ARCube: Supporting Ranking Aggregate Queries in Partially Materialized Data Cubes*

Tianyi Wu
University of Illinois, Urbana-Champaign
twu5@uiuc.edu

Dong Xin†
Microsoft Research
dongxin@microsoft.com

Jiawei Han
University of Illinois, Urbana-Champaign
hanj@cs.uiuc.edu

ABSTRACT

Supporting ranking queries in database systems has been a popular research topic recently. However, there is a lack of study on supporting ranking queries in data warehouses where ranking is on multidimensional aggregates instead of on measures of base facts. To address this problem, we propose a query execution model to answer different types of ranking aggregate queries based on a unified, partial cube structure, *ARCube*. The query execution model follows a candidate generation and verification framework, where the most promising candidate cells are generated using a set of high-level guiding cells. We also identify a bounding principle for effective pruning: once a guiding cell is pruned, all of its children candidate cells can be pruned. We further address the problem of efficient online candidate aggregation and verification by developing a chunk-based execution model to verify a bulk of candidates within a bounded memory buffer. Our extensive performance study shows that the new framework not only leads to an order of magnitude performance improvements over the state-of-the-art method, but also is much more flexible in terms of the types of ranking aggregate queries supported.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Experimentation, Performance

*The work was supported in part by the U.S. National Science Foundation NSF IIS-05-13678 and BDI-05-15813. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

†This work was done when the author was at the University of Illinois, Urbana-Champaign.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

Keywords

Data cube, partial materialization, ranking aggregate queries

1. INTRODUCTION

Ranking (or top-k) query processing [7, 14, 6, 9, 21, 24, 12, 4, 32, 27] has become an increasingly important problem for many applications in a variety of fields. While considerable research has been devoted to ranking query processing in Web search, data integration, multimedia data management, *etc.*, it has not been well-studied in the context of data warehousing and OLAP, including many important applications like business intelligence, decision support, interactive and explorative data analysis, *etc.*, where efficient methods are needed for producing ranked answers to aggregate queries in multidimensional analysis.

Example 1. Consider a product manager who is analyzing a sales database which stores the nationwide sales history organized by location and time. The manager may pose the following queries: *"What are the top-10 (state, year) cells having the largest total product sales?"* and he may further drill-down and ask *"What are the top-10 (city, month) cells?"* in order to make his investment decisions. ■

Example 2. Consider an organization donation database, where donators are grouped by "age", "income", and other attributes. Interesting questions include: *"Which age and income groups have made the* top-k average *amount of donation (per-donor)?"* and *"Which income group of donators has the* largest standard deviation *in the amount of donation?* ■

The above questions are examples of "*ranking aggregate queries*", characterized by the ranking on the aggregation of multidimensional data. It is challenging to process ranking aggregate queries efficiently due to the difficulties arising from both ranking and multidimensional analysis. Unfortunately, many existing ranking methods for traditional database systems [7, 21, 24] are not efficient for answering such queries, where data tuples must be aggregated before any ranked result is produced. On the other hand, from the perspective of OLAP and data warehousing, although data cube [17, 10], relying on data precomputation and materialization, has been a well-developed model to handle multidimensional analysis, its current techniques [19, 34, 18, 30, 22] do not address ranking analysis directly.

The most relevant studies on this problem fall into either one of the two categories: (1) *no materialization*, and

(2) *full materialization*. In the first category, the *rankagg* framework [23] is proposed to support ad-hoc ranking aggregate queries by making the query optimizer rank-aware and considering an ordering of tuple accesses during query execution. The maximum possible aggregate score is estimated for each incomplete cell that is not yet fully verified; a cell can be pruned if this score is no greater than the current top-k score. In the second category, the *Sorted Partial Prefix Sum Cube* [25] is proposed, and based on that, a query execution algorithm is developed to produce top-k answers to SUM queries for a given time range. The space usage of the cube is the same as that of a full data cube. In some cases, the current techniques may not assure satisfactory performance because, without materialization, it is expensive to compute aggregation from scratch, whereas maintaining a full cube could be cursed by high dimensionality. Moreover, existing methods often are tailored to predefined or monotone aggregation (*e.g.*, SUM and $COUNT$); therefore, flexible support for various other non-monotone aggregate measures such as AVG (*Average*) and $STDDEV$ (*Standard Deviation*) is needed.

Motivated by these observations, we propose a new method called *ARCube* (Aggregate-Ranking Cube), to address ranking aggregate queries flexibly and efficiently. The *ARCube*, in principle, seeks middle ground by adopting the *partial materialization* approach: it is a concise structure consisting of two types of materialized cuboids, *guiding cuboid* and *supporting cuboid*, where the former contains a number of guiding cells that provide concise, high-level data statistics to guide the ranking query processing, whereas the latter provides inverted indices for efficient online aggregation. Based on this structure, we develop a novel query execution model that exploits the interdependent relationships between cuboids. For ranking queries on a non-materialized cuboid, efficient online query execution is done via a *candidate generation-and-verification* framework, where the most promising top-k candidate cells are generated by combining a set of guiding cells, and their actual aggregate values are then verified using the inverted indices.

Consider, for example, a query asking for the top-10 most populated cities in the United States, and the population information is materialized only at the state-level (guiding cuboids). We can start by aggregating on cities in "New York", "California", and so on (candidate cells), which are likely to contain the top-k answers, instead of searching sparsely populated cities in "Alaska", "Wyoming", *etc.*. The intuition here is that "higher-ranking states imply higher-ranking cities with high probability"; in other words, the high-level, compact data statistics can be used to *guide* the query processing so that promising candidates are given higher priority. This intuition also applies to many other scenarios. For example, consider ranking in the DBLP dataset [1]. Finding the most productive authors (SUM aggregation) in a particular conference (such as "SIGMOD") can be guided by first looking at productive authors in the associated research field (such as "Database Systems").

The guiding cells, at the heart of the query execution framework, play the key role not only in guiding, but also in effective pruning. We identify for each guiding cell an *aggregate-bound*, which is derived from the materialized high-level statistics. By *bounding principle*, if a guiding cell's aggregate-bound is no greater than the current top-k aggregate value, then none of the candidates generated by this cell can be the top-k results, and therefore the cell can be pruned. As a result, under the unified framework, different types of aggregate query measures as well as complex measures can be flexibly supported using different aggregate-bounds. We will specifically discuss aggregate-bounds for a set of common measures including SUM, AVG, MAX and $STDDEV$.

To further optimize query processing, we develop a chunk-based execution algorithm that claims two advantages. First, while the current techniques must verify aggregates on a cell-by-cell basis, candidate cells in our framework are verified in bulky way and thus a memory buffer can be reused for efficient aggregation. Second, the query execution can be finished within a bounded buffer, as opposed to using a priority queue for all candidate cells: if the number of cells is large, the cost for maintaining the queue would be expensive.

The *ARCube* integrates ranking with aggregation, following the partial materialization cubing strategy that balances the cost and benefits of precomputation. To the best of our knowledge, such integration has not been studied before. Specifically, this paper has made the following contributions.

1. We develop the *ARCube*, a unified, partial cube structure to efficiently process different types of ranking aggregate queries. We show that this cube can be extended to support more complex ranking aggregate queries.
2. A query execution model is proposed based on the candidate generation and verification framework, and the bounding principle is developed for effective pruning. Under this framework, we further address the problem of efficient candidate verification by upgrading the original execution model to a chunk-based one that can utilize a bounded memory buffer. We also develop a chunk scheduling algorithm that can generate promising candidates and reuse the buffer simultaneously.
3. Encouraging results are shown in our extensive experiments conducted on both synthetic and the TPC-H decision support benchmark. The proposed method is much faster and more flexible than current techniques.

1.1 Problem Statement

Any ranking query problem can be characterized by either a maximization or a minimization criterion, which determines whether results with the highest or lowest scores are interesting. In this paper we study ranking queries with the maximization criterion, which is interesting in most database and OLAP scenarios.

The problem of aggregate ranking can be formalized as follows. Consider a fact table (or base relation), $R(\mathcal{A}, S)$, with d attributes, $\mathcal{A} = \{A_1, A_2, \ldots, A_d\}$, and a raw *score* attribute, S, with non-negative scores. A top-k aggregate query with an aggregate function $F^{\mathcal{Q}}_{agg}$, which is formulated on group-by attributes $\mathcal{A}^{\mathcal{Q}} = \{A^{\mathcal{Q}}_1, A^{\mathcal{Q}}_2, \ldots, A^{\mathcal{Q}}_q\}$ ($\mathcal{A}^{\mathcal{Q}} \subseteq \mathcal{A}, 1 \leq q \leq d$), asks for k cells $\{c_1, c_2, \ldots, c_k\}$ in the group-by view $R(\mathcal{A}^{\mathcal{Q}})$ such that for any other cell $c' \in R(\mathcal{A}^{\mathcal{Q}})$, $F^{\mathcal{Q}}_{agg}(c') \leq \min_{i=1}^{k} F^{\mathcal{Q}}_{agg}(c_i)$. Ties are broken arbitrarily.

In the remainder of the paper, we first present the structure of the *ARCube* in Section 2. In Section 3, we discuss the basic query execution algorithm and the chunk-based one, both using the SUM measure as an example. Additional ranking functions are discussed in Section 4. Section 5 reports the performance results, followed by a discussion of related work and extensions in Section 6. Finally, Section 7 concludes our study.

tid	A	B	C	*Score*
t1	a_1	b_1	c_3	63
t2	a_1	b_2	c_1	10
t3	a_1	b_2	c_3	50
t4	a_2	b_1	c_3	16
t5	a_2	b_2	c_1	52
t6	a_3	b_1	c_1	35
t7	a_3	b_1	c_2	40
t8	a_3	b_2	c_1	45

Figure 1: A sample database.

A	$s^1_{agg}(SUM)$
a_1	123
a_2	68
a_3	120

Figure 2: Guiding cuboid $C^{gd}(A, SUM)$.

B	$s^1_{agg}(SUM)$
b_1	154
b_2	157

Figure 3: Guiding cuboid $C^{gd}(B, SUM)$.

A	*Inverted Index*
a_1	$(t1, 63)$, $(t2, 10)$, $(t3, 50)$
a_2	$(t4, 16)$, $(t5, 52)$
a_3	$(t6, 35)$, $(t7, 40)$, $(t8, 45)$

Figure 4: Supporting cuboid $C^{sp}(A)$.

B	*Inverted Index*
b_1	$(t1, 63)$, $(t4, 16)$, $(t6, 35)$, $(t7, 40)$
b_2	$(t2, 10)$, $(t3, 50)$, $(t5, 52)$, $(t8, 45)$

Figure 5: Supporting cuboid $C^{sp}(B)$.

2. AR-CUBE STRUCTURE

In this section, we introduce the general partial cube structure, *ARCube*. The customization of the *ARCube* to different query measures and complex functions will be discussed in the next sections.

Definition 1. Given group-by attributes $\mathcal{A}' = \{A'_1, A'_2, \ldots, A'_{d'}\}$ ($\mathcal{A}' \subseteq \mathcal{A}, 1 \leq d' \leq d$) and m aggregate measures $\mathcal{M} = \{F^1_{agg}, \ldots, F^m_{agg}\}$ ($m \geq 1$), we define the **guiding cuboid** $C^{gd}(\mathcal{A}', \mathcal{M})$ as a list of entries as follows. Each entry has the format $\{g : s^1_{agg}, \ldots, s^m_{agg}\}$, where $g = (v_1, \ldots, v_{d'})$ is a distinct **guiding cell** in the group-by view $R(\mathcal{A}')$ and $s^i_{agg} = F^i_{agg}(g)$ ($1 \leq i \leq m$). That is, $s^1_{agg}, \ldots, s^m_{agg}$ are obtained by aggregating g on S using **guiding measures** F^1_{agg} through F^m_{agg}, respectively. ∎

Definition 2. Given group-by attributes $\mathcal{A}' = \{A'_1, A'_2, \ldots, A'_{d'}\}$ ($\mathcal{A}' \subseteq \mathcal{A}, 1 \leq d' \leq d$), we define the **supporting cuboid** $C^{sp}(\mathcal{A}')$ as a list of entries as follows. Each entry has the format $\{g : (t_1, s_1), \ldots, (t_{l_g}, s_{l_g})\}$, where $g = (v_1, \ldots, v_{d'})$ is a guiding cell in the group-by view $R(\mathcal{A}')$ and $(t_1, s_1), \ldots, (t_{l_g}, s_{l_g})$ is g's inverted index, which is a list of l_g (tid,$score$)-pairs corresponding to g's raw tuple id's and raw scores in $R(\mathcal{A}, S)$. ∎

Definition 3. Given a set of group-by's, $\mathcal{A}'_1, \ldots, \mathcal{A}'_D$, and a set of aggregate measures $\mathcal{M}$, an ***ARCube***, $\mathbb{C}(\mathcal{A}'_1, \ldots, \mathcal{A}'_D, \mathcal{M})$, consists of D guiding cuboids $C^{gd}(\mathcal{A}'_i, \mathcal{M})$ ($1 \leq i \leq D$) and D supporting cuboids $C^{sp}(\mathcal{A}'_i)$. ∎

Example 3. We illustrate part of an *ARCube* structure using an example. Figure 1 shows a sample data set R consisting of 8 base tuples and 3 attributes A, B, and C. For illustrative purposes, we create a *tid* column that shows the row number. The last column, *Score*, stores the raw score. Figure 2 depicts a guiding cuboid $C^{gd}(A, SUM)$, which contains 3 cells; in this cuboid, only *SUM* has been computed, *i.e.*, $\mathcal{M} = \{SUM\}$. Similarly, Figure 3 shows the guiding cuboid $C^{gd}(B, SUM)$. Figure 4 and 5 illustrate two supporting cuboids, $C^{sp}(A)$ and $C^{sp}(B)$. For instance, cell a_2 in Figure 4 has an inverted index of length 2 and the first element $(t4, 16)$ indicates that a_2 has a raw tuple id $t4$ and its raw score is 16. ∎

Given a data set R, we consider two materialization plans. The first is a baseline plan which materializes all cuboids with single dimensions, *i.e.*, $D = d$ and $\mathcal{A}_i = \{A_1, \ldots, A_d\}$. This plan is sufficient to support any query but may not be efficient. An alternative plan is that, given a cuboid size threshold θ_{CUBOID}, we keep all guiding cuboids in the cube space with cardinality no larger than θ_{CUBOID} and their corresponding supporting cuboids. For this plan, because guiding cuboids are at high-level storing aggregate information, they could fit in memory easily. Also, the overall materialization size would be much smaller than a full cube because only selected cuboids are materialized. More discussions on the materialization are deferred to Section 6.2.

3. QUERY EXECUTION FRAMEWORK

Existing database or data warehouse systems suffer from a lack of efficient support for ranking aggregate queries. In this section, we propose a novel query execution framework based on the *ARCube* and use *SUM* as an example. Systematic extensions of the framework to other aggregate measures and complex functions will be discussed in Section 4.

3.1 Query Execution: A Motivating Example

Example 4. We motivate the query model using an example continued from Example 3. Consider a query asking for the top-1 *SUM* (*i.e.*, $F^Q_{agg} = SUM$) aggregate grouped-by AB (*i.e.*, $A^Q = \{A, B\}$). Given any *ARCube*, there are two possibilities. First, if there exists a materialized guiding cuboid $C^{gd}(AB, \mathcal{M})$ such that $SUM \in \mathcal{M}$, then scanning the guiding cuboid once would answer the query efficiently. Otherwise, the result cannot be found from the materialization directly and we need to utilize materialized ancestor cuboids to help query processing. Suppose, as shown in Example 3, that $C^{gd}(A, SUM)$, $C^{gd}(B, SUM)$, $C^{sp}(A)$, and $C^{sp}(B)$ are materialized. Our intuition is that, because the two guiding cuboids store the aggregate information that is higher-level than what the top-k query is asking for, they can serve as guidance to finding the actual top-k results. The query execution process is illustrated in Figures 6 through 8. Figure 6 shows two **sorted lists**, labeled $\overline{F}_{agg}(a_i)$ and $\overline{F}_{agg}(b_j)$, respectively. Each sorted list contains a number of guiding cells and each cell g is associated with an aggregate value, which we call g**'s aggregate-bound** (or $\overline{F}_{agg}(g)$). We explain its meaning shortly.

Initially, as illustrated in Figure 6, the two sorted lists are obtained from $C^{gd}(A, SUM)$ and $C^{gd}(B, SUM)$. The aggregate-bound of each guiding cell is initialized to be the materialized *SUM* and the guiding cells in each list are sorted descendingly according to the aggregate-bound. Next, we pull out the top guiding cell from each list and combine them to generate the first **candidate cell**, (a_1, b_2). The intuition is straightforward as a_1 and b_2 have larger *SUM* than any other cell does. It is, however, not possible to terminate early until we verify its true aggregate value and make sure

$\overline{F}_{agg}(a_i)$	$\overline{F}_{agg}(b_j)$
$a_1 : 123$	$b_2 : 157$
$a_3 : 120$	$b_1 : 154$
$a_2 : 68$	
Candidate $(a_1, b_2) : 60$	

Figure 6: Initial sorted lists.

$\overline{F}_{agg}(a_i)$	$\overline{F}_{agg}(b_j)$
$a_3 : 120$	$b_1 : 154$
$a_2 : 68$	$b_2 : 97$
$a_1 : 63$	
Candidate $(a_3, b_1) : 75$	

Figure 7: Sorted lists after the first candidate verification.

$\overline{F}_{agg}(a_i)$	$\overline{F}_{agg}(b_j)$
$a_2 : 68$ (Pruned)	$b_2 : 97$
$a_1 : 63$ (Pruned)	$b_1 : 79$
$a_3 : 45$ (Pruned)	
Candidate: no more	

Figure 8: Sorted lists after the second candidate verification.

any unseen candidate cell has no greater aggregate. To verify the true aggregate, we turn to the inverted indices of a_1 and b_2 in $C^{sp}(A)$ and $C^{sp}(B)$, respectively. We retrieve their indices and intersect the two *tid*-lists, which results in $\{t2, t3\}$, and then compute the SUM of the raw scores over the intersecting *tid*-list, which results in $F^{Q}_{agg}(a_1, b_2) = 10 + 50 = 60$.

Having known that $SUM(a_1, b_2) = 60$, we can infer that $\sum_{j \neq 2} SUM(a_1, b_j) = SUM(a_1) - SUM(a_1, b_2) = 123 - 60 = 63$. This means that any unseen cell (a_1, b_j) $(j \neq 2)$ must satisfy $SUM(a_1, b_j) \leq 63$. Thus, we update the aggregate-bound $\overline{F}_{agg}(a_1)$ from 123 to 63 (Figure 7). For the same reason, $\overline{F}_{agg}(b_2)$ can be updated from 157 to 97, implying that $SUM(a_i, b_2) \leq 97$ $(i \neq 1)$. Now, we come to the following definition.

Definition 4. The **aggregate-bound** of a guiding cell g, $\overline{F}_{agg}(g)$, is the maximum possible aggregate (F^{Q}_{agg}) of any unseen candidate cell that could be combined by g. ∎

After the first candidate verification, as shown in Figure 7, the order of the guiding cells in the two sorted lists has changed due to the update. The top-1 cell from the two lists are now a_3 and b_1, respectively. We generate the second candidate (a_3, b_1) in the same fashion. To verify it, we intersect the *tid*'s of a_3 and b_1, which results in $\{t6, t7\}$ and $SUM(a_3, b_1) = 35 + 40 = 75$. Then, we update $\overline{F}_{agg}(a_3)$ to $120 - 75 = 45$ and $\overline{F}_{agg}(b_1)$ to $154 - 75 = 79$ and adjust the sorted lists, as shown in Figure 8. At the time, the maximum aggregate-bound in the sorted list for a_i, $\overline{F}_{agg}(a_2) = 68$, is no greater than the current top-1 aggregate value, 75. Although both $\overline{F}_{agg}(b_1)$ are $\overline{F}_{agg}(b_2)$ are still greater than 75, it is impossible to find any a_i such that $F^{Q}_{agg}(a_i, b_j)$ could be greater than 75. Therefore, we can terminate the query process and output $(a_3, b_1) : 75$ as the final top-1. ∎

In the example, there is a total number of 6 cells from the query group-by AB, whereas the query execution only touched 2 candidate cells without seeing the remaining 4. This is in contrast to the existing methods which must see all 6 cells, regardless of their raw scores or aggregate values, before termination.

3.2 Query Execution Algorithm

We now formally present the query execution algorithm. Consider a SUM query formulated on $\mathcal{A}^{\mathcal{Q}}$ asking for the top-k cells. We address the problem of answering the ranking query using a given query plan consisting of N $(1 \leq N \leq q)$ guiding cuboids, $C^{gd}(\mathcal{A}_1, SUM)$, ..., $C^{gd}(\mathcal{A}_N, SUM)$, and N supporting cuboids, $C^{sp}(\mathcal{A}_1)$, ..., $C^{sp}(\mathcal{A}_N)$. Assume that the query plan is valid: $\bigcup_{i=1}^{N} \mathcal{A}_i = \mathcal{A}^{\mathcal{Q}}$ and $\forall i \forall j, i \neq j \Rightarrow \mathcal{A}_i - \mathcal{A}_j \neq \phi$. Notice that the query plan selection problem is orthogonal to the query execution, and the baseline materialization plan of an *ARCube* guarantees that such a valid query plan exists.

The query execution algorithm QueryExec() is described in Table 1. The algorithm is made up of multiple iterations of candidate generation and verification. At the beginning, N sorted lists are initialized by scanning all the guiding cuboids into memory and $\overline{F}_{agg}(g)$ is initialized to be the materialized $SUM(g)$ for each guiding cell g (Line 1). θ indicates the k-th largest aggregate value seen so far (Line 2). Moreover, a candidate $\hat{c}$ is generated by combining the top guiding cell from each sorted list (Lines 3–4). This step is justified by two reasons. First, the rank of guiding cells can serve as good estimates of the rank of candidate cells so that large aggregate cells are likely to generate large candidate cells. Second, guiding cells with small aggregate-bounds are more likely to be pruned and should have low priority.

A slight technicality is that the top cells from the N sorted lists may not always be able to form a valid candidate (Line 5), because the candidate might have already been verified, or the top cells could not be combined (*e.g.*, (a_1, b_1) and (b_2, c_3)). In that case we would need to recursively increment the pointers to the sorted lists until the next non-verified candidate is found.

In addition to verifying $F_{agg}(\hat{c})$ for each candidate $\hat{c}$ (Line 6), the job of the candidate verification step also includes

Procedure: QueryExec()	
Input: Top-k, $\mathcal{A}^{\mathcal{Q}}$, $C^{gd}(\mathcal{A}_1, SUM), \ldots, C^{gd}(\mathcal{A}_N, SUM)$, $C^{sp}(\mathcal{A}_1), \ldots, C^{sp}(\mathcal{A}_N)$.	
Output: Top-k aggregate cells.	
Candidate generation	
1	[Sorted List Initialization] For each guiding cuboid $C^{gd}(\mathcal{A}_i, SUM)$ $(i = 1..N)$, create a sorted list;
2	$\theta \leftarrow 0$; Initialize the top-k priority queue;
3	For each sorted list $i = 1..N$, select the guiding cell g_i with the maximum $\overline{F}_{agg}(g_i)$. Terminate the algorithm if any sorted list is empty;
4	Generate a candidate cell $\hat{c}$ by combining $g_1, \ldots, g_N$;
5	WHILE $\hat{c}$ has been verified before DO $\hat{c} \leftarrow$ next candidate from the sorted lists; Terminate if no more $\hat{c}$ can be generated;
Candidate verification	
6	Verify $F_{agg}(\hat{c})$, the actual aggregate value of $\hat{c}$, by retrieving the inverted indices from the supporting cuboids;
7	Update the top-k priority queue and let θ be the current top-k threshold;
8	[Aggregate-Bound Updating] For each $i = 1..N$, $\overline{F}_{agg}(g_i) \leftarrow \overline{F}_{agg}(g_i) - F_{agg}(\hat{c})$;
9	For each sorted list $i = 1..N$, prune all guiding cells g where $\overline{F}_{agg}(g) \leq \theta$; Go to 3.

Table 1: Query execution algorithm for *SUM*.

updating the aggregate-bound (Line 8) and pruning (Line 9), which leads to the following lemmas.

Lemma 1. For an arbitrary guiding cell g, its aggregate-bound, $\overline{F}_{agg}(g)$, is *monotonically decreasing* during the query execution. ∎

Lemma 2. **(Bounding Principle)** Once a guiding cell g satisfies $\overline{F}_{agg}(g) \leq \theta$, g no longer qualifies to generate any candidate. ∎

Lemma 1 follows from the fact that the domain of the score attribute contains only non-negative values, while Lemma 2 can be justified by Definition 4.

The total cost of QueryExec() can be broken down into two parts. First, the disk access cost for fetching the N guiding cuboids. This part of cost is linear to the total number of guiding cells and is often not large because the guiding cuboids are at the higher-level of the cube lattice, having relatively fewer cells. Second, for each candidate verification, N random accesses are needed to fetch the inverted indices for the N guiding cells, respectively. If an inverted index occupies more than one disk block, more sorted accesses would be needed. This cost is approximately linear to the number of candidates generated. Such block-level access of inverted index [32] is much more efficient than verifying aggregates on a tuple-by-tuple basis.

3.3 Chunk-based Query Execution Algorithm with Buffer Reuse

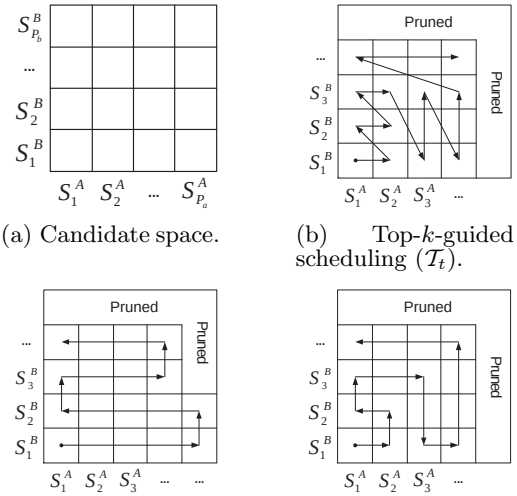
In QueryExec(), the order of guiding cells can dynamically change after each verification due to aggregate-bound updating. This method, however, has two limitations and we further optimize them in this section. Firstly, many disk accesses to the inverted index are wasted. Consider two consecutively generated candidates $\hat{c}'$, combined from guiding cells $g'_1, \ldots, g'_N$, and $\hat{c}''$, from $g''_1, \ldots, g''_N$. The main cost of verification for $\hat{c}'$ and $\hat{c}''$ lies in $2N$ random accesses for the inverted indices of $g'_1, \ldots, g'_N$ and $g''_1, \ldots, g''_N$. The actual number of random accesses for verifying these two candidates, however, could be reduced through inverted index reuse. For example, only $2N - N'$ random accesses are needed if for some $i_1, \ldots, i_{N'}$ we have $g'_{i_j} = g''_{i_j}$ $(1 \leq j \leq N' \leq N)$, because the inverted index of g'_{i_j} can be cached in the memory and reused by $\hat{c}''$. Unfortunately, the candidate scheduling method in QueryExec() (Line 3) is only concerned with the goal of generating promising candidates while ignoring another goal of seeking more opportunities for reusing in-memory inverted indices. Secondly, for the purpose of avoiding redundancy in candidate generation, QueryExec() needs to keep track of all the verified candidates (Line 5), which requires unbounded memory space. Thus maintaining such a flag array would employ a non-trivial amount of CPU and memory cost.

To address the limitations, we propose a *chunk-based query execution model*, which uses bounded memory buffer by partitioning the candidate space into chunks and scheduling the chunks in a way that facilitates both pruning and buffer reuse. As a result, inverted indices can be cached to benefit subsequent candidate verifications as much as possible. In addition, the scheduling algorithm would dynamically determine the order in which candidates are generated and waives the necessity of maintaining the flag array.

Sublist	$\overline{F}_{agg}(a_i)$
S_1^A	$a_6 : 150$
	$a_{12} : 115$
S_2^A	$a_8 : 90$
	$a_1 : 84$
	$a_{10} : 79$
...	...
$S_{P_a}^A$	$a_2 : 18$
	$a_7 : 18$
	$a_3 : 13$

Sublist	$\overline{F}_{agg}(b_i)$
S_1^B	$b_7 : 120$
	$b_3 : 95$
S_2^B	$b_4 : 85$
	$b_6 : 82$
...	...
$S_{P_b}^B$	$b_5 : 26$
	$b_9 : 22$

Figure 9: Sublists of A. Figure 10: Sublists of B.



(a) Candidate space. (b) Top-k-guided scheduling ($\mathcal{T}_t$). (c) Buffer-guided scheduling ($\mathcal{T}_b$). (d) Hybrid scheduling ($\mathcal{T}_h$).

Figure 11: Examples of chunking and scheduling.

3.3.1 Partitioning Candidate Space into Chunks

The candidate space is a N-dimensional space formed by the Cartesian product of the N sorted lists. Given a buffer size threshold θ_{CHUNK}, we adopt an equi-depth partition method and partition each sorted list into some **sublists** so that the total size of the inverted indices corresponding to each sublist must not exceed $\theta_{SUBLIST} = \frac{\theta_{CHUNK}}{N}$. This can be done at the initialization stage of the sorted lists. If the i-th $(1 \leq i \leq N)$ sorted list is partitioned into P_i sublists, then the candidate space would have $\prod_{i=1}^{N} P_i$ chunks.

Example 5. We make up a new example for candidate space partitioning. Figures 9 and 10 illustrate two imaginary sorted lists, partitioned into P_a and P_b sublists, respectively. The total inverted index size of all cells in each sublist (*e.g.*, a_8, a_1, and a_{10}) is no greater than $\theta_{SUBLIST}$. Figure 11(a) shows the corresponding candidate space with $P_a \cdot P_b$ chunks, where chunk $S_I^A \times S_J^B$ contains all candidates generated by the cells from S_I^A and S_J^B $(1 \leq I \leq P_a, 1 \leq J \leq P_b)$. For instance, $S_1^A \times S_1^B = \{(a_6, b_7), (a_6, b_3), (a_{12}, b_7), (a_{12}, b_3)\}$. ∎

One advantage of such chunking is that the candidates in the same chunk can share the same set of inverted indices (**intra-chunk buffer reuse**). Consider a chunk generated by N sublists, each having $|S|$ guiding cells. For QueryExec(), the worst-case random access cost of verifying

Procedure: ChunkQueryExec()	
Input: Top-k, $\mathcal{A}^{\mathcal{Q}}$, $C^{gd}(\mathcal{A}_1, SUM), \ldots, C^{gd}(\mathcal{A}_N, SUM)$, $C^{sp}(\mathcal{A}_1), \ldots, C^{sp}(\mathcal{A}_N)$, θ_{CHUNK}, Scheduling algorithm $\mathcal{T}$.	
Output: Top-k aggregate cells.	
Chunk-based candidate generation	
1	The same as in QueryExec(), Line 1;
2	The same as in QueryExec(), Line 2;
3	$\theta_{SUBLIST} = \theta_{CHUNK}/N$;
4	For each $i = 1..N$, partition the i-th sorted list into P_i sublists $S_1^{\mathcal{A}_i}, \ldots, S_{P_i}^{\mathcal{A}_i}$;
5	Initialize buffer $\mathcal{B} \leftarrow \phi$; $\vec{q} \leftarrow null$;
6	$\vec{p} \leftarrow \mathcal{T}.getNextChunkPos()$; Terminate if there is no available chunk left;
7	FOR $i \leftarrow 1$ TO N DO
8	IF $\vec{p}[i] \neq \vec{q}[i]$ THEN
9	FOR each cell g in $S_{\vec{q}[i]}^{\mathcal{A}_i}$ DO Clean the inverted index of g from $\mathcal{B}$;
10	FOR each cell g in $S_{\vec{p}[i]}^{\mathcal{A}_i}$ DO Fetch the inverted index of g into $\mathcal{B}$;
11	$\vec{q} \leftarrow \vec{p}$;
Chunk-based candidate verification	
12	FOR each candidate cell $\hat{c}$ in the current chunk DO
13	Verify $F_{agg}(\hat{c})$ using inverted indices in $\mathcal{B}$;
14	The same as in QueryExec(), Line 7;
15	The same as in QueryExec(), Line 8;
16	Prune all chunks $\mathbb{H}(\vec{p})$ where $\overline{F}_{\mathbb{H}(\vec{p})} \leq \theta$; Prune all guiding cells g where $\overline{F}_{agg}(g) \leq \theta$;
17	Go to 6;

Table 2: Chunk-based query execution for *SUM*.

all $|S|^N$ candidates in the chunk can be $\mathcal{O}(N|S|^N)$. In contrast, the cost can be reduced to $\mathcal{O}(N|S|)$ after chunking.

3.3.2 Chunk Scheduling

The job of a chunk scheduling algorithm, $\mathcal{T}$, is to sequentially pick a chunk from the candidate space at each iteration of candidate generation and verification. Assuming that such an algorithm $\mathcal{T}$ is given, we first discuss the chunk-based query execution model, followed by a discussion on $\mathcal{T}$. Table 2 shows the pseudocode for ChunkQueryExec(). Note that $\vec{p}$ and $\vec{q}$ are N-dimensional vectors recording the currently and previously scheduled chunk positions, respectively (Lines 5–6), and $\mathbb{H}(\vec{p})$ represents the corresponding chunk (Line 16, we explain $\overline{F}_{\mathbb{H}(\vec{p})}$ momentarily). For example, $\mathbb{H}(\vec{p})$, where $\vec{p} = [2, 1]$, refers to the chunk $S_2^A \times S_1^B$. For the i-th sublist, if $\vec{p}[i]$ is equal to $\vec{q}[i]$ (Line 8), then nothing needs to be done because the corresponding inverted indices have already been retrieved into the buffer (**inter-chunk buffer reuse**). The candidate generation of ChunkQueryExec() uses bounded buffer of size θ_{CHUNK}, having no overhead of bookkeeping any verified candidates regardless of what scheduling method is used. Although the bulky candidate verification may sacrifice the priority of some promising candidates to a minor degree, it is able to achieve significant efficiency gain, as shown earlier.

Given a fixed partitioning scheme, the cost of the chunk-based algorithm is determined by (i) the total number of chunks pruned; and (ii) the inter-chunk buffer reusage, both dependent on the scheduling algorithm $\mathcal{T}$. To consider (i), we first propose a *top-k-guided scheduling* method, $\mathcal{T}_t$, which tries to maximize the total number of chunks pruned. $\mathcal{T}_t$ maintains a priority queue for all $\prod_{i=1}^N P_i$ chunks, whose priority value, $\overline{F}_{\mathbb{H}(\vec{p})}$, is defined as follows.

Definition 5. Given a chunk position vector $\vec{p}$, we define the aggregate-bound of $\mathbb{H}(\vec{p})$ as

$$\overline{F}_{\mathbb{H}(\vec{p})} = \min_{i=1}^{N} \left\{ \max_{g \in S_{\vec{p}[i]}^i} \{\overline{F}_{agg}(g)\} \right\}. \quad \blacksquare$$

This aggregate-bound indicates the upper-bound aggregate score of all the candidates within chunk $\overline{F}_{\mathbb{H}(\vec{p})}$. Each time $\mathcal{T}_t.getNextChunkPos()$ (Line 6) is called, the chunk $\mathbb{H}(\vec{p})$ with the largest $\overline{F}_{\mathbb{H}(\vec{p})}$ would be returned. If more than one chunk share the same $\overline{F}_{\mathbb{H}(\vec{p})}$, $\mathcal{T}_t$ would return the one with the largest $\max\{\overline{F}_{agg}(g) | \ 1 \leq i \leq N, g \in S_{\vec{p}[i]}^i\}$. Notice that $\overline{F}_{\mathbb{H}(\vec{p})}$ may be dynamically updated after each chunk is verified. The drawback of $\mathcal{T}_t$ is that it does not consider inter-chunk buffer reuse, which may help save a considerable amount of cost. To this end, we propose another method called *buffer-guided scheduling*, $\mathcal{T}_b$, that aims at optimizing (ii). $\mathcal{T}_b$ starts at the chunk with the largest $\overline{F}_{\mathbb{H}(\vec{p})}$ and traverses the chunks in the axis order, greedily choosing the next chunk that shares as many sublists as possible. The following theorem shows that this greedy strategy is able to maximize the inter-chunk buffer reuse. Its proof is straightforward and omitted here.

Theorem 1. For the $\prod_{i=1}^N P_i$ chunks, a *buffer-guided schedule* minimizes the total disk accesses needed to fetch the inverted indices compared to any other schedules. $\blacksquare$

Example 6. Figures 11(b) and 11(c) illustrate two example schedules generated by $\mathcal{T}_t$ and $\mathcal{T}_b$ on a 2-d space, respectively. The $\mathcal{T}_t$-schedule starts at the lower-left chunk which contains the most promising candidates and subsequently chooses a remaining chunk with the largest priority value. As the top-k threshold θ becomes larger, the sublists as well as the corresponding chunks at the tail of the axes can be pruned. However, this schedule often cannot reuse any cached sublist. For example, no inverted index is shared by chunks $S_2^A \times S_1^B$ and $S_1^A \times S_2^B$. The $\mathcal{T}_b$-schedule, on the other hand, contiguously traverses chunks following S^A's axis horizontally and then S^B's axis vertically. A sublist is cached and reused at every step. The $\mathcal{T}_b$-schedule, nevertheless, unnecessarily visited chunks that should have been pruned, as illustrated. Although schedules may differ as to different database instances, it is clear that both $\mathcal{T}_t$ and $\mathcal{T}_b$ have drawbacks in principle. $\blacksquare$

To take into consideration both (i) and (ii), we develop a *hybrid scheduling* algorithm, $\mathcal{T}_h$, that bridges $\mathcal{T}_t$ and $\mathcal{T}_b$. The idea is that, based on $\mathcal{T}_t$'s priority queue, we further group together chunks with the same aggregate-bound and use $\mathcal{T}_b$ to schedule the chunks within a group. This idea follows from the fact that at any time of the query execution, there are at most $\sum_{i=1}^N P_i$ distinct aggregate-bounds for the $\prod_{i=1}^N P_i$ chunks, which can be justified by Definition 5. We call a group of chunks sharing the same aggregate-bound an equivalent group. At the beginning of the execution there are no more than $\sum_{i=1}^N P_i$ equivalent groups, each containing $\overline{\mu} = \prod_{i=1}^N P_i / \sum_{i=1}^N P_i$ chunks on average. $\overline{\mu}$ could be

F^Q_{agg}	Definition ($i = 1..n$)	$\mathcal{M}$	$\overline{F}_{agg} = \Gamma(\mathcal{M})$
SUM	$\sum_i s_i$	SUM	SUM
$COUNT$	n	$COUNT$	$COUNT$
AVG	$\overline{s} = \sum s_i/n$	MAX	MAX
MAX	$\max_i\{s_i\}$	MAX	MAX
MIN	$\min_i\{s_i\}$	MAX	MAX
VAR	$\sigma^2 = \sum_i (s_i - \overline{s})^2/n$	MAX,MIN	$(MAX\text{-}MIN)^2/4$
$STDDEV$	$\sigma = \sqrt{\sigma^2}$	MAX,MIN	$(MAX\text{-}MIN)/2$
MAD	$\sum_i \lvert s_i - \overline{s}\rvert/n$	MAX,MIN	$(MAX\text{-}MIN)/2$
$RANGE$	$\max_i\{s_i\} - \min_i\{s_i\}$	MAX,MIN	$MAX\text{-}MIN$

Table 3: Aggregate measures and their corresponding guiding measures and aggregate-bound.

large when P_i's and N are relatively large. Moreover, each equivalent group of chunks may not have distinctly different pruning power, but they are often spatially proximate to each other. Thus, $\mathcal{T}_h$ can be improved based on $\mathcal{T}_t$ as follows. Instead of returning a single chunk $\mathbb{H}(\vec{p})$ with the largest $\overline{F}_{\mathbb{H}(\vec{p})}$, we dynamically choose an equivalent group of chunks having the largest $\overline{F}_{\mathbb{H}(\vec{p})}$. Then we schedule the chunks in the group using the buffer-guided approach; that is, these chunks are retrieved and verified by following the axis order, where adjacent ones are greedily scheduled.

Example 7. To demonstrate $\mathcal{T}_h$, we draw a schedule in Figure 11(d). This schedule is still guided by $\overline{F}_{\mathbb{H}(\vec{p})}$ at high-level and chunks at the tail of the sublists can be pruned. At each step except for the first one, a sublist in the buffer can be reused. This amount of buffer reuse is comparable to that of the $\mathcal{T}_b$-schedule in Figure 11(c). ∎

For higher-dimensional candidate space ($3 \le N \le d$), the *getNextChunkPos*() operation cost of the scheduling algorithms is inexpensive because the total number of chunks is much smaller compared to the number of guiding cells. Also, each operation needs to consider only chunks in the proximity of the current chunk for the next step.

4. SUPPORTING GENERAL RANKING FUNCTIONS

In this section, we address extensions to other common aggregate and statistical measures, including AVG, MAX, VAR (*Variance*), $STDDEV$(*Standard Deviation*), $RANGE$, MAD(*Mean Absolute Deviation*), *etc.*, as listed in Table 3. We defer the discussion of supporting combination functions as well as more complex measures to Section 6.2.

To handle more general types of query measures, we extend ChunkQueryExec(), the chunk-based algorithm for SUM, to GeneralChunkQueryExec() as shown in Table 4. While the candidate generation and verification framework remains unchanged, their key difference lies in the computation of the aggregate-bound $\overline{F}_{agg}(g)$ for any guiding cell g. By bounding principle, $\overline{F}_{agg}(g)$ should represent the maximum possible aggregate value of any candidate $\hat{c}$ generated by g, *i.e.*, $\overline{F}_{agg}(g) \ge F^Q_{agg}(\hat{c})$ must always hold for the query measure F^Q_{agg}. Therefore, the aggregate-bound for SUM cannot work for other query measures. We address the problem of aggregate-bound computation for different measures in two aspects: (1) the initialization $\overline{F}_{agg}(g)$ for all guiding cells (Line 1); and (2) how to update $\overline{F}_{agg}(g)$ (Line 15), in the sections below.

Procedure: GeneralChunkQueryExec()	
Input: Top-k, F^Q_{agg}, $\mathcal{A}^{\mathcal{Q}}$, $C^{gd}(\mathcal{A}_1, \mathcal{M})$, ..., $C^{gd}(\mathcal{A}_N, \mathcal{M})$, $C^{sp}(\mathcal{A}_1), \ldots, C^{sp}(\mathcal{A}_N)$, θ_{CHUNK}, $\mathcal{T}$.	
Output: Top-k aggregate cells.	
1	[Sorted List Initialization] For each guiding cuboid $C^{gd}(\mathcal{A}_i, \mathcal{M})$ ($i = 1..N$), create a sorted list containing all guiding cells from that cuboid and for each guiding cell g, initialize $\overline{F}_{agg}(g)$ according to F^Q_{agg};
2–14	The same as in ChunkQueryExec()
15	[Aggregate-Bound Updating] For each $i = 1..N$, update $\overline{F}_{agg}(g_i)$ based on g_i's inverted index;
16–17	The same as in ChunkQueryExec()

Table 4: General chunk-based execution algorithm.

4.1 Initializing $\overline{F}_{agg}$

Table 3 lists the guiding measures and the initial aggregate-bound for different query measures. The initial aggregate-bound of the first five query measures, SUM, $COUNT$, AVG, MAX, and MIN, can be obtained from a single guiding measure, whereas the initial aggregate-bound of the remaining four query aggregate measures can be expressed as a function Γ of more than one guiding measure. To initialize the aggregate-bound of guiding cells for a particular query measure F^Q_{agg}, its corresponding guiding measures $\mathcal{M}$ should be already materialized. The initialization process is similar to ChunkQueryExec() in that N sorted lists are created, each containing all the guiding cells in the corresponding guiding cuboid. For each guiding cell, its materialized guiding measure values are retrieved, and its aggregate-bound is computed using Γ on the materialized measure values.

Example 8. Suppose $F^Q_{agg} = VAR$ and a guiding cuboid $C^{gd}(\mathcal{A}', \mathcal{M})$ is given. Assume that $MAX, MIN \in \mathcal{M}$. To initialize the sorted list of $\mathcal{A}'$, we scan the guiding cuboid and get $MAX(g)$ and $MIN(g)$ for each guiding cell g, followed by computing $\overline{F}_{agg}(g) \leftarrow \Gamma(MAX, MIN)(g) = (MAX(g) - MIN(g))^2/4$. Finally we sort the list descendingly according to $\overline{F}_{agg}(g)$. ∎

Notice that the same materialized guiding measures, $\mathcal{M}$, can be *shared* by different query measures. For example, materializing $\mathcal{M} = \{MAX, MIN\}$ would be able to provide aggregate-bound to multiple query measures like AVG, $STDDEV$, *etc.*. This is in contrast to traditional data cubes whose materialization often can only be tailored to a particu-

lar measure. Furthermore, the query measures and aggregate-bounds supported by this framework are not confined to Table 3. This integrated approach in fact gives users the freedom to customize $\mathcal{M}$ based on different needs. For example, one may materialize the "top-k AVG" [18] as a guiding measure to provide tighter aggregate-bound for AVG and more complex measures. It is also worth mentioning that the content of a supporting cuboid is independent of query measure; once materialized, the supporting cuboid can provide verification to an arbitrary ranking function. Below we give the proof of correctness to the aggregate-bounds.

Theorem 2. The aggregate-bound is correct for each measure F^Q_{agg} in Table 3.

Proof Sketch. The proof for SUM, $COUNT$, AVG, MAX, MIN, and $RANGE$ can be directly derived from the containment relationship between parent and children aggregates.

For $F^Q_{agg} = MAD$, we prove $F^Q_{agg}(s_1, \ldots, s_n) \leq (MAX - MIN)/2$. Let $s^+ = \sum_{s_i > \overline{s}}(s_i - \overline{s})$, $s^- = \sum_{s_i \leq \overline{s}}(\overline{s} - s_i)$, and $\lambda = |\{s_i | s_i > \overline{s}\}|$. Without loss of generality, assume $1 \leq \lambda < n$. We have $s^+ = s^-$, $s^+/\lambda \leq MAX - \overline{s}$, and $s^-/(n-\lambda) \leq \overline{s} - MIN$. Thus, $F^Q_{agg}(s_1, \ldots, s_n) = 2s^+/n \leq 2(MAX - \overline{s})(\overline{s} - MIN)/[(MAX - \overline{s}) + (\overline{s} - MIN)] \leq (MAX - MIN)/2$.

For $F^Q_{agg} = VAR$, we prove $F^Q_{agg}(s_1, \ldots, s_n) \leq (MAX - MIN)^2/4$. Without loss of generality, assume $s_i \in [0, 1]$ for $1 \leq i \leq n$. The inequality holds when $n = 1$. When $n \geq 2$, we have $\partial F^Q_{agg}/\partial s_i = 2s_i/n - 2\overline{s}/n$, meaning that F^Q_{agg} reaches local minimum when $s_i = \overline{s} \in [0, 1]$ and local maximum when $s_i = 0$ or 1 if s_j $(j \neq i)$ is fixed. Thus, $F^Q_{agg}(s_1, \ldots, s_n)$ reaches global maximum only if $\forall i \Rightarrow s_i = 0$ or 1. Next we can prove that F^Q_{agg} reaches global maximum when $\lfloor \frac{n}{2} \rfloor$ of s_i's are 0's and the remaining $\lceil \frac{n}{2} \rceil$ are 1's. Therefore, $F^Q_{agg} \leq \frac{1}{n} \cdot n(\frac{1}{2})^2 = 1/4$. $STDDEV$ can be proved similarly. ∎

4.2 Updating $\overline{F}_{agg}$

Updating the aggregate-bound is equally important as the initialization because the aggregate-bound is monotonically decreasing, providing an increasingly tighter bound to prune more guiding cells. Since aggregate-bound is derived from guiding measures using Γ, to update the aggregate-bound for a F^Q_{agg} would require to update its corresponding guiding measure values in the first place. Starting from the measures in Table 3, we can see that any F^Q_{agg} in the table can be derived from one or two of the four guiding measures, SUM, $COUNT$, MAX, and MIN. For SUM, as we have discussed, the updating procedure is $\overline{F}_{agg}(g) \leftarrow \overline{F}_{agg}(g) - F^Q_{agg}(\hat{c})$, where g a guiding cell and $\hat{c}$ is a candidate generated by g. This procedure applies to $COUNT$ as well. For MAX and MIN, the updating procedure is as follows. On the way to verify $\hat{c}$, g's inverted index must be fetched from the buffer (Line 13). We maintain the set of g's raw scores, denoted as S_g, during the fetching process. After the intersection of the inverted indices of all guiding cells, we can also know the set of $\hat{c}$'s raw scores, denoted as $S_{\hat{c}}$ $(\subseteq S_g)$. Thus, the updated $MAX(g)$ would be $\max(S_g - S_{\hat{c}})$, whereas the updated $MIN(g)$ would be $\min(S_g - S_{\hat{c}})$. Finally, after all the updated guiding measure values are obtained, Γ can be applied to calculate the updated aggregate-bound $\overline{F}_{agg}(g)$, which must be monotonically decreasing.

Synthetic Dataset Parameter	Notation	Default
Number of tuples	T	1M
Number of attributes	d	10
Number of score attributes	-	1
Average cardinality	C	10000
Skewness of score distribution	α	0.5

Table 5: Synthetic dataset parameters.

Query Workload Parameter	Notation	Default
Aggregate function	-	SUM
Workload size	-	5
Top-k	k	10
Number of guiding cuboids	N	2

Table 6: Query workload parameters.

Example 9. Suppose $F^Q_{agg} = VAR$ and g's inverted index is $\{(t1, 100), (t3, 80), (t4, 120), (t7, 40), (t9, 50)\}$ and its candidate cell $\hat{c}$ has inverted index $\{(t4, 120), (t7, 40)\}$ after verification. Then $\max(S_g - S_{\hat{c}}) = 100$ and $\min(S_g - S_{\hat{c}}) = 50$, which means $\overline{F}_{agg}(g)$ should be updated to $\Gamma(100, 50) = (100 - 50)^2/4 = 625$. ∎

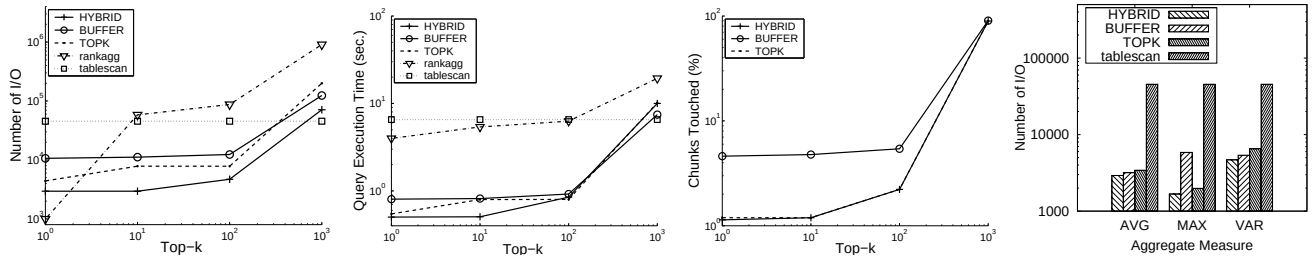
As long as guiding measures can be derived from the set $S_g - S_{\hat{c}}$, the aggregate-bound can always be updated under the framework. It is straightforward to prove that the general updating process guarantees that Lemmas 1 and 2 still hold for each of the query measures in Table 3, so we omit the proof here.

5. EXPERIMENTAL EVALUATION

We compare five different query execution algorithms: the *tablescan* approach that sequentially scans the data file and computes top-k; the *rankagg* approach, which is the state-of-the-art approach for ranking aggregate query processing studied in [23]; and the chunk-based query execution approaches $HYBRID$, $BUFFER$, and $TOPK$, which use the hybrid, buffer-guided, and top-k-guided scheduling methods, respectively. We use both synthetic data sets and the standard decision support benchmark TPC-H [2] for evaluation. The platform for the experiments is Pentium CPU 3GHz with 1G RAM. Source codes are executed using JRE1.6.0 on Windows XP.

5.1 Experiments on Synthetic Data Sets

We generate synthetic data sets in single flat files with the default parameters shown in Table 5 and query workloads with their default setting in Table 6. α denotes the Zipf's exponent that characterizes the distribution of the raw scores, where $\alpha = 1$ corresponds to a skewed score distribution whereas $\alpha = 0$ corresponds to a uniform score distribution. For *rankagg*, an in-memory multi-key hash table is built on the group-by attributes for each query and the *GroupOnly* execution plan is used. For the construction of *ARCube*, we use the baseline materialization approach that materializes cuboids for all single attributes on the synthetic data sets, which is approximately twice as large as the original data set. Moreover, we materialize four guiding measures, SUM, $COUNT$, MAX, and MIN for the guiding cells (*i.e.*, $\mathcal{M} = \{SUM, COUNT, MAX, MIN\}$) in order to provide aggregate-bound to different ranking query measures. We set the disk block size to be $1KB$ and the chunk size to be $1MB$.

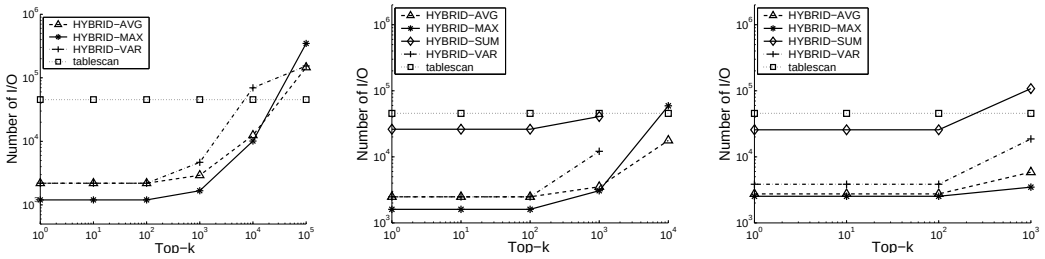


(a) Disk access comparison of query algorithms on SUM.

(b) Query execution time comparison on SUM.

(c) Chunk retrieval rate of chunk-based algorithms.

(d) Comparison on other query measures.



(e) Disk access vs. top-k on other query measures.

(f) 3-way combination on different query measures.

(g) 4-way combination on different query measures.

Figure 12: Performance vs. query parameters on synthetic data sets.

5.1.1 Performance w.r.t. Top-k

We first examine the performance of *tablescan*, *rankagg*, *HYBRID*, *BUFFER*, and *TOPK*on the *SUM* measure. We vary the parameter k and plot the disk access and query execution time of the five methods in Figures 12(a) and 12(b), respectively. For $k = 1$, 10, and 100, the chunk-based query algorithms consistently outperform *tablescan* in terms of both disk access and execution time. The disk access of *HYBRID* at $k = 1$ and 10 is 1/15 of that of *tablescan*. Also, the chunk-based algorithms consistently outperform *rankagg* by up to an order of magnitude except for that *rankagg*'s disk access at $k = 1$ is fewer; however, *HYBRID* is still 8 times faster than *rankagg* at $k = 1$, due to *rankagg*'s CPU overhead of scanning through the in-memory multi-key index and initializing a priority queue of many groups.

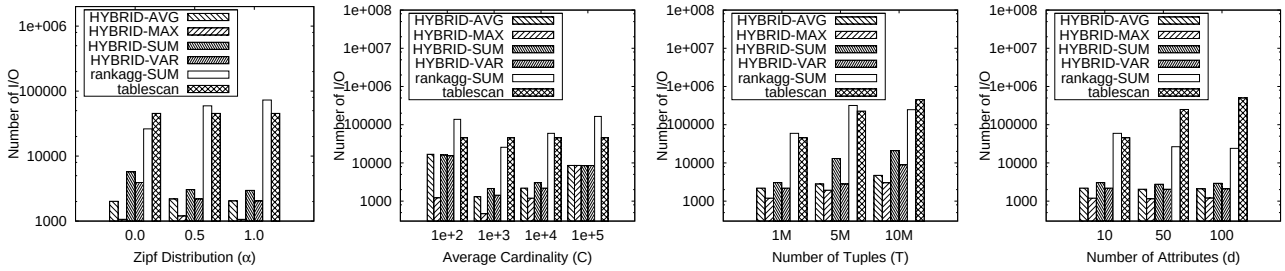
Among the three chunk-based algorithms, *HYBRID* consumes less I/Os and is faster than the other two. *BUFFER* needs more disk accesses than *TOPK* in general since its traversal path does not give particular preference to promising candidates and therefore may visit chunks that could have been pruned. This can be clearly seen from Figure 12(c), where we plot the percentage of chunks touched among all chunks in the candidate space w.r.t. the parameter k for the three chunk-based algorithms. The same query workloads are used for plotting this figure as for Figures 12(a) and 12(b). The number of chunks touched by *HYBRID* and *TOPK* are both lower than *BUFFER* and very close to each other because they give promising chunks of candidates higher priority. This confirms our intuition that *HYBRID*'s schedule can take advantage of both chunk pruning and buffer reuse.

On the other hand, when $k = 10^3$, *tablescan* is faster than all the other four ranking query algorithms, since the pruning power of the top-k threshold is no longer large. At this time, *BUFFER* outperforms *TOPK* or even *HYBRID* because it maximizes buffer reuse when most chunks cannot be pruned anyway. We use *disk access* (*i.e.*, total number of I/Os) as our major performance metric hereafter.

5.1.2 Performance w.r.t. Query Measures

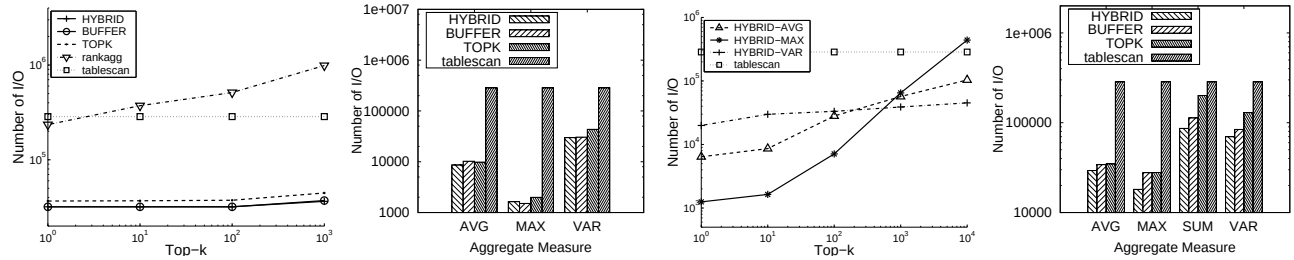
Besides SUM, we also evaluate the chunk-based algorithms on other aggregate measures (F^Q_{agg}). We select three representative aggregate measures, AVG, MAX, and VAR for the comparison. Notice that the commonly used $STDDEV$ (*Standard Deviation*) measure should generate the same performance results as VAR because the former measure can be derived by taking the square root of the latter. The three measures are representative in that MAX's aggregate-bound (as shown in Table 3) is itself; AVG and VAR's aggregate-bounds are linear and quadratic function of MAX and (MAX-MIN), respectively. We do not evaluate *rankagg* in this test case because it does not explicitly support non-monotone aggregate functions like VAR, *i.e.*, adding a tuple into a cell does not necessarily increase its aggregate.

The disk access cost of *HYBRID*, *BUFFER*, *TOPK*, and *tablescan* is depicted in Figure 12(d), from which we can see that *HYBRID* is better than the other methods in the figure: its cost is 1/15 of *tablescan*'s for AVG, 1/27 for MAX, and 1/9 for VAR. This matches our intuition because MAX itself can provide the "tightest" aggregate-bound to its candidates: a candidate's MAX value can be directly computed from its guiding cells' MAX values, enabling effective pruning. Therefore, for MAX, we set the default chunk size of the chunk-based algorithms to $0.1M$ to avoid loading unnecessary guiding cells. The tightness of VAR' aggregate-bound, which is a quadratic function of *(MAX-MIN)*, is less than that of MAX, as expected. AVG, however, has a tighter aggregate-bound than VAR because it is a linear function of *(MAX-MIN)*. For AVG and VAR, *BUFFER* has slightly less disk access cost than *TOPK* does, justifying the effectiveness of buffer reuse. Nevertheless, for MAX, *BUFFER*'s cost is approximately 3 times as much as *TOPK*'s, showing



(a) Comparison of performance w.r.t. data skewness. (b) Comparison of performance w.r.t. average cardinality. (c) Comparison of performance w.r.t. number of tuples. (d) Comparison of performance w.r.t. number of attributes.

Figure 13: Performance vs. synthetic data set parameters.



(a) Comparison on *SUM*. (b) Comparison on other query measures. (c) Disk access vs. top-k on other query measures. (d) 3-way combination on different query measures.

Figure 14: Experiments on the TPC-H benchmark.

that only considering buffer reuse may lead to the verification of unnecessary chunks. In Figure 12(e), we plot the disk access of *HYBRID* w.r.t. top-k on *AVG*, *MAX*, and *VAR* and compare it to *tablescan*. For *MAX* and *AVG*, *HYBRID* outperforms *tablescan* for $k \leq 10^4$, while for *VAR*, its cost is better than *tablescan* for $k \leq 10^3$. This shows that the aggregate-bounds of *AVG* and *MAX* indeed are better than that of *VAR*. At $k = 10^6$, where the top-k threshold becomes very small and barely has pruning power, *HYBRID*'s cost on *AVG* and *VAR* agree with each other because they tend to traverse the whole candidate space and therefore have similar cost. The cost on *MAX* is larger because the default chunk size has been set to $0.1M$ as discussed and therefore more chunks need to be visited. We observe that for a particular measure, *HYBRID*'s cost for $k = 1$, 10, and 100 is very close to each other. This can be explained by the bulky nature of the algorithm, which may be able to find the top-k results by visiting only one or a few chunks.

5.1.3 Performance w.r.t. Multi-Way Combination

The evaluation on *AVG*, *MAX*, and *VAR*, together with *SUM*, has demonstrated the effectiveness of the guiding measures and the aggregate-bound on 2-way combination, *i.e.*, candidates are generated by combining guiding cells from 2 cuboids. We now turn to the analysis of the *HYBRID* algorithm on multi-way combination (*i.e.*, $N \geq 3$). Figure 12(f) shows *HYBRID*'s disk access cost on the four query measures in comparison to *tablescan*. For *AVG*, *MAX*, and *VAR*, the ratio of *HYBRID*'s cost on 3-way combination ($N = 3$) to the cost of *tablescan* is no more than 1/18 at $k \leq 10^2$, which is slightly worse than the worst ratio in Figure 12(e), 1/20. On the other hand, when k becomes larger (*i.e.*, $k \geq 10^3$), the top-k threshold would become smaller accordingly and lead to a cubic increase of number of chunks that have to be verified. For the *SUM* measure, *HYBRID* costs almost a half of *tablescan*'s disk access. In fact, *SUM*'s aggregate-bound becomes looser for multi-way combination because the aggregate-bound of a guiding cell could be aggregated over many candidate cells and thus is harder to be bounded by the top-k threshold. Figure 12(g) further plots the disk access of *HYBRID* w.r.t. top-k for 4-way combination, which shows very similar trend of results as in Figure 12(f). The disk access cost ratio of *HYBRID* against *tablescan* is now at most 1/11 for *AVG*, *MAX*, and *VAR* at $k \leq 10^3$. For *SUM*, *HYBRID*'s cost at $k \leq 10^2$ is similar to the scenario in 3-way combination but exceeds *tablescan*'s cost at $k = 10^3$.

Figures 12(e) through 12(g) have characterized the chunk-based algorithm on different query measures and different numbers of combination dimensions. For $k \leq 10^3$, if the top-k threshold is reasonably large, many guiding cells (sublists) can be pruned and thus the total number of chunks to be verified is not very sensitive to N. If, on the contrary, the top-k threshold is small (because k is large), then the total number of chunks to be verified would grow exponentially w.r.t. N.

5.1.4 Performance w.r.t. Data Set Parameters

We compare the performance of *HYBRID*, *rankagg*, and *tablescan* by twisting the data set parameters. We first vary the Zipf distribution exponent, α (Figure 13(a)), which determines the skewness of raw scores. When α is large, different cells are likely to have skewed aggregate scores and, conversely, when α approaches 0, different cells are likely to have more uniform aggregate scores. We can see that *HYBRID*'s cost is lower than *rankagg* and *tablescan*. Furthermore, its

cost on all four query measures is monotonically decreasing w.r.t. α, showing that *HYBRID* favors more skewed score distributions. This is because when the score distribution is skewed, it becomes easier for the top-k threshold to prune more guiding cells at the tail of the sorted lists. In contrast, for the *SUM* measure, *rankagg* shows no preference to the skewed scores, because when the raw scores are more skewed, the upper bound of a cell (*i.e.*, the number of unseen raw tuples of that cell multiplied by the maximum raw score in the data set) could correspondingly become larger and harder to be pruned. For a fixed α, *HYBRID* performs consistently on the four aggregate measures; that is *MAX* has the lowest cost while *SUM* has the highest cost, which matches the previously presented results.

Next, we vary C, the average cardinality of the attributes and show the performance results in Figure 13(b). *HYBRID* works best at $C = 10^3$ and 10^4. This can be explained by the inverted index access method for candidate verification. The inverted index of a guiding cell is stored in, say, L ($L \geq 1$), consecutive disk blocks, and to retrieve the inverted index would need exactly 1 random access plus $L - 1$ sorted accesses. If the inverted index is too short (*i.e.*, the length of it is much smaller than the block size), then the retrieved disk page would be largely "wasted". On the other hand, if it is too long, the cost for verifying a single candidate will be very large. Therefore, the cost of *HYBRID* is higher at $C = 10^2$, where the cost-per-candidate verification is large, and 10^5, where the disk page utilization is low on average.

Figure 13(c) illustrates another set of test cases on different number of raw tuples (T). As expected, *HYBRID*'s cost increases w.r.t. T because the expected length of inverted indices is linear to T and the disk access cost for candidate verification is positively correlated with the length. For $T = 10M$, the disk access cost of *HYBRID* is 1/11 of *rankagg*'s cost on *SUM* and 1/150 of *tablescan*'s cost on *MAX*. Finally, we evaluate the algorithms w.r.t. the number of attributes (d). As depicted in Figure 13(d), *tablescan*'s cost is linear to the number of attributes, while both *HYBRID* and *rankagg* are not affected by parameter d. Actually, the "vertical format" (*i.e.*, inverted index format) of *ARCube* makes the chunk-based algorithms insensitive to d because candidates are generated using N guiding cuboids, where N is not determined by d. *rankagg* is also insensitive to d because there is a multi-key index on the query attributes and the disk cost to probe any raw tuple through the index is always a single I/O.

5.2 Experiments on the TPC-H Benchmark

For the TPC-H benchmark, we use the *dbgen* module to generate a database and then extract the largest relation *lineitem.tbl* which contains 2M tuples and 16 attributes stored in a flat file. We use the attribute *extendedprice* as the raw score column and ignore the last attribute *comment* that contains strings of variable length. Out of the remaining 14 attributes, 6 attributes have cardinality below 10, 2 attributes are between 11–50, 3 attributes are between 2400–2600, and the rest are above 10000. We materialize all guiding cuboids with less than $\theta_{CUBOID} = 200K$ (*i.e.*, 10% of the number of raw tuples) cells and their corresponding supporting cuboids; the materialization size is approximately 10 times of the data size. For query plan selection, we randomly choose one from all valid plans. We first plot the disk access w.r.t. top-k for all the five query execution algorithms for the *SUM* measure (Figure 14(a)). The chunk-based algorithms, *HYBRID*, *BUFFER*, and *TOPK*, consistently outperform *rankagg* and *tablescan*. The cost ratio of *HYBRID* to *tablescan* is no more than 1/9 at $k = 1, 10, 10^2$ and 1/7.8 at $k = 10^3$. Figure 14(b) shows the result of the chunk-based query algorithms on other query measures. The cost ratios of *HYBRID* to *tablescan* are now 1/33, 1/174, and 1/10, for *AVG*, *MAX*, and *VAR*, respectively. The difference of cost among the *HYBRID*, *BUFFER*, and *TOPK* is not large in this test case because only a few chunks have been verified. We conduct another test for *HYBRID* on *AVG*, *MAX*, and *VAR* (Figure 14(c)), which has similar results to Figure 12(e). *HYBRID*'s performance on *MAX* deteriorates faster than on *AVG* and *VAR* because a smaller default chunk size is used. We evaluate 3-way combination in the last test case. The cost ratios on *AVG*, *MAX*, *SUM*, and *VAR* are 1/10, 1/16, 1/3, and 1/4. This indicates that the tightness of the aggregate-bound for the four measures can be ranked descendingly in the following order: *MAX*, *AVG*, *VAR*, and *SUM*, which agrees with our previous results on the synthetic data sets.

6. DISCUSSION

6.1 Related Work

Recently the problem of ranking (or top-k) query processing attracts and holds the attention of increasingly more researchers. It has been studied from the perspective of RDBMS optimization [7, 21, 24, 12, 4, 32], middleware systems [14], and many other applications [6, 9, 3, 8, 20, 27]. A number of algorithms and techniques have been proposed to efficiently return a result set with a limited cardinality for easy digestion. In the field of data warehousing and OLAP, the data cube model [17, 10] has been extensively studied over a decade, playing the critical role in facilitating multidimensional aggregation and exploration. Data cubing methods rely on full materialization [34], partial materialization [19, 29, 26], or other forms of data compression, summarization, and optimization [11, 30, 22] to provide efficient answers to OLAP operations. The traditional techniques for ranking analysis, however, are often tailored to ranking functions on individual tuples, where ranking is on base facts of relational tables instead of on multidimensional aggregations. Therefore, they cannot handle ranking aggregate queries efficiently. Many techniques for data warehousing and OLAP are able to facilitate multidimensional data analysis through the data cube model, but, on the other hand, are unaware of the ordering of aggregate values and cannot yield satisfactory performance to ranking aggregate queries.

The closest known methods to our study are based on either *no materialization* (*rankagg* [23]) or *full materialization* (the *Sorted Partial Prefix Sum Cube* [25]) approach. The *rankagg* framework is proposed to support ad-hoc ranking aggregate queries. It makes the traditional query optimizer rank-aware and enforces an ordering on the physical access of database tuples. It also estimates the maximum possible aggregate score for the groups whose aggregate score is not completely verified yet, and then prunes the groups that have a score upper bound no more than the top-k scores already achieved. In [25], a new cube structure called *Sorted Partial Prefix Sum Cube* is precomputed to answer ranking aggregate queries on a given time interval. These two

methods represent two major types of approaches and have limitations in certain scenarios; while on-the-fly aggregation may burden query processing, computing a full data cube could be challenging when the dimensionality becomes high.

Another relevant problem is the iceberg cube computation. The iceberg cube is a practical model, where only cells with aggregate above a certain threshold are saved in the cube, while the other insignificant ones are discarded. In [5, 33], efficient methods are proposed to compute iceberg cubes given a threshold. [15] addresses the iceberg query optimization problem in the absence of precomputation. [18] further discusses an algorithm to support complex measures such as *AVG* for iceberg cubes. Note that iceberg cubes cannot be applied to ranking aggregates effectively, because the iceberg threshold needs to be predetermined, whereas the top-k results of a ranking query may or may not be above that threshold and have to be computed on the fly. Finally, the problem of organizing and accessing multidimensional data using chunks is not new. In [28, 34, 16, 13], various chunking and caching methods are proposed and studied. Nevertheless, none of them considers the aggregate ordering and thus cannot deal with ranking queries efficiently.

6.2 Extensions

We discuss three extensions to the ranking functions. First, if R has multiple score columns $S_1, \ldots, S_\xi$ ($\xi \geq 2$), and the ranking function consists of not only the aggregate function F^Q_{agg}, but also a combination function $F^Q_{comb} = f(s_1, \ldots, s_\xi)$ ($s_i \in dom(S_i)$) that aggregates over different score columns (*e.g.*, weighted sum). Then the ranking function can be written as $F^Q = F^Q_{agg}(F^Q_{comb})$ (*i.e.*, first combine then aggregate) or $F^Q_{comb}(F^Q_{agg})$. In such cases, we need to extend the guiding cuboids to materialize other columns so that $\mathcal{M}$ is materialized for each S_i. The computation of the aggregate-bound now becomes a function of Γ and f. For example, if $F^Q_{agg} = SUM$ and F^Q_{comb} is weighted sum, then the aggregate-bound for a guiding cell g would become the weighted sum of the materialized $SUM(\mathrm{g})$ of all score columns. If F^Q_{comb} is a monotone combination function and $F^Q = F^Q_{comb}(F^Q_{agg})$ (*i.e.*, first aggregate then combine), then our framework can be integrated with the TA-style algorithms [14]. Second, for complex ranking functions (*e.g.*, $F^Q_{agg} = (SUM - 100)^2$), the aggregate-bound must be computed properly according to the bounding principle, which requires that $\overline{F}_{agg}(g) \geq F^Q_{agg}(\hat{c})$ always holds. Third, for databases with concept hierarchies on which queries are formulated, this framework can be extended based on the interdependent relationship between guiding cells and candidate cells. Specifically, given N high-level guiding cells, all the candidate cells generated by their children cells in the concept hierarchy may naturally form one or more chunks. For example, guiding cells "United States" and "2008" at (country, year)-level can be combined to generate all possible (state, month)-level cells as candidates using a concept hierarchy.

The *ARCube* materialization and the query plan selection are two orthogonal problems to the query execution, which may be further optimized. To begin with, guiding cuboids with larger skewness and correlation with children cuboids should be preferably materialized. To reduce materialization size and query cost, inverted indices can be compressed using existing techniques [31] while closely related supporting cuboids (*e.g.*, $C^{sp}(A)$ and $C^{sp}(AB)$) may share the same set of inverted indices. Moreover, in Section 2, we assume that the group-by of a guiding cuboid and that of its corresponding supporting cuboid should match. In principle, however, these group-by's may not be the same as long as candidate cells can be appropriately verified. In that case, more flexible partition scheme and scheduling method should be developed to handle the group-by mismatch.

Different query plans may lead to different query cost due to the pruning power of different guiding cells. For instance, plan $C^{gd}(A, \mathcal{M})$ and $C^{gd}(BC, \mathcal{M})$ could be better than $C^{gd}(A, \mathcal{M})$, $C^{gd}(B, \mathcal{M})$, and $C^{gd}(C, \mathcal{M})$ because guiding cuboid $C^{gd}(BC, \mathcal{M})$ may provide tighter aggregate-bounds. Therefore, we will investigate in finding an effective way to select the best plan for query processing.

Finally, we may further speed up query processing by making the chunking algorithm adaptive to the cardinality and distribution of the sorted lists in order to determine an optimal chunk size and maximize the effectiveness of pruning. For example, skewed sorted lists with more guiding cells should be segmented into more sublists, which could result in larger pruned space. We plan to explore these issues in-depth in our follow-up research.

7. CONCLUSIONS

We have proposed a novel cube structure, *ARCube*, for supporting efficient ranking aggregate query processing. The *ARCube* consists of two types of materialized cuboids: guiding cuboid and supporting cuboid. A query execution framework has been developed based on the *ARCube*, where the guiding cuboids can guide ranking query processing by providing high-level data statistics and generating promising candidates, whereas the supporting cuboids provide efficient aggregate verification. We have also proposed a chunk-based optimization method to utilize the memory buffer. The performance evaluation verified that this unified framework is more efficient than the existing techniques in supporting various types of aggregate measures.

Acknowledgements. We would like to thank Cuiping Li for her constructive suggestions and Chengkai Li for his help with the experiments. Also thanks to the anonymous reviewers for their insightful comments.

8. REPEATABILITY ASSESSMENT RESULT

Figures 12(c)–(g), 13, and 14 have been verified by the SIGMOD repeatability committee.

9. REFERENCES

[1] DBLP. http://www.informatik.uni-trier.de/~ley/db/.
[2] TPC-H. http://www.tpc.org/tpch/.
[3] R. Agrawal, R. Rantzau, and E. Terzi. Context-sensitive ranking. In *SIGMOD Conference*, pages 383–394, 2006.
[4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.
[5] K. S. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD Conference*, pages 359–370, 1999.
[6] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–380, 2002.

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

[8] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects based on relationships. In *SIGMOD Conference*, pages 371–382, 2006.

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

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997.

[11] J. Claußen, A. Kemper, D. Kossmann, and C. Wiesner. Exploiting early sorting and early partitioning for decision support query processing. *VLDB J.*, 9(3):190–213, 2000.

[12] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.

[13] P. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *SIGMOD Conference*, pages 259–270, 1998.

[14] R. Fagin, A. Lotem, 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, 1998.

[16] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[17] 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. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[18] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD Conference*, pages 1–12, 2001.

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

[20] M. Hua, J. Pei, A. W.-C. Fu, X. Lin, and H. fung Leung. Efficiently answering top-k typicality queries on large databases. In *VLDB*, pages 890–901, 2007.

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

[22] L. V. S. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. In *VLDB*, pages 778–789, 2002.

[23] C. Li, K. C.-C. Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD Conference*, pages 61–72, 2006.

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

[25] H.-G. Li, H. Yu, D. Agrawal, and A. E. Abbadi. Progressive ranking of range aggregates. In *DaWaK*, pages 179–189, 2005.

[26] X. Li, J. Han, and H. Gonzalez. High-dimensional olap: A minimal cubing approach. In *VLDB*, pages 528–539, 2004.

[27] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, pages 115–126, 2007.

[28] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *ICDE*, pages 328–336, 1994.

[29] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, pages 488–499, 1998.

[30] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the petacube. In *SIGMOD Conference*, pages 464–475, 2002.

[31] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, May 1999.

[32] D. Xin, J. Han, H. Cheng, and X. Li. Answering top-k queries with multi-dimensional selections: The ranking cube approach. In *VLDB*, pages 463–475, 2006.

[33] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *VLDB*, pages 476–487, 2003.

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