Region-based Online Promotion Analysis*

Tianyi Wu† Yizhou Sun† Cuiping Li‡ Jiawei Han†
† University of Illinois at Urbana-Champaign
‡ Remin University of China
{twu5,sun22,hanj}@illinois.edu licuiping@ruc.edu.cn

ABSTRACT

This paper addresses a fundamental and challenging problem with broad applications: efficient processing of *region-based promotion queries*, *i.e.*, to discover the top-k most interesting *regions* for effective promotion of an object (*e.g.*, a product or a person) given by user, where a region is defined over *continuous ranged* dimensions. In our problem context, the object can be promoted in a region when it is top-ranked in it. Such type of promotion queries involves an exponentially large search space and expensive aggregation operations. For efficient query processing, we study a fresh, principled framework called *region-based promotion cube (RepCube)*. Grounded on a solid cost analysis, we first develop a partial materialization strategy to yield the provably maximum online pruning power given a storage budget. Then, cell relaxation is performed to further reduce the storage space while ensuring the effectiveness of pruning using a given bound. Extensive experiments conducted on large data sets show that our proposed method is highly practical, and its efficiency is *one to two orders of magnitude* higher than baseline solutions.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data mining*

General Terms

Algorithms, Performance

Keywords

Promotion analysis, ranked (top-k) query, region

1. INTRODUCTION

This paper examines a new class of online analytical processing (OLAP) and data mining queries, called *top-k region-based promotion query* (REPQUERY), which is a novel yet practically interesting type of queries for a broad range of decision support applications. The goal of REPQUERY can be intuitively stated as follows: given an object of interest, such as a product or a person, we would like to discover the *top-k promotion regions* to promote the given object. Here a *region* is defined as a continuous interval or block over one or a few OLAP range dimensions, and a *promotion region* intuitively refers to a region where the given object is highly ranked among all other objects.

*Example 1: To promote a hybrid car model H in a car sales database, a data analyst may discover that the top promotion region is {*Year = 2008∼2009; Price = \$15K∼\$20K*}, in which H is ranked as the 3rd bestselling model. Another example of such a top promotion region could be like {*CustomerIncome = \$20K∼\$30K*}, in which H is ranked the top-5. In contrast to the global region where H has a much lower rank, these regions offer more concrete and insightful information for the analyst to conduct market targeting and product positioning.*

Example 2: In the DBLP database[1]*, one may be interested in finding the best promotion region for a given author. While describing her as the 300-th most prolific author could be less interesting, using* REPQUERY*, however, one may find her to be the most prominent author in {*Year=*2000∼2010} in the database field.*

These examples illustrate the related application scenarios, where REPQUERY discovers promotion regions by drilling down to different parts of the data and surfacing the regions where the given object is highly ranked (note that most objects are not highly ranked in the global region). In business intelligence applications like marketing, REPQUERY is able to assist data analysts to quickly locate and understand which regions are the most likely to promote some specified product or business object among a myriad of candidate regions, and then they can leverage such promotion regions along with the highly ranked results to serve decision making purposes. In particular, such region-based ranked results can help, for example, (1) analyze the best market segments (*e.g.*, a particular customer space or a geographical area) for resource allocation and promotion; (2) advertise and enhance brand image (*e.g.*, a bestselling car model belonging to a certain price range or time frame); and, (3) discover and summarize interesting object features in not only categorical but also numerical feature spaces (*e.g.*, a highest-rated apartment rental business serving more than ten locations).

The most relevant work to this study is [15], where the multidimensional promotion analysis problem is originally proposed. Its goal can be briefly stated as finding subspaces (*i.e.*, multidimen-

*The work was supported in part by NASA grant NNX08AC35A, and the U.S. National Science Foundation grants IIS-08-42769 and IIS-09-05215

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

[1] http://dblp.uni-trier.de

sional selection conditions) where a given object is highly ranked. In comparison to the multidimensional promotion analysis problem, REPQUERY introduces three major new challenges.

First, using simple object ranking to measure top-k promotion regions could be insufficient in many cases because different regions may not be equally interesting; specifically, they may have (1) dramatically different sizes, and (2) containment relationships or overlaps that may cause redundancy in top-k results. To this end, the semantics of the model must incorporate *ad-hoc rank-independent weights* for regions, such that users can impose weights on regions based on at query time. Also, the *redundancy-aware* semantics should be supported such that the top-k results are discriminative, *i.e.*, no pair of top-k regions returned to user is very similar.

Second, because REPQUERY need to handle *continuous, ranged* dimensions whereas the previous problem deals with *categorical* dimensions, the search space of REPQUERY would be *significantly larger*. For example, a *Year* dimension with 50 distinct values may generate only 50 subspaces but $50 \times (50+1)/2 = 1275$ one-dimensional regions. We can see that if there are d range dimensions, each having cardinality N, the total number of regions would be $(\frac{N(N+1)}{2})^d$, quadratic of the number of subspaces. The huge search space would dwarf the cost saving of any online pruning method, making online optimization techniques simply infeasible on even moderately large data set.

Third, for REPQUERY, we have to tackle the *non-monotonicity* property of the aggregate measure when computing object rankings. Indeed, the previous work assumes that the aggregate measure for ranking be monotone such as *SUM* for the purpose of shared computation. The aggregate measures supported in this work can, nevertheless, be arbitrarily complex, ad-hoc measures defined by users.

Despite that the need for REPQUERY is commonplace, no existing work in the literature attempts to address all these challenges. What happens to users of a conventional database system is that they would need to go through a trial-and-error process to manually search for interesting promotion regions, meaning that they have to rely heavily on prior knowledge. The results obtained in this way could be rather incomplete or even misleading. On the other hand, among the numerous top-k query processing techniques, *none* can be applied toward solving REPQUERY, because they require that a region and the number of objects returned be specified as parameters, which are unknown a priori in our context.

Obviously, a naive implementation that exhaustively aggregates each region and computes the object's rank can be intolerable to users performing explorative analysis because of the exponential number of regions. At the other end of the spectrum, a full materialization approach would also be extremely costly even on a data set with moderate size. Thus, to efficiently answer top-k region-based promotion queries, in this paper we propose a novel, principled framework called the *Region-based Promotion Cube (RepCube)*, grounded on a *partial materialization strategy with solid theoretical analysis*.

A key ingredient of the framework is a model of the materialized cube cell's *pruning power*, which lays the foundation for an overall *cost model* that computes the pruning power of any cube structure. In the RepCube, the cell structure is similar to quantiles for estimating probability distributions; here we exploit each cell's capability in upper- and lower-bounding object ranks and hence pruning uninteresting regions at an early stage. However, unlike any previous work, our key observation is that a *uniform cell structure bookkeeping scores from evenly spaced rank positions does not generate satisfactory pruning power*. This is due to a unique property of our problem: an object is likely to be highly ranked in its top-k promotion regions. Thus, we present a cost model and its solution to generate an *optimized* cell structure adaptive to query distribution. These optimized cells are able to yield a *provably optimal* expected query execution cost.

A	B	O (Object)	M (Measure)
a_1	b_1	t_1	0.7
a_1	b_2	t_2	0.8
a_1	b_2	t_3	0.8
a_1	b_2	t_1	0.2
a_1	b_3	t_4	1.2
a_2	b_1	t_3	0.9
a_2	b_2	t_1	0.3
a_2	b_4	t_2	1.6
a_2	b_5	t_1	1.2

Table 1: An example fact table.

Another idea we explore is to condense regions sharing similar aggregate score distributions. For example, the sales of products might be similar for regions $\{Year = 2007 \sim 2009\}$ and $\{Year = 2008 \sim 2010\}$ after proper normalization. Thus, we select a few *relaxed cells* to represent score ranges instead of exact scores, and use them to summarize sets of original cube cells. This would lead to a further space saving. The effectiveness of a relaxed cell's pruning power can be controlled by a user-specified parameter ϵ. In summary, our contributions are the following:

- (Section 2) Present the class of *top-k region-based promotion queries* and the model and semantics;
- (Section 3) Introduce the generic *region-based promotion cube* framework that can achieve a desired tradeoff between storage space and query execution time;
- (Section 4) Propose a *cost model* and a *provably optimal solution* for generating the most cost-effective cell structure through a solid theoretical analysis;
- (Section 5) Develop a *cell relaxation* approach to further optimize the storage overhead; and
- (Section 6) Present comprehensive experimental evaluation on real and synthetic data sets to verify that our framework is *1 to 2 orders of magnitude faster* than baseline solutions.

In addition, Section 7 discusses the related work, and Section 8 concludes this paper.

2. MODEL AND SEMANTICS

In this section we present our data model and formalize the query semantics.

Data model: Consider a data set *FactTable*$(\mathcal{A}, \mathcal{B}, T, M)$ consisting of base tuples with the following dimensions.

- α **categorical dimensions** $\mathcal{A} = \{A_1, A_2, \ldots, A_\alpha\}$: for each $A_i \in \mathcal{A}$, *dom*(A_i) is a finite collection of categorical values;

R (Region)	$F(\tau, R)$	*Rank*(τ, R)	*PRank*(τ, R)
$R_1 : \{a_1, b_1 \sim b_1\}$	0.7	1	100%
$R_2 : \{a_1, b_1 \sim b_2\}$	0.9	1	33%
$R_3 : \{a_1, b_1 \sim b_3\}$	0.9	2	50%
$R_4 : \{a_2, b_1 \sim b_4\}$	0.3	3	100%
$R_5 : \{a_2, b_1 \sim b_5\}$	1.5	2	67%
other regions omitted...	...	...	...

Table 2: Example regions and an object of interest $\tau = t_1$'s aggregate score (SUM), rank, and percentile rank in each region.

- β **continuous ranged dimensions** $\mathcal{B} = \{B_1, B_2, \ldots, B_\beta\}$: these are discretized numeric dimensions. For each $B_i \in \mathcal{B}$, *dom*(B_j) consists of an ordered set of ranges of values. A typical example is *dom*$(Year) = \{2010, 2009, \ldots\}$;
- **Object dimension** T and **measure dimension** M: *dom*(T) is the collection of **objects** and let n be the total number of distinct objects, *i.e.*, $n = |T|$. Let *dom*(M) be real numbers $\mathbb{R}$.

A **region** R is defined as $\{a_1, \ldots, a_\alpha, b_1 \sim b'_1, \ldots, b_\beta \sim b'_\beta\}$, where $a_i \in$ *dom*(A_i) or $a_i =$ " $*$ " (the "don't care" value) and $b_j, b'_j \in$ *dom*(B_j) and $b_j \leq b'_j$. Denote by $\mathcal{R}$ the set of **all regions** in the data set (we refer readers to [2, 6, 7] for detailed complexity analysis of $\mathcal{R}$).

Example 3: Table 1 displays a sample fact table consisting of a categorical dimension A and a range dimension B. T and M are the object and measure dimensions, respectively. In Table 2, the first column R displays several example regions. For instance, $\{a_1, b_1 \sim b_3\}$ represents the region "$A = a_1 \wedge b_1 \leq B \leq b_3$".

Query model: Consider an arbitrary aggregate function F (*e.g.*, *SUM*, *Average*, *Variance*). Given any region $R \in \mathcal{R}$ and any object $t \in T$, denote by $F(t, R)$ the **aggregate score** of t in R. Similarly, denote by *Rank*(t, R) t's **rank** in R, obtained by ordering objects *descendingly* or *ascendingly* according to the aggregate score. For clarity descending order is assumed throughout the paper. The REPQUERY problem is defined as follows.

Definition 1 (Top-k Region-based Promotion Query): Given a query $Q(\tau, k)$ consisting of an object of interest $\tau \in dom(T)$ *for promotion and a non-negative integer k, return $\mathcal{P}$, the ordered list of top-k regions, such that for $\forall R_1 \in \mathcal{R} - \mathcal{P}, \forall R_2 \in \mathcal{P}$ we have* $\text{Rank}(\tau, R_1) \times w(R_1) \geq \text{Rank}(\tau, R_2) \times w(R_2)$*, where $w(\cdot)$ is any positive weight function over $\mathcal{R}$.*

Ties are broken arbitrarily. We can see that, by setting $w(\cdot)$ to a positive constant, the REPQUERY model admits a simple object ranking semantics in that it asks for the top regions where the given object τ is highly ranked. A user may further model **percentile rank** (*PRank*) by letting $w(R)$ be the inverse of the number of objects present in region R. One may also let $w(R)$ be the inverse of R's number of tuples to discount small regions. Note that these rank-independent weights can be specified by users or combined in an ad-hoc way to tackle more complex scenarios.

Example 4: Continuing from Example 3, let the object of interest τ be t_1 and Table 2 shows τ's aggregate score (using SUM), rank, and percentile rank in the example regions. Not counting the omitted regions, R_1 and R_2 would be the top-2 promotion regions using simple ranking (because t_1 is ranked top-1 in both regions), whereas R_2 and R_3 would be the top-2 promotion regions according to PRank *(because t_1 is ranked top-33% and top-50% in them, respectively).*

Now let us further describe a query model that incorporates the redundancy-aware semantics.

Definition 2 (Top-k Discriminative Region-based Promotion Query): Given a query $Q(\tau, k, \theta)$, return $\mathcal{P}$, the ordered list of top-k discriminative regions*, such that for $\forall R_1 \in \mathcal{R} - \mathcal{P}$, we have either* $\forall R_2 \in \mathcal{P} \Rightarrow \text{Rank}(\tau, R_1) \times w(R_1) \geq \text{Rank}(\tau, R_2) \times w(R_2)$*, or* $\exists R_2 \in \mathcal{P} \Rightarrow \text{Sim}(R_1, R_2) \geq \theta \wedge \text{Rank}(\tau, R_1) \times w(R_1) \geq \text{Rank}(\tau, R_2) \times w(R_2)$.

Here we define *Sim*$(R_1, R_2) = \frac{|R_1 \cap R_2|}{|R_1 \cup R_2|}$, where $|R|$ denotes the number of base tuples contained in R. Other symmetric similarity measures can also be applied in principle. Intuitively, this definition is useful when users want to avoid redundant regions appearing in top-k results.

Example 5: Following the running example (Tables 1 and 2), consider a top-k discriminative query $Q(t_1, 2, 0.6)$. We can see that R_2 and R_3 have a similarity value $\text{Sim}(R_2, R_3) = |\{a_1, b_1 \sim b_2\}|/|\{a_1, b_1 \sim b_3\}| = 4/5 = 0.8 > \theta = 0.6$*. Thus, based on* PRank*, R_3 is no longer a top-2 promotion region since it is redundant with R_2; instead, now the top-2 discriminative promotion regions should be R_2 and R_5 (not counting the omitted regions in Table 2).*

Before presenting our solutions, assume that the aggregate function F is fixed. For clarify of presentation, also assume Definition 1 is used and let $w(\cdot)$ be 1 (*i.e.*, simple object ranking). More complex semantics will be discussed in Section 4.5.

3. REPCUBE: THE REGION-BASED PROMOTION CUBE FRAMEWORK

In this section we first motivate and present the RepCube framework (Sections 3.1, 3.2, and 3.3) and then present the REPQUERY execution algorithm as an integral part of the framework (Section 3.4). This framework lays the foundation for the subsequent cube structure optimization techniques (Sections 4 and 5).

3.1 No Materialization and the `GetRank()` Primitive

Let us first consider a no-materialization strategy. In this case, a promotion query must be computed from scratch. The basic query execution method is to enumerate each region $R_i \in \mathcal{R}$ ($1 \leq i \leq |\mathcal{R}|$) and compute *Rank*$(\tau, R_i)$; the top-$k$ promotion regions are maintained and outputted. During query execution, we abstract out a data access primitive `GetRank(`i`)`, for computing *Rank*(τ, R_i). `GetRank(`i`)` accesses all base tuples in R_i, computes aggregate scores for all objects, and derives τ's rank. Clearly, `GetRank()` is expensive due to its *holistic* property: all objects must be aggregated to correctly compute τ's rank. This holistic property coupled with a large number of regions would make on-the-fly execution extremely expensive even with some aggregation cost sharing and pruning techniques.

3.2 Full Materialization and the `GetAgg()` Primitive

On the other extreme, a full-materialization approach means to precompute all object aggregate scores for all regions. During query

Algorithm 1: Query Execution

```
    /* Pruning phase */
1:  for i ← 1 to |R| do
2:      F(τ, R_i) ← GetAgg(i); /* not costly */
3:      LBRank_i ← φ_j + 1, where j satisfies
            PCell^i_j > F(τ, R_i) ≥ PCell^i_{j+1};
4:      UBRank_i ← φ_l − 1, where l satisfies
            PCell^i_{l−1} ≥ F(τ, R_i) > PCell^i_l;
5:  end
6:  δ = the k-th smallest UBRank_i for 1 ≤ i ≤ |R|;
7:  R* ← {R_i | LBRank_i ≤ δ}; /* unpruned set of regions */
    /* Verification phase */
8:  foreach unpruned region R_i ∈ R* do
9:      Rank(τ, R_i) ← GetRank(i); /* costly */
10: end
11: Return P, the top-k regions with the smallest
    Rank(τ, R_i);
```

Table 3: The complete query execution algorithm.

execution, we abstract out another data access primitive $\texttt{GetAgg}(i)$, which computes $F(\tau, R_i)$ as follows: simply retrieves all base tuples in R_i related to τ and aggregates them. Given a full materialization, a query can be executed in 2 steps for each of the $|\mathcal{R}|$ regions: first, call $\texttt{GetAgg}(i)$ to get $F(\tau, R_i)$, and second, derive $\mathit{Rank}(\tau, R_i)$ by counting the materialized aggregate scores in R_i greater than $F(\tau, R_i)$. The top-k answers can be subsequently computed. Compared to $\texttt{GetRank}()$, $\texttt{GetAgg}()$ is much less costly because it only accesses τ's base tuples, which is especially efficient in the presence of a clustered index on T.

Not surprisingly, the storage overhead would be prohibitive. For example, if a data set has 1 categorical dimension and 2 range dimensions with an average cardinality of 100 as well as $10K$ objects, the full materialization approach would approximately generate $101 * (100 * 101/2)^2 * 10K \approx 2.6 * 10^{13}$ values, or nearly $100TB$ of disk space for a single aggregate function!

3.3 The Uniform RepCube Structure

To balance the storage overhead and online execution time, we use a method similar to quantization that samples aggregate scores at predefined positions from a sorted list of aggregate scores for each region R_i. We call the materialized sample for each region a **p-cell**, defined as follows.

*Definition 3 (P-Cell): For any region $R_i \in \mathcal{R}$, denote by $F_1^i, F_2^i, \ldots, F_n^i$ the ranked object aggregate scores in R_i in decreasing order (w.l.o.g. assume no duplicate scores). Given a **position vector** of length m: $\vec{\phi} = (\phi_j)_{j=1}^m$ (where $0 \le m \le n$, $1 \le \phi_j \le n$, and $j < l \Rightarrow \phi_j < \phi_l$), define PCell^i as the vector of aggregate scores induced by $\vec{\phi}$, i.e., $\mathit{PCell}^i = (F^i_{\phi_j})_{j=1}^m$.*

The most common way of choosing the position vector is to select a collection of evenly spaced values from $\{1, 2, \ldots, n\}$. A materialization plan consisting of a collection of p-cells based on such a position vector is called a *uniform RepCube*.

Definition 4 (Uniform RepCube): A uniform region-based promotion cube is defined as a collection of p-cells, $\{R_i, \mathit{PCell}^i | 1 \le i \le |\mathcal{R}|\}$, where the position vector is set to $\vec{\phi} = (1 + \lfloor \frac{j-1}{m} \times n \rfloor)_{j=1}^m$.

Query execution	Storage overhead	Number of $\texttt{GetRank}()$ calls
No materialization	0	$\lvert\mathcal{R}\rvert$
Full materialization	Prohibitive	0
Uniform RepCube	Small	$< \lvert\mathcal{R}\rvert$ ($\vec{\phi}$ not optimized)
Optimal RepCube	Small	$\ll \lvert\mathcal{R}\rvert$ ($\vec{\phi}$ optimized)
Relaxed RepCube	Very small	Opt. with ϵ-relaxation bound

Table 4: A roadmap of different strategies studied in this paper.

The only parameter m controls the size of uniform RepCube; in particular, $m = 0$ corresponds to the no materialization strategy while $m = n$ the full materialization strategy. In effect, m is much smaller than the total number of objects n so that the uniform RepCube would be significantly smaller compared to a full-materialization approach.

3.4 Query Execution Algorithm

Let us describe the query execution algorithm given an object of interest τ and the uniform RepCube structure. Recall that its goal is to return the top-k regions where τ is the most highly ranked. The query execution works in 2 phases. First is a **pruning phase**, where upper and lower bound ranks of τ for each region can be computed using the uniform RepCube. Then the unpromising regions not possible to be in the top-k are pruned. The second is a **verification phase** where each of the potential top-k regions is verified such that τ's true rank can be computed.

Both phases can be succinctly represented using the $\texttt{GetRank}()$ and $\texttt{GetAgg}()$ primitives. The detailed algorithm is depicted in Table 3 and we elaborate on each step. The pruning phase computes the lower and upper bound ranks of τ for each region R_i and conduct pruning (Lines 1–7). Specifically, $\texttt{GetAgg}()$ is called to get τ's aggregate score (Line 2), which is subsequently compared to the region's materialized p-cell to obtain the highest possible rank LBRank_i (Line 3) and the lowest possible rank UBRank_i (Line 4) of τ in R_i (for correctness we define two dummy positions $\phi_0 = 0$ and $\phi_{m+1} = n + 1$ such that $\mathit{PCell}^i_0 = -\infty$ and $\mathit{PCell}^i_{m+1} = +\infty$). Next, δ is computed as a threshold, meaning that τ must rank no lower than δ in any top-k promotion region (Line 6). All regions with the best possible rank lower than δ can be safely pruned (Line 7). The second phase verifies the unpruned regions (Lines 8–10), where the $\texttt{GetRank}()$ method is called for obtaining the exact rank for each unpruned region (Line 9). Finally, the top-k promotion regions are outputted (Line 11).

Cost analysis: Since the costly $\texttt{GetRank}()$ method (Line 9) accounts for the bottleneck of the algorithm, the cost of the query execution algorithm is dictated by $|\mathcal{R}^*|$, the number of $\texttt{GetRank}()$ calls, which is in turn determined by the underlying pruning power of the materialized cube. In fact, the pruning power is correlated with the user-specified parameter m in the sense that when $m = 0$ (no materialization) no region would be pruned and when $m = n$ (full materialization) no $\texttt{GetRank}()$ call is needed since *LBRank* and *UBRank* are tight in this case. Therefore, the RepCube framework offers a controllable tradeoff between storage space and online execution cost.

4. PRUNING POWER OPTIMIZATION FOR REPCUBE

The uniform RepCube strategy samples aggregate scores at regularly spaced positions; however, an important intuition it fails to model is that typically in top-k promotion regions τ is very likely

to be highly ranked. Rather than using a uniform position vector, our idea here is to carefully select a position vector *adaptive to the underlying distribution of queries* in order to achieve much better pruning power given a limited amount of storage space. Intuitively, for example, when k is very small for most queries, it would be a better idea to store more samples toward lower positions in order to better bound the ranks. In this section, we model this intuition and present an optimal solution. Our high-level goal here is to solve the following optimization problem.

Definition 5 (The Pruning Power Optimization Problem): Given a limited space budget indicated by m, and a distribution of promotion queries, determine the best position vector $\vec{\phi}$ such that the expected promotion query execution cost is minimized.

To solve the problem, we first formulate a cost model to compute the expected REPQUERY cost as a function of the position vector (Sections 4.1 and 4.2). We then discuss a dynamic programming solution for selecting the positive vector that produces the provably optimal cost (Section 4.3).

Roadmap: Table 4 presents a summary of the methods studied in the paper. In the previous section we have explained the first 3 methods, namely on-the-fly execution, naive precomputation, and the uniform RepCube approaches. The optimal RepCube approach discussed in this section will further enhance the query efficiency as a result of a much smaller number of calls to `GetRank()`. Section 5 will discuss the *Relaxed RepCube* technique for reducing the storage space of the optimal RepCube.

4.1 The Unit Cost Model

As a building block to the overall cost model, we would like to model the basic case, that is, the cost of a single fixed REPQUERY $Q(\tau_0, k_0)$ given a position vector $\vec{\phi}$ with length 1 (*i.e.*, $m = 1$); in other words, only a single aggregate score sample is drawn for each region.

Let $R_{s_1}, R_{s_2}, \ldots, R_{s_{|\mathcal{R}|}}$ be the ordered list of regions sorted according to τ_0's rank, where $s_1, s_2, \ldots, s_{|\mathcal{R}|}$ is a permutation of $1, 2, \ldots, |\mathcal{R}|$ and $i < j \Rightarrow \mathit{Rank}(\tau_0, R_{s_i}) \leq \mathit{Rank}(\tau_0, R_{s_j})$. For ease of exposition we assume τ occurs in all regions and use a short notation $\mathit{Rank}(i)$ to denote $\mathit{Rank}(\tau_0, R_{s_i})$. Since $m = 1$, let $\vec{\phi}$ be a scalar ϕ_1 ($\in \{1, 2, \ldots, n\}$), and assume that ϕ_1's corresponding p-cells, $\{\mathit{PCell}^{s_1}, \mathit{PCell}^{s_2}, \ldots, \mathit{PCell}^{s_{|\mathcal{R}|}}\}$, have been precomputed.

Given these p-cells, let us hypothetically compute the rank bounds in the query execution algorithm. For $Q(\tau_0, k_0)$, the computation can be divided into two cases: (1) all regions $\{R_{s_i}\}$ satisfying $\mathit{Rank}(i) < \phi_1$ will have $\mathit{LBRank}_{s_i} = 1$ and $\mathit{UBRank}_{s_i} = \phi_1 - 1$ because the inequality $+\infty = \mathit{PCell}_0^{s_i} < F(\tau_0, k_0) < \mathit{PCell}_1^{s_i}$ holds; (2) conversely, all regions $\{R_{s_i}\}$ satisfying $\mathit{Rank}(i) > \phi_1$ will have $\mathit{LBRank}_{s_i} = \phi_1 + 1$ and $\mathit{UBRank}_{s_i} = n$. Without loss of generality, assume that there does not exist any region R_{s_i} such that $\mathit{Rank}(i) = \phi_1$ (any region satisfying this equation cannot be pruned since its *LBRank* would be 1 and *UBRank* would be n). Now, define i^* to be the value in $\{1, 2, \ldots, |\mathcal{R}|\}$ such that $\mathit{Rank}(i^*) < \phi_1 < \mathit{Rank}(i^* + 1)$ (let $\mathit{Rank}(|\mathcal{R}| + 1)$ be ∞). Based on i^*, we can precisely compute $\mathcal{R}^*$, the unpruned set of regions for $Q(\tau_0, k_0)$, as in either of the following cases:

- When $i^* < k_0$, since in this case there are less than k_0 regions with *UBRank* equal to $\phi_1 - 1$, the k_0-th *UBRank* would be n, meaning that $\delta = n$ (*i.e.*, the threshold δ has no pruning power). Hence, no region can be pruned and $\mathcal{R}^* = \mathcal{R}$.
- When $i^* \geq k_0$, there would be exactly i^* regions with *UBRank* equal to $\phi_1 - 1$ and *LBRank* equal to 1, so these regions will not be pruned. Hence, $\mathcal{R}^* = \{R_{s_1}, \ldots, R_{s_{i^*}}\}$.

Example 6: Suppose the query parameter k_0 is 2 and there are totally 100 objects and $|\mathcal{R}| = 6$ regions $R_1, R_2, \ldots, R_6$, in which τ_0 is ranked 38th, 35th, 26th, 41st, 29th, and 50th, respectively. Thus, $s_1 = 3, s_2 = 5, s_3 = 2, s_4 = 1, s_5 = 4$, and $s_6 = 6$. If we set the position vector (scalar) ϕ_1 to 27, we will obtain $i^* = 1$ because $\mathit{Rank}(1) < \phi_1 < \mathit{Rank}(2)$ (i.e., $26 < \phi_1 < 29$). In this case only in R_3 does τ_0 have UBRank $= \phi_1 - 1 = 26$ and in all other regions τ_0 has UBRank $= n = 100$, so δ, the k_0-th smallest UBRank, will be 100 and therefore none of the 6 regions can be pruned after the pruning phase. However, if we set ϕ_1 to 37, we will obtain $i^* = 3$ since $\mathit{Rank}(3) < \phi_1 < \mathit{Rank}(4)$. Thus R_3, R_5, and R_2 have UBRank $= 36$. In this case $\delta = 36$ and the remaining 3 regions can be pruned.

Now we are ready to present the unit cost model. We introduce some notation. Denote by

- $\mathit{COST}(Q|\phi_1)$ the overall query execution cost for $Q(\tau_0, k_0)$ given ϕ_1;
- Ω the constant cost of the pruning phase (Lines 1–7, Table 3);
- $\mathit{COST}(s_i)$ the cost of calling the `GetRank`(s_i) method for region R_{s_i}.

Since the overall query execution cost given ϕ_1 can be broken down to (1) the constant cost of the pruning phase and (2) the cost of the verification phase that consists of multiple `GetRank()` calls, we can formulate the unit query execution cost of a single query $Q(\tau_0, k_0)$ using a 1-length position vector $\vec{\phi} = \phi_1$ as the following (note that i^* can be computed using the method described earlier in this subsection):

$$\mathit{COST}(Q|\phi_1) = \begin{cases} \Omega + \sum_{i=1}^{i^*} \mathit{COST}(s_i), & \text{if } i^* \geq k_0 \\ \Omega + \sum_{i=1}^{|\mathcal{R}|} \mathit{COST}(s_i), & \text{if } i^* < k_0 \end{cases}$$

4.2 The Complete Cost Model

Now a step further. Consider the cost of a single query $Q(\tau_0, k_0)$ when a position vector $\vec{\phi} = \{\phi_j\}_{j=1}^m$ with arbitrary length $m \in [1, n]$ is given. As a more general case of the unit case, our goal now is to compute $\mathit{COST}(Q|\vec{\phi})$, the overall cost for $Q(\tau_0, k_0)$ given $\vec{\phi}$.

Note that now in each region we materialize m aggregate scores. For each $j \in \{1, 2, \ldots, m\}$, we let i_j^* be the value in $\{1, 2, \ldots, |\mathcal{R}|\}$ which satisfies $\mathit{Rank}(i_j^*) < \phi_j < \mathit{Rank}(i_j^* + 1)$. This intuitively means that if we materialize the aggregate score at position ϕ_j, we will be able to distinguish i_j^* regions from the remaining ones in terms of rank bounds. We have $i_1^* \leq i_2^* \leq \cdots \leq i_m^*$ because of the monotonicity of $\{\phi_j\}$ (*i.e.*, $\phi_1 < \phi_2 < \ldots < \phi_m$). Using a similar method for the unit cost model, the total cost of $Q(\tau_0, k_0)$ given $\vec{\phi}$ can be computed in either one of the following cases.

- When $i_m^* < k_0$, δ would be n for the same reason as in the case of $m = 1$. Hence, no region can be pruned and $\mathcal{R}^* = \mathcal{R}$.

- Otherwise, let i^* be the smallest value in $\{i_j^*\}$ to satisfy $i^* \geq k_0$. Let u be the subscript satisfying $i_u^* = i^*$. Observe that, based on the computation of *LBRank* and *UBRank*, there are exactly i^* regions having *UBRank* $\leq \phi_u - 1$, and the remaining $|\mathcal{R}| - i^*$ regions having *LBRank* $\geq \phi_u + 1$. This means that $\delta = \phi_u - 1$ and the latter $|\mathcal{R}| - i^*$ regions will be pruned. Hence, $\mathcal{R}^* = \{R_{s_1}, \ldots, R_{s_{i^*}}\}$.

Consequently, $COST(Q|\vec{\phi})$ can be formulated as:

$$COST(Q|\vec{\phi}) = \begin{cases} \Omega + \sum_{i=1}^{|\mathcal{R}|} COST(s_i) & \text{if } i_m^* < k_0 \\ \Omega + \sum_{i=1}^{i^*} COST(s_i) & \text{otherwise} \end{cases}$$

Finally, to complete the overall cost model formulation, suppose the top-k region-based promotion queries are drawn from a multivariate distribution $Q \sim p(\tau, k)$, and denote by $COST_{all}$ the variable of the overall cost induced by $p(\tau, k)$. Then the expected overall cost for any given position vector $\vec{\phi}$ can be computed as:

$$E(COST_{all}|\vec{\phi}) = \int_Q COST(Q|\vec{\phi}) p(Q) dQ. \quad (1)$$

Because our goal is to decide the best position vector so as to minimize the expected overall cost, the objective of the pruning power optimization problem becomes to obtain

$$\vec{\phi}^* = \arg\min_{\vec{\phi}} E(COST_{all}|\vec{\phi}). \quad (2)$$

4.3 An Optimal Solution for Maximizing the Pruning Power

This subsection discusses an efficient dynamic programming solution to compute the optimal position vector $\vec{\phi}^*$ defined in Equation 2. The idea of the dynamic programming solution is to solve a series of recurrences represented by a matrix $MinCost[i, j]$ ($0 \leq i \leq n, 0 \leq j \leq m$). Each element of the matrix $MinCost[i, j]$ represents the minimum expected overall cost that can be achieved when selecting a j-length position vector $\vec{\phi}$ with the very last position value being i (*i.e.*, $\phi_1 < \phi_2 < \ldots < \phi_j = i$). Corresponding to $MinCost[\cdot, \cdot]$, we use another matrix $\Phi[\cdot, \cdot]$ to remember the optimal position vector that achieves $MinCost[i, j]$, *i.e.*, $MinCost[i, j] = E(COST_{all}|\Phi[i, j])$ and the last value in vector $\Phi[i, j]$ is i. The minimum value in $MinCost[\cdot, \cdot]$ will be the optimal expected overall cost.

The set of recurrences can be computed as follows. Initially, set $MinCost[0, j] = MinCost[i, 0] = +\infty$ for $0 \leq i \leq n$ and $0 \leq j \leq m$, respectively, as boundary cases; also set the corresponding $\Phi[0, j]$ and $\Phi[i, 0]$ to empty vectors. This initial setting means that the cost is large when nothing is materialized. Then:

$$MinCost[i, j] = \min \begin{cases} MinCost[i, j-1]; \\ MinCost[l, j-1] - \Delta(i, j, l), \\ \qquad \text{for each } 0 \leq l < i; \end{cases}$$

$$\Phi[i, j] = \begin{cases} \Phi[i, j-1], \\ \quad \text{if } MinCost[i, j] = MinCost[i, j-1]; \\ \Phi[l, j-1] \oplus i, \\ \quad \text{if } MinCost[i, j] = MinCost[l, j-1]; \end{cases}$$

Before getting into the details of the above equations, we can see that the optimal position vector of $\Phi[i, j]$ can be derived by considering either the minimum cost of an existing solution $\Phi[i, j-1]$ (*i.e.*, the optimal $(j-1)$-length position vector by considering the last positions being i), or a set of new solutions $\Phi[l, j-1] \oplus i$ for $0 \leq l < i$ (*i.e.*, new vectors composed by appending the value i to previous solutions $\Phi[l, j-1]$). For each such new solution $\Phi[l, j-1] \oplus i$, its cost can be expressed as $MinCost[l, j-1] - \Delta(i, j, l)$, where $\Delta(i, j, l) = E(COST_{all}|\Phi[l, j-1]) - E(COST_{all}|\Phi[l, j-1] \oplus i)$, *i.e.*, the reduction in expected query execution cost.

Based on Equation 1, the computation of the expected overall costs $E(COST_{all}|\Phi[l, j-1])$ and $E(COST_{all}|\Phi[l, j-1] \oplus i)$ requires the knowledge about query distribution. To obtain the distribution, we may assume that a query workload W consisting of w queries $\{Q_1, Q_2, \ldots, Q_w\}$ is given. In the absence of such a query workload, one may either assume that $p(\tau, k)$ be a uniform distribution and draw sample queries from it, or use application-dependent knowledge (*e.g.*, a query interface that returns the top-10 regions). Given such a workload, $\Delta(i, j, l)$ can be computed as

$$\begin{aligned} & \Delta(i, j, l) \\ = \; & E(COST_{all}|\Phi[l,j-1]) - E(COST_{all}|\Phi[l,j-1]\oplus i) \\ = \; & \textstyle\sum_{Q\in W} p(Q) COST(Q|\Phi[l,j-1]) - \sum_{Q\in W} p(Q) COST(Q|\Phi[l,j-1]\oplus i) \\ = \; & \textstyle\sum_{Q\in W} p(Q) \big(COST(Q|\Phi[l,j-1]) - COST(Q|\Phi[l,j-1]\oplus i)\big). \end{aligned}$$

Therefore, using the above procedure, one can iterate through each $0 \leq i \leq n$ and $0 \leq j \leq m$ and fill in the matrices *MinCost* and Φ. The recurrence with the minimum cost, $\min\{MinCost[i, j]\}$, would indicate the minimum expected overall cost.

Claim 1 (Solution Optimality): Letting

$$(i^{opt}, j^{opt}) = \arg\min_{(i,j)} MinCost[i, j],$$

we have

$$\vec{\phi}^* = \Phi[i^{opt}, j^{opt}],$$

where $\vec{\phi}^*$ is defined in Equation 2.

Proof sketch: The proof of the optimality of the dynamic programming solution follows from two properties. (1) *Monotonicity*: appending a new position value to an existing vector would never increase the overall expected cost. This is obvious as materializing more aggregate scores would not hurt the efficiency query execution. (2) *Substructure optimality*: the cost reduction by appending a new position value, $\Delta(i, j, l)$, depends only on position l but none of the positions prior to l (in other words, once we know $MinCost[l, j-1]$, the values of $MinCost[l', j-1]$ for $l' < l$ would not affect $MinCost[i, j]$). As a result, at each iteration we can guarantee that $MinCost[i, j]$ is optimal to the subproblem corresponding to that iteration. Based on these properties, we can prove that the dynamic programming equations generate the best overall solution.

4.4 Implementation

Based on the selection of optimal positions, an *optimal RepCube* can be implemented in 2 steps. First, given a query distribution or workload, compute the optimal position vector $\vec{\phi}^*$ through dynamic programming. Second, for each region, materialize its p-cell according to $\vec{\phi}^*$. In this subsection, we discuss several issues concerning the implementation.

Dynamic programming complexity: The dynamic programming algorithm can be implemented using 3 nested loops for i, j, and

Algorithm 2: Generalized Query Execution

```
    /* Line 1 here is the same as Algorithm 1 Lines 1–5 */
1:  for i ← 1 to |R| do compute LBRank_i and UBRank_i
2:  C ← {R_1, R_2, ..., R_|R|}; /* candidate regions */
3:  P ← empty list;
4:  while |P| ≤ k ∧ |C| > 0 do
5:      R ← the region in C having the highest rank of τ;
6:      Append R to P;
7:      C ← C\{R'|Sim(R', R) ≥ θ};
8:  end
9:  Return P, the top-k discriminative regions;
```

Table 5: Computing top-k discriminative promotion regions.

l, respectively. At each loop iteration, evaluating $\Delta(i, j, l)$ is the bottleneck because of repetitive evaluations of $COST(Q|\vec{\phi})$. To be more efficient, we materialize $\{Rank(\tau, R_1), \ldots, Rank(\tau, R_{|\mathcal{R}|})\}$ for each $Q \in W$ upfront. This way, $COST(Q|\vec{\phi})$ can be evaluated efficiently without accessing the original data set. The complexity of the nested loop would be $O(n^2mw)$. When n is extremely large (*e.g.*, $1M$), a heuristic is to limit the choices of positions to a regularly sampled subset of $\{1, 2, \ldots, n\}$.

Cost model parameters: The assignment of the cost model's parameters Ω and $COST(s_i)$ (Sections 4.1 and 4.2) can be as follows: set Ω to τ's cardinality (*i.e.*, the number of base tuples containing τ) and $COST(s_i)$ to region R_{s_i}'s number of tuples. Other parameters may be used depending on the actual implementation.

4.5 Extensions

In our previous discussion we have proposed techniques for the simple object ranking semantics. We now discuss how we can extend those to support ad-hoc weight function as well as the top-k discriminative query semantics.

Ad-hoc weight function: To handle REPQUERY with arbitrary weight functions, the query execution algorithm can be extended with only minor modification. Specifically, in Algorithm 1, when computing the threshold δ (Line 6) and the unpruned set of regions $\mathcal{R}^*$ (Line 7), $LBRank_i$ and $UBRank_i$ should be replaced by $LBRank_i \times w(R_i)$ and $UBRank_i \times w(R_i)$ (Lines 3–4) respectively. Correspondingly, in the cost model formulation (Section 4.2), $COST(Q|\vec{\phi})$ need to compute the unpruned set of regions $\mathcal{R}^*$ in the same fashion.

Top-k discriminative promotion regions: To compute the top-k discriminative regions according to Definition 2, we show a *generalized query execution algorithm* in Table 5. The pruning phase of Algorithm 2 (Line 1) is the same as that in Algorithm 1 in that rank bounds are computed based on the RepCube. However, the verification phase of Algorithm 2 differs from the previous one in that the top-k discriminative regions are now computed one-by-one (Lines 4–8) rather than as a batch; in other words, at each iteration we verify only the next top discriminative region and remove all its similar regions.

In fact this algorithm generalizes Algorithm 1, because if we set $\theta = 1$, the two algorithms will verify the same set of regions. The only additional cost of Algorithm 2 lies in computing $Sim(\cdot, \cdot)$ (Line 7), which can be done efficiently. Also, for any region, Algorithm 2 guarantees that `GetRank()` will be called at most once. Due to the limited space, a detailed analysis of the cost model based on this generalized algorithm is not discussed here.

5. RELAXING CELLS FOR SPACE REDUCTION

In this section, we study techniques to further reduce the storage overhead of the RepCube. The idea here is to merge multiple p-cells with similar aggregate scores and represent them using a single *relaxed cell*. Specifically, instead of materializing the exact scores of a p-cell, we store *score ranges* within a predefined bound. Thus, other p-cells whose exact scores are *covered* by these score ranges can be represented by the relaxed cell.

Definition 5 (ϵ-Relaxed Cell): Given a region R's p-cell, $PCell = \{f_1, f_2, \ldots, f_m\}$, define its normalized cell as $\{1.0, \frac{f_2}{f_1}, \ldots, \frac{f_m}{f_1}\}$. Given a relaxation parameter $\epsilon \geq 0$, define the corresponding ϵ-relaxed p-cell as $RCell = \{1.0 \pm \frac{\epsilon}{m}, \frac{f_2}{f_1} \pm \frac{\epsilon}{m}, \ldots, \frac{f_m}{f_1} \pm \frac{\epsilon}{m}\}$.

We elaborate on this definition. The normalization step normalizes aggregate scores that could be at very different scales for subsequent cell merging. This step is important as we observe that different regions may have very similar trends in aggregate score distributions but the absolute values could be quite different. For example, the distribution of the *SUM* of sales in $\{Year = 2010\}$ could be similar to that in $\{Year = 2009 \sim 2010\}$ but differ by a factor 2 in scale. Since in any p-cell f_1 is the largest aggregate score, dividing each cell value by f_1 would make all normalized values to be within range $[0, 1]$. In principle, other normalization methods may also be applied here for the same purpose. Given the relaxation parameter ϵ, each value in an ϵ-relaxed cell would represent a set of ranges of aggregate scores. Specifically, the i-th value of the relaxed cell represents $[f_i - f_1 \times \frac{\epsilon}{m}, f_i + f_1 \times \frac{\epsilon}{m}]$ if f_1 is known. Now, we are ready to introduce a more compact RepCube structure based on a set of ϵ-relaxed cells.

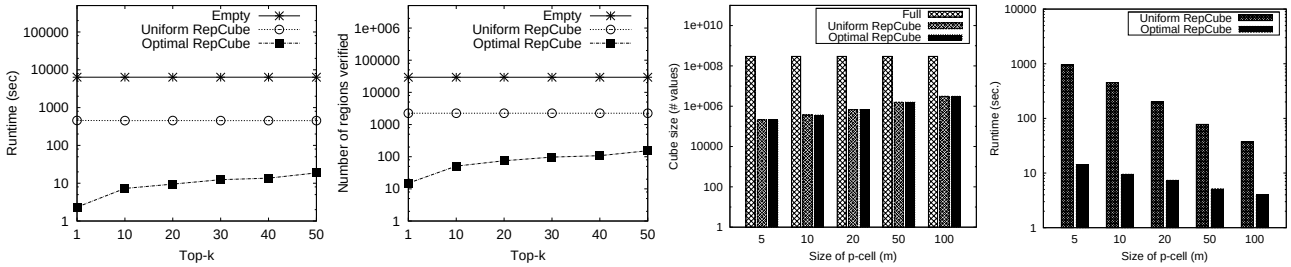
Definition 6 (ϵ-Relaxed RepCube): An ϵ-relaxed RepCube consists of a collection of r ϵ-relaxed cells $\{RCell^1, RCell^2, \ldots, RCell^r\}$ ($1 \leq r \leq |\mathcal{R}|$), and a surjective mapping function g from each R_i to some relaxed cell, i.e., $g : \{1, 2, \ldots, |\mathcal{R}|\} \rightarrow \{1, 2, \ldots, r\}$. Also the normalization score f_1 is stored for each R_i.

The ϵ-relaxed RepCube contains no more than r relaxed cells. This means that one or more regions are mapped to a same relaxed cell. We require that these regions' p-cells be covered by the ranges of the relaxed cell they are mapped to. Notice that each region still maintains m values, so the total size of a relaxed cube would be much smaller than the original cube when $r \ll |\mathcal{R}|$.

The query execution algorithm in Table 3 need slight modification at the pruning phase (Lines 3–4) to accommodate the relaxed cube. Given R_i, $F(\tau, R_i)$, and a relaxed cell $RCell^{g(i)} = \{1.0 \pm \frac{\epsilon}{m}, \frac{f_2}{f_1} \pm \frac{\epsilon}{m}, \ldots, \frac{f_m}{f_1} \pm \frac{\epsilon}{m}\}$, the computation of $LBRank_i$ (Line 3) now should be computed as $\phi_j + 1$ for j satisfying $f_j - \frac{\epsilon}{m} \times f_1 > F(\tau, R_i) \geq f_{j+1} - \frac{\epsilon}{m} \times f_1$. Similarly, $UBRank_i$ (Line 4) should be $\phi_l - 1$ for l satisfying $f_{l-1} + \frac{\epsilon}{m} \times f_1 \geq F(\tau, R_i) > f_l + \frac{\epsilon}{m} \times f_1$. Note that these bounds guarantee the precision of the query execution algorithm. The pruning power of the relaxed RepCube, on the other hand, will be similar to the original RepCube when the relaxation parameter ϵ is chosen to be very small.

5.1 A Greedy Algorithm

Given an original RepCube with $|\mathcal{R}|$ p-cells, we would like to select the smallest subset of their corresponding relaxed cells (*i.e.*, to



(a) Runtime vs. top-k. (b) Number of verified regions vs. top-k. (c) Storage overhead vs. the size of p-cell (d) Runtime vs. the size of p-cell

Figure 1: Comparison with baseline solutions on the default TPCH data set.

minimize r) with the constraint that each p-cell must be covered by some relaxed cell. It turns out that this problem is NP-hard with a reduction from the SETCOVER problem: the set of original p-cells are transformed to the *universe of elements* and each potential relaxed cell is transformed to a *set of elements*.

Due to the hardness of the problem, we use a greedy algorithm to iteratively select relaxed p-cells as follows. First, initialize the set of selected relaxed cells as an empty set, and mark all p-cells as "uncovered". Then, add the relaxed cell that is able to cover the largest number of uncovered p-cells into the selected set, and mark those newly covered p-cells as "covered". Repeat the above step until all p-cells are marked as "covered". The corresponding mapping function can be maintained during the above process. Finally the selected relaxed p-cells will be kept in memory or written back to disk.

The parameter ϵ controls the resulting size of the relaxed RepCube. When $\epsilon = 0$, only identical p-cells will be merged. On the other hand, when ϵ is too large, a single relaxed cell suffices to cover all p-cells but is unlikely to provide any pruning power. In effect, a small ϵ less than 0.1 often produces good tradeoff. We manually set it in our experiments and it remains an open problem to automatically determine ϵ.

6. EXPERIMENTS

We conduct case study on the DBLP data set and comprehensive experiments on the standard TPC-H benchmark. Our goal is to: (1) demonstrate the top-k results through a case study, (2) show that our proposed methods can significantly outperform a baseline solution in terms of *query execution time*, and (3) verify that the *storage space* used by our methods is very small.

All our experiments were done on a machine with a 2.5GHz duo-core CPU, 4GB of RAM, and 250GB hard disk. The OS is Windows XP Pro SP3 and all source code was written and compiled in Microsoft Visual C# 2008.

6.1 A Case Study on DBLP

We constructed a fact table from the DBLP data set using *Conference* as the categorical dimension, *Year* as the continuous ranged dimension, *Author* as the object dimension, and *Paper Count* as the measure dimension. The table contains about 1.8 million base tuples and $450K$ authors.

For the query author *Bruce Lindsay*, we found his global rank to be 5112th. However, the top-3 regions (except the global region) based on *PRank* are $\{$*VLDB*, $1990 \sim 1991\}$, $\{$*ICDE*, $1993 \sim 1993\}$, and $\{$*SIGMOD*, $1998 \sim 2002\}$, where he is ranked (5th, top-2.1%), (4th, top-2.2%), and (4th, top-2.8%), respectively. Not surprisingly, the results are quite meaningful.

On the other hand, his *(Rank, PRank)* in the promotion region $\{$*SIGMOD*, $1998 \sim 2002\}$'s five child regions, namely $\{$*SIGMOD*, $1998 \sim 1998\}$ through $\{$*SIGMOD*, $2002 \sim 2002\}$, are only (21st, top-9.3%), (36th, top-14.6%), (28th, top-11.8%), (29th, top-12.6%), and (37th, top-16%), respectively. These results indicate that it is indeed interesting to discover the "right" region for promotion. We leave a systematic evaluation of various other semantics to our future work.

6.2 Evaluation on TPCH

For efficiency evaluation, we chose the TPC-H benchmark[2] to generate large decision support data. The *default fact table* was generated as follows. We ran the *dbgen* executable with default parameters to generate a set of data files and extracted the *lineitem.tbl* file containing 6,001,215 base tuples. We set *l_linenumber* (cardinality=7) as the categorical dimension, *l_quantity* (50) and *l_linestatus* (2) as the range dimensions, *l_suppkey* (10000) as the object dimension (*i.e.*, $n = 10000$), and *l_extendedprice* (real numbers) as the measure dimension. Thus, for this default fact table there are totally 30600 regions. This table is stored in Microsoft SQL Server 2008.

Algorithm implementation: We implemented the following 5 methods.

- (Empty) On-the-fly query execution without any auxiliary materialization as a baseline for online query execution time;
- (Full) Precomputing aggregate scores for all objects in all regions as a baseline for storage overhead;
- (Uniform) The uniform RepCube approach;
- (Opt) The optimal RepCube approach; and
- (Relax) The relaxed RepCube approach.

All of these 5 methods rely on 2 interface primitives `GetAgg()` and `GetRank()`. The former primitive was implemented by ourselves, while the latter was implemented as a query in SQL Server. To speedup query processing, a clustered index was built on the object

[2]http://www.tpc.org/tpch/

RepCube method	Position vector $\vec{\phi}$
Uniform	1, 1001, 2001, 3001, 4001, 5001, 6001, 7001, 8001, 9001
Opt + Default	6, 18, 33, 48, 89, 142, 234, 357, 593, 1084
Opt + Large entropy	214, 659, 1009, 1621, 1869, 2465, 3326, 3726, 4416, 5448
Opt + Accurate	4, 7, 23, 26, 76, 111, 132, 187, 283, 328

Table 6: Position vectors used by different RepCube methods.

dimension and multi-key non-clustered indices were built on categorical and range dimensions. All materialization files were stored as plain text files.

To formulate a cost model and derive the optimal position vector for Opt, we set Ω, the constant cost of the pruning phase, to 0, and let $COST(s_i)$ be 1 for any region R_{s_i} (see Section 4 for the definitions of Ω and $COST(s_i)$).

To produce the set of top-k regions, *SUM* was used as the aggregation function F and we consider the k regions with the largest *percentile rank* as the top-k results.

Performance measure: We focus on *runtime* (*i.e.*, average online query processing time per query, in terms of *seconds*) and *size* (*i.e.*, offline storage space an algorithm is used, in terms of *number of values* stored) as the main performance metrics of these algorithms. We do not count the time for loading the materialized data.

6.3 Online Query Execution Time vs. Top-k

Now we compare the runtime of Empty, Uniform, and Opt by varying the query parameter k. The performance results for Relax will be reported shortly.

We set $m = 10$ for both Uniform and Opt such that the resulting size of Uniform is 367,201 values and that of Opt is 367,210 values. To compute the position vector for Opt based on the cost model, we generated a *default workload* consisting of 200 promotion queries $Q(\tau, k)$, where for each query τ was uniformly randomly generated and k was uniformly randomly distributed over $[1, 160]$.

A set of 5 *random test queries* was generated as follows: 5 objects were uniformly randomly generated, and k was varied from 1 to 50. Figure 1(a) displays the average runtime (in log-scale) of Empty, Uniform, and Opt on these 5 test queries. We can see that the baseline solution Empty is over an order of magnitude slower than Uniform, the basic RepCube strategy, while Empty is over 3 orders of magnitude slower than Opt at $k = 1$, and > 300 times slower than Opt at $k = 50$. Also, the performance of Empty does not change with respect to top-k because it does not involve any pruning. This test clearly shows that computing the region-based promotion query from scratch can be prohibitively expensive. In our subsequent experiments we will not evaluate Empty any more due to its apparent low efficiency.

Observe that Uniform it is 190 times slower than Opt at $k = 1$ and 24 times slower at $k = 50$. The Uniform approach, with some precomputed information, is able to significantly outperform Empty, but it turns out to be quite insensitive to k as well, because it is not able to leverage the fact that the object of interest is often highly ranked in the top-k promotion regions; in other words, the pruning power of Uniform would be similar no matter the object is highly ranked in the top-k regions or not. On the contrary, Opt offers much better pruning power as it is able to precompute sample aggregate scores in an adaptive way. As a result, it is more sensitive to k and more efficient when k is smaller.

To accurately explain the gap of runtime between different methods, in Figure 1(b) we plot the average number of verified regions (*i.e.*, the number of unpruned regions as shown in Table 3, Line 7) with respect to k in the same test. As can be seen, this figure matches Figure 1(a) well. This validates our claim that the query execution time is dominated by the cost of `GetRank()`. Indeed, Empty need on average about 30000 calls of `GetRank()` (note that there are some regions where the object of interest does not appear so `GetRank()` does not have to be called for them), while Uniform and Opt need no more than 2300 and 160 calls for any k, respectively.

6.4 Storage Overhead vs. P-Cell Size

Now we compare the storage overhead of Full, yet another baseline strategy, with Uniform and Opt. We vary m, the size of p-cell from 5 to 100 (*i.e.*, 5 to 100 aggregate scores are sampled for each region) and show the resulting storage space required by each method in Figure 1(c). Fully precomputing aggregate scores for all objects in all regions requires about $3 * 10^8$ values. Suppose each aggregate score uses 8 bytes to store, Full would consume 2.2GB of disk space, which is much larger than the size of the input data set. This tells us that Full may not be a practical solution for large applications. In contrast, Uniform and Opt store no more than $220K$ values (1/1380 of Full's size) at $m = 5$; $370K$ values (1/805 of Full's size) at $m = 10$; and $3.2M$ values (1/94 of Full's size) at $m = 100$, which significantly alleviate the problem of expensive storage overhead. Note that the difference of storage overhead between Uniform and Opt is very small ($< 10^{-4}$ of the total cube size). In principle, the resulting size of Opt is linearly related to m, so users are able to conveniently specify m to control the size and obtain their desired performance.

Figure 1(d) further displays how the p-cell's size would affect the online performance of our proposed methods. For each m, we used exactly the same set of 5 test queries and fix k to 20. We also used the same default workload to generate Opt's position vector as we did for the previous test. The average query execution time is reported for both Uniform and Opt. We can see that when m increases, the efficiency of both Uniform and Opt becomes higher. This is expected as increasing the p-cell size would help derive tighter upper- and lower- bounds for any object in any region, thereby leading to more pruned regions.

The relation between Uniform and Opt as shown in Figure 1(d) is also interesting. When $m = 5$, Opt is 66 times faster than Uniform. As m is increasing, the gap between the two approaches actually become smaller. For example, the speedup ratio is 15 at $m = 50$ and 9.1 at $m = 100$. Indeed, too large m might lead to a convergence of Opt to Uniform; an extreme case is that when $m = n$, the performance of Opt and Uniform would be identical

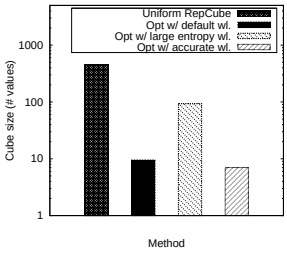
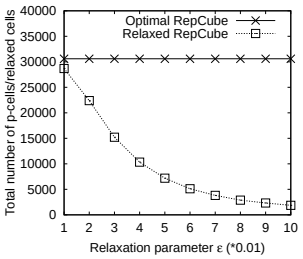
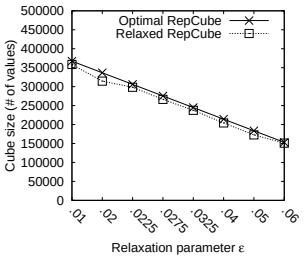


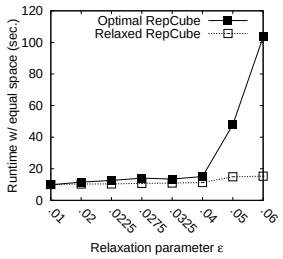
Figure 2: Query execution time vs. query distribution.



(a) Num. of relaxed cells generated by the greedy algorithm.



(b) Size of Opt (by varying m) and Relax (by varying ϵ).



(c) Query execution time vs. storage overhead

Figure 3: Performance results on the relaxed RepCube.

due to a same position vector. Nevertheless, this result validates our idea that Opt performs much better than Uniform when m is reasonably small (*e.g.*, $\frac{m}{n} < 1\%$), which is desirable for large data sets.

6.5 Performance of the Optimal RepCube over Different Query Distributions

In our previous tests we computed Opt's position vector $\vec{\phi}$ using the default workload, *i.e.*, we assumed that in the query distribution object IDs were drawn uniformly and k uniformly randomly from 1 through 160. Because in different applications, such query distributions can be vastly different, in this subsection we test the performance of Opt using different workloads. Besides the default workload used earlier, we generated two other workloads:

- *Large-entropy workload:* Consists of queries with τ uniformly randomly sampled from the set of all objects and k uniformly distributed on 1 through 10000. In other words, this workload assumes the promotion query has a large entropy (*i.e.*, randomness).
- *Accurate workload:* Given the 5 test queries used earlier, we generated a superset of them as an accurate workload (totally 10 queries). Therefore, Opt would be specifically optimized for the test queries using this workload.

We ran Opt against the 3 workloads and obtained 3 position vectors. Figure 2 plots the average query execution time of Opt vs. these workloads over the 5 test queries; we also plot the result of Uniform for comparison. Interestingly, we observe that Opt outperforms Uniform in all 3 cases. In particular, Opt based on large-entropy workload runs 5 times faster than Uniform, whereas Opt based on the accurate workload does 60 times faster. These results match our intuition, because the large-entropy workload can be considered as "adversarial" since the query distribution used by the cost model largely deviates from test queries, while the accurate workload would help maximize the pruning power of the optimal RepCube generated for those test queries. The gap between Uniform and Opt with the large-entropy workload, however, is unexpected.

To clearly explain the performance gap, Table 6 shows the position vectors used by each method. Uniform uses 10 evenly spaced positions. For Opt, however, the position vectors have smaller values; in fact, even the uniform workload "prefers" smaller position values strictly based on the cost model. Therefore, these results empirically proved that choosing an evenly spaced position vector (*i.e.*, uniform quantization) cannot produce desirable pruning power for promotion query.

6.6 Performance of the Relaxed RepCube

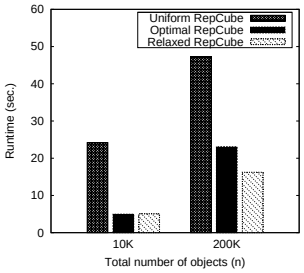
Now we evaluate Relax. Recall that Relax's parameters m and ϵ dictates the resulting size of materialization. Based on the optimal RepCube ($m = 10$) for the default fact table discussed earlier, we varied ϵ and ran the greedy relaxed cell selection algorithm. Figure 3(a) depicts the resulting size of Relax for ϵ ranging from 0.01 to 0.1 on an increment of 0.01. While Opt must store 30600 p-cells constantly, Relax need fewer and fewer relaxed cells as ϵ increases. For example, when $\epsilon = 0.01$, 28695 relaxed cells can cover all p-cells, and when $\epsilon = 0.1$, only 1865 cells would suffice, where each relaxed cell covers an average of 16.4 p-cells. Hence the result confirms that similar p-cells and can be merged effectively.

Let us turn to a comparison between Relax and Opt. Since it is unfair to compare their runtime using different amounts of storage space, the methodology adopted here is to first generate Relax and Opt with similar size, elaborated as follows. First, we generated a set of 8 optimal RepCubes by varying the p-cell size m from 10 down to 3. The resulting sizes of Opt ranges from 367,210 to 153,003. Second, to generate a relaxed cube with comparable size to each of the 8 optimal RepCubes, we manually tried different ϵ parameters and ran the greedy algorithm on Opt with $m = 10$. Finally we chose ϵ to be 0.01, 0.02, 0.0225, 0.0275, 0.0325, 0.04, 0.05, and 0.06, respectively, and Figure 3(b) displays the size of both Opt and Relax in the 8 cases. For instance, when $m = 8$ for Opt, we set ϵ to 0.0225, and then Opt's size is slightly above $300K$ and Relax's size is slightly less than $300K$. We guarantee that the size of Relax be no larger than Opt in all cases.

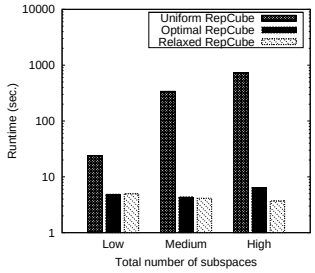
For each matched pair of Relax and Opt, we ran the 5 test queries and reported the average query execution time in Figure 3(c). The figure shows that Relax beats Opt in all but the first case. Relax is considerably more efficient than Opt in the last two cases. For the second to last case, Relax with $\epsilon = 0.05$ is 3.2 times faster than Opt with $m = 4$; whereas for the last case, Relax with $\epsilon = 0.06$ is 6.7 times faster than Opt with $m = 3$. Even in the first case the performance gap can be neglected. Hence, our conclusion is that Relax indeed gives the best tradeoff between storage space and query execution time, since it is faster than Opt yet using less space.

6.7 Varying Data Characteristics

To compare the performance of the proposed methods on different data characteristics, we first generated a new fact table using the $6M$-tuple *lineitem* table. We fixed the categorical dimension to



(a) Query execution time vs. number of objects (n).



(b) Query execution time vs. number of regions ($|\mathcal{R}|$).

Figure 4: Performance results with different data characteristics.

l_linenumber (7) but changed range dimensions to *l_discount* (11) and *l_linestatus* (2). We fixed the measure dimension to *l_extendedprice*, whereas the object dimension was changed to *l_partkey* ($200K$), indicating that the number of object is $200K$. The workload was the default one and the test queries were the same before.

The new data with $n = 200K$ contains a total number of 1584 regions. Full would consume more than $187M$ values, or $1.4G$ of disk space equivalently. When setting m to 10, Uniform and Opt have about the size of $19K$ values, about $\frac{1}{9850}$ of Full's size. We can see that the RepCube approaches achieve a better storage saving for larger number of objects as expected. For Relax, we first generated an optimal RepCube with $m = 18$ and then set ϵ to 0.059 to produce a relaxed RepCube using $< 19K$ values (701 relaxed cells generated).

As shown in Figure 4(a), Opt outperforms Uniform by 2 times. Since the total number of regions is smaller than in the previous test cases, the speedup ratio is not as large as in the previous tests; in fact, this ratio would increase with respect to $|\mathcal{R}|$ as will be shown shortly. Relax is in turn 1.4 times faster than Opt, thereby again beating Opt in both storage overhead and runtime. It also turns out that for $n = 10K$ (*i.e.*, using *l_suppkey* ($10K$) as the object dimension while keeping other dimensions fixed), the performance of Opt and Relax is similar due to fewer regions.

Let us turn to the total number of regions. In previous tests we have synthesized two fact tables with 1584 regions (denoted by "*LOW*" hereafter) and 30600 regions (denoted by "*MEDIUM*" hereafter), respectively. In addition to *LOW* and *MEDIUM*, we produced another *HIGH* data set from the *lineitem* table by setting *l_quantity* (50) and *l_returnflag* (3) as range dimensions while keeping other dimensions fixed ($10K$ objects). *HIGH* contains 61156 regions and the Full approach would generate $> 580M$ values ($4.5G$ disk space). For *HIGH*, we again set m to 10 and generated Uniform and Opt with considerably smaller space overhead (*i.e.*, $< 734K$ values), while repeated the previous approach for Relax (*i.e.*, $< 730K$ values and each relaxed cell on average covers 2.16 p-cells). Figure 4(b) shows the performance comparison of Uniform, Opt, and Relax on *LOW*, *MEDIUM*, and *HIGH*, respectively. We can see that (1) Opt becomes increasingly faster than Uniform, *i.e.*, 4.9 times faster on *LOW* and 113.6 times on *HIGH*; and (2) Relax again shows its scalability, *i.e.*, using less space than Opt yet being 1.7 times faster than it on *HIGH*.

Based on the experimental results displayed in Figure 4, our conclusions are: (1) Both Opt and Relax perform consistently and significantly faster than the basic RepCube implementation (Uniform) during online processing; and (2) Relax, although using less space, is more efficient than Opt on large data sets, demonstrating its scalable tradeoff in terms of the number of objects as well as regions.

6.8 Performance on Aggregate Function *AVG*

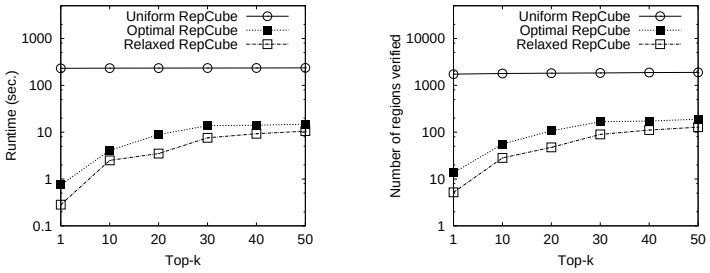
Our final test case in Figure 5 compares Uniform, Opt, and Relax based on another aggregate function *AVG*. That is, in each region, objects are ordered descendingly according to their average measure dimension values. The *MEDIUM* fact table is used here. The generation of Uniform and Opt remains unchanged with $m = 10$. For Relax, we first generated an optimal RepCube using $m = 18$. Then, ϵ was set to 0.027 such that we ensure Relax's size be smaller than Opt; this is a notable difference between *AVG* and *SUM* in that here the p-cells can be merged more easily using a smaller value of ϵ. This indicates that each relaxed cell represents a "tighter" range than for the *SUM* aggregate function.

Figure 5(a) reports the runtime of the methods when varying k from 1 to 50. We can see that Uniform does not have satisfactory performance, while Relax is consistently about 2 times faster than Opt. Figure 5(b) further confirms that the verification step dominates the query execution cost, which is invariant to the aggregate function. The results obtained here thus prove that the efficiency gain of Opt and Relax are not restricted to some particular monotone measures.

7. RELATED WORK

Data warehousing and online analytical processing (OLAP) techniques have been extensively studied over a decade for decision support applications [6, 2]. In a typical OLAP system, data is collected and consolidated in a repository for subsequent analysis and mining. The data cube model [5] is a widely accepted tool which materializes multidimensional aggregates to speedup online operations like roll-up, drill-down, thereby greatly facilitating interactive exploration of warehoused data. Researchers have also proposed methods for effectively and efficiently computing various OLAP functions, such as range queries [7], ranking [14, 16], regression and classification [3, 10], sequence data [4, 9], and others [1, 8]. However, none of these methods can be applied to solve the range-based promotion analysis problem, since for a promotion query the cube space (region) must be searched at query time as opposed to be given by user.

The most related work to our study is [15], which proposes the promotion analysis concept and discusses methods for removing spurious conditions and avoiding searching too "deep" into the mul-



(a) Runtime vs. top-k on *AVG*.

(b) Number of verified regions vs. top-k.

Figure 5: Performance results on aggregate function *AVG*.

tidimensional space. Compared to the previous work, this paper addresses a new problem with three salient features: (1) we model the region-based semantics with *ad-hoc rank-independent weights* which may or may not be monotone with respect to the size of region; also *region similarities* (*e.g.*, containment or overlapping relationships) are considered and the top-k discriminative query problem is formulated; (2) the REPQUERY problem is faced with a *significantly larger search space* than [15], making previously proposed query processing techniques infeasible; and, (3) the *non-monotonicity* property is assumed in our problem context, meaning that aggregate scores are not monotone across parent-children regions.

In addition to the above papers, [11] presents an algorithm that can simultaneously rank objects and cluster them into proper communities. However, such communities discovered must belong to some predefined granularity (*e.g.*, research areas) that cannot be combined with continuous dimensions like "Year". Moreover, [12] studies the reverse top-k query processing problem that aims to search for interesting user preference spaces where a given product is highly ranked. The techniques developed cannot address the region-based promotion query because they do not consider multidimensional aggregation. [13] is yet another paper discussing a problem related to marketing applications. Nevertheless, they use skyline dominance relationship rather than numerical ranking as the criteria to judge the interestingness of query results.

8. CONCLUSION

This paper studied a novel class of decision support queries called the top-k (discriminative) region-based promotion query. A region-based promotion cube framework was developed. We showed that a uniform materialization approach is indeed not the best; instead, an adaptive approach was developed based on a solid theoretical analysis to produce the provably optimal structure. In addition, a compact relaxed cube structure was studied to further optimize storage overhead. Comprehensive experiments on both real and synthetic data sets verified both the effectiveness and efficiency our proposed techniques.

9. REFERENCES

[1] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. Olap over uncertain and imprecise data. *VLDB J.*, 16(1):123–144, 2007.

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

[3] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *VLDB*, pages 323–334, 2002.

[4] C. K. Chui, E. Lo, B. Kao, and W.-S. Ho. Supporting ranking pattern-based aggregate queries in sequence data cubes. In *CIKM*, pages 997–1006, 2009.

[5] 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.

[6] J. Han and M. Kamber, editors. *Data mining: concepts and techniques, second edition*. Morgan Kaufmann, 2006.

[7] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *In Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 73–88, 1997.

[8] C. X. Lin, B. Ding, J. Han, F. Zhu, and B. Zhao. Text cube: Computing ir measures for multidimensional text database analysis. In *ICDM*, pages 905–910, 2008.

[9] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. Olap on sequence data. In *SIGMOD Conference*, pages 649–660, 2008.

[10] R. Ramakrishnan and B.-C. Chen. Exploratory mining in cube space. *Data Min. Knowl. Discov.*, 15(1):29–54, 2007.

[11] Y. Sun, J. Han, P. Zhao, Z. Yin, H. Cheng, and T. Wu. Rankclus: integrating clustering with ranking for heterogeneous information network analysis. In *EDBT*, pages 565–576, 2009.

[12] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag. Reverse top-k queries. In *ICDE*, 2010.

[13] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *PVLDB*, 2(1):898–909, 2009.

[14] T. Wu, D. Xin, and J. Han. Arcube: supporting ranking aggregate queries in partially materialized data cubes. In *SIGMOD Conference*, pages 79–92, 2008.

[15] T. Wu, D. Xin, Q. Mei, and J. Han. Promotion analysis in multi-dimensional space. In *VLDB*, 2009.

[16] 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.