GSTF Journal on Computing (JOC)
DOI: 10.5176/2251-3043_4.4.356
ISSN:2251 - 3043 ; Volume 5, Issue 1; 2016 pp. 69 - 76


# Efficient Computation of Group Skyline Queries on MapReduce

Ming-Yen Lin
Dept. of IECS
Feng Chia University
Taichung, Taiwan
linmy@mail.fcu.edu.tw

Chao-Wen Yang
Dept. of IECS
Feng Chia University
Taichung, Taiwan
m0260836@mail.fcu.edu.tw

Sue-Chen Hsueh
Dept. of IM
ChaoYang Univ. of Technology
Taichung, Taiwan
schsueh@cyut.edu.tw
*Corresponding author

***Abstract*—Skyline query is one of the important issues in database research and has been applied in diverse applications including multi-criteria decision support systems and so on. The response of a skyline query eliminates unnecessary tuples and returns only the user-interested result. Traditional skyline query picks out the outstanding tuples, based on one-to-one record comparisons. Some modern applications request, beyond the singular ones, for superior combinations of records. For example, fantasy basketball is composed of 5 players, fantasy baseball of 9 players, and a hackathon of several programmers. Group skyline aims at considering all the groups comprising several records, and finding out the non-dominated ones. Because of the high complexity, few studies have been conducted and none has been presented in either distributed or parallel computing. This paper is the first study that solves the group skyline in the distributed MapReduce framework. We propose the MRGS algorithm to generate all the combinations, compute the winners at each local node, and find out the answer globally. We further propose the MRIGS algorithm to release the bottleneck of MRGS on unbalanced computing load of nodes. Finally, we propose the MRIGS-P algorithm to prune the impossible combinations and produce indexed and balanced MapReduce computation. Extensive experiments with NBA datasets show that MRIGS-P is 6 times faster than the MRGS algorithm.**

***Keywords-skyline query, group skyline, combinatorial skyline query, MapReduce***

## I. INTRODUCTION

Modern databases and information systems have evolved support mechanisms to satisfy vague or imprecise user requirements [1, 2]. One such mechanism is the skyline query, which is widely used in commercial applications, such as multi-criteria decision analysis, data mining, and navigation. Many real-world scenarios require a combination of two or more tuples in order to find the best option. The use of *k* points to organize a group results in a k-group, the most famous of which is online fantasy sports games in which users select their favorite team from an active database of player statistics. Around the world, the fantasy sports industry is bringing in billions of dollars.

To further illustrate application of a k-group, let us consider two types of fantasy sports: basketball and baseball. Basketball requires five people to form a team; therefore, our aim was to compile a team of the five best players, referred to as the 5-group. These five players are selected from among four hundred players, whereupon statistical data for the team is compiled by summing the values associated with the five players. To ensure a competitive team the user aims to select a team that cannot be dominated by any other teams. Fantasy baseball operates similarly except that nine players are selected instead of five. Any increase in the number of people participating in a game will produce exponential growth in computing costs.

Group skyline queries have not attracted as much attention from researchers as have traditional skyline queries [7, 9, 23]. The intuitive approach would be to find skyline points in dataset D for the generation of a group skyline. However in practice this approach is generally not feasible. In the following, we provide examples to illustrate the contradictions inherent in this approach. Consider the six players listed in Fig. 1.1 from which we need to select three players to make up a team. Table 1.1 lists the statistics associated with the six players. As shown in Fig. 1.1, three of these points ($P_1$, $P_2$, $P_5$) are skyline points. An intuitive approach would result in the selection of the group (P1, P2, P5). The brute-force method leads to enumeration of all groups from C (6, 3), as shown in Table 1.2. The attributes of each combination are generated using the sum operation. In Fig. 1.2, we can see that g3, g6, and g13 form the group skyline. Only g3 includes a skyline point; therefore, the other groups are incompatible with the intuitive solution.

Obtaining the group skyline is a computationally heavy task, the complexity of which increases exponentially with the amount of data. For example, there are approximately 500 active NBA players and each is generally represented by the following five attributes: points, rebounds, steals, assists, and blocks. This leads to a total of $C_5^{500}$ possible combinations. Each group has five players; therefore, we need to sum the statistics to generate new group statistics. Only after generating all possible groups can we find all group skylines; however, this incurs high computational costs. Any increase in the number of tuples leads to exponential growth in computing costs. Selecting five people from among 50 produces $C_5^{50}$=2,118,760 possible combinations. Doubling the number of people increases this to to $C_5^{100}$ =75,287,520 possible combinations. In this sample, doubling the number of people increases the calculation by approximately 35 times. Thus, determining an effective means to solve these queries is a pressing challenge.

The group skyline approach has two main problems: considerable computational overhead, C (m, n) and high levels of memory required to store all candidate sets.

Thus far, researchers have failed to provide a distributed solution for the group skyline problem facing the immense computational costs of massive candidate sets. The purpose of this paper was to develop an efficient group skyline algorithm based on MapReduce and then conduct experiments to assess the validity and effectiveness of the proposed method.

Table 1 lists the symbols used in this paper. A given dataset **D** contains many points **P**; i.e., **D** = ***{P1, $P_2$ …, $P_n$}***. Each point **P** has m attributes; i.e., ***P* = [*$A_1$, $A_2$ …, $A_m$*]**. We assume that the value of each attribute is a positive integer. In the following discussion, we operate under the assumption that a larger value is always a better target.

In the following we use an example to introduce the concepts of group domination and aggregate functions. Figure 1.4 presents two three-member combinations: teams G and G'. The use of the SUM function results in <10, 9> and <8, 9>. According to ***Definition 5***, G dominates G '. However, using the MAX function, we obtain <4, 5> and <5, 5>. According to ***Definition 5***, G' dominates G. This example clearly illustrates that the domination relationship between these two groups differs according to ***Aggregate function F***.

TABLE I. PLAYER DATASET

| Player | Points | Rebounds |
|---|---|---|
| $P_1$ | 6 | 5 |
| $P_2$ | 10 | 0 |
| $P_3$ | 3 | 6 |
| $P_4$ | 3 | 3 |
| $P_5$ | 4 | 6 |
| $P_6$ | 2 | 2 |

TABLE II. TABLE 1.2 AL GROUP OF C (6, 3)

| Group | Member | Points | Rebounds |
|---|---|---|---|
| $G_1$ | P1, P2, P3 | 19 | 11 |
| $G_2$ | P1, P2, P4 | 19 | 8 |
| **$G_3$** | **P1, P2, P5** | **20** | **11** |
| $G_4$ | P1, P2, P6 | 18 | 7 |
| $G_5$ | P1, P3, P4 | 12 | 14 |
| **$G_6$** | **P1, P3, P5** | **13** | **17** |
| $G_7$ | P1, P3, P6 | 11 | 13 |
| $G_8$ | P1, P4, P5 | 13 | 14 |
| $G_9$ | P1, P4, P6 | 11 | 10 |
| $G_{10}$ | P1, P5, P6 | 13 | 13 |
| $G_{11}$ | P2, P3, P4 | 16 | 9 |
| **$G_{12}$** | **P2, P3, P5** | **17** | **12** |
| $G_{13}$ | P2, P3, P6 | 15 | 8 |
| $G_{14}$ | P2, P4, P5 | 17 | 9 |
| $G_{15}$ | P2, P4, P6 | 15 | 5 |
| $G_{16}$ | P2, P5, P6 | 16 | 8 |
| $G_{17}$ | P3, P4, P5 | 10 | 15 |
| $G_{18}$ | P3, P4, P6 | 8 | 11 |
| $G_{19}$ | P3, P5, P6 | 9 | 14 |
| $G_{20}$ | P4, P5, P6 | 9 | 11 |

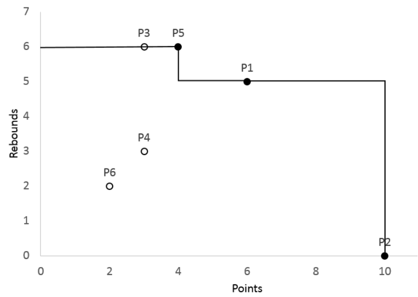


Fig. 1.1 Example of a skyline point

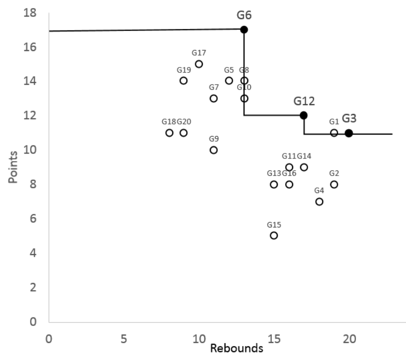


Fig. 1.2 Example of a group skyline

## II. RELATED WORK

The previous research most relevant to this study on skyline groups can be found in [6], [8], and [22]. The main bottleneck in a skyline group is memory, as an unfeasibly large amount of memory space is commonly required to store all of the candidate sets. An incremental approach is proposed in [8] to overcome this problem. That method is based on the following equation: $S_k(D) = S(S_k - (D - \{p\} \cup \{G_{k-1} \cup \{p\} | G_{k-1} \in S_{k-1}(D - \{p\})$

. This method aims to find $Sky_k^n$. This is accomplished by first finding $Sky_k^{n-1}$ and $Sky_{k-1}^{n-1}$ . $Sky_k^{n-1} \cup Sky_{k-1}^{n-1}$ is equal to $Sky_k^n$, as shown in Fig. 2.1.

In [22], search space pruning and input pruning are proposed to filter the number of input tuples. This approach enables the algorithm to reduce the number of combinations in subsequent generations. The aim of input pruning is to find the points dominated by k or more points. Points can then be safely removed without affecting the final results. If point P is dominated by h points (h ≥ k) and G contains point P, we generate another group G' by replacing P in G with h. Then G 'always dominates G. Thus, G containing point P is not a group skyline.

Search space pruning (SSP) and incremental pruning (IP) are based on the same concept; however, SSP is implemented using dynamic programming to reduce computational cost, as shown in Fig. 2.2.

Single tuples can be combined to generate new (combinatorial) tuples. Combinatorial skylines [6] and skyline groups pose similar problems. In [6], ***Aggregate function f*** is defined by ***Combinatorial functions f***. A plurality of tuples produce combination $\mathbf{g_p}$ using ***Combinatorial functions f***. This paper presents two methods to deal with combinatorial skyline problems.

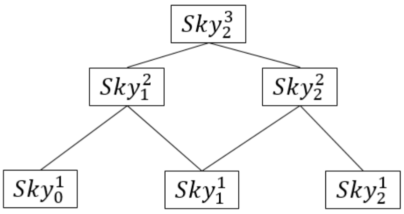


Fig. 2.1 Incremental method

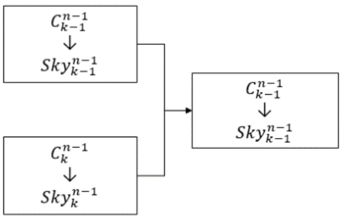


Fig. 2.2 Search space pruning

## III. Proposed Algorithms

In this section, we present three MapReduce algorithms. First, we propose the MR-Group Skyline (MRGS) method on which two-stage MapReduce is used to address the group skyline. This method is described in Section 3.1. Second, we use an index to ameliorate the problem of workload imbalance in the MRGS. This method is described in Section 3.2. Third, we propose the theorem Cascaded-pruning, which is used to reduce the number of candidate sets, this method is described in Section 3.3.

### *A. MR-Group Skyline (MRGS) method*

This algorithm employs MapReduce in two phases. Figure 3.1 illustrates the overall structure of the algorithm. The left half represents the first phase, which is responsible for generating all possible k point groups. The right half represents the second phase, which is responsible for detecting group skylines. Dataset D is input into the first phase to generate all combinations Dk. Then Dk is input into the second phase to generate the group skyline.

In the first phase, to calculate all possible combinations, dataset D is partitioned into three blocks, as shown in Fig. 3.2. Each mapper produces a number of keys according to the number of reducers. In the following example, **num** is used to represent the number of reducers. In Fig. 3.2, **num** is equal to 2; therefore, the mapper produces two key-value pairs, with the values of 1 and 2, respectively. Each point is duplicated **num** times before being sent to the reducer based on a key. After receiving the intermediate results, the reducer begins generating combinations. We assume that dataset D includes [$P_1$, $P_2$, $P_3$, $P_4$, $P_5$, $P_6$] and that there are two reducers. This method generates a combination of all $P_n$ prefixes, which undergo round-robin distribution. In Fig 3.2, r1 is used to illustrate the meaning of the prefix. Our aim is to find the 2-group. In the round robin stage, r1 obtains P1, P3 and P5 to generate [($P_1$, $P_2$), ($P_1$, $P_3$), ($P_1$, $P_4$), ($P_1$, $P_5$), ($P_1$, $P_6$), ($P_3$, $P_4$), ($P_3$, $P_5$), ($P_3$, $P_6$), ($P_5$, $P_6$)]. Following completion of this phase, all possible combinations are output.

Figure 3.2 illustrates the process of the Map phase in which each point is duplicated twice. The mapper generates key-value pairs and the reducer generates groups according to the $P_{in}$ prefix. In the following, we use r1 to illustrate this process.

Figure 3.2 illustrates the process of the Reduce phase. Reducers are used to generate combinations according to in prefix. For example, reducer r1 calculates the prefixes i1, i2, and i3. Prefix i4 is equal to 7, which is greater than |D|; therefore, it is not processed. The reducer r1 utilizes points $P_1$, $P_3$, and $P_5$ as prefixes for the generation of output combinations. There is no particular need to specify the key value at the output. Reducers r1 and r2 are set to 1, as shown in Fig. 3.2. When the first phase ends, we obtain obtain $D_k$.

In the second phase, the group skyline is calculated. The Map function uses the output Dk of the first phase as input for the second. Each map receives a portion of $D_k$ with which to calculate the local group skyline. The mapper generates 1-value pairs. Because only one reducer is used to process the global group skyline, all of the key values are 1. After the reducer receives the map output, it detects the global group skyline and outputs the results *as* $\mathbf{Sky_k}$, as shown in the Fig. 3.3.

The output from the first phase is received by the mapper of the second phase, as shown in Fig. 3.4. Each map calculates a local group skyline. Mapper m1 receives 5 groups and outputs 3 group skylines. The reducer collects all of the local group skylines from the mapper in order to calculate the global group skyline. Reducer r1 receives local skyline groups from mapper m1, m2, and m3 with which to calculate the final results, as shown in Fig. 3.4. In this paper, the skyline algorithms use the BNL method.

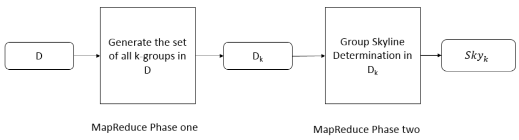


Fig. 3.1 Overview of proposed method

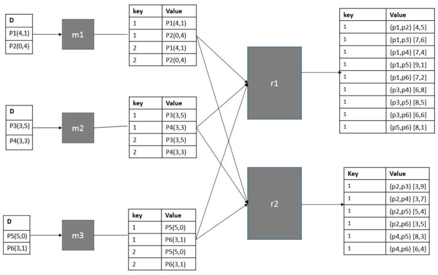


Fig. 3.2 Phase One: Group Generation

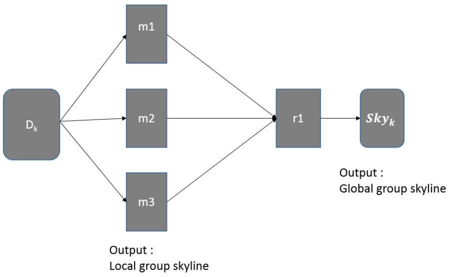


Fig. 3.3 Second phase of MapReduce

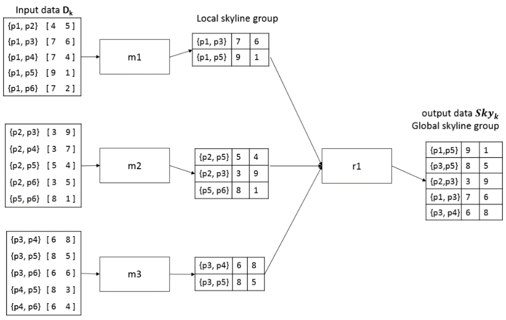


Fig. 3.4 Example of Phase Two

## B. *MR-Index Group Skyline (MRIGS) method*

The MRGS is able to process group skyline problems; however, this method can lead to computational load imbalance. Suppose that the dataset holds data associated with 300 players. MRGS uses four reducers to generate the 3-group. In the results of the first phase, the r1 produces 1.4 million groups, r2 and r3 produce 1.1 million groups, and r4 produce 0.8 million groups. Obviously the workload of r1 is larger than that of the other reducers. In this study, we propose a new method, referred to as the MapReduce Index Group Skyline (MRIGS), which uses C (n, k) group average distribution to achieve load balancing.

This algorithm implements MapReduce in two phases. It differs from the MRGS only in the first phase. The relevant modifications are illustrated in Fig. 3.5.

Our aim is to find all k-groups in D in order to determine the number of groups generated by C (m, k). S represents the number of generated groups, denoted as G [$G_1$, $G_2$, $G_3$..., $G_s$]. We can identify the members of $G_x$ ($1 \le x \le s$) by implementing the combination formula. For example, r1 generates G5. The members of G5 obtained using the combination formula are P1, P2, P3, P4, P8. These points are then used to calculate the value of G5. Therefore, when we know that will generate S group. Our aim is to distribute these groups evenly to every reducer.

In the Map phase, the number of reducers is used to generate key-value pairs to be sent to each reducer. In the Reduce phase, the reducer receives the data after calculating SD (i.e. SD=S/num). Then it based its ID to generate the combination. For example, the six points of D are used to obtain the 2-groups. This results in the generation of 15 groups (S=$C_2^6$=15). As shown in Fig. 3.6, this method is first builds an index of the dataset using two reducers, such that SD = 7. Reducer r1 then generates 7(SD) combinations: G1 ~ G7. Reducer r2 generates the remainder of the combinations: G8 ~ G15 (Fig. 3.6).

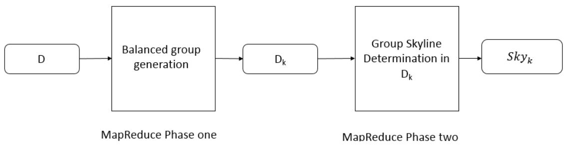


Fig. 3.5 Overview of proposed method

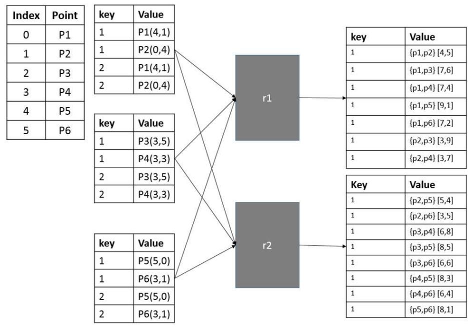


Fig. 3.6 Example of Reduce-Index used in first phase

## C. *MR-Index Group Skyline Pruning (MRIGS-P) method*

MRGS and the MRIGS are able to process group skyline problems; however, the computational complexity of these two methods is still high. To ameliorate this, certain unnecessary points can be pruned before the algorithm is tasked with producing all of the combinations.

In this study, we propose a new method based on input pruning. The proposed method uses two tests to achieve pruning and reveal new features in MapReduce. The proposed method is referred to as ***Cascaded-pruning***.

Given dataset D and point $\mathbf{P_i} \in$ D, let $\mathbf{P_i.C}$ denote the number of points that dominate $P_i$, and $\mathbf{P_i.DL}$ denote the set of points that are dominated by $P_i$. Theorem 1 below is used to prune the points that cannot be included in the combination.

**Lemma 1**. Point $P_i$ can be safely pruned if $P_i.C \ge k$.

**Lemma 2**. If point $P_i$ can be pruned, then all of the points in $P_i.DL$ can be pruned.

Lemma 1 is based on the supposition that if the number of points dominating Pi is greater than or equal to k, then any combination with Pi will be dominated by the combination of the points selected from the set dominating Pi. Lemma 2 is also obvious because $P_i$ dominates any single point in $P_i$.DL; therefore, when $P_i$ is pruned, all of the points in $P_i$.DL can be pruned.

***Theorem 1 (Cascaded-pruning).*** Given dataset D and point $P_i \in$ D, $P_i$, set $P_i$.DL can be safely pruned if $P_i.C \geq k$.

***Proof.*** Let $P_i.C = k$ and $P_x \in P_i$.DL, then $P_x$.C is at least k+1 since $P_x$ is dominated by $P_i$. Consequently, $P_x$ can be safely pruned because $P_x.C \geq k$. Therefore, all the points in Pi.DL can be safely pruned if Pi is pruned.

First, each point increase two attributes Pi.C and Pi.DL. In the Map phase, the algorithm performs input pruning and records $P_i$.C and $P_i$.DL from the surviving points. Map sends out the point when $P_i$.C is less than k. The remaining points are pruned to prevent unnecessary points being sent to the reducer. After the reducer receives all of the points, it performs ***Cascaded-pruning***. As it receives points from different maps, they must be checked at least once. The points received by the reducer are not compared with other points from the same map. In Cascaded-pruning, if there is a Pi.C larger than or equal to k, then this point and the set Pi.DL will be pruned. This feature can save the cost of pruning.

For the example, consider the sixteen players listed in Table 3.1, in which the points that will eventually be pruned are marked in boldface. In this example, as long as Pi.C is greater than or equal to 2, then Pi will be pruned. P2, P7, P8, P9, P11, P12, and P14 are surviving points. An input pruning method was proposed in [24]; this method accesses every point and records the count of these points. This count is then used to determine which points need to be pruned. The method proposed in this paper does not need to access every point in order to prune unnecessary points.

TABLE III. TABLE 3.1 INPUT DATASET

| *Player* | *Points* | Rebounds | Pi.C |
|---|---|---|---|
| ***$P_1$*** | ***6*** | ***5*** | ***2*** |
| $P_2$ | 10 | 0 | 0 |
| ***$P_3$*** | ***3*** | ***6*** | ***3*** |
| ***$P_4$*** | ***3*** | ***3*** | ***10*** |
| ***$P_5$*** | ***4*** | ***6*** | ***2*** |
| ***$P_6$*** | ***2*** | ***2*** | ***12*** |
| $P_7$ | 7 | 4 | 1 |
| $P_8$ | 8 | 5 | 0 |
| $P_9$ | 6 | 7 | 0 |
| ***$P_{10}$*** | ***4*** | ***5*** | ***5*** |
| $P_{11}$ | 0 | 10 | 0 |
| $P_{12}$ | 8 | 2 | 1 |
| ***$P_{13}$*** | ***6*** | ***4*** | ***4*** |
| $P_{14}$ | 4 | 7 | 1 |
| ***$P_{15}$*** | ***5*** | ***1*** | ***7*** |
| ***$P_{16}$*** | ***7*** | ***3*** | ***2*** |

TABLE IV. TABLE 3.2 INPUT DATASET M1 IN MAPREDUCE

| *Player* | *Points* | Rebounds | Pi.C | Pi.DL |
|---|---|---|---|---|
| ***$P_1$*** | ***6*** | ***5*** | ***2*** | ***$P_4$, $P_6$*** |
| $P_2$ | 10 | 0 | 0 | |
| ***$P_3$*** | ***3*** | ***6*** | ***2*** | |
| ***$P_4$*** | ***3*** | ***3*** | ***0*** | |
| $P_5$ | 4 | 6 | 1 | |
| ***$P_6$*** | ***2*** | ***2*** | ***0*** | |
| $P_7$ | 7 | 4 | 1 | |
| $P_8$ | 8 | 5 | 0 | $P_7$ |
| $P_9$ | 6 | 7 | 0 | $P_5$ |

In the MapReduce environment, D is partitioned into multiple sub-blocks. In this example, D is partitioned into two sub-blocks (M1 and M2), as shown in Tables 3.2 and 3.3. In the Map phase, the algorithm detects two blocks using the above method. Map sends out the point when Pi.C less than 2. In the Reduce phase, the reducer receives data from all of the blocks, as shown in Table 3.4. In this table, the mapper column lists the points that belong to each mapper, which are used to perform Cascaded-pruning. These points are not compared with other points from the same map. The reducer checks the result of P5 in Table 3.5. Pi.DL (N) represents the point newly added in the reducer phase. This column is established for the convenience of explanation; the actual algorithm is the same Pi.DL. After the ***Cascaded-pruning*** is complete, $P_{10}$ is recorded in the $P_5$.DL. $P_5$ is pruned because $P_5$.C is equal to 2. According to Theorem 1, when P is pruned, the point at $P_5$.DL is also pruned; therefore, $P_{10}$ can be pruned. When $P_7$ is checked by the reducer, $P_7$ is not compared with $P_{10}$.

Table 3.6 presents the final results. Table 3.6 and Table 3.1 contain the same results, which demonstrates that unnecessary points can indeed be safely pruned.

TABLE V. TABLE 3.3 INPUT DATASET M2 IN MAPREDUCE

| *Player* | *Points* | Rebounds | Pi.C | Pi.DL |
|---|---|---|---|---|
| $P_{10}$ | 4 | 5 | 1 | |
| $P_{11}$ | 0 | 10 | 0 | |
| $P_{12}$ | 8 | 2 | 0 | $P_{15}$ |
| $P_{13}$ | 6 | 4 | 0 | $P_{15}$ |
| $P_{14}$ | 4 | 7 | 0 | $P_{10}$ |
| ***$P_{15}$*** | ***5*** | ***1*** | ***3*** | |
| $P_{16}$ | 7 | 3 | 0 | |

TABLE VI. TABLE 3.4 INPUT DATA USED BY REDUCER

| Mapper | Player | Points | Rebounds | Pi.C | Pi.DL |
|---|---|---|---|---|---|
| M1 | $P_2$ | 10 | 0 | 0 | |
| M1 | $P_5$ | 4 | 6 | 1 | |
| M1 | $P_7$ | 7 | 4 | 1 | |
| M1 | $P_8$ | 8 | 5 | 0 | $P_7$ |
| M1 | $P_9$ | 6 | 7 | 0 | $P_5$, |
| M2 | $P_{10}$ | 4 | 5 | 1 | |
| M2 | $P_{11}$ | 0 | 10 | 0 | |
| M2 | $P_{12}$ | 8 | 2 | 0 | $P_{15}$ |
| M2 | $P_{13}$ | 6 | 4 | 0 | $P_{15}$ |
| M2 | $P_{14}$ | 4 | 7 | 0 | $P_{10}$ |
| M2 | $P_{16}$ | 7 | 3 | 0 | |

TABLE VII. TABLE 3.6 DATA OUTPUT BY REDUCER

| *Mapper* | *Player* | *Points* | **Rebounds** | **Pi.C** | **Pi.DL(O)** | **Pi.DL(N)** |
|---|---|---|---|---|---|---|
| ***M1*** | $P_2$ | 10 | 0 | 0 | | |
| ~~***M1***~~ | ~~$P_5$~~ | ~~4~~ | ~~6~~ | ~~***2***~~ | | ~~**P10**~~ |
| ***M1*** | $P_7$ | 7 | 4 | 1 | | **P13, P16** |
| ***M1*** | $P_8$ | 8 | 5 | 0 | $P_7$ | **P12, P13, P16** |
| ***M1*** | $P_9$ | 6 | 7 | 0 | $P_5$ | **P13** |
| ~~***M2***~~ | ~~$P_{10}$~~ | ~~4~~ | ~~5~~ | ~~***1***~~ | | |
| ***M2*** | $P_{11}$ | 0 | 10 | 0 | | |
| ***M2*** | $P_{12}$ | 8 | 2 | *1* | $\mathbf{P_{15}}$ | |
| ~~***M2***~~ | ~~$P_{13}$~~ | ~~6~~ | ~~4~~ | ~~***3***~~ | ~~$\mathbf{P_{15}}$~~ | |
| ***M2*** | $P_{14}$ | 4 | 7 | 1 | $\mathbf{P_{10}}$, | ***P5*** |
| ~~***M2***~~ | ~~$P_{16}$~~ | ~~7~~ | ~~3~~ | ~~***2***~~ | | |

## IV. EXPERIMENT RESULTS

The algorithms were implemented in Java 1.6. All experiments were executed on Hadoop 1.2.1 using a cluster of five commodity machines. Four of the machines use an Intel Core2 Duo E8400 3GHz processor with 4GB RAM. The last machine uses an Intel Core2 Duo E8400 3GHz processor and 2GB RAM. The machines were connected by a 100Mbps LAN.

For dataset D, we produced synthetic datasets and changed various sizes and attributes. The types of data distribution included independent data distribution, the correlated data distribution, and anti-correlated data distribution, all of which are commonly used in skyline queries. The parameters and ranges are summarized in Table 4.1.

Table 4.1 Configuration parameters

| Parameter | **values** |
|---|---|
| Number of Point per Group (K) | 2, 3, 4, 5 |
| Data size (\|D\|) | 100, 200, 300, 400, 500 |
| Attribute (m) | 2, 3, 4, 5 |
| Aggregate function (*F*) | SUM |

The performance of the three algorithms is compared in Section 3. We executed these methods use the SUM function. In most of the experiments, we measured the runtime of the algorithm, the number of groups, and the number of group skylines.

1) *Scalability with respect to K*

In this experiment we studied the effect of the number of points per group. Figures 4.1, 4.2 and 4.3 are used to plot the execution time against the number of points per group, from 3 to 5 for anti-correlated, independent and correlated datasets. The size of the dataset was fixed at 200 (i.e., the number of candidate sets is between $1.3*10^6$ and $2.5*10^9$). Note that the execution time of this experiment is in logarithmic scale for independent and correlated datasets.

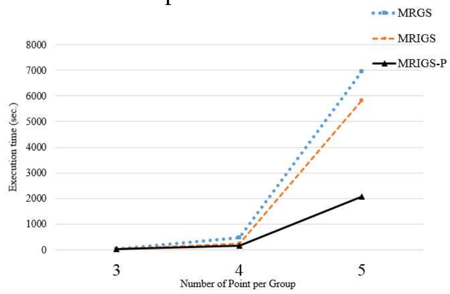


Fig. 4.1 Scalability with respect to k: anti-correlated

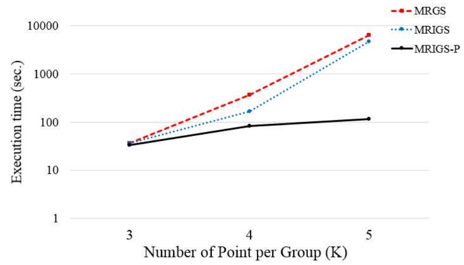


Fig. 4.2 Scalability with respect to k: correlated

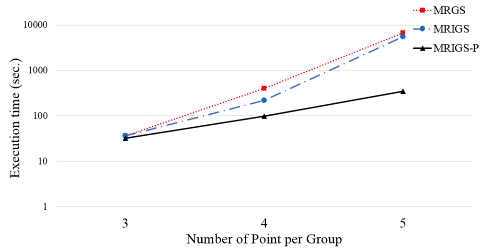


Fig. 4.2 Scalability with respect to k: independent

In all three data distributions, we found that MRIGS and MRIGS-P outperform MRGS, even when K is high. When K is 3, MRGS and MRIGS exhibit similar performance. Because the number of generated candidates is not very large, the execution times of the two algorithms are similar. When K is greater than 4, the number of candidates in each set was shown to grow exponentially. Both algorithms produce the same number of candidates; however, the execution time of MRIGS is significantly lower than that of MRGS, due to the effect of load imbalance in MRGS. MRIGS-P produced fewer candidates than either methods due to its use of pruning. In every case, MRIGS-P outperformed MRIGS and MRGS.

Figures 4.4, 4.5, and Fig. 4.6 plot the number of candidate sets against the points per group from 3 to 5 for anti-correlated, independent, and correlated datasets. Candidate-O represents MRGS and MRIGS generated candidate sets and Candidate-P represents MRIGS-P generated candidate sets. Figures 4.4, 4.5, and Fig. 4.6 clearly shows that MRIGS-P generates fewer candidates than does MRIGS. The number of candidate sets generated is proportional to the execution time. Similar results can be seen in the other two figures.

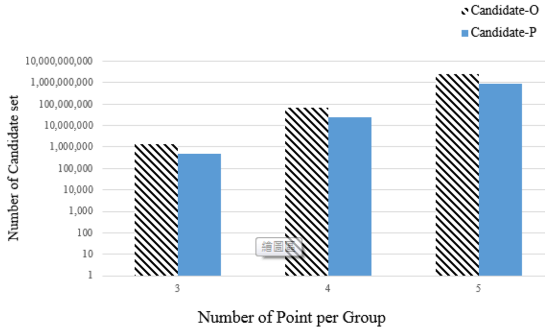


Fig. 4.4 Number of generated groups with respect to k: anti-correlated

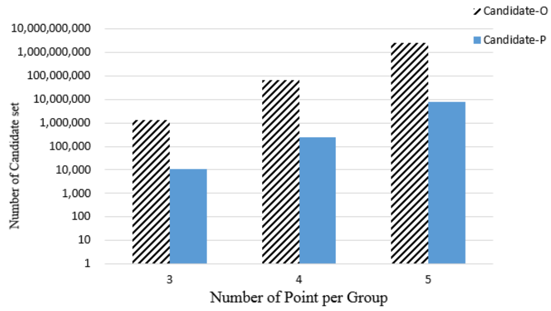


Fig. 4.4 Number of generated groups with respect to k: correlated

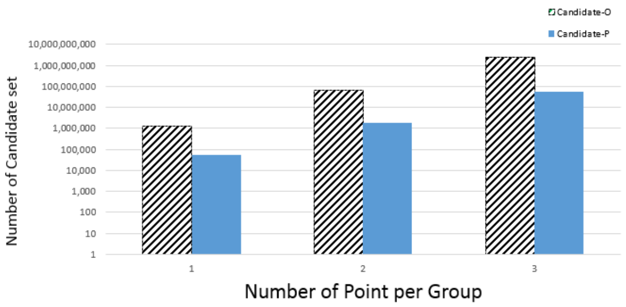


Fig. 4.4 Number of generated groups with respect to k: independent

*2) Scalability with respect to size of dataset (|D|)*

In this experiment, we examined the effects of the size of the dataset. k was fixed at 3 for the independent and correlated datasets. In the anti-correlated dataset, k was set at 2. Figures 4.7, 4.8, and 4.9 plot the execution time against the size of the dataset from 100 to 500 for anti-correlated, independent, and correlated datasets, respectively. Note that the results of MRGS is not shown in this or the following experiments because its performance does not exceed that of MRIGS.

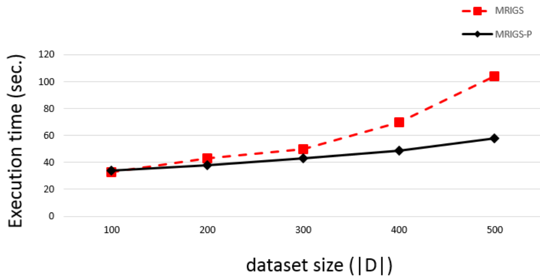


Fig. 4.7 Scalability with respect to dataset size: anti-correlated

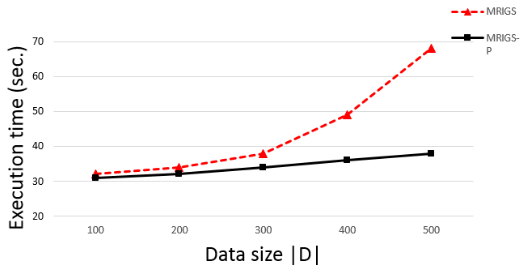


Fig. 4.8 Scalability with respect to dataset size: correlated

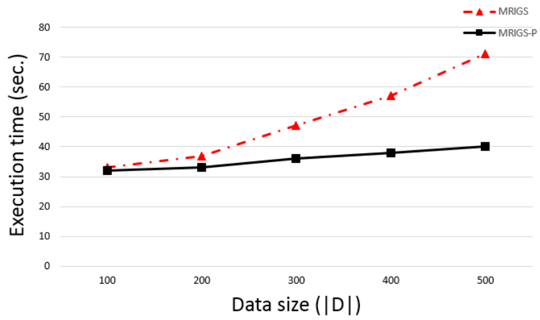


Fig. 4.9 Scalability with respect to dataset size: independent

In all cases, the performance of MRIGS-P was superior to that of MRIGS, except D, when it was equal to 100 and produced a smaller number of candidate sets, thereby reducing execution time. MRIGS-P needs to pre-process the dataset (i.e., run k-prune). In cases without a great deal of data, the effects of k-pruning are not obvious.

*3) Scalability with respect to number of attributes (m)*

In this experiment we studied the effect of the number of attributes. k was fixed at 4 and data size was fixed at 400. Figures 4.10, 4.11 and 4.12 plot the execution time against the number of attributes from 2 to 5 for anti-correlated, independent and correlated datasets.

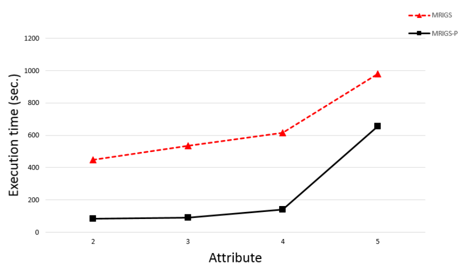


Fig. 4.10 Scalability with respect to number of attributes: anti-

correlated

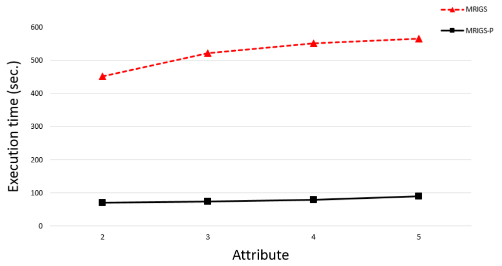


Fig. 4.11 Scalability with respect to number of attributes: correlated

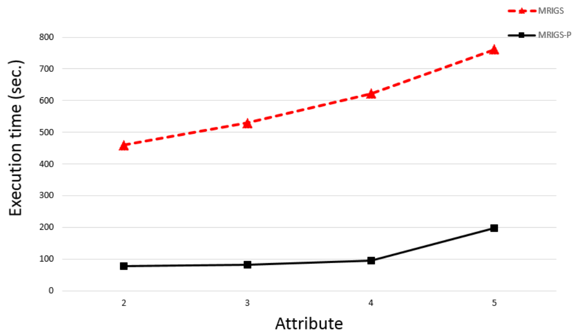


Fig. 4.12 Scalability with respect to number of attributes: independent

Basically, the time required to process queries increases with an increase in dimensionality. As a result, the increase in execution time displayed in these figures, particularly in the anti-correlated dataset, was more pronounced. This can be attributed to the fact that any increase in the dimensionality of data requires that the algorithm spend more time on the group skyline process.

## I. Conclusions

In this paper, we propose three novel algorithms, namely MRGS, MRIGS, and MRIGS-P, for parallelizing group skyline computation using a MapReduce framework. Our aim was to enhance input-pruning in the MapReduce environment. C***ascaded-pruning*** enables the removal of a large number of unnecessary points in order to reduce the number of candidate sets that are generated. Our experiment results show that MRIGS-P outperforms MRIGS and MRGS in all performance metrics.

In the future, we plan to further improve the performance by improving the second phase, which at present is limited to a single reducer. Constrained group skylines are another interesting topic worthy of further study.

## References

[1] Bartolini, I., P. Ciaccia, and M. Patella. "SaLSa: computing the skyline without scanning the whole sky," *Proceedings of the 15th ACM international conference on Information and knowledge management.* pp. 405-414, November 2006.

[2] Borzsony, S., D. Kossmann, and K. Stocker. "The skyline operator," *Proceedings of 17th International Conference on Data Engineering*, pp. 421–430, April 2001.

[3] Chaudhuri, S., N. Dalvi, and R. Kaushik. "Robust Cardinality and Cost Estimation for Skyline Operator," *Proceedings of the 22nd International Conference on Data Engineering*, pp. 1-10, April 2006.

[4] Chen, L., K. Hwang, and J. Wu. "MapReduce skyline query processing with a new angular partitioning approach," *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pp.2262-2270, May 2012

[5] Chomicki, J., et al. "Skyline with presorting," *Proceedings of the 19th International Conference on Data Engineering*, pp.717-719, March 2003.

[6] Chung, Y.-C., I.-F. Su, and C. Lee, "Efficient computation of combinatorial skyline queries," *Information Systems*, Vol. 38, NO. 3, pp. 369-387, May 2013.

[7] Dean, J. and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, Vol. 51, NO. 1, pp. 107-113, January 2008.

[8] Im, H. and S. Park, "Group skyline computation," *Information Sciences*, Vol. 188, pp. 151-169, April 2012.

[9] Kim, Y. and K. Shim. "Parallel top-k similarity join algorithms using MapReduce," *IEEE 28th International Conference on Data Engineering*, pp. 510-521, April, 2012.

[10] Kolb, L., Z. Sehili, and E. Rahm, "Iterative Computation of Connected Graph Components with MapReduce," *Datenbank-Spektrum*, Vol. 14, NO. 2, pp. 107-117, july 2014.

[11] Kolb, L., A. Thor, and E. Rahm, "Parallel sorted neighborhood blocking with mapreduce," *Datenbanken und Informationssysteme (DBIS)*, pp.45-64, 2011

[12] Kolb, L., A. Thor, and E. Rahm. "Load balancing for mapreduce-based entity resolution," *IEEE 28th International Conference on Data Engineering*,.pp. 618-629, April 2012.

[13] Kossmann, D., F. Ramsak, and S. Rost. "Shooting stars in the sky: An online algorithm for skyline queries," *Proceedings of the 28th international conference on Very Large Data Bases.* pp. 275-286, August 2002.

[14] Li, F., et al., "Distributed data management using MapReduce," *ACM Computing Surveys*, Vol. 46, No.3, pp. 1-42, January 2014.

[15] Mullesgaard, K., et al. "Efficient Skyline Computation in MapReduce," *Proceedings of the 17th International Conference on Extending Database Technology*, pp. 37-48, March 2014.

[16] Papadias, D., et al. "An optimal and progressive algorithm for skyline queries," *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp.467-478, june 2003.

[17] Papadias, D., et al., "Progressive skyline computation in database systems," *ACM Transactions on Database Systems*, Vol. 30, NO. 1, pp.41-82, March 2005.

[18] Siddique, M.A., H. Tian, and Y. Morimoto. "Distributed Skyline Computation of Vertically Splitted Databases by Using MapReduce," *Proceedings of 19th International Conference on Database Systems for Advanced Applications*, pp. 33-45, April 2014.

[19] Tan, K.-L., P.-K. Eng, and B.C. Ooi. "Efficient progressive skyline computation," *Proceedings of 27th International Conference on Very Large Data Bases*, pp. 301-310, September 2001.

[20] Thusoo, A., et al. "Hive - a petabyte scale data warehouse using hadoop," *Proceedings of the 26th International Conference on Data Engineering*, pp. 996-1005, March 2010.

[21] Zhang, B., S. Zhou, and J. Guan, "Adapting skyline computation to the mapreduce framework: Algorithms and experiments," *Proceedings of the 16th International Conference on Database Systems for Adanced Applications*, pp. 403-414, April 2011.

[22] Zhang, N., et al., "On skyline groups," *IEEE Transactions on Knowledge and Data Engineering*, Vol.26, NO. 4, pp. 942-956, January 2014.