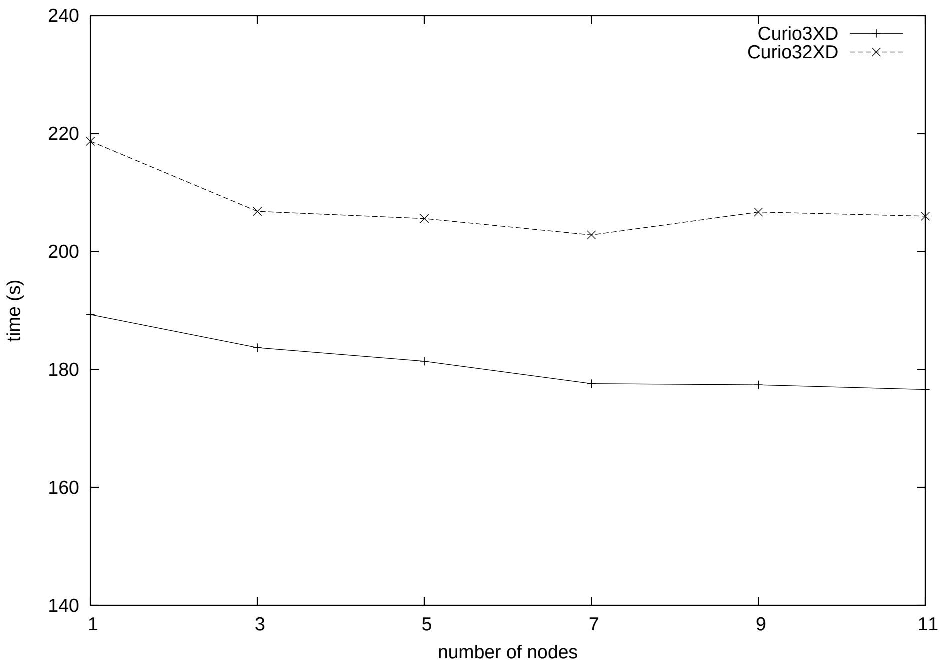


Figure 11. Average excution times (sec.) with the number of nodes, rescaled for Curio3XD and Curio32XD.

The error estimator for the run times averages was, to all cases, under 2%, and the error bars were not represented to avoid making the graphics less legible with virtually no extra information exposed. It was used the dataset with a dimensionality of 15 attributes, with 15 millions of instances, and the parameters were set as precision $P=8$ and tolerance $T=50$.

The execution times diminish fast with the raise of the nodes holding the distributed computation, but there is a visible asymptotical value, showing insignificant improvement after a certain number of machines, 7 for this case (see table 13 bellow, for the gains with the number of nodes). That repeats the results of Esteves, Pais and Rong, obtained with another class of algorithms[20], and justify the use of a cluster with a limited number of machines, in place of the EC2 cloud used on the preliminary tests. The optimal minimum number of nodes dedicated to a given task depend on the volume and nature of the data, and will compel, if such is a constrain in the design of a dedicated cluster, to some testing previously to any real data analysis.

nodes:	**3**	**5**	**7**	**9**	**11**
CurioD	0.62	0.73	0.80	0.75	0.85
CurioXD	0.59	0.72	0.79	0.78	0.84
CurioXXD	0.63	0.75	0.81	0.76	0.87
CurioXXcD	0.63	0.74	0.82	0.76	0.86
CurioXRRD	0.63	0.71	0.77	0.75	0.80
Curio3XD	0.03	0.04	0.06	0.06	0.07
Curio32XD	0.05	0.06	0.07	0.06	0.06

Table 13. Performance gains (%) relative to the Single-Node case.

It's interesting to note that without the later optimizations introduced to the CurioXD, which led to CurioXXD/CurioXXcD variants, this algorithm performs, in this test, worst than the original CurioD, result attributable to the bigger size output of the data mapping generated during the first step, the data partitioning, of the algorithm, carried all through the MapReduce process of being retrieved and processed across the cluster.

The results for the CurioD, CurioXXD and CurioXXcD algorithms in the 9 nodes case are somewhat irregular and unexpected. It is hard to justify such fluctuations relatively to the 7 and 11 nodes cases, possibly due to interferences from unforeseen background processes, although the tests' long times don't seems to sustain this hypothesis, or some undetected problem related with one of the nodes used for the 9 nodes test, either machine or network related. The general conclusion is not, although, altered, with even the unexpected higher times still exhibiting the steady tendency towards the asymptotic number of nodes with insignificant improvement in terms of performance.

The second general result, from this first series of tests, relates directly to the general excellent performance of the Curio3XD and Curio32XD algorithms to all cases of nodes configurations. The average times of around 20 times faster made them the indisputable winners of this test, at least for the dataset tested.

Nevertheless, situations may arise that can compromise the choice of those algorithms, dependent on the number of potential outliers cells and consequently on the data distribution, as was exposed in the section 3, Theory. A very sparse data distribution may create an excessively high number of potential outlier cells, to be kept in memory during the MapReduce process. Their reliance on the existent memory may eventually lead to problematic situations for some volumes of temporary data. For that reason, some of the other options still constitute interesting possibilities, in particular the CurioXRRD algorithm which indicates a superior performance, with almost half of the average times of the CurioX*D series and CurioD algorithm.

Although the predominant step of the algorithm is, in terms of computational cost, the second one, the nearest neighbors' validation of the potential outliers, is worthwhile to verify separately the response of the partitioners implemented for the first step. The execution times of that phase resulted to be of the same order as the second step for Curio3XD and Curio32XD, besides it will also be of interest to investigate the impact of the combiners' use that the PartitionerXD can resort and was employed to all variants after CurioXXcD.

The visualization of the average execution times for the partitioners only, for the four different versions implemented, with a growing number of distributed nodes on the cluster are shown in figure 12:

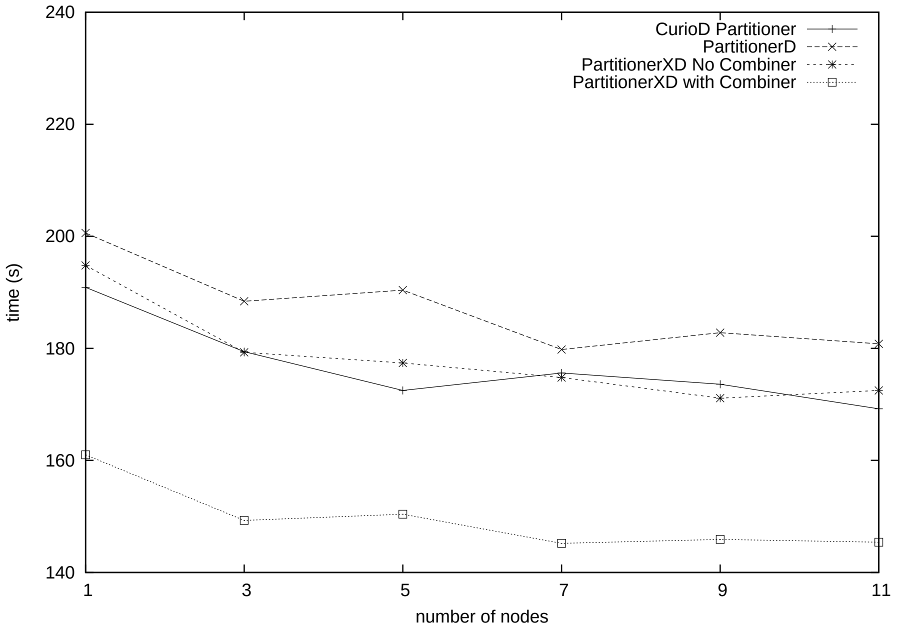


Figure 12. Average times (sec.) for the 4 partitioners implemented.

The worst result, as can been seen, was the expected PartitionerD, the partitioner for CurioXD, consequence of the bigger volume of data produced by the mapping, as it was mentioned before. The improvement introduced in PartitionerXD, without a combiner, brings the performance close to the original PartitionerD of CurioD. With the combiner the runtimes are clearly reduced, indicating the use of combiners as a good practice for the methods implemented.

In next section is represented and discussed the test results for increased volumes of data with a fixed number of nodes, to study the scalability with the datasets size, along with the discussion of the results of the average execution times for all algorithms.

6.2 Performance with the Volume of Data.

For the tests with variation of the data size, the volume of the original data was not considered big enough to allow the extraction of still meaningful smaller subsets, and for that motive it was opted instead to artificially increase the existent dataset. In order to obtain a considerable larger volume of data in a way still controllable, it was enlarged by simple replication of the dataset, in several multiples of the original file's size. The file chosen as base for replication, was one of the new datasets, with 15 dimensions, that still holding the same number of instances, allowed practical execution times and no memory problems for the resources available on the cluster.

That solution although extending the number of instances, from 5 up to 60 millions, naturally under the same data model, consists in practice in a repetition of occurrences on the same partitions, without allocations of instances to new partitions. In other words, the number of detected partitions is always the same to all of these artificially increased datasets. Only the number of potential outlier partitions will eventually change, if the replication of instances on those partitions raises their number above the defined tolerance value.

The test results, for parameters $T = 50$ and $P = 8$ in a cluster with 9 nodes, for the implemented data partitioners are represented in figure 13, bellow:

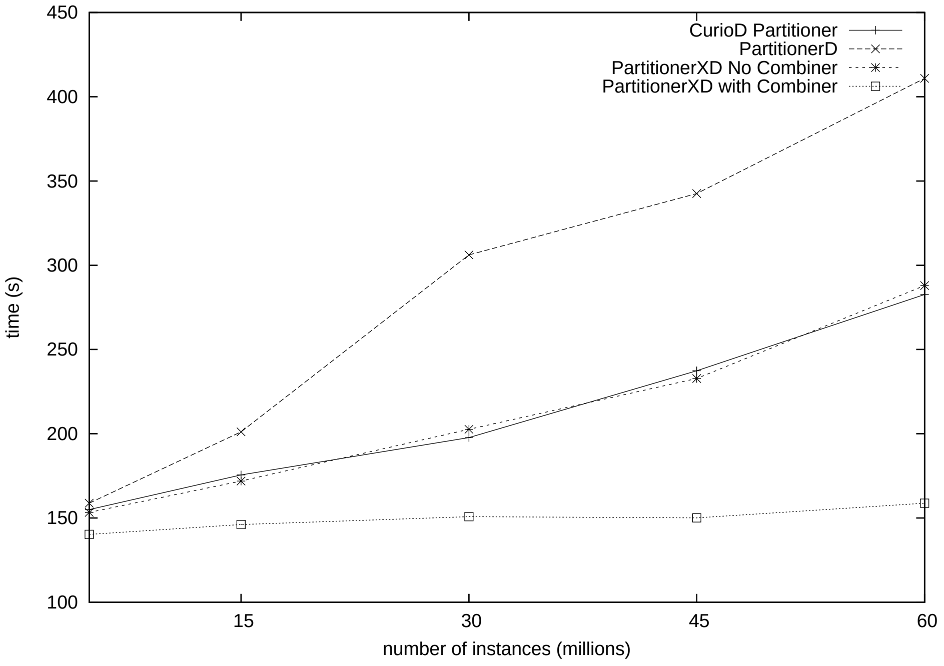


Figure 13. Average execution times (sec.) with size of dataset for the 4 partitioners implemented.

Again we can verify, from the figure 13, the best average runtimes for the data space partitioning are the ones of PartitionerXD with combiner. The expected linearity with N, the number of instances, is verified, but for that partitioner the rate of time increasing with the quantity of data processing is practically null, establishing that partitioner as best choice for the algorithms' first step, both in terms of performance and scalability with size.

The optimal results in terms of size scalability are mainly due to the use of combiners. Those execute partial sums on the counts of instances from partitions immediately classified as not potential outliers, in operations executed simultaneously at each node running the mappers. Besides the time optimization of that distributed sum, the outputted produced is considerable small and will require less time and network traffic while retrieved by the reducers.

As it was referred, PartitionerXD with the combiner was the data space partitioner chosen in all variants ahead of CurioXXcD, inclusively.

Figures 14 and 15 show the averages runtime evolution of all algorithms implemented with a growing volume of data (with 15D dimension) on a fixed number of 9 nodes:

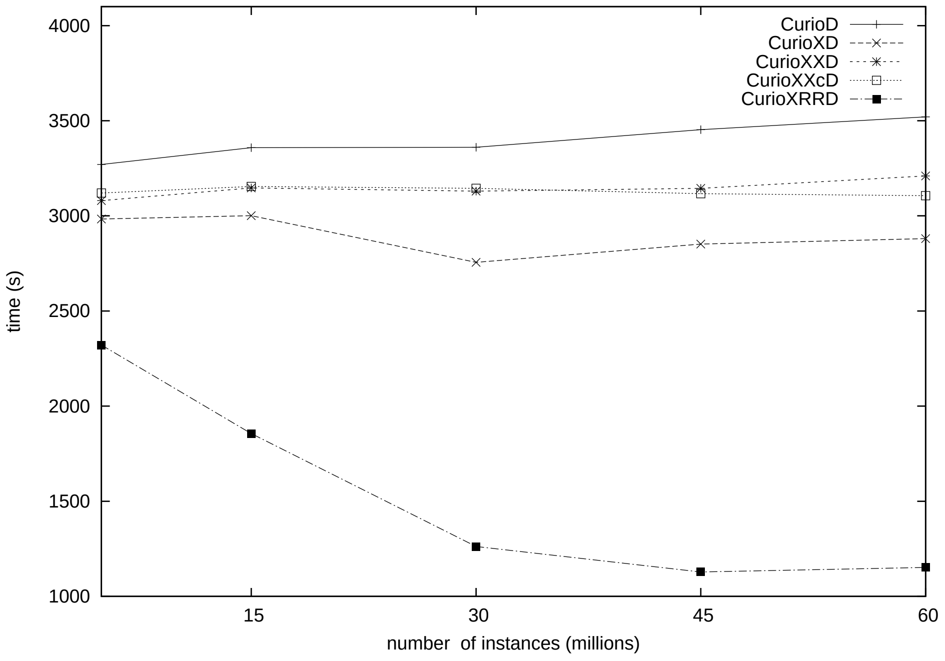


Figure 14. Average execution times (sec.) with size/number of instances of the dataset.

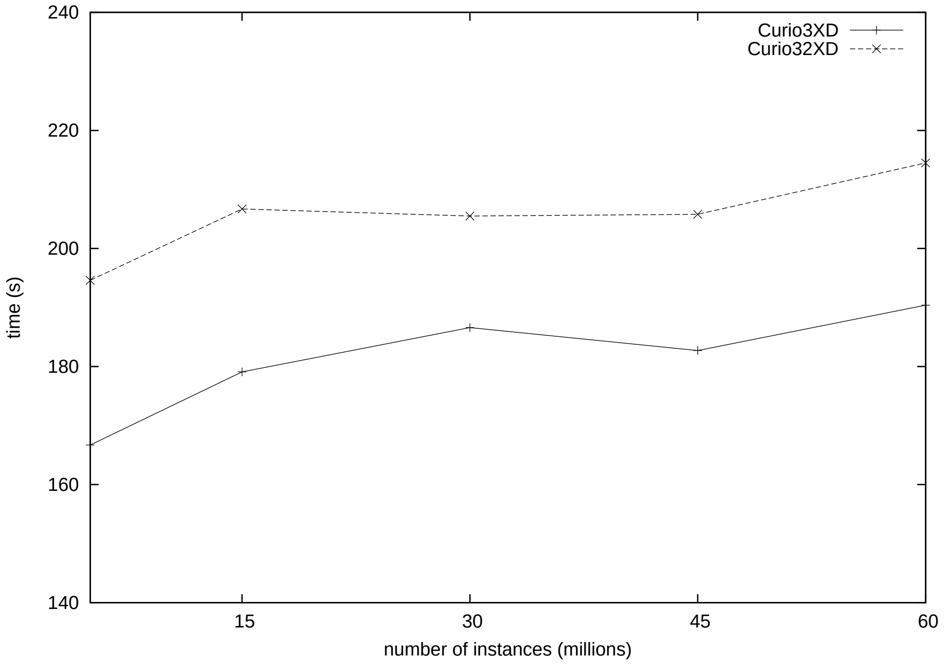


Figure 15. Average execution times (sec.) with size of dataset for the Curio3*XD variants.

In the first group (fig. 14), is the exponentially dependency on dimension that mainly dominates the behavior of the algorithms. The dependency on N, the number of instances, is $O(2N)$ for CurioD and $O(N+n) \sim O(N)$ for the others with a single reading executed on the first stage of the algorithm (and studied on previous fig. 13). The time scale for that first stage is much smaller than the one imposed by the processing of the 15 dimension of the dataset samples on the second stage, so the results for the full execution times of those algorithms should reflect little impact with the growing size of data, which is verified for CurioD, CurioXD and its optimized variants.

Generally the CurioD version shows a slight worst result comparatively to the others, due to the double reading of the datasets and the extra MapReduce job, but this extra efforts are smoothed by the distributed computation processes across the cores of the cluster and by the optimizations from Hadoop implementation which tries no minimize any idle times between processes and full usage of processors in all the machines.

The most notable result, for that group, is the CurioXRRD algorithm with runtimes reductions of close to 30% while raising the number of instances until 30 millions, 6 times more than the first case with 5 millions, to stabilize around the 1200 seconds until at least 60 millions of instances, that is, practically one third of the execution times of the others algorithms. That seems to indicate an excellent scalability of this particular algorithm with the size of the dataset.

Part of this result can be attributed to the dependency of this algorithm in the number of potential outlier cells, $O(p3^k)$, that in this case tends to diminish due to the process of artificially increase the number of instances. By merely duplicate instances, no new partitions are allocated and the existent ones became more density populated, eventually overpassing the limit defined for the tolerance. A smaller number of potential outlier partitions imply smaller execution times. This type of optimization, as the tests seems to point, may be costly for relatively smaller data volumes but becoming effective for larger datasets with data distributed by high-density regions and few sparse regions, or with a known low number of outliers.

Although the total averages times for the Curio3XD and Curio32XD are considerable smaller, around the 200 seconds, and the general performance and scalability are excellent (fig. 15), the above result for the CurioXRRD is still potential interesting. Curio3XD and its variant are dependent quadratically on the number of partitions detected, the non-empty partitions, and directly on the available physical memory. For the situations where such characteristic can impose a problem, like datasets with a very disperse data and consequently a high number of occupied partitions, CurioXRRD may constitute a viable alternative, only limited for high dimension datasets.

It may be seen from this test results that, in terms of datasets size, only the data partitioning stage is effectively affected. The validation of potential outlier cells by the nearest neighbors instances counting is in the origin of the high average execution times that are, although, fairly constant with variations in the data volume.

Therefore, it can be expected that any volume of data can be processed in an optimal time with the distributed computing, given a enough large number of nodes in a cluster, being successively handled both the linear growth of computational cost of the data partitioning and the time base, non-size dependent, required by the nearest neighbors validation.

The situation is not, nevertheless, generically applicable for datasets with higher dimensionality. For such cases, only the Curio3XD and Curio32XD variant in practice could deal with the times taken while validating the potential outliers. Even the reasonable well scored CurioXRRD produced extremely high times for any number of dimensions bigger than the ones tested so far.

The datasets with dimensions higher than 15, gave results, during the preliminary tests, of extremely lengthy runtimes, and frequent "Heap out of space" exceptions, that required constant raises on the memory allocated by the Hadoop's configuration to the Java Virtual Machine. Due to the extremely fast and consistent results obtained with those two last algorithms, no more tests were executed for the precedent variants, impracticable in practice for the cases of dimensions above the 20 attributes.

6.3 Performance with the Dimensionality of the Dataset.

For the study of the scalability with the number of dimensions, the original dataset had to be transformed in order to obtain a several variants with different number of dimensions. Several scaled down variants were generated, with 5 to 40 attributes, with increments of 5 attributes.

Obviously these new datasets had completely different structures, among themselves and in relation to the original dataset, since they were built from partial subspaces of the full dimensional data space. Neither the data structure nor the number of partitions can be preserved across the sectioning of the data space. In consequence, some irregularity is expected in the tests results, since the number of partitions with instances allocated can change in a way absolutely non-correlated from set to set, and the proposed algorithms depends directly on the number of partitions detected.

Figure 16 shows the results for the Curio3XD and its variant Curio32XD applied to that datasets, all of them with an approximated number of 15 millions of instances. The parameters used were a precision $P = 8$ and a tolerance of $T = 50$. Tests were performed in a cluster with a fixed number of 9 nodes.

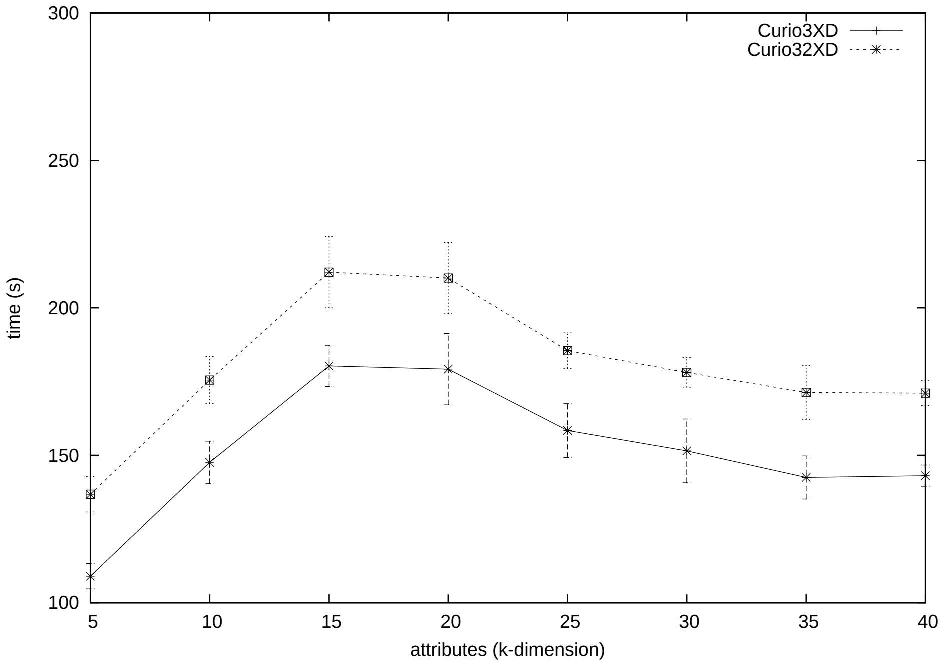


Figure 16. Average execution times (sec.) with dimension of dataset for the Curio3*XD variants.

As can be seen, and the most important result of these tests, the algorithms behavior is noticeable independent of the data dimension. The most time expensive, in this specific case, are the situations with 15 and 20 attributes, and even with constantly smaller average execution times for higher dimensions.

In general, the times for execution of the full algorithm process is considerably small, around 150 seconds for Curio32XD and 180 seconds for Curio32XD, in a range from 10 to 40 dimensions, with no remarkable growth, much less an exponential one, with the raise of dimensionality.

The average total times for the higher dimension case, the one with 40 attributes, are extremely fast and fully applicable in a real-world situation, an example outside the processing possibilities of the original CURIO or any of the previous distributed versions proposed here. Ceglar et all., shown in their results execution times in the order of 10^4 seconds for 16 dimensions[9] of the same dataset and in our tests, with the previous distributed versions, average execution times of the order of 10^3 seconds, for that number of dimensions.

The fact that, although derived from the same data, all datasets in this test are structurally different among each other is the main reason for the non-monotonous aspect of the execution times curve in the different dimensions. Each of these datasets represents some partial subspace of the full dataset (41 attributes), similar to sectioning a 2D plane of a tridimensional space, and so, no direct correlation exists between the partitions or their nature as potential outliers from a dataset and its closest variant with ± 5 dimensions.

The number of partitions will change, unpredictably, for each of those subsets of data, and being the algorithms dependent on the number of detected partitions, the runtimes will also change accordingly. But times variations from that fluctuations are however much smaller than the ones expected on an exponential growth with the variation of the dimension number. The graphic demonstrates the independency on the dimensions and the consequent potentiality of application of the algorithms on high dimension datasets, without the risk of exponentially high execution times.

The error bars shown are the standard deviation of the 10 runs executed for each case in order to estimate good average times and minimize the effects of fluctuations on the number of potential outlier partitions.

The two algorithms examined in this section, Curio3XD and Curio32XD, change the dependency on the total number of nearest neighbors partitions to the number of partitions detected with existent instances in the dataset, the non-empty ones. That's a quantity directly influenced, although unpredictably conforming the nature of data, by the refining of partitioning and thus by the precision parameter. The finer the partition size (higher value of precision) the bigger the number of partitions, but also the more sparse will be the density of instances on those cells, increasing, in general, the number of detected partitions and, of those, the potential outliers (for a fixed value of tolerance).

The raise of the precision value divides, for an increment of one, each dimension of data in two, raising the number of possible partitions to 2^k (see *CURIO: Partitioning*, in section 3). But the growth of detected partitions, c, is much slower, since most of the possible new partitions are empty, and depending on the local density of data and granularity of precision, some partitions may divide in a smaller number than the 2^k or even not divide at all. In any case, the limit situation will be each partition detected having only one instance, being then N the maximum number of detected partitions, c.

The way that the number of detected partitions growth with the different partitioning is not exactly determinable, and will vary among different datasets and precision values. For an empirical approach to the performance of these two algorithms with the refinement of partitioning, tests with several values of parameter precision were executed.

In this next series of tests, the values of tolerance needed to be changed to some that would allowed the existence of at least a minimal number of outliers for all values of precision, to achieve comparable and meaningful results. Tests were made with a tolerance of $T = 2000$ and repeated for $T = 5000$. The dataset used had 15 millions of instances and 40 dimensions, on the same cluster configuration as before, with a fixed number of 9 nodes.

The results for the variation with the parameters precision P and tolerance T, are shown in figure 17:

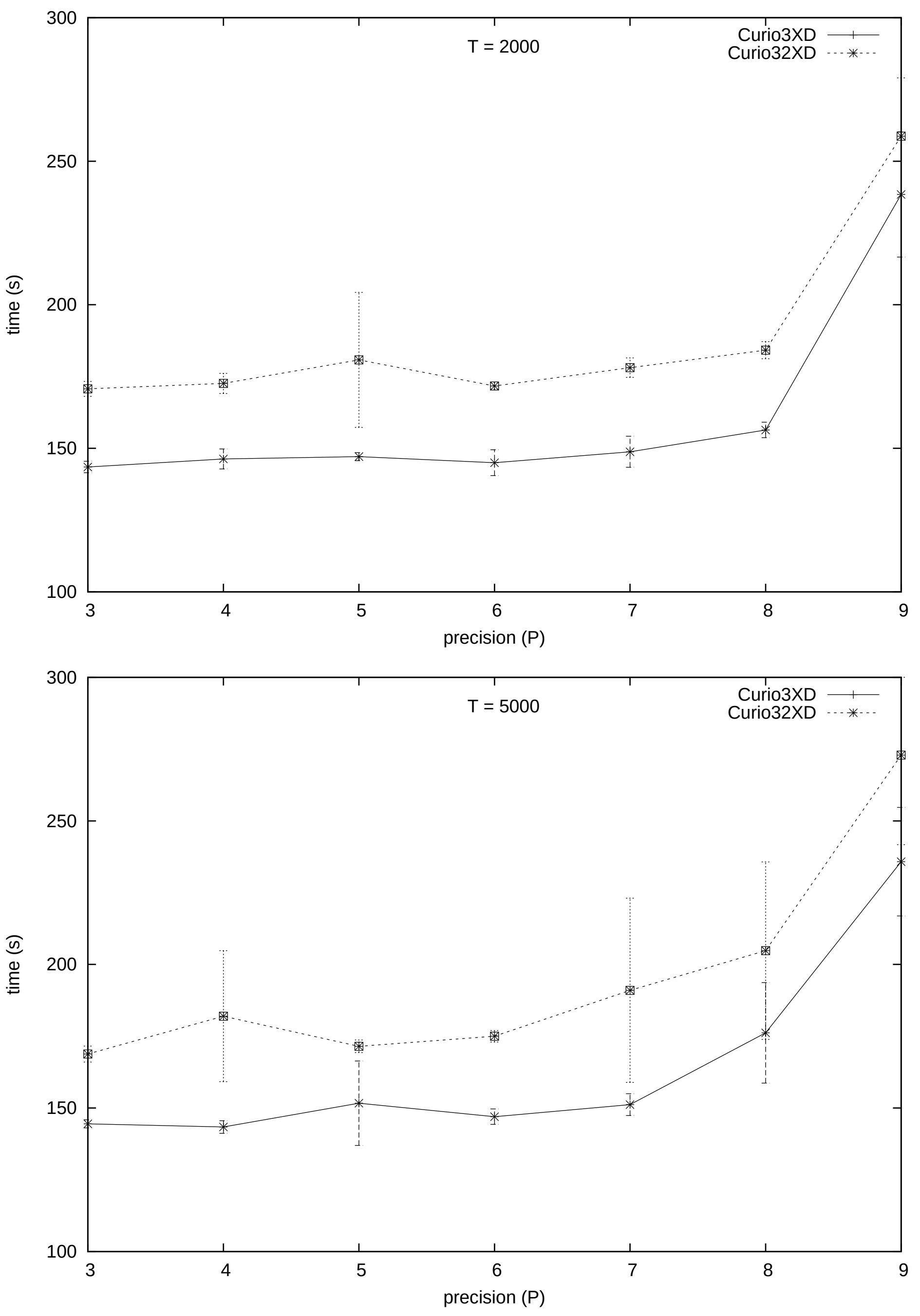


Figure 17. Average execution times (sec.) with Precision parameter for the Curio3*XD variants, for two values of tolerance, $T = 2000$ (above) and $T = 5000$ (bellow).

The execution times are fairly constant with the variation of precision, for values of 3 to 8, without a noticeable growth in the order of 2^P, and a bigger execution time for the precision value of 9.

In our tests, values above $P = 9$ resulted in memory problems, which prevented the successful completion of the tasks. This shows the limit abilities of these algorithms, associated to the precision chosen and subsequent partition granularity, rather than the dimensionality of the dataset. Nevertheless, it shows also a relatively broad range of precision values perfectly valid in terms of data exploration for outlier detection. It will be a question for the user to decide if the possible partition refinements are enough for the desired purpose on particular cases, or if a different algorithm, like CurioXRRD, should be chosen.

No significant variations in execution times were found for any of the tolerance values in test. The algorithms are directly dependent on the number of detected partitions and not in the ones that are potential outliers, the characteristic changed with the variation of tolerance. It would be, however, interesting to investigate if such parameter has an indirect influence in the performance of these algorithms.

Tolerance may affect the overall execution times due to the final emission of the outlier instances from validated potential outlier cells. For a given dataset, and with all other parameters fixed, the tolerance variation will proportionally change the number of potential outlier cells. And similarly the limit of acceptability from the neighbors' instances count will change, being the tolerance also the value of that limit. Those two factors lead to a probable raise in the number of validated outliers with the increment of the tolerance.

Obviously, the final number of outliers will always depend on the data distribution of each dataset treated. The way such variation happens can only be estimated.

If the dataset is large – heavily populated and with a big number of partitions, – the density of the detected partitions will tend to hold all possible values, from the single instance allocation to very dense occupations. Assuming a uniform distribution of such densities, implying the some probability for any density, then can be expected a linear variation with the tolerance. That's the limit situation; usually is expected that outliers, corresponding to low-density cells, will be rare, and so less probable.

Using the values from this test and the previous one for the dimensionality, for the precision $P = 8$ and the dataset with a dimension of 40 attributes, the figure bellow was made. The result obtained seems to sustain the estimated linearity in large datasets for the tolerance variation.

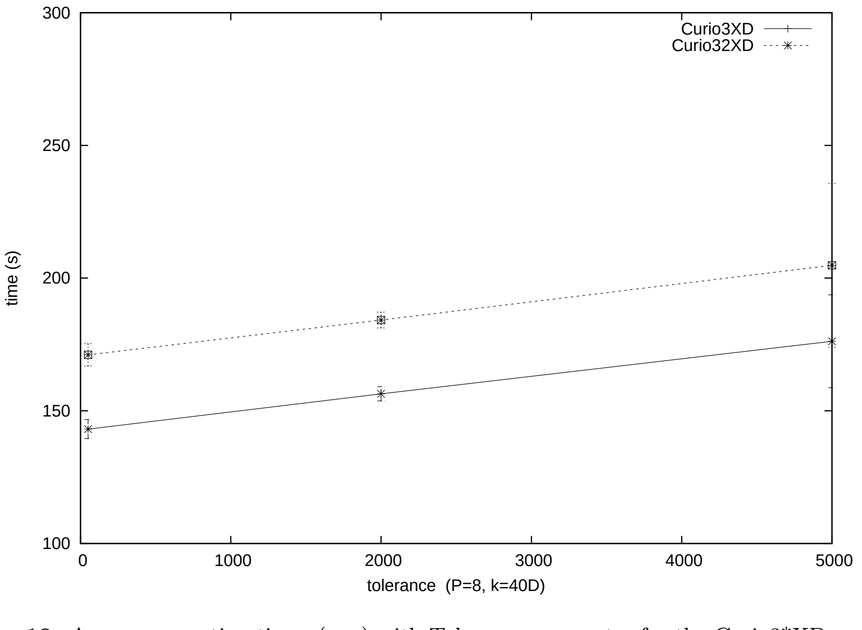


Figure 18. Average execution times (sec.) with Tolerance parameter for the Curio3*XD variants.

7. Conclusions

This work presents some new methods for an efficient detection of outliers on large datasets even in case of high dimensionality data. They are proposed for distributed computing in clusters or clouds of any number of machines, using the MapReduce data model.

The situation of clustering of outliers in distributed mode was not explored since the MapReduce model, in opposition of the case of original CURIO, would make that process unnecessarily complex and heavy, easier managed by a posterior application of one of the usual, well-known methods of clustering on the resultant set of outliers. Note that, by definition, outliers must be a small, partial portion of the precessed data, and consequently constitute a set of much smaller size.

The final algorithm and its variant is the result of a research process started by the implementation, within that model, of a distributed version of a pre-existent algorithm, CURIO[9], and continued in a series of optimizations applied to that initial version. They can be resumed as:

- CurioD; direct implementation of CURIO in distributed mode, with $O(2N+c3^k)$
- CurioX*D; first series of optimizations, with $O(N+c3^k)$
- CurioXRRD; Hadoop specific optimization, chained Reduce tasks, with $O(N+p3^k)$
- Curio3*XD; in memory mapping validation, with $O(N+c^2)$.

Tests performed shown that the CurioD and all variants of the CurioX*D are of little interest for real-world applications. They lack the performance of the later variants implemented, suffer from memory limitation problems, in particular when comes to datasets with a high dimensionality. They are presented for the simplicity in the understanding of the more efficient, subsequent algorithms and the process that led to these.

Total execution times for the Curio3*D series are consistently better, 20 times faster in general, for any number of nodes.

Tests of scalability with the number of machines constituting the distributed computing environment permit to verify the existence of an optimal minimal number of nodes for a dedicated algorithm processing. Above that number, the gains with the size of the cluster and the distributed computation for the task in question are of not much relevance (figures 10 and 11).

A separated analysis on the performance of the algorithms' first stage, the data space partitioners, allows to verify (fig. 12) by the order of their runtimes, that they are not the weakness of the proposed methods, at least for the group of the first five algorithms (in fig. 10), and yet of relative relevance for the algorithms Curio3XD and its variant. For the dataset used in this tests, that stage of algorithm represents 80% of the total average execution time, thus justifying a careful search for optimizations.

It was showed that the best choice, the PartitionerXD with Combiner, although more complex in code procedure and the extra load of the combiners itself, is notable better, approximately 150 sec. versus the average 190 sec. of the others, due to the optimization induced by distributed work of those local combiners.

The scalability with the dataset size is excellent for PartitionerXD with Combiner (fig. 13), being the execution times the faster, approximately 150 sec. in this test, and practically constant for the broad range from 5 to 60 millions of instances. Note that this stage of the algorithms is directly dependent on the size of the dataset.

Scalability with the data size is generally good for all algorithms, with constant times across all the range of sizes tested, from 5 to 60 millions of samples, although in terms of absolute performance this test is also clearly dominated by Curio3XD and Curio32XD algorithms (figures 14 and 15), 15 times faster than the CurioD or the CurioX*D series. Is still quite interesting the performance of CurioXRRD with larger volumes of data, for the dimension of the dataset in test (15 attributes), indicating the possibilities of that algorithm as an alternative for Curio3*XD, on datasets with exceptionally large volumes of data spreading across a big quantity of partitions with low/medium dimensionality.

Some preliminary experiments with dimension above 15-20 attributes, executed with all algorithms revealed execution times excessively lengthy and for some versions (in particular CurioD and CurioXD) with errors that lead to the unexpected termination of the MapReduce jobs, due to the full usage of available resources, like the memory assigned for the JVMs controlled by Hadoop.

Tests with the dimension were, therefore, executed exclusively for Curio3XD and Curio32XD. Results show an excellent performance runtimes (fig. 16) across the range of dimensions from 5 to 40 attributes, without much direct influence of the variation of dimensions on the performance. That is in accord with the expected theoretically, since those algorithms are no longer exponential with dimension, but quadratic on the number of partitions.

In order to test the dependency on the number of detected partitions, c, the parameter precision was used to generate a varied number of partitions in the same dataset. Results (fig. 17) appear to indicate that the algorithms are well behaved for a good range of precision values (3 - 8 in our tests and for this data), but result runtimes grow fast for a precision of $P = 9$ and memory issues were hit above that value.

This result shows to be the precision the limitative factor for the use of Curio3*XD algorithms, being however a parameter easily controllable by the user and holding yet a relatively large range of values, always accessible (in last resource by data rescaling), for data analysis and outlier detection.

Tolerance is a less relevant factor for the implemented algorithms in distributed computing. Our results show a good scalability, with a linear dependency on the raise of the values of that parameter for the same volume of data.

The good results of the Curio3*XD algorithms, are due both to the distributed approach to take care of the problems associated with the size of data, and the neighbors' memory mapping allied with the proposed method on validation phase, to workaround the problems related with dimensionality, making these versions applicable also for high dimensions dataset.

If such combination will solve the problem or alternatives, like CurioXRRD will be preferable or required, eventually to resolve problems with available memory, replacing such requirement with computational capacity and extended execution times, will be a question for the decision of the data analyst.

7.1 Future Work

Testing for the proposed algorithms are far from exhaustive or complete. More and bigger real-world datasets are needed for a deeper study of the abilities of these methods. In particular, the scalability with the size of datasets was not totally conclusive, in particular for the latest algorithms, able to deal with much larger datasets in a useful time.

It will be necessary to perform better testing for the CurioXRRD variant, validating the results obtained with different and larger datasets (instead of the one used with artificially enlargement of the data volume). Also, not executed here, would be desirable some comparative tests with the Curio3XD/Curio32XD versions, in the low-medium dimensions case, clarifying the limits of usability of CurioXRRD with the dimensionality.

A promising development, to be done in a next Hadoop version, a truly ChainReduce implementation of CurioXRRD (available in the recently released stable Hadoop 1.0), without the need of the heavy and unneeded shortcoming of the two MapReduce jobs just to obtain the succession of Reduces on the second stage of that algorithm.

An approach of great interest, not considered in this work, is the possibility of use the flexibility of Hadoop to expand on-the-run the number of nodes in conjugation with technologies like elastic clouds (e.g. EC2) with auto-scaling abilities. Such features would be of particular interest for the cases of application to dynamic datasets, with real time updates.

The authors of the original algorithm[9] do not refer any heuristics for the choice of the appropriated values for its two parameters: the precision and tolerance. Through this work, simple heuristics were defined, based on the definitions of the parameters and its applicability, to allow consistent values for testing across large variations of the clusters size and datasets size and dimension. For a real data analysis or any real-world applications better heuristics will need to be designed and tested across a diverse range of situations and datasets.

In this work was used without any further considerations the nearest neighbors' validation condition (see chapter 3. Theory) proposed in the original CURIO algorithm[9]. Although quite general, it's also very arbitrary and its proposal doesn't proceed from any reasoning or heuristics referred by their authors.

Situations like a partition with a density slightly higher than the tolerance, but with a sparse neighborhood with an instances count inferior of tolerance, can be argued to be still an outlier partition, since the sum of instances in the cell and in its nearest neighbors' partitions would still be smaller than $2T$. But such case will not even be considered for validation with the present validation rules.

The exploration of other validation conditions seems to be of the most importance for a more precise and more generic applicability of the proposed methods. A possibility suggested for posterior study would be consider the condition for a partition qualify as potential outlier as $\textit{count(instances)} < 2T$ and then requiring that:

$$\textit{count(instances)} + \textit{sum(neighborsCounts)} \leqslant 2T.$$

Such set of conditions validates as positive the above-mentioned case and is of easy implementation or adaptation. In the code proposed, CurioXD contain it already, as a comment line, since in that specific implementation the changes resumes to a single line of code with a validation proposition.

Other validation conditions are, of course, possible and its exploration interesting for the purpose of the algorithms and the outlier detection.

In order to consider a more general-purpose method, the situation of mixed attributes datasets must be approached with additional care. In this work, a numerical conversion to discrete values was taken from the categorical attributes, in order to reference them in the partitions index scheme of the proposed algorithms, considering that the purpose is just their allocation to the correspondent partition of the data space.

But, besides the extra step of pre-processing the data executing the conversions, the association of the categorical values with a numerical scale is not obvious or trivial, and no guaranties can be given that the conversion process still preserves the underlying model of data on analysis. Another problem, although directly related, arises from the concept of neighborhood, eventually meaningless in such case, aggravated by the sparseness of data space (considered in detail by Koufakou & Georgiopoulos[28]).

A future line of research and experiment would be the development of a separated data analysis of categorical attributes, like in methods exposed by Otey et al.[34] and Koufakou & Giorgiopoulos[28], with an implementation of a MapReduce dedicated process for that type of data. The adaptation of the existent MR-AVF[29] or some equivalent algorithm could create a full MapReduce method for outlier detection, able to deal with mixed attributes without the artificial conversion to numerical. Even the case of some lost of performance, such approach should benefit of a better accuracy of results.

Appendix A. Apache Hadoop Installation and Configuration

The distributed system was implemented in the cluster of the Department of Electrical and Computer Engineering of Universitetet i Stavanger and, during the initial tests, on Amazon Elastic Compute Cloud EC2, using a customized AMI image with installations of Hadoop and developed code (details of AMI creation and use are left to Appendix B, page 100).

On both situations, the operating system was a Linux Cloudera, a special Hadoop oriented GNU/Linux distribution. The pre-existent Hadoop installation included the support for Hadoop's distributed system as a generic system services. After some initial issues with that version, requiring occasional restarts of the system and consequently the impractical reboot of some machines in the cluster, was opted by a re-installation of Hadoop in its original, pristine version and the maintenance of Hadoop services and distributed system by the more conventional way of direct run of the included scripts.

This installation was made on both situations, the latest stable version available at the time, Hadoop 0.20.2, following closely the tutorial suggested on Apache site from Michael Noll[33] for a multi-node cluster setup. This also include the advantage of running all Hadoop related processes and applications as a specific-configured system user, for simplicity usually named hadoop, to a safer practice.

A detailed step-list of the installation (identical for all cluster nodes and for the AMI image) is shown next:

# Remove Cloudera Hadoop:

```
sudo yum remove hadoop
```

Sometimes problems can occur when connecting to localhost. It's suggested to move the existent .ssh file to a backup file, and generate a new key.

# Backup the .ssh file:

```
mv .ssh .ssh_BKUP
```

# and create new key:

```
ssh-keygen -t rsa -P ""
cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

# Connect to localhost:

```
ssh localhost
```

# Set Hadoop-related environment variables:

```
echo 'export HADOOP_HOME=/usr/local/hadoop' >> .bashrc
```

# and add Hadoop bin/directory to PATH:

```
echo 'export PATH=$PATH:$HADOOP_HOME/bin' >> .bashrc
```

- Hadoop Installation:

# Get the latest stable Hadoop (at the time of this work version 0.20.2):

```
wget ftp://apache.uib.no/pub/apache//hadoop/common/hadoop-0.20.2/hadoop-
0.20.2.tar.gz

sudo mv hadoop-0.20.2.tar.gz /usr/local

cd /usr/local

sudo tar xzf hadoop-0.20.2.tar.gz

sudo ln -s hadoop-0.20.2 hadoop
```

# Will install system user hadoop:

```
sudo chown -R hadoop:hadoop hadoop-0.20.2
```

# and will need the following directories with these ownership and permissions:

```
sudo mkdir /usr/local/hadoop_tmp_dir

sudo chown hadoop:hadoop /usr/local/hadoop_tmp_dir

sudo chmod 750 /usr/local/hadoop_tmp_dir
```

- Hadoop Configuration (Multi-Node):

# Edit *hadoop-env.sh*:

```
sudo nano /usr/local/hadoop/conf/hadoop-env.sh
```

# and add/edit the Java version to use and the amount to heap to Hadoop:

```
# The java implementation to use (Required!)
export JAVA_HOME=/usr/lib/jvm/java-1.6.0

# The maximum amount of heap to use, in MB (default is 1000)
export HADOOP_HEAPSIZE=2000
```

# Edit *core-site.xml* (on hadoop/conf directory):

```
sudo nano /usr/local/hadoop/conf/core-site.xml
```

# and add/edit the properties:

```
<property>
  <name>hadoop.tmp.dir</name>
  <value>/usr/local/hadoop_tmp_dir</value>
</property>

<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:54310</value>
</property>
```

# Edit *mapred-site.xml*:

```
sudo nano /usr/local/hadoop/conf/core-site.xml
```

# and add/edit:

```
<property>
  <name>mapred.job.tracker</name>
  <value>localhost:54311</value>
</property>


<property>
  <name>mapred.child.java.opts</name>
  <value>-Xmx2000m</value>
</property>
```

# Finally edit *hdfs-site.xml*:

```
sudo nano /usr/local/hadoop/conf/core-site.xml
```

# and add/edit value for the dfs.replication (3 is the default for fully distributed mode):

```
<property>
  <name>dfs.replication</name>
  <value>3</value>
</property>
```

In case of clusters or grids you can repeat these steps on each node, or copy the above installation and assign correct ownership and permissions.

It will also be necessary to update the files *masters* and *slaves*, on hadoop/conf directory, according to the correct quantity and alias of nodes to use. For simplicity it's better to add alias for each node on file */etc/hosts* of master, like slave1, slave2, etc., making simpler to deal with names, instead of IP addresses. The *masters* file, with a name somehow misleading, serves to define the machines that will run secondaryNameNodes[?]. These files need only to be changed on the master node.

When all is installed and ready, you need to format the namenode and run the Hadoop services, that is, NameNodes and DataNode, JobTracker and TaskTracker. In the master node run:

# Format the namenode:

```
hadoop namenode -format
```

# Start Hahoop services:

```
start-all.sh
```

And Hadoop should be up and running. To check if all is running up and well, check for any error messages on the previous commands output, and verify the Hadoop services/processes with the Java line-command **jps** utility.

# check Hadoop services running:

```
jps
```

It should show the 5 processes:

```
NameNode

SecondaryNameNode

DataNode

JobTracker

TaskTracker
```

The following table list the properties/options changed in the tests executed in this work:

property name	**default**	**custom value**	**property description**
io.file.buffer.size	4096	65536	Size of read/write buffer used in SequenceFiles.
fs.inmemory.size.mb	50	100	Amount of memory allocated for the in-memory file-system used to merge map-outputs at the reduces
io.sort.factor	10	50	The number of segments on disk to be merged at the same time (more streams merged at once).
io.sort.mb	100	200	Higher memory-limit while sorting data.
io.sort.spill.percent	0.80	0.80	Threshold for serialization buffers. When buffer's percentage is filled, the contents is spilled to disk.
dfs.block.size	64M	134217728 (128M)	HDFS blocksize, for large file-systems.
mapred.map.tasks		10× num. nodes	Number of mappers per node.
mapred.tasktracker.map.tasks.maximum	2	5× num. processors	Maximum number of MapReduce tasks which run simultaneously on a given TaskTracker, individually.
mapred.reduce.tasks		2× num. nodes	Number of reduces per node.
mapred.tasktracker.reduce.tasks.maximum	2	5× num. processors	Maximum number of MapReduce tasks which run simultaneously on a given TaskTracker, individually.
mapred.child.java.opts	1024	-Xmx2048m	Heap memory reserved for each child process.
mapred.reduce.parallel.copies	5	10	Higher number of parallel copies run by reduces to fetch outputs from large number of maps.

Table 14. Hadoop properties customized on the installations.

Appendix B. Cloud Computing and Amazon EC2

For execution on the cloud, it was used Amazon Elastic Cloud Computing (EC2) services. Amazon EC2 run over images of operating systems at users choice, called Amazon Machine Images, AMI. At the time of this writing there were no images with updated versions of Hadoop available.

The installation for the cloud was made based on the instructions in Amazon's "Creating Your own AMIs"[17] and in "Mahout on Amazon EC2"[31] which include detailed information about Hadoop installation (and, if desired, Mahout software for distributed Machine Learning) on this context, over an available Cloudera Linux distribution AMI.

A resume of the AMI creation is presented next, in a step-list:

From the **AWS Management Console/AMIs**, start the following AMI (*ami-8759bfee*):

```
cloudera-ec2-hadoop-images/cloudera-hadoop-ubuntu-20090623-x86_64.manifest.xml
```

From the **AWS Console/Instances**, select the instance and right-click "Connect" to get the connect string which contains your `<instance_public_DNS_name>`:

```
ssh -i <gsg-keypair.pem> root@<instance_public_DNS_name>
```

To update the operating system on the image and install some useful software, run:

```
apt-get update
apt-get upgrade
apt-get install ant subversion
```

Add the following to .profile file:

```
export JAVA_HOME=/usr/lib/jvm/java-1.6.0
export HADOOP_HOME=/usr/local/hadoop-0.20.2
export HADOOP_CONF_DIR=/usr/local/hadoop-0.20.2/conf
```

Upload the Hadoop version desired and configure it (or you can download it directly from Apache Hadoop):

```
scp -i <gsg-keypair.pem> hadoop-0.20.2.tar.gz root@<instance_public_DNS_name>:.
tar -xzf hadoop-0.20.2.tar.gz
mv hadoop-0.20.2 /usr/local/.
```

The subsequent configuration and general setup was made according to Noll[?] tutorial, as described in previous Appendix, page 95.

Finally, you will need to bundle your recent created image into a new AMI and upload it to Amazon S3 to registration (*See* Amazon's Creating Your own AMIs for more details), so it can be launched multiple times to build up an Hadoop cluster of machines.

Copy your AWS private key file and certificate file to the /mnt directory on your instance (you don't want to leave these around in the AMI):

```
scp -i <gsg-keypair.pem> <your AWS cert directory>/*.pem
root@<instance_public_DNS_name>:/mnt/.
```

NOTE: `ec2-bundle-vol` may fail if EC2_HOME is set! So you may want to temporarily unset EC2_HOME before running the bundle command. However the shell will need to have the correct value of EC2_HOME set before running the `ec2-register` step.

```
ec2-bundle-vol -k /mnt/pk*.pem -c /mnt/cert*.pem -u <your-AWS-user_id> -d /mnt -p
mahout
```

```
ec2-upload-bundle -b <your-s3-bucket> -m /mnt/mahout.manifest.xml -a <your-AWS-
access_key> -s <your-AWS-secret_key>
```

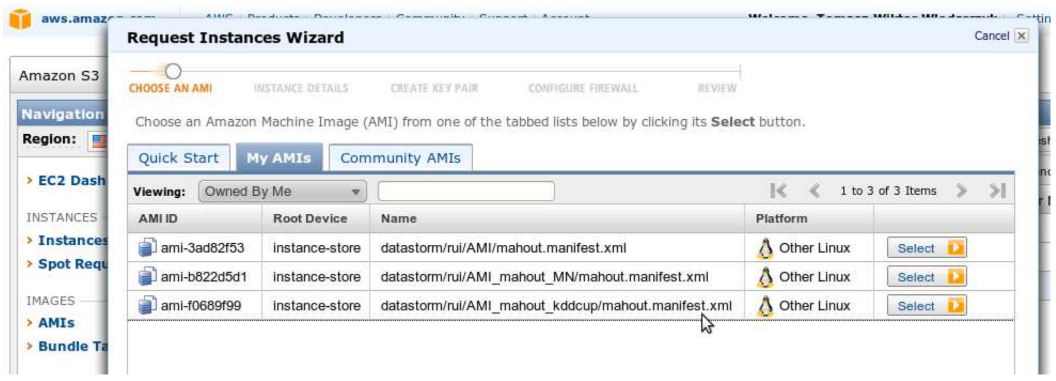
```
ec2-register -K /mnt/pk-*.pem -C /mnt/cert-*.pem <your-s3-
bucket>/mahout.manifest.xml
```

- **Procedures for use of the customized AMI image on Amazon EC2.**

These are the procedures for run a multi-node Hadoop cluster of machines from the customized AMI image, named AMI_mahout_kddcup, with above installation of Hadoop, extra software like Mahout and Weka, and some utility scripts. The dataset original, kddcup.data file used on these tests is also included.

· ***Initiate instance(s) of AMI Image***

On Amazon EC2 tab go for 'Launch Instance' wizard. From 'My AMIs' Instances Wizard choose **AMI_mahout_kddcup**, ID: f0689f99.



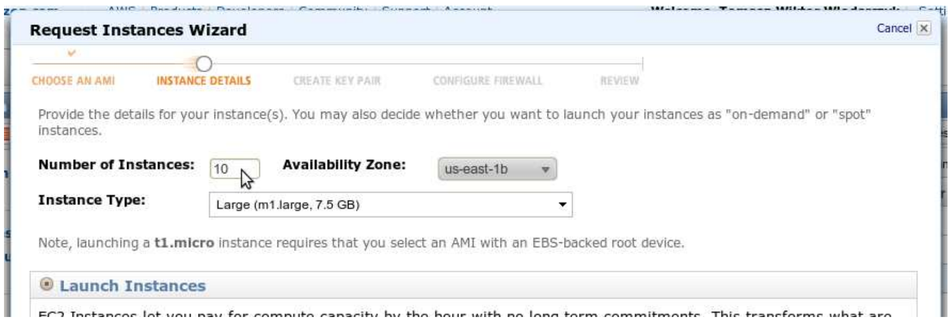
Request Instances Wizard

Choose an Amazon Machine Image (AMI) from one of the tabbed lists below by clicking its **Select** button.

Viewing: Owned By Me

AMI ID	Root Device	Name	Platform	
ami-3ad82f53	instance-store	datastorm/rui/AMI/mahout.manifest.xml	Other Linux	Select
ami-b822d5d1	instance-store	datastorm/rui/AMI_mahout_MN/mahout.manifest.xml	Other Linux	Select
ami-f0689f99	instance-store	datastorm/rui/AMI_mahout_kddcup/mahout.manifest.xml	Other Linux	Select

The next step of wizard allows to choose the number of instances to launch.


Request Instances Wizard

CHOOSE AN AMI · INSTANCE DETAILS · CREATE KEY PAIR · CONFIGURE FIREWALL · REVIEW

Cancel

Provide the details for your instance(s). You may also decide whether you want to launch your instances as "on-demand" or "spot" instances.

Number of Instances: 10 **Availability Zone:** us-east-1b

Instance Type: Large (m1.large, 7.5 GB)

Note, launching a **t1.micro** instance requires that you select an AMI with an EBS-backed root device.

Launch Instances

After all the instances starts, it's necessary to collect the private IPs assigned to each of them we want to set as slaves (not need for master). This can be done with some fastidious but simple work by selecting one by one and check the Private IP Address on Description low section of the page.

Connect to the instance choose as master and run the script configIPslaves.sh (*see* Appendix C) with the collected IP addresses. Here IPs are an example, for a configuration with 8 slaves:

```
./confIPslaves.sh 10.122.13.69 10.223.29.201 10.127.97.54 10.223.34.193 10.122.126.175
10.220.37.132 10.253.143.127 10.122.199.62
```

This will create alias for all slaves as: slave1, slave2, ... , slave8.

NOTE: If script stops before it adds all slaves to hosts file (it can happens when list of slaves is long, due to some time off from the underneath system) just repeat:

```
cp /etc/hosts.backup /etc/hosts
```

and run the scrip again.

The scrip will output, after run, the quantity and list added slaves, making easy to check that the number of slaves is the one chosen.

. *Run Hadoop*

For security reasons is required the login as `hadoop` user:

```
su - hadoop
```

and it's necessary to add the slaves alias as *know hosts* to the hosts list, to ensure that master connects to all slaves. This can be done by ssh each one of slaves, or simply by running the script:

```
./add_Slaves_to_know_hosts.sh
```

NOTE: if any of them fails to connect, run:

```
ssh <slaveN_that_failed>
```

to ensure that the slave is contactable.

And all is ready to start Hadoop:

```
hadoop namenode -format
```

```
start-dfs.sh
```

```
start-mapred.sh
```

It should output start-up references to the master and all slaves on both commands.

. ***Outliers Detection in Distributed Mode.***

Copy the dataset to analyse into the HDFS file system. As an example, consider the 40D kdd.cup dataset is on file kdd.cup40D.dat. To copy it to an HDFS folder named input, do:

```
hadoop fs -put kdd.cup40D.dat input
```

HDFS directories can have any name (and they shouldn't exist prior to hadoop comand!), but to work with the proposed `start_curio.sh` script (*see* Appendix D, page 184) the directory containing the data should be named **input**.

To run one of the implemented algorithms it's easier to use start_curio script. For instance, for the Curio3XD variant with parameters $P = 8$ and $T = 50$ run:

```
./start_curio.sh 8 50 Curio3XD
```

When script finishes, output data – the outlier intances, – will be written on an **output** directory of the HDFS system, and can be retrieved with:

```
hadoop fs -get output output40D
```

making all the final output of the Curio3XD algorithm available from a local folder, in this case named **output40D**, of the master disk.

Appendix C. Algorithms Implementation Under Hadoop

This appendix details the implementation of the proposed distributed algorithms and variants, with special focus on the use of the MapReduce libraries from Hadoop framework. The code for the algorithms proposed was developed in Java, to work within the stable version, 0.20 or above of Apache Hadoop.

Debugging phase was done on standalone and on pseudo-distributed modes[42], on a personal computer running Hadoop on top of an Ubuntu Linux dstribution. Preliminary tests were conducted on Amazon EC2 cloud running a customized Linux+Hadoop AMI image (see page 100), prepared for that purpose. The final tests were done on the cluster of the Department of Electrical and Computer Engineering of University of Stavanger (see 5.3).

The details of the installations and configuration of the cluster are given on Appendix A, on page 95.

C.1 Hadoop's MapReduce Development.

The code developed use the MapReduce implementation from Hadoop framework libraries. The MapReduce model is implemented via Java abstract classes, from version 0.20.0 ahead (older version use Java interfaces and are type-incompatible with this new API). The main classes are the Mapper and the Reducer and they should be extended with the code to provide the wanted map and reduce methods[23].

The libraries also include objects to manage the settings for the application, the `Configuration`, and its functioning and execution, namely the `Tool` and the `ToolRunner`. Although the implemented MapReduce process can be called from a `main` function of a class, is preferentially done by overriding the generic method `run` of the `Tool` object. This more generic approach is named the driver method[42]. Any class can then run the job(s) of the implemented MapReduce driver, by a single evocation of the `run` method of the `ToolRunner`, passing as arguments an instance of the driver, an optional instance of a `Configuration` object and any other arguments in an array.

As an example, to execute the CurioD driver a single line of code is needed:

```
int result = ToolRunner.run(new Configuration(), new CurioD(), args);
```

with `result` containing the exit code from the driver termination.

MapReduce jobs are configured through a `Configuration` object, either passed by the `ToolRunner` of defined inside the run method of the `Tool` object. Assign values to global variables and parameters of an application running under Hadoop on a distributed system, is neither direct nor trivial. The underlying mechanism of Hadoop needs to ensure that the assignment of global values is correctly done and values are identical in any machine involved in the MapReduce job.

Users interact only with the instance running on the master, all the copies of the application to be run on the slaves are started by the application itself, from the master node. No values are directly passed or assigned over the network, but they are all defined, parameter name and value, to the `Configuration` on the master, that in turn is propagated to the slaves during their initialization. `Configuration` parameters are set at the application start on the master and read by the override `setup` method of the Map or the Reduce on the slaves, which get those values from the `Configuration` copy and assigns them to its own respective variables. That way a value for a parameter is assigned identically to all machines in the cluster involved on that MapReduce job.

This mechanism is used by the distributed variants of CURIO to assign values for the Precision and Tolerance parameters and, optionally, to provide also the expected values for the maximum and minimum of the dataset.

The anatomy of a MapReduce job in the Hadoop's driver method is simple and relatively intuitive. A new Job object must be initialized, receiving the configuration object as argument (and also an optional description for the job, for easy reference on logs and debugs). The new `job` object must know which class, in the jar, it should use. That is assigned by the `setJarByClass` method. It will also need to know what are the Map and Reduce classes that implement the methods for the MapReduce process. And it will need to have defined the type of the keys and the values to be handled. The application will need to know also the input and output directories, where to collect the data and save the final output, respectively. The order in which such information is set is not important.

Finally, the job must be submitted to the cluster and executed. This is usually done by call (from 0.20 version ahead) the method `waitForCompletion`, that run all copies in all slaves and put the master in waiting mode until the completion of the job is reached and the method returns a success code. For exemplification purpose, the code for the `job1` of the first algorithm implemented is shown bellow:

```
Job job1 = new Job(conf, "Curio Algorithm Hadoop version");
job1.setJarByClass(CurioD.class);

job1.setMapperClass(CellsCountMap.class);
job1.setReducerClass(CellsCountReduce.class);


job1.setOutputKeyClass(Text.class);
job1.setOutputValueClass(IntWritable.class);


FileInputFormat.addInputPath(job1, new Path(args[0]));
FileOutputFormat.setOutputPath(job1, new Path(tempPath));


int success = (job1.waitForCompletion(true) ? 0 : 1);
```

C.2 Algorithms Implementation under Hadoop.

CurioD was implemented in Hadoop framework as straight as possible, following the proposed algorithm. In the code, no Hadoop-specific optimizations were employed, since the purpose of this version was essentially to proportionate a reference for the tests of the proposed variants.

In the present implementation the class expect two parameters, the Precision and the Tolerance, to be passed by the user at startup. The application cannot run without those values defined at beginning, and they depend solely on the user's choice and his/her data knowledge or prior tests.

Two more parameters can be passed then, the minimum and maximum values expected on the dataset. The partition indexing function implemented here was the one presented in the exposition of the CURIO algorithm (see section 3.3); the maximum and minimum values are wanted by the indexing function.

The minimum will trigger an automatic shift if negative scales are involved, to solve any issues with the binary conversion. The maximum is needed in order to verify the quantity of significant bits on the binary representation of the values, and some rescale may be applied to allow a sufficient detailed scale to the partitions to be drawn in. The rescale, possibly with a rounding to integers values, must ensure that a sufficient large number of bits on the binary representation enable the cutoff, for the subsequent concatenation, of the number of most significant bits defined by the Precision parameter.

These operations – shift, rescale, rounding and binary conversion, – are done automatically by the present implementation, upon the fixation of the maximum and minimum values expected, and this indexing function is common to all algorithms developed for this work. The shift, when required, is applied always by a value equal to -minimum, transferring the minimum expected to the zero of the scale. By default, all dataset values are rescaled by a factor of 10, or a factor of 100/maximum if maximum is less than 1. No rescale is applied if maximum expected is bigger than 1000.

To clarify, consider a dataset with values consisting of decimal numbers in the interval [-1, 1] and the parameter Precision set to 5, chosen for the sake of the example.

The minimum and maximum expected would be, respectively, -1 and 1. A shift would be necessary for reallocate the values to a full positive interval, and a rescale would be required to produce a higher number of significant bits, at least up to 5, the defined value of the Precision. And so, in this case, a shift of 1 and a rescale by a factor of 100 (from 100 divided by the maximum 1) of all dataset values would be applied, reallocating them in the interval [0, 200], and a conversion to binaries in the range 00000000 to 110010000 (note that a Precision greater than 8 would require a redefinition to a higher value of the rescale factor).

Consider that the following entry,

0.23 -0.34 0.988

is an instance sample from that dataset. The indexing function would produce, for this example:

$$\begin{aligned}
\text{Shift} &\rightarrow \ 1.23 \quad 0.66 \quad 1.988\\
\text{Rescale+rounding} &\rightarrow \ 123 \quad 66 \quad 199\\
\text{Binary Conversion} &\rightarrow \ \mathbf{01111}011 \quad \mathbf{01000}010 \quad \mathbf{11000}111\\
\text{Partition index} &\rightarrow \ 01111\ 01000\ 11000 \qquad (P=5)
\end{aligned}$$

The passage of the minimum and maximum expected at startup is kept optional, since these values may not be known or be easy to access by simple inspection with big datasets. For such cases an extra class, ExtremaD, was developed, and is presented in next subsection, that extract the extreme values of a dataset in a full distributed computing mode, at expense of an extra sequential reading of all data. It will be evoked automatically, if those values will not be present at run time, and the results, passed to the `Configuration` object of the application, which propagates the obtained extremes across all machines in the cluster involved in the MapReduce process.

The rest of CurioD code is a direct implementation of the algorithm proposed in the previous section. In the driver method the three jobs are called successively, being the intermediary output paths, defined as internal variables by the application itself, and set as the input path of the succeeding job.

In the code excerpt given above, the Map and the Reduce of job1 are implemented at classes CellCountMap and CelCountReduce repectively. The next job is composed by the classes ValidateNeighborsMap and ValidateNeighborsReduce, and the final output is get on the third job with the classes ListOutliersMap and ListOutliersReduce implementing the MapReduce methods.

With CurioXD variant, the number of jobs is reduced to two, being the final output produced by the Reduce with direct access to the dataset.

Assuming that the application deals only with data on plain ASCII files, as it is the case of Hadoop requirements for datasets, and each instance is contained on a single line, Hadoop's MapReduce framework use, by default, the file byte offset of each line start as Map key, and thus requiring, on this platform, no extra effort or code creating and indexing system.

Additionally, a possible situation needs to be addressed. Because the original dataset can be composed of multiple files in a directory, the Reduce function will need to know also, besides the offset position of the outlier instances, to which file those instances belongs.

To overcome this issue a simple file indexing can be implemented. A global generic ArrayList keeps the names of the files received by the mapper, and the Map function is re-written in order to concatenate the corresponding arrayList index to the offset position, to build the Map key (e.g., a key '3:454434' would refer an instance on the position 454434 of the 3rd file on the input directory).

The Reduce function, with access to the global ArrayList, upon a classification of a partition as outlier, will split the key in its two components, get the filename in the ArrayList with the index of the key's first part and read the file from the offset position on the second part until the next newline occurrence. The Reduce can then emit directly the instance that has just been read.

This extra procedure doesn't change the structure of the algorithm; just introduce a slightly extra computational cost from the ArrayList initialization and the concatenation and later split of the key.

Under Hadoop, data objects that are assembled to or from files across a cluster of computers must obey a particular interface, named Writable. This interface allows Hadoop to read and write the data in a serialized form for network transmission. Several stock classes, implementing Writable interface, are provided to support the most common data types needed while processing data: Text (for String data), IntWritable, LongWritable, FloatWritable, BooleanWritable, available in conjunction with several others for more specific uses (a complete list is available in Hadoop's documentation[23]).

Because of the change of output type, from plain counts to strings containing the indexed key, in CurioXD, the output of the job was changed from type IntWritable to the slightest slower type Text, both Hadoop's internal data types.

From CurioXXD version forward, the output of the MapReduce between jobs was set, for efficient purposes, to be executed in Hadoop's own file format: sequenceFile. This is a high-performance binary format, optimized to Hadoop's processes. Since the output (partitioning indexes and respective counting) between the two jobs is not needed for user reading, that option can contribute to decrease the overload of the complex output format and types, constituted by both numbers and lists of references.

The rest of the code of these two variants is a direct implementation of the proposed algorithms, and the code doesn't use any other specific feature from the Hadoop MapReduce framework.

The CurioXXcD variant is essentially the implementation of CurioXXD with combiners specifically written for each of the two jobs, including the specialized Reduce on the first job to work in conjugation with combiner, previewed in the development of the algorithm.

The code for the first job, the data space partitioning, is implemented in a separated file, named PartitionerXD.java, with the classes CellsCountMap.class, CellsCountCombine.class, CellsCountReduce.class for combination with combiner and an extra CellsCountReduceNoCombine.class, to work with CellsCountMap.class without combiner.

The separated object allows the easy re-use of this implementation of multi-dimensional numerical data spaces partitioning, employing the partition indexing system of CURIO method, in a full distributed computing fashion, with the extra feature introduced in this work, using indexed offsets as references to the instances direct access, for any numerical dataset. In this work, all algorithms proposed from CurioXXD forward use this same partitioner.

At the time of the writing of the code for this thesis, Hadoop stable version was 0.20.2, which did not had a working implementation,for the new API, of chained Mappers/Reducers. CurioXRRD implies its use, and was conceived precisely to explore this Hadoop feature.

In order to avoid a downgrade to previous deprecated versions, resulting in the writing of all code in the old API, or the move to an unstable, still experimental version, a tradeoff was made. The CurioXRRD developed here was done using a total of three jobs: the last data space partitioning version, referred above, a second job with a Map and the first Reduce, for the nearest neighbors validation, finalized by a third job without Map, but just a single Combiner and Reduce function.

This compromise solution, with all code deployed within the new API, allows an easy transition for the future implementation of chainReduces by Hadoop. It can be accomplished without great complexity in terms of code re-written, at an expense of some extra load on the cluster, from the full initialization of the extra third job, some extra disk space and network bandwidth consumption, due to the saving of the intermediary output on master disks, from the second to the third job (process that will be managed in memory by the future full implementation of the chainReduce). Obviously all intermediary output is saved using Hadoop's binary sequenceFile format, to minimize the effect of disk IO operations.

In the new introduced third job, the Combiner receives the data, for each machine, collected and grouped by indexes. No mapping is done. The input will be constituted by an index and the associated list of values, which can contain either a list of references or counts, exclusively or in combination. The Combiner function will just re-emit the list of references (it was already filtered on previous job, so its presence means that it's the case of a potential outlier), or sums all counts found in the values list, emitting the summing. The validation rule is checked at the second Reduce, along with any summing of counts still to perform, and emission of validated instances is made then, finalizing the task.

The algorithm for this implementation can be re-written as:

Algorithm 6b. CurioXRRD Hadoop's variant with 3 jobs.

```
// Job3, Validation Finalization:
 // Combiner
  receive { (index, list[values]) }
  for each token in list[values]
     if token isListOfReferences
        emit < index, token >
     else
        sumNeighbors = sumNeighbors + token
  emit < index, sumNeighbors >


 // Reduce2
  receive { (index, list[values]) }
  for each value in list[values])
     if value isListOfReferences
        Offsets = value
     else
        sumNeighbs = sumNeighbs + value
  if Offsets is not empty and sumNeighbs $\leqslant$ T
     for each offset in Offsets
        outlier = readDatasetAt(offset)
        emit < outlier >
```

For the Curio3XD algorithm, a list of all detected indexes – the ones referring to partitions with instances present in the dataset, – must be assembled and made globally accessible for the Map of the second job. On Hadoop, this procedure has to be implemented by read all the input (the output of the first job, the data space partitioning), and collect the indexes into some data structure, during the overriding of the setup method of Hadoop's Mapper class. The best data structure for optimal performance was found to be an ArrayList. The input is read with the Hadoop's own SequenceFile reader function, provided by the framework, for an easy access of sequence files (automatic format conversion and direct key/index acquisition).

The Map will emit, following the proposed algorithm, the associated value for each index received, either a list of references or a count. It then goes through the ArrayList, emitting the count (or a count of references on the list) in the cases where the ArrayList entry corresponds to a nearest neighbor.

The Reduce will be identical to the one implemented for CurioXXD/CurioXXcD. Curio3XD is essentially an optimization at the Map level, reducing the number of emissions for each partition from the possible 3^k nearest neighbors to the usually much smaller number of non-empty neighbor cells, with the upper limit of $O(c)$ from the indexes' ArrayList checking. In this work and for the later tests it was used the version with a Combiner.

The table solution proposed for the Curio32XD algorithm was implemented with two distinct Java ArrayLists (after some tests that showed faster results than HashMaps or HashTables).

The neighbors' listing is done on the Reduce function of the second job. Each entry from the list of indexes is inserted in one of the ArrayLists, while the other takes a new empty entry. Next, the function iterates over the indexes of the first ArrayList and compares each entry exclusively with the other entries ahead, verifying if they are nearest neighbors. So, only for the first index is the list checked in all extension. That is equivalent, if the neighborhood relationship were mapped in a square matrix, to proceed only for the upper triangular section, and mirroring the data by the main diagonal due to its symmetry nature. If two entries are neighbors, both indexes are appended (by String concatenation) in the second ArrayList, to any other indexes already existent.

As an example, if the index being checked is at position 3, only indexes from position 4 till last indexes on the array will be checked. If it's verified that the index at position 7 is neighbor of the index in position 3, the function will append, in the second ArrayList, the index at position 7 to any others already existent at position 3, and then append the index at position 3 to any other already existent in position 7.

In the end of the process all the values on the second ArrayList, a concatenated list of indexes, are emitted, one by one, by the Reduce.

The final third job is evoked afterwards, to validate the potential outliers, using the existent lists of neighbors built on the previous step. The output of the second job is read by a SequenceFile reader in same fashion of the Curio3XD procedure, obtaining this time a complete list of neighbors for each detected index, instead of the simple list of indexes as in the previous version. The Map in the third job, with access to its neighbors list, splits the concatenated indexes and emits the value of the sum for each of those indexes, without further checks.

All the code for the proposed algorithms was developed using Java 6 and Hadoop Java libraries and is listed on Appendix D, page 113.

C.3 Other Auxiliary Applications Developed.

This work does not try to investigate or prove the quality of outlier detection by the proposed methods. Ceglar et al. presents the results of they test on the CURIO original article[9], with a complete analysis and a comparison with some of the other most recent algorithms.

To verify the correctness of the distributed versions implemented here, it was considered, therefore, that an identical output result, meaning the same instances are classified as outliers as the original CURIO, were enough to validate those results. For that reason, it was also developed the code for the original non-distributed CURIO.

To complement the proposed tasks some auxiliary applications were also developed.

As it was mentioned above, this method requires that the maximum and minimum value of the dataset are known prior to the partitioning of the data space. If such knowledge isn't provided at start time, the application automatically search for those values, executing a full reading of the dataset and collect the extremes values in a distributed mode under the MapReduce model. The code developed for that is contained in a class named ExtremaD.

The computation is accomplished in a single job. The Map receives each instance, and emits a pair with the number of the dimension and the value of the respective attribute. The Reduces receive the lists of values for each dimension and emit the minimum and the maximum of each list, the extreme values for the dimension being processed. The algorithm for this simple job is presented next.

Algorithm 7. ExtremaD.

```
// Job1, Find Minimum and Maximum of a dataset:
  receive { instances }
 // Map
  for each attribute in instance
      dimension = dimension + 1
      emit < dimension, attribute >

 // Reduce
  receive { (dimension, list[values]) }
  for each dimension in { (dimension, list[values]) }
      min = minimumOf(list[values])
      max = maximumOf(list[values])
      emit < min, max >
```

The final output contains, therefore, the maximum and the minimum of each dimension. The absolute maximum and minimum can be retrieved by a simple check on a list of at most k entries of size.

Two more simple applications were developed to facilitate the testing of datasets with partial variations of their characteristics.

One of these applications generates variants of a dataset with different sizes, extracting random numbers of lines, and it is implemented in a class named: `GetRandomLines`. The other application extracts a given number of attributes from a dataset, in order to build variants with a different number of dimensions. It is named: `GetColumns`. The code for both classes is included in the Appendix D, page 171.

To simplify the task of run the different algorithms through Hadoop command line interface, a startup script was create, in order to ensure that some routine commands and the correct command line instruction was constantly used during the tests.

The script, named `start_curio.sh`, is also included in Appendix D, page 184. The script recompiles the proposed algorithms code against the existent Hadoop libraries and builds a new Jar file. It then instructs Hadoop to remove any temporary output directory from the HDFS file system and run the demanded algorithm with the given parameters. Finally it extracts the result from the HDFS (not always a trivial task) to a file on local disk of master, outputting the contents of the file on the terminal. Its usage is:

```
./start_curio.sh P T Algorithm [min max](optional)
```

A sample of is usage is, e.g. for a Precision of 5, Tolerance of 50, on a dataset with a minimum and a maximum of 0 and 100, respectively, using the Curio3XD:

```
./start_curio.sh 5 50 Curio3XD 0 100
```

Appendix D. Code Listing of Distributed Algorithms

D.1 CurioD

The CurioD algorithm code, in CurioD.java file:

```
/**
 * CurioD
 *  Use: Driver or Object Oriented approach.
 *   See main() function for examples.
 */

import java.io.IOException;
import java.util.ArrayList;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 *  <b>CURIO</b> algorithm, by Ceglar, Roddick and Powers,
 *      <br>Flinders UNIVERSITY, Adelaide, 2007.
 *  <p>This is an <b>Hadoop</b> MapReducing implementation,
 *  <br>for a distributed version of the original algorithm.
 *
 * @author Rui Pais
 * @version 0.1.0
 */
public class CurioD extends Configured implements Tool {
 /* NOTE: Hadoop require special initialization of variables,
  *  by conf.set(), and conf.get() on setup() implementations
  *  of MapReduces, in order to work on distributed mode.  */

        //** CURIO parameters:  **//
        // partition precision/division (P on article)
        private static int precision = 3;       // 3 it's the minimum interesting

        // tolerance (T on article):            // the number of instances (counts)
        private static int tolerance;           // counts to be a potential Outlier
```

```
//** private members: NOT CONFIGURABLE FROM OUTSIDE! **//
// A maximum and minimum expected values of dataset
// to define the bit string size on partitioning:
private static String bitFormat;     // = "%0" + bitStringSize + "d";
private static int scaleFactor = 10; // default, changes on setup of 1st Map
private static long shift = 0;       // default, changes only with negatives
private static String inputPath;


protected static void configParameters(Configuration conf, String[] otherArgs)
                                                    throws Exception {
       /* NOTE:
        * In distributed mode (global) variables/parameters
        * should be set to configuration by conf.set() method,
        * and read to the global variables on overrided setup()
        * of each class with conf.get() method.
        */
        long minExpected = Long.MAX_VALUE;
        long maxExpected = Long.MIN_VALUE;

        //TODO: a filter to get these parameters by flags on command line:
        if (otherArgs.length > 2) {
                conf.set("precision", otherArgs[2]);
                if (otherArgs.length > 3)
                    conf.set("tolerance", otherArgs[3]);
                else
                    conf.set("tolerance", "3");
        }
        if ( otherArgs.length > 4 ) {
                minExpected = Long.valueOf(otherArgs[4]);
                maxExpected = Long.valueOf(otherArgs[5]);
        } else {
                // Find Extreme Values:
                String[] args = {otherArgs[0], otherArgs[1] + "_Extrema_tmp"};
                ExtremaD extrValsD = new ExtremaD(args);
                minExpected = (long)Math.floor(extrValsD.getMinValue());
                maxExpected = (long)Math.ceil(extrValsD.getMaxValue());

        }

      System.out.println("Set parameter Precision = " + otherArgs[2]);
      System.out.println("Parameter Tolerance = " + otherArgs[3]);
      System.out.println("Minimum Value expected: " + minExpected);
      System.out.println("Maximum Value expected: " + maxExpected);


        // Rescale: To convert to binary need to be >0:
        if ( minExpected < 0 ) shift = -minExpected;
        conf.set("shift", String.valueOf(shift));

        // if normalized need to rescale to a bigger factor (default 10):
        if ( maxExpected <= 1 )   scaleFactor = (int)(100/maxExpected);
        if ( maxExpected > 1000 ) scaleFactor = 1;
        conf.set("scaleFactor", String.valueOf(scaleFactor));

        maxExpected = (maxExpected + shift) * scaleFactor;
        int bitStringSize = Long.toBinaryString(maxExpected).length();

        bitFormat = "%0" + bitStringSize + "d";
        conf.set("bitFormat", bitFormat);
        if ( precision > bitStringSize )
                // precision can be at most the size of bitStringSize:
                    conf.set("precision", String.valueOf(bitStringSize));

          System.out.println("Parameter shift = " + shift);
          System.out.println("Parameter scaleFactor = " + scaleFactor);
          System.out.println("Parameter bitFormat: " + bitFormat)
    }


  /**
   *  The CurioD Mapper 1 for MapReduce (data space partitioning).
   */
  public static class CellsCountMap
                              extends Mapper<Object, Text, Text, IntWritable> {

      @Override
      protected void setup(Context context) {
          try {
                        Configuration conf = context.getConfiguration();
                        precision = Integer.valueOf(conf.get("precision"));
                        scaleFactor = Integer.valueOf(conf.get("scaleFactor"));
                        shift = Long.valueOf(conf.get("shift"));
                        bitFormat = conf.get("bitFormat");

                        System.out.println("Precision set to " + precision);
                        System.out.println("scaleFactor set to " + scaleFactor);

                } catch (NumberFormatException e) {
                  //outputs when parameters are not defined by user (use defaults)
                            e.printStackTrace();
                }
    }

    private final static IntWritable one = new IntWritable(1);
    private Text index = new Text();

    @Override
    public void map(Object key, Text entry, Context context)
                                    throws IOException, InterruptedException {

            index.set(Indexing.calcIndex(entry, precision, scaleFactor, shift,
                                                              bitFormat));
            context.write(index, one);
     }

 }


 /**
  * The CurioD reducer 1 class (data space partitioning)
  */
  public static class CellsCountReduce
                          extends Reducer<Text, IntWritable, Text, IntWritable> {

      private IntWritable count = new IntWritable();

      @Override
      public void reduce(Text index, Iterable<IntWritable> values, Context context)
                                    throws IOException, InterruptedException {

                // count values in each cell:
                int sum = 0;
                do {
                    sum++;
                    values.iterator().next();
            } while (values.iterator().hasNext());

            // writes count in all Cells:
            count.set(sum);
            context.write(index, count);
        }
}


/**
 *  The CurioD Mapper 2 for Validate All Nearest Neighbors
 *      of each potencial Outlier Cell.
 */
public static class ValidateNeighborsMap
                             extends Mapper<Object, Text, Text, IntWritable> {

@Override
protected void setup(Context context) {
    try {
            Configuration conf = context.getConfiguration();
            precision = Integer.valueOf(conf.get("precision"));
      } catch (NumberFormatException e) {
           //outputs when parameters are not defined by user (use defaults)
               e.printStackTrace();
      }
}

private final IntWritable self = new IntWritable(-1);
private Text indexNN = new Text();
private IntWritable count = new IntWritable();
private int dims = 0;
private String stFormat;

@Override
public void map(Object key, Text line, Context context)
                                    throws IOException, InterruptedException {

       // get the values on input line (token[0]=index, token[1]=instance):
        String[] token = line.toString().split("\t");
        if ( dims == 0 ) {
                dims = token[0].length()/precision;
                stFormat = "%0" + dims + "d";
        }

        final String[][] nns =
                         Indexing.neighborsIndexs(token[0], precision, dims);

        count.set(Integer.valueOf(token[1])); // assign ONLY one time!

        // get a list of All NearNeighbors for each key/index:
        for (int i = 0; i < Math.pow(3, dims); i++) {
              // Next NearestNeighbours Index:
              String indNN = "";
              String nni = String.format(stFormat,
                                    Long.parseLong(Integer.toString(i,3)));

              for ( int d = 0; d < dims; d++ )
                      indNN += nns[Integer.valueOf(nni.substring(d, d+1))][d];

           // Neighbors of frontiers cells have bad index, but don't exist,
                 if ( indNN.length() == token[0].length() ){  // so, don't write
                       indexNN.set(indNN);
                       if ( indNN.equals(token[0]) ) {  // NN index same as cell
                           self.set(-count.get());
                           context.write(indexNN, self);
                       }
                       else { // a Nearest Neighbor:
                           context.write(indexNN, count);
                       }
                  }
            }
       }

   }


 /**
  *
  * The CurioD reducer 2  for Validate All Nearest Neighbors
  *       of each potencial Outlier Cell.
  */
  public static class ValidateNeighborsReduce
                      extends Reducer<Text, IntWritable, Text, IntWritable> {

      @Override
      protected void setup(Context context) {
          try {
                  Configuration conf = context.getConfiguration();
                  tolerance = Integer.valueOf(conf.get("tolerance"));
          } catch (NumberFormatException e) {
              //outputs when parameters are not defined by user (use defaults)
                  e.printStackTrace();
          }
      }

      private IntWritable cellSum = new IntWritable();

      @Override
      public void reduce(Text keyIndex, Iterable<IntWritable> counts,
                                                        Context context)
                                 throws IOException, InterruptedException {
              int sum = 0;
              int cellCount = 0;

              do {
                      int count = counts.iterator().next().get();
                      if ( count < 0 )
                          cellCount = -count;
                      else
                          sum += count;
              } while ( sum <= tolerance && cellCount <= tolerance &&
                                                counts.iterator().hasNext() );

              if ( cellCount>0 && cellCount <= tolerance && sum <= tolerance ) {
                      cellSum.set(cellCount);
                      context.write(keyIndex, cellSum);
              }
          }
  }


  /**
   *  The CurioD Mapper 3 class for List Outlier lines.
   */
  public static class ListOutliersMap
                                  extends Mapper<Object, Text, Text, Text> {

      @Override
      protected void setup(Context context) {
          try {
                  Configuration conf = context.getConfiguration();
                  precision = Integer.valueOf(conf.get("precision"));
                  scaleFactor = Integer.valueOf(conf.get("scaleFactor"));
                  shift = Long.valueOf(conf.get("shift"));
                  bitFormat = conf.get("bitFormat");
                  inputPath = ((FileSplit)context.getInputSplit()).getPath()
                                                        .getParent().toString();
            } catch (NumberFormatException e) {
              //outputs when parameters are not defined by user (use defaults)
                      e.printStackTrace();
            }
      }

      private Text index = new Text();
      private final Text empty = new Text("");

      @Override
      public void map(Object key, Text line, Context context)
                                      throws IOException, InterruptedException {

              if ( inputPath.endsWith("_tmp_ValidateCells") ) {
                      // index:
                      index.set(line.toString().split("\t")[0]);
                      context.write(index, empty);
              }
              else {
                      // instance:
                      String ind = Indexing.calcIndex(line, precision,
                                                 scaleFactor, shift, bitFormat);
                      index.set(ind);
                      context.write(index, line);
              }
      }

  }


  /**
  * The CurioD reducer 3 class for List outlier instances.
  */
  public static class ListOutliersReduce
                                     extends Reducer<Text, Text, Text, Text> {

      @Override
      protected void setup(Context context) {
          try {
                  Configuration conf = context.getConfiguration();
                  tolerance = Integer.valueOf(conf.get("tolerance"));

          } catch (NumberFormatException e) {
                  e.printStackTrace();
          }
  }
```

```
@Override
public void reduce(Text index, Iterable<Text> entries, Context context)
                                   throws IOException, InterruptedException {

        boolean outlier = false;
        ArrayList<String> instLines = new ArrayList<String>();
        do {
                String entry = entries.iterator().next().toString();
                if ( entry.length() == 0 )
                        outlier = true;
                else
                        // saves line with instances:
                        instLines.add(entry);
        } while ( instLines.size() <= tolerance &&
                                              entries.iterator().hasNext() );

        if ( outlier ) {
            for (String line : instLines) {
                  index.set(line);
                  // Outliers instances (same index on both inputs)
                  context.write(index, null);
            }
        }
    }

}


/**
*
* The driver for CurioD (Distributed version).
*/
@Override
public int run (String[] args) throws Exception {
        if (args.length < 2) {
            System.err.printf(
                 "Basic Usage: %s [generic options] <inputDir> <outputDir>\n",
                                                  getClass().getSimpleName() );
                ToolRunner.printGenericCommandUsage(System.err);
                return -1;
        }

        Configuration conf = new Configuration();
        String[] otherArgs = new
                           GenericOptionsParser(conf, args).getRemainingArgs();
        String tempPath = otherArgs[1] + "_tmp_CellsCounts";
        String tempPath2 = otherArgs[1] + "_tmp_ValidateCells";

        // clean old outputs:
        FileSystem fs =  FileSystem.getLocal(conf);
        fs.delete(new Path(otherArgs[1]), true);
        fs.delete(new Path(tempPath), true);
        fs.delete(new Path(tempPath2), true);

        System.out.println("Input: " + otherArgs[0]);
        System.out.println("temporary dir: " + tempPath);

        // set parameters passed from command line to conf:
        configParameters(conf, otherArgs);
        //NOTE: before pass conf to new jobs, ALL parameters must had been set!

        long start = System.nanoTime();
```

```
// -------------------------------------------------------------------
//  Job1 Partition Data Space:

   Job job1 = new Job(conf, "Curio Algorithm Hadoop version");
   job1.setJarByClass(CurioD.class);

   job1.setMapperClass(CellsCountMap.class);
   job1.setReducerClass(CellsCountReduce.class);

   job1.setOutputKeyClass(Text.class);
   job1.setOutputValueClass(IntWritable.class);

   FileInputFormat.addInputPath(job1, new Path(otherArgs[0]));
   FileOutputFormat.setOutputPath(job1, new Path(tempPath));

 System.out.println(" CURIO: Partitioning/Cells count ...");
   // the job1 (CellsCount):
   int success = (job1.waitForCompletion(true) ? 0 : 1);

 System.out.println("job1: " + (System.nanoTime()-start)/1000000+"ms.");
   long start2 = System.nanoTime();

 // -----------------------------------------------------------------
 //  Job2 Check Neighborhood:

   Job job2 = new Job(conf, "Do a Nearest Neighbor Search.");
   job2.setJarByClass(CurioD.class);

   job2.setMapperClass(ValidateNeighborsMap.class);
   job2.setReducerClass(ValidateNeighborsReduce.class);

   job2.setOutputKeyClass(Text.class);
   job2.setOutputValueClass(IntWritable.class);

   FileInputFormat.addInputPath(job2, new Path(tempPath));
   FileOutputFormat.setOutputPath(job2, new Path(tempPath2));

 System.out.println("CURIO: Validating Potential Outliers (NN search):");
   // the job2 (CheckNeighboorhood)
   success = (job2.waitForCompletion(false) ? 0 : 1);

 System.out.println("job2: " + (System.nanoTime()-start2)/1000000+"ms.");
   start2 = System.nanoTime();

 // -------------------------------------------------------------------
 //  Job3 List Outliers from classified Outliers Cells:

   Job job3 = new Job(conf, "List Outliers from Outliers Cells.");
   job3.setJarByClass(CurioD.class);

   job3.setMapperClass(ListOutliersMap.class);
   job3.setReducerClass(ListOutliersReduce.class);

   job3.setOutputKeyClass(Text.class);
   job3.setOutputValueClass(Text.class);

   FileInputFormat.addInputPath(job3, new Path(tempPath2));
   FileInputFormat.addInputPath(job3, new Path(otherArgs[0]));
   FileOutputFormat.setOutputPath(job3, new Path(otherArgs[1]));
```

```
                System.out.println(" CURIO: List Outliers ...");
                // the job3 (List Outliers)
                success = (job3.waitForCompletion(false) ? 0 : 1);

              System.out.println("job3: "+(System.nanoTime()-start2)/1000000+"ms.");
              System.out.println("All jobs:"+(System.nanoTime()-start)/1000000+"ms.");
              System.out.println("Output Directory: " + otherArgs[1]);

                // return success state:
                return success;

        }



        /**
        *
        * The main entry point.
        */
        public static void main(String[] args) throws Exception {
                // evoque the driver (using ToolRunner):
                int success = ToolRunner.run(new Configuration(), new CurioD(), args);

                System.exit(success);

        }

}
```

D.2 PartitionerD and PartitionerXD

The PartitionerD (PartitionerD.java) code:

```
/**
 * PartitionerD
 *  Use: Driver or Object Oriented approach.
 *   See main() function for examples.
 */

import java.io.IOException;
import java.util.ArrayList;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 *  <b>CURIO</b> algorithm, by Ceglar, Roddick and Powers,
 *      <br>Flinders UNIVERSITY, Adelaide, 2007.
 *  <p>Improved version for <b>Hadoop</b> MapReducing implementation
 *  <br>to distributed systems, of the original algorithm.
 *
 * @author Rui Pais
 * @version 0.1.1
 */
public class PartitionerD extends Configured implements Tool {
/* NOTE: Hadoop require special initialization of variables,
 *  by conf.set() and conf.get(), on setup() implementations
 *  of MapReduces, in order to work on distributed mode.  */

        /** CURIO parameters:  **/
        // partition precision/division (P on article)
        private static int precision;


        /**
         *  The Mapper (1) for Space Partitioning.
         */
        public static class CellsListsMap
                                          extends Mapper<Object, Text, Text, Text> {

                private static int scaleFactor;
                private static long shift;
                private static String bitFormat;  // = "%0" + bitStringSize + "d";
                private static String[] inFiles;
                private String fileIndex = "";

                @Override
        protected void setup(Context context) {
                try {
                        Configuration conf = context.getConfiguration();
                        precision = conf.getInt("precision", 3);
                        scaleFactor = conf.getInt("scaleFactor", 10);
                        shift = conf.getLong("shift", 0);
                        bitFormat = conf.get("bitFormat");
                        // index input files (reduce file size (less I/O) ... )
                        String inputfile = ((FileSplit)context.getInputSplit())
                                                                  .getPath().getName();
                            if ( inFiles == null ) // read only once each file
                                    inFiles = conf.getStrings("inputFiles");
                            for(int i=0; i<inFiles.length && fileIndex.isEmpty(); i++)
                                if ( inFiles[i].equals(inputfile) )
                                    fileIndex = i + ":";  // : -> separator. Format:
                                                          // lineIndex:offsetPosition
                                System.out.println("Precision set to " + precision);
                                System.out.println("input file get: " + inputfile);
                                System.out.println("fileIndex set: " + fileIndex);
                        } catch (NumberFormatException e) {
                        //outputs when parameters aren't defined by user (use defaults)
                                e.printStackTrace();
                        }
        }

                private Text linePosition = new Text();
        private Text index = new Text();

        @Override
        public void map(Object key, Text entry, Context context)
                                    throws IOException, InterruptedException {

                index.set(Indexing.calcIndex(entry, precision, scaleFactor,
                                                            shift, bitFormat));
                linePosition.set(fileIndex + key);
                context.write(index, linePosition);
        }

}


/**
* The reducer (1) for DataSet Space Partitioner (Distributed version).
*/
public static class CellsListsReduce
                                     extends Reducer<Text, Text, Text, Text> {

        private Text instsList = new Text();
        private StringBuilder insts = new StringBuilder();

        @Override
        public void reduce(Text index, Iterable<Text> values, Context context)
                                    throws IOException, InterruptedException {

                // List instances, by offsets file position:
                insts.setLength(0);
                for ( Text value : values )
                        insts.append(value.toString()).append(";");

                // writes file positions of instances in Cell:
                instsList.set(insts.toString());
                context.write(index, instsList);
        }
}


/**
 *  The Mapper (2) for Space Partitioning.
 */
public static class CellsCountsMap
                             extends Mapper<Object, Text, Text, IntWritable> {

        private IntWritable instsCounts = new IntWritable();
        private Text indexCell = new Text();

        @Override
        public void map(Object key, Text entry, Context context)
                                    throws IOException, InterruptedException {

                indexCell.set(entry.toString().substring(0, entry.find("\t")));
                instsCounts.set(entry.toString().split(";").length);
                        context.write(indexCell, instsCounts);
        }

}


/**
```

```
 *  The Mapper (1) for NearNeighbors Listing.
 */
public static class CellsListNeighborsMap
                             extends Mapper<Object, Text, IntWritable, Text> {

        private Text index = new Text();
        private final IntWritable k = new IntWritable(1);

        @Override
        public void map(Object key, Text entry, Context context)
                                    throws IOException, InterruptedException {

                index.set(entry.toString().substring(0, entry.find("\t")));
                context.write(k, index);
        }
}


/**
* The reducer (1) for List NearNeighbors (Distributed version).
*/
public static class CellsListNeighborsReduce
                              extends Reducer<IntWritable, Text, Text, Text> {

        @Override
        protected void setup(Context context) {
        try {
                        Configuration conf = context.getConfiguration();
                        precision = conf.getInt("precision", 3);
                } catch (NumberFormatException e) {
               //outputs when parameters aren't defined by user (use defaults)
                        e.printStackTrace();
                }
        }

        private Text index = new Text();
        private Text indNNList = new Text();
        private ArrayList<String> inds = new ArrayList<String>();
        private ArrayList<StringBuilder>indsNN =
                                new ArrayList<StringBuilder>();

        @Override
        public void reduce(IntWritable key, Iterable<Text> entries,
                                                          Context context)
                                    throws IOException, InterruptedException {

                // a unique key with all indexes of dataset:
                for ( Text entry : entries ) {
                        // write on arrayList:
                        inds.add(entry.toString());
                        indsNN.add(new StringBuilder());
                }

                // Assign the Near Neighbors to each cell:
                // due to neighborhood being symmetric,
                // check only 1/2(C^2-C), not C^2
                for (int i = 0; i < inds.size(); i++) {
                    for (int j = i + 1; j < inds.size(); j++) {
                        // Reduce the number of isNeighbours() call to half!!
                        if (Indexing.isNeighbor(inds.get(i),inds.get(j),
                                                            precision)) {
                                indsNN.get(i).append(inds.get(j) + " ");
                                indsNN.get(j).append(inds.get(i) + " ");
                                }
                        }
                        index.set(inds.get(i));
                        //// write the counting of the list:
                        long cnt = 0;
                        if ( indsNN.get(i).length() > 0 )
                            cnt = indsNN.get(i).toString().split(" ").length;
                        // write the list and they count:
                        indNNList.set(cnt + ":\t" + indsNN.get(i).toString());
                                context.write(index, indNNList);
                }
        }
}



/**
* The driver for Partitioner (Distributed version).
*/
@Override
    public int run (String[] args) throws Exception {
            if (args.length < 2) {
                    System.err.printf(
                      "Basic Usage: %s [generic options] <inputDir> <outputDir>\n",
                                                 getClass().getSimpleName() );
                    ToolRunner.printGenericCommandUsage(System.err);
                    return -1;
            }

            Configuration conf = new Configuration();
            String[] otherArgs = new
                                GenericOptionsParser(conf, args).getRemainingArgs();
            String tempPath = otherArgs[1] + "_tmp_CellsCounts";
            String tempPath2 = otherArgs[1] + "_CellsListNeighbors";

            // clean old outputs:
            FileSystem fs =  FileSystem.getLocal(conf);
            fs.delete(new Path(otherArgs[1]), true);
            fs.delete(new Path(tempPath), true);
            fs.delete(new Path(tempPath2), true);

            // set parameters passed from command line to conf:
            CurioXD.confSetParameters(conf, otherArgs);
            //NOTE: before passing conf to new jobs ALL parameter must been set!

            /** JOB 1: Partitioning and instances listing by cells **/

            long start = System.nanoTime();
            Job job1 = new Job(conf,
                                      "Curio Partitioner: Cells Instances Listing.");
            job1.setJarByClass(PartitionerD.class);

            job1.setMapperClass(CellsListsMap.class);
            job1.setReducerClass(CellsListsReduce.class);

            job1.setOutputKeyClass(Text.class);
            job1.setOutputValueClass(Text.class);

            FileInputFormat.addInputPath(job1, new Path(otherArgs[0]));
            FileOutputFormat.setOutputPath(job1, new Path(otherArgs[1]));
```

```
                System.out.println(
                                 " CURIO Partitioner: listing instances by Cells...");

                // the job1 (CellsListing):
                int success = (job1.waitForCompletion(true) ? 0 : 1);

               System.out.println("job1: "+(System.nanoTime()-start)/1000000 + "ms.");


                start = System.nanoTime();
                Job job3 = new Job(conf,
                                      "Curio Partitioner: Cells Neighbors Listing.");
                job3.setJarByClass(PartitionerD.class);

                job3.setMapperClass(CellsListNeighborsMap.class);
                job3.setReducerClass(CellsListNeighborsReduce.class);

                job3.setOutputKeyClass(IntWritable.class);
                job3.setOutputValueClass(Text.class);

                FileInputFormat.addInputPath(job3, new Path(otherArgs[1]));
                FileOutputFormat.setOutputPath(job3, new Path(tempPath2));

               System.out.println(" CURIO: Cells Near Neighbors listing ...");
                // the job3 (NNCellsListing):
                success = (job3.waitForCompletion(true) ? 0 : 1);

               System.out.println("job3: "+(System.nanoTime()-start)/1000000 + "ms.");

               System.out.println("Output Directory: " + otherArgs[1]);
               System.out.println("Output Directory: " + tempPath);
               System.out.println("Output Directory: " + tempPath2);

                // return success state:
                return success;

        }



        /**
        * The main entry point.
        */
        public static void main(String[] args) throws Exception {
            // the job:
           int success = ToolRunner.run(new Configuration(), new PartitionerD(), args);

             System.out.println(
                   "- Get the list of cells and the indexes of they neighbors: \n" +
                   "hadoop fs -getmerge output_CellsListNeighbors cellsListNeighbors");

                System.exit(success);

        }

}
```

and the PartitionerXD (PartitionerXD.java) code:

```java
/**
 * PartitionerXD
 *  Use: Driver or Object Oriented approach.
 *   See main() function for examples.
 */

import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 *  Partitioner (Distributed and improved version) for
 *  <b>CURIO</b> algorithm (part I), by Ceglar, Roddick and Powers,
 *      <br>Flinders UNIVERSITY, Adelaide, 2007.
 *
 *  <p>Improved version for <b>Hadoop</b> MapReducing implementation
 *  <br>to distributed systems, of the original CURIO algorithm.
 *
 * @author Rui Pais
 * @version 0.1.2
 */

public class PartitionerXD extends Configured implements Tool {
/* NOTE: Hadoop require special initialization of variables,
 *  by conf.set() and conf.get(), on setup() implementations
 *  of MapReduces, in order to work on distributed mode.  */
        /** CURIO parameters:  **/
        // partition precision/division (P on article)
        private static int precision;   // 3 it's the minimum reasonable
        // tolerance (T on article):    // the number of instances (counts)
        private static int tolerance;   // counts to be a potential Outlier

        /** private members: **/
        private static String[] inFiles;


        /**
         *  The Mapper for (1) Space Partitioning.
         *  Counts how many instances are in each cell
         *  if #i > T -> writes the count, #i, of instances
         *    else    -> writes a list of instance's offset references
         */
        public static class CellsCountMap
                                          extends Mapper<Object, Text, Text, Text> {
```

```
        private static int scaleFactor;
        private static long shift;
        private static String bitFormat; // = "%0" + bitStringSize + "d";
        private String fileIndex = "";

@Override
protected void setup(Context context) {
        try {
                    Configuration conf = context.getConfiguration();
                    precision = conf.getInt("precision", 3);
                    scaleFactor = conf.getInt("scaleFactor", 10);
                    shift = conf.getLong("shift", 0);
                    bitFormat = conf.get("bitFormat");
                    // index input files (reduce file size (less I/O) ... )
                    String inputfile = ((FileSplit)context.getInputSplit())
                                                     .getPath().getName();
                    if ( inFiles == null ) // read only once each file
                            inFiles = conf.getStrings("inputFiles");
                    for(int i=0; i<inFiles.length && fileIndex.isEmpty(); i++)
                        if ( inFiles[i].equals(inputfile) )
                                fileIndex = i + ":";

                        /** DEBUG **/
                        System.out.println("Precision set to " + precision);
                        System.out.println("input file get: " + inputfile);
                        System.out.println("fileIndex set: " + fileIndex);
                } catch (NumberFormatException e) {
                //outputs when parameters aren't defined by user (use defaults)

                        e.printStackTrace();
                }
}

private Text lineReference = new Text();
private Text index = new Text();

@Override
public void map(Object key, Text entry, Context context)
                                    throws IOException, InterruptedException {

            index.set(Indexing.calcIndex(entry, precision, scaleFactor,
                                                    shift, bitFormat));
            lineReference.set(fileIndex + key);
            context.write(index, lineReference);
    }

}

/**
 * The Combiner for (1) Space Partitioning.
 */
public static class CellsCountCombine
                                   extends Reducer<Text, Text, Text, Text> {

  @Override
  protected void setup(Context context) {
  try {
                    Configuration conf = context.getConfiguration();
                    tolerance = Integer.valueOf(conf.get("tolerance"));
        } catch (NumberFormatException e) {
        //outputs when parameters aren't defined by user (use defaults)
```

```
                    e.printStackTrace();
            }
        }

        /**
         *  Counts how many instances are in each cell
         *  if #i > T -> writes the count, #i, of instances
         *    else    -> writes a list of instance's offset references
         **/
        private Text linesList = new Text();
        private StringBuilder lines = new StringBuilder();

        @Override
        public void reduce(Text index, Iterable<Text> values, Context context)
                                        throws IOException, InterruptedException {

                // List instances, by line numbers, in each cell:
                lines.setLength(0);
                int count = 0;
                for (Text value : values) { // can be count, lineRef or a List
                    if ( value.find(":", 1) > 0 ) {
                        // lineRef or List (separated by ;)
                        count += Indexing.countOccurrences(':', value);
                        // check if it worth add to list
                        if ( count <= tolerance ) {
                                    lines.append(value);
                                        if ( !lines.toString().endsWith(";") )
                                                lines.append(";");
                                }
                        }
                        else {
                                count += Integer.parseInt(value.toString());
                        }
                }

                // writes count in all Cells:
                if ( count > tolerance )        //  Save only counts:
                        linesList.set(String.valueOf(count));
                else // Potential Outliers:     //   Save all info,
                        linesList.set(lines.toString());  // needed if an Outlier

                context.write(index, linesList);
            }
    }



    /**
    * The Reducer for (1) Space Partitioning.
    */
    public static class CellsCountReduce
                                        extends Reducer<Text, Text, Text, Text> {
      @Override
      protected void setup(Context context) {
      try {
                    Configuration conf = context.getConfiguration();
                    tolerance = Integer.valueOf(conf.get("tolerance"));
            } catch (NumberFormatException e) {
            //outputs when parameters are not defined by user (use defaults)
                    e.printStackTrace();
            }
        }
        /**
         *  Counts how many instances are in each cell
         *  if #i > T -> writes the count, #i, of instances
         *    else    -> writes a list of instance's offset references
         **/
        private Text linesList = new Text();
        private StringBuilder lines = new StringBuilder();

        @Override
        public void reduce(Text index, Iterable<Text> values, Context context)
                                        throws IOException, InterruptedException {

                // List instances, by line numbers, in each cell:
                lines.setLength(0);
                int count = 0;
                for ( Text value : values ) {
                       if ( value.find(":", 1) > 0 ) {
                                count += Indexing.countOccurrences(':', value);
                                // check if it worth to add to list:
                                if ( count <= tolerance )
                                            lines.append(value);
                        }
                        else  {
                                count += Integer.valueOf(value.toString());
                        }
                    }

                // writes count in all Cells:
                if ( count > tolerance )  // save only counts:
                    linesList.set(String.valueOf(count)); // don't need more info
                else  // Potential Outliers:     //    Save all info,
                    linesList.set(lines.toString());  // needed if an Outlier

              context.write(index, linesList);
        }
}


    /**
    * The Reducer (1) for Space Partitioning.
    */
    public static class CellsCountReduceNoCombine
                                      extends Reducer<Text, Text, Text, Text> {
        @Override
        protected void setup(Context context) {
        try {
                        Configuration conf = context.getConfiguration();
                        tolerance = Integer.valueOf(conf.get("tolerance"));
              } catch (NumberFormatException e) {
              //outputs when parameters are not defined by user (use defaults)
                            e.printStackTrace();
              }
          }

          private Text instsList = new Text();
          private StringBuilder insts = new StringBuilder();

          @Override
          public void reduce(Text index, Iterable<Text> values, Context context)
                                        throws IOException, InterruptedException {
```

```
            // List instances, by offset file position:
            insts.setLength(0);
            int count = 0;
            for ( Text value : values ) {
                count++;
                if ( count <= tolerance ) { // check if it worth to add to list
                        insts.append(value);
                        if ( !insts.toString().endsWith(";") )
                                insts.append(";");
                    }
            }

            // writes count in all Cells:
            if ( count > tolerance )  // save only counts:
                instsList.set(String.valueOf(count));  // don't need more info
            else   // Potential Outliers:     //    Save all info,
                instsList.set(insts.toString());  // needed if an Outlier

            context.write(index, instsList);
        }
}




/**
* The driver for CurioD (Distributed version).
*/
@Override
public int run (String[] args) throws Exception {
        if (args.length < 2) {
                System.err.printf(
               "Basic Usage: %s <inputDir> <outputDir> precision tolerance\n",
                                                 getClass().getSimpleName() );
                return -1;
        }

        Configuration conf = new Configuration();
        String[] otherArgs = new
                           GenericOptionsParser(conf, args).getRemainingArgs();
        // clean old outputs:
        FileSystem fs =  FileSystem.getLocal(conf);
        fs.delete(new Path(otherArgs[1]), true);

        // set parameters passed from command line to conf:
        CurioXD.confSetParameters(conf, otherArgs);
        //NOTE: before passing conf to new jobs ALL parameter must been set!

        /** JOB 1: Partitioning and instances listing by cells **/

        long start = System.nanoTime();
        Job job1 = new Job(conf, "Curio Algorithm Hadoop version");
        job1.setJarByClass(PartitionerXD.class);

        job1.setMapperClass(CellsCountMap.class);
        //EITHER:
        job1.setCombinerClass(CellsCountCombine.class); // Specialized comb
        job1.setReducerClass(CellsCountReduce.class);
        //OR, a single step, only a reducer:
        //job1.setReducerClass(CellsCountReduceNoCombine.class);

        job1.setOutputKeyClass(Text.class);
```

```
                job1.setOutputValueClass(Text.class);

                job1.setOutputFormatClass(SequenceFileOutputFormat.class);

                FileInputFormat.addInputPath(job1, new Path(otherArgs[0]));
                FileOutputFormat.setOutputPath(job1, new Path(otherArgs[1]));

                System.out.println(" CURIO: Partitioning/Cells count ...");
                // the job1 (CellsCount):
                int success = (job1.waitForCompletion(true) ? 0 : 1);

              System.out.println("job1: " + (System.nanoTime()-start)/1000000 + "ms.");

                // Just for performance testings:
                String tempPath = otherArgs[1] + "2";
                fs.delete(new Path(tempPath), true);
                start = System.nanoTime();

                Job job2 = new Job(conf, "Curio Algorithm Hadoop version");
                job2.setJarByClass(PartitionerXD.class);

                job2.setMapperClass(CellsCountMap.class);
                //EITHER:
                //job2.setCombinerClass(CellsCountCombine.class); // Specialized comb
                //job2.setReducerClass(CellsCountReduce.class);
                //OR, a single step, only a reducer:
                job2.setReducerClass(CellsCountReduceNoCombine.class);

                job2.setOutputKeyClass(Text.class);
                job2.setOutputValueClass(Text.class);

                job2.setOutputFormatClass(SequenceFileOutputFormat.class);

                FileInputFormat.addInputPath(job2, new Path(otherArgs[0]));
                FileOutputFormat.setOutputPath(job2, new Path(tempPath));

                System.out.println(" CURIO: Partitioning/Cells count ...");
                // the job1 (CellsCount):
                success = (job2.waitForCompletion(true) ? 0 : 1);

              System.out.println("job2: " + (System.nanoTime()-start)/1000000 + "ms.");
              System.out.println("Output Directory: " + tempPath);

                // return success state:
                return success;

        }



        /**
        * The main entry point.
        */
        public static void main(String[] args) throws Exception {
                // the job:
                int success =
                       ToolRunner.run(new Configuration(), new PartitionerXD(), args);

                System.exit(success);
        }

}
```

D.3 CurioXD

The CurioXD code:

```
/**
 * CurioXD
 *  Use: Driver or Object Oriented approach.
 *   See main() function for examples.
 */

import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 *  <b>CURIO</b> algorithm, by Ceglar, Roddick and Powers,
 *      <br>Flinders UNIVERSITY, Adelaide, 2007.
 *  <p>Improved version for <b>Hadoop</b> MapReducing implementation
 *  <br>to distributed systems, of the original algorithm.
 *
 * @author Rui Pais
 * @version 0.1.0
 */
public class CurioXD extends Configured implements Tool {
 /* NOTE: Hadoop require special initialization of variables,
  *  by conf.set() and conf.get(), on setup() implementations
  *  of MapReduces, in order to work on distributed mode.  */
        //** CURIO parameters:  **//
        // Partition precision/division (P on article)
        private static int precision;   // 3 it's the minimum reasonable
        // tolerance (T on article):    // the number of instances (counts)
        private static int tolerance;   // # on a cell to be a potential Outlier


        /** private members: **/
        private static FSDataInputStream rstream;
        private static FileSystem fs;
private static int lastReaded = -1;

/**
 * Reads a line of a file on HDFS (intended for input directory)
 * on the given path, listed on inFiles array, and indexed at
 * filePosition string, with the format: 'fileIndex:lineOffset'
 *
 * @param inputPath
 * @param inFiles
 * @param filePosition
 * @return
 * @throws IOException
 */
protected static String readLineHDFS(String inputPath, String[] inFiles,
                                     String filePosition) throws IOException {

    StringBuilder sb = new StringBuilder();
    String[] token = filePosition.split(":");
    try {
            if ( token[0].isEmpty() ) return ""; // it can happen!
            int fileIndex = Integer.parseInt(token[0]);
            if ( fileIndex > inFiles.length ) {
                    System.out.print("CHECK");
                        return "";
            }
            if ( lastReaded != fileIndex ) {  // read only if needed:
                    fs = FileSystem.get(new Configuration());
                    rstream = fs.open(new Path(inputPath+inFiles[fileIndex]));
                    lastReaded = fileIndex;
            }
            rstream.seek(Long.parseLong(token[1]));
            int readB;
            while ( (readB = rstream.readByte()) != 0xA )
                sb.append((char)readB);

    } catch (IOException e1) {
            e1.printStackTrace();
            return null;
    }
    return sb.toString();
}

/**
 * Configure the parameters, external and internal, for Curio Algorithm
 * (it will  be used by the other implementations in the class)
 *
 * @param conf
 * @param args
 * @throws Exception
 */
protected static void confSetParameters(Configuration conf, String[] args)
                                                            throws Exception {

       /* NOTE:
        * In distributed mode (global) variables/parameters
        * should be set to configuration by conf.set() method,
        * and read to the global variables on overrided setup()
        * of each class with conf.get() method.
        */
        int scaleFactor = 10; // default, changes on setup of 1st Map
        long shift = 0;           // default, changes only with negatives
        String bitFormat;     // = "%0" + bitStringSize + "d";
        long minExpected = Long.MAX_VALUE;
        long maxExpected = Long.MIN_VALUE;

        if (args.length > 2) {
            conf.set("precision", args[2]);
            if (args.length > 3)
                        conf.set("tolerance", args[3]);

        }
        if ( args.length > 4 ) {
            minExpected = Long.valueOf(args[4]);
            maxExpected = Long.valueOf(args[5]);
        }
        else {
            // Find Extreme Values:
            String[] ioPaths = {args[0], args[1] + "_Extrema_tmp"};
            ExtremaD extrValsD = new ExtremaD(ioPaths);
            minExpected = (long)Math.floor(extrValsD.getMinValue());
            maxExpected = (long)Math.ceil(extrValsD.getMaxValue());
        }

     System.out.println("Minimum Value expected: " + minExpected);
     System.out.println("Maximum Value expected: " + maxExpected);

        // Rescale: To convert to binary need to be >0:
        if ( minExpected < 0 ) shift = -minExpected;
        conf.set("shift", String.valueOf(shift));

        // if normalized need to rescale to a bigger factor (default 10):
        if ( maxExpected <= 1 )   scaleFactor = (int)(100/maxExpected);
        if ( maxExpected > 1000 ) scaleFactor = 1;
        conf.set("scaleFactor", String.valueOf(scaleFactor));

        maxExpected = (maxExpected + shift) * scaleFactor;
        int bitStringSize = Long.toBinaryString(maxExpected).length();

        bitFormat = "%0" + bitStringSize + "d";
        conf.set("bitFormat", bitFormat);
        if ( Integer.parseInt(args[2]) > bitStringSize ) {
            // Precision can be at most the size of bitStringSize:
            conf.set("precision", String.valueOf(bitStringSize));
           System.err.println("Changed parameter Precision to "+bitStringSize);
        }

        System.out.println("Original parameter Precision: " + args[2]);
        System.out.println("         parameter Tolerance: " + args[3]);

        System.out.println("   scaleFactor = " + scaleFactor);
        System.out.println("   shift = " + shift);
        System.out.println("   bitFormat: " + bitFormat);

        // get a list of files on input directory (some versions need them)
        FileSystem fs = FileSystem.get(new Configuration());
        FileStatus[] status = fs.listStatus(new Path(args[0]));
        String input = "";
        for (int i = 0; i < status.length; i++)
            input += status[i].getPath().getName() + ",";
            //must be separated by commas, to be read with getStrings to array
        conf.set("inputFiles", input);
        input = status[0].getPath().getParent().toUri().getRawPath() + "/";
        conf.set("inputPath", input);
}
```

```
/**
 *  The CurioXD Mapper 2 for Validate All Nearest Neighbors
 *      of each potencial Outlier Cell.
 */
public static class ValidateNeighborsMap
                                extends Mapper<Object, Text, Text, Text> {

    @Override
    protected void setup(Context context) {
      try {
                Configuration conf = context.getConfiguration();
                precision = Integer.valueOf(conf.get("precision"));
       } catch (NumberFormatException e) {
            //outputs when parameters are not defined by user (use defaults)
                e.printStackTrace();
       }
     }

     private Text index = new Text();
     private Text count = new Text();
     int dims = 0;
     String stFormat;

     @Override
     public void map(Object key, Text line, Context context)
                                    throws IOException, InterruptedException {

            // get the values on input line:
            String[] token = line.toString().split("\t");
            if ( dims == 0 ) {
                        dims = token[0].length()/precision;
                        stFormat = "%0" + dims + "d";
            }
            final String[][] nns = Indexing.neighborsIndexs(token[0],
                                                         precision, dims);

            // assign ONLY one time!  Will be written several times...
            count.set(String.valueOf(token[1].split(";").length));
            boolean set = false;

            //CAN BE OPTIMIZED?? this is one of the bottle necks of CURIO:
            // -> version XRRD optimize this!
            // get a list of Neighbors for each key/index:
            for (int i = 0; i < Math.pow(3, dims); i++) {
                    // Next NearestNeighbours Index:
                    String indNN = "";
                    String nni = String.format(stFormat,
                                       Long.parseLong(Integer.toString(i,3)));

                    for ( int d = 0; d < dims; d++ )
                       indNN += nns[Integer.valueOf(nni.substring(d, d+1))][d];

                    // Frontier' neighbors cells have bad index,
                    // but they don't exist. So, don't write
                    if ( indNN.length() == token[0].length() ) {
                            index.set(indNN);
                            if ( !set && indNN.equals(token[0]) ) {
                                    context.write(index, new Text(token[1]));
                                    set = true;  //and don't check again
                            }
```

```
                                else
                                        context.write(index, count);
                        }
                }
        }

    }


   /**
    * The CurioXD reducer 2  for Validate All Nearest Neighbors
    *       of each potencial Outlier Cell.
    */
    public static class ValidateNeighborsReduce
                                         extends Reducer<Text, Text, Text, Text> {

        private String inputPath;
        private String[] inFiles;

        @Override
        protected void setup(Context context) {
          try {
                  Configuration conf = context.getConfiguration();
                  tolerance = Integer.valueOf(conf.get("tolerance"));
                  //TODO: implement use of toleranceFactor
                  int toleranceFactor = conf.getInt("toleranceFactor", 1);
                  tolerance *= toleranceFactor; //see Note on reduce function
                  inFiles = conf.getStrings("inputFiles");
                  inputPath = conf.get("inputPath");
            } catch (NumberFormatException e) {
                //outputs when parameters are not defined by user, use defaults
                    e.printStackTrace();
            }
        }

        private Text linesOutliers = new Text();

        @Override
        public void reduce(Text key, Iterable<Text> counts, Context context)
                                        throws IOException, InterruptedException {
                int sumNN = 0;
                String line = "";
                do {
                        String value = counts.iterator().next().toString();
                        if ( line.length() == 0 && value.contains(":") ) {
                            if ( value.split(";").length <= tolerance ) {
                                //System.out.println("a PO Cell!");
                                    line = value;
                            } // else:
                            /*
                             * it's a case of a cell with a count bigger than
                             * the Tolerance, but with a sparse neighborhood,
                             * smaller than the Tolerance (an outlier cluster).
                             * A peculiarity of this version of algorithm!
                             */
                        }
                        else
                                sumNN += Integer.valueOf(value);
                } while ( sumNN <= tolerance && counts.iterator().hasNext() );

                // output a file with the list of Outliers instances:
                if ( line.length() > 0 && sumNN <= tolerance ) {
```

```
                String[] lines = line.split(";");
                for (int i = 0; i < lines.length; i++)
                    if ( !lines[i].isEmpty() ) { // It CAN HAVE empty lines!
                        linesOutliers.set(readLineHDFS(inputPath,
                                                          inFiles, lines[i]));
                        context.write(linesOutliers, null);
                    }
            }
        }

    }



    /**
    * The driver for CurioXD (Distributed version).
    */
    @Override
    public int run (String[] args) throws Exception {
            if (args.length < 2) {
                System.err.printf(
                  "Basic Usage: %s [generic options] <inputDir> <outputDir>\n",
                                                      getClass().getSimpleName() );
                ToolRunner.printGenericCommandUsage(System.err);
                return -1;
            }

            Configuration conf = new Configuration();
            String[] otherArgs = new
                                GenericOptionsParser(conf, args).getRemainingArgs();
            String tempPath = otherArgs[1] + "_tmp_CellsCounts";

            // clean old outputs:
            FileSystem fs =  FileSystem.getLocal(conf);
            fs.delete(new Path(otherArgs[1]), true);
            fs.delete(new Path(tempPath), true);

            // set parameters passed from command line to conf:
            confSetParameters(conf, otherArgs);
            //NOTE: before passing conf to new jobs ALL parameter must been set!

            /** JOB 1: Partitioning and instances listing by cells **/
            long start = System.nanoTime();
            Job job1 = new Job(conf, "CurioD: Partitioning Dataset Space.");
            job1.setJarByClass(CurioXD.class);

            job1.setMapperClass(PartitionerD.CellsListsMap.class);
            job1.setReducerClass(PartitionerD.CellsListsReduce.class);

            job1.setOutputKeyClass(Text.class);
            job1.setOutputValueClass(Text.class);

            FileInputFormat.addInputPath(job1, new Path(otherArgs[0]));
            FileOutputFormat.setOutputPath(job1, new Path(tempPath));

          System.out.println(" CURIO: Partitioning/Cells count ...");
            // the job1 (CellsCount):
            int success = (job1.waitForCompletion(true) ? 0 : 1);

          System.out.println("job1: " + (System.nanoTime()-start)/1000000 +"ms.");
            long start2 = System.nanoTime();
```

```
                /**  Job2 Check Neighborhood:   **/

                Job job2 = new Job(conf, " CurioD: Nearest Neighbors Sum Validation.");
                job2.setJarByClass(CurioXD.class);

                job2.setMapperClass(ValidateNeighborsMap.class);
                job2.setReducerClass(ValidateNeighborsReduce.class);

                job2.setOutputKeyClass(Text.class);
                job2.setOutputValueClass(Text.class);

                FileInputFormat.addInputPath(job2, new Path(tempPath));
                FileOutputFormat.setOutputPath(job2, new Path(otherArgs[1]));

              System.out.println(" CURIO: Validating Potential Outliers (NN search)");
                // the job2 (CheckNeighboorhood):
                success = (job2.waitForCompletion(false) ? 0 : 1);

              System.out.println("job2: "+ (System.nanoTime()-start2)/1000000 +"ms.");
              System.out.println(">All jobs: "
                                             + (System.nanoTime()-start)/1000000+"ms.");
              System.out.println("Output Directory: " + otherArgs[1]);

                // return success state:
                return success;

        }



        /**
        * The main entry point.
        */
        public static void main(String[] args) throws Exception {
            // the job:
            int success = ToolRunner.run(new Configuration(), new CurioXD(), args);

                System.exit(success);

        }

}
```

D.4 CurioXXD

The CurioXXD code:

```
/**
 * CurioXXD
 *  Use: Driver or Object Oriented approach.
 *   See main() function for examples.
 */

import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 *  <b>CURIO</b> algorithm, by Ceglar, Roddick and Powers,
 *      <br>Flinders UNIVERSITY, Adelaide, 2007.
 *  <p>Improved version for <b>Hadoop</b> MapReducing implementation
 *  <br>to distributed systems, of the original algorithm.
 *
 * @author Rui Pais
 * @version 0.1.2
 */
public class CurioXXD extends Configured implements Tool {
 /* NOTE: Hadoop require special initialization of variables,
  *  by conf.set() and conf.get(), on setup() implementations
  *  of MapReduces, in order to work on distributed mode.  */
        //** CURIO parameters:  **//
        // partition precision/division (P on article)
        private static int precision;   // 3 it's the minimum reasonable
        // tolerance (T on article):    // the number of instances (counts)
        private static int tolerance;   // # on a cell to be a potential Outlier

        /** private members: **/
        private static String[] inFiles;


        /**
         *  The CurioXXD Mapper (2) for Validate All Nearest Neighbors
         *      of each potencial Outlier Cell.
         */
        public static class ValidateNeighborsMap
                                     extends Mapper<Text, Text, Text, Text> {

            @Override
            protected void setup(Context context) {
              try {
                        Configuration conf = context.getConfiguration();
                        precision = Integer.valueOf(conf.get("precision"));
              } catch (NumberFormatException e) {
                      //outputs when parameters are not defined by user (use defaults)
                        e.printStackTrace();
              }
            }

            private Text index = new Text();
            private Text count = new Text();
            private int dims = 0;
```

```
    private String stFormat;

    @Override
    public void map(Text key, Text line, Context context)
                                throws IOException, InterruptedException {

            if ( dims == 0 ) { // define only once when mapping:
                    dims = key.getLength()/precision;
                    stFormat = "%0" + dims + "d";
            }

            // Set Only Once, will be written many times:
            if ( line.find(";", 1) > 0 )
                    // It's a PO <=> contains a list:
                    count.set(Indexing.occurrencesCount(';', line));
            else    // Not a PO <=> contains a count:
                    count.set(line);

            final String[][] nns = Indexing.neighborsIndexs(key.toString(),
                                                         precision, dims);
            boolean set = false;

            // get a list of Neighbors for each key/index:
            for (int i = 0; i < Math.pow(3, dims); i++) {
                // Next NearestNeighbours Index:
                String indNN = "";
                String nni = String.format(stFormat,
                                     Long.parseLong(Integer.toString(i,3)));

                for ( int d = 0; d < dims; d++ )
                    indNN += nns[Integer.valueOf(nni.substring(d, d+1))][d];
                    //Neighbors of frontier's cells have bad index,
                    // but don't exist,
                    if ( indNN.length() == key.getLength() ) {//so, don't write
                        if ( !set && indNN.equals(key.toString()) ) {
                                context.write(key, line);  // lineNumbers list
                                set = true;
                        }
                        else {
                                index.set(indNN);
                                context.write(index, count);
                        }
                 }
            }
    }

}


/**
* The reducer 2  for Validate All Nearest Neighbors
*       of each potencial Outlier Cell.
*/
public static class ValidateNeighborsReduce
                                extends Reducer<Text, Text, Text, Text> {

    private String inputPath;

    @Override
    protected void setup(Context context) {
        try {
                Configuration conf = context.getConfiguration();
```

```
                    tolerance = Integer.valueOf(conf.get("tolerance"));
                    inFiles = conf.getStrings("inputFiles");
                    inputPath = conf.get("inputPath");
            } catch (NumberFormatException e) {
                    //outputs when parameters are not defined by user, use defaults
                    e.printStackTrace();
            }
        }

        private Text lineOutlier = new Text();

        @Override
        public void reduce(Text key, Iterable<Text> counts, Context context)
                                        throws IOException, InterruptedException {
                long sumNN = 0;
                String line = "";
                do {
                        String value = counts.iterator().next().toString();
                        if ( line.isEmpty() && value.endsWith(";") )
                                line = value;
                        else
                                sumNN += Integer.valueOf(value);
                } while ( sumNN <= tolerance && counts.iterator().hasNext() );

                // output the list of Outliers instances:
                if ( line.length() > 0 && sumNN <= tolerance ) {
                         for (String ln : line.split(";")) {
                            if ( !ln.isEmpty() ) {  // It CAN HAVE empty lines!
                                  lineOutlier.set(
                                     CurioXD.readLineHDFS(inputPath, inFiles, ln));
                                  context.write(lineOutlier, null);
                            }
                        }
                }
        }

    }



    /**
    * The driver for CurioXXD (Distributed version).
    */
    @Override
    public int run (String[] args) throws Exception {
            if (args.length < 2) {
                System.err.printf(
                   "Basic Usage: %s <inputDir> <outputDir> precision tolerance\n",
                                                   getClass().getSimpleName() );
                return -1;
            }

            Configuration conf = new Configuration();
            String[] otherArgs = new
                              GenericOptionsParser(conf, args).getRemainingArgs();
            String tempPath = otherArgs[1] + "_tmp_CellsCounts";

            // clean old outputs:
            FileSystem fs =  FileSystem.getLocal(conf);
            fs.delete(new Path(otherArgs[1]), true);
            fs.delete(new Path(tempPath), true);
        System.out.println("Input: " + otherArgs[0]);
        System.out.println("temporary dir: " + tempPath);

        // set parameters passed from command line to conf:
        CurioXD.confSetParameters(conf, otherArgs);
        //NOTE: before passing conf to new jobs ALL parameter must been set!

        /** JOB 1: Partitioning and instances listing by cells **/

        long start = System.nanoTime();
        Job job1 = new Job(conf, "Curio Algorithm Hadoop version");
        job1.setJarByClass(CurioXXD.class);

        job1.setMapperClass(PartitionerXD.CellsCountMap.class);
        job1.setReducerClass(PartitionerXD.CellsCountReduceNoCombine.class);

        job1.setOutputKeyClass(Text.class);
        job1.setOutputValueClass(Text.class);

        job1.setOutputFormatClass(SequenceFileOutputFormat.class);

        FileInputFormat.addInputPath(job1, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job1, new Path(tempPath));

        System.out.println(" CURIO: Partitioning/Cells count ...");
        // the job1 (CellsCount):
        int success = (job1.waitForCompletion(true) ? 0 : 1);


      System.out.println("job1: "+ (System.nanoTime()-start)/1000000 + "ms.");
        long start2 = System.nanoTime();

        /**  Job2 Check Neighborhood:   **/

        Job job2 = new Job(conf, "Do a Nearest Neighbor Search.");
        job2.setJarByClass(CurioXXD.class);

        job2.setMapperClass(ValidateNeighborsMap.class);
        job2.setReducerClass(ValidateNeighborsReduce.class);

        job2.setInputFormatClass(SequenceFileInputFormat.class);

        job2.setOutputKeyClass(Text.class);
        job2.setOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job2, new Path(tempPath));
        FileOutputFormat.setOutputPath(job2, new Path(otherArgs[1]));

      System.out.println(" CURIO: Validating Potential Outliers (NN search)");
        // Job2 (CheckNeighboorhood):
        success = (job2.waitForCompletion(false) ? 0 : 1);

      System.out.println("job2: " + (System.nanoTime()-start2)/1000000+"ms.");
      System.out.println("> All jobs: "
                             + (System.nanoTime()-start)/1000000+"ms.");
      System.out.println("Output Directory: " + otherArgs[1]);

        // return success state:
        return success;
}
```

```
        /**
        * The main entry point.
        */
        public static void main(String[] args) throws Exception {
            // the job:
            int success = ToolRunner.run(new Configuration(), new CurioXXD(), args);

                System.exit(success);

      }

}
```

D.5 CurioXXcD

The code for the CurioXXcD variant:

```
/**
 * CurioXXcD
 *  Use: Driver or Object Oriented approach.
 *   See main() function for examples.
 */

import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 *  <b>CURIO</b> algorithm, by Ceglar, Roddick and Powers,
 *      <br>Flinders UNIVERSITY, Adelaide, 2007.
 *  <p>Improved version for <b>Hadoop</b> MapReducing implementation
 *  <br>to distributed systems, of the original algorithm.
 *
 * @author Rui Pais
 * @version 0.1.2
 */
public class CurioXXcD extends Configured implements Tool {
```

```
/* NOTE: Hadoop require special initialization of variables,
 *  by conf.set() and conf.get(), on setup() implementations
 *  of MapReduces, in order to work on distributed mode.  */
       //** CURIO parameters:  **//
       // partition precision/division (P on article)
       private static int precision;   // 3 it's the minimum reasonable
       // tolerance (T on article):    // the number of instances (counts)
       private static int tolerance;   // # on a cell to be a potential Outlier

       /** private members: **/
       private static String[] inFiles;



       /**
        *  The Mapper (2) for Validate All Nearest Neighbors
        *      of each potencial Outlier Cell.
        */
       public static class ValidateNeighborsMap
                                    extends Mapper<Text, Text, Text, Text> {

           @Override
           protected void setup(Context context) {
               try {
                       Configuration conf = context.getConfiguration();
                       precision = Integer.valueOf(conf.get("precision"));
               } catch (NumberFormatException e) {
                     //outputs when parameters are not defined by user (use defaults)
                       e.printStackTrace();
               }
           }

           private Text index = new Text();
           private Text count = new Text();
           private int dims = 0;
           private String stFormat;

           @Override
           public void map(Text key, Text line, Context context)
                                       throws IOException, InterruptedException {

                   if ( dims == 0 ) {  // define only once when mapping:
                            dims = key.getLength()/precision;
                           stFormat = "%0" + dims + "d";
                   }

                   // Set Only Once, will be written many times:
                   if ( line.find(";", 1) > 0 )
                           // It's a PO <=> contains a list:
                           count.set(Indexing.occurrencesCount(';', line));
                   else    // Not a PO <=> contains a count:
                           count.set(line);

                   final String[][] nns = Indexing.neighborsIndexs(key.toString(),
                                                                precision, dims);
                   boolean set = false;

                   // get a list of Neighbors for each key/index:
                   for (int i = 0; i < Math.pow(3, dims); i++) {
                        // Next NearestNeighbours Index:
                       String indNN = "";
                       String nni = String.format(stFormat,
                                            Long.parseLong(Integer.toString(i,3)));

                        for ( int d = 0; d < dims; d++ )
                            indNN += nns[Integer.valueOf(nni.substring(d, d+1))][d];
                         //Neighbors of frontier cells have bad index, but don't exist,
                            if ( indNN.length() == key.getLength() ) {//so, don't write

                                if ( !set && indNN.equals(key.toString()) ) {
                                        context.write(key, line);   // lineNumbers list
                                        set = true;
                                } else {
                                        index.set(indNN);
                                        context.write(index, count);
                                }
                            }
                    }
            }

    }


   /**
    * The combiner 2 for Validate All Nearest Neighbors
    *       of each potential Outlier Cell.
    */
    public static class ValidateNeighborsCombine
                                    extends Reducer<Text, Text, Text, Text> {

        private Text cellData = new Text();

        @Override
        public void reduce(Text key, Iterable<Text> counts, Context context)
                                    throws IOException, InterruptedException {
              long sumNN = 0;
              String line = "";
              do {
                      String value = counts.iterator().next().toString();
                      if ( line.isEmpty() && value.endsWith(";") )
                            line = value; // line unique, samples list in a PO cell
                      else
                            sumNN += Integer.valueOf(value);  // sum of NN counts
                } while ( counts.iterator().hasNext() );

                if ( sumNN > 0 ) { // isolated cell, don't need to write 0 ...
                      // output the sum of neighbors counts:
                      cellData.set(String.valueOf(sumNN));
                      context.write(key, cellData);       // reducer by exit if > T
                }
                // and the list of offset lines:
                if ( !line.isEmpty() ) { // Potential Outlier
                      cellData.set(line);
                      context.write(key, cellData);
                }
         }

    }


   /**
    * The reducer 2  for Validate All Nearest Neighbors
    *       of each potencial Outlier Cell.
    */
```

```java
public static class ValidateNeighborsReduce
                                     extends Reducer<Text, Text, Text, Text> {

    private String inputPath;

    @Override
    protected void setup(Context context) {
        try {
                Configuration conf = context.getConfiguration();
                tolerance = Integer.valueOf(conf.get("tolerance"));
                inFiles = conf.getStrings("inputFiles");
                inputPath = conf.get("inputPath");
        } catch (NumberFormatException e) {
               //outputs when parameters are not defined by user (use defaults)

                e.printStackTrace();
        }
    }

    private Text lineOutlier = new Text();

    @Override
    public void reduce(Text key, Iterable<Text> counts, Context context)
                                    throws IOException, InterruptedException {
          long sumNN = 0;
          String line = "";
          do {
                String value = counts.iterator().next().toString();
                if ( line.isEmpty() && value.endsWith(";") )
                        line = value;
                else
                        sumNN += Integer.valueOf(value);
          } while ( sumNN <= tolerance && counts.iterator().hasNext() );

          // output the list of Outliers instances:
          if ( line.length() > 0 && sumNN <= tolerance ) {
                for ( String ln : line.split(";") )
                    if ( !ln.isEmpty() ) {  // It CAN HAVE empty lines!
                        lineOutlier.set(
                                CurioXD.readLineHDFS(inputPath, inFiles, ln));
                        context.write(lineOutlier, null);
                    }
            }
    }

}



/**
* The driver for CurioXXcD (Distributed version).
*/
@Override
public int run (String[] args) throws Exception {
        if (args.length < 2) {
            System.err.printf(
               "Basic Usage: %s <inputDir> <outputDir> precision tolerance\n",
                                                   getClass().getSimpleName() );
                return -1;
        }

        Configuration conf = new Configuration();
```

```java
  String[] otherArgs = new
                  GenericOptionsParser(conf, args).getRemainingArgs();
  String tempPath = otherArgs[1] + "_tmp_CellsCounts";

  // clean old outputs:
  FileSystem fs =  FileSystem.getLocal(conf);
  fs.delete(new Path(otherArgs[1]), true);

  // set parameters passed from command line to conf:
  CurioXD.confSetParameters(conf, otherArgs);
  //NOTE: before passing conf to new jobs ALL parameter must been set!

  /** JOB 1: Partitioning and instances listing by cells **/

  long start = System.nanoTime();
  Job job1 = new Job(conf, "Curio Algorithm Hadoop version");
  job1.setJarByClass(CurioXXcD.class);

  job1.setMapperClass(PartitionerXD.CellsCountMap.class);
  job1.setCombinerClass(PartitionerXD.CellsCountCombine.class);
  job1.setReducerClass(PartitionerXD.CellsCountReduce.class);

  job1.setOutputKeyClass(Text.class);
  job1.setOutputValueClass(Text.class);

  job1.setOutputFormatClass(SequenceFileOutputFormat.class);

  FileInputFormat.addInputPath(job1, new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job1, new Path(tempPath));

  System.out.println(" CURIO: Partitioning/Cells count ...");
  // the job1 (CellsCount):
  int success = (job1.waitForCompletion(true) ? 0 : 1);

System.out.println("job1: " + (System.nanoTime()-start)/1000000+ "ms.");
  long start2 = System.nanoTime();

  /**  Job2 Check Neighborhood:   **/

  Job job2 = new Job(conf, "Do a Nearest Neighbor Search.");
  job2.setJarByClass(CurioXXcD.class);

  job2.setMapperClass(ValidateNeighborsMap.class);
  job2.setCombinerClass(ValidateNeighborsCombine.class);  //CHECKED: OK!
  job2.setReducerClass(ValidateNeighborsReduce.class);

  job2.setInputFormatClass(SequenceFileInputFormat.class);

  job2.setOutputKeyClass(Text.class);
  job2.setOutputValueClass(Text.class);

  FileInputFormat.addInputPath(job2, new Path(tempPath));
  FileOutputFormat.setOutputPath(job2, new Path(otherArgs[1]));

System.out.println("CURIO: Validating Potential Outliers(NN search)...");
  // Job2 (CheckNeighboorhood):
  success = (job2.waitForCompletion(false) ? 0 : 1);

System.out.println("job2: " + (System.nanoTime()-start2)/1000000+"ms.");
System.out.println("> All jobs: "
                          + (System.nanoTime()-start)/1000000+"ms.");
System.out.println("Output Directory: " + otherArgs[1]);
```

```
                // return success state:
                return success;

        }



        /**
        * The main entry point.
        */
        public static void main(String[] args) throws Exception {
            // the job:
            int success = ToolRunner.run(new Configuration(), new CurioXXcD(), args);

                System.exit(success);

        }

}
```

D.6 CurioXRRD

The CurioXRRD code:

```
/**
 * CurioXRRD
 *  Use: Driver or Object Oriented approach.
 *   See main() function for examples.
 */

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
//import org.apache.hadoop.mapreduce.lib.chain.ChainReducer; // on 0.21.0 or above!
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 *  <b>CURIO</b> algorithm, by Ceglar, Roddick and Powers,
 *      <br>Flinders UNIVERSITY, Adelaide, 2007.
 *  <p>Improved version for <b>Hadoop</b> MapReducing implementation
 *  <br>to distributed systems, of the original algorithm.
 *
 * @author Rui Pais
 * @version 0.1.2
 */
public class CurioXRRD extends Configured implements Tool {
 /* NOTE: Hadoop require special initialization of variables,
  *  by conf.set() and conf.get(), on setup() implementations
  *  of MapReduces, in order to work on distributed mode.  */
        //** CURIO parameters:  **//
        // partition precision/division (P on article)
        private static int precision ;  // 3 it's the minimum reasonable
        // tolerance (T on article):    // the number of instances (counts)
        private static int tolerance;   // # on a cell to be a potential Outlier


        /** private members: **/
        private static String[] inFiles;

        /**
         *  The CurioXRRD Mapper (2) for Validate All Nearest Neighbors
         *      of each potencial Outlier Cell.
         */
        public static class ValidateNeighborsMap
                                        extends Mapper<Text, Text, Text, Text> {

            private Text index = new Text();
            private Text cell = new Text();

            @Override
            protected void setup(Context context) {
                try {
                        Configuration conf = context.getConfiguration();
                        precision = Integer.valueOf(conf.get("precision"));
                } catch (NumberFormatException e) {
                      //outputs when parameters are not defined by user (use defaults)
                        e.printStackTrace();
                }
            }

            private int dims = 0;
            private String stFormat;

            @Override
            public void map(Text key, Text line, Context context)
                                               throws IOException, InterruptedException {
                // get the values on input line:
                if ( dims == 0 ) {
                        dims = key.getLength()/precision;
                        stFormat = "%0" + dims + "d";
                }
                String keyIndex = key.toString();
                boolean set = false;

                //Set Only Once, will be written many times
                if ( line.find(";", 1) > 0 ) {   // It's PO <=> contains a list:
                      cell.set(keyIndex + " " + Indexing.occurrencesCount(';', line));
                      final String[][] nns = Indexing.neighborsIndexs(keyIndex,
                                                              precision, dims);
                      // OPTIMIZED (CurioXRRD):  o(c3^D + N + n) -> o(n3^D + N + n)
                      // Only saves NearestNeighbors info of Potential Outliers Cells:
```

```
                // get a list of Neighbors for each key/index:
                for (int i = 0; i < Math.pow(3, dims); i++) {
                    // Next NearestNeighbours Index:
                    String indNN = "";
                    String nni = String.format(stFormat,
                                         Long.parseLong(Integer.toString(i,3)));
                    for ( int d = 0; d < dims; d++ )
                        indNN += nns[Integer.valueOf(nni.substring(d, d+1))][d];
                        // Frontiers Neighbors have bad index but dont exist
                        if ( indNN.length() == key.getLength() ) { //don't write
                            if ( !set && indNN.equals(keyIndex) ) {
                                // the cell itself, write all lineNumbers list:
                                context.write(key, line);
                                set = true; // ...and don't need to check again.
                            }
                            else {
                                index.set(indNN);
                                context.write(index, cell);
                            }
                        }
                 }
        }
        else { // Not a Potential Outlier <=> contains a count:
                cell.set(keyIndex + " " + line);
                context.write(key, cell);       // Not a Potential Outlier
        }
}


/**
* The CurioXRRD reducer 1  for Validate All Nearest Neighbors
*       of each potential Outlier Cell.
*/
public static class ValidateNeighborsReduce1
                                    extends Reducer<Text, Text, Text, Text> {

    private Text line = new Text();
    private Text countCell = new Text();
    List<String> inds = new ArrayList<String>();

    @Override
    public void reduce(Text key, Iterable<Text> values, Context context)
                                    throws IOException, InterruptedException {

            inds.clear();  // Clear previous indexes lists
            countCell.clear();
            for (Text entry : values) {
                if ( entry.find(";", 1) > 0 ) {  // a list (potential outlier)

                        context.write(key, entry);
                        countCell.set(Indexing.occurrencesCount(';', entry));
                }
                else {
                        String[] val = entry.toString().split(" ");
                        if ( val[0].equals(key.toString()) )
                                countCell.set(val[1]);
                        else
                                inds.add(val[0]);
                }
            }
            if ( countCell.getLength() > 0 )
```

```
                for (String ind : inds)  {
                        line.set(ind);
                        context.write(line, countCell);
                }
        }

    }


 /**
  * The CurioXRRD reducer 2  for Validate All Nearest Neighbors
  *       of each potential Outlier Cell.
  */
  public static class ValidateNeighborsCombine2
                                     extends Reducer<Text, Text, Text, Text> {

      private Text linesOutliers = new Text();

      @Override
      public void reduce(Text key, Iterable<Text> counts, Context context)
                                throws IOException, InterruptedException {
              int sumNN = 0;
              String line = "";
              do {
                  String value = counts.iterator().next().toString();
                  if ( line.length() == 0 && value.endsWith(";") ) {
                        if ( value.contains(":") )    // or is not a PO cell
                              line = value;  // ... and can be filtered here
                        }
                        else
                              sumNN += Integer.valueOf(value);
              } while ( counts.iterator().hasNext() );

              // outputs <indexKeys sumNN>:
              linesOutliers.set(String.valueOf(sumNN));
              context.write(key, linesOutliers);
              // and <indexKeys {list of offset of entrys on cell}> of PO cells:
              if ( line.length() > 0 ) {
                          linesOutliers.set(line);
                          context.write(key, linesOutliers);
              }
      }

  }


 /**
  * The CurioXRRD reducer 2  for Validate All Nearest Neighbors
  *       of each potential Outlier Cell.
  */
  public static class ValidateNeighborsReduce2
                                     extends Reducer<Text, Text, Text, Text> {
      private String inputPath;

      @Override
      protected void setup(Context context) {
          try {
                  Configuration conf = context.getConfiguration();
                  tolerance = Integer.valueOf(conf.get("tolerance"));
                  inFiles = conf.getStrings("inputFiles");
                  inputPath = conf.get("inputPath");
          } catch (NumberFormatException e) {
                //outputs when parameters are not defined by user (use defaults)

                 e.printStackTrace();
             }
         }

         private Text lineOutlier = new Text();

         @Override
         public void reduce(Text key, Iterable<Text> counts, Context context)
                                         throws IOException, InterruptedException {
                 int sum = 0;
                 String line = "";
                 do {
                       String value = counts.iterator().next().toString();
                       if ( line.length() == 0 && value.endsWith(";") )
                              line = value;
                       else
                              sum += Integer.valueOf(value);
                 } while ( sum <= tolerance && counts.iterator().hasNext() );

                 // output the list of Outliers instances:
                 if ( line.length() > 0 && sum <= tolerance ) {
                       String[] lines = line.split(";");
                       for (int i = 0; i < lines.length; i++) {
                             if ( !lines[i].isEmpty() ) {
                                   lineOutlier.set(CurioXD.readLineHDFS(inputPath,
                                                               inFiles, lines[i]));
                                   context.write(lineOutlier, null);
                             }
                       }
                 }
         }

     }


    /**
     * The driver for CurioXRRD (Distributed version).
     */
     @Override
     public int run (String[] args) throws Exception {
             if (args.length < 2) {
                 System.err.printf(
                    "Basic Usage: %s <inputDir> <outputDir> precision tolerance\n",
                                                     getClass().getSimpleName() );
                 return -1;
             }

             Configuration conf = new Configuration();
             String[] otherArgs = new
                                   GenericOptionsParser(conf, args).getRemainingArgs();
             String tempPath = otherArgs[1] + "_tmp_CellsCounts";
             String tempPath2 = otherArgs[1] + "_tmp_ValidateCells";

             // clean old outputs:
             FileSystem fs =  FileSystem.getLocal(conf);
             fs.delete(new Path(otherArgs[1]), true);
             fs.delete(new Path(tempPath), true);
             fs.delete(new Path(tempPath2), true);

             // set parameters passed from command line to conf:
                    CurioXD.confSetParameters(conf, otherArgs);
                    //NOTE: before passing conf to new jobs ALL parameter must been set!

                    /** JOB 1: Partitioning and instances listing by cells **/

                    long start = System.nanoTime();
                    Job job1 = new Job(conf, "Curio Algorithm Hadoop version");
                    job1.setJarByClass(CurioXRRD.class);

                    job1.setMapperClass(PartitionerXD.CellsCountMap.class);
                    job1.setCombinerClass(PartitionerXD.CellsCountCombine.class);
                    job1.setReducerClass(PartitionerXD.CellsCountReduce.class);

                    job1.setOutputKeyClass(Text.class);
                    job1.setOutputValueClass(Text.class);

                    job1.setOutputFormatClass(SequenceFileOutputFormat.class);

                    FileInputFormat.addInputPath(job1, new Path(otherArgs[0]));
                    FileOutputFormat.setOutputPath(job1, new Path(tempPath));

                    System.out.println(" CURIO: Partitioning/Cells count ...");
                    // the job1 (CellsCount):
                    int success = (job1.waitForCompletion(true) ? 0 : 1);

                  System.out.println("job1: "+ (System.nanoTime()-start)/1000000 + "ms.");
                    long start2 = System.nanoTime();

//          // version 0.21 will implement map/reduce chains:
//             Configuration reduceConf = new Configuration(false);
//             ChainReducer.setReducer(job1, CellsCountReduce.class, Text.class,
//                              Text.class, Text.class, Text.class, true, reduceConf);
//              ChainReducer.addMapper(job, CMap.class, Text.class, Text.class,
//               IntWritable.class, Text.class, false, null);

                    /**  Job2 Check Neighborhood:   **/

                    Job job2 = new Job(conf, "Do a Nearest Neighbor Search.");
                    job2.setJarByClass(CurioXRRD.class);

                    job2.setMapperClass(ValidateNeighborsMap.class);
                    job2.setReducerClass(ValidateNeighborsReduce1.class);

                    job2.setInputFormatClass(SequenceFileInputFormat.class);

                    job2.setOutputKeyClass(Text.class);
                    job2.setOutputValueClass(Text.class);

                    job2.setOutputFormatClass(SequenceFileOutputFormat.class);

                    FileInputFormat.addInputPath(job2, new Path(tempPath));
                    FileOutputFormat.setOutputPath(job2, new Path(tempPath2));

                  System.out.println(" CURIO: Mapping Potential Outliers (NN search)...");
                    // Job2 (CheckNeighboorhood):
                    success = (job2.waitForCompletion(false) ? 0 : 1);

                    /**  Job3 Check Neighborhood:   **/

                    Job job3 = new Job(conf, "Do a Nearest Neighbor Search Reducing.");
                    job3.setJarByClass(CurioXRRD.class);
```

```
                job3.setInputFormatClass(SequenceFileInputFormat.class);

                // NO Map just reduce the output of the last job:
                job3.setCombinerClass(ValidateNeighborsCombine2.class);
                job3.setReducerClass(ValidateNeighborsReduce2.class);

                job3.setOutputKeyClass(Text.class);
                job3.setOutputValueClass(Text.class);

                FileInputFormat.addInputPath(job3, new Path(tempPath2));
                FileOutputFormat.setOutputPath(job3, new Path(otherArgs[1]));

                System.out.println(" CURIO: Validating Potential Outliers ...");
                // Job2 (CheckNeighboorhood):
                success = (job3.waitForCompletion(false) ? 0 : 1);

              System.out.println("job2+3: "+(System.nanoTime()-start2)/1000000+"ms.");
              System.out.println("> All jobs: "
                                             + (System.nanoTime()-start)/1000000+"ms.");
              System.out.println("Output Directory: " + otherArgs[1]);

                // return success state:
                return success;
        }



        /**
        * The main entry point.
        */
        public static void main(String[] args) throws Exception {
            // the job:
            int success = ToolRunner.run(new Configuration(), new CurioXRRD(), args);

                System.exit(success);
        }

}
```

D.7 Curio3XD

The Curio3XD code:

```
/**
 * Curio3XD
 *  Use: Driver or Object Oriented approach.
 *   See main() function for examples.
 */

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 *  <b>CURIO</b> algorithm, by Ceglar, Roddick and Powers,
 *      <br>Flinders UNIVERSITY, Adelaide, 2007.
 *  <p>Improved version for <b>Hadoop</b> MapReducing implementation
 *  <br>to distributed systems, of the original algorithm.
 *
 * @author Rui Pais
 * @version 0.1.2
 */
public class Curio3XD extends Configured implements Tool {
 /* NOTE: Hadoop require special initialization of variables,
   *  by conf.set() and conf.get(), on setup() implementations
   *  of MapReduces, in order to work on distributed mode.  */
        //** CURIO parameters:  **//
        // tolerance (T on article):    // the number of instances (counts)
        private static int tolerance;   // # on a cell to be a potential Outlier

        /** private members: **/
        private static String[] inFiles;




        /**
         *  The Curio3XD Mapper (2) for Validate All Nearest Neighbors
         *      of each potencial Outlier Cell.
         */
     public static class ValidateNeighborsMap
                                   extends Mapper<Text, Text, Text, Text> {

          private int precision;

          @Override
          protected void setup(Context context) {
          try {
                  Configuration conf = context.getConfiguration();
                  precision = Integer.valueOf(conf.get("precision"));
                  String pathFiles = ((FileSplit)context.getInputSplit())
                                             .getPath().getParent().getParent() + "/";
                  pathFiles += FileOutputFormat.getOutputPath(context)
                                                                   + "_tmp_CellsCounts";
                ReadIndexesFiles(pathFiles);
             } catch (NumberFormatException e) {
```

```
            //outputs when parameters are not defined by user (use defaults)
                e.printStackTrace();
        }
    }

    private SequenceFile.Reader reader;
    private Text key = new Text();
    private Text value = new Text();
    private List<String> listOfCells = new ArrayList<String>();
    private void ReadIndexesFiles(String neighborsFiles) {
      // get a list of files on input directory:
      try {
           FileSystem fs = FileSystem.get(new Configuration());
           FileStatus[] status = fs.listStatus(new Path(neighborsFiles));
           for (int i = status.length-1; i >= 0; i--) {
                 if (  status[i].getLen() > 0 &&   /* to avoid the logs */
                       status[i].getPath().getName().startsWith("part-r-") ) {
                       reader = new SequenceFile.Reader(fs, status[i].getPath(),
                                                          new Configuration());
                       while ( reader.next(key) )
                            listOfCells.add(key.toString());

                       reader.close();
                 }
           }
     } catch (IOException e) {
                  System.err.println("Error while trying to read " +
                                                "the list of near neighbors");
                  e.printStackTrace();
     }
    }

    @Override
    public void map(Text index, Text instsRef, Context context)
                                    throws IOException, InterruptedException {

            // Set Only Once, will be written many times:
            if ( instsRef.toString().endsWith(";") )
                // It's a PO <=> contains a list:
                    value.set(Indexing.occurrencesCount(';',instsRef));
            else           // Not a PO <=> contains a count:
                value.set(instsRef);

             // write the list or count of the cell:
            context.write(index, instsRef);

             // write the count on NearNeighbors indexes:
            for ( String ind : listOfCells ) {
              if ( !ind.equals(index.toString()) &&
                  Indexing.isNeighbor(ind, index, precision) ) {
                                key.set(ind);
                                 context.write(key, value);
                      }
            }
     }

}


   /**
    * The combiner 2 for Validate All Nearest Neighbors
    *       of each potential Outlier Cell.
```

```
     */
public static class ValidateNeighborsCombine
                                          extends Reducer<Text, Text, Text, Text> {

       private Text cellData = new Text();

       @Override
       public void reduce(Text key, Iterable<Text> counts, Context context)
                                         throws IOException, InterruptedException {

               long sumNN = 0;
               String line = "";
               do {
                       String value = counts.iterator().next().toString();
                       if ( line.isEmpty() && value.endsWith(";") )
                             line = value; // unique line, samples list in a PO cell
                       else
                             sumNN += Integer.valueOf(value);  // sum of NN counts
               } while ( counts.iterator().hasNext() );

               if ( sumNN > 0 ) { // for isolated cells, don't need to write 0 ...
                     // output the sum of neighbors counts:
                     cellData.set(String.valueOf(sumNN)); // 1st sumNN, to speed up
                     context.write(key, cellData);        // reducer by exit if > T
               }
               // and the list of offset lines:
               if ( !line.isEmpty() ) { // Potential Outlier
                     cellData.set(line);
                     context.write(key, cellData);
               }
         }

     }


    /**
     * The Curio3XD reducer 2  for Validate All Nearest Neighbors
     *       of each potencial Outlier Cell.
     */
     public static class ValidateNeighborsReduce
                                          extends Reducer<Text, Text, Text, Text> {

         private String inputPath;

         @Override
         protected void setup(Context context) {
             try {
                     Configuration conf = context.getConfiguration();
                     tolerance = Integer.valueOf(conf.get("tolerance"));
                     inFiles = conf.getStrings("inputFiles");
                     inputPath = conf.get("inputPath");
             } catch (NumberFormatException e) {
                    //outputs when parameters are not defined by user (use defaults)

                     e.printStackTrace();
             }
         }

         private Text lineOutlier = new Text();

         @Override
         public void reduce(Text key, Iterable<Text> counts, Context context)
```

```
                                throws IOException, InterruptedException {

            long sumNN = 0;
            String line = "";
            do {
                    String value = counts.iterator().next().toString();
                    if ( line.isEmpty() && value.endsWith(";") )
                            line = value;
                    else
                            sumNN += Integer.valueOf(value);
            } while ( sumNN <= tolerance && counts.iterator().hasNext() );

            // output the list of Outliers instances:
            if ( line.length() > 0 && sumNN <= tolerance ) {
                  for ( String ln : line.split(";") ) {
                       if ( !ln.isEmpty() ) {  // It CAN HAVE empty lines!
                            lineOutlier.set(
                                   CurioXD.readLineHDFS(inputPath, inFiles, ln));
                            context.write(lineOutlier, null);
                        }
                  }
             }
      }

 }



/**
 * The driver for Curio3XD (Distributed version).
 */
 @Override
 public int run (String[] args) throws Exception {
         if (args.length < 2) {
             System.err.printf(
                "Basic Usage: %s <inputDir> <outputDir> precision tolerance\n",
                                                 getClass().getSimpleName() );
                return -1;
         }

         Configuration conf = new Configuration();
         String[] otherArgs = new
                           GenericOptionsParser(conf, args).getRemainingArgs();
         String tempPath = otherArgs[1] + "_tmp_CellsCounts";
         String tempPath2 = otherArgs[1] + "_CellsListNeighbors";

         // clean old outputs:
         FileSystem fs =  FileSystem.getLocal(conf);
         fs.delete(new Path(otherArgs[1]), true);
         fs.delete(new Path(tempPath), true);
         fs.delete(new Path(tempPath2), true);

         // set parameters passed from command line to conf:
         CurioXD.confSetParameters(conf, otherArgs);
         //NOTE: before passing conf to new jobs ALL parameter must been set!

         /** JOB 1: Partitioning and instances listing by cells **/

         long start = System.nanoTime();
         Job job1 = new Job(conf, "Curio Algorithm Hadoop version");
         job1.setJarByClass(Curio3XD.class);
```

```
        job1.setMapperClass(PartitionerXD.CellsCountMap.class);
        job1.setCombinerClass(PartitionerXD.CellsCountCombine.class);
        job1.setReducerClass(PartitionerXD.CellsCountReduce.class);

        job1.setOutputKeyClass(Text.class);
        job1.setOutputValueClass(Text.class);

        job1.setOutputFormatClass(SequenceFileOutputFormat.class);

        FileInputFormat.addInputPath(job1, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job1, new Path(tempPath));

        System.out.println(" CURIO: Partitioning/Cells count ...");
        // the job1 (CellsCount):
        int success = (job1.waitForCompletion(true) ? 0 : 1);

      System.out.println("job1: " + (System.nanoTime()-start)/1000000 + "ms.");

        long start2 = System.nanoTime();

        /**  Job2 Check Neighborhood:   **/

        start2 = System.nanoTime();
        Job job2 = new Job(conf, "Do a Nearest Neighbor Search.");
        job2.setJarByClass(Curio3XD.class);

        job2.setMapperClass(ValidateNeighborsMap.class);
        job2.setCombinerClass(ValidateNeighborsCombine.class);
        job2.setReducerClass(ValidateNeighborsReduce.class);

        job2.setInputFormatClass(SequenceFileInputFormat.class);

        job2.setOutputKeyClass(Text.class);
        job2.setOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job2, new Path(tempPath));
        FileOutputFormat.setOutputPath(job2, new Path(otherArgs[1]));

      System.out.println(" CURIO: Validating Potential Outliers (NN search)");
        // Job2 (CheckNeighboorhood):
        success = (job2.waitForCompletion(false) ? 0 : 1);

      System.out.println("job2: "+(System.nanoTime()-start2)/1000000 + "ms.");
      System.out.println("> All jobs: "
                              + (System.nanoTime()-start)/1000000+"ms.");
      System.out.println("Output Directory: " + otherArgs[1]);

        // return success state:
        return success;
}



/**
* The main entry point.
*/
public static void main(String[] args) throws Exception {
    // the job:
    int success = ToolRunner.run(new Configuration(), new Curio3XD(), args);

        System.exit(success);
}
```

```
}
```

D.8 Curio32XD

The code for Curio32XD algorithm:

```
/**
 * Curio32XD
 *  Use: Driver or Object Oriented approach.
 *   See main() function for examples.
 */

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 *  <b>CURIO</b> algorithm, by Ceglar, Roddick and Powers,
 *      <br>Flinders UNIVERSITY, Adelaide, 2007.
 *  <p>Improved version for <b>Hadoop</b> MapReducing implementation
 *  <br>to distributed systems, of the original algorithm.
 *
 * @author Rui Pais
 * @version 0.1.2
 */
public class Curio32XD extends Configured implements Tool {
 /* NOTE: Hadoop require special initialization of variables,
  *  by conf.set() and conf.get(), on setup() implementations
  *  of MapReduces, in order to work on distributed mode.  */
        //** CURIO parameters:  **//
        // partition precision/division (P on article)
        private static int precision;   // 3 it's the minimum reasonable
        // tolerance (T on article):    // the number of instances (counts)
        private static int tolerance;   // # on a cell to be a potential Outlier
```

```
/** private members: **/
private static String[] inFiles;


/**
 *  The Curio32XD Mapper (1) for NearNeighbors Listing.
 */
public static class CellsListNeighborsMap
                                     extends Mapper<Text, Text, Text, Text> {

    private final Text k = new Text("k");

    @Override
    public void map(Text key, Text entry, Context context)
                                    throws IOException, InterruptedException {
            context.write(k, key);
    }
}


/**
* The Curio32XD reducer (1) for List NearNeighbors (Distributed version).
*/
public static class CellsListNeighborsReduce
                                     extends Reducer<Text, Text, Text, Text> {

    @Override
    protected void setup(Context context) {
        try {
                Configuration conf = context.getConfiguration();
                precision = conf.getInt("precision", 3);
        } catch (NumberFormatException e) {
               //outputs when parameters are not defined by user (use defaults)

                e.printStackTrace();
        }
    }

    private Text index = new Text();
    private Text indNNList = new Text();
    private ArrayList<String> inds = new ArrayList<String>();
    private ArrayList<StringBuilder>indsNN = new ArrayList<StringBuilder>();

    @Override
    public void reduce(Text key, Iterable<Text> entries, Context context)
                                    throws IOException, InterruptedException {

            // a unique key with all indexes of dataset:
            for ( Text entry : entries ) { // entries will have only one entry
                // write on arrayList:
                 inds.add(entry.toString());
                 indsNN.add(new StringBuilder());
            }

            // Assign the Near Neighbors to each cell:
            for (int i = 0; i < inds.size(); i++) {
                for (int j = i + 1; j < inds.size(); j++) {
                    // Due to symmetric neighborhood relationship, and i != j,
                    // ONLY needs to check (C^2 - C)/2 cells, not C^2
                    if ( Indexing.isNeighbor(inds.get(i), inds.get(j),
                                                             precision) ) {
```

```
                        indsNN.get(i).append(inds.get(j) + " ");
                        indsNN.get(j).append(inds.get(i) + " ");
                    }
                }
                index.set(inds.get(i));
                indNNList.set(indsNN.get(i).toString());
                context.write(index, indNNList);
            }
    }

}


/**
 *  The Curio32XD Mapper (2) for Validate All Nearest Neighbors
 *      of each potencial Outlier Cell.
 */
public static class ValidateNeighborsMap
                                     extends Mapper<Text, Text, Text, Text> {

    private SequenceFile.Reader reader;
    private Text key = new Text();
    private Text value = new Text();

    private Map<String, String> listOfNN = new HashMap<String, String>();

    private void ReadNeighborsFiles(String neighborsFiles) {
        // get a list of files on input directory:
        try {
                FileSystem fs = FileSystem.get(new Configuration());
                FileStatus[] status = fs.listStatus(new Path(neighborsFiles));
                for (int i = status.length-1; i >= 0; i--) {
                    if ( status[i].getLen() > 0 &&  /* to avoid the logs */
                         status[i].getPath().getName().startsWith("part-r-") ){
                        reader = new SequenceFile.Reader(fs,
                                     status[i].getPath(), new Configuration());
                        while ( reader.next(key, value) )
                            listOfNN.put(key.toString(), value.toString());
                        reader.close();
                    }
                }
          } catch (IOException e) {
               System.err.println("Error while trying to read " +
                                               "the list of near neighbors");
                e.printStackTrace();
          }
    }

    @Override
    protected void setup(Context context) {
        try {
                String neighborsFiles = ((FileSplit)context.getInputSplit())
                                      .getPath().getParent().getParent() + "/";
                neighborsFiles += FileOutputFormat.getOutputPath(context)
                                                 + "_CellsListNeighbors";
                ReadNeighborsFiles(neighborsFiles);
        } catch (NumberFormatException e) {
               //outputs when parameters are not defined by user (use defaults)

                e.printStackTrace();
        }
    }
    @Override
    public void map(Text index, Text instsRef, Context context)
                                    throws IOException, InterruptedException {

          // Set Only Once, will be written many times:
          if ( instsRef.toString().endsWith(";") )
                  // It's a PO <=> contains a list:
                  value.set(Indexing.occurrencesCount(';', instsRef));
          else    // Not a PO <=> contains a count:
                  value.set(instsRef);

          // write the list or count of the cell:
          context.write(index, instsRef);
          // write the count on NearNeighbors indexes:
          String indsNN = listOfNN.get(index.toString());
          if ( !indsNN.isEmpty() ) {  // Can happens if cell has no Neighbors!
                for ( String ind : indsNN.split(" ")) {
                        key.set(ind);
                        context.write(key, value);
                }
          }
    }

}


/**
 * The Curio32XD combiner 2 for Validate All Nearest Neighbors
 *       of each potential Outlier Cell.
 */
public static class ValidateNeighborsCombine
                                     extends Reducer<Text, Text, Text, Text> {

    private Text cellData = new Text();

    @Override
    public void reduce(Text key, Iterable<Text> counts, Context context)
                                    throws IOException, InterruptedException {
          long sumNN = 0;
          String line = "";
          do {
                  String value = counts.iterator().next().toString();
                  if ( line.isEmpty() && value.endsWith(";") )
                        line = value; // unique line, samples list in a PO cell
                  else
                        sumNN += Integer.valueOf(value);  // sum of NN counts
          } while ( counts.iterator().hasNext() );

          if ( sumNN > 0 ) {  // isolated cell, don't need to write 0 ...
                // output the sum of neighbors counts:
                cellData.set(String.valueOf(sumNN)); // 1st sumNN, to speed up
                context.write(key, cellData);        // reducer by exit if > T
          }
          // and the list of offset lines:
          if ( !line.isEmpty() ) { // Potential Outlier
                cellData.set(line);
                context.write(key, cellData);
          }
    }
 }


/**
 * The Curio32XD reducer 2  for Validate All Nearest Neighbors
 *       of each potencial Outlier Cell.
 */
 public static class ValidateNeighborsReduce
                                     extends Reducer<Text, Text, Text, Text> {

     private String inputPath;

     @Override
     protected void setup(Context context) {
         try {
                 Configuration conf = context.getConfiguration();
                 tolerance = Integer.valueOf(conf.get("tolerance"));
                 inFiles = conf.getStrings("inputFiles");
                 inputPath = conf.get("inputPath");
         } catch (NumberFormatException e) {
               //outputs when parameters are not defined by user (use defaults)
                  e.printStackTrace();
         }
     }

     private Text lineOutlier = new Text();

     @Override
     public void reduce(Text key, Iterable<Text> counts, Context context)
                                     throws IOException, InterruptedException {

           long sumNN = 0;
           String line = "";
           do {
                   String value = counts.iterator().next().toString();
                   if ( line.isEmpty() && value.endsWith(";") )
                         line = value;
                   else
                         sumNN += Integer.valueOf(value);
           } while ( sumNN <= tolerance && counts.iterator().hasNext() );

           // output the list of Outliers instances:
           if ( line.length() > 0 && sumNN <= tolerance ) {
                 for ( String ln : line.split(";") )
                         if ( !ln.isEmpty() ) {  // It CAN HAVE empty lines!
                             lineOutlier.set(
                                  CurioXD.readLineHDFS(inputPath, inFiles, ln));
                             context.write(lineOutlier, null);
                         }
           }
     }

 }



/**
 * The driver for Curio32XD (Distributed version).
 */
 @Override
 public int run (String[] args) throws Exception {
         if (args.length < 2) {
```

```
        System.err.printf(
           "Basic Usage: %s <inputDir> <outputDir> precision tolerance\n",
                                          getClass().getSimpleName() );
            return -1;
    }

    Configuration conf = new Configuration();
    String[] otherArgs = new
                    GenericOptionsParser(conf, args).getRemainingArgs();
    String tempPath = otherArgs[1] + "_tmp_CellsCounts";
    String tempPath2 = otherArgs[1] + "_CellsListNeighbors";

    // clean old outputs:
    FileSystem fs =  FileSystem.getLocal(conf);
    fs.delete(new Path(otherArgs[1]), true);
    fs.delete(new Path(tempPath), true);
    fs.delete(new Path(tempPath2), true);

    // set parameters passed from command line to conf:
    CurioXD.confSetParameters(conf, otherArgs);
    //NOTE: before passing conf to new jobs ALL parameter must been set!

    /** JOB 1: Partitioning and instances listing by cells **/

    long start = System.nanoTime();
    Job job1 = new Job(conf, "Curio Algorithm Hadoop version");
    job1.setJarByClass(Curio32XD.class);

    job1.setMapperClass(PartitionerXD.CellsCountMap.class);
    job1.setCombinerClass(PartitionerXD.CellsCountCombine.class);
    job1.setReducerClass(PartitionerXD.CellsCountReduce.class);

    job1.setOutputKeyClass(Text.class);
    job1.setOutputValueClass(Text.class);

    job1.setOutputFormatClass(SequenceFileOutputFormat.class);

    FileInputFormat.addInputPath(job1, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job1, new Path(tempPath));

    System.out.println(" CURIO: Partitioning/Cells count ...");
    // the job1 (CellsCount):
    int success = (job1.waitForCompletion(true) ? 0 : 1);

  System.out.println("job1: " + (System.nanoTime()-start)/1000000 + "ms.");

    /**  Job2 List Neighborhood:   **/

    long start2 = System.nanoTime();
    fs.delete(new Path(tempPath2), true);

    Job job2 = new Job(conf, "Curio Partitioner: List Cells Neighbors .");
    job2.setJarByClass(Curio32XD.class);

    job2.setMapperClass(CellsListNeighborsMap.class);
    job2.setReducerClass(CellsListNeighborsReduce.class);

    job2.setInputFormatClass(SequenceFileInputFormat.class);
    job2.setOutputFormatClass(SequenceFileOutputFormat.class);

    job2.setOutputKeyClass(Text.class);
    job2.setOutputValueClass(Text.class);
```

```
            FileInputFormat.addInputPath(job2, new Path(tempPath));
            FileOutputFormat.setOutputPath(job2, new Path(tempPath2));

            System.out.println(" CURIO: NearNeighbors listing ...");
            // the job2 (CellsCounting):
            success = (job2.waitForCompletion(true) ? 0 : 1);

          System.out.println("job2: "+ (System.nanoTime()-start2)/1000000 + "ms.");

            /**  Job3 Check Neighborhood:   **/

            start2 = System.nanoTime();
            Job job3 = new Job(conf, "Do a Nearest Neighbor Search.");
            job3.setJarByClass(Curio32XD.class);

            job3.setMapperClass(ValidateNeighborsMap.class);
            job3.setCombinerClass(ValidateNeighborsCombine.class);
            job3.setReducerClass(ValidateNeighborsReduce.class);

            job3.setInputFormatClass(SequenceFileInputFormat.class);

            job3.setOutputKeyClass(Text.class);
            job3.setOutputValueClass(Text.class);

            FileInputFormat.addInputPath(job3, new Path(tempPath));
            FileOutputFormat.setOutputPath(job3, new Path(otherArgs[1]));

          System.out.println(" CURIO: Validating Potential Outliers (NN search)");
            // Job2 (CheckNeighboorhood):
            success = (job3.waitForCompletion(false) ? 0 : 1);

          System.out.println("job3: " + (System.nanoTime()-start2)/1000000+"ms.");
          System.out.println("> All jobs: "
                                       + (System.nanoTime()-start)/1000000+"ms.");
          System.out.println("Output Directory: " + otherArgs[1]);

            // return success state:
            return success;
    }



    /**
    * The main entry point.
    */
    public static void main(String[] args) throws Exception {
        // the job:
        int success = ToolRunner.run(new Configuration(), new Curio32XD(), args);

            System.exit(success);
    }

}
```

D.9 Indexing

The code for the Indexing functions (Indexing.java file):

```
import org.apache.hadoop.io.Text;

/**
 *
 * @author Rui Pais
 * @version 0.1.2
 *
 */
public class Indexing {

        private static int dims;

        /**
         *
         * @param instance
         * @param precision
         * @param scaleFactor
         * @param shift
         * @param bitFormat
         * @return
         */
        public static String calcIndex(Text instance, int precision, int scaleFactor,
                                                    long shift, String bitFormat) {
            try {
                    StringBuilder index = new StringBuilder();
                    String[] values = instance.toString().split(" ");
                    //TODO: the separator of data in file could/should be tunable...
                    for (int d = 0; d < values.length; d++) {
                          double v = Double.parseDouble(values[d]);
                          if ( shift > 0 ) v += shift;            // shift, if needed
                          if ( scaleFactor > 1 ) v *= scaleFactor;      // Rescale
                        // convert to binary:   //  REQUIRES v > 0 to be Long.toBinary
                          long b = Long.parseLong(Long.toBinaryString(Math.round(v)));
                          index.append(String.format(bitFormat, b).substring(0,
                                                                       precision));
                    }
                    return index.toString();
            } catch (NumberFormatException e) {
                    System.err.print("CURIO: Error processing instance: \n"
                                                           + instance + "\n");
                    System.out.print("Check definied maximum and minimum!");
                    e.printStackTrace();
            }
            return null;
        }


        /**
         *
         * @param cell
         * @param anotherCell
         * @param dims
         * @param precision
         * @return
         */
        public static boolean isNeighbor(String cell, String anotherCell,
                                                               int precision) {
            if ( dims == 0 ) // define only once:
                    dims = cell.length()/precision;
            String dimBinary;
            for (int i = 0; i < dims; i++) {
                    int p = i * precision;
                    dimBinary = cell.substring(p, p + precision);
                    int c = Integer.parseInt(dimBinary, 2);
                    dimBinary = anotherCell.substring(p, p + precision);

                    int n = Integer.parseInt(dimBinary, 2);
                    // will be a neighbor only:
                    //  if each dimension index is the same or +/-1
                    if ( !(n == c || n == c-1 || n == c+1) )
                            return false;
            }
            // this check was done before on Map:
            //if ( anotherCell.equals(cell) ) return false; // same cell indexs...

            return true;
        }


        /**
         *
         * @param cellIndex
         * @param anotherCellIndex
         * @param precision
         * @param dims
         * @return
         */
        public static boolean isNeighbor(String cellIndex, Text anotherCellIndex,
                                                                    int precision) {
                if ( dims == 0 ) // define only once when mapping:
                        dims = cellIndex.length()/precision;
                String dimBinary;
                String anotherCell = anotherCellIndex.toString();
                for (int i = 0; i < dims; i++) {
                        int p = i * precision;
                        dimBinary = cellIndex.substring(p, p + precision);
                        int c = Integer.parseInt(dimBinary, 2);
                        dimBinary = anotherCell.substring(p, p + precision);

                        int n = Integer.parseInt(dimBinary, 2);
                        // will be a neighbor only:
                        //  if each dimension index is the same or +/-1
                        if ( !(n == c || n == c-1 || n == c+1) )
                                return false;
                }
                // this check was done before on Map:
                //if ( cellIndex.equals(anotherCell) ) return false; // same cell
    indexs...
                return true;
        }


        /**
         * Helper function to make a list of neighbor indexes,
         *  will be called by a NN search.
         *
         * @param cellIndex
         * @param dims
         * @param precision
         * @return matrix with i, i-1 and i+1 neighbor indexes, by dimension.
         */
        public static String[][] neighborsIndexs(String cellIndex, int precision,
                                                                  int dims) {

              String[][] ks = new String[3][dims]; // 3: prevInd, coordInd , nextInd
              // line 1 have the index of the cell:
              for (int i = 0; i < dims; i++) {
                    int p = i*precision;
                    ks[1][i] = cellIndex.substring(p, p+precision);
              }
              // and line 0 and 2 have the previous and next cells indexes (by dim):
              ks[0] = new String[dims];
              ks[2] = new String[dims];
              // calculate the previous and next cell's index for each dimension:

              for (int k = 0; k < dims; k++) {
                    int val = Integer.parseInt(ks[1][k], 2);
                    ks[0][k]= String.format("%" + precision + "s",
                                  Integer.toBinaryString(val-1)).replaceAll(" ", "0");
                    ks[2][k] = String.format("%" + precision + "s",
                                  Integer.toBinaryString(val+1)).replaceAll(" ", "0");
              }
              return ks;
        }

        /**
         * Counts the number of occurrences of a character in a string
         *
         * @param aChar
         * @param instance
         * @return
         */
        public static int countOccurrences(char aChar, Text instance) {
                int count = 0;
                for (int i = 0; i < instance.getLength(); i++) {
                        if ( instance.charAt(i) == aChar )
                                count++;
                }
                return count;
        }

        /**
         * Counts the number of occurrences of a character in a string
         *
         * @param aChar
         * @param instance
         * @return
         */
        public static String occurrencesCount(char aChar, Text instance) {
            int count = 0;
            byte[] instances = instance.getBytes();         // not a copy, just a ref.
            for (int i = 0; i < instance.getLength(); i++) { // DON'T USE:
                    if ( instances[i] == aChar )              // instances.length!!
                            count++;
            }
            return String.valueOf(count);
        }

}
```

D.10 Code of Auxiliary Applications and Scripts

D.10.1 ExtremaD

```java
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


/**
 * @author Rui Pais
 * @version 0.0.4
 *
 */
public class ExtremaD extends Configured implements Tool {
        // private members:
        private static Path outputPath = null;
        private static double minValue = Double.MAX_VALUE;
        private static double maxValue = Double.MIN_VALUE;


        /**
         * Constructor  for ExtremaD Class
         * receiving the args[] as argument
         *
         * @param args
         * @throws Exception
         */
        public ExtremaD(String[] args) throws Exception {
                System.out.println("[ExtremaD] Find minimum and maximum values...");
                // run driver to get the extreme values of a Dataset:
                ToolRunner.run(this, args);
        }
```

```
/**
 * Default Constructor
 * @throws Exception
 */
public ExtremaD() throws Exception {
        // default don't do nothing,
        // being called as a driver.
}


public final double getMinValue() {
        return minValue;
}

public final double getMaxValue() {
        return maxValue;
}

/**
 *
 * @return
 */
public String getOutputRawPath() {
        return outputPath + Path.SEPARATOR + getOutputPath().getName();
}

/**
 *
 * @return
 */
public Path getOutputPath() {
        try {
            FileSystem fs = FileSystem.get(new Configuration());
                FileStatus[] status = fs.listStatus(outputPath);
              System.out.println("status.length = " + status.length);
                for (int i = 0; i < status.length; i++)
                        System.out.println("status = " + status[i].getPath());
                // Result on last line of last file:
                return status[status.length-1].getPath();
        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
        return null;
}


private void readResult() throws IOException {
        FileSystem fs = FileSystem.get(new Configuration());
        FileStatus[] status = fs.listStatus(outputPath);
        // Extrema result would be on last file

DoubleWritable min = new DoubleWritable();
DoubleWritable max = new DoubleWritable();

SequenceFile.Reader reader;
boolean readed = false;
for (int i = status.length-1; i >= 0 && !readed; i--) {
        if (  status[i].getLen() > 0 &&
                        status[i].getPath().getName().contains("part-r-") ) {
                reader = new SequenceFile.Reader(fs, status[i].getPath(),
                                                   new Configuration());
                while (reader.next(min, max) );
                reader.close();
                minValue = Math.min(minValue, min.get());
                maxValue = Math.max(maxValue, max.get()) ;
                if ( maxValue != Double.MIN_VALUE ) readed = true;
            }
        }
}


    /**
     *  The Mapper for MapReduce.
     */
    public static class ExtremaMap
                                extends Mapper<LongWritable, Text,
                                              DoubleWritable, DoubleWritable> {

        private DoubleWritable value = new DoubleWritable();
        private DoubleWritable counter = new DoubleWritable();

        public void map(LongWritable key, Text line, Context context)
                                throws IOException, InterruptedException {
                long count = 0; // inside the map make it output only once per key
                String[] values = line.toString().split(" ");
                for (int i = 0; i < values.length; i++) {
                        value.set(Double.parseDouble(values[i]));
                        counter.set(++count);
                        context.write(counter, value);
                }
        }
}

    /**
    * The reducer class
    */
    public static class ExtremaReduce
                extends Reducer<DoubleWritable, DoubleWritable,
                                             DoubleWritable, DoubleWritable> {

        DoubleWritable max = new DoubleWritable(Double.MIN_VALUE);
        DoubleWritable min = new DoubleWritable(Double.MAX_VALUE);

        public void reduce(DoubleWritable key, Iterable<DoubleWritable> values,
                                                        Context context)
                                 throws IOException, InterruptedException {

                // extreme values in DataSet:
                for ( DoubleWritable value : values ) {
                        max.set(Math.max(max.get(), value.get()));
                        min.set(Math.min(min.get(), value.get()));
                }
                context.write(min, max);
        }
    }

    /**
    * The driver.
```

```
        */
        public int run (String[] args) throws Exception {
                Configuration conf = new Configuration();
                String[] otherArgs = new
                                  GenericOptionsParser(conf, args).getRemainingArgs();


                FileSystem fs =  FileSystem.getLocal(conf);
                fs.delete(new Path(otherArgs[1]), true);

                long start = System.nanoTime();
                Job job = new Job(conf, "Get Max and Min Value from a DataSet");
                job.setJarByClass(ExtremaD.class);

                job.setMapperClass(ExtremaMap.class);
                job.setReducerClass(ExtremaReduce.class);

                job.setOutputKeyClass(DoubleWritable.class);
                job.setOutputValueClass(DoubleWritable.class);

                job.setOutputFormatClass(SequenceFileOutputFormat.class);

                FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
                if ( otherArgs.length == 1 )
                      FileOutputFormat.setOutputPath(job, new
                                                           Path("output_Extrema_Seq"));
                else
                      FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

                int success = (job.waitForCompletion(false) ? 0 : 1);

             System.out.println("job1: " + (System.nanoTime()-start)/1000000 + "ms.");


                // save the path where it was saved the output
                outputPath = FileOutputFormat.getOutputPath(job);

                // get max and min from last line of output:
                readResult();

                // return success state:
                return success;
        }



        /**
        * The main entry point.
        */
        public static void main(String[] args) throws Exception {
                //// Run the driver:
                //int success = ToolRunner.run(new ExtremaD(), args );

                // Or as an object:
                ExtremaD extrema = new ExtremaD(args);
              System.out.println(extrema.getMinValue() + " " + extrema.getMaxValue());
        }

}
```

D.10.2 CURIO

Java code for the original CURIO algorithm, the non-distributed version:

```
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.HashMap;

/**
 * CURIO -      A fast Outliers Detection Algorithm
 *                      for Large Datasets.
 *
 *              Based on paper/algorithm by:
 *              Ceglar, Roddick and Powers,
 *              Flinders UNIVERSITY, Adelaide, 2007.
 *
 *              <p>Basic Algorithm:
 *              <p> 1. Quantisation of space  (-> potential outliers)
 *              <p> 2. Discover Outliers  (-> validation of potential outliers)
 *
 * @author Rui Pais
 * @version 0.0.3
 */
public class Curio {
        // CURIO parameters:
        private int precision;  // partition precision/division (P on article)
        private int tolerance;
        private int scaleFactor = 10;
        // private members:
        private int bitStringSize;
        private boolean quantisized = false;  // check if quantisation already done
        private String datafile = "";
        // table with populated cells and their counts:
        HashMap<String, Integer> cells = new HashMap<String, Integer>();
        HashMap <Integer, String> outliers = new HashMap<Integer, String>();


        /**
         *
         * @param datafile
         * @param precision
         * @param tolerance
         * @param maxExpValue
         */
        public Curio(String datafile, int precision, int tolerance, int maxExpValue){
                // CURIO parameters:
                this.precision = precision;
                this.tolerance = tolerance;
                this.datafile = datafile;
                if (maxExpValue == 0) maxExpValue = getDataMaxValue(); // just in case
                maxExpValue *= scaleFactor;
                bitStringSize = Integer.toBinaryString(maxExpValue).length();
                if ( bitStringSize < precision )
                        this.precision = bitStringSize;
        }
```

```
/**
 *
 * @param datafile
 * @param precision
 * @param tolerance
 */
public Curio(String datafile, int precision, int tolerance) {
        // CURIO parameters:
        this.precision = precision;
        this.tolerance = tolerance;
        this.datafile = datafile;
        // if maxExpectedValue is not specified, must get it from data
        // ( implying an extra read of dataset and O(2n) -> O(3n) )
        int maxExpValue = getDataMaxValue()*scaleFactor;
        bitStringSize = Integer.toBinaryString(maxExpValue).length();
        if ( bitStringSize < precision )
                this.precision = bitStringSize;
}


private InputStreamReader getStreamReader() {
        InputStreamReader sr = null;
        try{ // Open the file:
            FileInputStream fstream = new FileInputStream(datafile);
            // Get the object of DataInputStream
            DataInputStream in = new DataInputStream(fstream);
            sr = new InputStreamReader(in);
        }catch (Exception e) { //Catch exception if any
      System.err.println("Error 1: " + e.getMessage());
    }
        return sr;
}


/**
 *
 * @return
 */
public int getDataMaxValue() {
        double max = Double.MIN_VALUE;
        try{ // Get the DataInputStream
            InputStreamReader in = getStreamReader();
            BufferedReader br = new BufferedReader(in);
            String strLine;
            // Read File Line By Line
            while ( (strLine = br.readLine()) != null )   {
                        // calculate the index of each data instance:
                String[] dim = strLine.split(" ");
                for (int i = 0; i < dim.length; i++) {
                        double d = Double.parseDouble(dim[i]);
                        if ( d > max ) max = d;
                        }
            }
            // Close the input stream
            in.close();
            // if normalized need to rescale to a bigger factor:
            if ( max <= 1 ) scaleFactor = (int)(100/max); // convert to binary
            if ( max > 1000 ) scaleFactor = 1;
            }catch (Exception e){  // Catch exception, if any
              System.err.println("Error 1.a: " + e.getMessage());
            }
            return (int)Math.ceil(max);
```

```java
}


/**
 *
 * @param instance
 * @return
 */
public String calcIndex(String instance){
        String index = "";
        String bitFormat = "%0" + bitStringSize + "d";
        //TODO: the separator of data in file can/should be tunable...
        String[] dims = instance.split(" ");
        for (int d = 0; d < dims.length; d++) {
            double dd = Double.parseDouble(dims[d]) * scaleFactor;
            // convert to binary:
            long b =
                  Long.parseLong(Integer.toBinaryString((int)Math.round(dd)));
            index += String.format(bitFormat, b).substring(0, precision) + " ";
        }
        return index;
}


private String[][] getNeighborIndexes(String cellIndex) {
        String[][] ks = new String[3][];
        // line 1 have the index of the cell:
        ks[1] = cellIndex.split(" ");
        // and line 0 and 2 have the previous and next cells indexes (by dim):
        int dims = ks[1].length;
        int size = ks[1][0].length();
        ks[0] = new String[dims];
        ks[2] = new String[dims];
        // calculate the previous and next cell's index for each dimension:

        for (int k = 0; k < dims; k++) {
                int i = Integer.parseInt(ks[1][k], 2);
                ks[0][k]= String.format("%" + size + "s",
                            Integer.toBinaryString(i-1)).replaceAll(" ", "0");
                ks[2][k] = String.format("%" + size + "s",
                            Integer.toBinaryString(i+1)).replaceAll(" ", "0");
        }
        return ks;
}


/**
 *
 * @param cellIndex
 * @return
 */
public boolean searchNeighborhood(String cellIndex) {
        String[][] nns = getNeighborIndexes(cellIndex);
        int dims = nns[1].length;
        String stFormat = "%0" + dims + "d";
        int ct = 0;     // the sum of count of elements in neighbor cells
        for (int i = 0; i < Math.pow(3, dims); i++) {
                // Next NearestNeighbours Index:
                String nn = String.format(stFormat,
                                       Long.parseLong(Integer.toString(i,3)));

                String ind = "";
                for ( int d = 0; d < dims; d++ ) {
                    ind += nns[Integer.valueOf(nn.substring(d, d+1))][d] + " ";
                }
                if ( cells.containsKey(ind) && !ind.equals(cellIndex) )
                        ct += cells.get(ind);
                if ( ct > tolerance )
                        return false;
        }
        return true;
}


public void doQuantisation() {
        // 1st Read of dataset:
        try{
            // Get the Input Stream Reader
                InputStreamReader in = getStreamReader();
                BufferedReader br = new BufferedReader(in);
            String strLine;
            cells.clear();              // clear any previous partitions
            // Read File Line By Line
            while ( (strLine = br.readLine()) != null )   {
                // calculate the index of each data instance:
                String ind = calcIndex(strLine); //, false);
                if ( cells.containsKey(ind) ) {
                    int c = cells.get(ind);
                    cells.put(ind, ++c);
                }
                else
                    cells.put(ind, 1);
            }
            // finish correctly:
                quantisized = true;
            // Close the input stream
            in.close();
    }catch (Exception e){ //Catch exception if any
      System.err.println("Error 2: " + e.getMessage());
    }
}


public boolean discoverOutliers() {
        // ALGORITHM 1. Discover Outliers cells

        /** 1. Check quantization       **/
        if (! quantisized ) doQuantisation();

        /** 2. Discover (validation of candidate cells) **/
        try{    // 2nd Read of dataset:
                // Get the input Stream Reader
            InputStreamReader in = getStreamReader();
            BufferedReader br = new BufferedReader(in);
            HashMap <String, Boolean> checked = new HashMap<String, Boolean>();
            String instance;
            int line = 1;   // files starts by line 1, just a key for the Map
            outliers.clear();       // clear any previous list
            while ( (instance = br.readLine()) != null )   {
                // calculate the index of data instance:
                String ind = calcIndex(instance);
                int s = cells.get(ind);
                if ( s <= tolerance ) { // potential outlier
                        if ( !checked.containsKey(ind) )
```

```
                              checked.put(ind, searchNeighborhood(ind));
                        if ( checked.get(ind) )
                              outliers.put(line, instance);
                }
                line++;
            }
            // Close the input stream
            in.close();
    }catch (Exception e){ //Catch exception if any
      System.err.println("Error 3: " + e.getMessage());
    }
    return (!outliers.isEmpty());
}


/**
 *
 * @param tolerance
 * @return
 */
public boolean discoverOutliers(int tolerance) {
        this.tolerance = tolerance;
        return discoverOutliers();
}



/** *************************
 *      Class members access
 ** *************************/

/**
 * @return the precision
 */
public int getPrecision() {
        return precision;
}

/**
 * @param precision the precision to set
 */
public void setPrecision(int precision) {
        this.precision = precision;
        doQuantisation(); // with new precision value
}

/**
 * @return the tolerance
 */
public int getTolerance() {
        return tolerance;
}

/**
 * @param tolerance the tolerance to set
 */
public void setTolerance(int tolerance) {
        this.tolerance = tolerance;
}

/**
 * Default it's 10, or 100 for normalized data.
```

```
         *
         * @return the scaleFactor
         */
        public int getScaleFactor() {
                return scaleFactor;
        }

        /**
         * Default it's 10, or 100 for normalized data.
         *
         * @param scaleFactor the scale factor to apply to all data
         */
        public void setScaleFactor(int scaleFactor) {
                this.scaleFactor = scaleFactor;
                doQuantisation();       // with new scaleFactor;
                if ( !outliers.isEmpty() ) outliers.clear();
        }

        /**
         * @return the outliers HashMap indexed by the line number of data file.
         */
        public HashMap<Integer, String> getOutliers() {
                if ( outliers.isEmpty() /* == null */ )
                        discoverOutliers();
                return outliers;
        }


        /** ************************** **
         **     FOR TEST PURPOSES      **
         ** ************************** **/
        public static void main(String[] args) {

                // CURIO parameters:
                int P = 5;              // partition precision
                int T = 3;              // Tolerance

                String data = "iris3d.data"; int maxVal = 8;

                // creates a new CURIO object with the desired parameters:
                Curio curioTest = new Curio(data, P, T, maxVal);

                System.out.println("\n DATA: = " + data);
                // Discover Outliers with several tolerance values:
                for (int p = 3; p <= 6; p++){
                    System.out.println("\n Precision = " + p);
                    curioTest.setPrecision(p);
                    for (int t = 2; t <= 5; t++) {
                        System.out.println("\n Tolerance = " + t);
                        curioTest.discoverOutliers(t);
                        if ( curioTest.getOutliers().size()<10 )
                            for ( String outlier : curioTest.getOutliers().values( )) {
                                        System.out.println(outlier);
                            }
                    }
            }
    }

}
```

D.10.3 GetColumns

Java code to extract a given number of columns/attributes from a dataset (on a text file with sepaces as separator):

```java
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;


public class GetColumns {

        private static String separator = " ";

        public static String getSeparator() {
                return separator;
        }

        public static void setSeparator(String separator) {
                GetColumns.separator = separator;
        }

        private static BufferedReader getBuffer(String fileName) {
                BufferedReader br;
                try { // read File:
                        br = new BufferedReader(new FileReader(fileName));
                        return br;
                } catch (FileNotFoundException e) {
                        e.printStackTrace();
                }
                return null;
        }

        private static void outputLines(String fileName, int numOfColumns) {
                try {
                        BufferedReader br = getBuffer(fileName);
                        String line = "";
                        // line by line and split the lines:
                        while ( (line = br.readLine()) != null ) {
                                String[] cols = line.split(separator);
                                line = cols[0];
                                // join the wanted number of columns:
                                for (int i = 1; i < numOfColumns; i++)
                                        line += separator + cols[i];
                                // output the first numOfColumns:
                                System.out.println(line);
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                } catch (ArrayIndexOutOfBoundsException e) {
                        e.printStackTrace();
                        System.err.println(
                          "\n SUGGESTION: Try to set another (different) separator.");
                }
        }
```

```
	private static void outputLinesSkipped(String fileName,
	                                       int numOfColumns, String skipList) {
		BufferedReader br = getBuffer(fileName);
		try { // read File:
			String line = "";
			// line by line and split the lines:
			while ( (line = br.readLine()) != null ) {
			    String[] cols = line.split(separator);
			    line = "";
			    // join the wanted number of columns:
			    int added = 0;
			    for ( int i=0; i<cols.length && added<numOfColumns; i++ ) {
			        	String add = ";" + String.valueOf(i) + ";";
			        	if ( !skipList.contains(add)  ) {
			        		line += cols[i] + separator;
			        		added++;
			        	}
			    }
			    // output the first numOfColumns (without last separator):
			    System.out.println(line.substring(0,
			                        line.length()-separator.length()));

			}
		} catch (IOException e) {
			e.printStackTrace();
		} catch (ArrayIndexOutOfBoundsException e) {
			e.printStackTrace();
			System.err.println(
			  "\n SUGGESTION: Try to set another (different) separator.");
		}
	}



	public static void main(String[] args) throws IOException {
		if (args.length < 2)
			System.out.println(
			"usage:\n java [-XmsXXXm/g] GetColumns NumberOfColumns file
			                               [SkipColumns List(optional)]");
		else {
			String fileName = args[1];
			int numOfColumns = Integer.parseInt(args[0]);

			if ( args.length > 2 ) { // skip this columns:
				String skips = "";
				for (int i = 2; i < args.length; i++)
					skips += ";" + args[i] + ";";
				outputLinesSkipped(fileName, numOfColumns, skips);
			}
			else {
				outputLines(fileName, numOfColumns);
			}

		}
	}
}
```

D.10.4 The *configIPslaves.sh* Script

```
#!/bin/bash
#############################################
# Automate Hadoop Multi-Node configuration
# 2011 by Rui Pais
#############################################

echo -e "Wait please... \n checking $@ as slave(s) ..."


## Set IPs of master and slaves:

IPMASTER=`ifconfig |grep Bcast|sed 's/Bcast:/\n:/'|sed '/^:/ d'|sed 's/inet
addr:/\n/g' |sed '/^ / d'`

HOSTFILE="/etc/hosts"
SLAVESFILE="/usr/local/hadoop/conf/slaves"

cp /etc/hosts "$HOSTFILE.bakup"

echo -e "\n# alias for nodes:" >> $HOSTFILE
echo "$IPMASTER master" >> $HOSTFILE
echo "master" > $SLAVESFILE

N=1
for arg in "$@"; do
        [[ `ping -c1 $arg |grep " 0% "` ]] &&
                echo "$arg  slave$N">>$HOSTFILE && echo "slave$N" >> $SLAVESFILE &&
N=$(( $N + 1 ))
done

## Propagate IP alias on network:

for arg in "$@"; do
        scp $HOSTFILE root@$arg:/etc/
done

## Set nodes replication number on hdfs-site.xml conf file:

HDFSXML=/usr/local/hadoop/conf/hdfs-site.xml

cp -a $HDFSXML $HDFSXML.backup
sed '11,99d' $HDFSXML.backup > $HDFSXML

echo "  <value>$N</value>" >> $HDFSXML
echo "</property>" >> $HDFSXML
echo "</configuration>" >> $HDFSXML

## output on exit:

echo -e "\nAdded $N slave(s) to conf files.\n"
tail -n $(( $N + 1 )) $HOSTFILE
```

D.10.5 The *start_ curio.sh* Script

For the simplicity of use the following script was created to easily launch any of the implemented distributed algorithms and retrieve the results from the distributed system:

```
#!/bin/bash
########################################
# Simplify launch of CurioD Algorithms
# 2011 by Rui Pais
########################################

if [[ $1 -eq "" || $2 -eq "" ]]; then
  echo -e "\n usage: $0 P T [Algorithm] [min Max](opt)\n"
  exit -1;
fi

HADOOP=/usr/local/hadoop
HADOOP_VERSION=hadoop-0.20.2
echo -e "\n Recompile CurioD ..."
javac -classpath $HADOOP/$HADOOP_VERSION-core.jar:$HADOOP/lib/commons-cli-1.2.jar -d
 bin src/*.java

echo -e "\n Build a fresh CurioD jar..."
jar -cvf curioXD.jar -C bin/ . > .log

echo -e "\n Script will clean old outputs..."
hadoop fs -rmr output
hadoop fs -rmr output_Extrema_tmp
hadoop fs -rmr output_tmp_CellsCounts
hadoop fs -rmr output_tmp_ValidateCells
hadoop fs -rmr output_CellsListNeighbors

echo -e "\nPrecision P = $1"
echo "Tolerance T = $2"

echo -e "\n Start Hadoop Jobs:" CURIO_VERSION=$3
if [[ $CURIO_VERSION == "" ]]; then
  CURIO_VERSION=Curio3XD
fi
echo "Algorithm version: $CURIO_VERSION"

hadoop jar curioXD.jar $CURIO_VERSION input output $1 $2 $4 $5

#hadoop fs -cat output/part-r-* > part-r-output
rm part-r-output_$CURIO_VERSION
hadoop fs -getmerge output "part-r-output_$CURIO_VERSION"

#cat part-r-output
echo -e "\n Output merged to local file: part-r-output_$CURIO_VERSION \n"
echo "total lines/cells: `wc -l part-r-output_$CURIO_VERSION`"
echo "file size: `wc -c part-r-output_$CURIO_VERSION` bytes"
```

Bibliography

[1] Charu C Aggarwal and Philip S Yu. An effective and efficient algorithm for high-dimensional outlier detection. *The VLDB Journal*, 14(2):211–221, 2005.

[2] Andreas Arning and Prabhakar Raghavan. A Linear Method for Deviation Detection in Large Databases. In *Complexity*, pages 164–169. AAAI Press, 1996.

[3] Stephen D Bay and Mark Schwabacher. Mining Distance-Based Outliers in Near Linear Time with Randomization and a Simple Pruning Rule. *Sciences-New York*, , 2003.

[4] Irad Ben-gal. Chapter 1 OUTLIER DETECTION. *Industrial Engineering*, 37(2):1–16, 2005.

[5] Dhruba Borthakur. The Hadoop Distributed File System : Architecture and Design. *Access*, :1–14, 2007.

[6] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng and Jörg Sander. LOF : Identifying Density-Based Local Outliers. *Sigmod Record*, 29(2):1–12, 2000.

[7] Vic Barnett and Toby Lewis. *Outliers in Statistical Data*, volume 3 of *Wiley series in probability and mathematical statistics*. Wiley, 1994.

[8] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice Hall, 1999.

[9] Aaron Ceglar, John F Roddick and David M W Powers. CURIO : A Fast Outlier and Outlier Cluster Detection Algorithm for Large Datasets. *Proceedings of the 2nd international workshop on Integrating artificial intelligence and data mining*, 84(2007):39–48, 2007.

[10] Varun Chandola, A Banerjee and V Kumar. Outlier detection-A survey. *ACM Computing Surveys*, , 2009.

[11] Amitabh Chaudhary, Alexander S Szalay and Andrew W Moore. Very Fast Outlier Detection in Large Multidimensional Data Sets K-d Tree based Outlier Detection. *Proceedings of ACM SIGMOD Workshop in Research Issues in Data Mining and Knowledge Discovery DMKD*, , 2002.

[12] A. L. C. Chiu and Ada Wai-Chee Fu. Enhancements on local outlier detection. *Seventh International Database Engineering and Applications Symposium, 2003. Proceedings.*, :298–307, 2003.

[13] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin and Andrew Y Ng. Map-Reduce for Machine Learning on Multicore. *Architecture*, 19(23):281, 2007.

[14] Cloudera. Palo Alto. CDH3 Installation Guide. In Cloudera, editor, *https://ccp.cloudera.com/display/CDHDOC/CDH3+Installation+Guide*. 2009.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):1–13, 2008.

[16] Amazon AWS Documentation. Amazon Elastic Cloud Computing EC2. In *http://aws.amazon.com/documentation/ec2*. Amazon, 2011.

[17] Amazon AWS Documentation. Creating your own AMIs. In *http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/creating-an-ami.html*. Amazon, 2011.

[18] Jaliya Ekanayake, Shrideep Pallickara and Geoffrey Fox. MapReduce for Data Intensive Scientific Analyses. *2008 IEEE Fourth International Conference on eScience*, 0:277–284, 2008.

[19] Martin Ester, Hans-Peter Kriegel and Xiaowei Xu. A database interface for clustering in large spatial databases. *Interface*, KDD-95(1993):181–196, 1995.

[20] Rui Maximo Esteves, Rui Pais and Chunming Rong. K-means clustering in the cloud - a Mahout test. *CCS2011*, :514–519, 2011.

[21] P Filzmoser, R Maronna and Mark Werner. Outlier identification in high dimensions. *Computational Statistics & Data Analysis*, 52(3):1694–1711, 2008.

[22] Ian Foster, Yong Zhao, Ioan Raicu and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. *2008 Grid Computing Environments Workshop*, abs/0901.0(5):1–10, 2008.

[23] Apache Software Foundation. Welcome to Apache Hadoop! In Apache, editor, *http://hadoop.apache.org*. Apache Software Foundation, 2011.

[24] V J Hodge and J Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[25] U Kang, Charalampos Tsourakakis and Ana Paula Appel. HADI : Fast Diameter Estimation and Mining in Massive Graphs with Hadoop. *Science*, 8(December), 2008.

[26] Edwin M Knorr and Raymond T Ng. Algorithms for Mining Distance-Based Outliers in Large Datasets. In Ashish Gupta, Oded Shmueli and Jennifer Widom, editors, *Proceedings of the 24th VLDB Conference*, volume 98, pages 392–403. Citeseer, Morgan Kaufmann Publishers Inc., 1998.

[27] Edwin M Knorr, Raymond T Ng and Vladimir Tucakov. Distance-based outliers: algorithms and applications. *The VLDB Journal The International Journal on Very Large Data Bases*, 8(3-4):237–253, 2000.

[28] Anna Koufakou and Michael Georgiopoulos. A fast outlier detection strategy for distributed high-dimensional data sets with mixed attributes. *Data Mining and Knowledge Discovery*, 20(2):259–289, nov 2009.

[29] Anna; Koufakou, Jimmy; Secretan, John; Reeder, Kelvin; Cardona and Michael; Gerogiopoulos. Fast Parallel Outlier Detection for Categorical Datasets using MapReduce. *Neural Networks*, Internatio(IJCNN 2008):3297–3303, 2008.

[30] Chuck Lam. *Hadoop in Action*. Manning Early Access Program, 2010.

[31] Apache Mahout. Mahout on Amazon EC2. In *https://cwiki.apache.org/confluence/display/MAHOUT/Mahout+on+Amazon+EC2*. Apache Software Foundation, 2011.

[32] Raymond T Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–155. Citeseer, 1994.

[33] Michael G Noll. Running Hadoop On Ubuntu Linux. In *http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster*, pages 1–19. Michael Noll, 2010.

[34] Matthew Eric Otey, Amol Ghoting and Srinivasan Parthasarathy. Fast Distributed Outlier Detection in Mixed-Attribute Data Sets. *Data Mining and Knowledge Discovery*, 12(2-3):203–228, apr 2006.

[35] S Papadimitriou, H Kitagawa, P B Gibbons and C Faloutsos. LOCI: fast outlier detection using the local correlation integral. *Proceedings 19th International Conference on Data Engineering Cat No03CH37405*, 1063(6382/03):315–326, 2002.

[36] S Papadimitriou and J Sun. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining. *2008 Eighth IEEE International Conference on Data Mining*, :512–521, 2008.

[37] Sridhar Ramaswamy, Rajeev Rastogi and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. *ACM SIGMOD Record*, 29(2):427–438, jun 2000.

[38] Shashi Shekhar, Chang-Tien Lu and Pusheng Zhang. A Unified Approach to Detecting Spatial Outliers. *Adelphi Papers*, 7(2):139–166, 2003.

[39] S J Stolfo. KDD cup 1999 dataset. *UCI KDD repository http.kdd.ics.uci.edu*, , 1999.

[40] Biplab Kumer Sarker and Hiroyuki Kitagawa. *A Distributed Algorithm for Outlier Detection in a Large Database*. 2005.

[41] Yufei Tao, Xiaokui Xiao and Shuigeng Zhou. Mining distance-based outliers from large databases in any metric space. *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, :394–403, 2006.

[42] Tom White. *Hadoop: The Definitive Guide*, volume 54. Yahoo Press, 2009.

[43] Tian Zhang, Raghu Ramakrishnan and Miron Livny. BIRCH: an efficient data clustering method for very large databases. *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, 1(2):103–114, 1996.

[44] Jiaogen Zhou, Chunjiang Zhao, You Wan, Wenjiang Huang, Baozhu Yang and Jixin Ge. *A Novel Outlier Detection Algorithm for Distributed Databases*, volume 5. IEEE, 2008.