Elephant against Goliath: Performance of Big Data versus High-Performance Computing DBSCAN Clustering Implementations

Helmut Neukirchen[0000−0001−8595−3748]

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, School of Engineering and Natural Sciences, University of Iceland, Reykjavik, Iceland
helmut@hi.is

Abstract. Data is often mined using clustering algorithms such as *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN). However, clustering is computationally expensive and thus for big data, parallel processing is required. The two prevalent paradigms for parallel processing are *High-Performance Computing* (HPC) based on *Message Passing Interface* (MPI) or *Open Multi-Processing* (OpenMP) and the newer big data frameworks such as Apache Spark or Hadoop. This paper surveys for these two different paradigms publicly available implementations that aim at parallelizing DBSCAN and compares their performance. As a result, it is found that the big data implementations are not yet mature and in particular for skewed data, the implementation's decomposition of the input data into parallel tasks has a huge influence on the performance in terms of run-time due to load imbalance.

1 Introduction

Computationally intensive problems, such as simulations, require parallel processing. Some problems are embarrassingly parallel (such as the many but rather small problems [1] resulting from the *Large Hadron Collider* (LHC) experiments) – most computational problems in simulation are, however, tightly coupled (such as, e.g., finite element modelling which may even involve model coupling [2]). The standard approach in this case is *High-Performance Computing* (HPC).

Highly praised contenders for huge parallel processing problems are big data processing frameworks such as Apache Hadoop or Apache Spark. Hence, they might be considered an alternative to HPC for distributed simulations. We have already shown [3] that for a typical embarrassingly parallel LHC simulation, an Apache Hadoop-based distributed processing approach is almost on par with the standard distributed processing approach used at LHC.

In this paper, we want to use a more tightly coupled parallel processing problem to investigate whether HPC or big data platforms are better suited for computationally intensive problems that involve some tight coupling: clustering using the *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) [4] clustering algorithm.

DBSCAN is computationally expensive and thus for clustering big data, parallel processing is required. However, the DBSCAN algorithm has been defined as a serial, non-parallel algorithm. Hence, several parallel variants of DBSCAN have been suggested. The main contribution of this paper is to investigate the run-time performance and scalability of different publicly available parallel DBSCAN implementations running either on HPC platforms or on big data platforms such as the MapReduce-based Apache Hadoop or the *Resilient Distributed Dataset* (RDD)-based Apache Spark.

The Bible's book of Samuel and chapter 2 of the Qur'an contain the story of the giant warrior Goliath (Jalut in Arabic): HPC clusters can be considered as such old giants. The graphical logo of the first popular big data platform, Apache Hadoop [5], is an elephant. The question whether HPC or big data is better can therefore be compared to a fight between Goliath and an elephant. – But beware: later, we encounter even ogres!

The outline of this paper is as follows: subsequent to this introduction, we provide foundations on DBSCAN, HPC and big data. Afterwards, in Section 3, we describe as related work other comparison of algorithms running on HPC and big data platforms as well as a comparison of non-parallel implementations of DBSCAN. In Section 4, we survey existing DBSCAN implementations. Those implementations that were publicly available are evaluated with respect to their run-time in Section 5. We conclude with a summary and an outlook in Section 6. This full paper is based on an extended abstract [6] and a technical report [7]. An annex with command line details can be found in a EUDAT B2SHARE persistent electronic record [8].

2 Foundations

2.1 DBSCAN

The spatial clustering algorithm *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) [4] has the nice properties that the number of clusters needs not to be known in advance, but is rather automatically determined; that it is almost independent from the shape of the clusters; and that it can deal with and filter out noise. Basically, the underlying idea is that for each data point, the neighbourhood within a given *eps* radius has to contain at least *minpts* points to form a cluster, otherwise it is considered as noise.

In the simplest implementation, finding all points which are in the *eps* neighbourhood of the currently considered point, requires to check all remaining $n-1$ points of the input data: doing this for each of the n input points leads to a complexity of $O(n^2)$. Using spatially sorted data structures for the *eps* neighbourhood search, such as R-trees [9], R*-trees [10], or kd-trees [11], reduces the overall complexity to $O(n \log n)$ if *eps* is reasonably small[1]. The standard algorithms to

[1] If the *eps* neighbourhood contains, e.g., all data points, the complexity of DBSCAN grows obviously again to $O(n^2)$ despite these tree-based approaches.

populate such spatially sorted data structures cannot run in parallel and require in particular to have the entire input data available in non-distributed memory.

Even if the problem of having a distributed, parallel-processing variant of populating and using a spatially sorted data structure (in order to bring the overall complexity down to $O(n \log n)$) is solved, there are further obstacles in parallelizing DBSCAN in a way that it scales optimally.

The actual clustering can be easily parallelized by partitioning the data spatially: typically, all points that belong to the same partition or cell (=a rectangle in case of 2 dimensional data, a cube in case of 3D data, or a hypercube for $n \geq 3$ dimensions) can be processed by one thread independently from other threads that process the remaining partitions of points. Only at the border of each rectangle/cube/hypercube, points from direct neighbour rectangles/cubes/hypercubes need to be considered up to a distance of *eps*. For this, the standard approach of ghost or halo regions [12] can be applied, meaning that these points from a neighbour partition need to be accessible as well by the current thread (in case of distributed memory, this requires to copy them into the memory of the respective thread). In a final step, those clusters determined locally in each partition which form a bigger cluster spanning multiple partitions need to be merged.

To achieve a maximum speed-up, not only an efficient spatially sorted data structure and low communication overhead (e.g. for halo regions or finally merging locally obtained clusters), but also the size of the partitions is crucial: the input domain needs to be decomposed so that each thread or processor core get an equal share of the work. The simple approach of dividing the input domain into spatially equally sized chunks (for example as many chunks as processor cores are available) yields imbalanced workloads for the different cores if the input data is skewed: some partitions may then be almost empty, others very crowded. For heavily skewed data, the spatial size of each partition needs to be rather adjusted, for example in a way that each partition contains an equal number of points or the same number of comparisons are performed in each partition. If ghost/halo regions are used, then also the number of points in these regions need to be considered, because they also need to be compared by a thread processing that partition.

As shown, parallelizing DBSCAN in a scalable way beyond a trivial number of parallel threads (or processing nodes respectively) or problem size is a challenge. For example, PDBSCAN [13] is a parallelized DBSCAN, however it involves a central master node to aggregate intermediate results which can be a bottleneck with respect to scalability. In particular, when processing "big data" (i.e. n is huge), the implementation with the standard complexity of $O(n^2)$ will be too slow and rather $O(n \log n)$ algorithms are needed.

2.2 HPC

High-Performance Computing (HPC) is tailored to CPU-bound problems. Hence, special and rather expensive hardware is used, e.g. compute nodes containing fast CPUs including many cores and large amounts of RAM, very fast interconnects (e.g. InfiniBand) for communication between nodes, and a centralised *Storage-Area Network* (SAN) with high bandwith due to a huge *Redundant Array of Independent Disks* (RAID) and fast attachment of them to the nodes.

To make use of the multiple cores per CPU, typically shared-memory multi-threading based on *Open Multi-Processing* (OpenMP) [14] is applied. To make use of the many nodes connected via the interconnects, an implementation of the *Message Passing Interface* (MPI) [15] is used. The underlying programming model (in particular of MPI) is rather low-level: the domain decomposition of the input data, all parallel processing, the synchronisation and communication needed for tight coupling has to be explicitly programmed. Typically rather low-level, but fast programming languages such as C, C++ and Fortran are used in the HPC domain. In addition to message passing, MPI supports parallel I/O to read different file sections from the SAN in parallel into the different nodes. To this aim, parallel file systems such as Lustre [16] or the *General Parallel File System* (GPFS) [17] provide a high aggregated storage bandwidth. Typically, binary file formats such as netCDF or the *Hierarchical Data Format* (HDF) [18] are used for storing input and output data in a structured way. They come with access libraries that are tailored to MPI parallel I/O.

While the low-level approach allows fast and tightly coupled implementations, their implementation takes considerable time. Furthermore, no fault tolerance is included: a single failure on one of the many cores will cause the whole HPC job to fail which then needs to be restarted from the beginning if no check-pointing has been implemented. However, due to the server-grade hardware, hardware failures are considered to occur seldom (but still, they occur).

2.3 Big Data

The big data paradigm is tailored to process huge amounts of data, however the actual computations to be performed on this data are often not that computationally intensive. Hence, cheap commodity hardware is sufficient for most applications. Being rather I/O-bound than CPU-bound, the focus is on *High-Throughput Computing* (HTC). To achieve high-throughput, locality of data storage is exploited by using distributed file systems storing locally on each node a part of the data. The big data approach aims at doing computations on those nodes where the data is locally available. By this means, slow network communication can be minimised. (Which is crucial, because rather slow Ethernet is used in comparison to the fast interconnects in HPC.)

An example distributed file system is the *Hadoop Distributed File System* (HDFS), introduced with one of the first open-source big data frameworks, Apache Hadoop [5] which is based on the MapReduce paradigm [19]. As it is intended for huge amounts of data, the typical block size is 64 MB or 128 MB. Hadoop has the disadvantage that only the MapReduce paradigm is supported which restricts the possible class of parallel algorithms and in particular may lead to unnecessarily storing intermediate data on disk instead of allowing to keep it in fast RAM. This weakness is overcome by Apache Spark [20] which is based on *Resilient Distributed Dataset*s (RDDs) [21] which are able to store a whole data set in RAM distributed in partitions over the nodes of a cluster. A new RDD can be obtained by applying transformations in parallel on all input RDD partitions. To achieve fault tolerance, an RDD can be reconstructed by re-playing transformations on those input RDDs partitions that survived a failure. The initial RDD is obtained by reads from HDFS. While RDDs are kept in RAM, required data may not be available locally in the RDD partition of a node. In this case, it is necessary to re-distribute data between nodes. Such shuffle operations are expensive, because slow network transfers are needed for them. Typically, textual file formats, such as *Comma-Separated Values* (CSV) are used that can be easily split to form the partitions on the different nodes.

High-level, but (in comparison to C/C++ or Fortran) slower languages such as Java or the even more high-level Scala or Python are used in big data frameworks. Scala has over Python the advantage that it is compiled into Java bytecode and is thus natively executed by the *Java Virtual Machine* (JVM) running the Hadoop and Spark frameworks. While the code to be executed is written as a serial code, the big data frameworks take behind the scenes care that each node applies in parallel the same code to the different partitions of the data.

Because commodity hardware is used which is more error prone than HPC server-grade hardware, big data approaches need to anticipate failures as the norm and have thus built-in fault tolerance, such as redundant data storage or restarting failed jobs.

2.4 Convergence of HPC and Big Data

Convergence of HPC and big data approaches is taking place in both directions: typically either in form of *High-Performance Data Analysis* (HPDA), meaning that HPC is used in domains that used to be the realm of big data platforms, or big data platforms are used in the realm of HPC. Alternatively, a mixture is possible: big data platforms are deployed on HPC clusters, however, sacrificing data locality-aware scheduling [22]. In this paper, we investigate how mature this convergence is by comparing DBSCAN clustering implementations for HPC and for big data platforms.

3 Related Work

HPC and big data data implementations for the same algorithms have been studied before. Jha et al. [22] compare these two parallel processing paradigms in general and introduce “Big Data Ogres” which refer to different computing problem areas with clustering being one of them. In particular, they evaluate and compare the performance of k-means clustering [23] implementations for MPI-based HPC, for MapReduce-based big data platforms such as Hadoop and HARP (which introduces MPI-like operations into Hadoop), and for the RDD-based Spark big data platform. In their evaluation, the considered HPC MPI k-means implementation outperforms the other more big data-related implementation with the implementation based on HARP being second fastest and the implementation for Spark ranking third.

A further performance comparison of a big data implementation and a traditional parallel implementation has been done by us [3] with respect to a typical embarrassingly parallel High Energy Physics analysis: as traditional embarrassingly parallel execution platform, the *Parallel ROOT Facility* (PROOF) [24] has been compared to using Apache Hadoop for this analysis: while the Hadoop-based implementation is slightly slower, it offers fault tolerance.

The influence of data storage locality as exploited by Spark and other big data platforms compared to centralized HPC SAN storage has been investigated by Wang et al. [25]. They use data intensive Grep search and compute intensive logistic regression as case study and come to the conclusion that even with a fast 47 GB/s bandwith centralized Lustre SAN, data locality matters for Grep and thus accesses to local SSDs are faster. However, for the logistic regression, locality of I/O does not matter.

Kriegel et al. [26] stresses the scientific value of benchmarking and evaluates in particular the run-time of serial, non-parallel implementations of DBSCAN. Concerning different programming languages, they show that a C++ implementation is one order of magnitude faster than a comparable Java implementation. They also observed a four orders of magnitude speed difference between different serial implementations of DBSCAN.

4 Survey of Parallel DBSCAN Implementations

This section contains a survey of the considered DBSCAN implementations. To be able to compare their run-time performance on the same hardware and using the same input, only open-source implementations have been considered.

For comparison, we also used also ELKI 0.7.1 [27], an *Environment for DeveLoping KDD-Applications supported by Index-Structures*. ELKI is an optimised serial open-source DBSCAN implementation in Java which employs R*-trees to achieve $O(n \log n)$ performance. By default, ELKI reads space- or comma-separated values and it supports arbitrary dimensions of the input data.

4.1 HPC DBSCAN Implementations

A couple of DBSCAN implementations for HPC platforms exist (as, for example, listed by Patwaryat et al. [28] or Götz et al. [29]). To our knowledge, only for two of them, the source is available: PDSDBSCAN and HPDBSCAN. Thus, we restrict in the following to these two. Both support arbitrary data dimensions.

PDSDBSCAN by Patwary et al. [28] (C++ source code available on request from the PDSDBSCAN first author [30]) makes use of parallelization either based on shared memory using OpenMP or based on distributed memory using MPI. For their OpenMP variant, the the input data needs to fit into the available RAM; for the MPI variant, a pre-processing step is required to partition the input data onto the distributed memory. Details of this pre-processing step are not documented as the authors do not consider this step as part of their algorithm and thus, it is neither parallelized nor taken into account when they measure their running times. Input data is read via the netCDF I/O library.

HPDBSCAN by Götz et al. [29] (C++ source code available from repository [31]) makes use of parallelization based on shared memory and/or distributed memory: besides a pure OpenMP and pure MPI mode, also a hybrid mode is supported. This is practically relevant, because an HPC cluster is a combination of both memory types (each node has RAM shared by multiple cores of the same node, but RAM is not shared between the many distributed nodes) and thus, a hybrid mode is most promising to achieve high performance. For the domain decomposition and to obtain a spatially sorted data structure with $O(\log n)$ access complexity, the arbitrary ordered input data is first indexed in a parallel way and then re-distributed so that each parallel processor has points in the local memory which belong to the same spatial partition. It is the only implementation that sizes the partitions using a cost function that is based on the number of comparisons (=number of pairs for which the distance function needs to be calculated) to be made for the resulting partition size: this obviously also includes points in adjacent ghost/halo regions. The command line version of the implementation reads the input data via the HDF I/O library.

4.2 Spark DBSCAN Implementations

Even though our search for Apache Spark big data implementations of DBSCAN[2] was restricted to JVM-based Java or Scala[3] candidates, we found several parallel open-source[4] implementations of DBSCAN: Spark DBSCAN, RDD-DBSCAN, Spark_DBSCAN, and DBSCAN On Spark. They are described in the following.

Spark DBSCAN by Litouka (source code via repository [35]) is declared as experimental and being not well optimised. For the domain decomposition, the data set is considered initially as a large box full of points. This box is then along its longest dimension split into two parts containing approximately the same number of points. Each of these boxes is then split again recursively until the number of points in a box becomes less than or equal to a threshold, or a maximum number of levels is reached, or the shortest side of a box becomes

[2] Remarkably, the machine learning library MLlib which is a part of Apache Spark does not contain DBSCAN implementations.

[3] Note that also purely serial Scala implementations of DBSCAN are available, for example GSBSCAN from the Nak machine learning library [32]. However, these obviously make not use of Apache Spark parallel processing. But still, they can be used from within Apache Spark code to call these implementations in parallel, however each does then cluster unrelated data sets [33].

[4] There is another promising DBSCAN implementation for Spark by Han et al. [34]: A kd-tree is used to obtain $O(n \log n)$ complexity. Concerning the partitioning, the authors state "We did not partition data points based on the neighborhood relationship in our work and that might cause workload to be unbalanced. So, in the future, we will consider partitioning the input data points before they are assigned to executors." [34]. However, it was not possible to benchmark it as is not available as open-source.

smaller than 2 *eps* [35]. Each such a box becomes a record of an RDD which can be processed in parallel, thus yielding a time complexity of $O(m^2)$ for that parallel processing step with m being the number of points per box [36].

RDD-DBSCAN by Cordova and Moh [37] (source code via repository [38]) is loosely based on MR-DBSCAN [39]. Just as the above Spark DBSCAN by Litouka, the data space is split into boxes that contain roughly the same amount of data points until the number of points in a box becomes less than a threshold or the shortest side of a box becomes smaller than 2 *eps*. R-trees are used to achieve an overall $O(n \log n)$ complexity [37].

Spark_DBSCAN (source code via repository [40]) is a very simple implementation (just 98 lines of Scala code) and was not considered any further, because of its complexity being $O(n^2)$ [36].

DBSCAN On Spark by Raad (source code via repository [41]) uses for domain decomposition a fixed grid independent from how many points are contained in each resulting grid cell. Furthermore, to reduce the complexity, no Euclidian distance function is used (which would be a circle with 2 *eps* diameter), but the square box grid cells (with 2 *eps* edge length) themselves are rather used to decide concerning neighbourhood (see function `findNeighbors` in [41]). So, while it is called "DBSCAN On Spark" it implements only an approximation of the DBSCAN algorithm and does in fact return wrong clustering results.

Common features and limitations of the Spark Implementations All the considered implementations of DBSCAN for big data platforms assume the data to be in CSV or space-separated format.

All the Apache Spark DBSCAN implementations (except for the closed-source DBSCAN by Han et al. [34]) work only on 2D data: On the one hand, the partitioning schemes used for decomposition are based on rectangles instead of higher-dimensional hyper-cubes. On the other hand, for calculating the distance between points, most implementations use a hard-coded 2D-only implementation of calculating the Euclidian distance[5].

Most Spark-based implementations aim at load balancing by having an approximately equal number of data points in each partition and a partition may not get smaller than 2 *eps*.

4.3 MapReduce DBSCAN Implementations

For further comparison, it would have been interesting to evaluate in addition also MapReduce-based DBSCAN implementations for the Apache Hadoop platform; candidates found to be worthwhile (because they claim to be able to deal with skewed data) were MR-DBSCAN [42, 39] by He et al. and DBSCAN-MR [43] by Dai and Lin. However, none of these implementations were available

[5] Note that spheric distances of longitude/latitude points should in general not be calculated using Euclidian distance in the plane. However, as long as these points are sufficiently close together, clustering based on the simpler and faster to calculate Euclidian distance is considered as appropriate.

as open-source and e-mail requests to the respective first authors to provide their implementations either as source code or as binary were not answered. Hence, it is impossible to validate the performance claims made by these authors.

5 Evaluation of Parallel DBSCAN Implementations

Typically, comparisons between HPC and big data implementations are difficult as the implementations run on different cluster hardware (HPC hardware versus commodity hardware) or cannot exploit underlying assumptions (such as missing local data storage when deploying big data frameworks at run-time on HPC clusters using a SAN).

5.1 Hardware and Software Configuration

In this paper, the same hardware is used for HPC and Spark runs: the cluster JUDGE at Jülich Supercomputing Centre. JUDGE was formerly used for HPC and has been turned into a big data cluster. It consists of IBM System x iDataPlex dx360 M3 compute nodes each comprising two Intel Xeon X5650 (Westmere) 6-core processors running at 2.66 GHz. For the big data evaluation, we were able to use 39 executor nodes, each having 12 cores or 24 virtual cores with hyper-threading enabled (=936 virtual cores) and 42 GB of usable RAM per node and local hard disk.

In the HPC configuration, a network-attached GPFS storage system, the JUelich STorage cluster JUST, was used to access data (measured peak performance of 160 GB/s), and the CPU nodes were connected via an Infiniband interconnect. The big data configuration relies on local storage provided on each node by a Western Digital WD2502ABYS-23B7A hard disk (with peak performance of 222.9 MB/s per disk, corresponding to 8.7 GB/s total bandwidth if all 39 nodes read their local disk in parallel). 200 GB on each disk were dedicated to HDFS using a replication factor of 2 and 128 MB HDFS block size. The CPU nodes were connected via Ethernet network connections.

The software setup in the HPC configuration was SUSE Linux SLES 11 with kernel version 2.6.32.59-0.7. The MPI distribution was MPICH2 in version 1.2.1p1. For accessing HDF5 files, the HDF group's reference implementation version 1.8.14 was used. The compiler was gcc 4.9.2 using optimisation level O3.

The big data software configuration was deployed using the Cloudera CDH 5.8.0 distribution providing Apache Spark version 1.6.0 and Apache Hadoop (including HDFS and YARN which was used as resource manager) version 2.6.0 running on a 64-Bit Java 1.7.0_67 VM. The operating system was CentOS Linux release 7.2.1511.

5.2 Input Data

Instead of using artificial data, a real data set containing skewed data was used for evaluating the DBSCAN implementations: geo-tagged tweets from a rectangle around the United Kingdom and Ireland (including a corner of France)

Table 1. Size of the Used Twitter Data Sets

	Data points	HDF5 size	CSV size	SSV with Ids
Twitter Small	3 704 351	57 MB	67 MB	88 MB
Twitter Big	16 602 137	254 MB	289 MB	390 MB

Table 2. Used Open Source Repository Versions

Implementation	Repository	Version date
HPDBSCAN	`bitbucket.org/markus.goetz/hpdbscan`	2015-09-10
Spark DBSCAN	`github.com/alitouka/spark_dbscan`	22 Feb 2015
RDD DBSCAN	`github.com/irvingc/dbscan-on-spark`	14 Jun 2016
DBSCAN on Spark	`github.com/mraad/dbscan-spark`	30 Jan 2016

in the first week of June 2014. The data was obtained by Junjun Yin from the National Center for Supercomputing Application (NCSA) using the Twitter streaming API. This data set contains 3 704 351 longitude/latitude points and is available at the scientific data storage and sharing platform B2SHARE [44]. There, the data is contained in file `twitterSmall.h5.h5`. A bigger Twitter data set `twitter.h5.h5` from the same B2SHARE location covers whole of June 2014 containing of 16 602 137 data points, some of them are bogus artefacts though (Twitter spam) – still we used it to check whether implementations are able to cope with bigger data sets; a 3D point cloud data set for the city of Bremen is also provided there, however it was not usable for benchmarking the surveyed DBSCAN implementations for Spark which typically support only 2D data.

The original file of the small Twitter data set is in HDF5 format and 57 MB in size. To be readable by ELKI and the Spark DBSCAN implementations, it has been converted using the `h5dump` tool (available from the HDF group) into a 67 MB CSV version and into an 88 MB *Space-Separated Values* (SSV) version that contains in the first column an increasing integer number as point identifier (expected by some of the evaluated DBSCAN implementations). The size of these two data sets is summarised in Table 1.

For all runs, $eps = 0.01$ and $minpts = 40$ were used as parameters of DBSCAN. The detailed command line parameters can be found in EUDAT [8].

5.3 DBSCAN Implementation Versions

The dates of the used DBSCAN implementation source code versions and their repository are provided in Table 2. The source code was used unmodified except for one change: by default, Spark makes each HDFS block of the input file a separate partition of the input RDD. With the above file sizes of the small Twitter data set being lower than the used HDFS block size of 128 MB, the initial RDD would contain just a single partition located on the node storing the corresponding HDFS block. In this case, no parallelism would be used to process the initial RDD. Therefore, if the Spark DBSCAN implementations did not anyway allow to specify the number of partitions to be used, the implementations

Table 3. Deviation of RDD-DBSCAN Run-times for the Same Configuration

Run	1.	2.	3.	4.
Time (s)	653	546	553	560

were changed so that it is possible to specify the number of partitions to be used for the initial file read. While this means that non-local reads will occur, the overhead of these non-local reads is negligible in particular since it leads to a better degree of parallel processing.

5.4 Measurements

Comparing C++ implementations (PDSDBSCAN and HPDBSCAN) to JVM-based DBSCAN implementations for Spark is somewhat comparing apples and oranges. Hence, we used as a further comparison a Java implementation, the pure serial ELKI (see section 4) with `-db.index "tree.metrical.covertree.SimplifiedCoverTree$Factory"` spatial indexing option running just on one of the cluster cores. The times were measured using the POSIX command `time`.

As usual in Spark, the Spark DBSCAN implementations create the output in parallel resulting in one file per parallel RDD output partition. If a single output file is intended, it can be merged afterwards, however this time is not included in the measurement. The reported times were taken from the “Elapsed” line of the application’s entry in the Spark web user interface for the completed run.

For the number of experiments that we did, we could not afford to re-run all of them multiple times to obtain averages or medians. However, for one scenario (RDD-DBSCAN, 233 executors, each using 4 cores with 912 initial partitions running on the small Twitter data set), we repeated execution four times. The observed run-times are shown in Table 3. For these 9–10 minute jobs, deviations of up to almost 2 minutes occurred. The fact that the first run was slower than the subsequent runs might be attributed to caching of the input file. In all of our experiments, we had exclusive use of the assigned cores.

Preparatory Measurements In addition to the DBSCAN parameters *eps* and *minpts*, the parallel Spark-based implementations are influenced by a couple of parallelism parameters which were determined first as described below.

Spark uses the concepts of *executors* with a certain number of threads per executor process. A couple of sample measurements using a number of threads per executor ranging from 3 to 22 have been performed and the results ranging from 626 seconds to 775 seconds are within the above deviations, hence the influence of threads per executor is not considered significant. (Most of the following measurements have been made with 8 threads per executor – details can be found in EUDAT [8].)

Parallelism in Spark is influenced by the number of partitions into which an RDD is divided. Therefore, measurements with varying initial partition sizes have been made (in subsequent RDD transformations, the number of partitions

Table 4. Influence of number of points used in domain decomposition

No. of points threshold	4 061	9 000	20 000	25 000	50 000
Times (s)	1 157	823	867	675	846

may however change depending on the DBSCAN implementations). Measurement for RDD-DBSCAN running on the small Twitter data set on the 932 core cluster (not all cores were assigned to executors to leave cores available for cluster management) have been made for a number of initial number of input partitions ranging from 28 to 912. The observed run-times were between 622 seconds and 736 seconds which are all within the above deviation range. Hence, these experiments do not give observable evidence of an optimal number of input partitions. However, in the remainder, it is assumed that making use of the available cores already from the beginning is optimal and hence 912 was used as the initial number of input partitions.

After the input data has been read, the DBSCAN implementations aim at distributing the read data points based on spatial locality: as described in section 4.2, most Spark DBSCAN implementations aim at recursively decomposing the input domain into spatial rectangles that contain approximately an equal number of data points and they stop doing so as soon as a rectangle contains only a certain number of points; however, a rectangle becomes never smaller than 2 *eps* edge length. Assuming that the subsequent clustering steps are also based on 912 partitions, the 3 704 351 points of the small Twitter data set divided by 912 partitions yield 4061 points per partition as a decomposition into an equal number of points. However, due to the fact a rectangle becomes never smaller than 2 *eps* edge length, some rectangles of that size still contain more points (e.g. in the dense-populated London area, some of these 2 *eps* rectangle contain up to 25 000 data points) and thus, the domain decomposition algorithm terminates with some rectangles containing a rather high number of points.

Experiments have been made with different numbers of points used as threshold for the domain decomposition by the Spark-based implementations. As shown in Table 4, a threshold of a minimum of 25 000 data points per rectangle promises fastest execution. This number is also the lowest number that avoids the domain decomposition to terminate splitting rectangles because of reaching the 2 *eps* edge length limit: a naïve explanation would be that all rectangles contain an approximately equal number of points thus leading to load balancing. However, later findings (Section 5.5) show a significant load imbalance.

Run-time Measurements on Small Data Set After the above parameters have been determined, a comparison of the run-times of the different implementations was made when clustering the small Twitter data set while increasing the number of cores and keeping the problem size fixed ("strong scaling").

Table 5 shows results using a lower number of cores in parallel. The C++ implementation HPDBSCAN (running in MPI only mode) performs best in all cases and scales well: even with just one core, only 114 seconds are needed to

Table 5. Run-time (in Seconds) on Small Twitter Data Set vs. Number of Cores

Number of cores	1	2	4	8	16	32
HPDBSCAN MPI	114	59	30	16	8	6
PDSDBSCAN MPI	288	162	106	90	85	88
ELKI	997	–	–	–	–	–
RDD-DBSCAN	7 311	3 521	1 994	1 219	889	832
DBSCAN on Spark(*)	1 105	574	330	174	150	147

(*) Does only implement an approximation of the DBSCAN algorithm.

Table 6. Run-time (in Seconds) on Small Twitter Data Set vs. Number of Cores

Number of cores	58	116	232	464	928
Spark DBSCAN	–	–	–	–	2 406
RDD-DBSCAN	622	707	663	624	675
DBSCAN on Spark(*)	169	167	173	183	208

(*) Does only implement an approximation of the DBSCAN algorithm.

cluster the small Twitter data set. Second in terms of run-time is C++ PDSDBSCAN (MPI variant), however, the scalability beyond 8 cores is already limited.

Even though the Java ELKI is optimised for serial execution, it is much slower than the parallel C++ implementations using a single core only. All implementations for Spark are much slower when using a single core only. (Spark DBSCAN was not measured using a low number of cores, because already with a high number of cores it was very slow.) When running on many cores, the Spark-based implementations beat the serial ELKI but are still by one (DBSCAN on Spark) or two (RDD-DBSCAN) orders of magnitude slower than HPDBSCAN and do not scale as well. While DBSCAN on Spark is faster than RDD-DBSCAN, it does only implement a simple approximation of DBSCAN and thus delivers completely different (=wrong) clusters than correct DBSCAN implementations.

Table 6 shows results using a higher number of cores. (No measurements of any of the two DBSCAN HPC implementations on the small Twitter data set have been made, as we can already see from Table 5 that using a higher number of cores does not give any gains on this small data set – measurements with many cores for HPDBSCAN running on the bigger Twitter data set are presented later). For Spark DBSCAN, an initial experiment has been made using 928 cores, but as it was rather slow, so no further experiments have been made for this implementation. For RDD-DBSCAN, no real speed-up can be observed when scaling the number of cores (run-times are more or less independent from the number of used cores and constant when taking into account the measurement deviations to be expected). The same applies to the DBSCAN on Spark implementation.

As pointed out in Section 4.3, it would have been interesting to compare the running time of MapReduce-based implementations using the same data set and hardware. Han et al. [34] who tried as well without success to get the implementations of MR-DBSCAN and DBSCAN-MR, developed for comparison

Table 7. Run-time (in Seconds) on Big Twitter Data Set vs. Number of Cores

Number of cores	1	384	768	928
HPBDBSCAN hybrid	2 079	10	8	–
ELKI	15 362	–	–	–
Spark DBSCAN	–	–	–	Exception
RDD-DBSCAN	–	–	–	5 335
DBSCAN on Spark(*)	–	–	–	1 491

(*) Does only implement an approximation the DBSCAN algorithm.

reasons their own MapReduce-based implementation and observed a 9 to 16 times slower performance of their MapReduce-based DBSCAN implementation in comparison to their implementation for Spark.

Run-time Measurements on Big Data Set While the previous measurements were made using the smaller Twitter data set, also the bigger one containing 16 602 137 points was used in experiments in order to investigate some sort of "weak scaling". While HPDBSCAN can easily handle it, the Spark implementations have problems with this 289 MB CSV file.

When running any of the Spark DBSCAN implementations while making use of all available cores of our cluster, we experienced out-of-memory exceptions[6]. Even though each node in our cluster has 42 GB RAM, this memory is shared by 24 virtual cores of that node. Hence, the number of cores used on each node had to be restricted using the `--executor-cores` parameter, thus reducing the parallelism, but leaving each thread more RAM (which was adjusted using the `--executor-memory` parameter).

The results for the big Twitter data set are provided in Table 7. HPDBSCAN (in the hybrid version using OpenMP within each node and MPI between nodes) scales well. Spark DBSCAN failed throwing the exception `java.lang.Exception: Box for point Point at (51.382, -2.3846); id = 6618196; box = 706; cluster = -2; neighbors = 0 was not found`. DBSCAN on Spark did not crash, but returned completely wrong clusters (it anyway does not cluster according to the original DBSCAN idea). RDD-DBSCAN took almost one and a half hour. Due to the long run-times of the DBSCAN implementations for Spark already with the maximum number of cores, we did not perform measurements with a lower number of cores.

5.5 Discussion of Results

Big data approaches aim at avoiding "expensive" network transfer of initial input data by moving computation where the data is available on local storage. In contrast, in HPC systems, the initial input data is not available locally, but via

[6] Despite these exceptions, we did only encounter once during all measurements a re-submissions of a failed Spark tasks – in this case, we did re-run the job to obtain a measurement comparable to the other, non failing, executions.

an external SAN storage system. Due to the fact that big data approaches aim at minimising network transfers, the fact that the Infiniband interconnection of CPU nodes used in the HPC configuration is faster than the Ethernet-based big data configuration, should not matter that much. In addition, because DBSCAN is not that I/O bound, but rather CPU bound, the I/O speed and file formats do not matter as much as the used programming languages and clever implementations in particular with respect to the quality of the domain decomposition for an effective load balancing of parallel execution.

HPDBSCAN outperforms all other considered implementations. Even the optimised serial ELKI is slower than a serial run of HPDBSCAN. This can attributed to C++ code being faster than Java and to the fact that HPDBSCAN uses the fast binary HDF5 file format, whereas all other implementations have to read and parse a textual input file (and respectively create and write a textual output file). Having a closer look at the scalability reveals furthermore that HPDBSCAN scales ("strong scaling") for the small data set very well up to 16 cores, but some saturation becomes visible with 32 cores (Table 5): Amdahl's law [45] suggests that sequential parts and overheads start then to dominate.

For the given skewed data set, scalability of RDD-DBSCAN is only given for a low number of cores (the run-time difference between 16 and 32 cores is within to be expected measurement deviations), but not beyond[7]. An analysis of the run-time behaviour reveals that in the middle of execution, only one long running task of RDD-DBSCAN is being executed by Spark: while one core is busy with this task, all other cores are idle and the subsequent RDD transformations cannot yet be started as they rely on the long running task. This means, the load is due to bad domain decomposition not well balanced and explains why RDD-DBSCAN does not scale beyond 58 cores: adding more cores just means adding more idle cores (while one core executes the long running task, the remaining 57 cores are enough to handle the workload of all the other parallel tasks). In fact, the serial ELKI using just one core is faster than RDD-DBSCAN using up to 8 cores and even beyond, RDD-DBSCAN is not that much faster and which does not really justify using a high number of cores. DBSCAN on Spark delivers completely wrong clusters, hence it has to be considered useless and it is pointless that it is faster than RDD-DBSCAN.

The comparison between HPDBSCAN and the Spark-based implementations shows that HPDBSCAN does a much better load balancing: the Spark-based implementations typically try to balance the number of points per partition before enlarging the partitions by *eps* on each side to add the ghost/halo regions which adds further extra points (which can be a significant amount in dense areas). Also, partitions cannot get smaller than 2 *eps*. In contrast, HPDBSCAN balances the number of comparisons to be performed (i.e. calculating the Euclidean distance) and takes to this aim also comparisons of points inside the partition with points in the ghost/halo regions of that partition into account. Furthermore, partitions can get smaller than 2 *eps* which is also important to balance heav-

[7] Remarkably, the authors of RDD-DBSCAN [37] performed scalability studies only up to 10 cores.

ily skewed data. As a result, HPDBSCAN is able to balance the computational costs much better in particular for skewed data.

While the HPC implementations are much faster than the big data implementations, it is at least in favour of the big data implementations that HPC implementations require much more lines of code (=more development efforts) than the more abstract Scala implementations for Spark. Also, the big data platforms provide fault tolerance which is not given on HPC platforms.

6 Summary and Outlook

We surveyed existing parallel implementations of the spatial clustering algorithm DBSCAN for *High-Performance Computing* (HPC) platforms and big data platforms, namely Apache Hadoop and Apache Spark. For those implementations that were available as open-source, we evaluated and compared their performance in terms of run-time. The result is devastating: none of the evaluated implementations for Apache Spark is anywhere near to the HPC implementations. In particular on bigger (but still rather small) data sets, most of them fail completely and do not even deliver correct results.

As typical HPC hardware is much more expensive than commodity hardware used in most big data applications, one might be tempted to say that it is obvious that the HPC DBSCAN implementations are faster than all the evaluated Spark DBSCAN implementations. Therefore, in this case study, the same hardware was used for both platforms: HPC hardware. – But using commodity hardware instead would not change the result: while the HPC implementations of DBSCAN would then not benefit from the fast HPC I/O, a closer analysis reveals that typical big data considerations such as locality of data are not relevant in this case study, but rather proper parallelization such as decomposition into load balanced parallel tasks matters. The Spark implementations of DBSCAN suffer from a unsuitable decomposition of the input data. Hence, the used skewed input data leads to tasks with extremely imbalanced load on the different parallel cores.

It can be speculated that in HPC, parallelization needs to manually implemented and thus gets more attention in contrast to the high-level big data approaches where the developer gets not in touch with parallelization. Another reason to prefer HPC for compute-intensive tasks is that already based on the used programming languages, run-time performance of the JVM-based Spark platform can be expected to be one order of magnitude slower than C/C++. While RDDs support a bigger class of non-embarrassingly parallel problems than MapReduce, Spark still does not support as tight coupling as OpenMP and MPI used in HPC which might however be required for, e.g., simulations.

To conclude our story of Goliath and the elephant: if you do not even get the parallelization and load balancing right, it does matter whether you are Goliath or an elephant. Or – looking at it the other way around – if you want to take for your big data a fast DBSCAN algorithm off-the-shelf, you are currently better off if you take HPDBSCAN [29].

However, it has to be said that in general, the big data platforms such as Apache Spark offer resilience (such as re-starting crashed sub-tasks) and a higher level of abstraction (reducing time spent implementing an algorithm) in comparison to the low-level HPC approach.

As future work, it would be interesting to investigate the performance of the different implementations on other data sets – e.g. for non-skewed data, the load imbalance can be expected to disappear; but still, the C/C++ HPC implementations can be expected to be faster than the Java big data implementation. Furthermore, it is worthwhile to transfer the parallelization concepts of HPDBSCAN to a Spark implementation, in particular the domain decomposition below rectangles/hypercubes smaller than 2 *eps* and the load balancing cost function of considering the number of comparisons of a partition including comparisons to points in the ghost/halo regions of that partition (in contrast to the Spark-based implementations considering only the number of points in a partition without taking ghost/halo regions into account). This would give the end user faster DBSCAN clustering on big data platforms. (Having more or less the same parallelization ideas implemented on both platforms would also allow better assessment of the influence of C/C++ versus Java/Scala and of MPI versus the RDD approach of Spark.) Also, the scientific binary HDF5 data file format can currently not be processed by Hadoop or Spark in a way that data storage locality is exploited. Hence, the binary file had to be converted to a text-based input file format and one might argue that this I/O issue slowed down the Spark implementations. But in fact, the load imbalance of the CPU load contributes much more to the run-time than any I/O. As soon as the algorithms become better load balanced, less CPU bound and instead more I/O bound, data locality matters. A simple approach to be able to exploit the harder to predict locality of binary formats is to create some sort of "street map" in an initial and easily to parallelize run and use later-on the data locality information contained in this "street map" to send jobs to those nodes where the data is locally stored. We have successfully demonstrated this approach [3] for processing with Apache Hadoop the binary file formats used in the LHC experiments; the same approach should also be applicable to HDF5 files.

Acknowledgment

The author likes to thank all those who provided implementations of their DBSCAN algorithms. Special thanks go to the division Federated Systems and Data at the Jülich Supercomputing Centre (JSC), in particular to the research group High Productivity Data Processing and its head Morris Riedel. The author gratefully acknowledges the computing time granted on the supercomputer JUDGE at Jülich Supercomputing Centre (JSC).

The HPDBSCAN implementation of DBSCAN will be used as pilot application in the research project DEEP-EST (Dynamical Exascale Entry Platform – Extreme Scale Technologies) which receives funding from the European Union

Horizon 2020 – the Framework Programme for Research and Innovation (2014-2020) under Grant Agreement number 754304.

References

1. Schmelling, M., Britsch, M., Gagunashvili, N., Gudmundsson, H.K., Neukirchen, H., Whitehead, N.: RAVEN – Boosting Data Analysis for the LHC Experiments. In: Applied Parallel and Scientific Computing PARA 2010, Revised Selected Papers, Part II. Volume 7134 of LNCS., Springer (2012) doi:10.1007/978-3-642-28145-7_21.
2. Memon, S., Vallot, D., Zwinger, T., Neukirchen, H.: Coupling of a continuum ice sheet model and a discrete element calving model using a scientific workflow system. In: Geophysical Research Abstracts. Volume 19 European Geosciences Union (EGU) General Assembly 2017., Copernicus (2017) EGU2017-8499.
3. Glaser, F., Neukirchen, H., Rings, T., Grabowski, J.: Using MapReduce for High Energy Physics Data Analysis . In: 2013 International Symposium on MapReduce and Big Data Infrastructure, IEEE (2013/2014) doi:10.1109/CSE.2013.189.
4. Ester, M., Kriegel, H., Sander, J., Xu, X.: Density-based spatial clustering of applications with noise. In: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, AAAI Press (1996)
5. Apache Software Foundation: Apache Hadoop. Web page (2017) `http://hadoop.apache.org/`.
6. Neukirchen, H.: Performance of big data versus high-performance computing: Some observations. In: Clausthal-Göttingen International Workshop on Simulation Science, 27-28 April 2017, Göttingen, Germany. (2017) Extended Abstract.
7. Neukirchen, H.: Survey and Performance Evaluation of DBSCAN Spatial Clustering Implementations for Big Data and High-Performance Computing Paradigms. Technical Report VHI-01-2016, Engineering Research Institute, University of Iceland (2016)
8. Neukirchen, H.: Elephant against Goliath: Performance of Big Data versus High-Performance Computing DBSCAN Clustering Implementations. EUDAT B2SHARE record (2017) doi:10.23728/b2share.b5e0bb9557034fe087651de9c263000c.
9. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on Management of data. Volume 14 Issue 2., ACM (1984) doi:10.1145/602259.602266.
10. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD international conference on Management of data. Volume 19 Issue 2., ACM (1990) doi:10.1145/93597.98741.
11. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM **18**(9) (1975) 509–517 doi:10.1145/361002.361007.
12. Kjolstad, F., Snir, M.: Ghost cell pattern. In: 2nd Annual Conference on Parallel Programming Patterns (ParaPLoP). March 30-31st, 2010, Carefree, AZ., ACM (2010) doi:10.1145/1953611.1953615.
13. Xu, X., Jäger, J., Kriegel, H.P.: A fast parallel clustering algorithm for large spatial databases. Data Mining and Knowledge Discovery **3**(3) (1999) 263–290 doi:10.1023/A:1009884809343.

14. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering **5**(1) (1998) 46–55 doi:10.1109/99.660313.
15. MPI Forum: MPI: A Message-Passing Interface Standard. Version 3.0 (September 2012) `http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.
16. OpenSFS and EOFS: Lustre. Web page (2017) `http://lustre.org/`.
17. IBM: General Parallel File System Knowledge Center. Web page (2017) `http://www.ibm.com/support/knowledgecenter/en/SSFKCN/`.
18. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An overview of the HDF5 technology suite and its applications. In: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases, ACM (2011) 36–47 doi:10.1145/1966895.1966900.
19. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems Design & Implementation, Berkeley, CA, USA, USENIX Association (2004) doi:10.1145/1327452.1327492.
20. Apache Software Foundation: Apache Spark. Web page (2017) `http://spark.apache.org/`.
21. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association (2012)
22. Jha, S., Qiu, J., Luckow, A., Mantha, P., Fox, G.C.: A tale of two data-intensive paradigms: Applications, abstractions, and architectures. In: 2014 IEEE International Congress on Big Data, IEEE (2014) 645–652 doi:10.1109/BigData.Congress.2014.137.
23. MacQueen, J., et al.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, Volume 1: Statistics, University of California Press, Berkeley, California (1967) 281–297
24. Ganis, G., Iwaszkiewicz, J., Rademakers, F.: Data Analysis with PROOF. In: Proceedings of XII International Workshop on Advanced Computing and Analysis Techniques in Physics Research. Number PoS(ACAT08)007 in Proceedings of Science (PoS) (2008)
25. Wang, Y., Goldstone, R., Yu, W., Wang, T.: Characterization and Optimization of Memory-Resident MapReduce on HPC Systems. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE (2014) 799–808 doi:10.1109/IPDPS.2014.87.
26. Kriegel, H.P., Schubert, E., Zimek, A.: The (black) art of runtime evaluation: Are we comparing algorithms or implementations? Knowledge and Information Systems **52**(2) (2016) 341–378 doi:10.1007/s10115-016-1004-2.
27. Schubert, E., Koos, A., Emrich, T., Züfle, A., Schmid, K.A., Zimek, A.: A framework for clustering uncertain data. PVLDB **8**(12) (2015) 1976–1979 doi:10.14778/2824032.2824115.
28. Patwary, M.M.A., Palsetia, D., Agrawal, A., Liao, W.k., Manne, F., Choudhary, A.: A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), IEEE (2012) 1–11 doi:10.1109/SC.2012.9.

29. Götz, M., Bodenstein, C., Riedel, M.: HPDBSCAN: highly parallel DBSCAN. In: Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, held in conjunction with SC15: The International Conference for High Performance Computing, Networking, Storage and Analysis, ACM (2015) doi:10.1145/2834892.2834894.
30. Patwary, M.M.A.: PDSDBSCAN source code. Web page (2017) `http://users.eecs.northwestern.edu/~mpatwary/Software.html`.
31. Götz, M.: HPDBSCAN source code. Bitbucket repository (2016) `https://bitbucket.org/markus.goetz/hpdbscan`.
32. Baldridge, J.: ScalaNLP/Nak source code. GitHub repository (2015) `https://github.com/scalanlp/nak`.
33. Busa, N.: Clustering geolocated data using Spark and DBSCAN. O'Reilly web page (2016) `https://www.oreilly.com/ideas/clustering-geolocated-data-using-spark-and-dbscan`.
34. Han, D., Agrawal, A., Liao, W.K., Choudhary, A.: A novel scalable DBSCAN algorithm with Spark. In: Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International, IEEE (2016) 1393–1402 doi:10.1109/IPDPSW.2016.57.
35. Litouka, A.: Spark DBSCAN source code. GitHub repository (2017) `https://github.com/alitouka/spark_dbscan`.
36. Stack Overflow: Apache Spark distance between two points using squaredDistance. Stack Overflow discussion (2015) `http://stackoverflow.com/a/31202037`.
37. Cordova, I., Moh, T.S.: DBSCAN on Resilient Distributed Datasets. In: 2015 International Conference on High Performance Computing & Simulation (HPCS), IEEE (2015) 531–540 doi:10.1109/HPCSim.2015.7237086.
38. Cordova, I.: RDD DBSCAN source code. GitHub repository (2017) `https://github.com/irvingc/dbscan-on-spark`.
39. He, Y., Tan, H., Luo, W., Feng, S., Fan, J.: MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. Frontiers of Computer Science **8**(1) (2014) 83–99 doi:10.1007/s11704-013-3158-3.
40. aizook: Spark_DBSCAN source code. GitHub repository (2014) `https://github.com/aizook/SparkAI`.
41. Raad, M.: DBSCAN On Spark source code. GitHub repository (2016) `https://github.com/mraad/dbscan-spark`.
42. He, Y., Tan, H., Luo, W., Mao, H., Ma, D., Feng, S., Fan, J.: MR-DBSCAN: an efficient parallel density-based clustering algorithm using MapReduce. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), IEEE (2011) 473–480 doi:10.1109/ICPADS.2011.83.
43. Dai, B.R., Lin, I.C.: Efficient map/reduce-based DBSCAN algorithm with optimized data partition. In: 2012 IEEE Fifth International Conference on Cloud Computing, IEEE (2012) 59–66 doi:10.1109/CLOUD.2012.42.
44. Bodenstein, C.: HPDBSCAN Benchmark test files. EUDAT B2SHARE record (2015) doi:10.23728/b2share.7f0c22ba9a5a44ca83cdf4fb304ce44e.
45. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS Conference Proceedings Volume 30: 1967 Spring Joint Computer Conference, American Federation of Information Processing Societies (1967) 483–485 doi:10.1145/1465482.1465560.