Supporting Ad-hoc Ranking Aggregates*

Chengkai Li
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801
cli@uiuc.edu

Kevin Chen-Chuan Chang
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801
kcchang@cs.uiuc.edu

Ihab F. Ilyas
School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada N2L 3G1
ilyas@uwaterloo.ca

ABSTRACT

This paper presents a principled framework for efficient processing of ad-hoc *top-k* (ranking) aggregate queries, which provide the k groups with the highest aggregates as results. Essential support of such queries is lacking in current systems, which process the queries in a naïve materialize-group-sort scheme that can be prohibitively inefficient. Our framework is based on three fundamental principles. The Upper-Bound Principle dictates the requirements of early pruning, and the Group-Ranking and Tuple-Ranking Principles dictate group-ordering and tuple-ordering requirements. They together guide the query processor toward a *provably optimal* tuple schedule for aggregate query processing. We propose a new execution framework to apply the principles and requirements. We address the challenges in realizing the framework and implementing new query operators, enabling efficient group-aware and rank-aware query plans. The experimental study validates our framework by demonstrating orders of magnitude performance improvement in the new query plans, compared with the traditional plans.

1. INTRODUCTION

Aggregation is a key operation in OLAP (On-Line Analytical Processing), and dominates a variety of decision support applications such as manufacturing, sales, stock analysis, network monitoring, *etc*. In aggregate queries, aggregates are computed over groups by some grouping attributes. As decision making naturally involves comparing and ranking data, among the large number of groups, often only the ones with certain significance are of interest. To support such applications, ranking (*top-k*) aggregate queries rank the groups by their aggregate values and return the top k groups with the highest aggregates. Further, decision makers often need to specify analysis queries in an *ad-hoc* manner, with respect to how the data is aggregated and ranked. Such ad-hoc ranking criteria fit very well to the exploratory nature of decision support and enable flexible and expressive data analysis. Moreover, decision support queries are commonly executed in an *interactive* environment where the results must be quickly presented to users and become the basis of further queries. *Efficient* processing of ad-hoc *top-k* aggregate queries is thus crucial.

However, such efficient support of ad-hoc ranking is critically lacking in current decision support systems. Most OLAP query processing focuses on *pre-computation* and *full-answer*. Such systems maintain pre-computed simple aggregates that can only benefit queries over these aggregates. Moreover, the notions of ranking and its optimization are missing as current systems always provide full answers and do not optimize for the small retrieval size k. This paper thus aims at supporting efficient ad-hoc ranking aggregates.

1.1 Query Model and Motivating Examples

Below is a SQL-like template for expressing *top-k* aggregate queries and an example query Q. While we use LIMIT, various RDBMS use different SQL syntax to specify k.

SELECT	$ga_1, ..., ga_m, F$
FROM	$R_1, ..., R_h$
WHERE	c_1 AND ... AND c_l
GROUP BY	$ga_1, ..., ga_m$
ORDER BY	F
LIMIT	k

```
Q:SELECT A.g, B.g, C.g, SUM(A.v
          B.v+C.v) as score
  FROM  A, B, C
  WHERE A.jc=B.jc AND B.jc=C.jc
  GROUP BY A.g, B.g, C.g
  ORDER BY score
  LIMIT k
```

That is, the groups are ordered by a *ranking aggregate* F=$G(T)$, where G is an *aggregate function* (*e.g.*, sum) over an *expression* T on the table columns (*e.g.*, $A.v$+$B.v$+$C.v$). The top k groups with the highest F values are returned as the query result. Formally, each group g=$\{t_1, \ldots, t_n\}$ has a *ranking score* $F[g]$, defined as

$$\begin{aligned} F[g] &= G(T)[g] = G(T[g]) = G(T[\{t_1, \ldots, t_n\}]) \\ &= G(\{T[t_1], \ldots, T[t_n]\}). \end{aligned} \quad (1)$$

As the result, Q returns a sorted list $\mathcal{K}$ of k groups, ranked by their scores according to F, such that $F[g] \geq F[g']$, $\forall g \in \mathcal{K}$ and $\forall g' \notin \mathcal{K}$. When there are ties in scores, an arbitrary *deterministic* "tie-breaker" function can be used to determine an order, *e.g.*, by the grouping attribute values of each group.

A distinguishing goal of our work is to support *ad-hoc* ranking aggregate criteria. With respect to G, we aim to support not only standard (*e.g.*, sum, avg), but also user-defined aggregate functions. With respect to the aggregated expression T, we allow T to be any expression, from simply a table column to very complex formulas. Below we show some motivating examples.

Example 1 (Motivating Queries):

```
Q1:
SELECT   zipcode, AVG(income*w1+age*w2+credit*w3) as score
FROM     customer
WHERE    occupation='student'
GROUP BY zipcode
ORDER BY score
LIMIT    5
```

*This material is based upon work partially supported by NSF Grants IIS-0133199, IIS-0313260, the 2004 and 2005 IBM Faculty Awards, Software Telecommunications Group, University of Waterloo and NSERC Discovery Grant 311671-05. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

```
Q2:
SELECT   P.category, S.zipcode,
         MID_SUM(S.price-P.manufacturer_price) as score
FROM     part as P, sales as S
WHERE    P.part_key=S.part_key
GROUP BY P.category, S.zipcode
ORDER BY score
LIMIT    5
```

The above query, $Q1$, returns the best 5 areas to advertise a student insurance product, according to the average customer score of each area. The score indicates how likely a customer will buy the insurance. A manager can explore various ways in computing the score, according to her analysis. For example, a weighted average of customer's income, age, and credit is used in $Q1$. Query $Q2$ finds the 5 best matches of part category and sales area that generate the highest profits. A pair of category and area is evaluated by aggregating the profits of all sales records in that category and area. A user-defined aggregate function mid_sum is used to accommodate flexible metrics in such evaluation. For example, it can remove the top and bottom 5% (with respect to profit) sales records within each group and sum up the rest, to reduce the impact of outliers. ■

We emphasize that such ad-hoc aggregate queries often run in sessions, where users execute related queries with similar Boolean conditions but different ranking criteria, for exploratory and interactive data analysis. For example, in the above $Q1$, the manager can try various aggregate function and/or many combinations of the values of w_1, w_2, and w_3 until an appropriate ranking criterion is determined. Moreover, such related queries also exist across different sessions of decision support tasks over the same data.

In this paper, we concentrate on a special but large class of aggregate queries $F = G(T)$, where the aggregate function G satisfies what we refer to as the *max-bounded* property: *An upper-bound of the aggregate F over a group g, denoted by $\overline{F}_g$, can be obtained by applying G to the maximum values of the member tuples in g.* The class of max-bounded functions include many typical aggregate functions such as sum, $weighted\,average$, *etc.*, as well as user-defined aggregate functions such as the mid_sum in query $Q2$ above. In fact, we believe that most ranking aggregate queries will use functions that satisfy this property.

1.2 Limitations of Current Techniques

A popular conceptual data model for OLAP is *data cube* [16]. A data cube is derived from a *fact table* consisting of a *measure attribute* and a set of *dimensional attributes* that connect to the *dimension tables*. A cube consists of a lattice of cuboids, where each cuboid corresponds to the aggregate of the measure attribute according to a Group-By over a subset of the dimensional attributes. With respect to various measures and dimensions, multiple cubes may be necessary. As a conceptual model, data cube is seldom fully materialized given its huge size. Instead, in ROLAP servers, many materialized views (or *summary tables*) are selected to be built to cover certain tables and attributes for answering aggregate queries [21]. Pervasive summary and index structures are further built upon the base tables and materialized views.

Many works studied the problem of answering aggregate queries using views [17, 32, 11, 1, 28]. They provide significant performance improvement when appropriate materialized views for the given query are available. However, they cannot answer ad-hoc ranking aggregate queries. Materialized views only maintain information of the *pre-determined* attribute or expression using the *prescribed* aggregate function. In contrast, ad-hoc ranking conditions are determined or defined *on-the-fly* during decision making. Therefore in order to answer a ranking aggregate F=$G(T)$, G must be the aggregate function used when the cubes (views) are materialized or can be derived from the materialized aggregate function (*e.g.*, avg derived from sum and $count$), and T must happen to be simply some measure attribute or expression that can be derived from the summary tables, instead of arbitrarily complex expression. Given the virtually infinite choices of G and T in ad-hoc data analysis, the pre-computed information easily become irrelevant when the query is different from what the summary tables are built for.

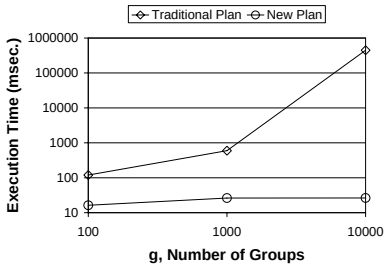


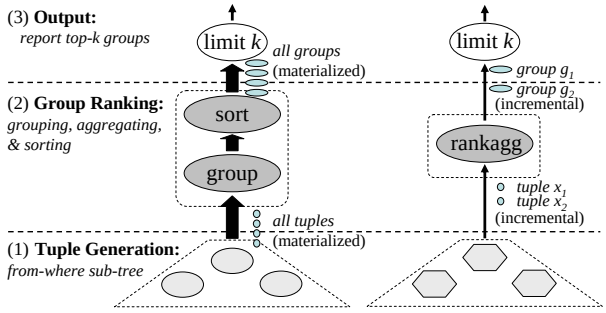
Figure 1: Traditional plan vs. new plan.

When pre-computed cubes cannot answer the query, the processing has to fall back to base tables, where the query is evaluated by the relational query engine of the ROLAP server as follows: (1) fully consume all the input tables; (2) fully materialize the selection and join results; (3) group the results by grouping attributes and compute the aggregates for every group; (4) fully sort the groups by their ranking aggregates; and (5) report only the top k groups. The user is only interested in the k top groups instead of a total order on all groups. The traditional processing strategy is thus an overkill, with unnecessary overhead from full scanning, joining, grouping, and sorting. Given the large amount of data in a warehousing environment, such a naïve *materialize-group-sort* scheme can be unacceptably inefficient. Moreover, the users may have to go through it many times in their exploratory and interactive analysis tasks. Such inefficiency thus significantly impacts the usefulness of decision support applications, resulting in low productivity.

1.3 Contributions

In this paper, we propose a principled framework for efficient processing of ad-hoc *top-k* aggregate queries. We define a cost metric on the number of "consumed" tuples, capturing our goal of producing only *necessary* tuples for generating top k groups. We identify the best-possible goal in terms of this metric that can be achieved by any algorithm, as well as the must-have information for achieving the goal. The key in realizing this goal is to find some good *order* of producing tuples (among many possible orders) that can guide the query engine toward processing the most promising groups first, and exploring a group only when necessary. We further discover that a *provably optimal* total schedule of tuples can be fully determined by two orders– the order of retrieving groups (*group ordering*) and the order of retrieving tuples (*tuple ordering*) within each group. Based on this view, we develop three fundamental principles and a new execution framework for processing *top-k* aggregate queries. We summarize our contributions as follows:

- **Principle for optimal aggregate processing:** We develop three properties, the *Upper-Bound*, *Group-Ranking* and *Tuple-Ranking Principles* that lead to the *exact-bounding*, *group-ordering* and *tuple-ordering requirements*, respectively. We formally show that the optimal aggregate query processing, with respect to our cost metric, can be derived by following these requirements.
- **Execution framework and implementations:** Guided by the principles, we propose a new execution framework, which enables query plans that are both *group-aware* and *rank-aware*. We further address the challenges of applying the principles and implementing the new query operators in this framework.



(a) Traditional query plan. (b) New query plan.

Figure 2: Top-k aggregates query processing.

- **Experimental Study:** We implement the proposed techniques in PostgreSQL. The experiments verify that our techniques can achieve orders of magnitude improvement in performance; *e.g.*, Figure 1 compares a new query plan with the traditional plan.

The rest of the paper is organized as follows. We present the principles in Section 2. Section 3 introduces the execution framework of *top-*k aggregate query plans and the implementations of physical operators. We experimentally evaluate the proposed framework in Section 4. Section 5 reviews related work, and Section 6 concludes.

2. PRINCIPLES: OPTIMAL AGGREGATE PROCESSING

Efficient support of ranking aggregate queries is critically lacking in current systems. To motivate, Figure 2(a) illustrates the traditional materialize-group-sort query plan, consisting of three components: 1) *tuple generation*: the from-where subtree for producing the member tuples of every group; 2) *group ranking*: the *group* and *sort* operators for generating the groups and ranking them; and 3) *output*: the *limit* operator returning the top-k groups. As Section 1 discussed, this approach fully materializes and aggregates all tuples, and then fully materializes and sorts all groups. Since only top-k groups are requested, much of the effort is simply wasted.

Our goal is thus to design a new execution model, as Figure 2(b) contrasts. We need a new non-blocking *rankagg* operator, which *incrementally* draws tuples, as its input from the underlying subtree, and generates top groups in the ranking order, as its output. For efficiency, *rankagg* must minimize its consumption of input tuples: Although in practice the cost formula can be quite complex with many parameters, this input cardinality (*i.e.*, number of tuples consumed) is always an essential factor. As our metric, for a group g with n tuples $\{t_1, \ldots, t_n\}$, how "deep" into the group shall we process, for determining the top-k groups? We refer to this number of tuples consumed for g as its *tuple depth*, denoted H_g. Our goal is thus to minimize the total cost of all groups, *i.e.*, $\Sigma_g H_g$.

As the foundation of our work, while the new *rankagg* can be implemented in different ways, what are the requirements and guidelines for *any* such algorithm? To minimize tuple consumption (*i.e.*, to stop processing and to prune the groups early), what information must we have and what is the criterion in such pruning? As tuples flow from the underlying subtree, in what *order* shall *rankagg* request and process tuples? Is there an optimal *tuple schedule* that minimizes the total tuple depths? We develop three fundamental principles for determining *provably optimal* tuple schedule (Theorem 1) that achieves the theoretical minimal tuple consumption: the Upper-Bound Principle for early pruning, the Group-Ranking Principle for asserting "inter-group" ordering, and the Tuple-Ranking Principle for further deciding "intra-group" ordering. These principles guide our implementation of *rankagg* and determine its impact to the underlying query tree. (The subtrees for tuple generation in Figure 2(a) and (b) use different shapes to emphasize that *rankagg* requires modifying the underlying operators such as scan and join.)

TID	R.g	R.v
r_1	1	.7
r_2	2	.3
r_3	3	.9
r_4	2	.4
r_5	1	.9
r_6	3	.7
r_7	1	.6
r_8	2	.25

(a) Relation R.

	Group g_2	Group g'_2
	TID R.v	TID R.v
	r_4 .4	r_4 1.0
	r_2 .3	r_2 .8
	r_8 .25	r_8 .0
desc.	$3 \xrightarrow{r_4} 1.2 \xrightarrow{r_2} 1 \xrightarrow{r_8} .95$	$3 \xrightarrow{r_4} 3 \xrightarrow{r_2} 2.6 \xrightarrow{r_8} 1.8$
asc.	$3 \xrightarrow{r_8} 2.25 \xrightarrow{r_2} 1.55 \xrightarrow{r_4} .95$	$3 \xrightarrow{r_8} 2 \xrightarrow{r_2} 1.8 \xrightarrow{r_4} 1.8$
hybrid	$3 \xrightarrow{r_8} 2.25 \xrightarrow{r_4} 1.05 \xrightarrow{r_2} .95$	$3 \xrightarrow{r_4} 3 \xrightarrow{r_8} 2 \xrightarrow{r_2} 1.8$
random	$3 \xrightarrow{r_2} 2.3 \xrightarrow{r_4} 1.7 \xrightarrow{r_8} .95$	$3 \xrightarrow{r_2} 2.8 \xrightarrow{r_8} 1.8 \xrightarrow{r_4} 1.8$

(b) Example tuple orders for *Group* g_2.

Figure 3: Relation R and some tuple orders.

The following query is our running example. The input relation R, with two attributes $R.g$ and $R.v$, is shown in Figure 3(a). The query groups by $R.g$, and our following discussion refers to those $R.g$=1, 2, and 3 as group g_1, g_2, and g_3, respectively. The query specifies a ranking aggregate function F= $G(T)$=$sum(R.v)$ and k=1. Throughout this paper, we assume T is in the range of $[0, 1]$.

Select $R.g$, **SUM**($R.v$) **From** R
Group By $R.g$ **Order By SUM**($R.v$) **Limit** 1

2.1 Upper-Bound Principle

Our first principle deals with the requirements of early pruning: what information *must* we have in order to prune? During processing, before a group g is fully evaluated, the obtained tuples of g can effectively bound its ultimate aggregate score. For a ranking aggregate F= $G(T)$, we define $\overline{F}_{\mathcal{I}_g}[g]$, the *maximal possible score* of g, with respect to a set $\mathcal{I}_g$ of obtained tuples ($\mathcal{I}_g \subseteq g$), as the upper-bound score that g may eventually achieve, *i.e.*, $F[g] \leq \overline{F}_{\mathcal{I}_g}[g]$.

The upper-bound score of a group thus indicates the best the group can achieve. For our discussion, call the lowest top-k score of the query as the *threshold*, denoted θ. (For instance, θ=$F[g_1]$=2.2 in our running example.) Note that θ would not be known before the processing ends. Given a group g, if its upper-bound score is higher than or equal to θ, it has a chance to make into the top k groups. To conclude that g does not belong to the top k and thus prune it from further processing, the upper-bound score of g must be below θ, otherwise we may incorrectly prune g that indeed belongs to the top k. Therefore the upper-bound score decides the minimal number of tuples that any algorithm (that processes by obtaining tuples) must obtain from g before it can prune g. As stated in the following property 1, this minimal tuple depth is the *best-possible goal* of any algorithm, due to that pruning a group with less obtained tuples can result in wrong query answers.

For the properties and theorems in this paper, we leave the details of proofs in an extended version [25] of the paper and only provide intuitive justification, as above.

Property 1 (Best-Possible Goal): With respect to a ranking aggregate F= $G(T)$, let the lowest top-k group score be θ. For any group g, let H_g^{min} be its minimal tuple depth, *i.e.*, the number of tuples to retrieve from g before it can be pruned from further processing, or otherwise determined to be in the *top-*k groups. The H_g^{min} is the smallest number of tuples from g that makes the maximal possible score of g to be below θ, *i.e.*,

$$H_g^{min} = \min\{|\mathcal{I}_g| | \overline{F}_{\mathcal{I}_g}[g] < \theta, \mathcal{I}_g \subseteq g\}, \quad (2)$$

or otherwise H_g^{min}=$|g|$ if such a depth does not exist. ■

We emphasize that, as Property 1 implies, an algorithm must have certain information about upper-bound score for pruning. Various algorithms may exploit various ways in computing $\overline{F}_{\mathcal{I}_g}[g]$, resulting in different pruning power, *i.e.*, different H_g^{min}. For an algorithm that has no knowledge to derive a non-trivial upper-bound, $\overline{F}_{\mathcal{I}_g}[g]$ would be in its most trivial form, that is, infinity. Such a trivial upper-bound cannot realize any pruning at all, since the upper-bound score would never go below θ.

Property 2 (Must-Have Information): For any group g, with a trivial upper-bound $\overline{F}_{\mathcal{I}_g}[g]= +\infty$ under every $\mathcal{I}_g$, $H_g^{min}=|g|$. ■

Therefore we must look for a non-trivial definition of $\overline{F}_{\mathcal{I}_g}[g]$ in order to prune. Since we focus on aggregate functions G that are max-bounded (which describes a wide class of functions, as Section 1 defined), the maximal-possible score can be obtained by Eq. 3 below, which simply substitutes unknown tuple scores with their maximal value of T, denoted by $\overline{T}_{\mathcal{I}_g}$.

$$\overline{F}_{\mathcal{I}_g}[g] = G\left(\{T_i | \begin{matrix} T_i = T[t_i] & \text{if } t_i \in \mathcal{I}_g \text{ (\textit{seen tuples})}; \\ T_i = \overline{T}_{\mathcal{I}_g} & \text{otherwise (\textit{unseen tuples}).} \end{matrix} \forall t_i \in g\}\right). \quad (3)$$

Example 2 (Maximal-Possible Scores): Consider our running example F=$G(T)$=$sum(R.v)$. Suppose the *rankagg* operator has processed tuples r_1 and r_2; *i.e.*, $\mathcal{I}_{g_1}$=$\{r_1\}$, $\mathcal{I}_{g_2}$=$\{r_2\}$, and $\mathcal{I}_{g_3}$=ϕ. Suppose the exact size of every group is known a priori. As g_1 has seen only r_1 with $T[r_1]$=.7 and the two unseen tuples (its count is 3) can score as high as 1.0, $F[g_1]$ can aggregate to $F[g_1]\leq sum(.7+1.0\times2)$=2.7, or $\overline{F}_{\mathcal{I}_{g_1}}[g_1]$=2.7. Similarly, $\overline{F}_{\mathcal{I}_{g_2}}[g_2]$=$sum(.3+1.0\times2)$=2.3; $\overline{F}_{\mathcal{I}_{g_3}}[g_3]$=$sum(1.0\times2)$=2.0. ■

Note that Eq. 3 requires to know $\overline{T}_{\mathcal{I}_g}$ and the count (size) of a group, or at least an upper-bound of this count, to constraint the number of unknown tuples. (For example, if 4 is used as the upper-bound of g_1's size in Example 2, $F[g_1]\leq sum(.7+1.0\times3)$=3.7.) We refer to these values as the "*grouping bounds*", consisting of *tuple count* ($\overline{|g|}$, the upper-bound of g's size) and *tuple max* ($\overline{T}_{\mathcal{I}_g}$).

Eq. 3 captures a class of definitions of maximal-possible score, as different ways can be explored in getting the grouping bounds, resulting in different $\overline{F}_{\mathcal{I}_g}[g]$ and thus different H_g^{min}. For instance, using infinity to bound the tuple count or the tuple max results in $\overline{F}_{\mathcal{I}_g}[g]$ as infinity, with no pruning power. Given Eq. 3, for any group g, the smaller $\overline{|g|}$ and $\overline{T}_{\mathcal{I}_g}$, the smaller $\overline{F}_{\mathcal{I}_g}[g]$. Therefore the most pruning power, *i.e.*, the smallest H_g^{min}, is realized by the exact group size and the exact highest T, as stated below.

Requirement 1 (Exact Bounding): With respect to a ranking aggregate F= $G(T)$, let the lowest top-k group score be θ. Given the definition of $\overline{F}_{\mathcal{I}_g}[g]$ in Eq. 3, to obtain the smallest H_g^{min}, we must use $\overline{|g|}$=$|g|$ and $\overline{T}_{\mathcal{I}_g}$=max$\{T[t_i] \mid t_i \in g - \mathcal{I}_g\}$. ■

Based on Requirement 1, our implementation choice of grouping bounds is the exact count of a group as $\overline{|g|}$ and a value very close to max$\{T[t_i] \mid t_i \in g - \mathcal{I}_g\}$ as $\overline{T}_{\mathcal{I}_g}$. We justify this choice and show how to obtain such grouping bounds in Section 3.1.1. Note that our discussion of the following principles is orthogonal to the choices of grouping bounds, which only result in different best-possible tuple depth H_g^{min} that our algorithm sets to achieve.

2.2 Group-Ranking Principle

Property 1 gives the minimal tuple depth H_g^{min} for each g, thus the minimal total cost $\Sigma_g H_g^{min}$. The essence of Eq. 2 lies in that we should stop processing a group as soon as it can be excluded from top-k answers. That is, we should only further process a group if it is *proven* to be absolutely necessary, *i.e.*, its upper-bound score above the threshold θ. While Eq. 2 hints on such "necessity", it does not suggest how to determine the necessity, because θ can only be known at the conclusion of a query. Therefore we wonder, as an algorithm retrieves tuples one by one, is there an optimal tuple schedule that achieves the minimum depth?

A schedule is determined by *inter-group* and *intra-group* ordering. Our Group-Ranking principle asserts inter-group ordering: When selecting the next tuple t to process, how to order *between* groups? Which group should t be selected from? (While this work defines such insight of "branch-and-bounding" for aggregate queries for the first time, similar intuition has also been applied to ordering *individual* tuples [6, 4, 26] in top-k queries.) Thus the *Group-Ranking Principle* builds upon the basis that groups with higher bounds must be processed further before others.

Such bounds guide our selection of the next tuple. Let's illustrate with Example 2: The next tuple *should* be selected from g_1. Consider g_1 vs. g_2 (and similarly g_3). If g_1 will be the top-1, we must complete its score. Otherwise, since $\overline{F}_{\mathcal{I}_{g_1}}[g_1] > \overline{F}_{\mathcal{I}_{g_2}}[g_2]$, whatever score g_2 can achieve, g_1 can possibly do better. Thus, first, although g_2 is incomplete, it may not be *necessary* for further processing, since g_1 may turn out to be the answer (*i.e.*, g_1 should be processed before g_2). Second, even if g_2 were complete, it is not *sufficient* to declare g_2 as the top-1, since g_1 may be a better answer. In all cases, we must process the next tuple from g_1.

The above explanation intuitively motivates the priority between g_1 and g_2, for the special case when k=1. The Group-Ranking Principle formally states this property, for general top-k ($k \geq 1$) situations (as aforementioned, proof in [25]), which mandates the priority of current top k groups (*i.e.*, g_1) over others (*i.e.*, g_2).

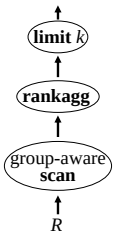
Property 3 (Group-Ranking Principle): Let g_1 be any group in the current top-k ranked by maximal-possible scores $\overline{F}$ and g_2 be any group not in the current top-k. We have 1) g_1 must be further processed if g_1 is not fully evaluated, 2) it may not be necessary to further process g_2 even if g_2 is not fully evaluated, and 3) the current *top-k* are the answers if they are all fully evaluated. ■

The Group-Ranking Principle guides our inter-group ordering for query processing, by prioritizing on $\overline{F}$. Essentially, the principle states that, to avoid unnecessary tuple evaluations, our algorithms must prioritize any incomplete g_1 within the current top-k over those g_2 outside. Thus, *first*, as the *progressive* condition, to reach the final top-k, any such g_1 must be further processed (or else there are no enough k complete groups to conclude as better than g_1). *Second*, as the *stop* condition, when and only when no such g_1 exists, *i.e.*, all top-k groups are completed, we can conclude these groups as the final answers. Below we summarize this requirement.

Requirement 2 (Group Ordering): To avoid the unnecessary tuple consumption, query processing should prioritize groups by their maximal-possible score $\overline{F}$:

- (**Progressive Condition**) If there are some incomplete groups g_1 in the top-k, then the next tuple should be selected from such g_1;
- (**Stop Condition**) Otherwise, we can stop and conclude the current top-k groups as the final answers. ■

Example 3 (Sample Execution 1): For our example F= $G(T)$ = $sum(R.v)$, to find the top-1 group, Figure 4(b) conceptually executes Requirement 2. (We discuss the corresponding Figure 4(a) in Section 3.) We prioritize groups by $\overline{F}$ scores, initially (3.0, 3.0, 2.0), when no tuples in any group g are seen ($\mathcal{I}_g$=ϕ) and thus



step	action	$\overline{F}[g_1]$	$\overline{F}[g_2]$	$\overline{F}[g_3]$
0.	initial	3.0*	3.0	2.0
1.	(r_1, 1, .7)	2.7	3.0*	2.0
2.	(r_2, 2, .3)	2.7*	2.3	2.0
3.	(r_5, 1, .9)	2.6*	2.3	2.0
4.	(r_7, 1, .6)	**2.2**	2.3*	2.0
5.	(r_4, 2, .4)	**2.2***	1.7	2.0

S*: *Top-1 score* **S**: *Final score*

(a) Query plan. (b) Tuple flow to **rankagg**.

Figure 4: Query execution 1: GroupOnly.

step	action	$\overline{F}[g_1]$	$\overline{F}[g_2]$	$\overline{F}[g_3]$
0.	initial	3.0*	3.0	2.0
1.	(r_5, 1, .9)	2.7	3.0*	2.0
2.	(r_4, 2, .4)	2.7*	1.2	2.0
3.	(r_1, 1, .7)	2.3*	1.2	2.0
4.	(r_7, 1, .6)	**2.2***	1.2	2.0

S*: *Top-1 score* **S**: *Final score*

(a) Query plan. (b) Tuple flow to **rankagg**.

Figure 5: Query execution 2: GroupRank.

$\overline{T}_{\mathcal{I}_g}$=1.0 in Equation 3. As the *Progressive Condition* dictates, we always choose the top-1 group (marked *) for the next tuple, thus accessing r_1 from g_1, r_2 from g_2, ..., and finally r_4 from g_2. Now, since the top-1 group g_1 is completed (with final score $F[g_1]$=$\overline{F}[g_1]$=2.2), the *Stop Condition* asserts no more processing necessary, and thus we return g_1 as the top-1. ■

2.3 Tuple-Ranking Principle

Our last principle addresses the *intra-group* order: When we must necessarily process group g (as the Group-Ranking Principle dictates), which tuple in g should we select? This tuple ordering, together with the group ordering just discussed, will determine a total schedule of tuple access for the *rankagg* operator (Figure 2(b)).

To start with, we note that different tuple orders result in different cost efficiency, in terms of tuple depth of each group. Given a tuple order α for group g, what would be the resulting tuple depth H_g^α that must be accessed? Recall in Example 3 we order tuples arbitrarily by tuple IDs (see relation R in Figure 3(a)), *i.e.*, group g_1 as x_1:$r_1 \rightarrow r_5 \rightarrow r_7$, g_2 as x_2:$r_2 \rightarrow r_4 \rightarrow r_8$, and g_3 as x_3:$r_3 \rightarrow r_6$. These orders result in depths $H_{g_1}^{x_1}$=3 (*i.e.*, all of r_1, r_5, r_7 accessed), $H_{g_2}^{x_2}$=2, $H_{g_3}^{x_3}$=0, as Figure 4(b) shows. To contrast, Example 4 below shows how different tuple orders result in different depths.

Example 4 (Sample Execution 2): Rerun Example 3 but with tuple orders as sorted by tuple scores T=$R.v$ in each group, thus ordering g_1 as d_1:$r_5 \rightarrow r_1 \rightarrow r_7$, g_2 as d_2:$r_4 \rightarrow r_2 \rightarrow r_8$, and g_3 as d_3:$r_3 \rightarrow r_6$. These descending orders, together with Requirement 2, result in the execution of Figure 5(b). (Again, Figure 5(a) is discussed in Section 3.) Note that, for each group, the *descending* order sorted by T effectively bounds the T-score of unseen tuples by the *last-seen* T-score. Thus, for group g_1, after r_5 at step 1 with $T[r_5]$=$r_5.v$=.9, $\overline{F}[g_1]$=0.9+0.9×2 (for 2 unseen tuples)=2.7. Then, after r_1 in step 3, $\overline{F}[g_1]$=0.9+0.7+0.7×1 (for 1 unseen tuple) = 2.3. In this execution, each group is accessed to depth $H_{g_1}^{d_1}$=3, $H_{g_2}^{d_2}$=1, and $H_{g_3}^{d_3}$=0. In particular, group g_2 has a depth $H_{g_2}^{x_2}$ = 2 and $H_{g_2}^{d_2}$ = 1 (out of 3), and thus d_2 is a better order than x_2. ■

To minimize the total costs (as the sum of tuple depths), how do we find the optimal order α for each group g such that H_g^α=H_g^{min}? Apparently, this "space" of orders seems prohibitively large: If there are n tuples in each of the m groups, as each group has $n!$ permutations, there will be $(n!)^m$ different orders. Thus, our Property 4, or the *Tuple-Ranking Principle*, addresses this tuple ordering issue. It has two main results:

First, *order independence*: To find the optimal orders, shall we consider the *combinations* among the orders of *different* groups? It turns out that, *if* we follow Requirement 2 for group ordering, the optimal tuple order of each group is independent of all others. That is, the tuple depth of a group g depends on only its *own* order α.

To see why, let's consider g_2 in Figure 4 and 5. As Requirement 2 dictates, by the Progressive Condition, we only necessarily access a next tuple from the group, *when* and *only when* $\overline{F}[g_2]$ remains in the top-k (k=1 in this case). The execution halts, as the Stop Condition asserts, when the top-k groups are completed and "surfaced" to the top, at which point $\overline{F}[g_2]<\theta$ and thus no longer needs further processing. Thus, in Figure 4, with tuple order x_2:$r_2 \rightarrow r_4 \rightarrow r_8$, $\overline{F}[g_2]$ progressively lowers its upper bounds as $3.0 \xrightarrow{r_2} 2.3 \xrightarrow{r_4} 1.7$, at which point it stops, because $1.7<2.2$, or $1.7<\theta$. To contrast, in Figure 5, the tuple order d_2 results in $3.0 \xrightarrow{r_4} 1.2$, where it stops as $1.2<\theta$. While the different orders result in different depths $H_{g_2}^{x_2}$=2 and $H_{g_2}^{d_2}$=1, both are the "smallest" depths (under the respective orders) that make $\overline{F}[g_2]$ go below θ– which are dependent on only the tuple order of g_2 and independent of others.

Second, *T-based Ranking*: While groups are independent, for each group, what orders α as $t_1 \rightarrow \cdots \rightarrow t_n$, out of the $n!$ permutations (for a group of n tuples), should we consider? As just explained above, a better order (*e.g.*, d_2 vs. x_2) of g will decrease the upper bounds $\overline{F}[g]$ more rapidly to go below θ with less tuple accesses. What orders can achieve such rapid decreasing?

As the upper bounds $\overline{F}$ are defined by Eq. 3, the answer naturally lies there. There are two components in the equation: 1) the scores $T[t_i]$ of the *seen* tuples t_i in $\mathcal{I}_g$, and 2) the upper bound $\overline{T}_{\mathcal{I}_g}$ of the *unseen* tuples. Intuitively, *first*, a good order can lower the seen scores, by accessing tuples with the smallest T. *Second*, it can also lower the upper bounds of those unseen, by retrieving tuples from high T to low, where the unseen are bounded by the last-seen tuple (as in Example 4). Following this intuition, we only need to consider *T-desc/asc*, a class of orders that always choose either the highest or the lowest from the unseen tuples as the next. That is, any other order must be inferior to some order in this class.

Property 4 formalizes the two results (proof in [25]), with our intuitive explanation above.

Property 4 (Tuple-Ranking Principle): With respect to a ranking aggregate F=$G(T)$, let the lowest top-k group score be θ. For any group g, let H_g^α be the tuple depth with respect to tuple order α:$t_1 \rightarrow \cdots \rightarrow t_n$, when the inter-group ordering follows Requirement 2.

- (**Order Independence**) The depth H_g^α depends on only α (the order of this group) and θ (the global threshold), and not on the order of other groups. Specifically, H_g^α is the smallest depth l of sequence α that makes the the maximal possible score of g to be below θ, *i.e.*,

$$H_g^\alpha = \min_{l \in [1:n]} \{l | \overline{F}_{\{t_1,\ldots,t_l\}}[g] < \theta\}, \quad (4)$$

  or otherwise H_g^α=n if such a depth does not exist.
- (**T-based Ranking**) To find the optimal order α that results in the minimum H_g^α, *i.e.*, H_g^α=H_g^{min}, we only need to consider the class of orders *T-desc/asc* =

$$\{\alpha : t_1 \rightarrow \cdots \rightarrow t_n | \begin{array}{l}\text{either } T[t_i] \geq T[t_j] \forall j > i \text{ (from top);} \\ \text{or } \quad T[t_i] \leq T[t_j] \forall j > i \text{ (from bottom).}\end{array} \forall t_i\}. \quad (5)$$

■

To conclude, we summarize the implementation implications of the Tuple-Ranking Principle as Requirement 3, which guides our design of a processing model for finding the optimal tuple ordering.

Requirement 3 (Tuple Ordering): If Requirement 2 is followed, to minimize the total tuple depths across all groups: 1) the order of each group can be optimized independently; and 2) the optimal order is one from T-*desc/asc*, that results in the minimum H_g^α as governed by Eq. 4. ∎

2.4 Putting Together: Overall Optimality

Together, the Upper-Bound Principle dictates the best-possible goal and the must-have information (the maximal-possible score) in early pruning for any algorithm; based on the maximal-possible score, the Group-Ranking and Tuple-Ranking Principles guide the tuple scheduling for our *rankagg* operator to selectively draw from the underlying query tree (Figure 2(b)). We stress that, as the following Theorem 1 states, the Group-Ranking and Tuple-Ranking Principles enable the finding of an *optimal* tuple schedule which processes every group minimally, thus achieving an overall minimum tuple depth $\Sigma_g H_g^{min}$ (*i.e.*, the best-possible goal) with respect to some upper-bound mechanism $\overline{F}_{\mathcal{I}_g}[g]$. While we provide proof in [25], we note that Requirement 2 determines an inter-group order that only accesses a group when *necessary*, and Requirement 3 further leads to a "cost-based" optimal intra-group order for each group, with a significantly reduced space of only T-desc/asc orders.

Theorem 1 (Optimal Aggregate Processing): If query processing follows Requirements 2 and 3, the number of tuples processed across all groups, *i.e.*, $\Sigma_g H_g$, is the minimum possible for query answering, *i.e.*, $\Sigma_g H_g^{min}$. ∎

3. EXECUTION FRAMEWORK AND IMPLEMENTATIONS

The principles developed in Section 2 provide a guideline in realizing the new model of execution plans in Figure 2(b). In this section, we propose an execution framework for applying the principles (Section 3.1). We address the challenges in implementing the new *rankagg* operator (Section 3.2) and discuss its impacts to the existing operators (Section 3.3).

3.1 The Execution Framework

We design a framework to apply the principles. The framework consists of two orthogonal components. The first component provides the grouping bounds, which define the maximal possible score $\overline{F}_{\mathcal{I}_g}[g]$, the must-have information according to the Upper-Bound Principle. The second component schedules tuple processing based on the grouping bounds by exploiting the Group-Ranking and Tuple-Ranking Principles. The two components are orthogonal because the Grouping-Ranking and Tuple-Ranking principles are applicable to any grouping bounds, from which the only impact is that different bounds result in different best-possible tuple depth H_g^{min} that can be achieved by tuple scheduling. Below we discuss how to obtain the grouping bounds (Section 3.1.1), how to implement the Tuple-Ranking Principle (Section 3.1.2), and how to implement the Group-Ranking Principle and how to enable new group-aware and rank-aware query plans that apply the principles (Section 3.1.3). Finally, we discuss variations of the querying plans that are applicable under various situations (Section 3.1.4).

3.1.1 Obtaining Grouping Bounds: Exploiting Upper-Bound Principle

Based on Requirement 1, the smallest H_g^{min} with respect to Eq. 3 is obtained by the tightest grouping bounds, *i.e.*, the exact tuple count and the highest unseen tuple value. In our framework we aim at using this tightest bounds for maximal pruning.

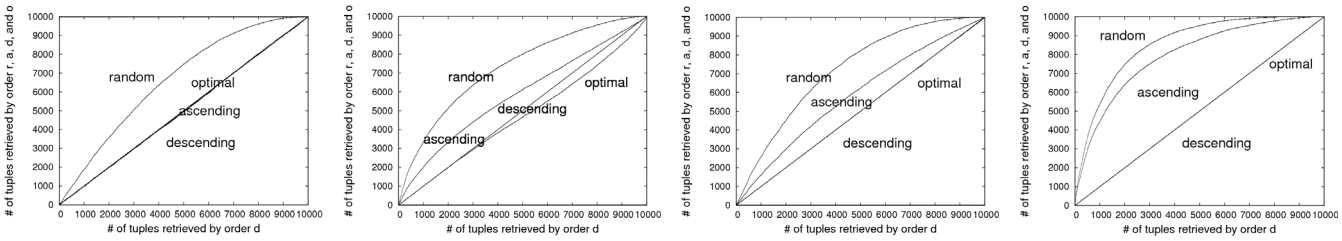
With respect to the tuple max, the tightest bound $\overline{T}_{\mathcal{I}_g}$=max$\{T[t_i] \mid t_i \in g - \mathcal{I}_g\}$ is impossible to obtain though. The reason is simply that, without actually seeing the unseen tuples, we cannot know the exact highest value among them. However, we can obtain a value that is very close to it. For instance, if the tuples in g are retrieved in T-*desc/asc* order, the T value of the last seen tuple from the top end bounds the value of the unseen tuples. Before any member tuple of g is retrieved, $\overline{T}_{\mathcal{I}_g}$ has an initial value $\overline{T}_g$, which is the maximum-possible value of T among all the tuples in g. A tight $\overline{T}_g$ can be obtained either by application semantic (*e.g.*, according to the definition of T), or by the indices that are pervasively built upon base tables and materialized views in OLAP environment. It can be either global (*e.g.*, using the overall highest T value according to the index), or group-specific (*e.g.*, using multi-key index over the grouping attribute and the attributes involved in T.)

With respect to the tuple count, the tightest bound (*i.e.*, the exact size of a group), $\overline{|g|}$=$|g|$, provides the most pruning power. Looser bounds can be also obtained. For example, we may use the size of a base table to bound the size of any base table group, and the product of base table group sizes to bound the joined group size (by assuming full join, *i.e.*, Cartesian product). However, such upper-bounds are very loose and are unlikely to realize early pruning. We note that any efficient method to compute a tight upper-bound of the count can be plugged into our framework as another choice of the tuple count. Below we discuss how to obtain the exact tuple count $\overline{|g|}$=$|g|$. There are three situations:

- **Counts ready:** In decision support, although the ranking aggregate function $G(T)$ can be very ad-hoc, the join and grouping conditions are largely shared across many related queries, as motivated in Section 1. In such an environment, materialized views are built based on the query workload to cover frequently asked query conditions. As a very basic aggregate function in OLAP, the count of each group is thus often ready through the materialized views, *e.g.*, in data cube.
- **Counts computed from materialized information:** In certain cases, the counts are not directly ready, but can be efficiently obtained by querying the materialized views [17, 32, 11, 1, 28]. For example, for a *top-k* aggregate query with selection conditions involving some dimensional attributes (*e.g.*, May$\leq month \leq$June), a group (*e.g.*, $city$='Chicago') corresponds to the aggregate of multiple underlying groups (*e.g.*, ($city$='Chicago', $month$=May) and ($city$='Chicago', $month$=June)). Its size can thus be obtained by aggregating upon the materialized views (*e.g.*, the view containing the count of each ($city$, $month$) group).
- **Counts computed from scratch:** When counts cannot be directly or indirectly obtained, we have to compute it from scratch. That is, we replace the ranking function F= $G(T)$ by $count(*)$ and remove the `ORDER BY` and `LIMIT` clauses. The resulting query (let's call it *count query*) is executed by any traditional approach to obtain the counts. For instance, the count query corresponding to our running example is

  Select $R.g$, **COUNT**$(*)$ **From** R **Group By** $R.g$

In Section 4, our experimental results show that our approach is orders of magnitude more efficient than the materialize-group-sort approach when counts are available. When we have to compute the counts from scratch (or similarly from materialized views), the cost of the first single query is comparable to that of materialize-group-sort. More importantly, the resulting counts can be materialized and maintained to benefit many subsequent related ad-hoc queries, thus the cost of computing the counts is amortized.



(a)Uniform distribution in $[0:1]$. (b)Normal distribution μ=0.5, σ=0.3. (c)Exponential distribution λ=1. (d)Gamma distribution α=0.5, β=0.5.

Figure 6: Number of tuples retrieved by random(r), ascending (a), descending (d), and optimal (o) order to get the same or lower upper-bound as descending order.

3.1.2 T-descending Heuristic: Implementing Tuple-Ranking Principle

As Requirement 3 states, we only need to consider the class of T-*desc/asc* orders for finding an optimal tuple order of a group. Such orders retrieve the tuples from only the top and bottom ends with respect to the value of T, thus exploit T-based ranking to either reduce the seen scores or the upper bound of those unseen. Based on the intuition of retrieving tuples from two ends, we thus consider two simple heuristics of choosing intra-group order, each of which produces a representative case of T-*desc/asc*, for α: $t_1 \rightarrow \cdots \rightarrow t_n$.

1. T-*descending*: Always choose the tuple with the *highest* T-score as the next, *i.e.*, $T[t_1] \geq \cdots \geq T[t_n]$.
2. T-*ascending*: Always choose the tuple with the *lowest* T-score as the next, *i.e.*, $T[t_1] \leq \cdots \leq T[t_n]$.

Example 5 (Tuple Orders): Consider group g_2 in our example. Figure 3(b) shows four example orders: *descending*, *ascending*, *hybrid*, and *random*. The *descending* and *ascending* are special cases of T-*desc/asc*, *hybrid* is another instance in T-*desc/asc*, and *random* is an order that does not belong to T-*desc/asc*. For each order, the figure shows how $\overline{F}[g_2]$ changes in sequence, *e.g.*, *descending* decreases $\overline{F}$ as 3, 1.2, 1, .95 (the final score). By comparison, *descending* is the best order, which lowers $\overline{F}$ most rapidly. ■

We choose T-*descending* as our implementation heuristic. We show that T-*descending* in practice is often the best choice for typical score distributions (*e.g.*, *uniform* and *normal*) and aggregate functions (*e.g.*, *sum* and *avg*). In Figure 6 we empirically compare T-*ascending* (a), T-*descending* (d), the random order (r), and the optimal order (o) which results in H_g^{min}. The tuple score $T[t_i]$ within a group g of n=10,000 tuples are generated by various distributions, in the range of $[0, 1]$. The aggregate function is *sum*. (The results for *avg* are similar.) Suppose the maximal-possible score is f after $|\mathcal{I}_g^d|$ tuples are retrieved by d. Ranging $|\mathcal{I}_g^d|$ from 1 to n (x-axis), we compare $|\mathcal{I}_g^a|$, $|\mathcal{I}_g^d|$, $|\mathcal{I}_g^r|$, and $|\mathcal{I}_g^o|$(y-axis), the number of retrieved tuples by a, d, r, and o, respectively, to get their maximal-possible scores lower than or equal to f. The curve for T-*descending* is the diagonal since it is the reference order. The figure shows that (1) T-*descending* in most cases overlaps with the optimal order, justifying our implementation heuristic; and (2) the random order is always worse than others, verifying that simply choosing any order is not appropriate.

There are data distributions where T-*descending* is worse than other orders. For instance, let's change g_2 to g_2' in Figure 3(b), which shows four example orders for both g_2 and g_2'. Now, T-*ascending* is the best order, by getting low scores from the bottom(*i.e.*, $r_8.v$=0). In general, in a dataset, if many tuples are in the high score end, T-*descending* at the beginning cannot effectively lower the upper-bound of unseen tuples, resulting in low efficiency.

Note that more sophisticated heuristic may be applied in determining intra-group tuple order. For instance, a heuristic can indeed retrieve both the high and low score ends, by determining to retrieve from top or bottom based on the distribution of seen data. Such heuristic would require complex implementation and bring more overheads, for ranking on both ends and the analysis of data distribution. More seriously, such greedy algorithm based on the seen data may led to local optimum. For instance, if there are several tuples with the same score clustered at the high score end, such heuristic may determine that retrieving tuples from the top end cannot reduce the upper-bound of the unseen tuples, thus will only retrieve from the bottom end. However, it may turn out that tuple scores decrease rapidly after those with the same score, thus retrieving from the top end can be much better in the long run. Compared with such heuristic, T-*descending* is much simpler and empirically almost as good as the optimal order, as discussed above.

3.1.3 Group-Aware and Rank-Aware Plans: Exploiting Group- and Tuple-Ranking Principles

To exploit the Group-Ranking Principle, our proposed new *rankagg* operator (Figure 2(b)) explicitly controls the inter-group ordering. Instead of passively waiting for the underlying subtree to fully materialize all the groups, the *rankagg* operator *actively* determines the most promising group g according to the maximal-possible scores of all valid groups, and draws the next tuple in g from the underlying subtree. (By Requirement 2, any current top-k incomplete group can be such g to request.) When the most promising group is complete, its aggregate is returned as a query result. Therefore, the groups are always output from the *rankagg* operator in the ranking order of their aggregates, eliminating the need for the blocking sorting operator in Figure 2(a).

This "active grouping" is a clear departure from the materialize-group-sort scheme and it requires changing the interface of operators. Specifically, we change the *GetNext* method of the iterator to take g as a parameter. Our operators are thus *group-aware* so that grouping is seamlessly integrated with other operations. Recursively starting from the *rankagg* operator, an upper operator invokes the *GetNext*(g) methods of its lower operators, providing the most promising group g as the parameter. For a unary operator, the same g is passed as the parameter to its lower operator. For a binary operator such as join, g is decomposed into two components g' and g'' and are passed as the parameters to the left and the right child operators, respectively. In response, each operator sends the next output tuple from the designated group g to its upper operator.

To enforce the T-*descending* heuristic (Section 3.1.2), the query tree underlying *rankagg* must be *rank-aware* as well. For this purpose, we leverage the recent work on ranking query processing [23, 26]. However, we must address the challenges in satisfying group-awareness and rank-awareness together.

Example 6: Consider again Example 3. Figure 4 illustrates (a) a group-aware plan which we call GroupOnly and (b) its execution. The group-aware scan operator can produce tuples from the group designated by the *rankagg* operator above it. The tuples within each

group are produced in their on-disk order. To contrast, Figure 5 illustrates (a) a plan that is both group-aware and rank-aware which we call GroupRank and (b) its execution. The group- and rank-aware scan operator in this plan produces tuples in the descending order of $R.v$ within each group. The executions of these two plans are already explained in Examples 3 and 4. Note that GroupOnly does not exhaust the tuples in g_2 and does not touch the tuples in g_3 at all. GroupRank takes even fewer steps than the GroupOnly plan, by exploiting order within groups. ■

3.1.4 *Variations of Query Plans: Trading off Group- and Rank-Awareness*

The group- and rank-aware query plans can be much more efficient than traditional plans. We call them GroupRank plans (*e.g.*, Figure 5(a)). However, there can be situations under which these plans are inapplicable or inefficient, therefore we propose variations of plans. These variations cover many different applicable situations, thus serve as a robust solution that provides better strategies than the traditional approach. Moreover, both group-awareness and rank-awareness can bring overhead, respectively. For example, to enable rank-aware join, we adapt techniques in recent work [23], where the join operator buffers the joined tuples in ranking queues. Maintaining the ranking queues can bring significant overheads or even offset the advantages of processing joins incrementally. Therefore the variations provide ways to trade off their overheads. We study the performances of these plans in Section 4. However, we leave the problem of optimizing among multiple applicable plans as our future topic.

First, GroupOnly plans (*e.g.*, Figure 4(a)), where the operators are group-aware but not rank-aware. The *rankagg* operator still gets the next tuple from the most promising group, but in arbitrary order within each group. Such plans are applicable when ranking on T cannot be efficiently supported. For example, ranking processing techniques require monotonic T [23] or splitting and interleaving T [26], which may not be applicable in certain situations.

Second, RankOnly plans where the operators are rank-aware only. Instead of telling the underlying operator the designated group, the *rankagg* operator gets interleaved tuples from all groups and orders the groups by their aggregate scores.

Finally, GroupRank-ϵ ($0 \leq \epsilon \leq 1$) plans which are the same as GroupRank except that the join operators output tuples out-of-order, while at the same time not in arbitrary order. Since full ranking can be expensive, we experiment with approximations, which trade ranking overhead with precision of tuple ranking. In a GroupRank-ϵ plan, upon the request of sending the next tuple from a given group, a join operator outputs the top tuple t in its ranking queue for that group if $ub_t \geq ub \times \epsilon$, where ub_t is the upper-bound of t and ub is the upper-bound of the unseen tuples. The greater value between ub_t and ub is reported to the upper operator as the upper-bound of any future tuples to be reported. Note that the scan operators in GroupRank-ϵ are still rank-aware and group-aware. It is clear GroupRank is actually an extreme case, GroupRank-1. As another extreme case, in GroupRank-0, a join operator outputs the top tuple in the ranking queue of a group whenever the queue is not empty. Note that GroupRank-0 is not a GroupOnly plan as all seen tuples in the ranking queue are still ordered.

3.2 Implementing the New `rankagg` Operator

The iterator interface for *rankagg* is shown in Figure 7. The *rankagg* operator maintains a priority queue storing the upper-bounds of groups that are not output yet. Note that the priority queue in *rankagg* and the ranking queues in the group- and rank-aware join operators serve different purposes. While the ranking queues

```
1: //input: the underlying operator.
2: //k: the requested number of groups.
3: //q: the priority queue of groups.
4: //g.obtained: the number of obtained tuples in g, i.e., |I_g|.
5: //g.count: the size of g.
Procedure Open()
1: input.Open(); q.clear()
2: for each group g do
3:    init_ub(g); q.insert(g)
4: return
Procedure GetNext()
1: while true do
2:    if k==0 ∨ q.isEmpty() then
3:       Close()
4:       return
5:    g ← q.top()
6:    if g.count==g.obtained then
7:       finalize_ub(g); k ← k − 1
8:       return g
9:    t ← input.GetNext(g); update_ub(g,t); q.insert(g)
Procedure Close()
1: input.Close(); q.clear()
2: return
```

Figure 7: The interface methods of *rankagg*.

```
1: //g.ub: the maximal-possible score of a group g, i.e., F̄_{I_g}[g].
2: //g.sum: the sum of T for obtained tuples in g.
3: //T̄_{I_g}: the maximal-possible value of T among g's unseen tuples, retrieved in T-descending order.
4: //T̄_g: the initial T̄_{I_g} when no tuple is obtaind.
Procedure init_ub(g)
1: g.sum ← 0; g.obtained ← 0
2: T̄_{I_g}=T̄_g
3: g.ub = g.count × T̄_{I_g}
4: return
Procedure update_ub(g,t)
1: g.sum ← g.sum + T[t]
2: g.obtained ← g.obtained + 1
3: T̄_{I_g}=T[t]
4: g.ub ← g.sum + (g.count − g.obtained) × T̄_{I_g}
5: return
Procedure finalize_ub(g)
1: //nothing needs to be done
2: return
```

Figure 8: The upper-bound routines for G=sum.

in joins are used to buffer tuples for providing ranking access to the tuples, the priority queue is used for efficiently maintaining the current top group dynamically. The *rankagg* always gets the next tuple from the top group in the priority queue and updates its upper-bound. When the top group is complete, it is guaranteed to be the best among those in the queue, thus can be reported. (In RankOnly plans, a hash table instead of priority queue is used to give fast access to the upper-bounds. An iteration through the hash table is performed periodically and the top group is output when it is complete.) Below we discuss how to maintain the routines for upper-bound computation and how to manage the priority queue.

Upper-Bound Computation: For a ranking aggregate F=G(T), the maximal-possible score of a group g with obtained tuples $\mathcal{I}_g$, $\overline{F}_{\mathcal{I}_g}[g]$, can be computed by Eq. 3. Starting from the initial upper-bound, we must keep updating $\overline{F}_{\mathcal{I}_g}[g]$ when tuples are incrementally obtained. When the last tuple from g is obtained, $\overline{F}_{\mathcal{I}_g}[g]$ becomes the aggregate value $F[g]$. This description clearly indicates that the upper-bound itself can be maintained by an external aggregate function. (Let's call it *upper-bound routine*.) For example, in PostgreSQL, a user-defined aggregate function is defined by an ini-

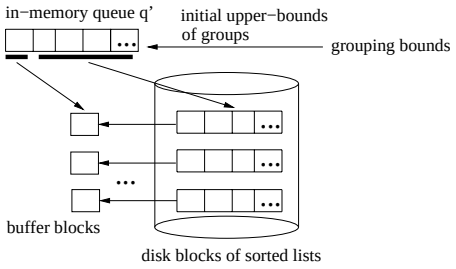


Figure 9: Priority queue.

tial state, a state transition function, and a final calculation function. Therefore for the G in a ranking aggregate query F=$G(T)$, the corresponding upper-bound routines consist of *init_ub*, *update_ub*, and *finalize_ub*. They are invoked in the interface methods of *rankagg* (Figure 7). Such routines can be pre-defined if G is a built-in function. For example, Figure 8 illustrates the upper-bound routines for G=sum. As an alternative, in GroupOnly plans, the upper-bound in the *update_db* procedure should become $g.ub \leftarrow g.sum + (g.size - g.count) \times \overline{T}_g$. When G is a user-defined aggregate function itself, the upper-bound routine is defined by straightforward adaptation of the utilities (initialization, state transition, final calculation) of G, mainly to substitute the value of unknown tuples with $\overline{T}_{\mathcal{I}_g}$. We omit further discussion of these details.

Efficient Ranking Priority Queue Implementation: For a ranking aggregate query, the total number of groups can be huge although only the top k groups are requested. For example, joining three tables with $1,000$ groups on each table can potentially lead to 1 billion joined groups. Managing the upper-bounds of the huge number of groups by a simple priority queue implementation can thus bring significant overhead.

We address this challenge from two aspects, illustrated by our new priority queue in Figure 9. *First*, we populate the priority queue incrementally. It is necessary to insert a group into the priority queue only when its maximal-possible score is among the current top-k, by Requirement 2. By using a global tuple max (the overall highest T value across all groups), the tuple count effectively determines the initial maximal-possible score of every group, based on Eq. 3. Therefore the groups can be incrementally inserted from higher to lower counts, utilizing the index on the tuple count. Such index over summary tables is extensively built and utilized in decision support. Moreover, there are techniques (*e.g.*, [27]) for getting the groups with the largest sizes (incrementally). *Second*, when the (incrementally expanding) priority queue does become too big to fit in the memory, we use a 2-level virtual priority queue q consisting of (1) an in-memory priority queue q' (implemented by the heap algorithm), and (2) a set of in-memory buffer blocks of sorted lists and a set of on-disk sorted lists.

Initially, only the first batch of groups ($1,000$ in our experiments) with the largest counts are inserted into q'. Whenever q' is full, it is emptied and its elements are converted into a sorted list (ordered by upper-bounds), of which the first top block is kept in buffer and the rest is sent to the disk. When a request is issued to get the top element (group) from q, the top elements from q' and from every buffer block are compared and the overall top group is returned. When a buffer block is exhausted, the next block from the corresponding sorted list is read from the disk into the buffer. If the top group is complete, it is returned as a query result, otherwise the next tuple from the group is obtained to update its upper-bound and the group is inserted back to q. It is possible the upper-bound of the top group becomes smaller than that of the group with the largest size among those that are not inserted. Under such situation, the next batch of groups are inserted into q'.

With the new priority queue, only the top groups (which are more likely to remain at the top) are kept in memory, in analogy to various cache replacement policies. Moreover, many groups may have initial upper-bounds smaller than the *top-k* threshold θ, thus may even never be necessarily touched when the top k answers are obtained. Therefore our concern with the potentially huge number of groups is addressed, as verified by the experiments in Section 4.

3.3 Impacts to Existing Operators

In this section we discuss the impacts of *rankagg* to other query operators, scan and join in particular.

Scan: To be group-aware, the new scan operator must access the next tuple in the group g requested by its upper operator. In [22], a round-robin index striding method was introduced to compute on-line aggregates with probabilistic guarantees. Our scan operator adopts the index striding technique. Multiple *cursors*, one per group, are maintained on the index to enable such striding. A cursor is advanced whenever a tuple is obtained from the cursor. However, there are two important differences: (1) in our case, index retrieval is governed by the dynamically designated group instead of fixed weights; and (2) to access tuples within each group in the descending order of T, *i.e.*, to be rank-aware, we build multi-key index, by using the grouping attribute as the first key and the attribute in T as the second key. For example, for the following query:

Select $R.g$, $S.g$, **SUM**($R.v$+$S.v$) **From** R, S
Group By $R.g$, $S.g$ **Order By SUM**($R.v$+$S.v$) **Limit** 1,

a multi-key index on ($R.g$, $R.v$) can be used for accessing R and another index on ($S.g$, $S.v$) for S. (Similarly when there are multiple grouping attributes on a table.) Note that we do not discuss how to select which indices to build, as such index selection problem has been studied before (*e.g.*, [18]) and is complementary to our techniques. When index on a table is unavailable, we have to scan the whole table and build a temporary index or search structure.

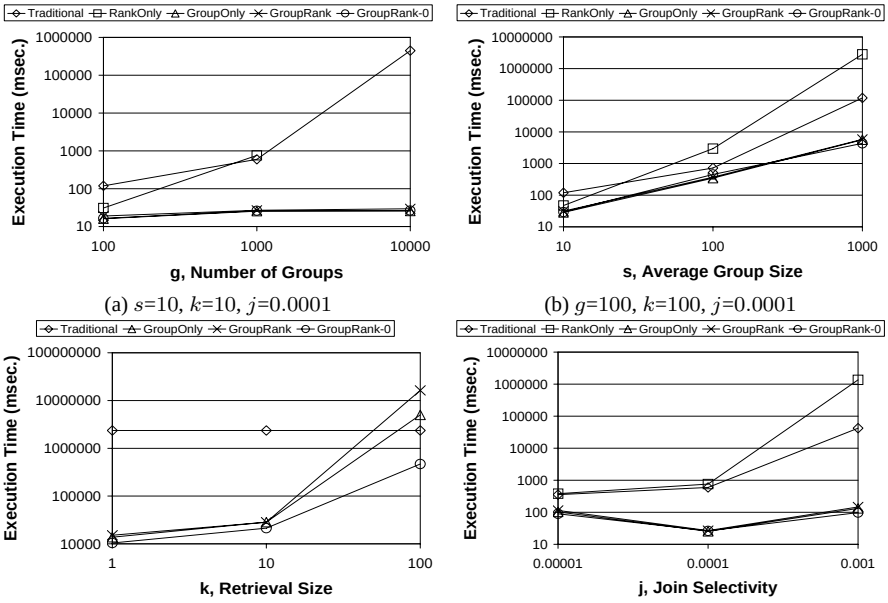
Join: For group-awareness, when a join operator is required to produce a tuple of group g, it outputs such a tuple from its buffer when available, otherwise it recursively invokes the *GetNext*(g') and *GetNext*(g'') methods of its left and right input operators, respectively. For instance, for the above query, suppose a join operator that joins R and S is requested by *rankagg* to output the next tuple from a group ($R.g$=1, $S.g$=2). The join operator directly returns a joined tuple from its buffer when available. Otherwise, it requests the next tuple with $R.g$=1 from R or the next tuple with $S.g$=2 from S.

To be rank-aware, the join operator must output joined tuples in the order with respect to T, *e.g.*, $R.v$+$S.v$. We adopt the HRJN algorithm [23]. The algorithm maintains a ranking priority queue (not to be confused with the priority queue in Section 3.2) for buffering joined tuples, ordered on their upper-bound scores. The top tuple from the queue is output if its upper-bound score is greater than a threshold, which gives an upper-bound score of all unseen join combinations. Otherwise, the algorithm continues by reading tuples from the inputs and performs a symmetric hash join to generate new join results. The threshold is continuously updated as new tuples arrive. In the new implementation, we manage multiple ranking queues, one for each joined group and use a hash table to maintain the pointers to each ranking queue. In GroupOnly plans, the join operator uses a FIFO queue instead of priority queue to buffer join results (thus HRJN becomes the hash ripple join [19]).

4. EXPERIMENTS

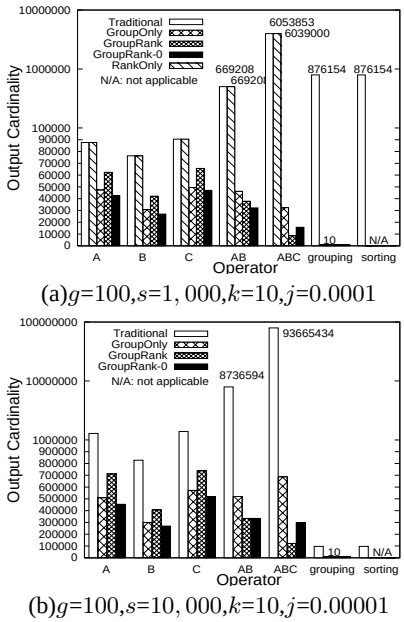
4.1 Settings

The proposed techniques are implemented in PostgreSQL. The experiments are conducted on a PC with 2.8GHz Intel Xeon SMP



(a) s=10, k=10, j=0.0001
(b) g=100, k=100, j=0.0001
(c) g=100, s=10, 000, j=0.00001
(d) g=1, 000, s=10, k=10

Figure 10: Performance of different execution plans.



(a)g=100,s=1, 000,k=10,j=0.0001
(b)g=100,s=10, 000,k=10,j=0.00001

Figure 11: Output cardinalities.

(dual hyperthreaded CPUs each with 512KB cache), 2GB RAM, and 260GB RAID5 array of 3 SCSI disks, running Linux 2.6.9.

We use a synthetic data set of three tables (A, B, C) with the same schema and similar size. Each table has one join attribute jc, one grouping attribute g and one attribute v that is aggregated. For each tuple, the three attribute values are independently generated random numbers. In each base table, the values of v follow the uniform distribution in the range of $[0, 1]$. The number of distinct values of j is $\frac{1}{j}$, where j is a configurable parameter capturing join selectivity. The values of j follow the uniform distribution in the range of $[1, \frac{1}{j}]$. The number of distinct values of g is g, *i.e.*, g captures the number of groups on each table. For example, when g=10, the maximal number of joined groups over ABC is g^3=1, 000. The number of tuples corresponding to each distinct value of g follows normal distribution, with average s, *i.e.*, s is the average size of base table groups.

We use the star-join query $\mathcal{Q}$ in Section 1. We compare five execution plans, Traditional, RankOnly, GroupOnly, GroupRank (*i.e.*, GroupRank-1), and GroupRank-0. They have the same plan structure that joins A with B and then with C. Traditional is an instance of the materialize-group-sort plan in Figure 2(a). It uses sort-merge join as the join algorithm and scans the base tables by the indices on the join attributes. The RankOnly, GroupOnly, GroupRank, and GroupRank-0 use the new *rankagg* operator. Moreover the join and scan operators in these plans are group-aware and/or rank-aware, as described in Section 3.1.4. We executed these plans under various configurations of four parameters, which are the number of requested groups (k), the number of groups on each table (g), the average size of base table group (s), and the join selectivity (j). We use $gWsXkYjZ$ to annotate the configuration g=10^W, s=10^X, k=10^Y, and j=10^{-Z}.

4.2 Results

We first performed 4 sets of experiments. In each set, we varied the value of one parameter and fixed the values of other three parameters, among k, g, s, and j. The plan execution time under these settings is shown in Figure 10. (Both x and y axes are in logarithmic scale.) The figure clearly shows that our new plans outperformed the traditional plan by orders of magnitude. Traditional is only comparable to the new plans when there are not many groups, the group size is small, many results are requested, and joins are very selective. RankOnly is as inefficient as Traditional. It did not finish after running for fairly long under some configuration (g=10, 000 in Figure 10(a)) and is excluded from Figure 10(c) for the same reason. As an intuitive explanation, if the top-1 group has a member tuple that is ranked at the last place, all the groups must be materialized in order to obtain the top-1 group. This indicates that being rank-aware itself [23, 26] does not help to deal with *top-k* aggregate queries.

The differences among the new plans are not obvious in Figure 10(a)(b)(d) because Traditional and RankOnly are too far off the scale. However, Figure 10(c) clearly illustrates their differences. In Figure 12, we further compare GroupOnly, GroupRank and GroupRank-0 under the 8 configurations in Figure 10(c)(d). For each plan, we show the ratio of its execution time to the execution time of GroupRank. The results show that GroupOnly in many cases is better than GroupRank, verifying that the ranking overhead can offset the advantages of group-awareness in certain cases. On the other hand, the performance is much improved when we reduce the ranking overhead, as GroupRank-0 almost always outperformed GroupOnly and GroupRank.

We further analyze these plans by comparing the output cardinalities of their operators. Figure 11 reports the comparisons under two configurations. The results for other configurations are similar. As it shows, Traditional enforces full materialization. RankOnly was not able to reduce the cardinalities and further incurred ranking overhead, which explains why it is even worse than Traditional in many cases. GroupOnly reduced the cardinalities significantly by partial consumption of base tables and partial materialization of join results. GroupRank produced less join results than GroupOnly because of rank-awareness. However, it also consumed more base table inputs because join operators must buffer more inputs to produce ranked outputs (the ranking overhead). Finally, GroupRank-0 balanced the benefits and overhead of rank-awareness, as explained in Section 3.1.4. Therefore it consumed less number of base table inputs, although produced some more join results.

To further study the tradeoff in being rank-aware, we show the performance of GroupRank-ϵ in Figure 13 by ranging ϵ from 0 to 1.

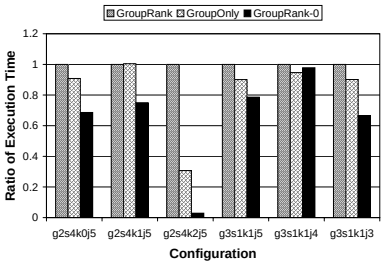


Figure 12: Comparing new plans.

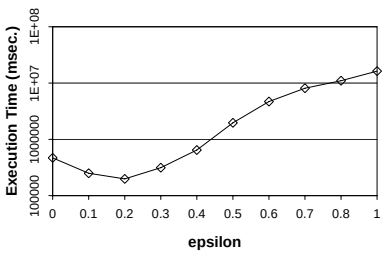


Figure 13: Performance of GroupRank-ϵ**.**

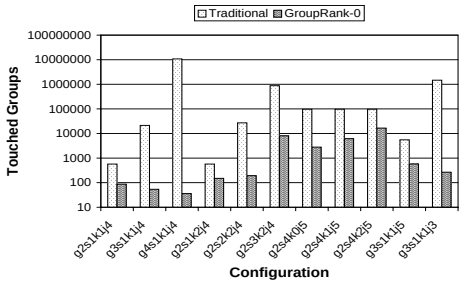


Figure 14: Touched groups.

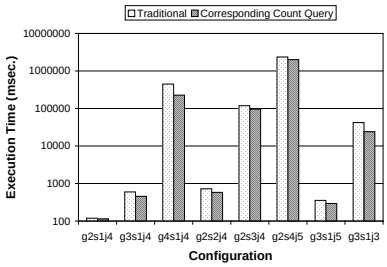


Figure 15: The cost of computing counts from scratch.

Note that GroupRank and GroupRank-0 are extreme cases for ϵ=1 and 0, respectively. Interestingly none of them is the best, which indicates the choice of ϵ should be captured by query optimizer.

We verify that managing the priority queue of *rankagg* (Section 3.2) does not require significant overhead, although the total number of groups can be potentially huge. In Figure 14, we compare the number of joined groups touched by GroupRank-0 and Traditional under the 11 distinct configurations from Figure 10. (We count a group as "touched" if at least 1 tuple from the group is produced during the plan execution. Therefore the touched groups are maintained by the priority queue and the top k groups come from the touched groups.) The results show that most of the groups never need to be touched by the new plans, therefore it is not expensive to maintain the priority queue. Figure 14 also clearly illustrates why the new plans outperform Traditional, together with Figure 11. While Traditional processes every group and every tuple in each group due to its nature of full materialization, our new plans save significantly by the early pruning resulting from the group-ranking and tuple-ranking principles.

Our framework requires tuple count, which can be obtained as discussed in Section 3.1.1. Specifically, when the tuple count must be computed from scratch by a count query, the cost of answering one ranking aggregate query consists of the cost of the corresponding count query and executing the new query plan based on the obtained counts. In Figure 15, we compare the costs of Traditional and the count query under 8 configurations from Figure 10. Note that k is irrelevant in this experiment since Traditional generates the total order of all groups and the count query generates the count of every group. (There are overlapping configurations in Figure 10(a)-(d) when k is ignored, resulting in totally 8 distinct configurations.) The results verify that computing the count query is slightly cheaper than the original ranking aggregate query. Since our new query plans are orders of magnitude more efficient than the traditional plan, the total cost of a count query and a new plan is comparable to, or even cheaper than, that of the traditional plan. More importantly, the materialized tuple counts are then used by the future related ranking aggregate queries that share the same Boolean conditions with the original query (scenario 1 in Section 3.1.1), or of which the tuple count can be computed from the materialized counts (scenario 2). Nevertheless, it brings us the advantages of "paying one, getting the following (almost) free".

Discussions: We should emphasize that although the new query plans are not always equally efficient, they provide better strategies than the traditional approach in processing *top-k* aggregate queries, under various applicable conditions, as discussed in Section 3.1.4. Moreover, the experimental results indicate that none of the plans is always the best and their costs can be orders of magnitude different. Their diverse applicability and performance thus call for new query optimization techniques. Especially, the performance of our methods depends on multiple parameters, including the number of groups, the sizes of groups, the distribution of tuple scores, the memory buffer size, *etc*. Thus a cost model incorporating these parameters to estimate the costs of plans is the key to the new optimizer. The estimates can enable us to choose among the new plans and even the traditional plans. Developing such cost model and optimizer thus is an important topic of future research.

5. RELATED WORK

To the best of our knowledge, this is the first piece of work that provides efficient support of ad-hoc *top-k* aggregate queries. In this section, we highlight the recent efforts related to this work.

Ranking or *top-k* queries have recently gained much attention of the research community. Works in this area mainly include those in the middleware scenario [13, 14, 4, 6], or in RDBMS setting [5, 12, 7, 23, 24, 8, 26]. None of these works support aggregate queries.

Order optimization [31] was proposed in relational query optimizer to avoid sorting or to minimize the number of sorting columns. *Eager aggregation* [34, 9] and *lazy aggregation* [35] were proposed to optimize GROUP-BY processing by functional dependencies. [10, 29] proposed algorithms and framework for combining order and grouping optimization. [33] extended eager aggregation in OLAP environment by utilizing special access methods.

Efficient computation of data cubes [2, 36, 30] focuses on sharing the computation across different cuboids instead of how to process a single cuboid. Answering aggregate queries using materialized views was addressed in [17, 32, 11, 1, 28].

Semantically similar to *top-k* aggregate queries, *iceberg queries* [15] retrieve only the groups that qualify under a Boolean condition (expressed in the HAVING clause). The techniques in [15] are confined to single-table queries (joins have to be materialized beforehand) and sum or $count$ instead of general aggregate functions. The notion of iceberg queries was extended to *iceberg cubes* [3, 20]. Iceberg cuboids with the same dimensional attributes involve the computation of an iceberg query, or essentially a ranking aggregate query. These works focused on pruning cuboids in an iceberg cube, while how to efficiently compute a cuboid was not considered. Hence, we consider our work to be complementary in evaluating iceberg cubes.

Online aggregation [22, 19] supports *approximate* query answering by providing running aggregates with statistical confidence intervals. We extend its index striding technique to support the group-aware and rank-aware scan operator.

The work closest to ours is [27], where ranking aggregates are computed over a specified range on some dimensions other than the grouping dimensions, by storing pre-computed partial aggregate information in the data cube. Therefore it can only support predetermined aggregate function and aggregated expression, lacking the ability to support ad-hoc queries. However, it is complementary to our work as it can be used to obtain the group sizes when there are selection conditions over the dimensional attributes, as mentioned in Section 3.2.

6. CONCLUSION

We introduced a principled and systematic framework to efficiently support ad-hoc *top-k* (ranking) aggregate queries. As the foundation, we developed the Upper-Bound Principle that dictates the requirements of early pruning, and the Group- and Tuple-Ranking Principles that dictate the group-ordering and tuple-ordering requirements. The principles together realize optimal aggregate query processing. We proposed an execution framework for applying the principles and addressed the challenges in implementing the framework. The experiment results validate our framework by showing significant performance improvement. To the best of our knowledge, this is the first work that provides efficient support of ad-hoc *top-k* aggregates. The techniques address a significant research challenge and can be useful in many decision support applications.

Acknowledgements: We thank Jiawei Han and Xiaolei Li for helpful discussions regarding the positioning of the paper.

7. REFERENCES

[1] F. N. Afrati and R. Chirkova. Selecting and using views to compute aggregate queries (extended abstract). In *ICDT*, pages 383–397, 2005.

[2] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *VLDB*, pages 506–521, 1996.

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cube. In *SIGMOD*, pages 359–370, 1999.

[4] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–380, 2002.

[5] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *SIGMOD*, pages 219–230, 1997.

[6] K. C.-C. Chang and S. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346–357, 2002.

[7] S. Chaudhuri and L. Gravano. Evaluating top-*k* selection queries. In *VLDB*, pages 397–410, 1999.

[8] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *CIDR*, pages 1–12, 2005.

[9] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, pages 354–366, 1994.

[10] J. Claussen, A. Kemper, D. Kossmann, and C. Wiesner. Exploiting early sorting and early partitioning for decision support query processing. *VLDB J.*, 9(3), 2000.

[11] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *PODS*, pages 155–166, 1999.

[12] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB*, 1999.

[13] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.

[14] R. Fagin, A. Lote, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[15] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *VLDB*, pages 299–310, San Francisco, CA, USA, 1998.

[16] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[17] A. Gupta, V. Harinarayan, and D. Quass. Aggregate query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[18] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *ICDE*, pages 208–219, 1997.

[19] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, 1999.

[20] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD*, 2001.

[21] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, 1996.

[22] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[23] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[24] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *SIGMOD*, pages 203–214, 2004.

[25] C. Li, K. C.-C. Chang, and I. F. Ilyas. Efficient processing of ad-hoc top-k aggregate queries in OLAP. Technical Report UIUCDCS-R-2005-2596, Department of Computer Science, UIUC, June 2005. `http://aim.cs.uiuc.edu`.

[26] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.

[27] H.-G. Li, H. Yu, D. Agrawal, , and A. E. Abbadi. Ranking aggregates. Technical report, UCSB, July 2004.

[28] V. Lin, V. Vassalos, and P. Malakasiotis. MiniCount: Efficient rewriting of COUNT-queries using views. In *ICDE*, 2006.

[29] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *VLDB*, 2004.

[30] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *VLDB*, pages 116–125, 1997.

[31] D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, 1996.

[32] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *VLDB*, pages 318–329, 1996.

[33] A. Tsois and T. K. Sellis. The generalized pre-grouping transformation: Aggregate-query optimization in the presence of dependencies. In *VLDB*, pages 644–655, 2003.

[34] W. P. Yan and P.-Å. Larson. Performing group-by before join. In *ICDE*, pages 89–100, 1994.

[35] W. P. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. In *VLDB'95*, pages 345–357, 1995.

[36] Y. Zhao, P. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD*, pages 159–170, 1997.