Extracting k Most Important Groups from Data Efficiently

Man Lung Yiu [a], Nikos Mamoulis [b], Vagelis Hristidis [c]

[a] *Department of Computer Science, Aalborg University, DK-9220 Aalborg, Denmark*

[b] *Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong*

[c] *School of Computing and Information Sciences, Florida International University, Miami, FL 33199, USA*

Abstract

We study an important data analysis operator, which extracts the k most important groups from data (i.e., the k groups with the highest aggregate values). In a data warehousing context, an example of the above query is "find the 10 combinations of product-type and month with the largest sum of sales". The problem is challenging as the potential number of groups can be much larger than the memory capacity. We propose on-demand methods for efficient top-k groups processing, under limited memory size. In particular, we design top-k groups retrieval techniques for three representative scenarios as follows. For the scenario with data physically ordered by measure, we propose the write-optimized multi-pass sorted access algorithm (WMSA), that exploits available memory for efficient top-k groups computation. Regarding the scenario with unordered data, we develop the recursive hash algorithm (RHA), which applies hashing with early aggregation, coupled with branch-and-bound techniques and derivation heuristics for tight score bounds of hash partitions. Next, we design the clustered groups algorithm (CGA), which accelerates top-k groups processing for the case where data is clustered by a subset of group-by attributes. Extensive experiments with real and synthetic datasets demonstrate the applicability and efficiency of the proposed algorithms.

Keywords: 28.3 Optimization and Performance

Contact Author: Man Lung Yiu

1. Introduction

Aggregate queries summarize information in a database, by dividing tuples into groups, where some target attributes agree on their values, and applying an aggregate function (e.g., `COUNT`, `SUM`, `MAX`, etc.) to each group. As an example, consider a data warehouse [1] that stores detailed information about the transactions of a company in a huge fact table [15] with schema *Sales*(*TID*, *ProdID*, *StoreID*, *Time*, *Quantity*, *Price*). Assume the following online analytical processing (OLAP) query:

```
SELECT ProdID, StoreID, SUM(Quantity)
FROM Sales
GROUP BY ProdID, StoreID
```

In practice, the number of product/store combinations can be large and the results could overwhelm the user. Besides, the aim of the original analysis should be to identify the most important groups, rather than viewing the distribution of aggregates in all product/store combinations. We could express 'importance' by a HAVING clause that selects only those groups whose aggregate value exceeds a threshold t. This class of aggregate queries were identified in [7] and named *iceberg* queries.

Nevertheless, from the user's perspective, it is hard to specify an appropriate value for t so the iceberg query is far from being user-friendly. When t is too large, no results may be returned at all. On the other hand, for small values of t, numerous groups are retrieved eventually. A better way for capturing the group interestingness is to consider a *ranking* of the groups based on their aggregate values and select the k groups in this order. This way, the user is able to control the result size naturally. As an example, a top-k groups query can be expressed in SQL, by adding to the statement above the following lines:

```
ORDER BY SUM(Quantity) DESC
STOP AFTER k
```

Apart from finding heavy groups in data warehouses, the top-k groups query also finds application in other data mining tasks. For example, the problem of extracting top-k frequent patterns [11] (which is a variant of frequent itemset mining [2]) can be viewed as a top-k groups query. Here, all combinations of items are candidate groups and the objective is to find the ones with the largest `COUNT` in a transactional database. Another application (from information retrieval) is to retrieve the Web documents with the largest number of incoming (or outgoing) links. Finally, as demonstrated in our experiments, top-k queries can be used to identify pairs of network ports with high volume of information flow, from traffic traces of TCP packets.

The evaluation of top-k groups queries could be facilitated by exploiting materialized views [12] over the base data. However, the selection of attributes in such queries could be ad-hoc. Pre-computation, materialization, and maintenance of group-bys for all possible combinations of attributes incur prohibitively high cost, especially for cases where the data are updated frequently. Therefore, in this paper, we study *on-demand* processing of top-k groups query on a very large base table. Specifically, we focus on the class of distributive aggregate functions

(say, COUNT, SUM, MAX, MIN), and ignore holistic aggregate functions (e.g., MEDIAN).

A straightforward solution is to keep a counter for each group in memory and update the corresponding counts while scanning the base table. This method only requires one pass of the data. However, it may be infeasible to keep a counter for each group in memory even though today's machines have large memory size. For example, it is not uncommon to have attributes with domain size in the order of 1000. For a query with four group-by attributes on a peta-byte warehouse, the number of required counters is $(1000)^4$, which translates to 4 tera-bytes (assuming 4 bytes per counter).

The traditional method (by an RDBMS) for evaluating iceberg and top-k groups queries (using limited memory) is to compute the aggregate scores for all groups and select the qualifying ones according to the threshold t, or the ranking parameter k. This method (implemented by hashing or sorting with early aggregation) can be quite expensive, since the group-by operation may apply multiple passes over the data to compute the aggregates for *all* groups, while most of them are expected to be eventually pruned. Previous work on iceberg queries [7,17] employed sampling- and/or hash-based techniques to eliminate groups having small aggregates early and minimize the number of passes over the base data. The extension of these methods for top-k groups queries is not straightforward, since they rely on a fixed threshold t, which cannot be determined a priori.

Top-k groups retrieval has been studied in [20], albeit in a specialized context. First, the group-by attributes are regarded as ad-hoc ranges in continuous or spatial domains. Second, aggregate extensions of multidimensional indexes [16] were presumed on all relevant attributes of the query. [18] studies top-k groups queries in RDBMS in the presence of a striding index [13] for the group-by attributes. Existence of multidimensional indexes for all group-by attributes is a rather strong assumption, since the number of attributes in a relation (or combination of relations) could be arbitrary and the query may involve any subset of them. There is also a number of theoretical studies on one-pass approximate top-k groups retrieval from data streams, given limited memory (e.g., see [5,22]). Nevertheless, such techniques are not directly applicable to situations where exact retrieval of groups and their aggregates is essential.

The goal of this paper is to provide solutions for on-demand and exact top-k groups extraction, under bounded memory size. The key contribution of this paper is a set of algorithms to efficiently handle top-k groups queries in the realistic scenario where neither specialized multi-dimensional indexes are available, nor approximation of results is acceptable. Specifically, we investigate three representative scenarios and develop comprehensive techniques for them:

- For the case where tuples are physically ordered by measure, we propose the write-optimized multi-pass sorted access algorithm (WMSA), that exploits available memory to compute top-k groups efficiently.
- Under the scenario of unordered data, we study the Recursive Hashing Algorithm (RHA), coupled with branch-and-bound techniques and derivation heuristics for tight aggregation bounds of partitions.
- For the special case where the tuples to be aggregated are clustered according to a subset of group-by attributes, we develop the Clustered Groups Algorithm (CGA), which accelerates

top-k groups processing.

Our algorithms are cross-compared with traditional RDBMS approaches for on-demand top-k groups retrieval, using real and synthetic data.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 discusses the different problem settings for a top-k groups query. Section 4 presents top-k groups processing algorithms for the case where the input is physically ordered according to the aggregated measure. In Section 5, we propose techniques for the case of unordered input. Section 6 discusses the case, where the input is clustered according to the values of some group-by attributes. Experimental evaluation of all above techniques are presented in the respective sections. Finally, Section 7 concludes the paper.

2. Related Work

Our work is related to top-k aggregates retrieval, iceberg queries, and top-k algorithms for middleware. In this section, we review past research on these problems and discuss their relationship to our work.

2.1. *Top-k Aggregate Queries in OLAP*

Mamoulis et al. [20] proposed an algorithm for computing top-k groups queries from a data warehouse, assuming that the fact table is indexed by an aggregate R–tree [16]. This index is an extension of the popular R–tree [10], where measures (e.g., sales quantity) are indexed based on their dimensional values at the finest granularity (e.g., prodID, storeID, timestamp, etc). Non-leaf entries of the tree are augmented with aggregates that summarize all measures in the sub-tree pointed by it. Group targets are attributes at some hierarchical level of the dimensions (e.g., product-type, store-city, month, etc.) and each target is presumed to be a continuous range in the corresponding dimensional domain (e.g., product-type 'toy' consists of product-ids 100 to 250). The main idea behind the top-k groups retrieval method is to traverse the tree and maintain upper and lower bounds for the aggregate of each group, based on (i) the visited nodes of the tree and (ii) the overlap (in the multidimensional space) of remaining sub-trees to be visited with the cell formed by the Cartesian product of the sub-domains that define the group. Traversal of the tree is based on a heuristic that chooses the next node to visit, based on its contribution to the best groups found so far.

Although this method is effective for a particular class of OLAP queries, its applicability is restricted by two hypotheses: (i) the set of dimensions have to be indexed by an aggregate multidimensional index, and (ii) the group targets must be formed by continuous sub-domains of the dimensions. The first hypothesis requires numerous multidimensional indexes (one for each combination of dimensions), which are inefficient or infeasible to maintain if the total number of dimensions is large (especially true for non-OLAP queries that may involve arbitrary attributes from input tables). An alternative choice is to employ a single multidimensional

index consisting all the dimensions; however, such an index renders inefficient query processing as the attributes irrelevant for the query are also stored in the index. The second hypothesis restricts the targets to be related to some hierarchy of a base (detailed) attribute and, in addition to that, each target should be a decomposition for a total value ordering of a base attribute.

Recently, [18] proposed an RDBMS operator for top-k groups queries that schedules the accesses of groups and their tuples, in order to minimize the amount of accessed information before the query result is guaranteed to be found. The proposed methodology has practical limitations because it is based on the following strong assumptions. First, it assumes that the algorithm has control on the order by which the groups will be examined and on the order by which the tuples within a group will be visited. However, the preprocessing cost required to achieve this control might as well exceed the cost of the query itself, especially for queries with ad-hoc sets of grouped attributes. In addition, it is assumed that the size of each group and the upper bound for the aggregate expression are known. If the set of groups is ad-hoc, computing the size of each group has similar cost to computing the aggregate of the group itself.

[19] propose methods for processing top-k *range* queries in OLAP cubes. Given an arbitrary query range, the problem is to find the top-k measures in this range. This query is a generalization of *max* range queries (i.e., for $k = 1$ it is a *max* query). The data cube is partitioned into sufficiently small cells and the top-k aggregate values in each partition are pre-computed. These values can then be used to compute the top-k results in query regions that cover multiple cells. Top-k range queries are essentially different from top-k groups queries, since the latter deal with the retrieval of top-k aggregated values of *groups* (as opposed to top-k measures) in the whole space (as opposed to a particular range).

2.2. *Iceberg Queries*

Iceberg queries were first addressed in [7]. A typical query optimizer [9] would compute the aggregates for *all* groups and then return the ones whose score exceeds the query threshold t. [7] present several methods based on sampling and hashing with output-sensitive cost for iceberg queries. They were later extended in [3] for selecting exceptional groups in a whole hierarchy of OLAP cubes. These methods aim at avoiding useless aggregations for groups, which disqualify the threshold condition. The main idea is to divide the data to be aggregated into buckets based on a hash function that applies on the targets. Buckets with smaller count than the threshold are immediately pruned. Sampling is used to identify early potential targets that could end-up in the result, which are treated separately.

In [17], these methods are compared with sorting and hashing with early aggregation [9]. The results show that the methods of [7] prevail for cases when both the skew in the scores of the candidate groups and the number of candidate groups are high; in other cases, applying either sorting or hashing to compute the scores of all groups and selecting the result in the final pass of the algorithms is the most appropriate choice. This makes sense, since the methods of [7] first identify heavy hitters and then disregard their tuples at later passes, thus shrinking the

data to be processed. However, if the scores between groups do not vary much, the attempts to shrink the data prove futile and a simple hash-based or sort-based method becomes better.

The extension of the techniques of [7] for rank-based retrieval of important groups is not straightforward, since the k-th group (the pruning threshold in this case) is not easy to compute. In this paper, we propose such an extension in addition to an optimized algorithm that extends hash-based aggregation for top-k groups retrieval. In addition, we consider the special case, where the query input is sorted or clustered based on some group-by attributes and propose a carefully optimized technique for this case.

2.3. *Top-k Algorithms for Middleware*

Middleware top-k algorithms have been proposed for combining different rankings of the same set of objects in order to return the k objects with the highest combined score according to an aggregate function. Assume for example that we wish to retrieve the restaurants in a city in decreasing order of their aggregate scores with respect to how cheap they are, their quality, and their closeness to our hotel. If three separate services can incrementally provide ranked lists of the restaurants based on their scores in each of the query components, the problem is to identify the k restaurants with the best combined (e.g., average) score. Fagin et al. [6] present a comprehensive analytical study of various methods for top-k aggregation of ranked inputs by monotone aggregate functions. They identify two types of accesses to the ranked lists; *sorted* accesses and *random* accesses. The sorted access operation iteratively reads objects and their scores sequentially in each ranked list, whereas a random access is a direct request for an object's score with respect to some feature. [6] suggests methods that are asymptotically optimal in the number of accesses they perform, for both cases depending on whether random accesses are allowed or not.

The *threshold algorithm* (TA) accesses sequentially the ranked inputs and, for each object seen there, it performs random accesses to the lists where the object has not been seen yet to compute its aggregate score. TA terminates when the k-th object found so far has higher aggregate score than the aggregate score of the last values seen at each input. The *no random accesses* (NRA) algorithm applies only sorted accesses to the inputs, until the best k objects seen so far have higher aggregate score than the score that any other object can reach. In [4], extensions of these methods for the case where only random accesses are possible for most inputs are presented. [14] shows how to adapt NRA for joins followed by ranking in relational databases. Finally, [21] optimized the NRA algorithm to a version that minimizes computational cost, accesses and required memory and applied this optimization for several variants of top-k search. In this paper, we study the adaptation of top-k algorithms to the retrieval of important groups, exploiting a potential physical order of the tuples based on their measure values.

3. Problem Settings

In this section, we define the problem of top-k groups queries for various cases and motivate the design of the proposed algorithms. Then, we discuss the setting used for empirically evaluating our algorithms.

3.1. *Problem Definition*

We consider a query with a set $\mathcal{G}$ of group-by attributes and an aggregate function $agg(\cdot)$ that applies on a single measure attribute v. The tuples to be aggregated may physically be stored in the disk or produced by underlying operators (e.g., selections). We assume that (i) the set of group-by attributes is ad-hoc and no multi-dimensional indexes exist on the tuples to be grouped, and (ii) the memory of the system is not large enough to hold a counter for each distinct group in the input. We develop methods that minimize the number of passes over the data, aiming at *exact* query evaluation. In particular, we explore various orderings of the query input that may apply in practice, as follows.

- The data are physically ordered or clustered with respect to the measure attribute v, i.e., supporting the sorted access operation on v. This case may apply if a clustered B^+–tree exists on v. As another scenario, consider the data to be aggregated distributed in different servers. Each server sorts its part (cheaply, if we assume that it is small) and then forwards it to a central server, which merges them and performs the aggregation. For these cases, we adapt the NRA algorithm [6] to minimize the number of accessed tuples before the top-k result is finalized (see Section 4).
- The order of the aggregated tuples is random. This is the most generic case, which we study in Section 5, by adapting algorithms for iceberg queries and generic hash-based aggregation.
- The data are clustered with respect to one or more group-by attributes. For stored data, this case may apply if a clustered index or a hash-file organization exists. In a data stream scenario, tuples could be produced ordered by a group-by attribute (e.g., time, location). We examine such cases in Section 6, by reducing the top-k groups problem to finding the top-k groups for each value (or for a set of values) of the clustered attribute, since the tuples are well-separated according to that attribute. We show that this case can be reduced to multiple, smaller problems. Roughly speaking, for each batch of tuples, having the same value (or set of values) for the clustered attribute, we only have to solve an unordered data subproblem.

For the ease of exposition, throughout the presentation, we consider data aggregated from a single table $\mathcal{T}$ with three columns; the tuple-id *tid*, the group-id *gid* and the value v to be aggregated. In practice, *gid* corresponds to a combination of group-by values and it is implemented as such in our evaluation tests. The domain of v is assumed to be the interval $[0, 1]$; nevertheless, the extension for arbitrary domain interval $[V_{min}, V_{max}]$ (derived from database statistics) is straightforward. Unless otherwise stated, we consider the `SUM` function for aggregation, which is the most common in OLAP queries; nonetheless our methods can be easily

adapted for other monotone distributive functions, e.g., `COUNT`, `MAX`, `MIN`. The symbols listed in Table 1 will be used in the discussions throughout the paper.

Table 1
List of symbols

Notation	Meaning
$\mathcal{T}$	the input table
N	the number of tuples (in $\mathcal{T}$)
G	the number of distinct groups (in $\mathcal{T}$)
k	the result size
ρ	the lower bound score of the best k-th group found so far
M	the number of memory pages
B	the number of tuples that a page can fit

3.2. *Experimental Setting*

To enhance the readability, the experimental evaluation of our proposed algorithms have been decomposed into their respective sections (Sections 4.4, 5.3, and 6.2). In the following, we discuss their common experimental setting.

We evaluate the efficiency of the proposed algorithms on a real dataset *dec_wrl_4* used in [23]. It is a traffic trace of TCP packets, containing 3.86M tuples with the following attributes: timestamp, source host, destination host, source TCP port, destination TCP port, and packet size. All host and port values are integers. Since the domain of timestamps is continuous, it is discretized to the granularity of one second. We consider the packet size as the measure attribute v for aggregation and run a top-k query ($Q1$) that computes the connections with the largest traffic at 1-second intervals:

```
SELECT Tstp,Sip,Dip,Sport,Dport,SUM(PktSz)
FROM dec_wrl_4
GROUP BY Tstp,Sip,Dip,Sport,Dport
ORDER BY SUM(PacketSize) DESC
STOP AFTER k
```

Our second query $Q10$ is the same as $Q1$, except that the timestamp is discretized at every 10 seconds. The number of distinct groups for $Q1$ and $Q10$ are 765K and 232K respectively. ⋆

We generated a set of synthetic datasets in order to test the performance of the algorithms for different data distributions and to verify their scalability. To obtain meaningful top-k results, the data generation is such that only few of the groups have high aggregate values compared to the rest. We experimented with different data sizes N (default: N=4 million tuples). The sizes of groups follow a Zipfian distribution (at a fixed skewness θ_s=0.5), such that the total number

⋆ Although this number of counters can easily fit in memories of modern computers, we simulate the case that they don't by assuming a smaller system buffer. Our results can easily be generalized for realistic cases where the distinct number of groups and the memory size are multiples of the sizes we used in our evaluation.

of groups is $N/4$. The measure values v in individual tuples also follow Zipfian distribution (at skewness $\theta_v = 1$).

We considered four distributive aggregate functions (i.e., `COUNT`, `SUM`, `MAX`, `MIN`). `COUNT` is a special case of `SUM` where all tuples have the same value, thus the results can be evaluated by all our algorithms. On the other hand, the query for `MIN` can be reduced into that of `MAX` as follows: (i) in the beginning, each tuple value v is mapped to the value $1 - v$ (assuming that their value domain is $[0, 1]$), (ii) the `MAX` function is applied to compute the results, and (iii) each result value t is mapped to $1 - t$ for obtaining the final results.

We implemented all tested algorithms in C++ and the experiments were executed on a PC with a Pentium 4 CPU of 2.3GHz. We used a default memory buffer whose size is 2% of the dataset. The page size is 4K bytes so that each page can accommodate $B = 200$ tuples. The default value of the result size k in the query is set to 16. In the experiments, we measure the *ratio of page accesses* of an algorithm, which is defined as the ratio of total disk page accesses (for both read/write operations) to the number of pages in the dataset.

4. Algorithms for Inputs Ordered by Measure

In this section, we propose techniques for top-k groups queries, applicable when the tuples are physically ordered based on the aggregated value v. In Section 4.1 we propose an algorithm that applies on a descending order of the tuples by v, assuming that the memory can fit a counter per group. The algorithm is extended in Sections 4.2 and 4.3 for cases of limited memory.

These algorithms enable early termination as long as the *lower bound* scores of the top-k groups found so far are guaranteed to be higher than the *upper bound* score of any other group. As a consequence, they retrieve only the top-k groups, without necessarily obtaining their *exact* scores.

4.1. *Preliminaries*

Our first algorithm, termed "sorted accesses" (SA) is an adaptation of the "no random accesses" (NRA) top-k technique, discussed in Section 2.3. It scans the input $\mathcal{T}$, while updating lower and upper aggregate scores for the groups seen so far. If, at some point, there exist k groups for which the lower bound of their aggregate is at least the upper bound of the remaining ones, the algorithm terminates. Figure 1 shows an exemplary query and the table $\mathcal{T}$, which we will use to describe SA.

Before presenting SA, we discuss the computation of the lower/upper bounds it utilizes while scanning the input, considering a `SUM` aggregate function. Initially, we assume that the maximum cardinality C_{max} of a group is known in advance (i.e., $C_{max} = 3$ in our example). Later, we relax this assumption. Let g_x be the group of tuples having x as gid and $\psi(g_x)$ be the set of accessed tuples in the group. A lower bound $lb(g_x)$ and an upper bound $ub(g_x)$ for the aggregate score of group g_x are given by:

```
SELECT gid, SUM(v)
FROM Tbl
GROUP BY gid
ORDER BY SUM(v) DESC
STOP AFTER k
```

(a) Query

tid	gid	v
8	2	0.70
9	2	0.69
5	4	0.50
3	5	0.41
11	1	0.40
4	4	0.39
2	5	0.33
1	5	0.13
6	3	0.12
7	3	0.11
10	2	0.10
12	1	0.05

(b) Table

Fig. 1. Running example

$$lb(g_x) = \sum_{t \in \psi(g_x)} t.v \tag{1}$$

$$ub(g_x) = lb(g_x) + v_{max} \cdot (C_{max} - |\psi(g_x)|) \tag{2}$$

where v_{max} represents the maximum value v of unseen tuples, e.g., the value of the latest seen tuple. These equations are applicable to the `COUNT` function as well, by fixing each value $t.v$ to 1. Regarding the `MAX` function, the lower and upper score bounds are defined as follows.

$$lb_{max}(g_x) = \begin{cases} 0 & \text{if } \psi(g_x) = \varnothing \\ \max\limits_{t \in \psi(g_x)} t.v & \text{if } \psi(g_x) \neq \varnothing \end{cases} \tag{3}$$

$$ub_{max}(g_x) = \begin{cases} v_{max} & \text{if } \psi(g_x) = \varnothing \\ \max\limits_{t \in \psi(g_x)} t.v & \text{if } \psi(g_x) \neq \varnothing \end{cases} \tag{4}$$

Since the tuple values are accessed in their descending order, the first tuple value seen in a group must be the maximum value in the group. For the `MIN` function, the problem could be reduced into that of the `MAX` function, as discussed in Section 3.2.

Figure 2 illustrates a pseudo-code of SA. The algorithm organizes groups that have been seen so far and can end-up in the top-k result in a hash-table $\mathcal{H}$ (with group-id as search key). It also maintains an initially empty min-heap $\mathcal{W}_k$ with the k groups of the highest $lb(g_x)$ so far. When a tuple in group g_x is retrieved, we update its information in $\mathcal{H}$ and check whether it

can enter $\mathcal{W}_k$. If an existing group g_y (in $\mathcal{H}$) satisfies both conditions (i) $g_y \notin \mathcal{W}_k$, and (ii) g_y's upper bound is at most ρ (the lowest $lb(g_x)$ in $\mathcal{W}_k$), then g_y can be pruned as it cannot become the top-k result. We perform this pruning test for all seen groups after each tuple access only if $\tau := v_{max} \cdot C_{max} \leq \rho$. The reason is that while $\tau > \rho$ any group can enter the top-k result (see [21] for a theoretical proof in the NRA context), thus it is meaningless to attempt pruning. We say that the algorithm enters the *shrinking phase* once $\tau \leq \rho$ is satisfied, because the set of candidates can only shrink after that point. Finally, the algorithm checks whether the remaining (non-pruned) groups are k, in order to terminate. The pruning at lines 12–15 is performed efficiently with the help of a priority queue that organizes groups seen so far by their upper bound scores.

Theoretically, the worst case tuple access cost of SA is:

$$cost_{SA}(N) = N \tag{5}$$

where N is the number of tuples in the input table $\mathcal{T}$. It happens when all the groups have the same score so that no pruning occurs. In practice, the access cost of SA is expected to be much smaller, thanks to the help of the above pruning technique.

```
Algorithm SA(Table T,Value k)
1.   H:=new hash-table (managing seen groups);
2.   W_k:=∅; /* empty min k-heap */
3.   insert k pairs of (null, 0) into W_k;
4.   ρ:=minimum value in W_k;
5.   while (nonempty(T))
6.      (tid, x, v):=GetNext(T);
7.      v_max:=v;
8.      update the lb(g_x) score of g_x in H by v;
9.      if (lb(g_x) > ρ) then
10.        update W_k and ρ;
11.     τ:=v_max · C_max;
12.     if (τ ≤ ρ) then /* prune groups */
13.        for each g_y ∈ H such that g_y ∉ W_k
14.           if (ub(g_y) ≤ ρ) then
15.              remove g_y from H;
16.        if (|H| = k) then
17.           return W_k;
```

Fig. 2. Sorted access algorithm (SA)

Example We illustrate the functionality of SA on the table of Figure 1, assuming $k = 1$. Note that $C_{max} = 3$ in this example. First, we access a tuple of g_2 and set v_{max} to its value 0.7. Then, we update $|\psi(g_2)| = 1$ and $lb(g_2) = 0.7$. Also, the best lower bound score ρ is updated to 0.7 ($\mathcal{W}_k = \{g_2\}$). The procedure continues since $\tau = v_{max} \cdot C_{max} = 0.7 \cdot 3 > \rho$. Next, we retrieve a g_2 tuple of value 0.69, update $|\psi(g_2)| = 2$, $lb(g_2) = 1.39$, and set $\rho = 1.39$. As $\tau = 0.69 \cdot 3 > \rho$, we continue and access a g_4 tuple of value 0.5, update $|\psi(g_4)| = 1$ and $lb(g_4) = 0.5$. Since $\tau = 0.5 \cdot 3 > \rho$, we continue. Then, we retrieve a g_5 tuple of value 0.41 and update $|\psi(g_5)| = 1$, $lb(g_5) = 0.41$. Now, we have $\tau = 0.41 \cdot 3 \leq \rho$, meaning that any group which has not been seen cannot end-up in the result. Hence, the result must be one of the groups that have been accessed (i.e., groups g_2, g_4, g_5). For this, we must access the remaining tuples until ρ is no smaller than the upper bound scores of all accessed groups not in $\mathcal{W}_k$. Since $ub(g_4) = 1.32 \leq \rho$ and $ub(g_5) = 1.23 \leq \rho$, the algorithm terminates and reports g_2 as the result with $lb(g_2) = 1.39$.

In practice, the value of C_{max} may not be known in advance. For this case, we can apply SA$'$, an adapted version of SA that replaces Equation 2 by

$$ub(g_x) = lb(g_x) + v_{max} \cdot C_{remain} \tag{6}$$

and defines $\tau = v_{max} \cdot C_{remain}$, where C_{remain} denotes the number of tuples in $\mathcal{T}$ that have not been accessed yet. As we will see in the experimental evaluation, when C_{max} is replaced by C_{remain}, the upper bound scores of the groups become much looser, forcing the algorithm to access more tuples before its termination.

Observe that SA (and its extensions presented in subsequent sections) compute the top-k groups and a *lower bound* of their scores. Upper bounds of these scores can be determined by Equations 2 or 6. In other words, SA does not necessarily return the *exact* scores of the query results.

We now discuss the computational requirement of SA. After τ drops below ρ, the pruning step (Lines 13-15) is computationally demanding as the whole hash table $\mathcal{H}$ in memory has to be traversed in each iteration. In order to speed up the process, we suggest that, the pruning step is executed each time a block of tuples is read, instead of every time a tuple is read. This heuristic would greatly reduce the CPU cost of SA, and the only overhead is that at most one extra disk page needs to be accessed.

4.2. *Multi-pass Sorted Access Algorithm*

SA becomes inapplicable in practical applications as the memory cannot accommodate all candidate groups for the top-k result. In this section, we extend this method to *multi-pass sorted access algorithm* (MSA), which can be applied for memory-bounded settings.

The basic idea of MSA is to use up all available memory while scanning $\mathcal{T}$ and compute the top-k groups among those maintained in the buffer. Tuples that belong to other groups are written to a temporary table $\mathcal{T}'$. After the top-k set among the groups that fit in memory has been finalized, the algorithm frees the memory, keeping only information about the top-k set, and continues scanning $\mathcal{T}$, after having appended $\mathcal{T}'$ in its beginning. If necessary, $\mathcal{T}'$ is reconstructed and the appending process is repeated, until the top-k groups are guaranteed to be found.

Figure 3 shows a pseudo-code of MSA. Let M be the number of memory pages and B be the number of tuples that can fit in a page. MSA uses a hash table $\mathcal{H}$ of size bounded by $M \cdot B$ for tracking candidate groups in memory, and a creates a temporary table $\mathcal{T}'$ for storing tuples in groups that cannot fit in memory. Initially, variable $fullH$ is set to `false`, indicating that the hash table is not full. The algorithm operates similarly to SA. However, after $\mathcal{H}$ becomes full, tuples of groups not in memory are appended to $\mathcal{T}'$ for later processing. If only k groups remain after pruning of groups from $\mathcal{H}$ (shrinking phase), MSA terminates if $\mathcal{T}'$ is empty. If $\mathcal{T}'$ is not empty, the algorithm "appends" $\mathcal{T}'$ to $\mathcal{T}$, by "moving" the current cursor to the beginning of $\mathcal{T}'$ and directing the cursor to the current position of $\mathcal{T}$, after $\mathcal{T}'$ has been read. This guarantees that all unprocessed tuples will be seen in sorted order (since they are written in sorted order

```
Algorithm MSA(Tables 𝒯,𝒯′,Value k)
1.  ℋ:=new hash table (managing seen groups);
2.  𝒲_k:=∅;
3.  insert k pairs of (null, 0) into 𝒲_k;
4.  ρ:=minimum value in 𝒲_k;
5.  fullH:=false;
6.  while (nonempty(𝒯))
7.      (tid, x, v):=GetNext(𝒯);
8.      v_max:=v;
9.      if (fullH ∧ g_x ∉ ℋ) then
10.         append the tuple (tid, x, v) to 𝒯′;
11.     else
12.         update (g_x, lb(g_x)) in ℋ;
13.         if (lb(g_x) > ρ) then
14.             update 𝒲_k and ρ;
15.         fullH:=fullH ∨ (|ℋ| = M · B);
16.     τ:=v_max · C_max;
17.     if (τ ≤ ρ) then /* prune groups */
18.         for each g_y ∈ ℋ such that g_y ∉ 𝒲_k
19.             if (ub(g_y) ≤ ρ) then
20.                 remove g_y from ℋ;
21.         if (𝒯′ = ∅ ∧ |ℋ| = k) then
22.             return 𝒲_k;
23.         if (𝒯′ ≠ ∅ ∧ |ℋ| = k) then
24.             append 𝒯′ before 𝒯;
25.             clear 𝒯′; goto Line 5;
```

Fig. 3. Multi-pass sorted access algorithm (MSA)

in $\mathcal{T}'$ and all tuples in $\mathcal{T}'$ have greater values than the remainder of $\mathcal{T}$). In addition, tuples from $\mathcal{T}$ before the current cursor which have already been considered by the groups of $\mathcal{H}$ will not be processed again. Note that at each loop, the algorithm initializes a new $\mathcal{T}'$ to be used if necessary during the next pass. Figure 4 illustrates this virtual appendix of $\mathcal{T}'$ to $\mathcal{T}$ at each loop of MSA.

Note that it is possible that some tuples appended to $\mathcal{T}'$ may belong to groups that have already been pruned (i.e., in a previous pass). However, this does not affect the correctness of MSA. When those tuples are processed from $\mathcal{T}'$ in the next pass, their corresponding groups will only have smaller upper score bounds and eventually get pruned. MSA can also be adapted to a variant MSA$'$ that does not use C_{max} (in the same way as SA is adapted to SA$'$).

We proceed to analyze the worst case tuple access cost of MSA, by using the notations in Table 1. In the worst case, the minimum number of groups that have been completely aggregated in each pass is MB (assuming $MB \gg k$), leading to a reduction of $MB(N/G)$ tuples in the data cardinality. There are at most $G/(MB)$ passes. In the i-th pass, at most $N(1-(i-1)MB/G)$ tuples are read from $\mathcal{T}$ and $N(1-iMB/G)$ tuples are written into $\mathcal{T}'$. As a result, the worst case tuple access cost of MSA is:

$$cost_{MSA}(N, M, B, G) = \sum_{i=i}^{\frac{G}{MB}} N \cdot \left(2 - \frac{(2i-1)MB}{G}\right) \tag{7}$$

Example We demonstrate the execution steps of MSA on the example in Figure 1, assuming that $k = 2$ and the memory can fit at most 3 tuples. After reading the first four tuples, we have $lb(g_2) = 1.39$, $lb(g_4) = 0.5$, $lb(g_5) = 0.41$, and the best lower bound score $\rho = 0.5$. Since

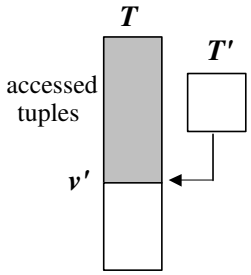


Fig. 4. Temporary output table $\mathcal{T}'$

$\tau = 0.41 \cdot 3 > \rho$, we need to continue. The memory is now full and the next retrieved tuple (of g_1) is inserted into the temporary table $\mathcal{T}'$. Then, we read a tuple of g_4, updating $lb(g_4) = 0.89$ and $\rho = 0.89$. After that, we access the next two tuples (of g_5), updating $lb(g_5) = 0.87$. Since $\tau = 0.13 \cdot 3 \leq \rho$, no unseen group will enter the top-k. In addition, we prune the group g_5 as $ub(g_5) = 0.87$. Now, there are exactly $k = 2$ groups in the memory (and free space). We continue with the process after the table $\mathcal{T}'$ is appended to the beginning of $\mathcal{T}$. The group g_1 gets pruned after the next tuple (of value 0.12) is read. Eventually, MSA returns the groups g_2 and g_4 as the top-2 results.

4.3. *Write-optimized MSA*

Recall that when SA and MSA are in the shrinking phase (i.e., $\tau \leq \rho$) the number of candidate groups in $\mathcal{H}$ can only be reduced. On the other hand, while the top-k results for the currently processed groups are finalized in MSA, if the group of the current tuple is not in $\mathcal{H}$, it will immediately be flushed to $\mathcal{T}'$. However, due to the pruning of $\mathcal{H}$, there might be enough memory to accommodate these temporary tuples, thus there is no need to flush them to disk immediately. More significantly, if we use the memory space freed from $\mathcal{H}$ to temporarily store tuples, we could apply early aggregation for some groups; two or more tuples of the same group temporarily held in memory can be aggregated and shrunk to a single tuple for the current group.

We propose WMSA, a write-optimized version of MSA, which utilizes the memory freed from $\mathcal{H}$ in the shrinking phase. In that phase, we introduce an extra hash table $\mathcal{H}'$ for aggregating tuples of groups which do not appear in $\mathcal{H}$. The initial capacity of $\mathcal{H}'$ is set to 1 memory page, and later extended by the freed memory pages, as $\mathcal{H}$ shrinks. When $\mathcal{H}'$ becomes full, we flush its content to the temporary table $\mathcal{T}'$. Tuples of the same group are written to $\mathcal{T}'$ as a single tuple, containing the group-id (i.e., values of group-by attributes), the number of tuples seen in the group% ** and the partial aggregate for the group (e.g., sum). Note that $\mathcal{T}'$ may contain multiple records of the same group, which are not flushed at the same time.

In the next pass, the groups in $\mathcal{T}'$ are read and processed according to their flushed order. Nevertheless the aggregated values of the groups in $\mathcal{T}'$ may now not appear in sorted order, while we need this order to derive an appropriate value for v_{max} (e.g., in Line 8 of MSA) to set a pruning and terminating condition. To solve this problem we apply the following heuristic.

**This quantity is required to compute $|\psi(g_x)|$, used by Equation 2.

Every time $\mathcal{H}'$ is flushed to $\mathcal{T}'$, together with the flushed partition, we attach the value of the last tuple read from $\mathcal{T}$. Thus, if $\mathcal{T}'$ is created by flushing $\mathcal{H}'$ R times, it can be thought of as a sequence of R partitions $\mathcal{T}'_1, \ldots \mathcal{T}'_R$, such that for each $\mathcal{T}'_i$ we know m_i; the minimum value of all (early-aggregated) tuples in it. When $\mathcal{T}'$ is processed after being appended to the beginning of $\mathcal{T}$'s remainder, we update v_{max} to m_i after each partition $\mathcal{T}'_i$ has been processed.

For example, assume that $\mathcal{H}'$ has become full and it was flushed three times to the $\mathcal{T}'$ shown in Figure 5. For the three partitions we keep the minimum value of the original tuples from $\mathcal{T}$ there. When we process $\mathcal{T}'$, for each tuple we read, we do not update v_{max} as in Line 8, but only after the whole partition $\mathcal{T}'_1$ is read, we update v_{max}:=m_1, before we proceed to the next partition $\mathcal{T}'_2$. Similarly, when $\mathcal{T}'_2$ is complete, we set v_{max}:=m_2, etc. Thus, although we cannot construct a total ordering for the aggregated tuples in $\mathcal{T}'$, we can determine bounds between the different partitions of it, using the known information from the original tuples between flushes.

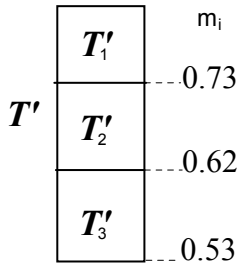


Fig. 5. Flushed partitions in $\mathcal{T}'$

If, while $\mathcal{T}'$ is read, $\mathcal{H}$ (and $\mathcal{H}'$) becomes full and tuples must be written to a $\mathcal{T}''$, m_i of the current $\mathcal{T}'_i$ being read is used as m_j for the current partition $\mathcal{T}''_j$ flushed to disk. The correctness of the algorithm is not affected, since m_i will be a conservative upper bound for the values in $\mathcal{T}''_j$.

As a remark, the worst case access cost of WMSA is the same as that of MSA. This occurs when the number G of groups in the data is much larger than the memory size and; and thus no aggregation occurs in the extra hash table $\mathcal{H}'$.

A final thing to note is that SA-based algorithms (MSA and WMSA) can also be used for cases where the input is *partially* ordered. Consider a case, where $\mathcal{T}$ is divided into multiple partitions $\mathcal{T}_1, \mathcal{T}_2, \ldots$, such that each partition has a different range of v values and these ranges are disjoint, covering the whole domain of v. For such cases, we can apply a process similar to SA (and its extensions), except that it operates on a partition-by-partition basis instead of a tuple-by-tuple basis.

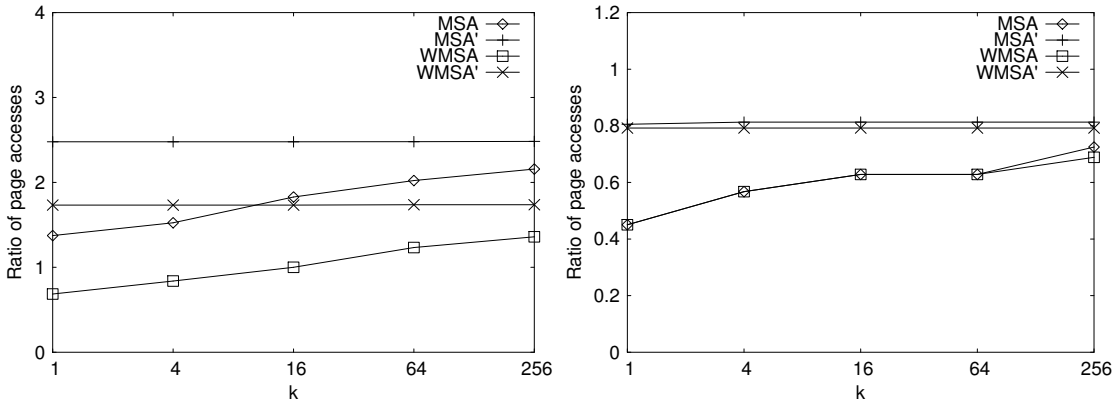
4.4. *Experimental Results*

Following the experimental setting of Section 3.2, we now evaluate the performance of bounded-memory algorithms on input physically ordered by the measure attribute: MSA, MSA′, WMSA

and WMSA′.

Figure 6 shows their access costs with respect to k. MSA′ and WMSA′ do not use C_{max} so they are more expensive than MSA and WMSA respectively. The cost of MSA′ (WMSA′) is insensitive to k and converges to the cost of MSA (WMSA) for large k values. WMSA (WMSA′) outperforms MSA (MSA′) because it performs early aggregation of groups and reduces the sizes of temporary tables $\mathcal{T}'$. The algorithms have lower costs for query $Q10$ than $Q1$ because $Q1$ considers a larger number of distinct groups, having higher memory requirements. Note also that MSA (MSA′) has similar cost to WMSA (WMSA′) in $Q10$, as the temporary tables $\mathcal{T}'$ are of negligible size (note the sub-linear costs of the methods).

Indeed, we have also implemented versions of the *threshold algorithm* [6], trying to take advantage of potential (different) orders of both the measure values and other attributes (e.g., if multiple B$^+$–trees exist) on different attributes. However, such methods were almost always found to be more expensive than SA and its extensions, mainly due to the overhead of required random accesses.

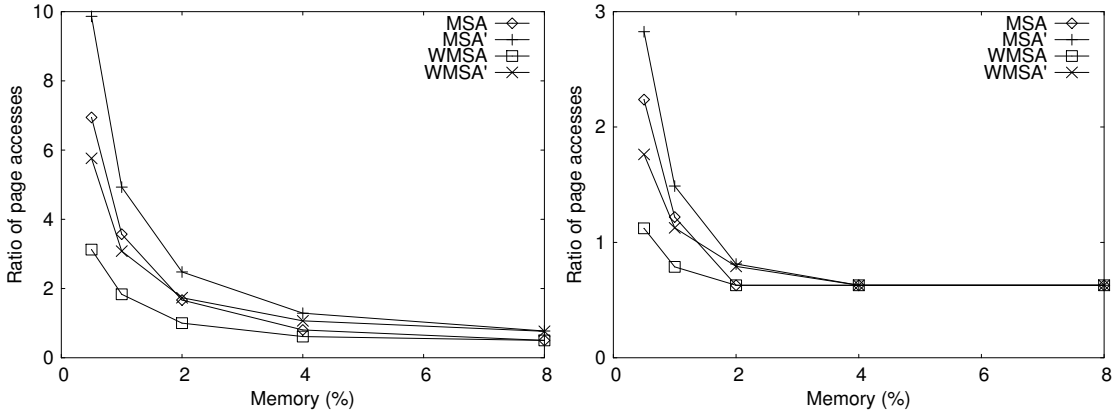


(a) Query $Q1$ (b) Query $Q10$

Fig. 6. Ratio of page accesses vs result size k

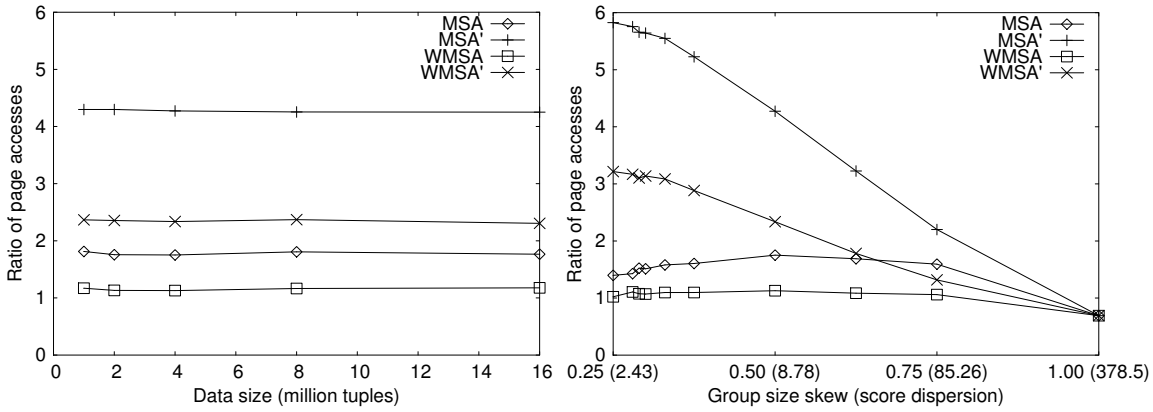
We now study how the performance of algorithms is affected by the available memory (see Figure 7). As expected, their costs decrease as the memory size increases. With a larger memory buffer, more candidates can be stored in memory and fewer passes are performed over the data. Again, WMSA outperforms the other methods. Observe that with as little memory as 2% of the data size, the cost of this method becomes sublinear.

In the next two experiments, we study the behavior of the algorithms on synthetic datasets. Figure 8a shows the access costs of the algorithms with respect to the database size. Note that they are insensitive to this parameter, given a proportional memory buffer (2%). Figure 8b shows their access costs for different group size skew θ_s. To help understanding the results, we show the *group score dispersion* of each tested dataset in a bracket, which is defined as the ratio of the standard deviation of the groups' scores to their mean score. In general, the cost of the algorithms decreases at high group size skew. However, at low group size skew, most of the groups have similar sizes and C_{max} has a low value. Thus, upper bound scores (see Equation 2) of the groups become very tight and MSA/WMSA can terminate early.



(a) Query $Q1$ (b) Query $Q10$

Fig. 7. Ratio of page accesses vs memory size, result size $k = 16$



(a) vs data size (b) vs group size skewness

Fig. 8. Ratio of page accesses on synthetic data

5. Algorithms for Inputs with Random Order

In this section, we study the evaluation of top-k queries for generic inputs $\mathcal{T}$, where the tuples appear in random order. We assume that memory is bounded such that we cannot accommodate a counter for each group. Thus, multiple passes over the data might be required to derive the exact set of top-k groups. We present (i) an adaptation of an effective iceberg query algorithm [7] for our problem and (ii) an extension of the hash-based aggregation algorithm [9] that minimizes the number of accesses for top-k groups computation, using branch-and-bound techniques in combination with appropriate heuristics for deriving tight upper bounds for the candidate groups.

Our *bucket algorithm* (BA) is an adaptation of the Defer-Count method of [7], which was proposed for the evaluation of iceberg queries; i.e., the identification of groups with `COUNT` (or `SUM`) value above a given threshold ρ. In our problem, the threshold ρ is not given, but corresponds to the score of the k-th group, which is unknown a priori. Thus, the main difference between Defer-Count and BA is the online tuning of a lower bound for ρ, during the execution of the algorithm.

Figure 9 shows a detailed pseudo-code for BA. BA follows a two-step filter/refinement paradigm. The filter step (Lines 3–15) consists of a number F of passes over the dataset, during which, groups that may not end up in the result are identified and eliminated. During the refinement step (Lines 16–21), the remaining candidates are aggregated and compared. Parameter F denotes the number of passes over the data, during the filter step. This parameter balances the cost of the filter and refinement steps. A high F value results in high filter effort, but leads to few candidates. On the other hand, a low F implies low filter cost but high refinement cost. We will discuss how to eliminate this parameter in Section 5.1.2.

First, the algorithm draws a random sample $\mathcal{T}'$ of $\mathcal{T}$.The $\kappa \geq k$ groups with the highest aggregate scores in the sample (i.e., groups expected to be heavy targets) are written to a set $\mathcal{C}$ of *candidates*.[⋆⋆⋆] The algorithm then executes the filter step F times. In the j-th filter step, an array A of counters and a bitmap E_j are employed. Each position of A and E_j corresponds to a value of a hash function $h_j(x)$, which takes the group-id x as input. Moreover, different hash functions are applied for different filter passes.

In the first filter pass, BA scans $\mathcal{T}$ and for each tuple t belonging to group x, (i) if $x \in \mathcal{C}$, x's aggregate score $agg(x)$ is updated (ii) otherwise, bucket $A[h_j(x)]$ is updated by v. Thus, after the pass, the exact aggregate scores for all groups in $\mathcal{C}$ are known. The k groups in $\mathcal{C}$ with the highest scores are inserted into the candidate top-k result $\mathcal{S}_k$. ρ is the lowest score in $\mathcal{S}_k$ and is used as a pruning bound. Finally, for each bucket $A[z]$, bitmap position $E_j[z]$ is set to 1 if the bucket exceeds ρ.

In the subsequent filter passes, tuples that do not belong to groups in $\mathcal{C}$ (these groups have already been counted) are hashed to buckets, only if their bucket counters in all previous passes exceed ρ. This is verified by the use of the previous bitmaps. Such a bitmap manipulation technique has been introduced in [7], and was shown to effectively reduce the number of candidate groups that cannot lead to the result.

During the refinement step, only groups having 1 in all bitmap positions of the filter passes are counted exactly and the top-k result is derived by considering also the counted groups of $\mathcal{C}$. Note that if the set of all candidates in the refinement step cannot fit in the memory, then they must be partitioned and counted over multiple passes of data.

[⋆⋆⋆] In general, we count a larger number κ than k of heavy hitters exactly, in order to decrease the value of the hash buckets for the remaining attributes and reduce the number of candidates that will eventually be counted.

Algorithm **BA**(Table $\mathcal{T}$,Values k,F,κ)

1. draw a sample $\mathcal{T}'$ of $\mathcal{T}$;
2. $\mathcal{C}$:= the top-κ groups in the sample $\mathcal{T}'$;
   /* filter step */
3. **for each** $j \in [1, F]$
4. initialize hash buckets A;
5. **for each** tuple $(tid, x, v) \in \mathcal{T}$ /* scan $\mathcal{T}$ */
6. **if** $(x \in \mathcal{C})$ **then**
7. **if** $(j = 1)$ **then** /* first filter pass */
8. update $agg(x)$; /* score of group x */
9. **else if** $(E_i[h_i(x)], \forall\, i \in [1, j-1])$ **then**
10. update the bucket $A[h_j(x)]$ by v; /* by using the function agg */
11. **if** $(j = 1)$ **then** /* first filter pass */
12. $\mathcal{S}_k$:=top-k groups in $\mathcal{C}$;
13. ρ:=k-th score in $\mathcal{C}$;
14. **for each** bucket $A[z]$
15. $E_j[z]$:=$(A[z] > \rho)$;
    /* refinement step */
16. $\mathcal{C}'$:=$\varnothing$;
17. **for each** tuple $(tid, x, v) \in \mathcal{T}$ /* scan $\mathcal{T}$ */
18. **if** $(x \notin \mathcal{C})$ **then**
19. **if** $(\forall\, i \in [1, F]\ B_i[h_i(x)])$ **then**
20. update $agg(x)$ in $\mathcal{C}'$;
21. update $\mathcal{S}_k$ and ρ using $\mathcal{C}'$;
22. return $\mathcal{S}_k$;

Fig. 9. Bucket algorithm (BA)

We continue to analyze the worst case tuple access cost of BA, based on the symbols in Table 1. Clearly, the filter step leads to $F \cdot N$ tuple accesses. In the worst case, no pruning occurs and all the groups belong to the candidate set. Since the memory can fit MB groups in the same refinement pass, there are $G/(MB)$ refinement passes. Each refinement pass incurs N tuple accesses. Therefore, the worst case tuple access cost of BA is:

$$cost_{BA}(N, M, B, G, F) = N \cdot \left(F + \frac{G}{MB}\right) \tag{8}$$

5.1.1. *An Example*

As an example, assume that we run BA on the table of Figure 10, with parameters $k = 1$, $F = 2$ and $\kappa = 1$. Assume that in the random sample, the top $\kappa = 1$ group is g_5, thus $\mathcal{C} = \{g_5\}$. Assume that we run BA with $F = 2$ filter passes. Figure 11 illustrates the buckets for the groups (e.g., function h_1 places groups g_2 and g_4 in the first bucket). In the first pass, we compute the counters A for the two buckets and the exact aggregate scores (i.e., 0.91) of the groups in $\mathcal{C}$ (i.e., g_5), and thus we have $\rho = 0.91$. After the pass, the first bit of the bitmap E_1 is set, as the aggregate value of the first bucket $A[1]$ is greater than ρ. In the second pass, we scan the table again and compute new buckets A, using another hash function. Since $E_1[1] = 0$ (from the previous pass), tuples of g_1 and g_3 are ignored in the current pass. This reduces the counters of the current buckets and only $E_2[2]$ is set. Note that the only non-pruned group at

this point (by either hash function) is g_2. Thus, in the refinement step, only g_2 will be counted, $agg(g_2)$ will be compared to $\mathcal{C} = \{g_5\}$, and eventually g_2 will be output as the top-1 group.

tid	*gid*	*v*
1	5	0.20
8	2	0.70
4	4	0.40
9	2	0.69
2	5	0.33
5	4	0.50
10	2	0.10
11	1	0.15
3	5	0.38
7	3	0.11
6	3	0.12
12	1	0.05

Fig. 10. Example of query input

bucket slot	1	2
groups	g_2, g_4	g_1, g_3
A	2.39	0.43
E_1	1	0

First pass, $\rho = 0.91$

bucket slot	1	2
groups	g_4	g_2
A	0.90	1.49
E_2	0	1

Second pass, $\rho = 0.91$

Fig. 11. Filter step of the bucket algorithm

5.1.2. *Optimizations*

We developed two optimization techniques for terminating the filter step in BA early. The first one is to terminate when the number of bits set in the current filter pass is not smaller than that in the previous filter pass. This indicates that additional filter passes may not filter any groups. The second optimization is to terminate the filter step when the expected number of candidates is small enough to fit in memory. For this, we compute an FM sketch [8] in the first

filter pass to estimate the number of distinct groups in the dataset. An upper bound of the candidate set size can be estimated by multiplying the percentage of bits set in the current pass with the estimated distinct number of groups. In addition to the cost savings, the above optimizations lead to automatic termination of the filter step so that users need not specify the number F of filter passes.

5.2. *The Recursive Hash Algorithm*

One potential problem with BA is that it is not easy to determine an appropriate value for the number of buckets (i.e., bitmap length). In case there are too few buckets, many groups are hashed to the same bucket and the filter becomes ineffective. The effectiveness of the filter step can be improved by employing more buckets, however, the main memory may not be large enough to accommodate a huge number of buckets.

To avoid this problem, we extend the *recursive hash algorithm* (RHA) [9] with early aggregation to retrieve the top-k groups. Graefe et. al. [9] compute the aggregate scores of *all* groups unconditionally, even though most of them cannot be the top-k groups. Motivated by this, we present two carefully designed optimizations for enhancing the performance of RHA on top-k groups extraction. First, during hashing and early aggregation, our RHA algorithm derives upper bound aggregate values for groups in hash partitions and uses them to eliminate partitions that cannot lead to any result. Second, we propose an appropriate ordering of processing the partitions, which leads to early discovery of groups with high aggregate values and significant access cost reduction of the algorithm.

Figure 12 illustrates the pseudo code of RHA. Let B be the number of group counters that can fit in a page and M be the number of memory pages. RHA first initializes an initially empty set $\mathcal{S}_k$ of current top-k groups and then invokes the recursive routine on the table $\mathcal{T}$. The recursive routine consists of three phases: the hashing phase (Lines 1–11), the clearing phase (Lines 12–17), and the branching phase (Lines 18–26). At each recursion level i, RHA uses a hash function $h_i()$ that partitions the tuples to $\mathcal{R} < M$ buckets.

In the hashing phase, for each bucket r, a variable ub_r tracks the maximum possible aggregate value of a group hashed there. For each tuple read from $\mathcal{T}$, we apply $h_i()$ on its group-id x, to determine its partition r. We then check whether there is already a partial result for x in the part $\mathcal{M}_r$ of r in memory. In this case, we simply update this partial aggregate $agg(x)$ (early aggregation), otherwise we create a new memory slot for x. If $\mathcal{M}_r$ overflows, we first (i) compute the maximum upper bound for a group in $\mathcal{T}_r$, by adding to ub_r the maximum $agg(x) \in \mathcal{M}_r$, and then (ii) flush the content of $\mathcal{M}_r$ to the corresponding disk partition $\mathcal{T}_r$.

After scanning all tuples from $\mathcal{T}$, the recursive routine enters the clearing phase. We update the set $\mathcal{S}_k$ of the best k groups so far (and the k-th score ρ, used as a pruning bound), from the partitions for which no groups have been flushed to disk. For these groups, $agg(x)$ is guaranteed to be their complete scores. The remaining memory partitions are flushed to the corresponding disk partitions for further processing and their ub_r values are updated.

```
Algorithm Recur_RHA(Table T, Set S_k, Level i)
    /* hashing phase */
1.  for each r ∈ [1, R]
2.      ub_r:=0;
3.  for each tuple (tid, x, v) ∈ T
4.      r:=h_i(x); /* recursion level i: apply h_i() */
5.      if (x ∈ M_r) then /* in memory */
6.          update agg(x) in M_r; /* partial aggregate */
7.      else
8.          if (M_r is full) then /* bucket overflow */
9.              ub_r:=ub_r+max{agg(x) ∈ M_r};
10.             append M_r to its disk partition T_r;
11.         add the group x in M_r and set agg(x) to v;
    /* clearing phase */
12. for each r ∈ [1, R]
13.     if (T_r = ∅) then
14.         retrieve groups in M_r, update S_k and ρ;
15.     else
16.         ub_r:=ub_r+max{agg(x) ∈ M_r};
17.         append M_r to its disk partition T_r;
    /* branching phase */
18. for each T_r ≠ ∅ in descending order of ub_r
19.     if (ub_r > ρ) then
20.         if (|T_r| ≤ B · M) then
21.             load T_r into memory;
22.             aggregate groups in T_r, update S_k and ρ;
23.         else
24.             Recur_RHA(T_r,S_k,i + 1);
25.     else break for-loop; /* no need to check more T_r */
26. return S_k;

Algorithm RHA(Table T,Values k)
1.  S_k:=∅;
2.  insert k pairs of (null, 0) into S_k;
3.  ρ:=minimum value in S_k;
4.  Recur_RHA(T,S_k,0);
5.  return S_k;
```

Fig. 12. Recursive hash algorithm (RHA)

Next, in the branching phase, RHA is recursively applied to each partition in descending order of their ub_r. The rationale is to discover groups with high aggregate values as early as possible. We prune partitions for which $ub_r \leq \rho$, since they cannot contain groups with higher aggregate values than those currently in $\mathcal{S}_k$. If the next processed partition $\mathcal{T}_r$ fits in memory, it is loaded and its groups are aggregated while $\mathcal{S}_k$ and ρ are updated. Otherwise, the algorithm invokes the recursive routine for the disk partition $\mathcal{T}_r$.

5.2.1. *Deriving Tighter Upper Bounds*

We can derive a tighter upper bound than ub_r for the groups in a partition $\mathcal{T}_r$, if ub_r is extended to a set of multiple counters. Specifically, for each partition r, we use an array of L counters and a secondary hash function h' for them. Initially, all L counters for partition r are set to 0. For each memory part $\mathcal{M}_r$ of r, before $\mathcal{M}_r$ is flushed to disk, we repartition it in memory into L segments, by applying $h'(x)$ to each group $x \in \mathcal{M}_r$. In each segment j, $1 \leq j \leq L$, we find the group with the maximum $agg(x)$ and add this value to the j-th counter. Thus, after $\mathcal{T}_r$ has been finalized, the maximum counter will give an upper bound ub_r for the maximum group

aggregate in $\mathcal{T}_r$.

The use of multiple counters presents us with a trade-off between memory allocation and tightness of the computed bound. The more counters allocated for a partition, the less memory can be used for holding and early-aggregating groups. Note that for a partition r, allocating space for counters is not necessary until the first time $\mathcal{M}_r$ is flushed to disk. After that point, L counters are preserved for them in the memory. Thus, the average memory allocated per partition in the worst-case is $\frac{M}{R} - L$. The impact of L to the overall performance of RHA is studied in the experimental section.

As a final note, the upper bound of RHA's cost is the cost of hash-based aggregation [9] without any pruning:

$$cost_{RHA}(N, M, B, G, \mathcal{R}) = 2 \cdot N \cdot \lceil \log_{\mathcal{R}} \frac{G}{M \cdot B} \rceil \tag{9}$$

where $\mathcal{R}$ is the number of memory partitions, and the descriptions of other symbols can be found in Table 1. This is indeed the worst case of RHA, where all groups have the same aggregate score. However, in practice, skew in the distribution of scores brings significant cost savings to the algorithm, as we demonstrate in the next section.

Example We proceed to apply RHA for computing the top-1 group over the example of Figure 10. Assume that we have $\mathcal{R} = 2$ memory partitions, and each one is able to hold 2 groups at the same time. Suppose that the groups g_2, g_3 are hashed to the partition $\mathcal{M}_1$ whereas the groups g_1, g_4, g_5 are hashed to the partition $\mathcal{M}_2$. After reading the first 7 tuples from the input table $\mathcal{T}$, we have: (i) the group g_2 in $\mathcal{M}_1$, and (ii) the groups g_4 and g_5 in $\mathcal{M}_2$. The next retrieved tuple (with the value 0.15) comes from the group g_1, belonging to the memory partition $\mathcal{M}_2$. Since $\mathcal{M}_2$ is already full, the disk partition $\mathcal{T}_2$ is flushed the following (partial) group scores: $lb(g_4) = 0.9$ and $lb(g_5) = 0.53$. Its score bound ub_2 is set to $\max\{0.9, 0.53\} = 0.9$. Now, the group g_1 can be hashed into $\mathcal{M}_2$. The remaining tuples are read iteratively and their corresponding main memory partitions are updated. Subsequently, $\mathcal{M}_1$ contains $lb(g_2) = 1.49$ and $lb(g_3) = 0.23$; whereas $\mathcal{M}_2$ contains $lb(g_1) = 0.2$ and $lb(g_5) = 0.38$. As the disk partition $\mathcal{T}_1$ is empty, we obtain $\rho = \max\{1.49, 0.23\} = 1.49$. Then, we compute the upper score bound(s) for the non-empty disk partition(s): $ub_2 = 0.9 + \max\{0.2, 0.38\} = 1.28$. Since $ub_2 \leq \rho$, there is no need to process the disk partition $\mathcal{T}_2$. Eventually, the algorithm reports g_2 to be the top-1 group, having the score 1.49.

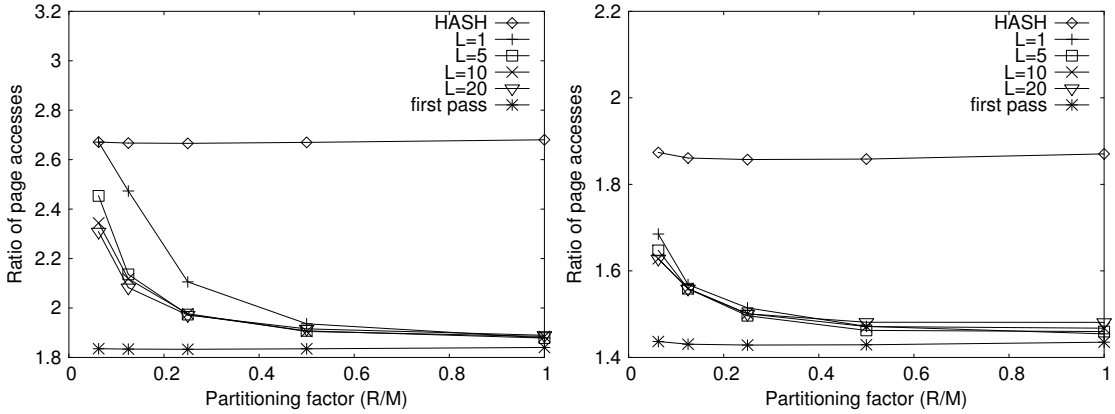
5.3. *Experimental Results*

In this section, we adopt the experimental setting of Section 3.2 and compare the relative performance of algorithms that operate on unordered data; BA and RHA. The algorithm BA is configured to use 10000 bucket counters and $\kappa = 2000$ heavy targets in memory, after drawing a 1%-sample from the data. The remaining memory is used for computing the FM sketch in the first filter pass.

In addition these methods, we implemented two baseline algorithms SORT and HASH. These correspond to the sorting and (recursive) hashing algorithms with early aggregation [9], which

compute the scores of all groups. In its final pass, SORT computes top-k groups while merging the runs of the previous pass. Similarly, HASH maintains a heap of the top-k groups, while computing the scores of the groups at its final pass. Thus, HASH corresponds to the worst-case of RHA, where no bucket can be pruned until the last pass.

We first assess the impact of the number of counters L and $\frac{R}{M}$ (the number of partitions over the total number of pages used for hashing) to the performance of RHA. Figure 13 shows the access cost of RHA for both queries $Q1$ and $Q10$, as a function of these parameters. As a basis for comparison, we also included HASH. We also included the essential cost of the algorithm; performing the first pass and writing to disk the overflown memory partitions. The results show that HASH has cost insensitive to the number of partitions, which is expected, since, for a given memory, the same number of tuples will be hashed and early aggregated, no matter how many partitions we have. On the other hand, our RHA algorithm that employs pruning becomes very efficient as $\frac{R}{M}$ increases. Note that for a large number of partitions RHA is only slightly more expensive than the essential cost of hashing in the first pass. First, the upper bounds ub_r of the partitions become smaller as partitions become smaller. Second, after the algorithm is recursively applied to a large number of small disk partitions $\mathcal{T}_r$, many small re-partitions are early and cheaply identified that help further tightening the bound ρ and terminating the algorithm. Note also that the more counters we use, the better ub_r values we derive, especially if the number of partitions is small. In limited-memory cases, where we are forced to use a small number of buckets, using counters pays off, however, if R is large, there is little or no improvement over using a single ub_r bound, as in Figure 12. In the rest of the experiments, we use a default number of $L = 10$ counters for RHA and as many partitions as permitted by the available memory.



(a) Query $Q1$ (b) Query $Q10$

Fig. 13. Ratio of page accesses of RHA, result size $k = 16$

Table 2
Ratio of page accesses of RHA vs ordering of partitions, result size $k = 16$

Ordering / Query	$Q1$	$Q10$
Ascending	2.690	1.895
Descending	1.885	1.467
Random	2.040	1.516

We now investigate the effect of ordering the partitions (Line 18 in Figure 12) for processing in RHA. The default ordering is to process the partitions $\mathcal{T}_r$ in descending order of their upper bound values ub_r. We also test the effect of ascending ordering and random ordering on the access cost of RHA. Table 2 shows the access cost of RHA with different orderings, for queries $Q1$ and $Q10$, at $k = 16$. The descending ordering leads to the lowest cost as groups with high aggregate values are discovered as early as possible. Observe that the descending ordering achieves 30% and 23% cost reduction over the ascending ordering, for queries $Q1$ and $Q10$ respectively.

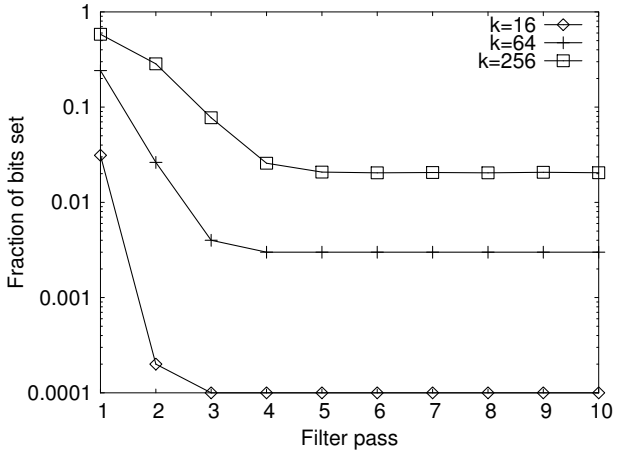
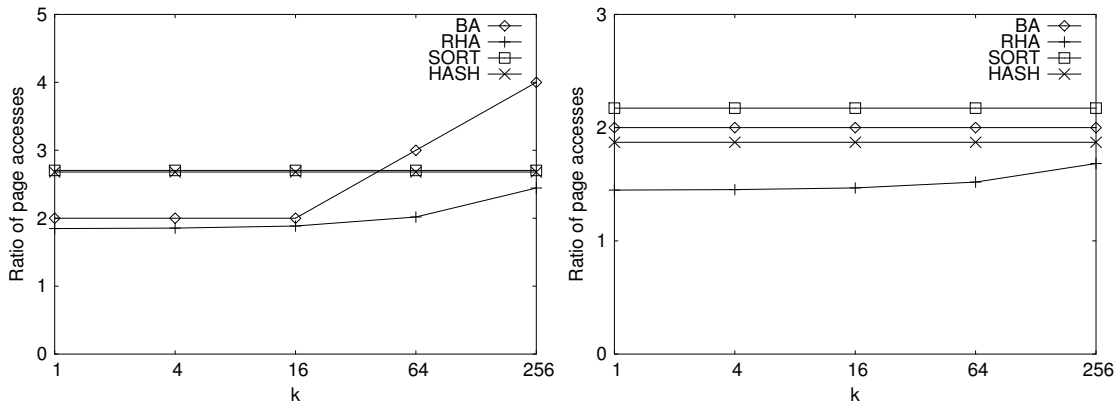


Fig. 14. Fraction of bits set in each filter pass, Query $Q1$

We proceed to study the effectiveness of our optimization techniques for terminating BA early. For this, we run BA for the query $Q1$ at $F = 10$, but without early termination. In this case, the filter step terminates after all F filter passes or the bitmap in the current filter pass has no bits set. Figure 14 shows the number of bits set in each filter pass, for different values of k. Although the number of bits set decreases rapidly after the first few filter passes, it never reaches zero. As a result, the algorithm has to execute $F = 10$ filter passes. Then, we run BA with early termination techniques for the same experiment. Now, BA performs only 1-3 filter passes (depending on the value of k) as it estimates that the remaining groups (with possible aggregate value greater than ρ) fit in memory. Eventually, those groups are aggregated in a single refinement pass. In the subsequent experiments, we apply early termination techniques for BA and set the maximum number of filter passes to $F = 10$.

In the remainder of the experiments, we compare all algorithms together. Figure 15 shows the access costs of BA, RHA, SORT, and HASH as a function of k. The costs of BA and RHA increase only for large value of k and the effect is more significant on BA. As k increases, (i) the pruning bound ρ becomes smaller and (ii) the top-k results are scattered in more partitions that are essentially examined. Both SORT and HASH are insensitive to k as they do not utilize the pruning bound. RHA outperforms the other methods as it can prune early the partitions that cannot lead to better results; its performance in the worst case becomes slightly better than HASH (for large values of k). BA performs well only for small k, where the sample contains the top-k groups with high probability. Observe that, for query $Q10$, BA is worse than HASH,

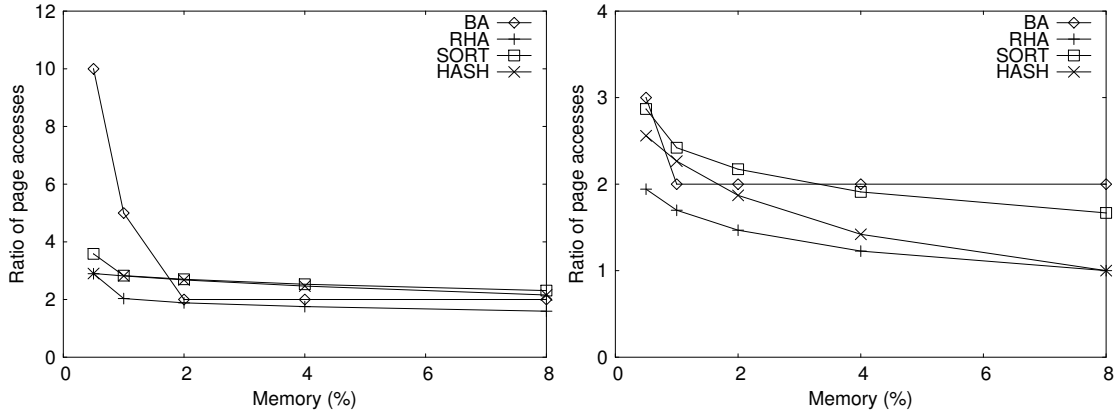
a fact indicating that defer-counting methods could be worse than simple early aggregation techniques (as also shown in [17]).



(a) Query $Q1$ (b) Query $Q10$

Fig. 15. Ratio of page accesses vs result size k

Figure 16 plots the access costs of the algorithms as a function the memory size. For BA, the number of hash buckets, heavy targets, and samples are scaled in proportion to the memory size. RHA has the best performance and it degrades gracefully for limited memory. The performance gap between RHA and HASH shrinks as memory size increases because at high memory size many groups can be aggregated in memory and the effect of pruning diminishes. For $Q1$, BA has very high cost at the lowest M, as it does not have adequate hash buckets to filter out disqualified groups, which then must be verified in multiple refinement passes. The large set to be refined cannot be reduced by additional filter passes, because the buckets A are large (independently of the hash-function used) and groups are rarely pruned.



(a) Query $Q1$ (b) Query $Q10$

Fig. 16. Ratio of page accesses vs memory size, result size $k = 4$

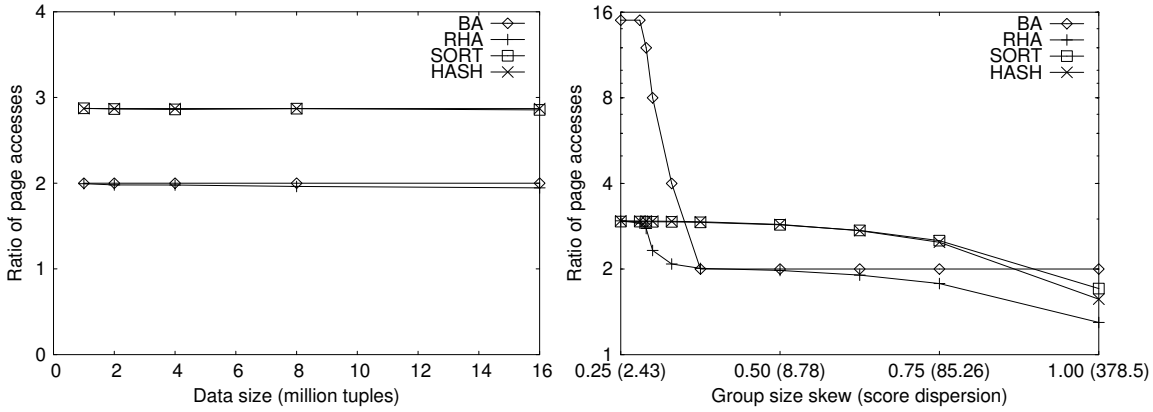
So far, we only considered the `SUM` aggregate function. Our algorithms can also be applied for other distributive functions: `COUNT`, `MAX`, and `MIN` (see Section 3.2). For instance, both BA and RHA are applicable to both `MAX` directly, after modifying the statements for updating group

scores and upper bound values of buckets/partitions (e.g., Lines 8, 10, 20 of BA; Lines 6, 9, 16 of RHA). Table 3 shows the access cost of the algorithms with respect to the aggregate function. We applied variants of $Q1$ and $Q10$ that replace the `SUM` function with some other aggregate. For the query $Q1$, BA is more expensive for `COUNT` than for `SUM`, due to the fact that `COUNT` considers only group sizes (but not the values of tuples in the groups). The cost of BA is even higher for `MAX`. The reason is that a group has high score when it has one (as opposed to many) tuple with high value. BA has the worst performance for `MIN` because most of the tuples have low values, leading to a high number of groups with low scores. RHA outperforms its competitors and its pruning effectiveness is stable for different aggregate functions. Since SORT and HASH do not apply any pruning technique, they have the same access cost for all aggregate functions. For $Q10$, the algorithms have similar relative performances.

Table 3
Ratio of page accesses vs aggregate function type

	Query $Q1$				Query $Q10$			
Algorithm	`SUM`	`COUNT`	`MAX`	`MIN`	`SUM`	`COUNT`	`MAX`	`MIN`
BA	2.000	3.000	5.000	12.00	2.000	2.000	2.000	5.000
RHA	1.885	1.880	1.851	1.847	1.467	1.467	1.457	1.449
SORT	2.703	2.703	2.703	2.703	2.172	2.172	2.172	2.172
HASH	2.680	2.680	2.680	2.680	1.870	1.870	1.870	1.870

Figure 17 shows the performance of the algorithms on synthetic datasets as a function of the data size and group size skew. Observe that their access costs are not sensitive to data size. In Figure 17b, each value shown in a bracket indicates the *group score dispersion* of the tested dataset, as the ratio of the groups' scores standard deviation to their mean score. The algorithms behave differently for different group size skew. At low group size skew, many groups have similar scores as the top-k groups. RHA could hardly prune any partition and thus degenerates into HASH. Similarly, BA requires many data passes as its buckets cannot help pruning unqualified groups effectively. Since RHA, SORT, and HASH apply early aggregation to reduce the data size during each pass, they perform much better than BA for the worst distribution of group sizes.



(a) vs data size (b) vs group size skewness

Fig. 17. Ratio of page accesses on synthetic data

6. Data Clustered by a Subset of Group-by Attributes

In this section, we show how the case of data ordered or clustered based on some group-by attributes can be reduced to the random-order case, where the same algorithms discussed in the previous section can be used as modules for query evaluation. At the end, we present an experimental study for the proposed method.

6.1. *The Clustered Groups Algorithm*

Consider a query with a set of group-by attributes $\mathcal{G}$ and assume that the data are clustered based on a set of attributes $\mathcal{G}'$, such that $\mathcal{G}' \subset \mathcal{G}$.[⋆⋆⋆⋆] That is, for each combination of values for the attributes in $\mathcal{G}'$, all tuples that have this combination are stored (or arrive, in case of a streaming input) continuously without any other tuples being interleaved.

For example, consider the query with $\mathcal{G} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, shown in Figure 18 together with an exemplary table T where it is applied. Note that, in the table, for each combination of values for attributes `a` and `b`, the tuples that contain this value are stored continuously (e.g., for `a`=a_1 and `b`=b_2 the only tuples that contain these values are the second and the third one, which are continuous in the table). In other words, $\mathcal{G}' = \{\mathtt{a}, \mathtt{b}\}$.

```
SELECT a,b,c,d,SUM(v)
FROM T
GROUP BY a,b,c,d
ORDER BY SUM(v) DESC
STOP AFTER k
```

(a) Query

a	b	c	d	v
a_1	b_1	c_1	d_1	0.70
a_1	b_2	c_2	d_2	0.62
a_1	b_2	c_1	d_3	0.23
a_1	b_4	c_2	d_3	0.14
a_3	b_2	c_1	d_2	0.51
a_3	b_2	c_1	d_3	0.62
a_3	b_4	c_2	d_4	0.73
a_4	b_1	c_1	d_1	0.80

(b) Table T

Fig. 18. Table clustered by group-by attributes

To solve this top-k query we can apply the *clustered groups* (CGA) algorithm shown in Figure 19. The algorithm reads all tuples from $\mathcal{T}$ that contain the same values for the group-by attributes in $\mathcal{G}'$ and for each batch it applies an unordered top-k groups algorithm, like RHA. In practice, only attributes in $\mathcal{G} - \mathcal{G}'$ are considered for each batch (since all $\mathcal{G}'$ attributes have exactly the same value). The results of these batches are merged to derive the global top-k

⋆ ⋆⋆⋆ The case where $\mathcal{G}' = \mathcal{G}$ can be trivially solved by one pass over the data; for each target group the tuples are clustered together, thus we only need a counter for the current group and a priority queue for the top-k groups so far.

groups. In our implementation, the algorithm does not wait until the whole batch is read before it begins the unordered algorithm, but the first phase of the algorithm used for the batch (e.g., hashing in the case of RHA) is executed while tuples of the batch are read. As soon as the batch is known to have been read completely, the next phases of the algorithm begin to compute the top-k groups for the batch. Observe that the best top-k groups from the previous batches are effectively used as bounds to prune the search space of the currently processed batch.

Algorithm **CGA**(Table $\mathcal{T}$, Sets $\mathcal{G}$, $\mathcal{G}'$)
1. **while** there are more tuples in $\mathcal{T}$
2. τ:=next tuple of $\mathcal{T}$;
3. add τ to $\mathcal{T}'$;
4. **while** next tuple τ' of $\mathcal{T}$ equals τ on all $\mathcal{G}'$ attributes
5. add τ' to $\mathcal{T}'$;
6. apply an unordered top-k groups algorithm on $\mathcal{T}'$;
7. update the global top-k groups;

Fig. 19. Clustered groups algorithm

We proceed to study the worst case tuple access cost of CGA, by using the symbols in Table 1. Let G' be the number of distinct groups per batch; there are G/G' batches in total. Thus, the cost of CGA is:

$$cost_{CGA}(N, M, B, G) = \frac{G}{G'} \cdot cost_{RHA}(\frac{NG'}{G}, M, B, G') \tag{10}$$

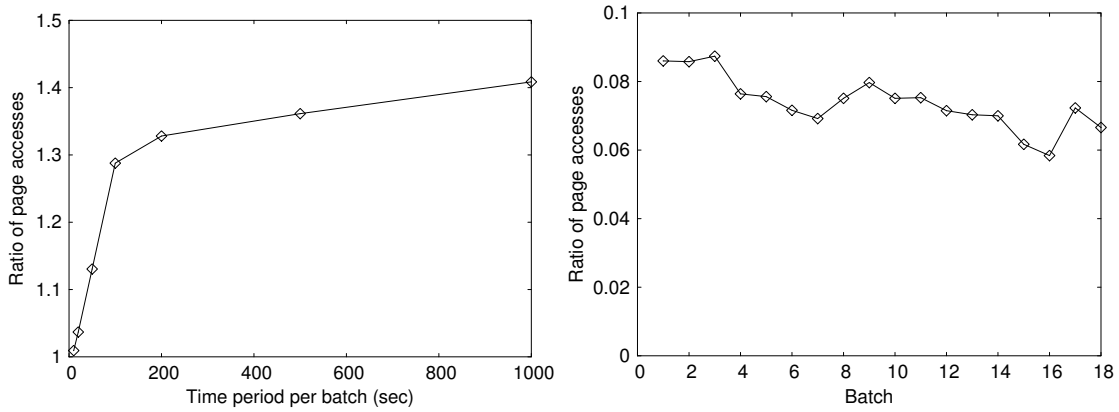
assuming that CGA applies RHA to compute the top-k groups within a batch of tuples. In order to simplify the equation, the hidden parameter $\mathcal{R}$ (specific to RHA) is not shown here.

6.2. *Experimental Results*

We followed the experimental setting of Section 3.2 and performed two experiments to validate the performance of the clustered groups (CGA) algorithm. For this purpose, we used the real dataset (i.e., TCP packets trace). We assume that the dataset is ordered by the time attribute. We then experimented with queries in the form of Q1, but with different time granularities for the `Tstp` attribute. In specific, in $Q1$, values are grouped in 1-second time intervals. We experimented with different time interval granularities, defining queries from $Q1$ to $Q1000$. Note that the timespan of the whole dataset is 1h = 3600 sec, thus the number of batches in the finest (coarsest) granularity is 3600 (4). The smaller the time interval, the larger the total number of groups, but the smaller the number of distinct sub-groups per interval (which affect the cost of the algorithm used at each batch). In addition, we used a relatively small memory buffer of 0.25% of the total dataset size; otherwise the batches would easily fit in memory and queries would be processed by a single database scan.

For each batch we used the RHA algorithm to find the top-k groups. Figure 20a plots the total cost of CGA as a function of the query granularity. At finest granularities, the distinct number of groups per batch fits in memory for a large percentage of batches. As the time-grouping becomes coarser, more groups are formed (the distinct combinations of the remaining group-by attributes increase), thus it becomes more likely that the number of groups exceed the

available memory, increasing the total cost of the algorithm. Note that the cost of CGA is much lower than that of plain RHA, due to the effective division of the problem to smaller ones that are solved separately with full utilization of the system resources. Figure 20b shows the cost breakdown per batch for each of the 18 batches in the $Q200$ query. Note that the average cost per batch drops as we proceed from the first to last one. The reason is due to the optimization of CGA we mentioned in Section 6; the score of the k-th group in the previous batches is used as initial value for ρ in RHA. As we proceed to latter batches, the score of the k-th group so far becomes larger, thus ρ becomes more effective in pruning early hash buckets at the current batch. The reason for the fluctuation among the costs at different batches is that the number of distinct groups between batches may vary significantly, affecting the corresponding bucket sizes and causing variable memory overflows among RHA executions.



(a) cost vs. number of batches (b) breakdown per batch

Fig. 20. Experiments with data clustered by group-by attributes

7. Conclusions

We studied an important data analysis operator that retrieves the k groups with the highest aggregate values. In data warehousing, this operator can be used directly by data analysts to identify important groups. It can also be used in frequent itemsets mining applications, as well as data mining tasks in information retrieval. The main challenge of this problem is to find the top-k groups in the case where the distinct number of groups exceeds the number of counters that can fit in memory. In this paper, we addressed this problem by proposing algorithms that do not rely on pre-computation or materialization, but apply on-demand retrieval of groups for ad-hoc sets of group-by attributes.

For the generic case of unordered tuples, we developed the Bucket Algorithm (BA) and the Recursive Hash Algorithm (RHA). BA adapts from an iceberg query algorithm [7] and utilizes counts of buckets to reduce the candidate size. In addition, we proposed optimizations for early terminating BA. On the other hand, RHA applies the branch-and-bound paradigm to minimize the number of buckets to be examined, until the top-k result is finalized. To improve the efficiency of RHA, we suggested two optimizations: (i) ordering the processing partitions

in an appropriate way, leading to early discovery of groups with high aggregate values, and (ii) introducing few counters for each hash partition, for refining tighter upper score bounds for partitions. Our experimental results on both synthetic and real datasets show that RHA outperforms BA and traditional RDBMS methods by wide margin in terms of data access cost.

Assuming that the tuples to be aggregated are ordered by their measure values, we proposed the multi-pass sorted access algorithm (MSA) and its write-optimized version (WMSA). MSA extends the NRA solution [6] for the memory-bounded setting by writing a temporary output table to the disk when memory becomes full. WMSA is an optimized version of MSA that exploits the available memory more effectively and reduces unnecessary disk accesses. Our experimental results show that WMSA is the most efficient method, if tuples are ordered by the measure to be aggregated and the distribution of top-k groups is skewed. However, RHA outperforms WMSA at small memory sizes or data of low skew, even though it does not take advantage of the ordering.

Next, for the case where the data are clustered or sorted based on some group-by attributes, we proposed CGA; a high-level algorithm that performs a scan and for each batch of tuples having the same values at the clustered attributes it uses an unordered method, like RHA. The top-k results of each sub-problem are progressively merged until the whole dataset is completely scanned.

We proceed to summarize our experimental findings, providing a qualitative comparison of the proposed top-k groups algorithms, depending on the data characteristics. Our results show that RHA outperforms all other alternatives for unordered data. The algorithm is very robust and adaptive to the available memory.

If the input is ordered by the measure attribute, WMSA is the best algorithm for cases with skewed data and moderate memory size. However, even for ordered input, RHA has lower access cost than WMSA at small memory sizes or data of low skew, even though it does not take advantage of the ordering. An additional advantage of RHA over WMSA is that it computes the top-k groups as well as their exact scores, as opposed to WMSA which determines only lower bounds for them.

For the case of clustered or ordered data with respect to one or more group-by attributes, CGA utilizing RHA for each batch (the most efficient algorithm) should be used. CGA performs better than plain RHA, since it takes advantage of the limited number of groups to be considered per batch, for better memory utilization and early pruning of light groups. Note that CGA can also be used in combination with WMSA, if the input is ordered primarily by some group-by attributes and secondarily by the measure attribute.

In the future, we will investigate top-k groups retrieval subject to user-defined constraints. For instance, a user may be interested in finding top-k products (with the highest sum of sales) that exist (do not exist) in another table holding valid (invalid) targets. Although such queries can be handled by a join, followed by aggregation, we believe that interesting optimizations can be developed to accelerate the retrieval process.

References

[1] Data Warehousing and OLAP: A Research-Oriented Bibliography, http://www.daniel-lemire.com/OLAP/index.html.
[2] R. Agrawal, R. Srikant, Fast Algorithms for Mining Association Rules in Large Databases, in: Proc. of VLDB, 1994.
[3] K. S. Beyer, R. Ramakrishnan, Bottom-Up Computation of Sparse and Iceberg CUBEs, in: Proc. of ACM SIGMOD, 1999.
[4] N. Bruno, L. Gravano, A. Marian, Evaluating Top-k Queries over Web-Accessible Databases, in: Proc. of ICDE, 2002.
[5] G. Cormode, S. Muthukrishnan, What's Hot and What's Not: Tracking Most Frequent Items Dynamically, in: Proc. of ACM PODS, 2003.
[6] R. Fagin, A. Lotem, M. Naor, Optimal Aggregation Algorithms for Middleware, in: Proc. of ACM PODS, 2001.
[7] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J. D. Ullman, Computing Iceberg Queries Efficiently, in: Proc. of VLDB, 1998.
[8] P. Flajolet, G. N. Martin, Probabilistic Counting Algorithms for Database Applications, Journal of Computer and System Sciences 31 (2) (1985) 182–209.
[9] G. Graefe, Query Evaluation Techniques for Large Databases, ACM Comput. Surv. 25 (2) (1993) 73–170.
[10] A. Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching, in: Proc. of ACM SIGMOD, 1984.
[11] J. Han, J. Wang, Y. Lu, P. Tzvetkov, Mining Top-K Frequent Closed Patterns without Minimum Support, in: ICDM, 2002.
[12] V. Harinarayan, A. Rajaraman, J. D. Ullman, Implementing Data Cubes Efficiently, in: Proc. of ACM SIGMOD, 1996.
[13] J. M. Hellerstein, P. J. Haas, H. J. Wang, Online Aggregation, in: Proc. of ACM SIGMOD, 1997.
[14] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, Supporting Top-k Join Queries in Relational Databases, in: Proc. of VLDB, 2003.
[15] R. Kimball, The Data Warehouse Toolkit, John Wiley, 1996.
[16] I. Lazaridis, S. Mehrotra, Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure, in: Proc. of ACM SIGMOD, 2001.
[17] K. P. Leela, P. M. Tolani, J. R. Haritsa, On Incorporating Iceberg Queries in Query Processors, in: Proc. of DASFAA, 2004.
[18] C. Li, K. C.-C. Chang, I. F. Ilyas, Supporting Ad-hoc Ranking Aggregates, in: Proc. of ACM SIGMOD, 2006.
[19] Z. X. Loh, T. W. Ling, C.-H. Ang, S. Y. Lee, Analysis of Pre-computed Partition Top Method for Range Top-k Queries in OLAP Data Cubes, in: Proc. of CIKM, 2002.
[20] N. Mamoulis, S. Bakiras, P. Kalnis, Evaluation of Top-k OLAP Queries Using Aggregate R-trees, in: Proc. of SSTD, 2005.
[21] N. Mamoulis, M. L. Yiu, K. H. Cheng, D. W. Cheung, Efficient Top-k Aggregation of Ranked Inputs, ACM TODS 32 (3) (2007) 19.
[22] A. Metwally, D. Agrawal, A. E. Abbadi, Efficient Computation of Frequent and Top-k Elements in Data Streams, in: Proc. of ICDT, 2005.
[23] V. Paxson, S. Floyd, Wide-Area Traffic: The Failure of Poisson Modeling, IEEE/ACM Transactions on Networking 3 (3) (1995) 226–244.