Efficient Top-K Processing Over Query-Dependent Functions

Lin Guo Sihem Amer Yahia Raghu Ramakrishnan Jayavel Shanmugasundaram
Utkarsh Srivastava Erik Vee

Yahoo! Research
{guolin, sihem, ramakris, jaishan, utkarsh, erikvee}@yahoo-inc.com

Abstract

We study the efficient evaluation of top-k queries over data items, where the score of each item is dynamically computed by applying an item-specific function whose parameter value is specified in the query. For example, online retail stores rank items by price, which may be a function of the quantity being queried: "Stay 3 nights, get a 15% discount on double-bed rooms." Similarly, while ranking possible routes in online maps by predicted congestion level, the score (congestion) is a function of the time being queried, e.g., "At 5PM on a Friday in Palo Alto, the congestion level on 101 North is high." Since the parameter—the number of nights or the time the online map is queried, in the above examples—is only known at query time, and online applications have stringent response-time requirements, it is infeasible to evaluate every item-specific function to determine the item scores, especially when the number of items is large. Further, space considerations make it infeasible to pre-compute and store the score of each item for each value of the input parameter. In this paper, we develop a novel technique that compresses the (large) set of item scores for all parameter values by dividing the parameter range into intervals, taking into account the expected query workload. This compressed representation is then used to do top-k pruning of query results. Our experiments show that the proposed techniques are scalable and efficient.

1. Introduction

We address the problem of efficiently executing a new class of top-k queries on very large sets of items. A query can specify selection criteria on the items and the value v of a parameter that controls item scores, and ask to see the top k items ranked by score, where the score of each item is computed by an item-specific function that depends on v.

Such queries arise naturally in a number of domains such as online-shopping, traffic planning, and financial analysis. For example, in online-shopping,[1] users can specify selection criteria (e.g., the color/model of a cell phone) over the inventory of items through a form-based interface, and ask to have selected items sorted by price. The novel aspect of the problem that we consider is that item price could reflect, for example, price discounts based on promotional rules such as the following: "Stay 3 nights, get a 15% discount," or "Buy 2 Canon printer cartridges, get the third one free," or "Buy 2 Motorola Razr cell-phones, get $50 off." Thus, the score for ranking (i.e., the price) is a function of a parameter in the query—such as the quantity of items being purchased, or the number of nights stayed—that determines which promotional rules apply.

As another example, consider online traffic information sites,[2] which return recommended traffic routes to users, and incorporate time-of-day traffic congestion data in making a recommendation. For a particular time-of-day, the congestion level of a route may be estimated by rules such as "At 3PM, congestion level on highway 280 in a 10-mile radius around Palo Alto is high." When a user asks for the least congested routes in a particular geographic region (the selection criterion) at a given time, the system may wish to quickly determine the top few least-congested routes in that region. In this case too, the score (i.e., the congestion level) of a particular route is not constant but is a function of the time-of-day, which is a parameter specified in the query.

As a final example, consider predictive financial modeling, where the prices of individual stocks are modeled as a function of time, and analysts wish to determine the top few predicted gainers/losers in a particular category (selection condition) at a future time instant. Again, the score (i.e., the expected gain/loss of a stock) of a particular stock is a function of the future time instant of interest, which is a query parameter.

Problem Definition. The top-k problem we consider can be formalized as follows. We are given a set of items I, a set of parameter values $\mathcal{V}$, and a family of functions, $f_i : \mathcal{V} \rightarrow \mathcal{R}$ associated with each item $i \in I$. For a query $Q = (\texttt{Pred}, v, k)$, where `Pred` is a selection condition on items, $v \in \mathcal{V}$ is the value of the parameter controlling item scores, and k is the desired number of results, we wish to compute a result set R that contains the k lowest-score

[1] *http://travelocity.com, http://shopping.yahoo.com*
[2] *http://maps.yahoo.com, http://maps.google.com*

PVLDB '08, August 23-28, 2008, Auckland, New Zealand

items satisfying Pred, *where the score of an item i is defined to be* $f_i(v)$. (Note: lower score $\Rightarrow$ higher rank)

We restrict our attention in this paper to discrete sets $\mathcal{V}$ that have a few hundred to a few thousand parameter values. We note that when $\mathcal{V}$ is a continuous set, rather than a discrete set, our approach can be modified by approximating each f_i with a piecewise linear function, where again, the number of pieces is relatively small. However, we do not explore these modifications here.

As an illustration of the above problem definition, consider our online-shopping example. Here, I is the set of products being sold, $\mathcal{V}$ is the set of possible product quantities in an order, and $f_i(v)$ is a function that computes the unit price of product i for a given quantity v. For a query Q, Pred is a selection condition on products (e.g., "Make = Canon" and "Color = Blue"), v is the desired quantity of a given product, and k is the number of products that can be shown on the result page. For the online traffic information example, I is the set of routes, $\mathcal{V}$ is the set of possible times of day, and f_i is a function that computes the congestion level of route i for a given time of day. For a query Q, Pred is a selection condition on routes (e.g., "Location = Palo Alto"), v is the time of day, and k is the desired number of least congested routes. The predictive financial modeling example can also be mapped similarly.

A naive solution to this problem is to select all the items i that satisfy the query predicate, compute the score $f_i(v)$ for each i, sort by score, and return the items with the lowest score. This approach does not scale to a large number of items and/or unselective predicates, especially if the functions f_i are expensive.

Another simple solution is to precompute and store the score for every (item, parameter value) pair. Queries can then be answered efficiently by simply looking up the top-scored selected items for a given query value. However, the typically large number of items (e.g., all products, all stock symbols) taken in conjunction with many possible parameter values (e.g., all possible quantities, all times-of-day) makes precomputing prohibitively expensive in terms of space. Note that even if every item does not have an f_i that varies with the query parameter value (e.g., even if every item is not on sale), a large number of items have such f_i's (e.g., a significant number of items are on sale online), which results in a large space overhead. This space overhead is particularly bad for large online applications, where all the data and indices are stored in main-memory in order to achieve acceptable throughput and response time (see Section 4 for more details).

To address these limitations, we propose a novel approach that works as follows. Instead of storing the score for every (item, parameter value) pair as in the precomputation approach, we store a dramatically compressed representation of this data. We do so by exploiting the fact that the query parameter values are drawn from (or can be mapped to) an underlying ordered domain; in our examples, quantity and time-of-day are such parameters. The key idea is to split the parameter values associated with an item into one or more *intervals*, and then store only the *minimum* score for each (item, interval) pair. The total number of intervals is chosen such that they fit within a specified space budget. We then perform top-k query processing by adapting threshold-based pruning [12, 13] to prune a large number of intervals (and the corresponding items) that cannot possibly make it to the top few results.

To see the benefits of our approach, consider the function f_i shown in Figure 1(b). It can naturally be split into two intervals $Ival_1$ and $Ival_2$. $Ival_1$ captures value range $v = 1$ (i.e., $1 \leq v \leq 1$) and the minimum score of f in that range is 150; $Ival_2$ captures the value range $v \geq 2$ and the minimum score of f in that range is 0.90×150. Thus, in this example, just by storing two intervals for the item, we obtain a representation that does not lose any information about the function. A more complex function f_j is shown in Figure 1(c). Here, f_j is split into three intervals, but the minimum score for each interval only approximates f_j.

Clearly, the effectiveness of our approach depends on how well the intervals are chosen. One of the main technical contributions of this paper is an algorithm that takes as input a given set of items, the corresponding functions, and a space budget, and then uses query workload information to produce a set of intervals that are provably close to optimal for that workload. The algorithm scales linearly with the number of items, and makes few assumptions about the nature of functions. Specifically, the algorithm only assumes that (a) we can efficiently find the minimum value of f_i (or a relatively tight lower bound of the minimum value of f_i) for a given parameter range, which is true for most rule-based score computations such as the ones described above, and (b) the number of distinct parameter values (such as quantities, times-of-day, etc.), t, in the query workload is of the order of hundreds or thousands because the complexity of the algorithm is cubic in t.

We have experimentally evaluated our approach using Yahoo! Shopping data. Our experimental results show that the proposed approach offers significant savings in space and time when compared to alternative approaches. Further, our results show that exploiting the query workload to choose the appropriate intervals for top-k query processing also results in significant gains. To the best of our knowledge, this is the first attempt to optimize the performance of top-k queries using query workloads.

In summary, the main contributions of the paper are:

- Efficient top-k query processing techniques over function intervals (Section 2).
- Showing how the the problem of computing optimal intervals for a set of items given a query workload can be decomposed into a set of small sub-problems corresponding to each item, which can then be solved and combined efficiently using standard optimization techniques (Section 3).
- Experimental evaluation of the proposed algorithms (Section 4).

2. Proposed Approach

Figure 2 describes the high level architecture of our proposed approach. The query processing module uses the indexed items and intervals to answer queries (the process is initially started with a default set of intervals, such as a single interval for each item). The interval generation module gathers statistics from the query processing module and refines the intervals so that they are tuned to the query workload and fit into the space budget. We focus on the query processing module in this section, and address interval generation in Section 3.

The query processing module conceptually uses two tables as shown in Figures 3 and 4 (these tables are physically optimized using indices, as will be described shortly). The Item table contains a row for each item i, which includes i's attributes and the function f_i. The Interval table contains a set of intervals corresponding to each item i, and each interval contains the low value of the interval ($lowv$), the high value of the interval ($highv$) and the minimum value of f_i for that particular interval (That is, $minf_i$

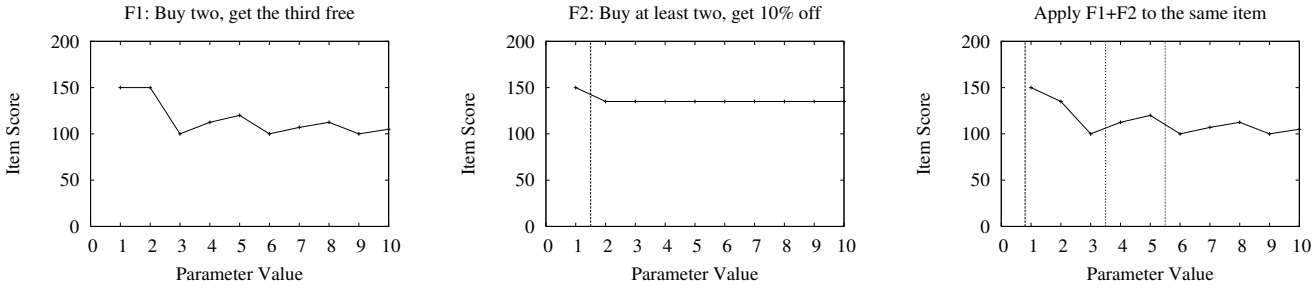

Figure 1. Functions Corresponding to Promotional Rules

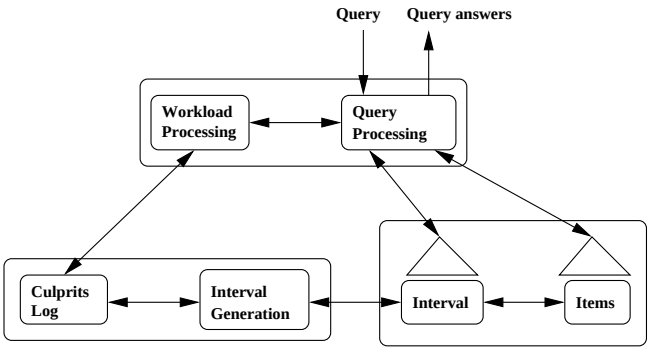

Figure 2. System Architecture

Id	Make	Model	Weight	f_i
1	Panasonic	DC643	0.35lbs	f_1
2	Panasonic	GDC65	0.22lbs	f_2
3	Siemens	D345	0.45lbs	f_3
4	Motorola	Razr	0.22lbs	f_4
5	Motorola	Sleek	0.21lbs	f_5
6	Motorola	Rokr	0.23lbs	f_6
7	Motorola	Krzr	0.23lbs	f_7
8	Motorola	Q	0.30lbs	f_8
9	Motorola	Rizr	0.26lbs	f_9

Figure 3. The `Item` Table

IntId	ItemId	$lowv$	$highv$	$minf_i$
1	3	2	∞	55
2	6	1	∞	60
3	3	1	1	80
4	5	3	∞	80
5	2	1	∞	90
6	7	3	∞	95
7	4	6	∞	100
8	4	1	3	100
9	1	3	∞	110
10	4	4	5	112
11	7	2	2	115
12	5	1	2	120
13	7	1	1	150
14	8	1	∞	230
15	1	1	2	250
16	9	1	∞	255

Figure 4. The `Interval` Table

ItemId	quantity, v	$f_4(v)$
4	1	150
4	2	135
4	3	100
4	4	112
4	5	120
4	6	100
4	7	106
4	8	112

Figure 5. Scores for `ItemId` 4

$= \min_{v \in [lowv, highv]} f_i(v)$.) As an illustration, consider the item with `ItemId` 4. A table summarizing its score (i.e. f_4) is shown in Figure 5. We see in Figure 4 that it has 3 intervals associated with it — $[1, 3]$, $[4, 5]$, $[6, \infty]$ — each of which is associated with a different `IntId` and its corresponding $minf_4$. Specifically, the smallest value of f_4 on the interval $[1, 3]$, associated with `IntId` 8, is (referring again to Figure 5) $100 = \min\{150, 135, 100\}$, while the smallest value of f_4 on the interval $[4, 5]$, associated with `IntId` 10, is $112 = \min\{112, 120\}$.

Given a query $Q = (\mathtt{P}reds, v, k)$, the main idea of the query processing algorithm is to (a) identify the intervals containing v that correspond to items satisfying $\mathtt{P}reds$, (b) process these intervals in the order of their score, and (c) stop as soon as the top-k items corresponding to these intervals of found. In order to speed up (a), we index the item attributes and intervals so that we can rapidly look up the items that satisfy $\mathtt{P}reds$ and intervals that contain v. Specifically, we create a (temporary) table that is the join of the item table and the interval table, index the item attributes using traditional B+-trees, and index the interval attribute using an interval/segment tree [2]. The temporary table is sorted in ascending order of $minf_i$ so that the B+-tree indices and interval/segment tree store the interval row ids in score order. Given these index structures, producing the intervals that match the query in the order of their $minf_i$ is simple: we look up the indices correspoding to the attributes in $\mathtt{P}reds$ and v and simply do a merge-based intersection to produce ids in score order.

Algorithm 1 shows the query processing algorithm. Given a query $Q = (\mathtt{P}reds, v, k)$, the algorithm first produces intervals in $minf_i$ order as described in step (a) above (line 2). For exam-

ple, suppose we had query $Q = (\texttt{Make} = \text{'Motorola'}, 5,2)$. Then the algorithm will (conceptually) generate the list L represented in Figure 6. Of course in practice, this entire list will not be generated; only the first few interval ids necessary for the algorithm will be processed. For each interval id, the algorithm determines the interval $ival$ corresponding to that id (line 4) and checks to see if $minf_i$ is greater than or equal to the score of the kth element in the heap (line 5; if there are less than k elements in the heap, the condition always fails). If this condition is true, then we know that the item corresponding to $ival$, and all items corresponding to intervals that occur after $ival$ in L, can never make it to the top-k results (note that this reasoning is similar to that in the Threshold Algorithm [12, 13]). Consequently, query processing can terminate and the algorithm can return without accessing the remaining items. Otherwise, the item corresponding to $ival$ is added to the ResultHeap if its $f_i(v)$ is lower than the score of the k^{th} item in the result heap (lines 9-10).

Algorithm 1 Query Processing Algorithm

Require: $Preds$, v, k
1: **return** top-k answers in order of score (lowest score first)
2: L := List of ids of `Interval`s that contain v and whose items satisfy $Preds$ (found via indices on Item and Interval tables)
3: Initialize ResultHeap of size k
4: **for** (id in L in increasing order of id) **do**
5: ival = getInterval(id);
6: **if** ($ival.minf_i \geq$ score of kth item in ResultHeap) **then**
7: break;
8: **else**
9: $iscore = getItem(id).f_i(v)$;
10: **if** ($iscore <$ score of k^{th} item in ResultHeap) **then**
11: ResultHeap.add($ival.itemId$, $iscore$);
12: **end if**
13: **end if**
14: **end for**

In our example with $Q = (\texttt{Make} = \text{'Motorola'}, v = 5, k = 2)$, the algorithm first considers `ItemId` 6, with $minf_i$=60, the first entry in Figure 6. It finds the true score $f_6(5)$, which is 108.5. Since ResultHeap is empty, it adds `ItemId` 6 to the heap with score 108.5 (lines 10-11). Next, the algorithm considers `ItemId` 5, with true score $f_5(5) = 150$, now adding `ItemId` 5 to the heap with score 150. The algorithm continues on to `ItemId` 7 with score $f_7(5) = 130$. Since this is smaller than the kth score of the heap (here, $k = 2$), it replaces `ItemId` 5 with `ItemId` 7 in the heap. Next, the algorithm examines `ItemId` 4 with score $f_4(5) = 120$. Again, this is better than the kth score of the heap, thus, `ItemId` 4 replaces `ItemId` 7. Finally, the algorithm processes `ItemId` 8. Now, $minf_i$ is larger than the kth score of the heap, so `ItemId` 8 and any further items in the list cannot possibly be better than what we have seen so far. Thus, the algorithm terminates (line 7), returning `ItemId` 6 and 4.

3. Interval Selection Algorithm

We now focus on the problem of selecting intervals. The key challenge is to use the query workload to determine the best set of intervals that (a) reduce the overall query processing time, and (b) satisfy the space budget constraints. The naive solution—enumerating all possible sets of intervals—has computational complexity that is exponential in the number of items, which is clearly infeasible. However, we show that we can exploit some key properties relating f_i's and item intervals to develop an algorithm that is both efficient and provably close to optimal. Since the algorithm is efficient, it can be periodically run offline to tune the intervals to changing query workloads.

IntId	ItemId	$lowv$	$highv$	$minf_i$	$f_i(5)$
2	6	1	∞	60	108.5
4	5	3	∞	80	150
6	7	3	∞	95	130
10	4	4	5	112	120
14	8	1	∞	230	230
16	9	1	∞	255	260

Figure 6. List of intervals matching query $Q =$ ($\texttt{Make} =$ 'Motorola', $5, 2$)

3.1 Problem Statement

We begin by motivating our cost model. If we consider the cost of evaluating a query Q using Algorithm 1, we can identify two components of the overall cost. The first component is the *fixed cost*, which is the cost of evaluating Q, *independent* of the choice of intervals. The fixed cost has three parts: (1) the index probes (line 2),[3] (2) k iterations of the for loop that add the top-k results to the result heap (lines 10-11), and (3) the final iteration of the for loop when the termination condition is satisfied (lines 6-7). If we computed and stored all possible intervals, then each query would only incur the fixed cost. In the example algorithm run using $Q = (\texttt{Make} = \text{'Motorola'}, 5,2)$, processing `ItemId` 6 and 4 count as fixed cost (part (2)), as does processing `ItemId` 8 (part (3)), since these are necessary steps regardless of the intervals chosen. Note that under a different choice of intervals, we may process a different item than 8 in part 3. However, we must process some item to trigger our termination condition.

The second component of the cost is the *variable cost*, which is the cost of evaluating a query after excluding the fixed cost. This component of the cost depends on the choice of intervals. Given a query Q and a specific choice of intervals $\mathcal{P}$, if the Algorithm 1 iterates over its for loop m times, then the variable cost is the cost of evaluating $m - k - 1$ iterations. (We arrive at the number $m - k - 1$ because out of the total of m iterations, k iterations are used to produce the actual top-k results, and the last iteration is for the termination condition.). These $m - k - 1$ iterations correspond to items/intervals that are processed by the algorithm but which never make it to the top-k results, i.e., they correspond to intervals whose minimum score is lower than the max top-k score (otherwise, they would not have been processed by the algorithm) but whose actual score for the particular query parameter is greater than the max top-k score (because they are not part of the top-k results). Each of the items that correspond to these $m - k - 1$ iterations of the variable cost is called a *culprit*. In our example, both `ItemId` 5 and 7 are culprits. The max top-k score was actually 120, but we processed both `ItemId` 5 and 7 since their respective $minf_i$ scores were 80 and 95, less than 120. The actual score, $f_i(5)$, of these items is 150 and 130, respectively, so they were not in the

[3]The results of the index probes may differ based on the choice of intervals, but the number of index probes (which is the main determinant of the cost) is independent of the choice of intervals.

top-k list. (Had we chosen different intervals, these items would never need to be processed.)

Our goal is to minimize the total variable cost over all queries in a query workload $QW = [Q_1, ..., Q_n]$. In other words, we wish to minimize all costs other than the minimum fixed cost that must be incurred for each query Q_i. Before formalizing this problem statement, we define the notion of *partition*. Let I be the set of items, and let $Ivals$ be the set of all possible parameter intervals.

DEFINITION 1. **Partition.** A *partition* $\mathcal{P}$ is a function $\mathcal{P} : I \rightarrow 2^{Ivals}$ such that for all $i \in I$, the intervals in $\mathcal{P}(i)$ (a) are non-overlapping (to avoid redundancy), and (b) cover the entire value range (to avoid missing values).

Intuitively, a partition is just a formal way to denote a specific choice of intervals. In particular, $\mathcal{P}(i)$ specifies the set of intervals for item i. For example, $\mathcal{P}(4) = \{[1, 3], [4, 5], [6, \infty]\}$ in our running example, while $\mathcal{P}(3) = \{[1, 1], [2, \infty]\}$.

Recall that the variable cost of evaluating a query Q using a partition $\mathcal{P}$ is defined as the cost of evaluating each one of the $m-k-1$ iterations (lines 10-11 in Algorithm 1). We consider the cost of each iteration to be a single unit and then define the variable cost of query Q using partition $\mathcal{P}$, denoted $varcost(I, \mathcal{P}, Q)$, to be $m - k - 1$. We use the notation $culprits(I, \mathcal{P}, Q)$, to refer to the set of items whose intervals are processed in the $m - k - 1$ iterations of Q that contribute to its variable cost. In our example, $culprits(I, \mathcal{P}, Q) = \{5, 7\}$, and $varcost(I, \mathcal{P}, Q) = 2$.

As we mentioned earlier, if there were no limit on the number of intervals, $\sum_{i \in I} |\mathcal{P}(i)|$, then we could choose a partition $\mathcal{P}$ such that $varcost(I, \mathcal{P}, Q) = 0$ and $culprits(I, \mathcal{P}, Q) = \emptyset$ for all Q — essentially, this creates an interval for each distinct query parameter value for each item. In practice, we must limit the number of intervals due to memory limitations.

We can now formally state our problem:

Problem Definition. *Given a set of items I, the set of all possible value intervals $Ival$, a query workload QW, and a space budget s, find a partition $\mathcal{P}$ that minimizes the overall variable cost $\sum_{Q \in QW}(varcost(I, \mathcal{P}, Q))$, subject to the space constraint $\sum_{i \in I} |\mathcal{P}(i)| \leq s$.*

3.2 Proposed Algorithm

A simple way to solve the above problem is to explicitly enumerate all the partitions that satisfy the space budget, compute the cost for each such partition, and finally pick the partition that has the minimum cost. However, this is likely to be very inefficient due to the large number of possible partitions. Specifically, if the number of distinct query parameter values is t, then the number of possible partitions is $\binom{2t \times |I|}{s - |I|}$. (There are $2t$ interval split points for each f_i, one before and one after every parameter value seen; thus, the total number of interval split points for all items is $2t \times |I|$. From these, we need to choose $s - |I|$ split points, since we start with $|I|$ intervals and each additional split increases the number of intervals by one.) Thus, for even modest sized databases, such as one having 10000 items, 10 parameter values and a space budget of 20000, we have $\binom{2 \times 10^5}{10^4}$ possible partitions!

Fortunately, it turns out that we can exploit a key property relating partitions that dramatically reduces the set of partitions that need to be considered. We first introduce some notation.

DEFINITION 2. **Variable Cost of an Item.** The variable cost for an item $i \in I$ given a partition $\mathcal{P}$ and a query workload QW is defined to be:

$$vc_i(I, \mathcal{P}, QW) = |\{Q \mid Q \in QW \wedge i \in culprits(I, \mathcal{P}, Q)\}|$$

(In this definition, $\{\}$ refers to a bag, not a set, in order to deal correctly with duplicate queries.)

In other words, the variable cost for an item i is the number of times the item appears as a culprit in the query workload, i.e., the number of times an interval associated with an item is processed by Algorithm 1 without the item being part of the final top-k result (recall that a culprit is defined just above the problem definition in the previous section). It is easy to see that $\sum_{i \in I} vc_i(I, \mathcal{P}, QW) = \sum_{Q \in QW} varcost(I, \mathcal{P}, Q)$, i.e., the sum of the variable costs of all items is the same as the sum of the variable costs of all queries (which in turn is the same as the overall variable cost).

Let $maxscore(I, Q)$ denote the kth largest score from among the results obtained by evaluating Q over I (i.e., the maximum value of $f_i(v)$ among the items in the top-k results). The following lemma says essentially that if two different partitions agree on item i, then vc_i agrees for both partitions, over any query workload.

LEMMA 1. **Independence Property.** Given a set of items I and a space budget s, let $AllParts$ be the set of all partitions that satisfy the space budget. Then, given a query workload QW:

$$\forall i \in I, \forall \mathcal{P}_1, \mathcal{P}_2 \in AllParts, (\mathcal{P}_1(i) = \mathcal{P}_2(i)$$
$$\Rightarrow\ vc_i(I, \mathcal{P}_1, QW) = vc_i(I, \mathcal{P}_2, QW))$$

PROOF. Consider a partition $\mathcal{P} \in AllParts$ and a query $Q = (\texttt{Preds}, v, k) \in QW$. Let $vIval_{Q,i}$ be the interval in $\mathcal{P}(i)$ containing v. (Recall that the $\mathcal{P}(i)$'s are non-overlapping and cover the entire value range, so there is exactly one interval that satisfies this condition.) From Algorithm 1, we can see that for an item i and query Q:

$$i \in culprits(I, \mathcal{P}, Q) \Leftrightarrow maxscore(I, Q) < \min_{v' \in vIval_{Q,i}} f_i(v')$$

i.e., i is a culprit iff its minimum score in the interval that contains v is less than the top-k maximum score. Thus, $vc_i = |\ \{Q \mid Q \in QW \wedge maxscore_Q < min_{v' \in vIval_{Q,i}} f_i(v')\}\ |$, which only depends on $\mathcal{P}(i)$ (in the definition of $vIval_{Q,i}$), and does not depend on $\mathcal{P}(j), j \neq i$. This proves the claim. □

Informally, the property states that the benefit of choosing a particular set of intervals for item i is *independent* of the choice of intervals for other items. That is, the value chosen for $\mathcal{P}(i)$ does not affect the value of vc_j for item $j \neq i$. Consequently, we can solve the problem for each item separately, and then combine to produce the overall solution. We will show the overall complexity of our algorithm that exploits this observation is $O(t^3 \times |I| + s \log |I| + |I| \times |QW|)$, and it produces a solution that is within a factor $(s - |I| - 2t + 1)/(s - |I|)$ of optimal (we will show later that in fact, the complexity of the algorithm is usually much less, especially for the $|I| \times |QW|$ component).

Our algorithm works in two steps. It first finds the optimal way to choose w intervals, $1 \leq w \leq 2t + 1$, for each item (recall that t is the number of parameter values seen, so there are $2t$ possible split points, one before and one after each seen value, and thus a maximum of $2t + 1$ intervals). It then finds the global optimum by choosing $w_1, w_2, ..., w_{|I|}$ such that $w_1 + w_2 + ... + w_{|I|} \leq s$

ItemId	Value	MaxTopKScore
4	5	110
4	5	109
4	5	105
4	5	108.5
4	5	109.75
4	4	108
4	4	106
4	7	102
4	7	104

Figure 7. Example `Culprits` Table

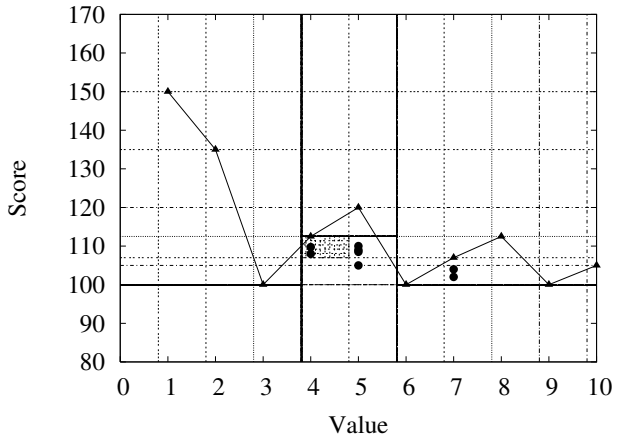

Figure 8. Culprits for `ItemId` 4, shown divided into regions

and choosing w_i intervals for item i gives us the globally optimal partition. We now describe these two steps in more detail.

3.2.1 Finding Optimal Intervals for a Single Item

The current problem is to find for each item i, the optimal way to choose 1 interval, 2 intervals, ..., $2t+1$ intervals. Here, optimal means minimizing the variable cost vc_i. In order to solve this problem, we first create a `Culprits` table using the query workload. The `Culprits` table has three columns, `ItemId`, `Value` and `MaxTopKScore`, and it contains the following set of rows:

$$\{(\texttt{ItemId}, \texttt{Value}, \texttt{MaxTopKScore}) \in \texttt{Culprits} \mid Q \in QW \wedge \texttt{ItemId} \in culprits(I, P_0, Q) \wedge \texttt{Value} = Q.v \wedge \texttt{MaxTopKScore} = maxscore(I, Q)\}$$

where P_0 is the partition in which each item is assigned the single interval that covers its entire value range. Intuitively, the `Culprits` table has one row for each culprit of each query, and the row contains the `ItemId` of the culprit, the parameter value of the query, and the maximum score of the top-k results of the query. Figure 7 shows an example `Culprits` table for different queries/parameter values for item 4. Note that the culprits table is dependent both on the score of the item for different values (i.e. f_i), as well as the query workload.

Creating the `Culprits` table does not require additional processing; it can be created easily during regular query processing by initially running the algorithm using the P_0 partition, and logging information for each culprit.

Given the `Culprits` table, we can determine the value of vc_i for a given choice of intervals for an item i. As an illustration of how this can be done, consider the item corresponding to `ItemId` 4 in Figures 3 and 4, with f_4 and intervals shown in Figure 5. We can augment this figure by selecting the rows in the `Culprits` table that correspond to `ItemId` 4, and plotting each of these rows as a point on the figure where the x-coordinate of a row is its `Value` and the y-coordinate is `MaxTopKScore`. Each of these points represents a potential culprit. Figure 8 shows Figure 5 augmented by plotting the points for `ItemId` 4 from the `Culprits` table (the scale on the x-axis has been altered slightly so that the points can be seen more clearly; we have also shown the graph divided into cells, which we will use later.). Now, suppose that item 4 is broken into intervals $[1,3], [4,5], [6,\infty]$. (We represent this in Figure 8 by drawing darkened vertical lines just before $x = 4$ and $x = 6$.) For each interval, we can draw a darkened line that represents the minimum score of f_4 in that interval. For example, for the interval $[6,\infty]$, the minimum score line (MSL) is drawn at a value of 100. In this case, exactly two points (i.e. potential culprits) fall between that line and the function graph in Figure 8. For the interval $[4,5]$, the MSL is drawn at a value of 112, and we see all seven points (i.e. potential culprits) lie below this line. Finally, the MSL for $[1,3]$ occurs at score 100, and no points lie above it. In general, the total number of points that appear above these MSLs is exactly the value of vc_i. The intuition behind this reasoning is that if we choose a particular set of intervals for an item i, then i can only be a culprit for a query Q if the minimum score of the relevant interval of i is less than the max top-k score of Q (otherwise, i would be pruned by the algorithm before it is processed). Consequently, only the points above the MSL for an interval contribute to vc_i. So in this particular case, $vc_4 = 2$. Had we used just the single interval $[1,\infty]$, then all potential culprits would have contributed, making vc_4 rise to 9.

Recall that our goal is to minimize the value of vc_i for a given number of intervals w. Thus, in pictorial terms, we want to choose w intervals such that the number of points above the MSLs is minimized. Since it is convenient to think of this problem as a maximization problem, we can equivalently view the problem as *maximizing* the number of points *below* the MSLs. Thus, we can define the benefit for each interval to simply be the number of points below its MSL, and our goal then is to find a set of intervals such that the total benefit is maximized. More formally, for interval $Ival$ of item i, we can define its benefit to be:

$$\begin{aligned}&\textsc{Benefit}_i(Ival)\\ &= |\{(\texttt{ItemId}, \texttt{Value}, \texttt{MaxTopKScore}) \in \texttt{Culprits} \mid \\ &\quad \texttt{ItemId} = i.id \wedge \texttt{MaxTopKScore} < \min_{v \in Ival} f_i(v)\}|\end{aligned}$$

and the best benefit for item i broken into w intervals:

$$\textsc{BestBenefit}_i(w) = \max_{\mathcal{P}:|\mathcal{P}(i)|=w} \sum_{Ival \in \mathcal{P}(i)} \textsc{Benefit}_i(Ival).$$

For instance, in our running example, $\textsc{Benefit}_4([4,5]) = 7$, while $\textsc{Benefit}_4([3,5]) = 0$ since including 3 in the interval drops $minf_4$ to 100, below the $maxscore$.

Given the above definitions, we can use a dynamic programming algorithm to find the total benefit for the optimal set of intervals. Algorithm 2 shows the pseudocode. The algorithm is run on each item. The initialization phase first computes the benefit

w	BESTBENEFIT$_4(w)$	best $\mathcal{P}(4)$ with $\lvert\mathcal{P}(4)\rvert = w$
1	0	$\{[1,\infty]\}$
2	0	$\{[1,2],[3,\infty]\}$
3	7	$\{[1,3],[4,5],[6,\infty]\}$
4	7	$\{[1,3],[4,5],[6,6],[7,\infty]\}$
5	9	$\{[1,3],[4,5],[6,6],[7,8],[9,\infty]\}$
6	9	$\{[1,2],[3,3],[4,5],[6,6],[7,8],[9,\infty]\}$

Figure 9. BESTBENEFIT$_4()$ **for item 4**

for every interval. Then, for each point between 1 and $2t+1$, the algorithm computes the best number of intervals generated up to that point. This is done in line 5, by finding the maximum benefit of a choice of intervals for that point.

Algorithm 2 Interval Generation Algorithm

Require: Intervals $\{Ival_{jk}\}$ for item i.
1: Initialize $\mathrm{B}(Ival_{jk}) = \text{BENEFIT}_i(Ival_{jk})$ for $j,k = 1,2,...,2t{+}1$.
2: Initialize $arr_j[1] = \mathrm{B}(Ival_{1j})$ for $j = 1,2,\ldots,2t+1$.
3: **for** $w = 2$ to $2t+1$ **do**
4: **for** $j = 1$ to $2t+1$ **do**
5: $arr_j[w] = \max_{j'>j}\{arr_{j'-1}[w-1] + \mathrm{B}(Ival_{j'j})\}$
6: **end for**
7: **end for**
8: BESTBENEFIT$_i(w) = arr_1[w]$ for all $w = 1,2,...2t+1$.

We summarize the results of Algorithm 2 on our running example for `ItemId` 4 in Figure 9.

Once the values for BENEFIT$_i(\cdot)$ have been computed, the algorithm takes time $O(t^3 \times |I|)$. However, we still need to address how we actually compute these values.

Note that for each entry in the culprits table, there are at most $O(t^2)$ intervals whose BENEFIT$(\cdot)$ value is affected. Hence, a simple way of implementing the initialization in step 1 is to take a pass through the `Culprits` table, updating BENEFIT$_i(Ival_{jk})$ for each affected $Ival_{jk}$. This takes time $O(t^2 \times |Table|)$, where $|Table|$ is the size of the `Culprits` table. However, we can do much better than this naive implementation, as we now describe.

Fix an item i. Roughly speaking, we divide the space of Values $\times$ Scores into $O(t^3)$ rectangular pieces (shown, e.g., in Figure 8). The top and bottom of each rectangle correspond to MSLs of two intervals associated with item i, while the left and right side correspond to vertical lines drawn though two consecutive split points. We will aggregate the number of culprits in each of these cells.

Somewhat more precisely, define $\mathrm{MSL}_i(Ival) = \min_{v \in Ival} f_i(v)$, and label the $\binom{2t+1}{2}$ intervals associated with item i by $Ival^1, Ival^2, ...,$ so that $\mathrm{MSL}_i(Ival^j) \le \mathrm{MSL}_i(Ival^{j'})$ for all $j < j'$. (For notational convenience, we will also define $\mathrm{MSL}_i(Ival^0) = -\infty$.) Let s_ℓ denote the ℓ-th split point for item i, and define region $R^i_{j\ell}$ as follows, for $j = 1,...\binom{2t+1}{2}$, $\ell = 1,2,...,2t$:

$$\begin{aligned} R^i_{j\ell} \;=\; \{&(value, score) \mid \\ &value \in [s_\ell, s_{\ell+1}) \\ &\wedge\, score \in [\mathrm{MSL}_i(Ival^{j-1}), \mathrm{MSL}_i(Ival^j))\}. \end{aligned}$$

(In Figure 8, region $R^4_{6,4}$ has been highlighted.) Rather than processing each entry of the `Culprits` table individually, we first aggregate over each of these regions. Specifically, we compute $c^i_{j\ell} = |\{(\texttt{ItemId}, \texttt{Value}, \texttt{MaxTopKScore}) \in \texttt{Culprits} \mid \texttt{ItemId} = i.id, (\texttt{Value}, \texttt{MaxTopKScore}) \in R^i_{j\ell}\}|$, for each j, ℓ, and item i. (As we can see in Figure 8, $R^4_{6,4}$ contains 2 entries from the culprits table, hence $c^4_{6,4} = 2$.) Note that we can calculate these $c^i_{j\ell}$ in one pass through the `Culprits` table. Once these values of $c^i_{j\ell}$ are known, we can calculate the values of the BENEFIT$_i(\cdot)$ in $O(t^3)$ time for each item i. The algorithm for calculating BENEFIT$_i(\cdot)$ for a fixed item i is given below, labeled as Algorithm 3.

Algorithm 3 Benefit Calculation Algorithm

Require: Item i and the values of $c^i_{j\ell}$ as defined above
1: **for** $\ell = 1$ to $2t$ **do**
2: Reset $b := 0$.
3: **for** $j = 1$ to $\binom{2t+1}{2}$ **do**
4: Update $b := b + c^i_{j\ell}$.
5: **if** interval $Ival^j$ contains the interval $[s_\ell, s_{\ell+1})$ **then**
6: Increment BENEFIT$_i(Ival^j)$ by b.
7: **end if**
8: **end for**
9: **end for**

The time to calculate the $c^i_{j\ell}$ is $O(|Table| \times \log t)$; for each entry in the `Culprits` table, we must find in which region $R^i_{j\ell}$ it belongs, taking $O(\log t)$ time per entry. Once we have the $c^i_{j\ell}$, we spend an additional $O(t^3 \times |I|)$ time to calculate the values for BENEFIT$_i(\cdot)$. Finally, as we mentioned earlier, the dynamic programming portion runs in time $O(t^3 \times |I|)$. Hence, the total running time for this piece is $O(t^3 \times |I| + |Table| \times \log t)$.

3.2.2 *Combining Single Item Intervals*

In the previous subsection, we saw how to break the interval of a given item into w pieces in such a way that we maximized the number of avoided culprits, for any given w. For the i-th item, we denoted this number by BESTBENEFIT$_i(w)$. Recalling that we have a storage constraint limiting us to use at most s items, we must find $w_1 + w_2 + \ldots + w_{|I|} \le s$ such that BESTBENEFIT$_1(w_1) + \ldots +$ BESTBENEFIT$_{|I|}(w_{|I|})$ is as large as possible.

Throughout, we assume that each item will be broken into at most $2t+1$ pieces. For each i and j, we keep track of the incremental improvement of using $j+1$ intervals to describe the i-th item, instead of just j. We use c_{ij} to denote that improvement.

$$c_{ij} = \text{BESTBENEFIT}_i(j+1) - \text{BESTBENEFIT}_i(j).$$

Notice that $\sum_{j=1}^{k} c_{ij} = \text{BESTBENEFIT}_i(k{+}1)$ since the sum telescopes. Thus, our problem is equivalent to finding $k_1 + \ldots + k_{|I|} \le s_{\text{diff}}$ such that $\sum_{i=1}^{|I|}\sum_{j=1}^{k_i} c_{ij}$ is maximized. (For readability, we define $s_{\text{diff}} = s - |I|$.)

For the sake of intuition, consider the following rephrasing of the problem: We are given $|I|$ decks of cards. Each card has some nonnegative number written on it, and we can see all of these numbers. (The jth card in the ith deck has the number c_{ij}.) We must choose s_{diff} cards in total from our decks so that the sum of the numbers is maximized, subject to the constraint that if we choose the jth card from some deck, then we must also choose the 1st, 2nd, ..., $(j-1)$th cards from that deck as well.

To continue the running example, the table in Figure 10(a) contains several items and their interval benefits. The item with `ItemId` 4, for example, contains the sequence 0, 7, 0, 2, indicating that using two intervals gives no benefit over using one, while using three intervals gives a benefit of 7 over using two intervals. Using four intervals gives a no benefit over using three intervals,

and using five intervals gives a benefit of 2 over using four intervals. (That is, $c_{41} = 0, c_{42} = 7, c_{43} = 0, c_{44} = 2$.) These values, of course, follow directly from Figure 9, the output of Algorithm 2 for `ItemId` 4. (The other values for c_{ij} given in Figure 10 are not derivable from the information we have seen previously in the running example.) For simplicity, we will assume that there are only five items in I.

3.2.2.1 *Exact Algorithm for Combining Intervals.*

There is a dynamic programming algorithm to solve this problem exactly. Continuing our running example with $s_{\text{diff}} = 5$, the exact algorithm would take $1, 8$ from the item with `ItemId` = 6; it would take the $8, 4$ from item 7; and it would take the 3 from item 8. Thus, the total benefit is 24, and the algorithm indicates that item 4 should be described with just one interval, items 6 and 7 using three intervals, and item 8 using two intervals.

Although the dynamic programming algorithm works in polynomial time, the approach takes $O(s_{\text{diff}} \times |I|)$ time just to execute its outer loop. Since s_{diff} and $|I|$ are both very large, this approach is impractical, even in our off-line setting.

However, we note that if $c_{ij} \geq c_{ij'}$ for all i and all $j < j'$, the exact solution can be found very efficiently using the greedy algorithm: Simply find the s_{diff} largest c_{ij}, where we break ties in favor of the smaller j. In this case, we thus have an efficient algorithm.

LEMMA 2. *Let $t > 0$, I,s and $s_{diff} = s - |I|$ be as above, and let c_{ij} be nonnegative numbers, as above, for $i = 1, 2, ..., |I|$, and $j = 1, ..., t$, with $c_{ij} \geq c_{ij'}$ for all i and $j < j'$. Then there is an algorithm running in $O(s \log |I|)$ time that finds nonnegative integers $k_1, ..., k_{|I|}$ that maximize*

$$\sum_{i=1}^{|I|} \sum_{j=1}^{k_i} c_{ij}$$

subject to $k_1 + \ldots + k_{|I|} \leq s_{diff}$.

PROOF. It is helpful to think of this algorithm using our "decks of cards" formulation. Recall that we have $|I|$ decks of cards, each with a top card (i.e. the 1st card of the deck). From these $|I|$ top cards, take the one with the largest value. (Hence, if we remove the top card from the jth deck, then the 2nd card is now the top card of the deck.) Simply repeat this until we have taken s_{diff} cards.

Clearly, this algorithm works in the specified time: we simply need to maintain a heap-like datastructure to find the largest-valued top card at each step, taking $O(\log |I|)$ time. Since we repeat the step s_{diff} times, this is $O(s_{\text{diff}} \log |I|)$ time spent iterating. Counting the set-up time for the heap of $O(|I| \log |I|)$, this is a total time of $O(s \log |I|)$. However, we still need to argue that the algorithm produces the maximum value.

But notice that by the assumption that $c_{ij} \geq c_{ij'}$ for all $j < j'$, we have that the largest remaining card is the top card for some deck. So the algorithm described above actually finds the s_{diff} cards with the largest values; hence we must have the cards with the largest total sum. □

3.2.2.2 *Smoothing.*

Unfortunately, the c_{ij}s will not be decreasing in general. In fact, the table in Figure 10(a) produced from Figure 8 reflects this. More concretely, consider the example with `ItemId` $= 4$ in Figure 8, ignoring the intervals shown. If we want to split this item into two intervals, then no choice of an interval split point would avoid any culprits (because queries are only for values 4, 5, and 7, and splitting on either side of these values offers no benefit because the MSLs of the resulting intervals will still be at 100). Thus, $c_{4,1} = 0$ in this case. However, if we decide to split the item into three intervals, then we can split it into the intervals shown in Figure 8, and this would avoid 7 culprits. Thus, $c_{42} = 7 > c_{41}$.

So in general, it is not the case that $c_{ij} \geq c_{ij'}$ for all i and $j < j'$. To address this, we "smooth" the c_{ij} to produce c'_{ij} such that $c'_{ij} \geq c'_{ij'}$ for all i and $j < j'$, along with other important properties. This smoothing technique corresponds to a "convex hull" approach that was first discovered by Dyer [10] and, independently, by Zemel [22] in the context of the multiple-choice knapsack problem. Although the majority of work on the multiple-choice knapsack problem has focused on strong theoretical guarantees, several papers [16, 17, 1] have produced much simpler algorithms with incredibly strong guarantees in practice. While previous work focused on producing a $(1 + \epsilon)$ approximation, giving an algorithm whose running time depended on ϵ, the work of [16, 17, 1] gives an an algorithm with no such dependence, and whose solution in our context is at least $(s_{\text{diff}} - 2t + 1)/s_{\text{diff}}$ times as good as optimal. Since we expect s_{diff} to be thousands of times larger than t in practice, this shows that the approximate solution is better than 99.9% of optimal. For the sake of completeness, we describe the smoothing algorithm below, and give a simple example to illustrate the main ideas.

For readability, we define the notation $\text{AVG}(c_{ij}, \ldots, c_{ik}) = \frac{1}{k-j+1} \sum_{l=j}^{k} c_{il}$

Algorithm 4 Smoothing Algorithm[16, 17, 1]

Require: $c_{i1}, c_{i2}, \ldots, c_{it}$.
1: Initialize $c_{i,t+1} \leftarrow -\infty$, $j \leftarrow 1$.
2: **while** $j \leq t$ **do**
3: Let $k = \text{argmax}_{k' \geq j} \text{AVG}(c_{ij}, \ldots, c_{ik'})$.
4: Assign $c'_{i\ell} \leftarrow \text{AVG}(c_{ij}, ..., c_{ik})$ for all $\ell \in [j, k]$.
5: Update $j \leftarrow k + 1$.
6: **end while**

Essentially, the algorithm starts at a c_{ij} and looks ahead to see if there is any subsequent $c_{ij'}$ that can increase the average value of all intermediate $c_{ik}, j \leq k < j'$. As can be seen, this algorithm has complexity $O(t^2)$. Although there is a somewhat faster algorithm running in time $O(t \log t)$, improvements here do not significantly improve the overall running time of the entire algorithm.

As an illustration of the smoothing technique, consider again the item with `ItemId` 4 in Figure 10(a). Intuitively, we would like to get the 7; however, we need to use the 0 first. So we replace the $0, 7$ with two copies of their average: $3.5, 3.5$. Notice that taking 0, then 7, is helpful exactly when taking 3.5 followed by 3.5 is helpful. Continuing, we replace the $0, 2$ with two copies of their average: $1, 1$. In general, we find the prefix sequence with the largest average; this may simply be the first item of the sequence. We then replace each of those values with the average, and recursively iterate on the remaining sequence. We repeat this smoothing for items 6, 7, and 8. The smoothed values are shown in Figure 10(b).

With the smoothed values c'_{ij} in hand, we simply find the s_{diff} largest values, as described in Lemma 2. As we noted in the lemma, this can be done in $O(s \log |I|)$ time.

To illustrate, consider our example, again with $s_{\text{diff}} = 5$. The

ItemID	c_{ij}
4	$0, 7, 0, 2$
6	$1, 8, 0, 6$
7	$8, 4, 0, 2$
8	$3, 0, 0, 6$

(a)

ItemID	c'_{ij}
4	$3.5, 3.5, 1, 1$
6	$4.5, 4.5, 3, 3$
7	$8, 4, 1, 1$
8	$3, 2, 2, 2$

(b)

Figure 10. Example Benefits and Smoothing

smoothed values we would extract are 8, 4.5, 4.5, 4, 3.5, corresponding to the original values 8, 1, 8, 4, 0. Notice that the smoothed values $4.5 + 4.5$ exactly equal the original values $1 + 8$. However, the last smoothed value we extracted, 3.5, corresponds to 0. (Were we able to take another value, in this case 3.5, then we would again see that the smoothed values $3.5 + 3.5$ exactly equal the original values, $0+7$.) In general, at most the last $2t-1$ values (which all come from the same item) will be overestimates of the original values. Thus, when translating the c'_{ij} back to the original c_{ij}, the total benefit we obtain using these smoothed values is at least $(s_{\text{diff}} - 2t + 1)/s_{\text{diff}}$ of optimal. So we have the following theorem, first proven in [16].

THEOREM 1 ([16, 17]). *Let I, s, and $\text{BESTBENEFIT}_i(\cdot)$ for $i = 1, ..., |I|$ be defined as in the previous subsection. Then there is an algorithm running in time $O(|I|t^2 + s \log |I|)$ that finds $w_1, ..., w_{|I|}$ such that the value of*

$$\text{BESTBENEFIT}_1(w_1) + \ldots + \text{BESTBENEFIT}_{|I|}(w_{|I|})$$

subject to $w_1 + w_2 + \ldots + w_{|I|} \leq s$ is at least $(s_{diff} - 2t + 1)/s_{diff}$ of optimal.

3.2.3 Overall Complexity

As we already noted, processing the query workload takes time at most $O(|I| \times |QW|)$, although this is actually the size of the log, which will usually be much smaller. The running time to find optimal partitions for each item takes a total of $O(t^3 \times |I|)$ over all items. (We ignore the cost of processing the `Culprits` table, since it is subsumed in the processing time of the query workload.) The running time for finding a nearly optimal combination of intervals across times is $O(s \log |I|)$, and smoothing takes $O(t^2 \times |I|)$. Hence, the total complexity is $O(t^3 \times |I| + s \log |I| + |I| \times |QW|)$.

4. Experiments

We considered four approaches in our experimental evaluation: (1) `NAIVE`: the naive approach of looping over all the selected items, (2) `PRECOMPUTE`: the precomputation approach described in the introduction, (3) `PI` (for Precompute Interval): the proposed approach, and (4) `PS` (for Precompute Single): the proposed approach, but using only a single interval per item. These approaches were implemented using main-memory tables and indices and were evaluated using various data sets, promotional rules, and query workloads. The main finding from our experiments are the following:

- `NAIVE` performs at least an order of magnitude worse in terms of response time than `PI` and `PS`.
- `PRECOMPUTE` requires many orders of magnitude more space than `PI` or `PS`.
- `PI` outperforms `PS` by a factor of 2-6 using just 50% more space, illustrating the effectiveness of our interval selection algorithm.
- `PI` scales much better than `PS`, i.e., the performance gap between the two approaches widens with increasing data sizes.
- The backend processing time, i.e., the time needed by our interval-generation algorithm to generate optimal intervals given a space budget, is acceptably low and increases only roughly linearly with increasing size of the `Culprits` table.

4.1 Experimental Setup

We ran our experiments on an Intel machine with 2GB RAM. We used a real data set from Yahoo! Shopping, which has hundreds of thousands of items for a given category (laptops) due to products listed from many different vendors. We built regular main-memory indices on 5 searchable attributes of the laptops table to evaluate query predicates. The size of the relation was varied from $100K$ to $1M$ rows with a default value set to $200K$. Each run represents the running time of 1000 queries.

4.1.1 Rules and Queries Generation

The promotional rules and queries were synthetically generated by using the parameters in Figure 11 (the default value for each parameter is shown in bold).

Our rule generation tries to simulate reality as closely as possible and takes two parameters: the maximum number of rules per item, and the maximum discount given by any rule. We generated three different kinds of rules:

(a) $R_1(q, d)$: Buy some quantity $> q$, get $d\%$ off.

(b) $R_2(q, d)$: Buy some quantity $> q$, get $\$d$ off.

(c) $R_3(q_1, q_2)$: Buy quantity q_1, get quantity q_2 free.

To generate rules, we first choose whether an item gives a discount or not. As in the real world, the probability that an item gives a discount increases with increasing price of the item. Once an item is chosen to give a discount, we assign it a number of rules chosen at random between 1 and the maximum number of rules per item. Each rule is then chosen at random from among the three types. The quantity threshold at which the rule starts to apply is also chosen according to price, with more expensive items offering discounts even for low quantities while cheap ones offering discounts only for high quantities. The amount of discount given by a rule was chosen based on both the price and the quantity threshold: cheaper items gave less discount than expensive ones, and the amount of discount given was higher for higher quantities.

Query generation takes two parameters: the maximum quantity being queried, and the number of results requested (k). The quantity queried was chosen uniformly at random between 1 and the maximum quantity. Besides, the query had predicates on the laptops table that were chosen at random to ensure that our workload consisted of queries with varying selectivities.

4.2 Space Overhead of `PRECOMPUTE`

To investigate the space overhead, we used 1 million listings (which is only a small subset of all Yahoo! Shopping listings) and 100 quantities. Using 4 bytes per score, the space overhead

Parameter	Values (default in bold)
Number of items	$[100K - 1M]$ - **200K**
Interval budget	$[100\% - 500\%]$ - **200**% of number of items
Promotional rules	Number of rules/laptop: $[1 - 5]$ - **3** Maximum Discount: $[10\% - 60\%]$ - **50**%
Queries	Maximum quantity: $[1 - 100]$ - **50** Results required (k): $[1 - 100]$ - **10**

Figure 11. Experimental Parameters

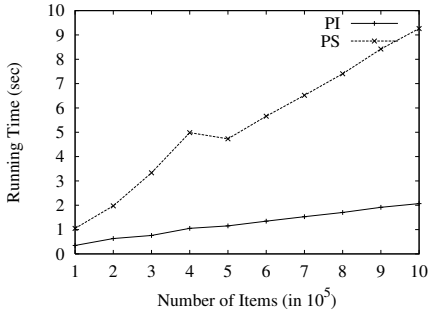

Figure 12. Varying Data Size

for PRECOMPUTE is 2.4GB (1M * 100 * 4 = 400MB for the Intervals table index, and 400MB each for the 5 searchable Yahoo! Shopping attribute indices corresponding to each item). In contrast, the PI method only required 40MB of space (as shown in the next section), including the space for the regular indices. Consequently, PRECOMPUTE is not a feasible solution to our problem, and we do not consider it further.

We now describe the impact of the parameters in Figure 11 on the *query response time* (Section 4.3). We also study the processing time of our backend (i.e., interval generation algorithm) as a function of the size of the Culprits table (Section 4.4).

4.3 Response Time Experiments

We report the performance of the PS and PI approaches. We do not report the performance of NAIVE as it is always an order of magnitude worse than PS and PI.

4.3.1 Varying Data Size

Figure 12 shows the effect of varying the data set size on the response time. The performance gap between PS and PI increases with increasing data size because the number of culprits increases correspondingly, and PI is able to avoid these culprits by an intelligent choice of intervals.

4.3.2 Varying Space Budget for Intervals

Figure 13 reports the response time of PI and PS with increasing space budget to store intervals. Since PS uses only one interval per item, the performance of PS remains constant with the space budget. The response time of PI however improves drastically as soon as it is given some extra space budget to store intervals. In fact, with a space budget which is only 1.5 times the size of the laptops relation, PI outperforms PS by a factor of 5. Thus, much of the performance benefit of PI can be obtained with very little space overhead. remains constant. The performance of PI, however, improves. Note that for a space budget factor of 1, PI performs slightly worse than PS because it has the overhead of indexing general intervals.

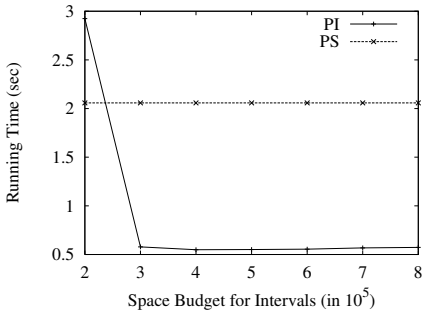

Figure 13. Varying Space Budget for Intervals

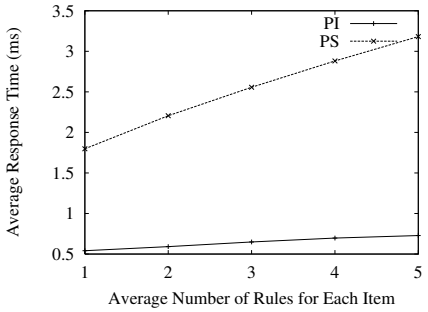

Figure 14. Varying Number of Rules per Item

4.3.3 Varying Promotional Rules

Figure 14 reports the response time of PI and PS as we vary the maximum number of rules per item. PI performs substantially better than PS and as the number of rules is increased, the performance gap between them increases. As more rules are added, the probability that an item gives a high discount increases, and so does the probability of that item being a culprit for a low-quantity query. This increase in the number of culprits slows down PS, but the effect on PI is not substantial as the culprits are avoided by using intervals.

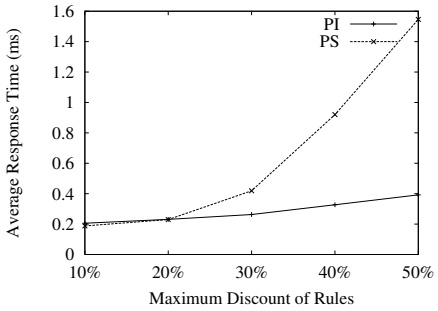

Figure 15. Varying the Maximum Discount

Figure 15 reports the response time of PI and PS as we vary the maximum amount of discount offered by any rule. Here again, the trend is similar to the previous experiment. For low discounts, the probability of an item being a culprit is small, hence the performance of PI and PS is similar. However, with higher discounts, more culprits are generated and the performance of PS severely

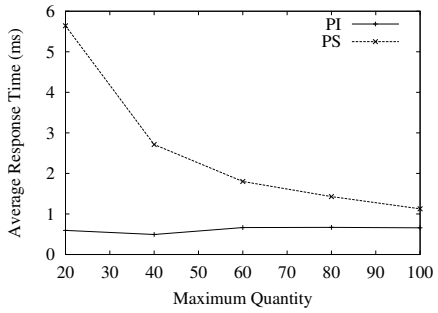

Figure 16. Varying Maximum Quantity Value

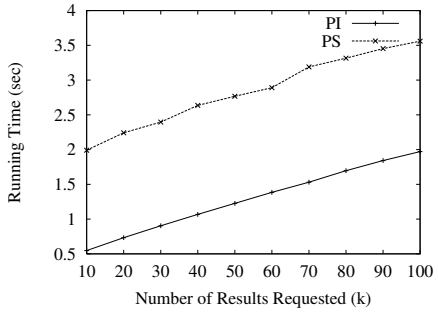

Figure 17. Varying Number of Results

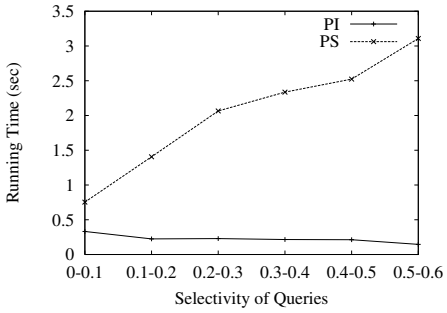

Figure 18. Varying Query Selectivity

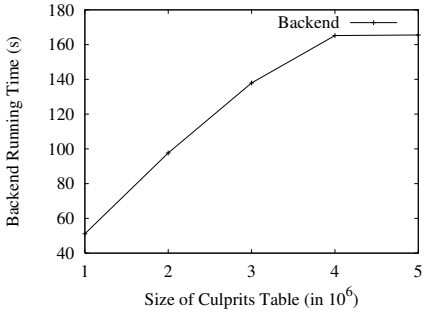

Figure 19. Backend Processing Time

worsens, but that of PI only degrades slightly.

4.3.4 *Varying Query Parameters*

Figure 16 reports the response time of PS and PI as we varied the maximum quantity being queried in the workload. We observe that while PI outperforms PS, the difference between their performance decreases as higher quantities are queried: with a higher quantity value in the query, more rules are likely to apply, thus reducing the number of culprits. Hence the performance of PS improves with increasing quantities. Note that in our workload the quantities are chosen uniformly at random. In practice, we expect a query distribution that is very heavily biased towards low quantities (which is similar to reducing the maximum quantity), in which case, PI performs 2 to 6 times better than PS.

Figure 17 shows the response time of PI and PS as the number of results requested (k) is varied. With increasing k, more items have to examined in order to return the top k results. Thus, the response time of both PS and PI increases linearly with k. At the same time, the probability of an item being a culprit decreases with increasing k. Thus, the response time of PS (in terms of its ratio to the response time of PI) improves with increasing k. In the limit when k = number of query results, we expect PI and PS to have the same performance.

Figure 18 shows the response time of PS and PI for queries of different selectivity. We grouped queries according to selectivity and measured the average response time in each group. As the number of items satisfying the query increases, the number of culprits also proportionally increases and hence the performance of PS worsens. However, the performance of PI remains constant with an increasing query selectivity.

4.4 Backend Processing Time

In this experiment, we study the time needed by our interval-generation algorithm (Section 3) to generate the optimal intervals given a space budget. Our experiments show that the overall backend processing time increases only roughly linearly with the size of the Culprits table. Even when the number of distinct quantities being queried is 50, and there are as many as 4 million culprits, our interval generation algorithm runs in under 3 minutes. Since the interval-generation algorithm runs in the backend, it does not have any stringent response-time requirements. Hence this overhead is entirely acceptable in return for the huge performance benefits of PI. We also found that the backend processing time is roughly independent of the space budget for intervals.

5. Related work

The problem introduced in this paper bears some resemblance to top-k processing in Information Retrieval (IR) systems using query-dependent scores such as TF-IDF [20]. Just as IR systems build an inverted list of documents for each term, where the list corresponding to a term is ordered by the TF score corresponding to that term, we could build a list of items for each parameter value, where each list is ordered by the score of item for that parameter value. While this technique works for IR systems because the inverted list is usually sparse (a term does not usually occur in all documents), it can be very large in our context because each item has a score for every/most parameter values (e.g., every product has a price for every quantity). In fact, this approach directly corresponds to the naive pre-computation strategy, which was shown to be infeasible in our experimental evaluation.

The most prominent family of top-k algorithms is the one by

Fagin et al [12, 13] over multimedia repositories. These algorithms use equi-joins to evaluate top-k answers over multiple sources and assume monotonic score aggregation functions. In relational databases, existing work has focused on extending the evaluation of SQL queries for top-k processing when scoring is achieved through a SQL order-by clause [4, 6, 8]. For example, Carey and Kossmann [4] study how to optimize such queries by limiting the cardinality of intermediate result tuples while Ilyas et al [18] propose a score-aware relational algebra and show how a scoring function can be pushed in a query plan in order to achieve early pruning. The use of a query plan to produce scored results has a dynamic aspect where scores depend on the algebraic operators. In our work, scoring functions are black boxes and we focus on pre-computing indices for the efficient evaluation of selection queries.

Other works [3, 5, 14] use statistical information to map top-k queries into selection predicates which may require restarting query evaluation when the number of answers is less than k. The key difference with our work is that the score of an item is not static and is not assumed to be monotonic. Hence, we index our functions in a way that (i) optimizes finding applicable functions per item by computing appropriate parameter value intervals and (ii) identifies the items to prune by associating the lowest score to each interval.

We could model item-specific functions such as promotional rules as queries in publish/subscribe systems [7, 11] or triggers in rule-based databases [21]. While this would address the efficient evaluation of a large number of functions, these systems are not designed to process top-k answers. To the best of our knowledge, our solution is the first that could be considered as an extension to these systems by introducing scoring and top-k pruning.

One of the main contributions of this paper is showing that the problem of computing the optimal set of intervals for a set of items can be decomposed into a set of small problems for each item, which can then be solved independently and combined together efficiently. In order to solve the interval problem for each item, we use a standard dynamic programming algorithm (Section 3.2.1), which has also been used in other contexts [15].

6. Conclusion

We have introduced a new class of top-k queries over query-dependent functions that arise in a variety of applications ranging from online shopping to online maps to predictive financial modeling. We have developed efficient algorithms for evaluating such queries using function intervals, and our experimental evaluations have shown that the proposed approach is efficient both in terms of performance and in terms of space.

There are several avenues for future work. One possible extension is to deal with functions over multiple parameters and items, e.g., travel packages such as “Stay 3 nights, get a 10% discount on Casino tokens.” Another possible extension is incorporating suggestions based on function values, e.g., “Leave one hour later and save 30 minutes in traffic.” Both of these extensions require enhancing the model with crossitem queries and multi-dimensional indices on functions.

7. References

[1] Md. M. Akbar, M. Sohel Rahman, M. Kaykobad, E. G. Manning, G. C. Shoja. Solving the Multidimensional Multiple-Choice Knapsack Problem by Constructing Convex Hulls. Comput. Oper. Res., 33(5), 2006.

[2] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. Commun. ACM 18(9): 509-517(1975).

[3] N. Bruno, S. Chaudhuri, L. Gravano. Top-K Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation. ACM TODS, 27(2), 2002.

[4] M. J. Carey, D. Kossmann. On Saying ”Enough Already!” in SQL. SIGMOD 1997.

[5] C. Chen, Y. Ling. A Sampling-Based Estimator for Top-K Query. ICDE 2002.

[6] F. C. Chu, J. Y. Halpern, P. Seshadri. Least Expected Cost Query Optimization: An Exercise in Utility. PODS 1999.

[7] Y. Diao, P. M. Fischer, M. J. Franklin, R. To. YFilter: Efficient and Scalable Filtering of XML Documents. ICDE 2002.

[8] D. Donjerkovic, R. Ramakrishnan. Probabilistic Optimization of Top N Queries. VLDB 1999.

[9] K. Dudziński, S. Walukiewicz. Exact Methods for the Knapsack Problem and its Generalizations. European Journal of Operational Research, 1987.

[10] E. Dyer. An $O(n)$ Algorithm for the Multiple-Choice Knapsack Linear Program. Mathematical Programming, 1984.

[11] F. Fabret, H. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. SIGMOD 2001.

[12] R. Fagin, A. Lotem, M. Naor. Optimal Aggregation Algorithms for Middleware. PODS 2001.

[13] R. Fagin. Combining Fuzzy Information from Multiple Systems. PODS 1996.

[14] V. Hristidis, N. Koudas, Y. Papakonstantinou. PREFER: A system for the Efficient Execution Of Multiparametric Ranked Queries. SIGMOD 2001.

[15] Y. E. Ioannidis, V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. SIGMOD 1995.

[16] C. Lee and D. Siewiorek. An Approach for Quality of Service Management. Technical Report CMU-CS-98-165, Computer Science Department, CMU, 1998.

[17] C. Lee, J. Lehoczky, R. Rajkumar, Y. P. Siewiorek. On Quality of Service Optimization with Discrete QoS Options. IEEE Real Time Technology and Applications Symp., 1999.

[18] C. Li, K. Chen-Chuan Chang, I. F. Ilyas, S. Song. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. SIGMOD 2005.

[19] D. Pisinger. Algorithms for Knapsack Problems. Ph.D. Thesis, 1995.

[20] G. Salton and M. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, 1983.

[21] J. Widom, S. Ceri. Active Database Systems: Triggers and Rules For Advanced Database Processing. Morgan Kaufmann 1996

[22] E. Zemel. An $O(n)$ Algorithm for the Linear Multiple Choice Knapsack Problem and Related Problems. Information Processing Letters, 1984.