Efficient Skyline Computation for Large Volume Data in MapReduce Utilising Multiple Reducers

Kasper Mullesgaard
Aalborg University
Aalborg, Denmark
kmulle08@student.aau.dk

Jens Laurits Pedersen
Aalborg University
Aalborg, Denmark
jlpe08@student.aau.dk

ABSTRACT

A skyline query is useful for extracting a complete set of interesting tuples from a large data set according to multiple criteria. The sizes of data sets are constantly increasing and the architecture of backends are switching from single node environments to cluster oriented setups. Previous work has presented ways to run the skyline query in these setups using the MapReduce framework, but the parallel possibilities are not taken advantage of since a significant part of the query is always run serially. In this paper, we propose the novel algorithm MapReduce - Grid Partitioning Multiple Reducers Skyline (MR-GPMRS) that runs the entire query in parallel. This means that MR-GPMRS scales well for large data sets and large clusters. We demonstrate this using experiments showing that MR-GPMRS runs several times faster than the alternatives for large data sets with high skyline percentages.

1. INTRODUCTION

Skyline queries have a wide application domain ranging from e-commerce and quality based service selection, to stock trading and, generally speaking, any process involving multi-attribute decision making. Skyline query processing is computationally intensive. In order to handle this high computation cost, commodity computing can be used. Commodity computing is a paradigm where a high number of low cost computers are connected in a cluster to run demanding computations across multiple nodes. Commodity computing is deployed by several notable companies [1, 7, 8, 9, 11].

When using commodity clusters, the idea is to to take advantage of the high number of nodes by processing queries parallelly. High fault tolerance is a requirement since machines can fail, and a larger cluster have a higher probability of machines faulting. MapReduce is designed specifically for high fault tolerance parallel computing.

The problem of this article is to develop a MapReduce algorithm that finds the skyline of a large volume data set efficiently. Similar work on this subject has been done before but previous work presents solutions that uses only a single reducer to find the global skyline, failing to utlilise the MapReduce framework to its full potential. To the best of our knowledge, applying multiple reducers to find the global skyline is unprecedented when finding the skyline for large data sets, and it is the focus of the work presented here.

As a solution to the problem, the MapReduce - Grid Partitioning Multiple Reducers Skyline (MR-GPMRS) algorithm is proposed. The basis of the MR-GPMRS algorithm is, as the name suggests, to partition the input data set using a grid. A bitstring is used to keep track of which partitions in the grid are non-empty, which makes it possible to make decisions based on the distribution of the entire data set. How dominance in the skyline query works combined with the grid partitioning scheme allows splitting the data into parts that can be processed independently of each other, which means that these parts can be processed by different reducers. The goal of MR-GPMRS is to minimise the query response time for data sets with high skyline percentages. To measure this, it is compared to other skyline query processing algorithm that applies the MapReduce framework.

The rest of this paper is organized as follows: Section 2 contains the preliminaries. In Section 3 we introduce MapReduce - Grid Partitioning Single Reducers Skyline (MR-GPSRS): Our bitstring based grid partitioning algorithm for skyline query processing in the MapReduce framework. In Section 4 we introduce MR-GPMRS, a novel extension to MR-GPSRS that allows processing skyline queries in MapReduce using multiple reducers. In Section 6 we present experimental evaluation of the proposed solutions compared with algorithms from the litterature, and finally, in Section 7, we conclude the paper and propose future directions for research.

2. PRELIMINARIES

In this section, the skyline query and the MapReduce framework is described. A table of common symbols used throughout this paper is shown in Table 1.

2.1 The Skyline Query

Given a set of multi-dimensional tuples R, the skyline query returns a set of tuples S_R, such that S_R consists of all the tuples in R that are not dominated by any other tuple in R [3].

Definition 1. A tuple r_i dominates another tuple r_j, denoted by $r_i \prec r_j$, if and only if, for all dimensions, the value

Table 1: A list of common terms.

Symbol	Interpretation
R	A set of tuples
S_R	The skyline of the set of tubles R
t	A tuple
n	Partitions per dimension (PPD)
d	Dimensionality
p	A partition of the data
P	A set of partitions
BS	A bitstring
IG	A group of independent partitions

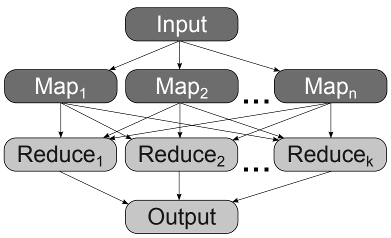


Figure 1: This figure shows the MapReduce process. The input is split between the mappers. The mappers process their input split and output the results to the reducers. The reducers process the results from the mappers, generating the final output.

of r_i is not worse than the corresponding value of r_j, and for at least one dimension, the value of r_i is better than the value of r_j [3].

Whether a value is better or worse than another value is determined by the configuration of the skyline query. Typically, a value v_1 has to be either larger or smaller than another value v_2 for v_1 to be better than v_2. In this paper it is assumed that a smaller value is better.

2.2 The MapReduce Framework

MapReduce is a framework for distributed computing. It is based on a Map and a Reduce function [6]. The Map function is invoked for each record in the input file and it produces a list of key-value pairs. The Reduce function is then invoked once for each unique key and the associated list of values. This produces key-value pairs that are the result of the MapReduce job, i.e., $Map(k1, v1) \rightarrow list(k2, v2)$ and $Reduce(k2, list(v2)) \rightarrow list(k3, v3)$. Several MapReduce jobs can be chained together, later phases being able to refine and/or use the results from earlier phases. The MapReduce process is illustrated in Figure 1.

A distributed file system is used to store the data processed and produced by the MapReduce job. The input file(s) is split up, stored, and possibly replicated on the different nodes in the cluster. The nodes are then able to access their local splits when processing data. When the data from the Map function has been processed by the different nodes, the results are shuffled between the nodes so the required data can be accessed locally when the Reduce function is invoked.

It can be necessary to replicate some data across all nodes. In Hadoop [2], the implementation of MapReduce used for this paper, the Distributed Cache can be used for this purpose. In the beginning of a MapReduce job, data written to the Distributed Cache is transferred to all nodes, making it accessible in the Map and Reduce functions. This paper assumes that the Distributed Cache, or something similar, is available.

2.3 Skyline Query Processing in MapReduce

In an article by Zhang et al. [12] skyline algorithms are adapted for the MapReduce framework. Three different algorithms are presented: MapReduce - Block Nested Loop (MR-BNL), MapReduce - Sort Filter Sort (MR-SFS), and MR-Bitmap. MR-BNL uses BNL and grid partitioning. The second algorithm, MR-SFS, modifies MR-BNL with presorting, but it is shown to perform worse than MR-BNL. MR-Bitmap is based on a bitmap which is used to determine dominance. It is fast computationally but requires a large amount of disk space and is only viable for data sets with few distinct values. A single reducer is used to calculate the final resulting skyline in MR-BNL and MR-SFS. MR-Bitmap does use multiple reducers, and is the only MapReduce algorithm for finding the skyline of a data set we know of to do so. However, as mentioned, it can only handle data sets with low data distinction. In [12], it was not tested on data sets with more than ten thousand distinct values, which is below the threshold for data sets used for testing in this article.

Angular partitioning is a different partitioning technique proposed by Vlachou et al. [10]. Angular partitioning is based on making partitions by dividing the data space up using angles. The idea is based on the observation that skyline tuples are located near the origin. So by dividing the data space up using angles, skyline tuples should be distributed into several partitions while non-skyline tuples should be grouped with skyline tuples that dominates them. The technique is shown to be effective but the global skyline is found using a single node. In an article by Chen et al. [4] the angular partitioning technique is adapted to MapReduce resulting in the algorithm MapReduce - Angle (MR-Angle). The results are comparable to those in [10], and the global skyline is found using a single reducer.

3. GRID PARTITIONING BASED SINGLE REDUCER SKYLINE COMPUTATION

In this section, a skyline algorithm for MapReduce is proposed. The algorithm utilises grid partitioning and bitstrings in order to prune dominance checks between tuples.

3.1 Grid Partitioning

Grid partitioning is a method of splitting up a space where each dimension is divided into n parts, referred to as the Partitions per Dimension (PPD). This gives a regular grid of n^d partitions, termed as P, where d is the dimensionality of the data set. In the context of skyline queries, the dominating relationship between the partitions $p_1, p_2, \ldots, p_{n^d} \in P$ can be exploited to exclude dominance checks between tuples. Partitions have a dominating relationship with each other similar to that between tuples. The main difference is that a dominating relationship between two partitions p_i and p_j is based on their maximum corners $p_i.max$ and $p_j.max$, and minimum corners, $p_i.min$ and $p_i.min$. The maximum corner of a partition is defined as the corner of the partition that has the highest (worst) values. Similarly, the minimum corner of a partition is defined as the corner of the partition that has the lowest (best) values.

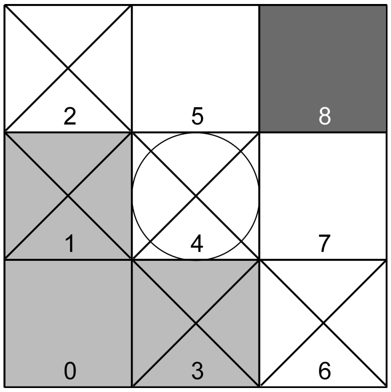


Figure 2: An example of partitions and their offsets. Non-empty partitions are marked with crosses and the dominating (dark grey) and anti-dominating (light grey) regions of the partition with offset 4 (circle) are shown.

Definition 2. A partition p_i dominates another partition p_j, denoted by $p_i \prec p_j$, if and only if $p_i.max$ dominates $p_j.min$. This ensures that all tuples in p_i dominates all tuples in p_j.

$$p_i \prec p_j \Leftrightarrow p_i.max \prec p_j.min \tag{1}$$

If this is not the case, p_i does not dominate p_j, denoted by $p_i \not\prec p_j$.

The dominating relationships between the partitions can be expressed using their individual dominating (Cui et al. [5]) and anti-dominating regions.

Definition 3. Given a partition p_i, its dominating region $p_i.DR$ contains all partitions dominated by p_i:

$$p_i.DR = \{p_j \mid p_j \in P \wedge p_i \prec p_j\} \tag{2}$$

Meanwhile, p_i's anti-dominating region $p_i.ADR$ contains all partitions that can have tuples that dominates $p_i.max$:

$$p_i.ADR = \{p_j \mid p_j \in P \wedge p_j.min \prec p_i.max\} \tag{3}$$

Figure 2 shows an example of the dominating and anti-dominating region of the partition marked as 4 in a two dimensional data set. The non-empty partitions are marked with a cross. The dominating region of partition 4 contains partition 8 and the anti-dominating region contains partitions 0, 1, and 3.

3.2 Bitstring Representation

In grid partitioning, the only partitions of interest are those that are non-empty, i.e. $\{p_i \mid p_i \in P \wedge p_i \neq \emptyset\}$. The partitioning scheme can be represented as a bitstring $BS(0, 1, 2, \ldots, n^d - 1)$ where $for\ 0 \leq i \leq n^d - 1$:

$$BS[i] = \begin{cases} 1 & \text{if } p \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The resulting bitstring can be constructed using either row-major order or column-major order, the only difference being how the offset of a partition in the bitstring is calculated. Column-major order is used in this paper. For example, the offset of the partitions of the two dimensional data set in Figure 2 is indicated by the digit in their lower left corner of the partitions, resulting in the bitstring 011110100.

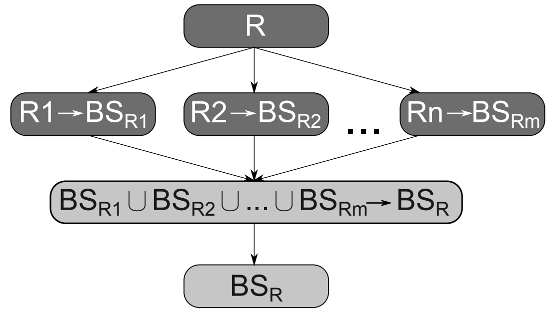


Figure 3: The data flow of the bitstring generation phase of MR-GPSRS with the data set R.

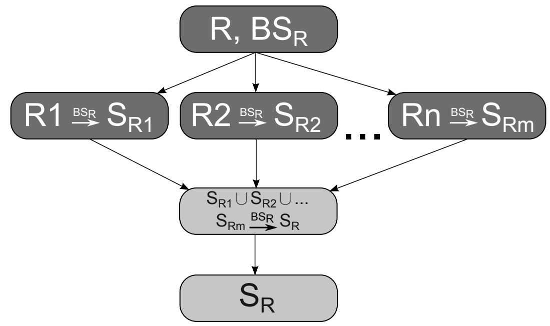


Figure 4: The data flow of the skyline computation phase of MR-GPSRS with the data set R and the bitstring BS_R.

The bitstring can be traversed to prune partitions such that fewer partitions and data tuples are involved in the skyline computation. This can be done by using the dominating relationships between partitions. If $p_i \prec p_k\ for\ p_i, p_k \in P$ then the value of p_k in the bitstring is set to 0, thereby eliminating it from further consideration. If n subsets $R_1, R_2, \ldots, R_n \subset R$ are partitioned with the same grid scheme, this will result in n bitstrings $BS(R_1)$, $BS(R_2)$, ..., $BS(R_n)$. Two or more of these bitstrings can be merged using bitwise or, and if $R_1 \cup R_2 \cup \ldots \cup R_n = R$ then $BS(R_1) \vee BS(R_2) \vee \ldots \vee BS(R_n) = BS(R)$.

3.3 MR-GPSRS Algorithm

The algorithm is divided into two phases: The bitstring generation phase and the skyline computation phase.

In the mappers of the bitstring generation phase, shown in Algorithm 1, a bitstring BS_{Ri} is initialized (line 1), the status of the partitions of the tuples in Ri are set to 1 in BS_{Ri} (lines 2-5), and all mappers send their BS_{Ri} to a single reducer (line 6. In the reducer (Algorithm 2) the global bitstring BS_R is initialized (line 1). A logical OR operation is then performed on the global bitstring BS_R and each of the bitstrings received from the mappers BSS (lines 2-4). The bitstring is then traversed and pruned (lines 5-7) for dominated partitions. The data flow of the bitstring generation phase of MR-GPSRS is represented in Figure 3 where it is shown how the data set R is split into subsets $[R1, Rm]$ that are processed by the mappers into local bitstrings [BS_{R1}, BS_{Rm}]. The reducer then finds the global bitstring BS_R using [BS_{R1}, BS_{Rm}] and outputs BS_R.

In the skyline computation phase, shown in Algorithm 3, the mappers partition their subset Ri of the data set R (lines 1-2). By using the bitstring BS_R from the bitstring genera-

Algorithm 1 *Mapper of the bitstring generation phase*

Input: A subset Ri of the data set R, the dimensionality of the data set d, and the PPD n.
Output: A bitstring BS_{Ri} of the empty and non-empty status of the partitions in the data set RS.
Initialize a bitstring BS_{Ri} with length n^d where all bits are set to 0
for each $t \in Ri$ **do**
  Decide the partition p that t belongs to
  Set the bit that represents the status of p in bitstring BS_{Ri} to 1
end for
Output$(null, BS_{Ri})$

Algorithm 2 *Reducer of the bitstring generation phase*

Input: A set of local bistrings BSS, the dimensionality of the data set d, and the PPD n.
Output: BS_R, the bitstring of the data set R.
Initialize a bitstring BS_R with length n^d where all bits are set to 0
for each $BS_{Ri} \in BSS$ **do**
  $BS_R \leftarrow BS_R \vee BS_{Ri}[i]$
end for
for each partition p with status 1 in BS_R **do**
  set status of partitions in $p.DR$ to 0 in BS_R
end for
Output$(null, BS_R)$

tion phase, the tuples belonging to partitions that have been pruned are discarded (line 3). The *InsertTuple* function (Algorithm 5) is then called on the remaining tuples (line 4), where a tuple t is inserted into a partition p and t is compared with all tuples in p, such that only the local skyline of p is maintained. The mappers, after all tuples have been partitioned, compute their local skyline (lines 7-9) by calling the *ComparePartitions* function (Algorithm 4), where the dominating relationships of the partitions, as specified in Definition 3 (Section 3.1), are used to remove any tuples dominated by other tuples local to that mapper. Each mapper then outputs their set of local partitions P to a single reducer (line 10). At this point in the algorithm, P is equal to S_{Ri}, the skyline of the mappers local subset Ri of the data set R.

In Algorithm 6, the reducer receives the local skylines, in the form of a set of partition sets LS, from the mappers and merges them (lines 1-8). Merging the partitions uses the same function for inserting tuples into partitions as in Algorithm 3, where only the local skyline is maintained in each partition. The reducer then calculates and outputs S_R, the skyline of R, by iterating through the partitions and comparing them with the partitions in their anti-dominating regions (lines 9-11).

The data flow of the skyline computation phase of MR-GPSRS is represented in Figure 4 where it is shown how the data set R is split into subsets $[R1, Rm]$ that are processed by the mappers into local skylines $[S_{R1}, S_{Rm}]$. The reducer then finds the global skyline S_R using $[S_{R1}, S_{Rm}]$ and outputs S_R.

Algorithm 3 *Mapper of MR-GPSRS Skyline Computation*

Input: A subset Ri of the data set R and the bitstring BS_R.
Output: A set of local partitions S_{Ri} where each partition contains local skyline tuples.
1: **for each** $t \in Ri$ **do**
2:   decide the partition p in the set of local partitions P that t belongs to
3:   **if** status of p in BS_R is 1 **then**
4:     $p \leftarrow$ InsertTuple(t, p)
5:   **end if**
6: **end for**
7: **for each** $p \in P$ **do**
8:   $p \leftarrow$ CompareПartitions(p, P)
9: **end for**
10: Output$(null, P)$

Algorithm 4 *ComparePartitions(partition p, set of partitions P)*

Input: A partition p and a set of partitions P
Output: Returns p such that all tuples in p dominated by a tuple in any partition in P are removed.
$ADR = p.ADR \cap P$
for each $p' \in ADR$ **do**
  remove from p tuples that are dominated by tuples in p'
end for
return p

Algorithm 5 *InsertTuple(tuple t, partition p)*

Input: A tuple t and a partition p
Output: Returns p such that it contains t if t is not dominated by any tuples in p. If t dominates any tuples in p, they are removed.
$check = true$
for each $t' \in p$ **do**
  if $t \prec t'$ **then**
    remove t' from p
  end if
  if $t' \prec t$ **then**
    $check = false$
    BREAK
  end if
end for
if check **then**
  add t to p
end if
return p

Algorithm 6 *Reducer of MR-GPSRS Skyline Computation*

Input: The set of local skylines from all the mappers LS in the form of a set of partition sets.
Output: The global skyline S_R.

for each $P \in LS$ **do**
  for each $p \in P$ **do**
    for each $t \in p$ **do**
      decide the partition p' in the set of global partitions P_G that t belongs to
      $p' \leftarrow$ InsertTuple(t, p')
    end for
  end for
end for
for each $p' \in P_G$ **do**
  ComparePartitions(p', P_G)
end for
for each $p' \in P_G$ **do**
  Output$(null, p')$
end for

3.3.1 Choosing the Number of Partitions per Dimension

The PPD is a parameter that is significant for the performance of the algorithm. The reason the PPD is important is that it determines the number of Tuples per Partition (TPP). TPP is important since if there are too few TPP, then the process of comparing each set of partitions is not worthwhile compared to checking the tuples in the partitions. Conversely, if there are too many TPP, the grid is too rough and the number of partitions that can be pruned when comparing partitions becomes less than optimal.

What the optimal PPD is depends on the cardinality, distribution, and dimensionality of the data set, as well as the number of active mappers. Without having extensive knowledge of the algorithm and the data set, choosing the optimal PPD, or even a good PPD, is guesswork. To avoid this, an extension to MR-GPSRS is proposed where the algorithm chooses the PPD itself.

The extension is based on a guess of a good PPD n made in the mappers. This guess is based on the data sets cardinality c, dimensionality d, and the desired TPP. The number of TPP in a given grid can be approximated with the following expression:

$$\frac{c}{n^d} = \text{TPP} \tag{5}$$

From this, n can be isolated:

$$\sqrt[d]{\frac{c}{\text{TPP}}} = n \tag{6}$$

It is then necessary to determine the desired TPP. For the data sets in this article, $100/d$ was found to be a good number. This takes into account the number of dimensions, which affects the time it takes to compare them. What this number should be in different setups might vary. From the guess n, several bitstrings are generated. For example, a bitstring based on n and then four more bitstrings based on PPD values 1 and 2 higher and lower than n. The values should only be used if $2 \leq n \wedge n^d < c$.

The set of bitstrings generated by each mapper, as well as the number of points each mapper processed, is then send to a reducer. To get the global bitstrings, the reducer runs a logical OR operation on the bitstrings from the different mappers that are based on the same n. The reducer then makes an estimate on the number of TPP for each bitstring by dividing the number of non-empty partitions, taken from the bitstring, with the number of tuples processed by each mapper. The reducer can then estimate the remaining tuples each bitstring would have after pruning by using the estimated TPP and the difference in the number of set bits in the bitstrings before and after pruning. The remaining number of tuples and the number of non-empty partitions in each bitstring after pruning are then used by the reducer to make a final estimate of the TPP in each bitstring after pruning. The one that is closest to the desired TPP is then chosen as the final global bitstring used in the rest of the algorithm. This extension is used for the experiments in Section 6.

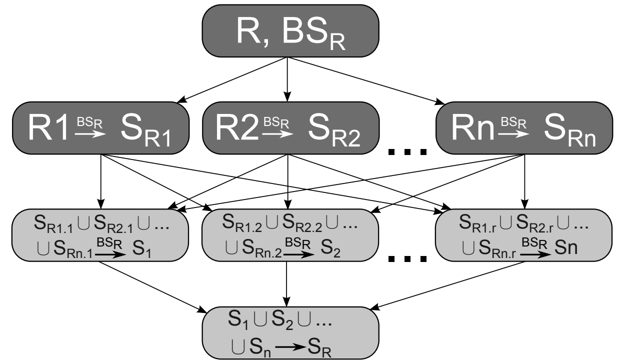


Figure 5: The data flow of the second phase of MR-GPMRS with the bitstring BS_r of the data set R

4. GRID PARTITIONING BASED MULTIPLE REDUCERS SKYLINE COMPUTATION

In this section, an extension to MR-GPSRS is proposed that utilises multiple reducers. MR-GPSRS relies on a single reducer for computing the global skyline, which increasingly becomes a bottleneck when the skyline of the data set becomes larger.

This bottleneck is alleviated by utilising the fact that the grid partitioning technique can be used to identify subsets of partitions for which the skyline can be computed independently, allowing the use of multiple reducers.

4.1 Skyline Query Processing Using Grid Partitioning With Multiple Reducers

In the current methods of computing skyline in MapReduce, the final step of the algorithms require the local skylines from the mappers to be merged into the global skyline by a single reducer. This is due to the inability of mappers to communicate with each other, thereby not having a global awareness of the data set, and that with some partitioning methods, it is not possible to distribute partitions into groups that can be processed independently.

The issue of communication between the mappers is addressed by utilising the bitstring to ensure that the necessary information is available to the mappers. The issue of identifying independent groups of partitions is addressed by

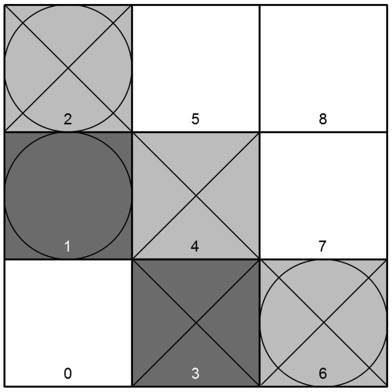


Figure 6: An example showing the distribution of non-empty partitions on two mappers (cross and circle), the partitions belonging to an independent group (grey), and the replicated partitions (dark grey).

utilising the anti-dominating relationship between the partitions in the grid partitioning scheme. The combination of these methods allows the mappers in the algorithm to unanimously decide how the partitions are to be sent to multiple reducers. In order for the mappers to output their partitions to reducers they need to be aware of which partitions are non-empty. A single mapper $mapper_1$ is aware of which partitions are non-empty in its subset. This is, however, not enough for the mapper to decide how the partitions are to be sent, as the distribution of non-empty partitions in another mapper $mapper_2$ might be different. The decision on how to send the partitions made by $mapper_1$ might be different than that made by $mapper_2$, which results in a partition p not being compared with all the partitions in its global anti-dominating region $p.ADR$.

For example, Figure 6 illustrates the non-empty partitions of the mappers $mapper_1 = \{p_1, p_2, p_6\}$ and $mapper_2 = \{p_2, p_3, p_4, p_6\}$. A decision is made by $mapper_1$ to send the partitions p_1 and p_2 to $reducer_1$ and partition p_6 to $reducer_3$. Meanwhile, $mapper_2$ decides to send partition p_2 to $reducer_1$, partitions p_3 and p_4 to $reducer_2$, and partitions p_3 and p_6 to $reducer_3$. In this example, $reducer_2$ would need partition p_1 because $p_1 \in p_4.ADR$. Since $mapper_1$ does not know that p_4 is non-empty, however, it has no way of knowing that it should send p_1 to $reducer_2$. A global bitstring is a way to resolve this problem, since it allows the mappers to know the empty or non-empty status of all partitions.

4.1.1 Grouping Partitions

The grid partitioning scheme allows for identifying *independent groups*: Sets of partitions that can be processed independently to obtain the skyline of those partitions.

Definition 4. A set of partitions IG is independent if and only if the following holds:

$$\{\forall p \in IG \mid p.ADR \subseteq IG\} \tag{7}$$

Independent groups makes it possible for the combined output of multiple reducers to be the global skyline while only having each reducer process a subset of the data set. The following lemma provides a general way to identifying *independent groups*.

Lemma 1. Any group of partitions that consists of a partition p, and the partitions in $p.ADR$, is independent.

PROOF. Consider three partitions p_1, p_2, p_3 that are part of the same regular grid for which the following statements hold:

$$p_2.min \prec p_1.max \tag{8}$$
$$p_3.min \prec p_2.max \tag{9}$$

Since the grid partitions p_1, p_2, p_3 belong to is regular, from Statements 8 and 9, it follows that:

$$p_3.min \prec p_1.max \tag{10}$$

Considering Definition 3 (Section 3.1), from Statements 8, 9, and 10 it follows that:

$$p_2 \in p_1.ADR \land p_3 \in p_2.ADR \Rightarrow p_3 \in p_1.ADR \tag{11}$$

which can be generalized as:

$$p_2 \in p_1.ADR \Rightarrow p_2.ADR \subseteq p_1.ADR \tag{12}$$

Considering Definition 4, it follows that the set of partitions P is independent if:

$$P = p_1.ADR \cup p_1 \tag{13}$$

□

Generating *independent groups* should not be done using arbitrarily chosen partitions and their anti-dominating regions, since this does not exclude the possibility of the *independent groups* being subsets of each other. One way to identify *independent groups* from a set of partitions P, that cannot be subsets of each other, is by using the *maximum partitions* in P.

Definition 5. A partition $p_{max} \in P$ is a *maximum partition* if and only if the following holds:

$$\{\forall p \in P \mid p_{max} \notin p.ADR\} \tag{14}$$

When a group of partitions consists of a *maximum partition* p_{max}, and the partitions in $p_{max}.ADR$, it is independent and it cannot be a subset of another *independent group*.

For example, consider Figure 6. The partition p_2 is a *maximum partition* because it is not in the anti-dominating region of another partition . Making an *independent group* IG_1 from p_2 and $p_2.ADR$ gives $IG_1 = \{p_1, p_2\}$. Similarly, partitions p_4 and p_6 are also *maximum partitions* and making *independent groups* IG_2, IG_3 from p_4 and $p_4.ADR$ and p_6 and $p_6.ADR$ respectively, gives $IG_2 = \{p_1, p_3, p_4\}$ and $IG_3 = \{p_3, p_6\}$. Since these groups are made based on *maximum partitions*, they are not subsets of each other.

It is necessary to replicate some partitions among the *independent groups* as they lie in the anti-dominating regions of partitions in multiple groups, e.g. p_1, p_3. The skyline tuples in a replicated partition are only output by one of the reducers to which the partition is sent. The mappers decides which reducers are responsible for outputting the replicated partitions.

4.1.2 Merging Independent Groups

The number of *maximum partitions* in a data set can be high, and therefore the number of *independent groups* that can be generated will also be high. If the number of *independent groups* is higher than the number of cluster nodes, multiple *independent groups* will be send to the same nodes, and because partitions can be present in multiple *independent groups*, this will cause partitions to be sent to the same nodes multiple times.

Consider the previous example from Figure 6 with the three *independent groups* $IG_1 = \{p_1, p_2\}$, $IG_2 = \{p_1, p_3, p_4\}$, $IG_3 = \{p_3, p_6\}$. In a scenario where IG_1 and IG_2 are sent to the same node, for example, partition p_1 is send twice. It is notable that the overlap between partitions becomes more prominent in higher dimensions where the number of replicated partitions increases, since the dimensionality of the anti-dominating regions of the partitions also increases.

The *independent groups* can be merged, however, to avoid sending the same partitions to the same nodes multiple times. The two groups IG_1 and IG_2 can be merged to form the group IG_{merged} =$\{p_1,p_2,p_3,p_4\}$. These merged groups are then able to be send to reducers without any duplication in their list of partitions, and are called *reducer groups*. The merging of *independent groups* influences both the communication cost and the balancing of computation cost between the reducers. One method of merging is based on optimizing the communication cost: *Independent groups* that have the most partitions in common are merged. This method, however, does not guarrenty any balance of the computations among the reducers as this could leave some *reducer groups* with more unique partitions than other groups. Consider the previous example, the *reducer group* IG_{merged} consists of 3 unique partitions and IG_3 consists of 1 unique partition. This would result in a larger amount of computations for IG_{merged}, assuming uniform amount of tuples in the partitions, as it is forced to compute the skyline for the 3 partitions as they are not present in any other *reducer group*. Preliminary tests have shown that a merging method based on balancing the computations cost between the reducers performed the best, and it was therefore the one used throughout this paper.

How many *reducer groups* that should be generated depends on the size of the data set, the size of the cluster, and the available memory in the nodes. The reducer part of the skyline generation phase requires that the skyline of the local data set is kept in memory.

This means that if the number of *reducer groups* is set too low compared to the size of the data set, the memory may overflow which will cause the runtime to increase significantly. If the number of *reducer groups* is set too high, the same partitions can be sent to the same node several times, or be sent to multiple nodes unnecessarily, causing additional communication. A balance between the two is desirable, such that memory does not overflow and communication cost is not unnecessarily high. In this paper, the number of generated *reducer groups* is set to be equal to the number of nodes in the cluster r. This is fitting since the local skylines of the data sets tested for fits in the memory of the nodes and any unnecessary communication cost is avoided. For larger clusters or larger data sets, generating a number of *reducer groups* equal to the number of nodes might not be optimal.

4.1.3 Outputting Replicated Partitions

Since partitions are replicated in different *reducer groups*, it is necessary to control which reducers output the replicated partitions. When a reducer is responsible for outputting the skyline tuples in a partition, the *reducer group* of that reducer is referred to as being responsible for that partition. A *reducer group* is responsible for all the partitions unique for that *reducer group*. If a partition is present in more than one *reducer group*, one of the *reducer groups* in which it is present is designated as being responsible for that partition. The *reducer group* chosen to be responsible for a replicated partition is based on a calculation of how many comparisons are made between the partitions it contains. The number of comparisons necessary for each partition the *reducer group* is responsible for is calculated based on the bitstring and added up. In order to balance the computation cost for each *reducer group*, the *reducer group* with the lowest number of calculated partition comparisons is made responsible for a replicated partition until responsibility of every partition has been given to some *reducer group*. Since the mappers use the bitstring to form *reducer groups*, the allocation of responsibility is identical across all mappers.

4.2 GPMRS Algorithm

The MR-GPMRS algorithm has two phases; the bitstring generation phase and the skyline computation phase, where the bitstring generation phase is the same as in MR-GPSRS. The skyline computation phase of MR-GPMRS, Algorithm 7, is the same as the skyline computation phase of MR-GPSRS until line 9. There after the *independent groups* are found (line 10), grouped together (line 11), and responsibility of replicated partitions are assigned (lines 12-16). The *reducer groups* are then sent to the reducers (lines 17-21).

Algorithm 7 *Mapper of MR-GPMRS Skyline computation*

Input: A subset Ri of the data set R, the bitstring BS_R, and the number of reducers r.
Output: A set of *reducer groups* RG containing local partitions with the local skyline.

```
1: for each t ∈ Ri do
2:     Decide the partition p in the set of local partitions
       P that t belongs to
3:     if status of p in BS_R is 1 then
4:         p ← InsertTuple(r, p)
5:     end if
6: end for
7: for each p ∈ P do
8:     p ← ComparePartitions(p, P)
9: end for
10: IG ← IndependentGroups(P, BS_R)
11: RG ← ReducerGroups(IG, r)
12: while BS_R contains set bits do
13:     rg_min ← a reference to rg ∈ RG with
        the lowest amount of computations
14:     Assign responsibility of a single p ∈ rg_min
        to rg_min
15:     Set index of p in BS_R to 0
16: end while
17: i = 0
18: for each rg ∈ S_Ri ← RG do
19:     Output(i, rg)
20:     i + +
21: end for
```

In order to maintain consistency throughout the mappers, the bitstring is used to generate the *independent groups*. In Algorithm 8, the independent groups are generated by traversing the bitstring BS_R in reverse in order to find the *maximum partitions* (line 2). For each maximum partition p_{max} an *independent group* is created consisting of p_{max} and the partitions in $p_{max}.ADR$ (lines 3-4). The *independent group ig* is then added to the set of *independent groups IG* (line 5) and the bitstring indexes of the partitions in the *independent group ig* are cleared from the bitstring BS_R (lines 6-7), so that the next traversed bit will be a new maximum partition. The result is a set of independent groups IG (line 9). Consider the previous example of Figure 6 with the independent groups $IG_1 = p_1, p_2$, $IG_2 = p_1, p_3, p_4$ and $IG_3 = p_3, p_5$, the result of the algorithm is the set of independent groups $IG = IG_1, IG_2, IG_3$.

Algorithm 8 *IndependentGroups(P, BS˙R)*

Input: A set of partitions P and a bitstring BS_R.
Output: A set of independent groups IG.
1: **while** BS_R contains set bits **do**
2: $\quad p_{max} \leftarrow$ the $p \in P$ represented by the last set bit in BS_R
3: $\quad ig \leftarrow p_{max}$
4: $\quad$ add $p_{max}.ADR$ to ig
5: $\quad$ add ig to IG
6: $\quad BS_R \leftarrow BS_R \wedge \neg$ bitstring of p_{max}
7: $\quad BS_R \leftarrow BS_R \wedge \neg$ bitstring of $p_{max}.ADR$
8: **end while**
9: **return** IG

Algorithm 9 *ReducerGroups(IG, r)*

Input: A set of independent groups IG and the required number of *reducer groups* r.
Output: A set of *reducer groups* RG.
1: $RG \leftarrow r$ number of empty *reducer groups*
2: **while** $IG \neq \emptyset$ **do**
3: $\quad rg_{min} \leftarrow$ a reference to the $rg \in RG$ with the lowest amount of unique partitions
4: $\quad$ move the partitions in the largest $ig_{max} \in IG$ to rg_{min}, ignoring duplicate partitions
5: **end while**
6: **return** RG

In Algorithm 7 the *independent groups IG* are grouped together into r *reducer groups*, where r is the number of reducers (line 11). In Algorithm 9, the r *reducer groups* are constructed by continually moving *independent groups* from IG to the *reducer group* rig_{min} with the lowest amount of unique partitions (lines 2-5). In the start r empty reducer groups are initialized as the *reducer groups* (line 1). Then rg_{min} is found and the largest (in number of partitions) *indepdent group* ig_{max} is moved from IG to rg_{min} (lines 3-4). The procedure then returns the set of *reducer groups RG* (line 6). Continuing the previous example, with the set of *independent groups* $IG = \{IG_1, IG_2, IG_3\}$. In this example, 2 *reducer group* are generated, so 2 empty reducer groups rg_1 and rg_2 are initialized. The *reducer group* with the lowest number of partitions is chosen, which can be either one, so the first RG_1 is chosen. The *independent group* with the highest number of partitions, IG_2, is then added to the reducer group, $rg_1 = \{p_1, p_3, p_4\}$. Then the *reducer group* with the lowest number of partitions is chosen again, rg_2, and the *independent group* with the most partitions is added to it. This can be either IG_1 or IG_3 since they contain the same number of partitions, so the first one is chosen, $rg_2 = \{p_1, p_2\}$. The *reducer group* with the lowest number of partitions is still RG_2 so it is chosen again and the last *independent group* is added to it, $rg_2 = \{p_1, p_2, p_3, p_6\}$. The set of *independent groups IG* is empty so the reducer groups are finished being generated, $RG = \{rg_1, rg_2\}$.

Responsibility of tuples are assigned to *reducer groups* in Algorithm 7 (lines 12-16). This is done by calculating the increase in number of required dominance checks between partitions the partition would cause if added to the *reducer groups*, which is how many more times line 3 in Algorithm 4 would be run. The partition is then assigned to the *reducer group* with the lowest number of added partition comparisons, rg_{min}.

Algorithm 10 *Reducer of MR-GPMRS Skyline computation*

Input: A set of reducer groups LS containing subsets of the local skylines send to the same reducer from different mappers.
Output: The partitions the *reducer groups* in LS are responsible for.
1: **for each** $RG \in LS$ **do**
2: $\quad$ **for each** $p \in RG$ **do**
3: $\quad\quad$ **for each** $t \in p$ **do**
4: $\quad\quad\quad$ decide the partition p' in the set of global partitions P_G that t belongs to
5: $\quad\quad\quad p' \leftarrow$ InsertTuple(t, p')
6: $\quad\quad$ **end for**
7: $\quad$ **end for**
8: **end for**
9: **for each** $p \in P_g$ **do**
10: $\quad$ **if** *p is responsible* **then**
11: $\quad\quad p \leftarrow$ ComparePartitions$(p,\ P_G)$
12: $\quad$ **end if**
13: **end for**
14: **for each** *responsible* $p \in P_G$ **do**
15: $\quad$ Output$(null, p)$
16: **end for**

Consider the previous example $RG = \{rg_1, rg_2\}$. *Reducer group* rg_1 starts with taking responsibility of the partition it contains that adds the lowest amount of partition comparisons. The number of comparisons a partition requires is equal to the number of partitions in its anti-dominating region. So the partitions in rg_1 with the lowest amount of comparisons are p_1 and p_3, and it chooses the first one p_1. Partition p_1 requires no comparisons so it chooses again and takes responsibility for p_3, which also does not require any comparisons. So rg_1 chooses again and takes responsibility for partition p_4 that requires 2 comparisons. *Reducer group* rg_1 has no more partitions to take responsibility for, so rg_2 takes responsibility of its partitions in order p_2, p_6 and ends up requiring 2 comparisons.

In Algorithm 10, each reducer receives a set of *reducer groups LS*, containing their subset of the local skyline from one or more mappers, which are then merged into a single group MRG (lines 1-8). The reducers then calculates their subset of the skyline of R, by iterating through the partitions they are responsible for and comparing them with the partitions in their anti-dominating regions (lines 9-11). The result is then output (lines 14-16). The data flow of the skyline

Table 2: A list of common terms.

Symbol	Interpretation
$p_{total}(n,d)$	Partitions in a grid
$p_{rem}(n,d)$	Remaining partitions in a grid after pruning dominated partitions
$p_{dom}(n,d)$	Partition dominance checks for a single partition
$s(n,d)$	Partition dominance checks for a single surface in a grid
$g_{mapper}(n,d)$	Partition dominance checks for a single mapper
$g_{reducer}(n,d)$	Partition dominance checks for the reducer with the most partition dominance checks

computation phase of MR-GPMRS is represented in Figure 5 where it is shown how the data set R is split into subsets $R1, R2, \ldots Rn$ that are processed by the mappers into local skylines $S_{R1}, S_{R2}, \ldots S_{Rn}$. The reducers then find subsets of the global skyline $S_1, S_2, \ldots S_n$ using subsets of the local skylines $\{\{S_{R1.1}, \ldots, S_{R1.n}\}, \{S_{R2.1}, \ldots, S_{R2.n}\}, \{S_{Rn.1}, \ldots, S_1, \ldots S_n\}\}$ which are then combined to produce the global skyline S_R as output.

5. COST ESTIMATION

In this section estimations of the cost of MR-GPMRS are made. A table of common terms is shown in Table 2.

5.1 Partition-wise Dominance Tests Estimate

The purpose of this section is to estimate the number of dominance checks performed between partitions in the MR-GPMRS algorithm. Specifically, what is estimated is how many times the line 3 in Algorithm 4 is executed. Due to several uncertainties of the algorithm, it is necessary to make some assumptions.

- **Every partition in every mapper is non-empty.** The data distribution is an important factor of the estimation. It is necessary to assume that there are no empty partitions in order to predict the required number of comparisons. It is comparable to uniform data distribution where it can be expected that most partitions are non-empty.

- **The dominance checks between partitions in the mappers do not lower the number of non-empty partitions.** In practice, the partition checks in the mappers are likely to leave some partitions empty. This is unpredictable and is therefore not accounted for.

These assumptions mean that the scenario for which the estimations are made is a worst case scenario for a data set with a uniform data distribution. When every partition is non-empty, after the grid has been pruned, the location and amount of the remaining partitions is predictable using the dimensionality d and the PPD n. A d dimensional grid has a number of $d-1$ dimensional surfaces equal to $d \times 2$. Half of these surfaces, i.e. d surfaces, are filled with remaining partitions. The remaining part of the other half of the surfaces, as well as the rest of the partitions, are dominated. For example, consider Figure 6. In this 2 dimensional, 3 PPD grid, there are $d \times 2 = 4$ surfaces with a dimensionality of $d - 1 = 1$. These four surfaces consists of the partitions $surf_1 = \{p_2, p_1, p_0\}$, $surf_2 = \{p_0, p_3, p_6\}$, $surf_3 = \{p_6, p_7, p_8\}$, and $surf_4 = \{p_8, p_5, p_2\}$. If every partition were non-empty, the partitions p_4, p_5, p_7, and p_8 would be dominated and pruned. This would leave $d = 2$ intact surfaces, $surf_1$ and $surf_2$. There is an overlap between the surfaces that must be considered. In this case, the overlap between the remaining surfaces $surf_1$ and $surf_2$ is p_0.

The number of remaining partitions after pruning a grid where every partition is non empty $p_{rem}(n,d)$ can be calculated by finding the total number of partitions in a grid $part_{total}(n,d)$ and subtracting a grid one PPD smaller:

$$p_{total}(n,d) = n^d \tag{15}$$

$$p_{rem}(n,d) = part_{total}(n,d) - part_{total}(n-1,d) \tag{16}$$

From the previous example, the pruned partitions, p_4, p_5, p_7, and p_8, can be contained by a $d = 2$ dimensional $n - 1 = 2$ PPD grid. This means the the number of remaining partitions after pruning a 2 dimensional, 3 PPD grid can be calculated as $3^2 - 2^2 = 5$. The dominance checks to be done by a single partition p depends on its anti-dominating region. A partition p_i performs dominance checks against another partition p_j if $p_j \in p_i.ADR$. The number of dominance checks $p_{dom}(n,d)$ for a partition p is equal to its grid position values multiplied with each other minus one:

$$p_{dom}(n,d) = p.pos.d_1 \times p.pos.d_2 \times \ldots \times p.pos.d_d - 1 \tag{17}$$

The position of a partition in a grid is how many partitions, including itself, from the origin a partition is located in the different dimensions.

For example, in Figure 6 the partition p_2, for the first dimension, has the position $p_2.pos.d_1 = 3$, and for the second dimension it has the position $p_2.pos.d_2 = 1$. Summing this up to get the number of partition checks for every partition in a surface $s(n,d)$ yields the following expression:

$$s(n,d) = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \ldots \sum_{i_d=1}^{n} (i_1 \times i_2 \times \ldots \times d_n - 1) \tag{18}$$

To get the number of partition checks for all surfaces, the overlap between surfaces, where they meet on the axes, has to be considered. The first surface is calculated as above. The second surface is also calculated as above but with the overlap between the first and the second surface removed. The overlap between the first and the third surface and the overlap between the second and the third surface has to be subtracted from the third surface, and so on. To account for this, the start index i of one of the summations is incremented for each surface that is calculated. So the number of dominance checks $g_{map}(n,d)$ between partitions in a single mapper for all surfaces of a grid is as follows:

$$s_1(n,d) = \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} \ldots \sum_{i_{d-1}=1}^{n} (i_1 \times i_2 \times \ldots \times i_{d-1} - 1) \tag{19}$$

$$s_2(n,d) = \sum_{i_1=2}^{n} \sum_{i_2=1}^{n} \ldots \sum_{i_{d-1}=1}^{n} (i_1 \times i_2 \times \ldots \times i_{d-1} - 1) \quad (20)$$

$$s_3(n,d) = \sum_{i_1=2}^{n} \sum_{i_2=2}^{n} \ldots \sum_{i_{d-1}=1}^{n} (i_1 \times i_2 \times \ldots \times i_{d-1} - 1) \quad (21)$$

...

$$s_d(n,d) = \sum_{i_1=2}^{n} \sum_{i_2=2}^{n} \ldots \sum_{i_{d-1}=2}^{n} (i_1 \times i_2 \times \ldots \times i_{d-1} - 1) \quad (22)$$

$$g_{map}(n,d) = \sum_{i=1}^{d} s_i(n,d) \quad (23)$$

For the reducer, only a single surface has to be considered. The reason for this is that each surface is an *independent group* that can be calculated individually by the reducers. The reducer with the most dominance checks is the one that has the biggest surface, which is the one where no overlap is considered. This allows the surface calculation from before to be reused when calculating the number of partition dominance checks $g_{reducer}(n,d)$ in the reducer with the most dominance checks:

$$g_{reducer}(n,d) = s_1(n,d) \quad (24)$$

6. EXPERIMENTS

In this section, the results from experimental runs of the algorithms are presented. A cluster of thirteen commodity machines have been used for the experiments. Twelve of the machines have an Intel Pentium D 2.8 GHz Core2 processor. Three of these have a single gigabyte of RAM, four of them have two, and five of them have three. The last machine has an Intel Pentium D 2.13 GHz Core2 processor and two gigabytes of RAM. The machines are connected with a 100 Mbit/s LAN connection. The operating system used is Ubuntu 12.04 and the version of Hadoop is 1.1.0. The algorithms are implemented in Java. Tests are performed on the algorithms MR-GPMRS, MR-GPSRS, MR-BNL from [12], and MR-Angle from [4]. The tests are performed on several different data sets, with varying cardinality, dimensionality, and data distribution.

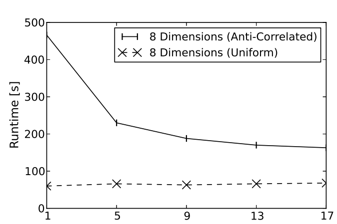
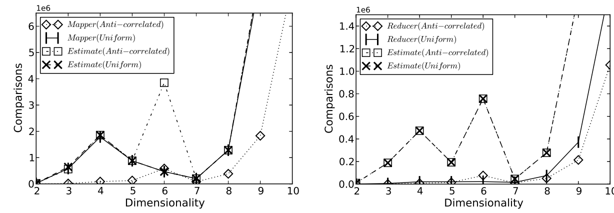


Figure 7: The graph shows the runtime of the algorithm MR-GPMRS run on an anti-correlated data set with a dimensionality of 8 and with a cardinality of 1×10^6. The number of reducers used is varied. The result for one reducer is the runtime of MR-GPSRS.



(a) Mappers **(b)** Reducers

Figure 8: The estimated and actual number of dominance checks between partitions for data sets when run on a data set with a cardinality of 1×10^6 and varying dimensionality.

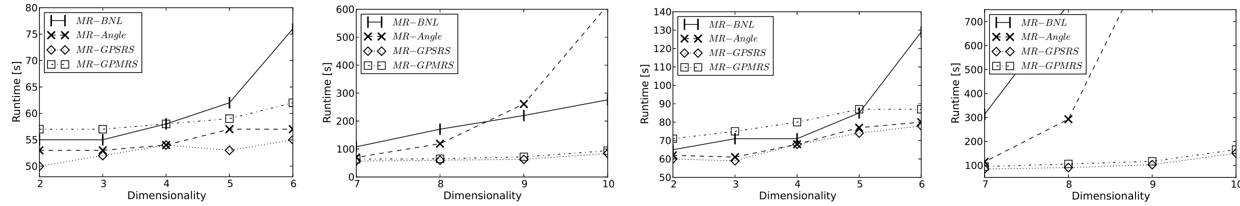
The cardinalities are 1×10^5, 5×10^5, 1×10^6, 2×10^6, and 3×10^6. The dimensionality range is $[2 \ldots 10]$. The distributions are anti-correlated and uniform. For the test runs of MR-GPMRS for comparison with the other algorithms, MR-GPMRS is set to use one reducer per node, i.e. thirteen. To see the effect of the number of reducers, MR-GPMRS has been run on select data sets with varying number of reducers.

6.1 Effect of Dimensionality

The runtime results for uniform data sets with varying dimensionality are shown in Figure 9. The results for anti-correlated data sets are shown in Figure 10.

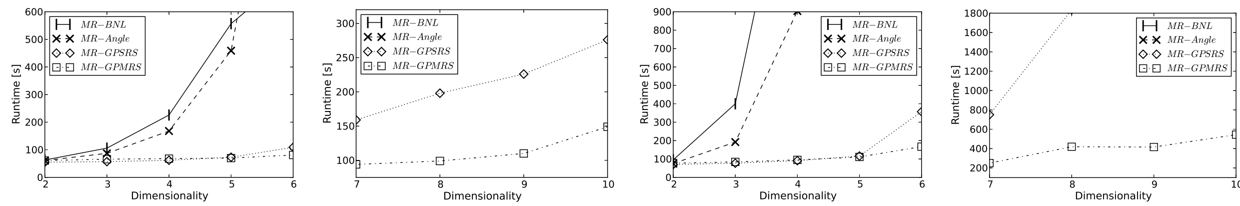
As can be seen from the uniform data set results (Figure 9), the MR-GPMRS algorithm runs a little slower for the lower dimensions $(2-5)$. For the rest of the dimensions $(6-10)$, however, the runtime of MR-GPMRS increases linearly, while the runtime of the other algorithms increases exponentially. The exception is MR-GPSRS that runs faster than MR-GPMRS even for higher dimensions. The difference is marginal but consistent. The reason for these results are that for the low dimensionalities, the skyline percentage is so low that the runtime of the algorithms is dominated by the communication cost and the time it takes to read the data set. This overhead in MR-GPMRS is slightly larger than the other algorithms, which is why the runtime is a bit higher. For the higher dimensions, when the skyline percentage becomes more significant, MR-GPMRS is able to obtain the skyline more efficiently than MR-Angle and MR-BNL. MR-GPSRS is the fastest algorithm for all uniform data sets, which shows that the pruning power of the grid partitioning scheme used in both MR-GPSRS and MR-GPMRS is superior, and that using multiple reducers is not worth the extra communication cost when the skyline percentage is low.

For the anti-correlated data sets (Figure 10), MR-GPMRS is superior for all dimensions, except to MR-GPSRS which is better for the dimensionalities less than 5. It is clear that MR-GPMRS scales well for higher dimensionalities even for large cardinalities, having a runtime of less than ten minutes for a data set with a cardinality of 2×10^6 and 10 dimensions. Results for MR-Angle and MR-BNL are not shown for the higher dimensionalities since they are not able to terminate in a reasonable amount of time. MR-GPSRS is running significantly slower than MR-GPMRS for the low cardinality and high dimensionality data sets. For the higher cardinality, the difference is more clear and MR-GPSRS does not



(a) 2-6 dimensionality 1×10^5 cardinality
(b) 7-10 dimensionality 1×10^5 cardinality
(c) 2-6 dimensionality 2×10^6 cardinality
(d) 7-10 dimensionality 2×10^6 cardinality

Figure 9: Each graph shows the run time of the algorithms MR-BNL, MR-Angle, MR-GPSRS, and MR-GPMRS run on uniform data sets with varying dimensionality.



(a) 2-6 dimensionality 1×10^5 cardinality
(b) 7-10 dimensionality 1×10^5 cardinality
(c) 2-6 dimensionality 2×10^6 cardinality
(d) 7-10 dimensionsilty 2×10^6 cardinality

Figure 10: Each graph shows the run time of the algorithms MR-BNL, MR-Angle, MR-GPSRS, and MR-GPMRS run on anti-correlated data sets with varying dimensionality.

terminate in a reasonable amount of time for the highest dimensionalities. The reason here is that in anti-correlated data sets, the skyline is significant even when the dimensionality is low, and it increases quickly for higher dimensionalities.

This means that the ability of MR-GPMRS to find the skyline more efficiently outweighs its increased overhead. For higher dimensionalities, this is even more evident. That MR-GPSRS performs better than MR-GPMRS for the lower dimensionalities continues to show that using multiple reducers is not worth the extra communication cost when the skyline percentage is below a certain point.

6.2 Effect of Cardinality

The results for varying cardinality is shown in Figure 11. For uniform data, MR-GPMRS is slowest for all cardinalities when the dimensionality is 3, almost 20 seconds at a cardinality of 3×10^6. MR-GPSRS has the best runtime for all cardinalities, MR-Angle tying with it for the cardinalities 1×10^6 and 3×10^6. For the higher dimensionality of 8, MR-GPMRS and MR-GPSRS has the fastest runtimes, MR-GPSRS being a bit faster given the small skyline percentage of the uniform data where the multiple reducers are not an improvement given the extra communication cost. What these results show is that using multiple reducers consistently causes a slower runtime when the skyline percentage of the data set is low.

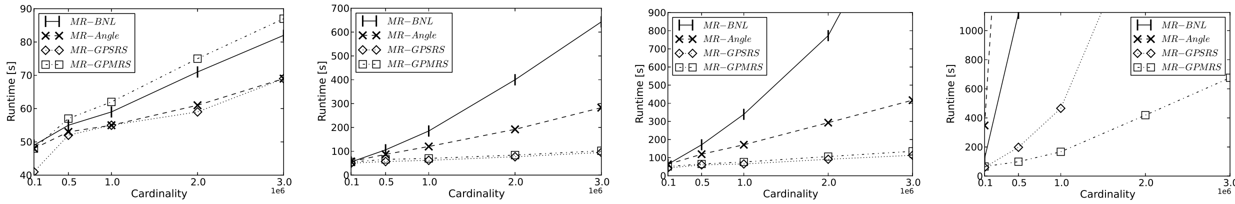
For anti-correlated data, MR-GPMRS and MR-GPSRS is superior for all cardinalities and both dimensionalities. For the lower dimensionality of 3, MR-GPSRS is marginally better than MR-GPMRS, but for the higher dimensionality of 8, MR-GPSRS fails to terminate in a reasonable amount of time for the highest cardinality and is consistently worse than MR-GPMRS.

6.3 Effect of Number of Reducers

Figure 7 shows how the runtime of MR-GPMRS changes when the number of reducers used is varied. What can be seen from the runtimes of the anti-correlated data set is that it lowers when the number of reducers is increased. For this particular data set the runtime is best when the number of reducers is at the highest. It is better even when the number of reducers is higher than the number of nodes. The reason for this is that Hadoop is able to utilise the multiple cores in the nodes to parallelize multiple reducers on the same node. This is not necessarily the case for other data sets, as evident from the results where MR-GPSRS outperforms MR-GPMRS. The skyline percentage needs to be high enough for the extra communication cost the multiple reducers causes for the high number of reducers to be useful. The runtime for the data set with uniform distribution almost does not change when the number of reducers is increased. There is a small increase, caused by the additional overhead of multiple reducers. This uniform data set has a much smaller skyline than the anti-correlated one, which means that the use of multiple reducers is not worthwhile given the extra overhead.

6.4 Evaluation of Estimate

Figure 8 shows the estimated number of dominance checks between partitions, as described in Section 5, compared with the measured number of partition comparisons in MR-GPMRS. The measured numbers has been taken from the mapper and the reducer that had the highest number of comparisons. The results for the mappers show that the estimate does not deviate far from the measured number when the data distribution is uniform, and in some cases there is no deviation. The estimate, however, is inaccurate



(a) Uniform 3 dimensionality **(b)** Anti-correlated 3 dimensionality **(c)** Uniform 8 dimensionality **(d)** Anti-correlated 8 dimensionality

Figure 11: The graphs show the runtime of the algorithms MR-BNL, MR-Angle, MR-GPSRS, and MR-GPMRS run with a dimensionality of 3 and with varying cardinality.

when estimating the number of partition dominance checks for mappers run on the anti-correlated data set. This is to be expected as the estimation was done on the premise that the data set is uniform. It is notable that the estimated number of dominance checks is above the measured one in every case, which means that the estimate can be still be used as a worst case estimate for anti-correlated data sets. The results for the reducers show that the estimate is inaccurate for the reducers, both for the uniform and the anti-correlated data set. The reason that the estimate is inaccurate even for the uniform data set, is that no way was found to mathematically predict the way the *reducer groups* are generated, so instead a number was used that was ensured to be the worst case.

7. CONCLUSION

In this paper, two novel algorithms, MR-GPSRS and MR-GPMRS, for skyline query processing in MapReduce are proposed. The main feature of the algorithms are that they allow decision making across mappers and reducers. This is accomplished by using a bitstring describing the partitions empty and non-empty state across the entire data set. In addition, the common bottleneck of having the final skyline computed at a single node is avoided in the MR-GPMRS algorithm by utilizing the bitstring to partition the final skyline computation among multiple reducers.

The experiments conducted show that the algorithms proposed in this paper consistently outperforms existing algorithms and they scale well with the skyline and data set size. Which of the two proposed algorithms performs better depends on the data set. When the skyline percentage is high, MR-GPMRS performs significantly better while MR-GPSRS performs marginally better when the skyline percentage is low. Running test on a large cluster would be interesting in order investigate how well MR-GPMRS scales for large numbers of reducers.

The increased communication cost incurred by having reducers receive replicated partitions is a consequence of the method that allows for the utilization of multiple reducers. One research direction, therefore, is to develop a scheme that balances the *reducer groups* in MR-GPMRS such that the computations is balanced across the reducers and the communication cost is minimized at the same time.

As the results show, using multiple reducers is not the best option when the skyline percentage is low. So for the algorithm to perform optimally for any data set, it is necessary to develop a scheme that allows MR-GPMRS to intelligently decide how many reducers to use. It is possible that the method for choosing the PPD in the proposed algorithm can be improved.

The method does not incur a noticeable amount of extra runtime, and it chose a viable PPD for all the data sets tested for. It did not necessarily choose the optimal PPD, however, so finding a method that is guaranteed to choose the optimal PPD for any data set would be a significant improvement.

8. REFERENCES

[1] Amazon. Amazon elastic compute cloud. `http://aws.amazon.com/ec2/`.

[2] Apache. Welcome to apache™hadoop®! `http://http://hadoop.apache.org//`.

[3] S. Brzsnyi, D. Kossmann, and K. Stocker. The skyline operator. *17th Internation Conference on Data Engineering*, 2001.

[4] L. Chen, K. Hwang, and J. Wu. Mapreduce skyline query processing with a new angular partitioning approach. *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012.

[5] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, and Y. C. Zhou. Parallel distributed processing of constrained skyline queries by filtering. *24th ICDE*, 2008.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

[7] Facebook. Hive - a petabyte scale data warehouse using hadoop. `https://www.facebook.com/note.php?note_id=89508453919`.

[8] LinkedIn. A professional network built with java technologies and agile practices. `http://www.slideshare.net/linkedin/linkedins-communication-architecture`.

[9] Twitter. Hadoop at twitter. `http://engineering.twitter.com/2010/04/hadoop-at-twitter.html`.

[10] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. *SIGMOD '08*, 2008.

[11] Yahoo. Hadoop at yahoo! `http://developer.yahoo.com/hadoop/`.

[12] B. Zhang, S. Zhou, and J. Guan. Adapting skyline computation to the mapreduce framework: Algorithms and experiments. *DASFAA Workshops*, 2011.