RT-DBSCAN: Real-time Parallel Clustering of Spatio-Temporal Data using Spark-Streaming

Yikai Gong[1], Richard O. Sinnott[1], and Paul Rimba[2]

[1] University of Melbourne, Melbourne VIC, Australia,
[2] Data61, CSIRO, Sydney, NSW, Australia,
Contact: yikaig@student.unimelb.edu.au

Abstract Clustering algorithms are essential for many big data applications involving point-based data, e.g. user generated social media data from platforms such as Twitter. One of the most common approaches for clustering is DBSCAN. However, DBSCAN has numerous limitations. The algorithm itself is based on traversing the whole dataset and identifying the neighbours around each point. This approach is not suitable when data is created and streamed in real-time however. Instead a more dynamic approach is required. This paper presents a new approach, RT-DBSCAN, that supports real-time clustering of data based on continuous cluster checkpointing. This approach overcomes many of the issues of existing clustering algorithms such as DBSCAN. The platform is realised using Apache Spark running over large-scale Cloud resources and container based technologies to support scaling. We benchmark the work using streamed social media content (Twitter) and show the advantages in performance and flexibility of RT-DBSCAN over other clustering approaches.

Keywords: DBSCAN; Clustering; Real-time systems

1 Introduction

Clustering is one of the major data mining methods used for knowledge discovery [11] on big data. Density-based clustering algorithms like DBSCAN [8] are in widespread use and numerous extensions are now available for discovering patterns and clusters in large data sets [2,12,15]. However, neither DBSCAN nor its extensions support real-time processing or allow to tackle streamed (high velocity) data [16]. Rather, DBSCAN operates in a batch mode where all the data is acquired and then processed. This feature makes it unsuitable for supporting the ever growing data from real-time data streams.

There is a strong need for real-time cluster discovery in many diverse application domains such as urban traffic monitoring, emergency response, network accessing analysis. The demands for real-time clustering of big data raise several needs and requirements for improvements and refinements of the DBSCAN algorithm, including the ability to: 1) generate a series of up-to-date intermediate result checkpoints when processing real-time (incoming) data; 2) support scalable parallel execution capabilities to reduce the response time for generating checkpoints; and 3) offer consistent performance in tackling ever growing amounts of data.

Existing extensions of DBSCAN offer no solution to the combination of these requirements. In this paper, we present a real-time parallel version of

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

DBSCAN (RT-DBSCAN) to address the above requirements. Compared to the original version of DBSCAN, optional parameters are added to the algorithm for controlling the efficiency and granularity of parallel-workload division. For clustering spatio-temporal data in time and space, a spatio-temporal distance is applied, noting that the definition of distance in this algorithm can be adapted to other kinds of higher dimensional data. We have implemented RT-DBSCAN using Apache Spark Streaming. We benchmark the system using large-scale streamed social media data (tweets) on the National Research Cloud (NeCTAR) in Australia.

2 Related Clustering Algorithms

Extensions of DBSCAN can be classified into two types: performance optimized DBSCAN and application optimized DBSCAN. The former aims at reducing the execution time for data clustering [12,18], whilst the latter focuses on adapting DBSCAN to different high-dimensional data structures required for specific application scenarios [6,17,19].

There are many good ideas in performance-oriented extensions of DBSCAN. *l*-DBSCAN [18] proposes a method to reduce the size of datasets before running DBSCAN. It employs a graph-based hybrid clustering algorithm [3] to pre-generate a few candidate (approximate) clusters. Only the points in those clusters are then input into DBSCAN for final clustering. However, this two-phased clustering method has several major limitations. Firstly, two critical parameters are added for hybrid clustering. As the authors point out, unsuitable selection of these parameters can lead to inconsistent clustering results. Secondly, the pre-clustering phase is used for filtering out noise data, e.g. data outliers. If a highly skewed dataset is input, this phase can become useless and consume unnecessary computing resources. This extension also does not meet any of the real-time clustering requirements, but the idea of reducing or sampling all of the data in DBSCAN is meaningful and been incorporated into RT-DBSCAN.

MR-DBSCAN [12] proposes a parallel version of DBSCAN in a MapReduce manner [5]. The major contribution of this extension is that it provides a method to divide a large dataset into several partitions based on the data dimensions. Localized DBSCANs can be applied to each partition in parallel during a map phase. The results of each partition are then merged during a final reduce phase. For the overall cost, a partition-division phase is added into DBSCAN. A division method called *Cost Balanced Partition* is used to generate partitions with equal workloads. This parallel extension meets the requirements of scalable execution for handling large scale data sets and the MapReduce approach makes it suitable for many popular big data analytic platforms like Hadoop MapReduce and Apache Spark [10]. However, this extension does not meet all the requirements of real-time clustering. It needs to traverse the whole dataset for parallel clustering which means that its execution time is still dependent on the size of the dataset. Thus, whilst MR-DBSCAN has good performance for batch-oriented data scenarios, it is not suitable for high velocity datasets.

For those application-oriented extensions to DBSCAN, we consider two of them which are most closely related to our approach.

Stream data is often spatio-temporal in nature and comprised of time-stamped, geographic location information [6]. This can be, for instance, social media data,

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

trajectory data, Internet of Things data. This raises a requirement for clustering those data in time and space according to their spatio-temporal characteristics. [2] presents a method for handling this requirement. It provides an example for clustering spatio-temporal data according to its non-spatial, spatial and associated temporal values. In addition, they propose a notion of density factor for each cluster which is helpful to identify the density of clusters.

Incremental DBSCAN [7] is another extension of DBSCAN suitable for mining in data warehouses. It supports incremental updates when clustering by inserting new data and deleting old data. It provides controls over the size of the involved data. Old data are excluded from clustering processes based on a time-based threshold which can be specified by the user. This method meets the real-time requirements of tackling ever increasing volumes of data. However, it only works for time-based clusters. The definition of old data is a critical factor in this algorithm. Essential information can be lost by dropping data if inappropriate thresholds are set. Although this method is designed for daily batch-oriented tasks, the idea of dropping old irrelevant data and inserting new data into existing clusters is essential when designing real-time, high velocty clustering solutions.

Apart from DBSCAN, there are many other density-based clustering algorithms such as OPTICS [1], DENCLUE [13] and CURD [14]. D-Stream [4] is a density based clustering approach for handling real-time streams, but it cannot handle data arriving in arbitrary time-stamped orders. In this paper, we present a new DBSCAN-based clustering approach that overcomes many of the issues and limitations related to both DBSCAN and the above mentioned systems when dealing with high velocity, streamed data.

3 Real-time Parallel DBSCAN Clustering

Clustering algorithms like DBSCAN normally need to input the whole dataset into a clustering process (all-in with single-out). The complexity of DBSCAN is $O(n^2)$. A typical DBSCAN traverses the whole dataset, and identifies the neighbors of each point. Each data element can be used/processed multiple times (e.g. as candidates to different clusters). Although incremental-DBSCAN supports updating of clusters by inserting new input data into existing clusters, this algorithm does not cope with ever growing sizes of historical data due to the data traversal demands of DBSCAN. To deal with this, incremental-DBSCAN drops outdated data to keep a fit size of dataset.

A key challenge of real-time DBSCAN is in controlling the size of traversal data needed to cluster ever growing data volumes. In the DBSCAN algorithm, for each new input data, a group of potential near-by points needs to be identified for cluster detection. For each new input point, if there is an efficient way to identify a full set of near-by context points in the historical dataset, only this subset of data is needed for clustering against any new input. Therefore, we can input the data point-by-point into the cluster process and get a series of up-to-date cluster checkpoints. If the performance of this pre-filtering method is not sensitive to the size of dataset, then we can cluster real-time stream data on-the-fly without being challenged by the ever growing volume of data streamed over potentially extended time periods.

This idea forms the basis for the definition of our real-time clustering (`RT-DBSCAN`) method. Specifically, *for a new input point p and a group of historical clustered points $cPoints$, $nearbyCtx(p, cPoints)$ is used to obtain a subset*

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

of cPoints which contains essential information of nearby inputs. Checkpoints produced by `RT-DBSCAN` *on this subset must be identical to the result of applying normal* `DBSCAN` *on the whole dataset for each input.*

3.1 Identify a Full Set of Nearby Context Points for Each Input

DBSCAN involves two key parameters: ε and *minPts*. The distance parameter ε defines how close two points need to be to be considered in the same neighbourhood. The border parameter *minPts* defines how many neighbourhoods related to a single point there should be for this to be considered as a cluster. In the following examples, we consider a scenario where $\varepsilon = 1$ unit and $minPts = 3$ points. In Figure 1, we highlight three scenarios related to putting a new point P_i into a set of historical clustered points. In scenario A, we consider firstly retrieving all the historical data within 1-ε distance to P_i, to get 3 non-clustered (noise) points. Since their distances to P_i are smaller than ε, P_i now have 3 neighbours and thus these four points form a new cluster. This seems sensible, but it can be wrong. If there is a point P_b that is less than 1-ε away from point P_a but more than 1- ε away from P_i, as shown in scenario A of the Figure 1, P_b will be ignored by this procedure. Although a new cluster is identified, P_b is not marked as a member of this cluster which it should be. This result disobeys the assertion made in the previous definition. If we consider extending the range of near-by-context from 1-ε to 2-ε in scenario B, 4 points are discovered including P_b. These 5 points are grouped into the same cluster. A similar question naturally arises. What if there is a point P_c which is less than 1-ε away from P_b and is more than 2-ε away from P_i as shown in scenario C? This sounds like an endless issue but it is not. Since P_a, P_b and P_c are historical processed points and *minPts* = 3, a cluster must have already been identified when the last of these three points was input in a previous iteration. In scenario C, although P_c is ignored as a nearby context point, its cluster information is carried by P_a and P_b and subsequently passed to P_i. With this information, P_i and two other noise points can be absorbed into the existing cluster. This result meets the requirements of the previous definition. In this case where $minPts = 3$, we find that historical points within distance 2-ε away from the input data contain enough information to establish connections between the input point and the existing historical noise points and existing clusters. The minimum distance ($minDis$) of nearby-contexts, which is 2ε in this case, is related to the parameter ε and *minPts*. $minDis$ is given as: $minDis = (minPts - 1) \times \varepsilon$. In the following section, we mark this procedure as $nearbyCtx(p, cPoints)$ where p is the input and $cPoints$ is the historical dataset. If $cPoints$ are indexed in a database, the response time of this procedure needs to be fast and non-sensitive to the growing size of $cPoints$. An incremental-DBSCAN is then applied on $nearbyCtx$

3.2 Convert Point-by-Point Clustering to Tick-by-Tick Clustering

Executing a single process for RT-DBSCAN by tackling the incoming data stream point-by-point is not an efficient approach and would not meet the requirements of real-time data intensive applications. The next challenge is to process the incoming data in parallel. However, considering the nature of DBSCAN, it is hard to process data streams in a fully parallel manner. The reason is that DBSCAN and RT-DBSCAN are based on a sequential processing model, e.g. using data

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

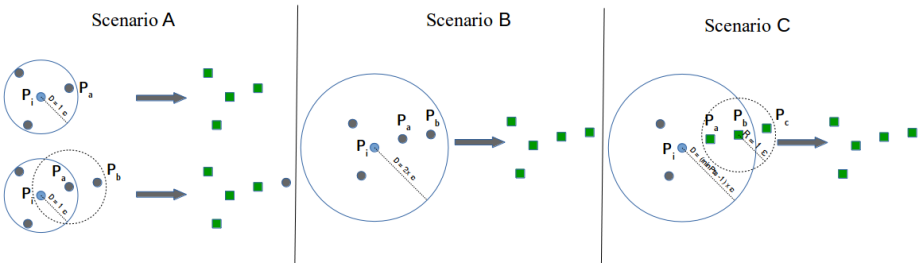

Figure 1. Illustration of getting nearby points context

in batches or micro-batches. For each input data, a group of nearby data will be queried and involved in the calculation. It is very likely that one input point can be used within the context of another input point. Therefore, these input data cannot be processed without impacting one another when used in a fully parallel manner. In addition, writing the historical datasets simultaneously can lead to consistency issue. From this we can conclude that: 1) each input point should know the other input data that is being processed in parallel; 2) conflicts between the results of each parallel processor need to be solved before clusters are used/persisted, and previous results need to be persisted into historical datasets before tackling new (incoming) data.

To tackle these challenges for parallel processing, we convert the point-by-point RT-DBSCAN to a (temporal) tick-by-tick RT-DBSCAN where data points from incoming data-streams are divided into separate ticks based on their arriving times; within each tick, data are processed and clustered in parallel; the results of each parallel processing step (within one tick) need to be merged before being persisted to solve any/all conflicts in data consistency, and at the end of each tick, the result must be persisted into the historical dataset before a new tick is started. The updated dataset at the end of each tick is a checkpoint for the clusters. A series of checkpoints forms the growth history of clusters.

Parallel processing is applied on data of each tick. To get nearby context points for a group of inputs in a given tick, the first task of each tick is to get the nearby points context for input point-set (iPs) $\{P_1, P_2, \ldots, P_j\}$. Depending on how the historical data (cPoints) is persisted, there are different ways to achieve this. The first option is to execute *nearbyCtx* multiple times for each point and merge the returned sets. The second option is to get the nearby context at one spot by generating a bounding box for iPs. If cPoints are stored in a database, the second option is normally preferred since it reduces the number of queries to the database. Firstly, a minimum bounding box is calculated to cover every point in the iPs. Then a new bounding box is generated by extending the previous rectangle with minDis for each border. The new bounding box is used for establishing the nearby points context from cPoints. Compared to option 1, the drawback of this method is that it can add unnecessary historical data into the nearby context. These noise points are filtered out by a partitioning method before applying Incremental-DBSCAN. The nearby context aggregates historical points together with the iPs passed into parallel processing.

The next task is to support location-based data partitioning and parallelisation of RT-DBSCAN. Inspired by MR-DBSCAN [12], each tick of our parallel RT-DBSCAN is designed in a MapReduce manner. In step 1, the data space is geographically divided into many cells. Each cell contains localised input points.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

In step 2, multiple local DBSCANs are executed on each cell in parallel. In step 3, the results from each cell are merged to recover the border information broken by the space division. For example, single clusters appearing in multiple cells need to be identified for merging. This is achieved in several steps. Firstly, a fast data division for parallel RT-DBSCAN is required. MR-DBSCAN uses cost-balanced (CB) partitioning to divide the data points in space into cells. It first divides the data space into equal sized small unit cells. Then, a balanced tree is calculated for merging these unit cells into many CB cells where each CB cell contains nearly the same number of points. This method is good at generating balanced workloads for parallel DBSCAN, but generating a balanced tree is computationally expensive. MR-DBSCAN is designed for processing a large amount of data in a single batch task and its CB partition is only applied once at the beginning of the clustering procedure. However in our RT-DBSCAN realisation, partitions are calculated at the beginning of each tick. This approach makes it is impossible to reuse the partitions in previous ticks since point-sets within different ticks have different nearby point contexts. To address this, a new partition method , Fast Clustering (FC) partitioning, is designed which is more suited to RT-DBSCAN. As illustrated in Figure 2, the idea of this partitioning method is to iteratively divide a 2D space into four sub-cells until a threshold is reached (i.e. a threshold on the number of points in a cell). Then, we drop the cells where the number of contained points is less than *minPts*. Finally we extend each quad-cell by 1-ε distance on each border. One benefit of this extension is to find overlapping areas between cells so that a merge phase can be applied at the end of each tick. Another purpose is to get the nearby points context for each cell since some essential contexts can be carried in dropped cells that need to be re-used.

As shown in Figure 2, there are two kinds of thresholds used for dividing the space. The space is iteratively divided into 4 cells until either the number of points within the current cell is less than 6 or the minimum border of current cell is less than 2ε.

The first condition, called $maxPts$, is to prevent a single cell from having too many points to process. The value 6 for $maxPts$ in Figure 2 is only used for demonstration purpose. In real-case, $maxPts$ must be greater or equal to $minPts \times 2^n$, where n is the number of dimensions in FC partitioning. This is to avoid over-partitioning a potential cluster. The second condition, called $minSize$, is to prevent the partitioned cells from having very small sizes. If the cell size is less than 1-ε, points that are potentially in the same cluster will likely be divided into separate cells. Although they will be merged and sorted out in the final merging task, this can lead to a large amount of work for merging and thus be very inefficient. These thresholds are free to be customized depending on the specific cases. After the space is divided into a quad-tree, certain cells are dropped if either the number of points within the cell is less than *minPts* or no new input points fall inside this cell. The propose of this dropping is to reduce the number of parallel tasks/partitions. The first condition helps to drop blank cells

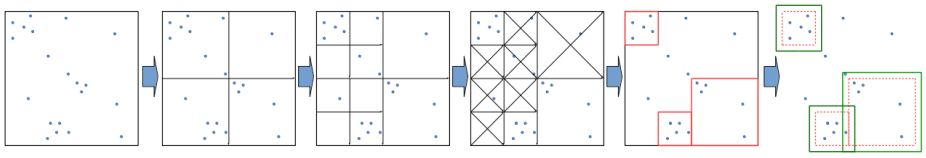

Figure 2. Illustration of the Fast Clustering Partition Method for RT-DBSCAN

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

and/or cells with very limited numbers of points. It is predetermined that points inside this cell cannot form a cluster. Hence these cells can be dropped. Some of the dropped cells may contain points that belong to clusters in other cells. Those points will be re-selected when other valid cells are extended by 1-ε distance. The second condition is to overcome the flaw of $nearbyCtx(iPs, cPoints)$ mentioned in the previous section, where cells with only historical data can be dropped. Similarly, those dropped essential-nearby-context to other cells with new data will be re-selected when those cells are extended. If the input spatio-temporal data arrives in arbitrary order, the single bounding box for nearby context could be huge in size. Although dropping 'blank' cells can filter out non-necessary historical data before starting DBSCAN, it can still generate many workload to I/O and FC partitioning. Using multiple discrete bounding boxes can be a solution for that case. In this paper, we only use single bounding box which is more suitable for data that arrives in (time) sequence. After dropping blank cells, all remaining cells (the red rectangle in Figure 2) now meet both of the following conditions: they have more than one new input data and they have more than one core point of a cluster (regardless of whether they are historical or new points).

Finally, those cells are extended by 1-ε distance (the green rectangle in Figure 2). As mentioned above, the purpose of this extension is to identify overlapping areas when merging cells and pick up lost contexts during the cell dropping process. Figure 3 illustrates some of these scenarios. If data are in a high-dimensional space, this method can be adjusted by dividing the space based on multiple dimensions.

Iteratively divding a space into 4 cells is a naive version of FC partitioning in 2D space. This version suffers from dividing flat rectangle shaped cells, *i.e.,* partitioning can stop in the first iteration due to the smallest border of a flat rectangle reaches the threshold. This problem is solved by dividing each cell into $2^{(n-m)}$ sub-cells. n is the number of total dimensions and m is the number of dimensions which their corresponding borders reach the size threshold. After this improvement, the partitioning is driven by each dimension and its corresponding border of the target cell.

After this FC partition method, those points in $iPs \cup nearbyCtx(iPs, cPoints)$ are divided into two groups: $aPts$ where each point belongs to one or multiple cells and group $dPts$ where each point does not belong to any cell. Only $aPts$ will be applied parallel and DBSCAN for each cell. This procedure is marked as $PCluster(aPts)$. The result of $PCluster(aPts)$ may contain duplicated points (i.e. points belonging to duplicate cells). This result will be union-ed with $dPts$ before being merged/cleaned. All points in $aPts$ and $dPts$ will be persisted at the end of this tick.

Points inside each partition contain both new input and historical clustered data. An incremental-DBSCAN approach is applied to those points for each

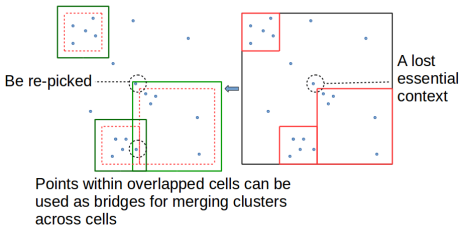

Figure 3. Illustration of Cell Extensions

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

partition. It can generate duplicated data belonging to different clusters. Since the final merging procedure handles this problem, duplicated points are not fixed in the local partition. The implementation supports a customizable function for calculating the distance between two data points in multi-dimensional spaces. A spatio-temporal distance function is created for clustering social-media data. This spatio-temporal distance is given in Equation 1, where P_i and P_j are vectors representing two spatio-temporal data (*e.g.,* Tweets). x and y in vector are values of GPS information (*e.g.,* longitude/latitude) and t is the time-stamp value. This equation is based on Euclidean distance. A customized spatio-temporal ratio s is used to convert the temporal value t into a spatio-value, so that all spatio-temporal values (*i.e.,* x, y, t) can have the same unit in the distance calculation.

$$P_i = (x_i, y_i, t_i), P_j = (x_j, y_j, t_j), D_g = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, \triangle t = |t_i - t_j| \\ Distance = \sqrt{D_g^2 + (\triangle t \times s)^2} \tag{1}$$

After the parallel local DBSCAN finishes, the result sets (U_r) are union-ed with $dPts$, i.e. $U_r = PCluster(aPts) \cup dPts$.

Duplicated points can appear in U_r. For example, in Figure 3 point P is on the overlapped area thus it exists in two partitions. After applying local DBSCAN on each partition, one instance of P is a NOISE point and another instance belongs to a cluster. In this final procedure, these kinds of inconsistencies are handled by a merge function: $U_q = merge(U_r)$. Points inside U_q are ensured to be unique. The merging solutions are described in the following paragraphs.

If a point belongs to multiple clusters, the use of this point is key. When a point P has duplicates: if all instances of P are NOISE then we keep one of them and drop the others. If all non-noise instances of P belong to the same cluster A, then we create a singleton of P, mark it as a member of cluster A and merge its roles in cluster A using the priority order: CORE > BORDER > NOISE. If non-noise instances belong to multiple clusters then we create a singleton of P and merge its roles among multiple clusters using the priority order: CORE > BORDER. If the merged role is 'BORDER', where a border point can belong to multiple clusters we add $clusterIds$ into a list and attach it to the new singleton. If the merged role is 'CORE' then the point is a solid joining point for multiple clusters. All such clusters must be merged into one cluster. To do this we create a singleton of P and mark it as 'CORE'. We then randomly pick a cluster A from the non-noise instances and attach it to the singleton and then identify members in other clusters which are to be merged. Finally we change their $clusterId$ to cluster A. However, what if some to-be-merged cluster points are not in U_r? Data in U_r come from $iPs \cup nearbyCtx(iPs, cPoints)$. There is no guaranteed that all the cluster member are covered by this set. Figure 4 illustrates this issue.

In Figure 4, two triangle-shaped points are not merged because they are absent in U_r. Since the clusterIds of those potential absent points are known, according to the cluster conversion table, all points in the to-be-merged clusters can be retrieved from cPoints. This procedure is given as: $U_c = getClusterPoints(ConvTbl)$. A union operation is then applied on U_r and U_c. Cluster merging is finally applied on this combined set.

Almost all the merging code can be executed in parallel except the procedure for building a global cluster conversion table. Each computation node needs to report their discoveries to a centralised node for generating the global conversion

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

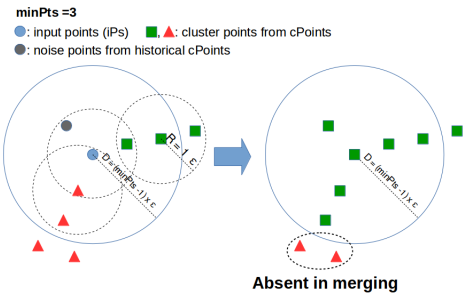

Figure 4. Illustration of the pitfall in merging clusters

table. After the above procedures (*i.e.,* partitioning; clustering; merging), the output set U_q contains unique points with their up-to-date cluster information. Finally U_q should be persisted before a next tick starts.

4 Implementation of RT-DBSCAN

The RT-DBSCAN algorithm proposed above has been implemented using Apache Spark Streaming leveraging a platform called SMASH [9]. SMASH is a platform for hosting and processing historical data and Spark applications. It provides a Cloud based software infrastructure utilizing HDFS, Accumulo, GeoMesa, Spark, Kafka and GeoServer integrated together as a platform service for analysing spatial-temporal data. This platform was originally used for traffic data analysis. In deploying the RT-DBSCAN on SMASH, Kafka is used as the streaming source for feeding new data. GeoMesa over Accumulo provides a distributed geo-spatial database. It is used for storing and querying historical data. Spark Streaming is a framework for building streaming applications on Apache Spark. This framework treats data streams as ticks of chunks and executes micro-batch tasks for each tick of data. Spark itself has many other interfaces for running MapReduce functions. These functions suit the needs of RT-DBSCAN and save a lot of work in implementing the clustering algorithm. Twitter data (tweets) are used for the case studies and benchmarking of the platform.

Figure 5 illustrates the procedure of realizing RT-DBSCAN using Spark nodes and Spark Streams as the framework for tackling data stream as a series of data chunks in ticks. At the beginning of each tick, a FC partition is applied against incoming data chunks on a single master node. Data shuffling (*i.e.,* sending data to the node which holds the cell it belongs to) and local DBSCAN are then executed in parallel on each worker node. A cluster merging table is generated on the master node after all local DBSCANs stop. Finally result merging and data persistence are handled in parallel on worker nodes. A new tick procedure starts after the results of the previous tick have been persisted.

5 Benchmarking of RT-DBSCAN on Spark Stream

Our case studies and benchmarking works were realised on the federally funded Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR) Research Cloud (https://nectar.org.au/research-cloud/). The NeCTAR Research Cloud is based on OpenStack. NeCTAR provides almost 30,000 servers across multiple availability zones across Australia including Melbourne, Monash, Brisbane, Canberra, Adelaide and Tasmania. Fifteen computation nodes (Virtual

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

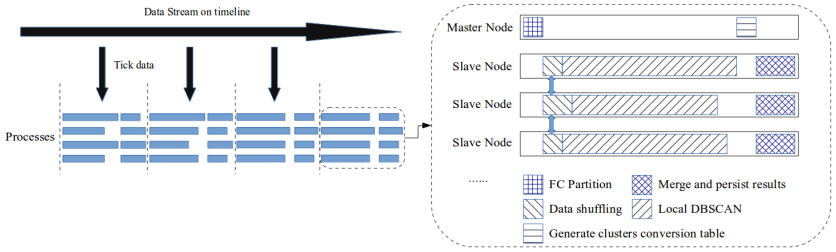

Figure 5. Illustration of RT-DBSCAN Procedure Using Spark Streaming.

Machines) from the Melbourne zone were used to form the core infrastructure for the case studies on RT-DBSCAN. The specification of each node was as follows: one master node: VCPUs @2.60GHz $\times$ 4 ; 12GB RAM; 120GB HDD; 13 slave nodes: VCPUs @2.60GHz $\times$ 2; 8GB RAM; 70GB HDD and one interface node: VCPUs @2.60GHz $\times$ 4; 16GB RAM; 130GB HDD. The results of I/O benchmarking on the computational resources was: bi-directional network bandwidth: $7.88 \pm 1.03 Gbits/sec$; cached reads rate: $3036.1 \pm 67.1 MB/sec$; buffered disk reads rate: $134.6 \pm 18.1 MB/sec$, and disk write rate: $599.7 \pm 39.1 MB/sec$.

Figure 6 illustrates the architecture of the software-stack (SMASH) used on the Cloud nodes for implementing RT-DBSCAN. The software components were packaged into Docker images for scaling of the platform. Instead of using scaling tools like Kubernetes (https://kubernetes.io) or Docker Swarm (https://docs.docker.com/engine/swarm/), a bespoke tool (https://github.com/project-rhd/grunt-clouddity) was developed and used for auto deployment and scaling the SMASH platform, *e.g.,* creating/terminating VMs, managing security groups/rules and scaling Docker containers. This is a command line interface tool that relies on http clients interfacing to OpenStack and Docker.

In Figure 6, the software containers are divided into three layers: an *Application layer* comprising GeoServer (v2.9.4) and Apache Kafka (v0.11.0.0). These two applications/containers are deployed on interface nodes for data visualization and stream pipelines. A *Computation layer* including Apache Spark (v2.1.1) with a master node and thirteen slave nodes, and a *Data Storage* layer deployed across the master node and slave nodes. The Hadoop Distributed File System (HDFS) (v2.7.4) is installed as the file system of the SMASH platform (block replication 2 and *sync.behind.writes* and *synconclose* are enabled to ensure data is written immediately into disk; Hadoop data directories are mounted to the local file system of the nodes; Default values are used for other configurations). Apache Accumulo (v1.8.1) is deployed over HDFS as a key/value data store which is similar to Google's BigTable storage. GeoMesa (v1.3.2) is a distributed, spatio-temporal database which is deployed on top of the Accumulo cloud data

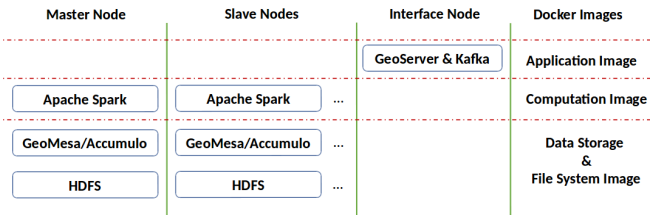

Figure 6. Illustration of Dockerized SMASH Platform

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

storage. In RT-DBSCAN, historical/checkpoint data created is persisted and indexed in GeoMesa/Accumulo.

Tweets are used as the data source for our benchmarking cases studies. We collected a fixed size of tweets data which were post in Melbourne within 6 months in 2015. This dataset contains 604,529 tweets ($\approx 220MB$) with GPS and time-stamps. An application was built for reading this dataset and generating the actual data stream. This generator controls the output rate of the stream and pushes it into the Kafka service on the SMASH platform. Our stream application running on Spark gets continuous data streams from Kafka, applies the RT-DBSCAN algorithm on this stream and persists/updates the results on GeoMesa/Accumulo clusters.

Figure 7 illustrates a series of checkpoints generated by RT-DBSCAN (on-the-fly). The results are visualized by GeoServer on SMASH. Each red point on the maps of Figure 7 represents a noise point which does not belong to any clusters. Each larger green point represents a in-cluster point. RT-DBSCAN is able to start either from a blank or using existing clusters result as the initial starting point (checkpoint). Incremental update is then conducted at each tick (corresponding to micro-batches in Spark Stream) which handles new inputs and generates up-to-date checkpoints based on previous checkpoint.

RT-DBSCAN on Spark Streaming is naturally a "micro-batching" based architecture. A streaming engine is built on the underlying batch engine, where the streaming engine continuously creates jobs for the batch engine from a continuous data stream. There are two important concepts in this architecture. One is the "Batch Interval Time" (BIT) which is a fixed interval value decided by the user of Spark Streaming. This value controls the interval used for generating micro-batch tasks for the underlying batch engine, *i.e.,* the tick interval. Another concept is the "Batch Processing Time" (BPT) which is the exact execution time for each batch task. Ideally, Spark Stream needs to ensure a batch task is completed before the next batch is queued. If the processing times of arriving batches continuously takes longer than the batch interval time, a "snowball effect" can take place. This effect can eventually exhaust Spark resources. Depending on the setting, Spark may discharge this pressure by killing the application or by pushing this pressure back to the broker of data source, *e.g.,* Apache Kafka. Therefore it is important to ensure that for most of the batches that the BPT < BIT. The scheduling delay τ_i for each batch i in sequence is defined as: $\delta_i = BPT_i - BIT$ where

$$\tau_i = \begin{cases} 0, & \text{if } i = 0 \\ \tau_{i-1} + \delta_i, & \text{else if } \tau_{i-1} + \delta_i > 0 \\ 0, & \text{else} \end{cases} \tag{2}$$

The total delay T_i for each batch i in sequence is defined as: $T_i = \tau_i + BPT_i$. The average processing delay ν_i for each arriving input data involved in batch i

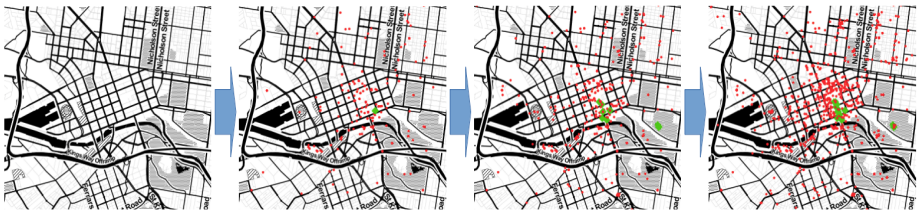

Figure 7. Cluster Checkpoints Generated by RT-DBSCAN

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

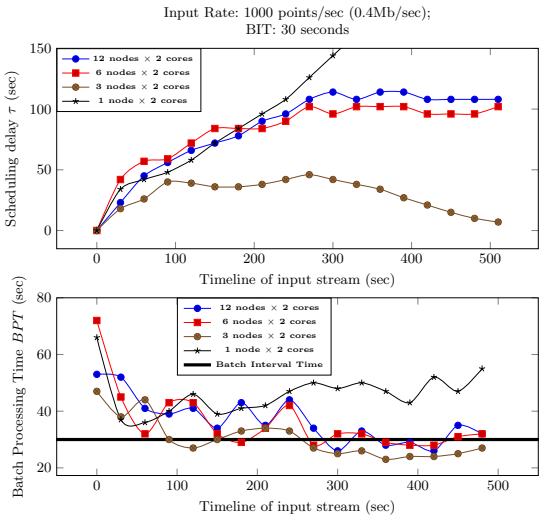

Figure 8. Benchmarking Scalability under Input Rates = 1,000 points/sec

can be estimated by: $\nu_i = T_i + \frac{BIT}{2}$, where BIT is a fixed configurable parameter on Spark Streaming. When $\tau_i = 0$, the system achieves its optimal performance. If the value of τ_i and T_i keep increasing, a "snowball effect" occurs and the system is considered unstable against the input stream. On the other hand, if the value of τ_i and T_i are stable under a red line, this system is considered stable. In the following case studies, we benchmark the RT-DBSCAN on Spark Streaming by using different numbers of Spark executors and different input rates of data streams. τ_i and BPT_i are monitored for evaluating the stability and performance of RT-DBSCAN. The default parameters used in our following RT-DBSCAN benchmarking included DBSCAN parameters: spatio-temporal ϵ calculated by Equation 1, where inputs are: $D_g = 100m$; $\triangle t = 600sec$; $s = 1.667m/sec$ and $minPts = 3$. as well as non-DBSCAN parameters: FC partition config: $maxPts = 100$; $minSize = 2 \times \epsilon$. time $(BIT) = 30$ seconds.

The selected value for the spatial-temporal ratio s reflects walking speed. The source code of RT-DBSCAN implementation using Spark Streaming is available at our GitHub repository (https://github.com/project-rhd/smash-app/tree/master/smash-stream).

Figure 8 presents the results of benchmarking the scalability of RT-DBSCAN. The data input rate here is set to 1,000 points per second. Performance on

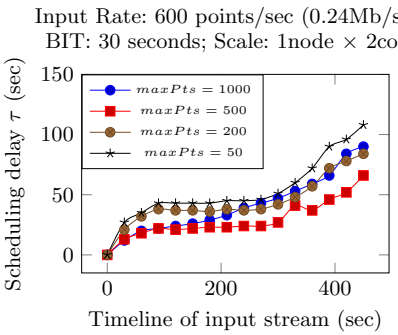

Figure 9. Performance Benchmarking using Different maxPts Settings

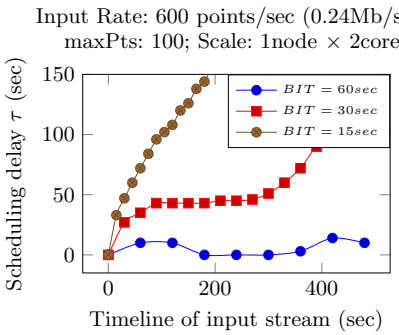

Figure 10. Performance Benchmarking using Different BIT Values

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

different numbers (scales) of Spark executors are benchmarked under the same data stream. The two charts in Figure 8 show the timeline on the x axis and τ_i, BPT_i on the y axis. According to the charts, the first batch usually takes a longer time to process. This is because several initiations are conducted at the beginning of the first tick, *e.g.,* database connections. Almost all the BPT_i of 1 *node* $\times$ 2 *cores* are larger then BIT and thus τ_i keeps growing. This pattern means 1 *node* $\times$ 2 *cores* is not stable with the default parameters in tackling this rate of input data stream. Following this rule, 3 *nodes* $\times$ 2 *cores* is the most stable and efficient scale among the candidates in Figure 8 since its τ_i reaches zero after several batches from the initiation. The delay of 6 *nodes* $\times$ 2 *cores* and 12 *nodes* $\times$ 2 *cores* are even larger than 1 *node* $\times$ 2 *cores* at the beginning of processing. However the trends of their τ_i stablilises and their BPT_i wavers near the BIT line. These two scales are considered stable under this data rate but they are not the optimal options. The overhead of network I/O among multiple nodes is the major bottleneck with these numbers of nodes. To conclude, scaling the number of executors of RT-DBSCAN has a positive effect on its performance but having too many nodes can impact the efficiency due to the data transmissions required over the network. In addition to the number of Spark executors, there are several non-DBSCAN parameters that can impact on the efficiency and stability of RT-DBSCAN under high data velocity situations, *e.g.,* FC partition parameters and the BIT (tick time). Figure 9 benchmarks the performance of RT-DBSCAN according to $maxPts$ which is a parameter/threshold used in the FC partition method. A 1 *node* $\times$ 2 *cores* Spark cluster is used and the input rate is 600 *points/sec.* $maxPts$ has a direct impact on the number of points in each cell and the number of data partitions needed for parallel computing, *i.e.,* it impacts on the granularity of parallelization. As seen, $maxPts = 200$ is the optimal setting for the cluster among all other candidates. We also find that the value of $maxPts$ does not have a significant impact on the performance (delay). Figure 10 benchmarks the performances on BIT *i.e.,* the tick time. A 1 *node* $\times$ 2 *cores* Spark cluster is used and the input rate is 600 *points/sec.* As seen, BIT has a significant impact on the stability of RT-DBSCAN. A larger value of BIT can make the system more stable under higher input rates. However increasing BIT will improve the average delay for processing each input data. Therefore, an elastic BIT is a good strategy to balance the stability and output delay for RT-DBSCAN.

6 Conclusions

In this paper, we propose a new extension of the DBSCAN algorithm for clustering spatio-temporal data targeted specifically to real-time data streams. The novelties of this algorithm are that it tackles ever-growing high velocity data streams and utilizes density based clustering. Furthermore, the spatio-temporal data does not need to arrive in time based sequence.

In the benchmarking, we identify and discuss several configurations that were explored for the performance of RT-DBSCAN over Spark Stream. For future works we shall consider auto-scaling at both the platform level and Spark workers level. We shall also consider In-memory indexing/caching for recent data, *e.g.,* if streamed data arrives in a particular sequence. Data label sensitive clustering *e.g.,* identifying social media data created by the same user in a cluster can also be considered. The FC partition can also be applied to other algorithms for

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40

real-time parallel processing. Finally we are considering variability in the tick times and use of other stream processing engine such as Apache Storm.

References

1. Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record*, volume 28, pages 49–60. ACM, 1999.
2. Derya Birant and Alp Kut. St-dbscan: An algorithm for clustering spatial–temporal data. *Data & Knowledge Engineering*, 60(1):208–221, 2007.
3. B Chandra. Hybrid clustering algorithm. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 1345–1348. IEEE, 2009.
4. Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142. ACM, 2007.
5. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
6. Martin Erwig, Ralf Hartmut Gu, Markus Schneider, Michalis Vazirgiannis, et al. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.
7. Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB*, volume 98, pages 323–333, 1998.
8. Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.
9. Yikai Gong, Luca Morandini, and Richard O Sinnott. The design and benchmarking of a cloud-based platform for processing and visualization of traffic data. In *IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 13–20. IEEE, 2017.
10. Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. The stark framework for spatio-temporal data analytics on spark. In *BTW*, pages 123–142, 2017.
11. Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. On clustering validation techniques. *Journal of intelligent information systems*, 17(2):107–145, 2001.
12. Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, 2014.
13. Alexander Hinneburg, Daniel A Keim, et al. An efficient approach to clustering in large multimedia databases with noise. In *KDD*, volume 98, pages 58–65, 1998.
14. Shuai Ma, TengJiao Wang, ShiWei Tang, DongQing Yang, and Jun Gao. A new fast clustering algorithm based on reference and density. *Advances in Web-Age Information Management*, pages 214–225, 2003.
15. Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. Density-based clustering in spatial databases: The algorithm gdbscan and its applications. *Data mining and knowledge discovery*, 2(2):169–194, 1998.
16. Saed Sayad. Real time data mining. 01 2017.
17. Christian Spieth, Felix Streichert, Nora Speer, and Andreas Zell. Clustering-based approach to identify solutions for the inference of regulatory networks. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 1, pages 660–667. IEEE, 2005.
18. P Viswanath and Rajwala Pinkesh. l-dbscan: A fast hybrid density based clustering method. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 1, pages 912–915. IEEE, 2006.
19. Ji-Rong Wen, Jian-Yun Nie, and Hong-Jiang Zhang. Query clustering using user logs. *ACM Transactions on Information Systems*, 20(1):59–81, 2002.

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93698-7_40