Simultaneous Processing of Multi-Skyline Queries with MapReduce

Junsu KIM†a, Student Member, Kyong-Ha LEE††b, and Myoung-Ho KIM††c, Nonmembers

SUMMARY With rapid increase of the number of applications as well as the sizes of data, multi-query processing on the MapReduce framework has gained much attention. Meanwhile, there have been much interest in skyline query processing due to its power of multi-criteria decision making and analysis. Recently, there have been attempts to optimize multi-query processing in MapReduce. However, they are not appropriate to process multiple skyline queries efficiently and they also require modifications of the Hadoop internals. In this paper, we propose an efficient method for processing multi-skyline queries with MapReduce without any modification of the Hadoop internals. Through various experiments, we show that our approach outperforms previous studies by orders of magnitude.

key words: multi-query processing, skyline query, MapReduce framework

1. Introduction

Given a set of tuples with $d$ attributes, a skyline query returns tuples that are not dominated by any other tuples. Consider two queries $q_1$ and $q_2$ that finds items from the list shown in Fig. 1 (a) that satisfy their selection conditions. $p_{q_1}^{A_j}$ means the query $q_1$’s selection condition on the attribute $A_j$.

- $q_1$, $p_{q_1}^{price}$: $30 \leq price < 70$, $p_{q_1}^{cate}$: category="blouse"
- $q_2$, $p_{q_2}^{price}$: $40 \leq price < 80$, $p_{q_2}^{cate}$: category="blouse"

Gray-colored circles in Fig. 1 (b) show the items satisfying the conditions of $q_1$. Among these items, it is clear that users prefer items of high review scores together with low prices.

In the figure, for example, there seems to be no proper reason to choose items $e$ and $i$ rather than $f$ that has better review scores as well as better prices. The same argument applies to items $a$ and $b$. Skyline is useful to retrieve only preferable items such as $b$ and $f$ that satisfy multi-criteria.

Meanwhile, multi-query processing in MapReduce (MR) has attracted great attention since the data size and the number of queries in many applications increase dramatically. Online retailers that serve a large number of customers, e.g., Amazon and Ebay, try to provide personalized daily report services. The report of each user is made based on his/her preferences (or purchasing patterns) e.g., ranges of prices, ranges of review scores, categories, etc.

Skyline queries are useful to support this kind of services because they retrieve only the interesting items satisfying various user preferences. As the number of users rapidly grows, the importance of efficiently processing a large number of personalized skyline queries will also increase.

There have been some studies on processing skyline queries in the MR framework [1]–[4]. Most of them, however, focused on processing of a single skyline query. Recently, some studies on multi-query optimization in MR have been made [5], [6]. They proposed techniques that enabled systems to reduce redundancies between multiple queries. However, these studies show limitations to process skyline queries efficiently and also require modifications of the Hadoop internals. In this paper, we propose an efficient method to process multiple skyline queries with MR by reducing redundancies without modifying the Hadoop internals.

2. Preliminaries

The following definitions can be found in the literature [3].

**Definition 1** (Dominance relationship). Given a set of tuples with $d$ attributes, a tuple $t_1$ dominates $t_2$, if $t_1$ is no worse than $t_2$ in any other attributes and $t_1$ is better than $t_2$ in at least one attribute.

**Definition 2** (Skyline). Given a set of tuples $T$, a skyline is a set of tuples that are not dominated by any other tuples.

The cost of a MR job consists of I/O cost (input scan, write/read/sort map output), network cost (shuffling) and computational cost (map/reduce methods) [5]. To process multiple skyline queries efficiently, it is important to reduce redundant data processing which increases the above costs.
To compute skylines in our method, we choose a straightforward algorithm, BNL [3]. However, any of skyline algorithms can be easily adopted.

3. Baseline Methods

We introduce two baseline methods for processing multiple skyline queries without modifying the Hadoop internals. We will compare the performance of the proposed method with these methods. This is because, as far as we are aware of, there is no previous work on the problem of multiple skyline queries on the MR framework.

Naïve approach. This approach processes multiple queries independently. A MR job is executed for each query, where the common input file is scanned for each job. In the map phase of each job, the common input file is scanned to generate a map output for the corresponding query. In the reduce phase of each job, the map output is distributed to the reducer and used for skyline calculation.

SH-scan approach. This approach provides sharing of input scan for multiple queries. In the map phase, the common input file is scanned once to generate map outputs for multiple queries. In the reduce phase, if the corresponding query of the map output is $q_i$, we distribute that map output to the reducer of $q_i$. Finally, each reducer calculates a skyline by using the given map output.

Figure 2 shows examples of baseline methods when they process $q_1$ and $q_2$ simultaneously. In Fig. 2, $T$ is an input file. $M_{q_1}$ and $M_{q_2}$ are map outputs, and $S_{q_1}$ and $S_{q_2}$ are skylines for $q_1$ and $q_2$, respectively. Figure 2(a) shows Naïve approach to process two queries independently, where the same input file $T$ is scanned for each query separately. In addition to two separate scans of $T$, the same map outputs for the price range [$40, $70) which is the intersection of two ranges [$30, $70) and [$40, $80), are separately used to reducers of both queries. That is, the overlapped range of map outputs are redundantly generated and used for skyline calculations. Figure 2(b) shows SH-scan approach which avoids redundant scans of the same input file. However, the map output for the overlapped price range [$40, $70) are still redundantly generated and used for skyline calculations.

4. Proposed Method

4.1 Basic Concept

Unlike the previous approaches, our method, called SH-adv avoids the redundant generation of map outputs and skyline calculations as well as redundant input scans. To avoid those redundancies, we divide data space of queries into disjoint fragments. The definition of a fragment is as follows:

Definition 3 (fragment). A fragment $f$ is a $k$-dimensional space defined by $k$ number of 1-dimensional value ranges. Each 1-dimensional value range is on each attribute used for one of the selection conditions of a query. $f_{\text{adj}}$ represents the value range of a fragment $f$ on the attribute $A_j$.

Algorithm 1 shows how to generate fragments from a set of input queries $Q$. For each iteration, we create a new fragment $f_{\text{new}}$ from a query in $Q$ (line 3). We find the fragments whose value ranges are overlapped with the $f_{\text{new}}$'s one, and then collect them to $\text{OVF}$ which means the set of overlapped fragments with $f_{\text{new}}$ (lines 4–7). We then divide space by using divideSpace algorithm with $\text{OVF} \cup \{f_{\text{new}}\}$ as input (line 8). In this algorithm, we separately collect value ranges of input fragments for each attribute. For the value ranges of the attribute $A_j$, we extract lower and upper bounds, and insert those bounds into the sorted set $E^{A_j}$ which maintains its elements in ascending order (lines 11–12). We then make 1-dimensional fragments by using two consecutive edges in $E^{A_j}$ and insert the fragments into $M^{A_j}$ (lines 13–15). Finally, we have $k$-dimensional fragments $F$ resulting from a cartesian product of all $M^{A_j}$ (line 16). Note that all the fragments in $F$ resulting from Algorithm 1 are disjoint among one another.

Given a set of fragment $F$ which are generated from

![Algorithm 1: Fragment generation](image-url)
a set of queries $Q$, the following lemma is satisfied. $C_u(T)$ denotes a set of tuples in $T$ which satisfies condition $u$.

**Lemma 1.** For two fragments $f_i$ and $f_j$, $C_f(T) \cap C_f(T) = \emptyset$

Given a set of fragment $F_q$ which consists of the fragments that are included in $q$, the following lemma is satisfied.

**Lemma 2.** $\bigcup_{f \in F_q} C_f(T)$ is equal to $C_q(T)$

The proof of above lemmas are trivial, and hence is omitted.

Figure 3 shows an example of fragments which are generated from input queries $q_1$ and $q_2$. In the example, we have three disjoint fragments $f_1$, $f_2$ and $f_3$ whose price ranges are [$30, $40), [$40, $70) and [$70, $80), respectively.

4.2 Procedure of Proposed Method

The key idea of SH-adv approach is to calculate skylines in fragments prior to calculate skylines of queries. SH-adv approach consists of three steps below:

**Preprocessing.** In this step, we make fragments from input queries by using Algorithm 1 and store two information; a set of fragments $F$ and a mapping table $FMAP$ to indicate that which fragment belongs to which queries.

**Fragment-level processing.** In this step, we produce skyline per fragment from the input file. In the map phase, we start with loading the set of fragments $F$. For each input tuple, we find the corresponding fragment whose data space includes that tuple. We then generate the map output by using an input tuple as a value and an ID of the corresponding fragment as a key. In the reduce phase, we computer a skyline for each key, fragment ID, and produce output of the form (fragment ID, skylines). We call the result skylines of this step as fragment-skylines.

**Query-level processing.** In this step, we produce skyline per query by using the output of the previous step. In the map phase, we start with loading the mapping table $FMAP$. For each fragment in the input tuple of the form (fragment ID, skylines), we find queries which include that fragment by looking up the $FMAP$. For each query in the results of lookup, we generate the map output by using ID of that query as a key and skylines as a value. In the reduce phase, we compute a skyline for each query by using the fragment-skylines which come from the associated fragments. We call the result skylines of this step as query-skylines.

Figure 4 shows an example of our SH-adv approach when it processes $q_1$ and $q_2$ simultaneously. In the pre-processing step, we generate a set of fragments $F$ and a mapping table $FMAP$. In the first MR job, fragment-level processing, we share scans of input file $T$ and generate map outputs $M_{f_1}$, $M_{f_2}$ and $M_{f_3}$ for fragments $f_1$, $f_2$ and $f_3$, respectively. We then calculate fragment-skylines $S_{f_1}$, $S_{f_2}$ and $S_{f_3}$ by using $M_{f_1}$, $M_{f_2}$ and $M_{f_3}$ as inputs, respectively. In the second MR job, query-level processing, we generate map outputs by using fragment-skylines as input. Since the fragments $f_1$ and $f_3$ are included in $q_1$ and $q_2$, respectively, we generate a part of map output $M_{q_1}$ from $S_{f_3}$ and a part of map output $M_{q_2}$ from $S_{f_5}$. We generate the same part of both $M_1$ and $M_2$ from $S_{f_3}$ because $f_2$ is commonly included in both $q_1$ and $q_2$. Finally, skylines of queries $S_{q_1}$ and $S_{q_2}$ are produced by using $M_{q_1}$ and $M_{q_2}$ as inputs, respectively.

Since the fragments resulted from Algorithm 1 are disjoint by Lemma 1, all the redundant tuples in the map outputs of the first MR job are totally eliminated. In the second MR job, redundant tuples could be generated, but the size of them is quite small because they have made from only the tuples in fragment-skylines. When we have tuples whose size is $R$ for the intersected range of two queries, baseline methods generate redundant tuples with the size of $R$, whereas the proposed method decreases the size of them to $\Theta(\log^d R)$, where $d$ is the number of dimensions [7]. Surprisingly, as the number of queries increases, the proposed method produces fewer map outputs than the baseline methods even though it is composed of two MR jobs.

Also, we benefit from sharing of calculation results of fragment-level skylines. In the example of Fig. 4, the skyline within fragment $f_2$ is shared to calculate skylines of queries $S_{q_1}$ and $S_{q_2}$. The sharing of skyline calculation is based on Lemma 2 and Lemma 3 [4]. In Lemma 3, $SKY(T)$ means a skyline of a tuple set $T$.

5. Experiments

We have evaluated our method through various experiments. All algorithms have been implemented with Hadoop V.1.2.1
on CentOS 7.0 and executed on a cluster of 11 machines.

Each node is equipped with Intel i5-Haswell 3.5GHz CPU and 16GB memory. We use synthetic datasets and queries. Each tuple in a dataset consists of ten numerical attributes. All values of attributes are randomly generated except the key attribute. The queries have constraints between value_a and value_b, which were randomly selected such that value_a < value_b in each dimension. The default parameter values are as follows: the dimensions of data is 2, the size of data is 40GB (240M tuples), the number of concurrent queries is 40, and the cluster size is 6 nodes. We have compared the proposed method with two baseline methods.

Figure 5 presents the performance gain of the proposed method over Naïve and SH-scan approaches. To test scalability, we generate various size datasets. When data size is 80GB, SH-adv outperforms SH-scan by up to 236%. Note that the execution time increases at a lower rate in the proposed method than other methods as the data size increases. Figure 6 presents the performance of SH-adv in terms of the number of processed queries. We obtain a significant improvement as the number of queries increases. As shown, SH-adv reduces the execution time by up to 1,410%. Furthermore, SH-adv achieves an order of magnitude performance gain with increasing the number of queries. Figure 7 shows the performance improvement of SH-adv compared with Naïve and SH-scan as the number of nodes is varied from 4 to 10. We evaluate the experiment with 80GB data and 80 queries. In the experiment, SH-adv outperforms clearly SH-scan by up to 995%. The substantial performance gain of SH-adv is achieved by sharing the input scans and using the fragments which enable the reduction of redundant generation of map outputs and skyline calculation.

Figures 8–10 present the execution time when the dimensions of data increases from 2 to 5. Since the data distribution affects the size of skylines, we evaluate SH-adv in three different distributions: independent, correlated and anti-correlated. As we expected, SH-adv outperforms SH-scan in all data distributions. The reason of performance improvement is not only the reduction of redundantly generated map outputs, but also the elimination of redundant skyline calculation. Specifically, our method exhibits the steady performance in the first two distribution. In anti-correlated distribution, the execution time of the proposed method slightly increases as the number of dimensions increases. The reason is that the size of fragment-skylines increases as the number of dimension increases in that distribution. However, in the fragment-skylines, there exist tuples which are not included in query-skylines. Thus, we will study efficient methods to thin out those tuples in the future.

6. Conclusion

In this paper, we proposed an efficient method to simultaneously process multiple skyline queries without modifications of the Hadoop internals. Our method was successful to process multiple skyline queries efficiently by considerably reducing redundancies in skyline query processing. The proposed method is composed of two consecutive MR jobs, but outperforms baseline approaches significantly by reducing the redundant generation of map outputs and skyline calculations as well as redundant input scans.
Acknowledgements

This work was partly supported by the Bio-Synergy Research Project (2013M3A9C4 078137) of the MSIP, Korea, through the NRF, and by the MSIP, Korea under the ITRC support program (IITP-2017-2013-0-00881) supervised by the IITP, and by KISTI.

References


