Efficient Processing of Ad-Hoc Top-k Aggregate Queries in OLAP

Chengkai Li\(^1\)     Kevin Chen-Chuan Chang\(^1\)     Ihab F. Ilyas\(^2\)
\(^1\)Department of Computer Science, University of Illinois at Urbana-Champaign
ccli@uiuc.edu, kcchang@cs.uiuc.edu
\(^2\)School of Computer Science, University of Waterloo
ilyas@uwaterloo.ca

Abstract

In this paper, we develop a principled framework for efficient processing of ad-hoc top-k (ranking) aggregate queries in OLAP. Such queries provide the k groups with the highest aggregates to decision makers. Essential support of top-k aggregate queries is lacking in current RDBMSs, which process such queries in a naive and overkill materialize-group-sort scheme, therefore can be prohibitively inefficient. Our new framework is based on two fundamental properties, the Group-Ranking and Tuple-Ranking Principles. The principles dictate group-ordering and tuple-ordering requirement that together guide the query processor toward the optimal aggregate query processing. To realize the requirements, we propose a new execution model and address the challenges of implementing new query operators, enabling efficient top-k aggregate query plans that are both group-aware and rank-aware. The experimental study validates our framework by demonstrating orders of magnitude performance improvement in the new query plans, compared with the traditional approach.

1 Introduction

Aggregation is a key operation in data warehousing for OLAP (On-Line Analytical Processing), which is very important for decision support in a variety of applications such as manufacturing, sales management, stock analysis, etc. In aggregation queries, aggregates are computed over groups with respect to some grouping attributes. Decision making procedures naturally involve comparing and ranking data. Among the large number of groups, only the ones with certain significance are of interest to decision makers in many cases. 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. As a matter of fact, 15 of the 22 queries in the TPC-H benchmark are ranking aggregate queries.

The sketch of top-k aggregate queries is illustrated below\(^1\), followed by an example query Q.

\[
\begin{align*}
\text{SELECT} & \quad g_{a1},...,g_{am}, F \\
\text{FROM} & \quad R_{1},...,R_{h} \\
\text{WHERE} & \quad c_{1} \text{ AND } ... \text{ AND } c_{t} \\
\text{GROUP BY} & \quad g_{a1},...,g_{am} \\
\text{ORDER BY} & \quad F \\
\text{LIMIT} & \quad k
\end{align*}
\]

Q: SELECT A.g, B.g, C.g, SUM(A.v+B.v+C.v) \\
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, upon the result of conjunctive Boolean join and selection conditions of conjuncts \(c_{1},...,c_{t}\) over the relations \(R_{1},...,R_{h}\), groups are formed by the grouping attributes \(g_{a1},...,g_{am}\), and ordered by a ranking aggregate \(F=G(T)\), where \(G\) is a monotonic aggregate function (e.g., \(\text{sum}\)) over an aggregated expression \(T\) (e.g., \(A.v+B.v+C.v\)). The top \(k\) groups with the highest \(F\) values are returned as the query result, with projected attributes specified in the SELECT clause. Formally, each group \(g=\{t_{1},...,t_{n}\}\) has a ranking score \(F[g]\), defined as\(^2\)

\[
F[g] = G(T)[g] = G(T[g]) = G(T[[t_{1},...,t_{n}]]) = \forall t_{1},...,t_{n}, T[t_{1}],...,T[t_{n}] \subseteq T
\]

(1)

As the result, \(Q\) returns a sorted list \(K\) of \(k\) groups, ranked by \(F\) scores, such that \(F[g] \geq F[g']\) \(\forall g \in K\) \(\forall g' \notin K\). 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.

Decision support queries in daily operations are commonly executed in an interactive manner where the results must be quickly presented to users and become the basis for formulating further queries. Therefore, given the large

\(^{1}\)While we use LIMIT (in PostgreSQL), various RDBMS use different SQL syntax to specify the retrieval size \(k\), e.g., FETCH FIRST in DB2.

\(^{2}\)More rigorously, instead of \(F=G(T)\), we should write as \(F=G \circ T\), i.e., \(F[g] = (G \circ T)[g] = G(T[g])\).
amount of data in a warehouse, efficiently processing top-\(k\) aggregate queries is very important and challenging. We address this challenge in the paper.

A distinguishing goal of our work is to support ad-hoc ranking aggregation criteria. With respect to \(G\), we aim to support not only standard aggregate functions such as sum, count, avg, min, and max, but also user-defined aggregate functions. With respect to \(T\), we aim to allow the aggregated expression to be specified on-the-fly by users instead of pre-determined. Such ad-hoc ranking criteria fit in very well with the exploratory nature of decision support applications and enable flexible and expressive data analysis.

Essential support of ranking aggregate is clearly lacking in current RDBMSs, which process such queries in the following way: (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. Given the large amount of data in a warehousing environment, such a naive materialize-group-sort scheme can be unacceptably inefficient. The user is only interested in the \(k\) top groups instead of a total order on all groups. The current strategy is thus an overkill, with unnecessary overhead from full scanning, joining, grouping, and sorting.

Efficient support of ad-hoc ranking aggregate is not available in OLAP either. A popular conceptual data model for OLAP is data cube [15]. A data cube is derived from a set of dimensional attributes and a measure attribute. ROLAP servers map the data cube into relational models by dimension tables and a fact table. Pervasive summary and index structures, and materialized views are further built upon these tables for answering aggregation queries. However, a data cube only maintains information of the pre-determined measure attribute using the prescribed aggregate function. The cubing approach essentially exploits pre-computation, whose result can easily become useless when the query is different from what the data cube is built for. Given our goal of supporting both ad-hoc aggregate function and ad-hoc aggregated expression, it is futile to use pre-computed results in answering ad-hoc aggregation queries.

In this paper, we propose a principled framework for efficient processing of ad-hoc top-\(k\) aggregate queries. We define a cost metric on retrieved tuples, capturing our goal of producing only necessary tuples for generating top \(k\) groups. As we find out, 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 each group only when necessary. We further discover that a total schedule of tuples can be fully determined by two orders—the order of retrieving groups (group ordering) and the order of retrieving tuples within each group (tuple ordering).

Based on this view, we develop two fundamental principles and a new execution model for processing top-\(k\) aggregate queries. We summarize our contributions as follows:

- **Principle for optimal aggregate processing:** We develop two fundamental properties, the Group-Ranking and the Tuple-Ranking Principles, which lead to the group-ordering and the tuple-ordering requirements, respectively. We formally show that the optimal aggregate query processing, with respect to our cost metric, can be derived by following both requirements together.

- **Execution model and physical implementations:** Guided by the principles, we propose a new execution model, which enables query plans that are both group-aware and rank-aware. We further address the implementation of the new query operators in this framework.

- **Experimental Study:** We implement the proposed techniques in RankSQL [23, 24] based on PostgreSQL. Our experiments verify that this new framework can achieve orders of magnitude improvement in many cases; e.g., Figure 1 compares a new query plan with the traditional materialize-group-sort approach. We experimentally study the reason for the performance differences and their applicability.

Neither RDBMS nor OLAP is sufficient for our purpose and they are at the two extremes, as aforementioned. Our approach is a hybrid that integrates the advantages of both world: the flexible processing of ad-hoc queries by RDBMS and the pre-computation by OLAP. The only materialized information required in our technique is the size of each group and the upper-bound value of the aggregated expression, for bounding the maximal-possible aggregate scores of groups. Such information is readily available, from basic aggregates (count) maintained in data cubes and the indices commonly built in OLAP.

In the rest, we present the principles of our techniques in Section 2. Section 3 introduces the execution model 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

As our running example, we use the following simple query. As input, Figure 3(a) shows relation $R$, with two attributes $R.g$ and $R.r$. The query will group by $R.g$ and returns top-k groups according to the aggregate function $F(R.g)$. The query will group by $R.g$ and returns top-k groups according to the aggregate function $F(R.g)$.

Our first property is to select the first group in each group.
or $\bar{F}_{I_{g_1}}[g_1]=2.7$. Similarly, $\bar{F}_{I_{g_2}}[g_2]=\text{sum}(3.1.0 \times 2)=2.3$; $\bar{F}_{I_{g_3}}[g_3]=\text{sum}(1.0 \times 2)=2.0$.

Such bounds guide our selection of the next tuple. Let’s illustrate with Example 1: 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 $\bar{F}_{I_{g_1}}[g_1] > \bar{F}_{I_{g_2}}[g_2]$, 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 $g_1$.

Our explanation above 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 (see Appendix A for proof), which mandates the priority of current top $k$ groups (i.e., $g_1$) over those groups that are outside (i.e., $g_2$).

**Property 1 (Group-Ranking Principle):** Let $g_1$ be any group in the current top-$k$ ranked by maximal-possible scores $\bar{F}$ 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-$k$ groups are the query answers if they are all fully evaluated.

The Group-Ranking Principle guides our inter-group ordering for query processing, by prioritizing on $\bar{F}$. 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 1 (Group Ordering):** To avoid unnecessary tuple evaluation, query processing should prioritize groups by their maximal-possible score $\bar{F}$:

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

**Example 2 (Sample Execution 1):** For our example $F=\mathcal{G}(T)=\text{sum}(R.v)$ to find the top-1 group, Figure 4(a) conceptually executes Requirement 1. (Ignore for now the query plan in Figure 4(a), which Section 3 will discuss.)

**Figure 4. Query execution 1: GroupOnly.**

As Requirement 1 dictates, we prioritize groups by $\bar{F}$ scores, initially $(3.0, 3.0, 2.0)$, when no tuples in any group $g$ are seen ($I_g=\emptyset$) and thus $\bar{F}_{I_g}=1.0$ in Equation 2. 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 $\bar{F}[g_1]=\bar{F}[g_1]=2.2$), the Stop Condition asserts no more processing necessary, and thus we return $g_1$ as the top-1.

**2.2 Tuple-Ranking Principle**

Our next 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 rankaggr operator (Figure 2(b)).

To start with, we note that different tuple orders will result in different cost efficiency, in terms of tuple depth accessed for each group. Given a tuple order $\alpha$, for group $g$, what would be the resulting tuple depth $H_g^\alpha$ that must be accessed? Recall in Example 2 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 $H_{x_1}^g=3$ (i.e., all of $r_1, r_5$, $r_7$ accessed), $H_{x_2}^g=2$, $H_{x_3}^g=0$, as Figure 4(b) shows. To contrast, Example 3 below shows that how different tuple orders result in different depths of access.

**Example 3 (Sample Execution 2):** Rerun Example 2 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 tuple orders, together with Requirement 1, result in the execution of Figure 5(b). (Again, ignore the query plan for now.) 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 $T[T_2]=r_1, v=9, \bar{F}[g_1]=0.9 + 0.9 \times 2$ (for 2 unseen tuples) $= 2.7$. Then, after $r_1$ in step 3, $\bar{F}[g_1]=0.9 + 0.7 + 0.7 \times 1$ (for 1 unseen tuple) $= 2.3$.

In this execution, as Figure 5(b) shows, each group is
Example query, group addresses this tuple ordering issue. It has two main results: next tuple from the group, when by the Progressive Condition, we only necessarily access a
are completed and "surfaced" to the top, at which point
halt, as the Stop Condition asserts, when the top-
independent of others.

e.g. [Image 50x259 to 58x260] Figure 4 and 5. For our discussion, call the lowest top-
score of the query as the
1 (out of 3), and thus
m
7, 1, .6) 4, 2, .4) 5, 1, .9) 1, 1, .7) 2.2 in our example. As Requirement 1 dictates,
and thus no longer needs further processing.

Property 2 (Tuple-Ranking Principle). With respect to a
ranking aggregate \( F = \{T_1, T_2, \ldots, T_k\} \), let the lowest top-k
group

Example 4 (Tuple Orders): Consider group \( g \), in our ex-
ample. Figure 3(b) central column shows one example
for each of the four classes of orders: \( \alpha \), \( \theta \), \( \tau \)-score as the next, \( I \), \( T \)-score as the next. Let \( I \) be
the best order depends on the actual data. To contrast,
hybrid

1. \( \alpha \)-descending: Always choose the tuple with the lower
2. \( \tau \)-descending: Always choose the tuple with the lowest
3. \( \theta \)-descending: Always choose the tuple with the lower
4. \( I \)-descending: Always choose the tuple with the lowest
5. \( T \)-descending: Always choose the tuple with the lowest

Second. Tuple Order. While groups are indepen-
dent, for each group, what orders

\[ H^g \] at which point it stops, because

\[ H^g \text{ is the} \]

Example 4 (Tuple Orders): Consider group \( g \), in our ex-
ample. Figure 3(b) central column shows one example
for each of the four classes of orders: \( \alpha \), \( \theta \), \( \tau \)-score as the next, \( I \), \( T \)-score as the next. Let \( I \) be
the best order depends on the actual data. To contrast,
hybrid

1. \( \alpha \)-descending: Always choose the tuple with the lower
2. \( \tau \)-descending: Always choose the tuple with the lowest
3. \( \theta \)-descending: Always choose the tuple with the lower
4. \( I \)-descending: Always choose the tuple with the lowest
5. \( T \)-descending: Always choose the tuple with the lowest

Second. Tuple Order. While groups are indepen-
dent, for each group, what orders

\[ H^g \] at which point it stops, because

\[ H^g \text{ is the} \]

Example 4 (Tuple Orders): Consider group \( g \), in our ex-
ample. Figure 3(b) central column shows one example
for each of the four classes of orders: \( \alpha \), \( \theta \), \( \tau \)-score as the next, \( I \), \( T \)-score as the next. Let \( I \) be
the best order depends on the actual data. To contrast,
hybrid

1. \( \alpha \)-descending: Always choose the tuple with the lower
2. \( \tau \)-descending: Always choose the tuple with the lowest
3. \( \theta \)-descending: Always choose the tuple with the lower
4. \( I \)-descending: Always choose the tuple with the lowest
5. \( T \)-descending: Always choose the tuple with the lowest

Second. Tuple Order. While groups are indepen-
dent, for each group, what orders

\[ H^g \] at which point it stops, because

\[ H^g \text{ is the} \]

Example 4 (Tuple Orders): Consider group \( g \), in our ex-
ample. Figure 3(b) central column shows one example
for each of the four classes of orders: \( \alpha \), \( \theta \), \( \tau \)-score as the next, \( I \), \( T \)-score as the next. Let \( I \) be
the best order depends on the actual data. To contrast,
hybrid

1. \( \alpha \)-descending: Always choose the tuple with the lower
2. \( \tau \)-descending: Always choose the tuple with the lowest
3. \( \theta \)-descending: Always choose the tuple with the lower
4. \( I \)-descending: Always choose the tuple with the lowest
5. \( T \)-descending: Always choose the tuple with the lowest

Second. Tuple Order. While groups are indepen-
dependent, for each group, what orders

\[ H^g \] at which point it stops, because

\[ H^g \text{ is the} \]

Example 4 (Tuple Orders): Consider group \( g \), in our ex-
ample. Figure 3(b) central column shows one example
for each of the four classes of orders: \( \alpha \), \( \theta \), \( \tau \)-score as the next, \( I \), \( T \)-score as the next. Let \( I \) be
the best order depends on the actual data. To contrast,
hybrid

1. \( \alpha \)-descending: Always choose the tuple with the lower
2. \( \tau \)-descending: Always choose the tuple with the lowest
3. \( \theta \)-descending: Always choose the tuple with the lower
4. \( I \)-descending: Always choose the tuple with the lowest
5. \( T \)-descending: Always choose the tuple with the lowest

Second. Tuple Order. While groups are indepen-
dependent, for each group, what orders

\[ H^g \] at which point it stops, because

\[ H^g \text{ is the} \]

Example 4 (Tuple Orders): Consider group \( g \), in our ex-
ample. Figure 3(b) central column shows one example
for each of the four classes of orders: \( \alpha \), \( \theta \), \( \tau \)-score as the next, \( I \), \( T \)-score as the next. Let \( I \) be
the best order depends on the actual data. To contrast,
hybrid

1. \( \alpha \)-descending: Always choose the tuple with the lower
2. \( \tau \)-descending: Always choose the tuple with the lowest
3. \( \theta \)-descending: Always choose the tuple with the lower
4. \( I \)-descending: Always choose the tuple with the lowest
5. \( T \)-descending: Always choose the tuple with the lowest

Second. Tuple Order. While groups are indepen-
dependent, for each group, what orders

\[ H^g \] at which point it stops, because

\[ H^g \text{ is the} \]
smallest depth \( l \) of sequence \( \alpha \) that makes the the maximal possible score of \( g \) to be below \( \theta \), i.e.,

\[
H^\alpha_g = \min_{l \in [1:n]} \{ \sum_{gi \in \{ t_1, \ldots, t_k \}} [g] < \theta \},
\]

or otherwise \( H^\alpha_g = n \) if such a depth does not exist.

- (\( T \)-based Ranking) To find the optimal order \( \alpha \) that results in the minimum \( H^\alpha_g \), we need to only consider three classes of orders: \( T \)-ascending, \( T \)-descending, and \( T \)-hybrid, all of which are rankings by \( T \) scores.

To conclude, we summarize the implementation implications of the Tuple-Ranking Principle as our Requirement 2, which will guide our design of a query processing framework for finding optimal tuple ordering.

**Requirement 2 (Tuple Ordering):** If Requirement 1 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 \( T \)-ascending, \( T \)-descending, or \( T \)-hybrid that results in the minimum \( H^\alpha_g \) as governed by Eq. 4.

**2.3 Putting Together**

Together, our Group-Ranking Principle (Section 2.1) and Tuple-Ranking Principle (Section 2.2) guide the tuple scheduling for our rankaggr operator to selectively draw from the underlying query tree (Figure 2b). We stress that, in principle, the two properties enable our finding of an optimal tuple schedule, for processing every group minimally in principle, the two properties enable our finding of an optimal intra-group order for each group.

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

**In Practice:** While the principles give the theoretical basis for best tuple scheduling, in practice, a framework should weigh the overhead of “optimization” versus its benefit. In particular, while Requirement 1 can be “deterministically” realized, Requirement 2 demands runtime optimization (to choose among orders) and sorting (for \( T \)-based ranking) for each group. To avoid such overheads, first, as a heuristic, our implementation assumes \( T \)-descending as tuple ordering. Instead of passively waiting for the underlying subtree to fully materialize all the groups, by the Progressive Condition of Requirement 1, the rankaggr operator will actively determine 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 1, 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 rankaggr operator in the order of the ranking 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 a fundamental change in the iterator interface of query operators. More specifically, we change the GetNext method of the new iterator interface 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 rankaggr 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 being invoked, the same \( g \) is the parameter to invoke its lower operator. For a binary operator such as join, \( g \) is divided into two components \( g' \) and \( g'' \) as the parameters to invoke the left and right child operators. As the response, each operator sends the next output tuple from the designated group \( g \) to its upper operator.

**Example 5:** Consider again \( F = G(T) = \sum(R,v) \) on relation \( H \) and \( k = 1 \), as in Example 2. 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 rankaggr operator above it. The tuples within each group are produced in their on-disk order. The execution of that plan is already explained in Example 2. Note that tuples in \( g_2 \) are not exhausted and tuples in \( g_3 \) are not touched at all.

---

\footnote{Correction: in an earlier version submitted for publication, we mistakenly wrote \( H^\alpha_g = \arg \min_{l \in [1:n]} \{ \sum_{gi \in \{ t_1, \ldots, t_k \}} [g] < \theta \}. \)
3.2 Rank-Awareness: Exploiting the Tuple-Ranking Principle

To follow Requirement 2, the tuples within each group must be produced in the optimal order from \( T^{\text{ascending}} \), \( T^{\text{descending}} \), or \( T^{\text{hybrid}} \). As Section 2.3 explained, we choose \( T^{\text{descending}} \) as the implementation heuristic. To enforce \( T^{\text{descending}} \), the query tree underlying rankagg must be rank-aware as well. For this purpose, we can leverage the recent work on rank-aware operators and plans [21, 23] (without grouping). In addition, however, we must address the challenges in satisfying group-awareness and rank-awareness together.

**Example 6:** Consider our running example again. To contrast with Example 5, Figure 5 illustrates (a) a plan that is 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 execution of this plan is already explained in Example 3. Note that it takes fewer steps than the GroupOnly plan in Figure 4 to output \( g_1 \) as the top-1.

 Processing ranking aggregate queries in the above framework imposes several challenges, which we address in the rest of this section. First, we devise a new rankagg operator that is able to determine the most promising group, based on maximal-possible scores (Section 3.3). Second, governed by rankagg, we enhance other operators in the underlying subtree to comply to the parameterized GetNext interface. This interface change mandates the new implementations of group-aware query operators. Finally, we discuss how to achieve group-awareness and rank-awareness at the same time for the scan operator (Section 3.4) and the join operator (Section 3.5), respectively.

3.3 The New rankagg Operator

The iterator methods for rankagg are shown in Figure 6. (The routines init\_ub, update\_ub, and final\_ub are given in Figure 7). The rankagg operator maintains a priority queue storing the upper-bounds of all the groups that are not output yet. The upper-bound of each group is initialized by assuming the highest possible score for every tuple in the group. The rankagg operator always gets the next tuple from the top group of the priority queue. When a tuple is obtained, the upper-bound of the corresponding group is updated. When the top group is complete, it is guaranteed to be the best among those in the queue, thus can be removed from the queue and reported.

Three questions must be answered to enable rankagg: First, how to obtain the valid groups and their sizes? Second, how to update the upper-bound of a group \( g, T_{\pi_g}[g] \), with respect to the obtained tuples \( I_g \)? Finally, how to manage the priority queue? We address these issues below.

Lazy materialization of the priority queue: The priority queue for groups’ upper-bounds can be too big to fit in memory. When the priority queue becomes so big that we have to store it in secondary storage, the performance can be degraded significantly because the elements in the queue must be updated frequently. Hence, we design a lazy materialization scheme for manipulating the priority queue efficiently, shown in Figure 8. It implements a virtual priority queue \( q \) using (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. The size of each group is maintained in data cube.

Initially, only a small number (1000 in our experiments) of groups with the largest sizes are initialized from the data cube and 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 \( g \), the top elements

**Figure 6. The interface methods of rankagg.**

1: //input: the underlying operator.
2: //k: the requested number of groups.
3: //q: the priority queue of groups.
4: //q.count: the number of obtained tuples in \( g \), i.e., \( |I_g| \).
5: //q.size: the size of \( g \).

Procedure Open()
1: //input.Open(); q.clear()
2: for each group \( g \) do
3: \( \text{init}\_\text{ub}(g); q.\text{insert}(g) \)
4: return

Procedure GetNext()
1: while true do
2: if \( k == 0 \lor q.\text{isEmpty}() \) then
3: \( \text{Close}() \)
4: return
5: \( g \leftarrow q.\text{top}() \)
6: if \( g.\text{count} == g.\text{size} \) then
7: \( \text{final}\_\text{ub}(g); k \leftarrow k - 1 \)
8: return \( g \)
9: \( t \leftarrow \text{input.GetNext}(g); \text{update}\_\text{ub}(g,t); q.\text{insert}(g) \)

Procedure Close()
1: //input.Close(); q.clear()
2: return

**Figure 7. The upper-bound for \( g = \text{sum} \).**

Lazy materialization of the priority queue: The priority queue for groups’ upper-bounds can be too big to fit in memory. When the priority queue becomes so big that we have to store it in secondary storage, the performance can be degraded significantly because the elements in the queue must be updated frequently. Hence, we design a lazy materialization scheme for manipulating the priority queue efficiently, shown in Figure 8. It implements a virtual priority queue \( q \) using (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. The size of each group is maintained in data cube.

Initially, only a small number (1000 in our experiments) of groups with the largest sizes are initialized from the data cube and 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 \( g \), the top elements
from $q'$ and from every buffer block are compared and the overall top group $g$ 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 that the upper-bound of the top group $g$ becomes smaller than the upper-bound of the unseen group (in the data cube) with the largest size. Under such situation, the next batch (1000 in our experiments) of groups with the largest sizes are initialized from the data cube and inserted into $q'$. With this scheme, only the top elements are kept in memory. Moreover, most groups in the cube will never be touched. Therefore disk access will be rare.

**Grouping statistics**: In a data warehousing environment, decision support queries operate on non-operational data, where join and grouping conditions are commonly stable for daily usages. As we discussed in Section 1, data cube provides useful summary information such as the size (count) of each group. Index structures are pervasively built upon such summary data.

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 aggregation of multiple groups (e.g., (city='Chicago', month=May) and (city='Chicago', month=June)) on the data cube. Therefore the size of each aggregated group must be obtained first. Given that a concept hierarchy can be built on each dimension in the data cube to allow multiple levels of summarization, such pre-processing can be performed efficiently. Moreover, as we discussed above, only the top groups are ever touched. Techniques for getting the groups with the largest sizes (incrementally) with selection conditions over dimensional attributes are introduced in [25].

**Computing the upper-bound of a group**: At the beginning, we need to give an initial upper-bound score to each group, based on the group size. For example, if $G=sum$, the initial upper-bound of a group $g$ with $g.size$ can be defined as $g.size \times \overline{T}_g$, where $\overline{T}_g$ is the maximum-possible value of $T$ among $g$’s tuples. The $\overline{T}_g$ can be obtained either by application semantic (e.g., according to the definition of $T$), or by the index over the data. 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 $T$.) More information about such index is discussed in Section 3.4.

Starting from the initial upper-bound, we must keep updating $\overline{F}_{x,h}[g]$ when tuples are incrementally obtained. When the last tuple from $g$ is obtained, $\overline{F}_{x,h}[g]$ indeed becomes the aggregate value $F[x,h][g]$. This description clearly indicates that the upper-bound itself can be maintained by a user-defined aggregate function. (Let us call it the upper-bound function.) In fact, the above steps are exactly how modern database systems support user-defined aggregate functions. For example, in PostgreSQL, a user-defined aggregate function is defined by an initial state value, a state transition function, and a final calculation function. Therefore, for the aggregate function $G$ in the query, the corresponding upper-bound function is provided as routines in the interface methods of rankaggr (Figure 6). Such routines can be pre-defined if $G$ is a built-in function such as $sum$, $min$, $max$, $avg$, standard deviation, and variance. For example, Figure 7 illustrates the upper-bound function for $G=sum$. When $G$ is a user-defined aggregate function itself, the upper-bound function can be defined by just using the utilities (initialization, state transition, final calculation) of $G$ or with straightforward adaptation. We omit further discussion of these details.

### 3.4 Modifying the Scan Operator

To be group-aware, the new scan operator must access the next tuple in the requested group $g$ given by its upper operator. In OLAP environments, index structures are commonly built upon the base tables using their grouping attributes as index keys. In [20], 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, must be maintained on the index to enable such striding. A cursor must be forwarded whenever a tuple is obtained from the cursor. However, there are several important differences, as discussed below.

**First**, in our case, index retrieval is governed by the dimensionally designated group instead of fixed weights. **Second**, 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:

```sql
Select R, g, S, SUM(R.v+S.v) From R, S Group By R, g, S 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., [17]) 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.

3.5 Modifying the Join Operator

The join operator must be both group-aware and rank-aware. For group-awareness, when a join operator is required to produce a tuple in the 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. For instance, for the query in Section 3.4, suppose a join operator \(op\) joins tuples from tables \(R\) and \(S\). When \(op\) is requested by \(rankaggr\) to output the next tuple from a group \((R.g=1, S.g=2)\), \(op\) 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 [21]. HRJN maintains a threshold value that gives an upper-bound of the score of all join combinations not yet seen, which is continuously updated as new tuples arrive. The algorithm maintains a ranking queue 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 the computed threshold. Otherwise, the algorithm continues by reading tuples from the inputs and performs a symmetric hash join to generate new join results. 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.

3.6 Alternative \(top-k\) Aggregate Query Plans

The group-aware and rank-aware query operators enable new plans that can be much more efficient than traditional plans. We call them \(GroupRank\) plans (e.g., Figure 5(a)). However, there can be various situations under which these plans are inapplicable or inefficient, therefore we propose variations of the operators for constructing alternative plans. We study the performances of these plans in Section 4.

First, \(GroupOnly\) plans (e.g., Figure 4(a)), where the operators are group-aware but not rank-aware, are applicable when \(T\) is not monotonic (21, 23) assume \(T\) is monotone) or indices are only available on the grouping attributes. To be more specific, in such plans, the \(rankaggr\) operator still gets the next tuple from the most promising group, but in arbitrary order within each group. For example, in the \(update_{db}\) procedure of Figure 7, the upper-bound should become \(g.ub \leftarrow g.ub + (g.size \times g.count) \times T_g\). The join operator uses a FIFO queue instead of priority queue to buffer join results (thus HRJN becomes the hash ripple join [18]). The scan operators stride on the grouping attributes only.

Second, \(RankOnly\) plans where the operators are rank-aware only, are applicable when indices are unavailable for the grouping attributes. Instead of telling the underlying operator the designated group, the \(rankaggr\) operator gets interleaved tuples from all groups, ordered by their aggregate scores. A hash table instead of priority queue is used to give fast access to the upper-bounds of groups for the obtained tuples. An iteration through the hash table is performed periodically (for every 1000 obtained tuples in our experiments). The group with the highest upper-bound is output when it is complete. The other operators become the same as the rank-aware operators in [21, 22, 23].

Finally, \(GroupRank-\epsilon(0 \leq \epsilon \leq 1)\) 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. The join operator in Section 3.5 buffers the joined tuples in the ranking queues. Maintaining the ranking queues can bring significant overheads or even offset the advantages of processing joins incrementally. Therefore in \(GroupRank-\epsilon\) plans, upon the request of sending the next tuple from a given group, a join operator outputs the top tuple \(t\) in the 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 \(GroupRank-\epsilon\) 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.

4 Experiments

We implemented our proposed techniques in \textit{RankSQL} [23, 24], a rank-aware relational database system based on PostgreSQL. The experiments are conducted on a PC with 2.8GHz Intel Xeon SMP (dual hyperthreaded CPUs each with 512KB cache), 2GB RAM, and 260GB RAID5 array of 3 SCSI disks, running Linux 2.6.9 operating system. 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. We use the star-join query \(Q\) in Section 1.

We compare five execution plans, Traditional, \(RankOnly\),
GroupOnly, GroupRank (i.e., GroupRank-1), and GroupRank-0. These plans have the same tree structure that joins $A$ and $B$ first and then joins with $C$. The difference is that Traditional uses normal grouping and sorting operators, while other plans use the new rankagg operator. Moreover, the operators in RankOnly, GroupOnly, GroupRank, and GroupRank-0 are group-aware and/or rank-aware, as described in Section 3.6.

We executed these plans under various configurations on 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 (with random variations) ($s$), the join selectivity ($j$, e.g., $j=0.0001$ means that there are 10,000 distinct values of each join attribute). Note that $g$ represents the number of different values of attribute $g$ on each individual table. For example, when $g=10$, the maximum number of overall joined groups is 1, 000.

We performed 4 sets of experiments. In each set, we varied the value of one parameter and fixed the values of the other three parameters. We measured the execution time of each plan under these parameter settings. The results are shown in Figure 9. (Note that 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. The performance of Traditional is only comparable to the efficient 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 one configuration ($g=10,000$) in Figure 9(a) (It is excluded from Figure 9(c) for the same reason.) As an intuitive explanation, if the top-1 group has a member tuple that is ranked in the last place, all the groups must be materialized in order to obtain the top-1 group. This indicates that being rank-aware itself [21, 23] does not help in dealing with top-$k$ aggregate queries. GroupOnly, GroupRank and GroupRank-0 always outperformed Traditional and RankOnly. The differences among these three plans are not obvious in Figure 9(a)(b)(d) because Traditional and RankOnly are too far off the scale. However, Figure 9(c) clearly illustrates their differences. As we observe, GroupOnly in many cases is better than GroupRank, which verifies that the overhead of ranking can offset the advantages of group-awareness in certain cases. On the other hand, we get much improved performance 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 10 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 3.6. 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 exploited GroupRank-$\epsilon$ by ranging $\epsilon$ from 0 to 1, as shown in Figure 11. Note that GroupRank and GroupRank-0 are the two extreme cases where $\epsilon$ is 1 and 0. Interestingly none of them is the best in the range, which indicates the choice of $\epsilon$ should be captured by the query optimizer.

Figure 12 compares the number of joined groups touched by GroupRank-0 and Traditional, which touches every group, under the two settings in Figure 10. It verifies that most of the groups never need to be touched by the new plans, as explained in Section 3.3.

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, individually covering different situations where they are applicable, as discussed in Section 3.6. 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 perfor-
5 Related Work

In contrast, existing techniques are not designed for the

...
to support the group-aware and rank-aware scan operator.

The work closest to ours is [25], 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 pre-determined aggregate function and aggregated expression, lacking the ability to support ad-hoc ranking aggregate queries. However, it is complementary to our work as it can be used to obtain the group sizes (count as pre-determined aggregate function) when there are selection conditions over the dimensional attributes, as discussed in Section 3.3.

6 Conclusion

We introduced a principled and systematic framework to support ad-hoc top-k (ranking) aggregate queries in OLAP efficiently. As the foundation, we develop the Group-Ranking Principle and Tuple-Ranking Principle, which dictate the group-ordering and tuple-ordering requirement that together realize optimal aggregate query processing. Guided by the principles, we proposed a new execution model that enables efficient top-k aggregate query plans. We addressed the challenges in implementing the model and the new query operators. The experiment results validate our framework by showing significant performance improvement. To the best of our knowledge, this is the first piece of work that provides efficient support of ad-hoc top-k aggregation. Thus the techniques address a significant research challenge and can be very useful in many data warehousing and decision support applications.

References

Appendix

A Proof of Property 1

Property 1 (Group-Ranking Principle): Let \( g_1 \) be any group in the current top-\( k \) ranked by maximal-possible scores \( \mathcal{F} \) 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-\( k \) groups are the query answers if they are all fully evaluated.

Proof: Suppose at any moment, the current top-\( k \) groups are \( \mathcal{K} \), such that \( \mathcal{F}_{\mathcal{I}_g}(g_1) > \mathcal{F}_{\mathcal{I}_g}(g_2), \forall g_1 \in \mathcal{K} \) and \( \forall g_2 \notin \mathcal{K} \). Assume there is no tie in scores since ties can be broken by the “tie-breaker” mentioned in Section 1.

For any incomplete group \( g_1 \in \mathcal{K} \) (\( \mathcal{I}_{g_1} \subset g_1 \)), further processing \( g_1 \) is necessary, otherwise \( g_1 \) remains belonging to \( \mathcal{K} \) (based on Eq. 2, \( \mathcal{F} \) never increases as more tuples are obtained) and incomplete, resulting in that it is impossible to get the real top \( k \) groups and their exact aggregates. \( g_1 \) may be one of the real top \( k \) groups since it remains belonging to \( \mathcal{K} \); and the exact aggregate of \( g_1 \) cannot be obtained since it is not fully evaluated.

Given any such \( g_1 \in \mathcal{K} \) and \( g_2 \notin \mathcal{K} \), since \( \mathcal{F}_{\mathcal{I}_g}(g_1) > \mathcal{F}_{\mathcal{I}_g}(g_2) \), whatever score \( g_2 \) can achieve, \( g_1 \) can possibly do better. Thus, further processing \( g_2 \) (when \( g_2 \) is incomplete) may not be necessary, since \( \mathcal{K} \) may turn out to be the real top \( k \) groups, i.e., \( \mathcal{F}[g_1] > \mathcal{F}[g_2], \forall g_1 \in \mathcal{K} \). Further processing of \( g_2 \) can become necessary only if we further process some incomplete group \( g_1 \in \mathcal{K} \) such that \( g_2 \) belongs to the updated \( \mathcal{K} \).

When all the groups in \( \mathcal{K} \) are complete, we can sufficiently declare \( \mathcal{K} \) as the real top \( k \) groups, because \( \mathcal{F}[g_1] = \mathcal{F}_{\mathcal{I}_g}(g_1) > \mathcal{F}_{\mathcal{I}_g}(g_2), \forall g_1 \in \mathcal{K} \) and \( \forall g_2 \notin \mathcal{K} \).

B Proof of Property 2

We present the following lemma before the proof.

Lemma 1: With respect to a ranking aggregate \( \mathcal{F} = \mathcal{G}(T) \), let the lowest top-\( k \) group score be \( \theta \). At any moment, suppose \( \mathcal{K} \) is the current top-\( k \) groups ranked by maximal-possible scores \( \mathcal{F} \), we have \( \mathcal{F}[g] \geq \theta, \forall g \in \mathcal{K} \).

Proof: Suppose the set of the real top \( k \) groups is \( \mathcal{K}' \). For any \( g' \in \mathcal{K}' \), at any moment, \( \mathcal{F}[g'] \geq \mathcal{F}[g''] \geq \theta \), and thus there are at least \( k \) groups with maximal-possible scores over \( \theta \). Therefore for the current top-\( k \) groups \( \mathcal{K} \) at the moment, \( \mathcal{F}[g] \geq \theta, \forall g \in \mathcal{K} \).

Property 2 (Tuple-Ranking Principle): With respect to a ranking aggregate \( \mathcal{F} = \mathcal{G}(T) \), let the lowest top-\( k \) group score be \( \theta \). For any group \( g \), let \( H^\alpha_g \) be the tuple depth with respect to tuple order \( \alpha \): \( t_1 \rightarrow \cdots \rightarrow t_n \), when the inter-group ordering follows Requirement 1.

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

\[
H^\alpha_g = \min_{l \in [1,n]} \{ l | \mathcal{F}(t_1,\ldots,t_l)[g] < \theta \},
\]

or otherwise \( H^\alpha_g = n \) if such a depth does not exist.

- (\( T \)-based Ranking) To find the optimal order \( \alpha \) that results in the minimum \( H^\alpha_g \), we need to only consider three classes of orders: \( T \)-ascending, \( T \)-descending, and \( T \)-hybrid, all of which are rankings by \( T \) scores.

Proof:

Order Independence: By definition, \( H^\alpha_g \) is the number of tuples retrieved from \( g \) when the query processing stops. If Requirement 1 is followed, by the Stop condition, when we stop and conclude the real top \( k \) groups, say \( \mathcal{K}' \), \( g \) is complete \( \forall g \in \mathcal{K}' \) and therefore \( H^\alpha_g = n \).

For any \( g \notin \mathcal{K}' \), if it has been accessed to some depth \( l \), \( t_1, \ldots, t_l \) are the accessed tuples since the order is \( \alpha: t_1 \rightarrow \cdots \rightarrow t_n \). We now prove the tuple depth \( H^\alpha_l \) is the smallest \( l \) that makes \( \mathcal{F}(t_1,\ldots,t_l)[g] < \theta \). On the one hand, \( H^\alpha_g \) cannot be smaller than such \( l \), otherwise \( \mathcal{F}(t_1,\ldots,t_l^l)[g] > \theta \) (say, \( H^\alpha_g = l' < l \)). (Note that we assume no ties in scores.) Under such situation, we could not have concluded \( \mathcal{K}' \) as the real top \( k \) groups if Requirement 1 is followed, since the Progressive condition would require to further process some incomplete group such as \( g \). (Remember \( \theta \) is the lowest \( \mathcal{F}[g] \) among groups in \( \mathcal{K}' \).) On the other hand, \( H^\alpha_g \) cannot be larger than such \( l \), because we already have \( \mathcal{F}(t_1,\ldots,t_l)[g] < \theta \) when \( l \) tuples from \( g \) are retrieved. Under such situation, \( g \) cannot get into the current top \( k \) groups \( \mathcal{K} \) anymore (Lemma 1). By the Progressive condition of Requirement 1, \( g \) cannot get any chance to be further processed.

\( T \)-based Ranking: We prove that for any rest order, there is always a better order in \( T \)-descending, \( T \)-ascending, or \( T \)-hybrid. To prepare, we formally define the space of all \( T \)-hybrid orders as

\[
\Omega_h = \{ \alpha | \alpha : t_1 \rightarrow \cdots \rightarrow t_n, \forall i, (\forall j > i, T[t_i] \leq T[t_j] \lor \forall j > i, T[t_i] \geq T[t_j]) \}.
\]

That is, given any \( \alpha \in \Omega_h \), the “next” tuple obtained at each step is the one with either the highest or the lowest score among the unseen tuples. Therefore \( T \)-descending and \( T \)-ascending are special instances in \( \Omega_h \) since they always obtain the tuple with the highest (\( T \)-descending) or lowest (\( T \)-ascending) score. Hence, we now only need to
prove that for any rest order, there is always a better $T$-hybrid order.

As the complement of $\Omega$, the space of all orders in the rest class is defined as

$$\Omega_r = \{\alpha' : t_1 \rightarrow \cdots \rightarrow t_n, \exists x,j,k : x > k \land k > x \rightarrow T[t_x] > T[t_j] \land T[t_k] < T[t_k]\}.$$ 

That is, given any $\alpha' \in \Omega_r$, there exists at least one “next” tuple obtained at some step that is the highest or the lowest score among the unseen tuples. (Without such “middle” tuple, $\alpha'$ becomes a hybrid order.)

There can be multiple instances of such “middle” tuple $t_x$ at various steps of the tuple sequence of $\alpha'$. Without loss of generality, let us focus on the first instance of such $t_x$. Before that retrieval, suppose $T_g$ is the set of retrieved tuples and the maximal-possible score for unseen tuples is $T_g$ (Eq. 2). After the retrieval, the maximal-possible score of unseen tuples is unchanged, i.e., $T_{g\cup\{t_x\}} = T_g$. Therefore

$$\mathcal{F}_{g \cup \{t_x\}}[g] = \mathcal{F}_g[g] \quad \text{if } t_x \in \mathcal{I}_g \text{ (seen tuples)};$$

$$\mathcal{F}_{g \cup \{t_x\}}[g] = \mathcal{F}_g[g] \quad \text{if } t_x = t_j \text{ (just retrieved); } \forall t_i \in g \} \quad \text{otherwise (unseen tuples).}$$

However, if the one with the lowest score among unseen tuples, $t_\perp = \arg\min_{t \in \mathcal{I}_g \cup \{t_x\}} T[t]$ is instead of $t_x$ was retrieved, we have $\mathcal{F}_{g \cup \{t_x\}}[g] < \mathcal{F}_{g \cup \{t_x\}}[g]$ according to the following Eq. 6, which is the same as Eq. 5 except that $T[t_\perp] < T[t_x]$. Therefore

$$\mathcal{F}_{g \cup \{t_x\}}[g] = \mathcal{F}_g[g] \quad \text{if } t_x \in \mathcal{I}_g \text{ (seen tuples)};$$

$$\mathcal{F}_{g \cup \{t_x\}}[g] = \mathcal{F}_g[g] \quad \text{if } t_x = t_j \text{ (just retrieved); } \forall t_i \in g \} \quad \text{otherwise (unseen tuples).}$$

Now we show that we can convert any $\alpha_0 \in \Omega_r$ into a hybrid order $\alpha_m \in \Omega_h$ by a series of transformations, $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_m$, and $H_{\alpha_0} \leq H_{\alpha_m}$. In each transformation from $\alpha_i$ to $\alpha_{i+1}$, we find the first instance of such “middle” tuple $t_x$ in $\alpha_i$, and swap the position of $t_x$ and $t_{\perp}$, i.e.,

$$\alpha_i : \{t_1, \ldots, t_x, \ldots, t_{\perp}, \ldots \} \Rightarrow \alpha_{i+1} : \{\ldots, t_x, \ldots, t_{\perp}, \ldots \}.\]$$

According to Eq. 5 and Eq. 6, the value of $\mathcal{F}_{g \cup \{t_x\}}[g]$ at each step of $\alpha_{i+1}$ is equal to or smaller than that of $\alpha_i$, therefore $H_{\alpha_{i+1}} \leq H_{\alpha_i}$ based on Eq. 4. Thus $H_{\alpha_m} \leq H_{\alpha_0}$. Moreover, for a group of $n$ tuples, such transformation ends in at most $n-1$ steps, therefore we reach a better hybrid order. The reason is as follows. Since $t_x$ is the first instance of “middle” tuple in $\alpha_i$, the retrieved tuple at each step before $t_x$ is the one with either the highest or the lowest score among the unseen tuples. Suppose there are $s$ tuples before $t_x$ in the sequence of $\alpha_i$. Then after the transformation from $\alpha_i$ to $\alpha_{i+1}$, there are at least $s+1$ tuples before the first instance of “middle” tuple in $\alpha_{i+1}$, and finally there is no such instance in $\alpha_m$.

C Proof of Theorem 1

Theorem 1 (Optimal Aggregate Processing): If query processing follows Requirements 1 and 2, the number of tuples processed across all groups, i.e., $\Sigma g H_g$, is the minimum possible for query answering.

Proof: Among all the tuple orders that follow Requirement 1, for any group $g$, according to Property 2, the number of processed tuples $H_g$ is minimized when Requirement 2 is followed as well, i.e., $H_g = H_g^*$ where $\alpha$ is the optimal intra-group order for $g$.

Moreover, this $H_g^*$ is indeed the smallest number of $g$’s tuples to retrieve under any total tuple order, because any smaller number would result in $\mathcal{F}[g] \geq \theta$ so that we can neither get the exact aggregate of $g$ (if $g$ belongs to the real top $k$ groups) nor conclude $g$ does not belong to the real top $k$. In other words, the optimality of $\alpha$ is independent from whether Requirement 1 is followed or not because $\alpha$ is the optimal among all possible orders of retrieving $g$’s tuples. In summary, the minimal total number of retrieved tuples is $\Sigma g H_g^*$ and this minimum is achieved when both Requirement 1 and 2 are followed.

D Implementation Heuristic: T-descending Order

Below we show that $T$-descending is the best choice of intra-group tuple order in practice for several typical score distributions (e.g., uniform and normal) and aggregate functions (e.g., sum and avg).

First, for uniform distribution, $T$-descending is at least as good as $T$-ascending, according to the corollary below. It indicates that $T$-descending and $T$-ascending have the same expectations of maximal-possible scores when they retrieve the same number of tuples, therefore their tuple depths under any given top-$k$ threshold $\theta$ are (statistically) the same.

Corollary 1 (Tuple Order for Uniform Distribution):

Suppose $T[t_i]$ for group $g = \{t_1, \ldots, t_n\}$ follows the uniform distribution and the aggregate function $G$ is sum or avg. The expectations of maximal-possible scores for $T$-descending and $T$-ascending, with respect to the same number of retrieved tuples $m$, are equal, i.e., $E(\mathcal{F}_{T_x}^E[g] - \mathcal{F}_{T_x}^E[g]) = 0$ where $|I_g^E| = |I_g^E| = m$, $\forall 0 < m \leq n$.

Proof: Consider the case when $G$ is sum (avg is similar). Suppose $\mathcal{F}_{T_x}^E[g] = (n-m) \times \max + \sum_{i=m+1}^n T[t_i]$ and $\mathcal{F}_{T_x}^E[g] = (n-m) \times \max + \sum_{i=m+1}^n T[t_i]$. Therefore

$$E(\mathcal{F}_{T_x}^E[g] - \mathcal{F}_{T_x}^E[g])$$

$$= E(\sum_{i=m+1}^n T[t_i] - \sum_{i=m+1}^n T[t_i] + (n-m)(\max - T[t_m]))$$

Appendix-2
Based on the results from Appendix-3, the tuple score oscillates around $|I|^g_d$, in the range of $[0 : 1]$. In Figure 13 the curve for $|I|^g_d$-ascending is as good as $|I|^g_d$-descending for uniform distribution, i.e. $|I|^g_d$-descending is a special case of $|I|^g_d$-ascending, although $|I|^g_d$-ascending and $|I|^g_d$-descending are empirically compared. The reason is such optimization technique requires descending although we empirically compare $|I|^g_d$-ascending and $|I|^g_d$-descending. Figure 13 shows that $|I|^g_d$-ascending is as good as $|I|^g_d$-descending for uniform distribution, i.e. $|I|^g_d$-descending is a special case of $|I|^g_d$-ascending, although we empirically compare $|I|^g_d$-ascending and $|I|^g_d$-descending.

To get the same or lower upper-bound as $|I|^g_d$-ascending (p), $|I|^g_d$-descending (q), and $|I|^g_d$-hybrid (y) order.

In our experiments, we make the decision based on which tuples, depending on runtime optimization technique. In the examples above, we empirically compare $|I|^g_d$-ascending, $|I|^g_d$-descending, and $|I|^g_d$-hybrid. The reason is such optimization technique requires descending although we empirically compare $|I|^g_d$-ascending and $|I|^g_d$-descending. In the examples above, we empirically compare $|I|^g_d$-ascending, $|I|^g_d$-descending, and $|I|^g_d$-hybrid. The reason is such optimization technique requires descending although we empirically compare $|I|^g_d$-ascending and $|I|^g_d$-descending. The reason is such optimization technique requires descending although we empirically compare $|I|^g_d$-ascending and $|I|^g_d$-descending. The reason is such optimization technique requires descending although we empirically compare $|I|^g_d$-ascending and $|I|^g_d$-descending.