Efficient Main-Memory Top-K Selection For Multicore Architectures

Vasileios Zois
University of California, Riverside
vzois001@ucr.edu

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

Walid A. Najjar
University of California, Riverside
najjar@cs.ucr.edu

ABSTRACT

Efficient Top-k query evaluation relies on practices that utilize auxiliary data structures to enable early termination. Such techniques were designed to trade-off complex work in the buffer pool against costly access to disk-resident data. Parallel in-memory Top-k selection with support for early termination presents a novel challenge because computation shifts higher up in the memory hierarchy. In this environment, data scan methods using SIMD instructions and multithreading perform well despite requiring evaluation of the complete dataset. Early termination schemes that favor simplicity require random access to resolve score ambiguity while those optimized for sequential access incur too many object evaluations. In this work, we introduce the concept of *rank uncertainty*, a measure of work efficiency that enables classifying existing solutions according to their potential for efficient parallel in-memory Top-k selection. We identify data reordering and layering strategies as those having the highest potential and provide practical guidelines on how to adapt them for parallel in-memory execution (creating the VTA and SLA approaches). In addition, we show that the number of object evaluations can be further decreased by combining data reordering with angle space partitioning (introducing PTA). Our extensive experimental evaluation on varying query parameters using both synthetic and real data, showcase that PTA exhibits between 2 and 4 orders of magnitude better query latency, and throughput when compared to prior work and our optimized algorithmic variants (i.e. VTA, SLA).

PVLDB Reference Format:
Vasileios Zois, Vassilis J. Tsotras and Walid A. Najjar. Efficient Main-Memory Top-K Selection For Multicore Architectures. *PVLDB*, 13(2): 114-127, 2019.
DOI: https://doi.org/10.14778/3364324.3364327

1. INTRODUCTION

Top-k queries retrieve the k highest ranking objects as determined through a user-defined monotone function. Many variations of this problem exist, including but not limited to, simple selection [15, 29, 18, 22], Top-k group-by aggregation [28], Top-k join [31, 23], as well as Top-k query variations on different applications (spatial, spatiotemporal, web search, etc.) [7, 11, 8, 26, 1, 5]. All variants concentrate on minimizing the number of object evaluations while maintaining efficient data access. In relational database management systems (RDBMS), the most prominent solutions for Top-k selection fit into three categories: (i) using sorted-lists [15], (ii) utilizing materialized views [22], and (iii) applying layered-based methods (convex-hull [6], skyline [27]).

Efficient Top-k query processing entails minimizing object evaluations through candidate pruning [29] and early termination [15, 2]. This can be achieved using intricate indexing techniques and auxiliary information which are intended to efficiently guide processing [18, 27, 3] and reduce the candidate maintenance cost [29]. Such approaches were developed primarily to enable efficient processing on disk-resident data, addressing issues related to main memory buffering and batch I/O operations. The premise was that using the main memory buffer pool to store and operate on auxiliary information is less costly than performing a full data scan.

The decrease in DRAM cost, coupled with higher capacity and bandwidth guarantees, motivated the development of in-memory database systems that leverage multicore processing to maximize throughput and reduce latency. Efficient parallel in-memory Top-k selection should favor low number of object evaluations while avoiding complex strategies used to enable early termination. This is crucial for main memory query evaluation because complicated processing methods translate to excessive number of memory accesses which count against query latency. In the same context, the wide availability of SIMD instructions and multithreading make data scan solutions strong contenders for high performance Top-k selection.

Enabling low cost early termination becomes increasingly difficult for a number of reasons. *Firstly*, simple processing strategies often rely on random accesses [15, 2, 3] to resolve score ambiguity, a practice inherently detrimental for high throughput. *Secondly*, techniques favoring sequential access enable such behavior at the expense of more object evaluations [20] or having to maintain too many candidates [29, 27], the end result of which is an increased number of memory accesses.

In this work, we study the related literature in order to discover suitable practices for efficient parallel main memory Top-k selection. In order to identify these methods, we establish a new measure of algorithmic efficiency called *rank uncertainty*. As opposed to the number of object evalua-

Proceedings of the VLDB Endowment, Vol. 13, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3364324.3364327

tions (a measure concentrating on memory accesses related to score aggregation), rank uncertainty considers the proportion of total accesses to that of object evaluation accesses. Using the notion of *rank uncertainty* we empirically quantify the cost of early termination and classify (Figure 2) disk-based related work. This classification indicates that data reordering and layering techniques bear the highest potential for efficient parallel in-memory execution.

We first adapt these practices to create their parallel in-memory variants, thus creating VTA (Vectorized Threshold Algorithm) and SLA (Skyline Layered Algorithm) approaches. VTA uses reordering while SLA applies reordering and layering. Nevertheless, we show experimentally that they incur large number of object evaluations. To overcome this limitation, we introduce PTA (Partitioned Threshold Algorithm) which combines reordering and angle space partitioning. Our contributions are summarized below:

- We introduce (Section 4.2) the notion of *rank uncertainty*, a robust measure of algorithmic efficiency, designed to identify appropriate methods for efficient parallel in-memory Top-k selection.
- We provide practical guidelines geared towards efficient adaptation of reordering (Section 5.2) and data layering (Section 6.1) algorithms in a parallel environment (creating the VTA and SLA approaches).
- We develop a new solution (PTA) that relies on angle space partitioning (Section 6.2) combined with data reordering to improve algorithmic efficiency while also maintaining low *rank uncertainty*.

The paper is organized as follows. Section 2 reviews the related literature and Section 3 presents a formal definition of the Top-k selection problem. In Section 4.1 three parallel Top-k models are presented and Section 4.2 the concept of *rank uncertainty* is described. Sections 5 and 6 propose guidelines for implementing optimized algorithms for scalar, SIMD, and multithreaded execution. Section 7 concludes with extensive experiments and result discussion.

2. RELATED WORK

Solutions that support efficient Top-k query evaluation fall into three categories: (1) List-based methods, (2) View-based methods, (3) Layered-based methods.

2.1 List-Based Methods

Fagin et al [15] formalized the problem of Top-k query evaluation over sorted-lists presenting FA, TA, and NRA. These algorithms access the individual database objects in round robin order dictated by a collection of sorted attribute lists. FA maintains all seen objects until k of them have been detected in all lists, evaluating their scores only after that point. TA evaluates each object as soon as it is seen, terminating execution only after discovering k objects with scores greater or equal than the combined threshold of the associated list level. NRA focuses on enabling sequential access which requires keeping track of the lower and upper bounds for each seen object, terminating only when k objects with lower bounds greater than all objects' upper bounds are discovered.

Stream-Combine (SC) [17] improves NRA using heuristics to choose the most promising list for evaluation. LARA [29] aims at reducing the cost of maintaining the upper bounds for each seen object and improve candidate pruning. IO-Top-k [3] utilizes selectivity estimators and score predictors to efficiently schedule sorted and random accesses. TBB [32] relies on a pruning mechanism and bloom filters to efficiently process Top-k queries over bucketized sorted lists.

BPA [2] improves TA's stopping threshold by considering attributes seen both under sorted and random access. T2S [18] promotes reordering the database objects based on their first seen position in the sorted lists favoring sequential access for disk-resident data. ListMerge [41] relies on intelligent result merging to efficiently evaluate Top-k queries over large number of sorted lists.

2.2 View-Based Methods

PREFER [22] aims at reducing the cost associated with Top-k query processing by effectively managing and updating materialized views in-memory. LPTA [10] employs linear programming to avoid accessing the disk when the combined query attributes appear within overlapping materialized views. LPTA+ [38] aims at reducing the number of solved linear programming problems per query to improve performance. TKAP [19] combines early pruning strategies from list-based methods and materialized views to support Top-k queries on massive data.

2.3 Layered-Based Methods

The Onion technique [6] linearly orders the objects in the database by computing disjoint convex hulls on all attributes. This method offers guarantees which state that the Top-k objects appear within the first k layers (convex hulls). The Dominant Graph (DG) [43, 44] orders objects according to their dominance relationship while utilizing a graph traversal algorithm to evaluate Top-k queries. The partitioned layer algorithm (PLA) [21] relies on convex skyline layering and fixed line partitioning to further improve object pruning. The HL-index [20] is a hybrid method that combines skyline layering and TA ordering within each layer to reduce object evaluations. The Dual Resolution (DL) [27] index suggest relying on skyline layering and the convex skyline properties to improve DG's graph traversal algorithm.

2.4 Parallel In-Memory Top-K Selection

Top-k query processing techniques that support early stopping have focused mainly on disk-resident data. Existing solutions for in-memory processing reduce the problem of query evaluation to that of list intersection [36, 35, 25, 40, 13], while other methods avoid reordering the dataset and try to maximize skipping irrelevant objects during evaluation [14, 16, 12]. These optimizations are contingent on the attribute lists having different sizes. This may not be a reasonable assumption for a DBMS environment and is heavily dependent on the application (e.g. text mining). In this paper, we consider a setting, in which all objects/tuples have a value for each attribute (even if that value is close to zero). Our goal is to incorporate an early stopping mechanism that strikes a balance between algorithmic efficiency and the ability to support vectorization, parallel execution, and high memory bandwidth utilization.

3. PROBLEM DEFINITION

Let R be a relation consisting of n objects/tuples, each one having d attributes/scores ($o = \{a_0, a_1, ...a_{d-1}\}$) ranging in $(0, 1]$ without loss of generality. Equivalently, R can

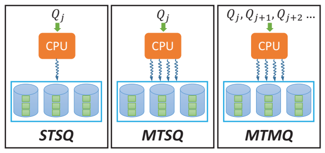


Figure 1: Parallel Top-k evaluation models

be thought of as a set of multidimensional points assigned in euclidean space. A user-defined scoring function $F(o)$ maps the objects in R to values in the range $(-\infty, \infty)$. A Top-k query retrieves the k objects having the k highest (or lowest) score under F. For the rest of this paper it is assumed that we are searching for the highest ranked objects. Hence, our goal is to discover a collection S of objects $[o_1, o_2, o_3 ... o_k]$ such that $\forall j \in [1, k]$ and $\forall o_i \in (R - S)$, $F(o_j) \geq F(o_i)$.

In related work the user-defined function has been either linear [6, 10, 22] or monotone [10, 22, 34, 39]. We formally define an arbitrary linear function as follows:

$$F(o) = \sum_{i=0}^{m} (w_i \cdot a_i) \tag{1}$$

An arbitrary ranking function F is identified through a unique declaration of weights which refer to a specific subset of the corresponding relational attributes. These weights, denoted with w_i, constitute the *preference* vector of a given Top-k query.

A *monotone* scoring function satisfies that:

$$\begin{aligned} &if\ o_u(a_i) \geq o_v(a_i), \forall i \in [0, d-1] \\ &then\ F(o_u) \geq F(o_v) \end{aligned} \tag{2}$$

This means that any objects having higher values on all attributes should rank higher than those with smaller attributes. This is guaranteed for any linear function when all weights in of the preference vector are non-negative. Nevertheless, our solutions are applicable for both linear and non-linear monotone functions (since they rely on static object re-ordering similar to TA [15] and T2S [18]). The majority of previous work support exclusively monotone functions with the only exceptions being ONION [6] and HL [20] which can also support non-monotone functions.

4. PARALLEL TOP-K QUERIES

In this section, we attempt to identify such practices that provide satisfactory parallelism, and efficient in-memory processing without sacrificing algorithmic efficiency.

4.1 Parallel Execution Models

In the context of in-memory Top-k query evaluation, there are two ways to enable parallelism: (1) utilize SIMD instructions to evaluate multiple objects in parallel, (2) leverage multithreading to either evaluate many queries concurrently or partition the data so as to evaluate a single query in parallel. It is important to note that both of these methods implicitly improve memory bandwidth utilization, as they promote sequential streaming access and memory latency masking by issuing many outstanding memory requests, respectively. In addition, they can be intertwined together to create three separate parallel Top-k query evaluation models as indicated in Fig. 1; this includes: (i) Single Thread Single Query (STSQ), (ii) Multiple Thread Single Query (MTSQ) and (iii) Multiple Thread Multiple Query (MTMQ) (i.e. one thread per query). There is apparent correlation

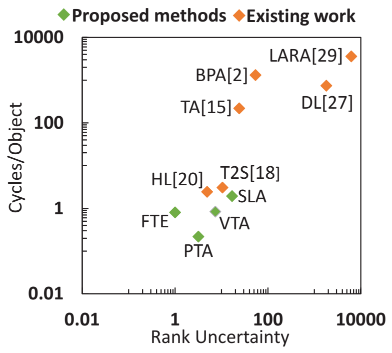


Figure 2: Cycles/Object vs Rank Uncertainty.

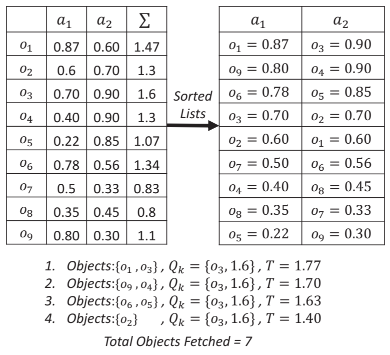
between developing optimal STSQ/MTSQ algorithms and applying them also towards MTMQ processing. For this reason, we focus on developing optimal STSQ and MTSQ methods which are also tested on top of MTMQ environments.

4.2 Rank Uncertainty

Existing Top-k algorithms [15, 3, 29, 2, 20, 18] improve query latency using auxiliary information to guide processing, skip object evaluations through early termination and reduce the candidate maintenance cost. These practices are favorable in systems where the relative data access cost (aka latency gap), as experienced by the CPU, between the primary and secondary storage media is high. For example systems operating on disk-resident data, experience high random access latency gap ($\times$100000) between DRAM (primary) and disk (secondary). Therefore any intricate strategies geared towards skipping object evaluations and enabling early termination are less costly than a direct access to non-essential data from the secondary storage medium. In contrast when data are memory resident, the latency gap between CPU cache and DRAM is much smaller ($\times$30). In that case, complicated pruning and early termination schemes may result in performance degradation as the cost of enabling them cannot be justified solely by less object evaluations. In fact, using a simple streaming solution may prove to be more or equally effective than some over-complicated early termination strategies.

In order to quantify the suitability of previous work, when employed in a parallel main-memory environment, we introduce the concept of *rank uncertainty*. The *rank uncertainty* ($R(A) = \frac{M_T(A)}{M_E(A)}$) of a Top-$k$ algorithm (A) is the ratio of total memory accesses ($M_T(A)$) over memory accesses associated with object score aggregation and ranking ($M_E(A)$). *Rank uncertainty* is a superior measure of algorithmic efficiency because it concentrates on the relationship between supportive and meaningful work. Supportive work is affiliated with practices intended to guide processing (e.g. selectivity estimators) or early termination (e.g. threshold calculations), and maintain partially or fully evaluated candidate objects. Breaking down memory accesses into supportive and meaningful ones help us reason about why they occur and how to reduce them independently. Barring procedures geared towards high throughput, practices attaining low *rank uncertainty* are equally important for efficient parallel main-memory Top-k query processing.

We validated the above hypothesis by conducting experiments measuring latency per object evaluation and *rank*



	a_1	a_2	$\sum$
o_1	0.87	0.60	1.47
o_2	0.6	0.70	1.3
o_3	0.70	0.90	1.6
o_4	0.40	0.90	1.3
o_5	0.22	0.85	1.07
o_6	0.78	0.56	1.34
o_7	0.5	0.33	0.83
o_8	0.35	0.45	0.8
o_9	0.80	0.30	1.1

Sorted Lists

a_1	a_2
$o_1 = 0.87$	$o_3 = 0.90$
$o_9 = 0.80$	$o_4 = 0.90$
$o_6 = 0.78$	$o_5 = 0.85$
$o_3 = 0.70$	$o_2 = 0.70$
$o_2 = 0.60$	$o_1 = 0.60$
$o_7 = 0.50$	$o_6 = 0.56$
$o_4 = 0.40$	$o_8 = 0.45$
$o_8 = 0.35$	$o_7 = 0.33$
$o_5 = 0.22$	$o_9 = 0.30$

1. *Objects*:$\{o_1, o_3\}$, $Q_k = \{o_3, 1.6\}$, $T = 1.77$
2. *Objects*:$\{o_9, o_4\}$, $Q_k = \{o_3, 1.6\}$, $T = 1.70$
3. *Objects*:$\{o_6, o_5\}$, $Q_k = \{o_3, 1.6\}$, $T = 1.63$
4. *Objects*:$\{o_2\}$, $Q_k = \{o_3, 1.6\}$, $T = 1.40$

Total Objects Fetched = 7

Figure 3: TA execution and data access example

uncertainty for different threshold-based solutions (Fig. 2). Rank uncertainty was calculated as the ratio of the total memory accesses (MT(A)) using performance counters[1] over the accesses related to score aggregation (ME(A)) by multiplying the number of evaluated objects to the corresponding query attributes. Figure 2 was created by evaluating 8 attribute queries on a collection of 256 million objects that were synthetically generated following a uniform distribution.

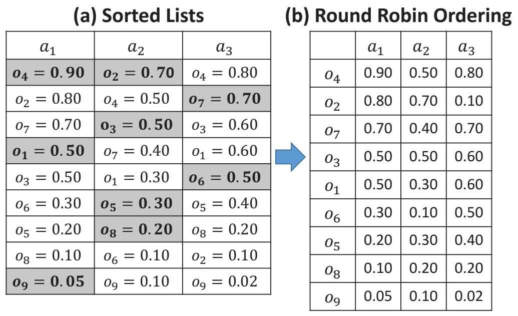
LARA, BPA, and DL experience higher *rank uncertainty* because of memory accesses associated with candidate maintenance, seen position tracking (i.e. best position threshold), and candidate generation (i.e. graph traversal), respectively. Although BPA and DL require less object evaluations compared to TA, their total workload is much higher, contributing to higher latency. Full Table Evaluation (FTE) attains the lowest possible *rank uncertainty* because it performs work related only to evaluating and ranking objects. HL and T2S leverage on data layering and reordering, techniques that require some threshold calculations and maintenance of few candidate objects while performing increased number of object evaluations. Hence, their *rank uncertainty* is relatively low while the attainable cycles per object are somewhat higher compared to FTE. We adopted the practices of HL and T2S, and developed their optimized parallel in-memory variants (i.e. SLA and VTA). These solutions utilize blocking which results to less threshold calculation, thus lower *rank uncertainty*, while being optimized for parallel main-memory execution enabling lower cycles per object. We improved *rank uncertainty* further, designing an improved solution called PTA which utilizes a sophisticated partitioning mechanism (i.e. angle space partitioning). As indicated by the previous figure, its *rank uncertainty* is close to FTE because of less object and threshold calculations while the attained cycles/object remain very low.

5. SINGLE-THREAD TOP-K SELECTION

In this section, we review TA's execution using the toy example of Figure 3. In that figure, the left table depicts the initial collection of objects (and their attributes) and the right table the corresponding sorted-lists (maintained using indexes, e.g. B-trees). Below those tables we enumerate the individual steps of TA's execution for each list level when executing Top-1 query evaluation.

TA operates by retrieving in round-robin order the objects seen at each list-level. For every newly seen object, TA

[1]Reported using *mem_uops_retired.all_loads*.



(a) Sorted Lists

a_1	a_2	a_3
$\boldsymbol{o_4 = 0.90}$	$\boldsymbol{o_2 = 0.70}$	$o_4 = 0.80$
$o_2 = 0.80$	$o_4 = 0.50$	$\boldsymbol{o_7 = 0.70}$
$o_7 = 0.70$	$\boldsymbol{o_3 = 0.50}$	$o_3 = 0.60$
$\boldsymbol{o_1 = 0.50}$	$o_7 = 0.40$	$o_1 = 0.60$
$o_3 = 0.50$	$o_1 = 0.30$	$\boldsymbol{o_6 = 0.50}$
$o_6 = 0.30$	$\boldsymbol{o_5 = 0.30}$	$o_5 = 0.40$
$o_5 = 0.20$	$\boldsymbol{o_8 = 0.20}$	$o_8 = 0.20$
$o_8 = 0.10$	$o_6 = 0.10$	$o_2 = 0.10$
$\boldsymbol{o_9 = 0.05}$	$o_9 = 0.10$	$o_9 = 0.02$

(b) Round Robin Ordering

	a_1	a_2	a_3
o_4	0.90	0.50	0.80
o_2	0.80	0.70	0.10
o_7	0.70	0.40	0.70
o_3	0.50	0.50	0.60
o_1	0.50	0.30	0.60
o_6	0.30	0.10	0.50
o_5	0.20	0.30	0.40
o_8	0.10	0.20	0.20
o_9	0.05	0.10	0.02

Figure 4: Round robin reordering example

calculates its score and inserts it into the associated priority queue (denoted as Q_k), if and only if the corresponding score is greater than the current minimum. At the same time, a threshold value is computed from the attributes of every object seen under round robin access. Query evaluation terminates when Q_k contains k objects and its current minimum score is greater or equal to the threshold value. In our example, the first iteration will access objects o_1 and o_3 and keep only o_3 because it attains the highest score according to $\sum$. The threshold value is $1.77 = (o_9.a_1) + (o_4.a_2) = 0.87 + 0.9 \geq \sum(o_3) = 1.6$, thus processing continues. In steps 2 and 3, the newly seen objects o_9, o_4 and o_6, o_5 attain lower score than o_3, thus they are never inserted into Q_k. Query evaluation continues because the corresponding threshold values (1.70 and 1.63) are still greater than 1.6. At step 4, o_2 is discarded because it ranks lower that o_3 and processing terminates because the threshold value (1.4) is greater than the minimum in Q_k. TA incurs too many random accesses and requires keeping track of of seen objects to avoid reevaluation. This results in cache pollution as the number of evaluated objects grows. This puts increased pressure on the main memory bus, especially during parallel processing, and can be overall detrimental to performance.

In the wake of these issues, we develop three different algorithmic solutions; mainly Vectorized Threshold Algorithm (VTA), Skyline Layered Algorithm (SLA), and Partitioned Threshold Algorithm (PTA). All these methods rely on static object reordering as proposed by Han et al. [18] and are optimized for SIMD and multithreaded execution. They leverage on our newly proposed and easily maintainable data layout known as Threshold Block Layout (TBL; to be discussed next). VTA, SLA, and PTA differ in that they utilize their own data partitioning strategies to enable multithreaded execution.

5.1 TBL List and TBL Node

Fig. 4 presents an example showcasing how to order a relation based on the round robin access. Each sorted-list contains objects (highlighted in gray), indicating the first *seen* position for a given object under sorted access. For a collection of objects o_i where $i \in [0, n-1]$ and their corresponding collection of sorted-lists SL_j with $j \in [0, d-1]$, there exists a unique position of first appearance denoted with $p_i[j] \in [0, n-1]$. This position is calculated based on the following formula:

$$\widetilde{p_i} = \underset{\forall j \in [0, d-1]}{\operatorname{argmin}} \{p_i[j]\} \tag{3}$$

During query evaluation, not all threshold calculations are necessary since they do not contribute towards satisfying the stopping conditions. For example in Fig. 3, only the fourth threshold evaluation was needed. It is not possible to pinpoint the exact threshold for arbitrary data distributions and query configurations (i.e. result size, preference vector). However, it is possible to maintain a small fraction of all thresholds sacrificing some algorithmic efficiency for better processing throughput.

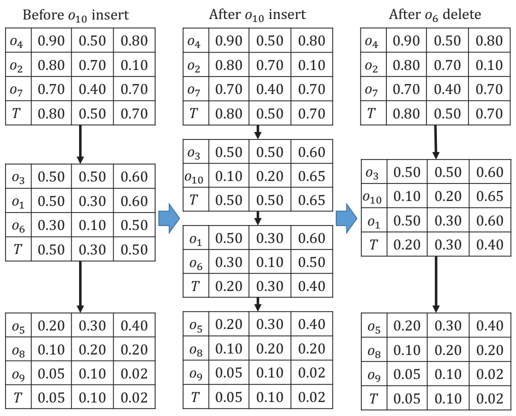
Algorithm 1 Build TBL List

```
$D$ = Input dataset., $N_{TBL}$ = TBL node size.
 1: for $j$ in $[0, m-1]$ do
 2:     $c = \text{sort}(D[:, j])$                ▷ Sorted <id,score> pairs.
 3:     for $i = 0$ to $n-1$ do
 4:         $P_s[c[i].id] = min(P_s[c[i].id], i)$
 5:         if $i \% N_{TBL} == 0$ then $L.set(i/N_{TBL}, c[i].score)$
 6:     end for
 7: end for
 8: $P_s = sort(P_s)$
 9: for $i = 0$ to $n-1$ do
10:     $L.assign(i/N_{TBL}, P_s[i].id, D)$
11: end for
```

In order to achieve this goal, we develop a data layout, called Threshold Block Layout (TBL) node. Each TBL node contains a fixed collection of objects, and a set of attributes that correspond to the node's threshold. For a given relation and depending on the TBL node size, we maintain a list of multiple nodes called TBL list. This data structure has similar properties to a clustered index, in that it stores data in close proximity and according to a predetermined ordering. Fig. 5 (left) showcases a TBL list configuration with node size 3 (i.e. each node has three objects plus the threshold T) for the list ordering shown in Fig. 4. The threshold of each node equates to the last object's threshold first seen position. For example, node 2 is assigned threshold attributes $0.5, 0.3, 0.5$ because o_6 (the last object) appears in column a_3 at the fifth level where the threshold contains these exact attributes (see Fig. 4 (a)). Choosing a small TBL node size results in estimating the true stopping threshold with greater accuracy but demands higher memory footprint and proportional threshold calculations. Modern multiprocessors benefit from large node size because it equates to a large pool of unordered work, providing opportunities for better instruction level parallelism.

Algorithm 1 summarizes the steps related to building a TBL list. For each attribute column (Line 1), we create a sorted list of <id,score> pairs in descending order of score (Line 2). For each sorted list, we update the first seen position of every object (Line 4). We assign the i-th threshold attribute for the given attribute column to partition i/N_{TBL} when i is divisible by the TBL node size (Line 5). We sort the objects in ascending order to the first seen position (Line 8). Finally, we assign object i to partition i/N_{TBL} (Lines 9-11).

Maintenance: The TBL list can easily support insertion, and deletion of objects. Assume that the TBL node has a minimum (B_{min}) and a maximum (B_{max}) size, where $B_{max} = 2 \cdot B_{min} - 1$ and the root node can have minimum 1 object. A new object $o_v = \{a_0, a_1, ..a_{d-1}\}$ is inserted into the list by performing binary search to discover node B having a threshold $T = \{t_0, t_1 ... t_{d-1}\}$ such that $\exists a_i \in o_v$ where $a_i \geq t_i$. This assignment process guarantees that any newly inserted object follows the first seen position principle, hence we do not need to update the thresholds because



Before o_{10} insert:

o_4	0.90	0.50	0.80
o_2	0.80	0.70	0.10
o_7	0.70	0.40	0.70
T	0.80	0.50	0.70

o_3	0.50	0.50	0.60
o_1	0.50	0.30	0.60
o_6	0.30	0.10	0.50
T	0.50	0.30	0.50

o_5	0.20	0.30	0.40
o_8	0.10	0.20	0.20
o_9	0.05	0.10	0.02
T	0.05	0.10	0.02

After o_{10} insert:

o_4	0.90	0.50	0.80
o_2	0.80	0.70	0.10
o_7	0.70	0.40	0.70
T	0.80	0.50	0.70

o_3	0.50	0.50	0.60
o_{10}	0.10	0.20	0.65
T	0.50	0.50	0.65

o_1	0.50	0.30	0.60
o_6	0.30	0.10	0.50
T	0.20	0.30	0.40

o_5	0.20	0.30	0.40
o_8	0.10	0.20	0.20
o_9	0.05	0.10	0.02
T	0.05	0.10	0.02

After o_6 delete:

o_4	0.90	0.50	0.80
o_2	0.80	0.70	0.10
o_7	0.70	0.40	0.70
T	0.80	0.50	0.70

o_3	0.50	0.50	0.60
o_{10}	0.10	0.20	0.65
o_1	0.50	0.30	0.60
T	0.20	0.30	0.40

o_5	0.20	0.30	0.40
o_8	0.10	0.20	0.20
o_9	0.05	0.10	0.02
T	0.05	0.10	0.02

Figure 5: TBL list insert-delete example.

they roughly approximate those seen under sorted list access. When the node size becomes larger than B_{max}, we split it into two nodes using Algorithm 1 for all objects within the node. The second node's threshold is initialized with the maximum attributes of the subsequent node. In Fig. 5 (center), o_{10} is inserted and the third node is assigned a threshold consisting of the maximum attributes of the fourth node. Deleting an object may result in two nodes being merged. In that case, we merge with the previous node in-order and update its threshold with the one of the merging node. Fig. 5 (right) shows the deletion of o_6 which results in merging nodes 2 and 3. In the worst case, a merge can cause at most another split to happen when the new node size exceeds B_{max}. This happens because a node's size ranges in $[B_{min}, 2 \cdot B_{min} - 1]$, and the merging node's size is $B_{min} - 1$, hence the total size of the new node will range in $[2 \cdot B_{min} - 1, 4 \cdot B_{min} - 2]$. In any case, we can create two nodes following the splitting steps outlined previously. Updates are implemented by combining a delete and an insert operation.

Algorithm 2 Vectorized Threshold Algorithm

```
$L_{TBL}$ = TBL List., $W$ = Preference Vector
$Q_k$ = Priority Queue., $k$ = Query result size.
1: for $B \in L_{TBL}$ do
2:     $vta\_kernel(B, W, Q_k, k)$
3:     for $j = 0$ to $m-1$ do
4:         $T += B.threshold[j] \cdot W[j]$
5:     end for
6:     if $T \leq Q_k.min()$ & $Q_k.size() == k$ then return $Q_k$
7: end for
```

The TBL list was designed to improve Top-k selection performance, which is inline with related work [15, 29, 18, 19, 33] focusing solely on selection. It could be extended on rank joins [24] and possibly combined with other operators (i.e. group-by, join) , but this is out of this paper's scope.

5.2 The Vectorized Threshold Algorithm

Algorithm 2 summarizes the steps of VTA which operates on a single TBL list. For each TBL node (Line 1), the algorithm evaluates the objects associated with it by utilizing the VTA kernel (Line 2) and then calculates the node's threshold (Lines 3-5). When both stopping conditions are satisfied, the algorithm halts processing and returns a priority queue consisting of the k-highest ranked objects (Line 6). TBL nodes store their data using column-major order to en-

Algorithm 3 VTA Kernel

```
B = TBL Node., W = Preference Vector, $Q_k$ = Priority Queue.
1: for $i = 0$ to $|B| - 1$ do
2:     for $m = 0$ to $d - 1$ do
3:         $p_v$=_mm256_set_ps($W[m]$)
4:         $j = |B| * m + i$
5:         $ld_0$ =_mm256_load_ps(&$B[j]$)
6:         $ld_1$ =_mm256_load_ps(&$B[j+8]$)
7:         $r_0$ =_mm256_add_ps($r_0$, _mm256_mul_ps($ld_0, p_v$))
8:         $r_1$ =_mm256_add_ps($r_1$, _mm256_mul_ps($ld_0, p_v$))
9:     end for
10:    _mm256_store_ps(&$buf[0], r_0$)
11:    _mm256_store_ps(&$buf[8], r_1$)
12:    for $r \in buf$ do
13:        if $Q_k$.size() $< k$ then $Q_k$.push($id, r$)
14:        else if $Q_k$.min() $< r$ then $Q_k$.pop(), $Q_k$.push($id, r$)
15:    end for
16:    $i+ = 16$
17: end for
```

able SIMD vectorization. Our implementation makes use of AVX instructions that support 8 lane operations. The VTA kernel (Algorithm 3) evaluates the score for a fixed group of objects per iteration (Lines 2-9). Once 16 objects have been evaluated using SIMD operations, their scores are written back to a local buffer (Lines 10-11). This local buffer is used to update the contents of the associated priority queue (Lines 12-15). A new object is inserted into the queue if no more than k objects already exist (Line 13), or when its score is greater than that of the minimum scored object, at which point the latter object is evicted (Line 14).

5.3 VTA Complexity Analysis

VTA does not require keeping track of evaluated objects and is able to maintain a constant candidate set at each processing step. In addition, it favors instruction level parallelism and vectorization which improves bandwidth utilization. However, it exhibits increased *rank uncertainty* for queries on a subset of the reordered attributes, resulting in many more object evaluations compared to TA.

Let n_p be the depth at which TA is able to stop processing new objects. In the worst case, the total number of object evaluations will be $n_p \cdot m$ for a query with m attributes. In contrast, VTA requires $(n_p + n_{TBL}) \cdot d$ evaluations where d is the number of attributes and n_{TBL} the node size. This inefficiency motivates the development of a solution having better algorithmic efficiency. In the following section, we describe two possible solutions, one based on previous work (i.e. skyline layering) and a new method relying on angle space partitioning.

Algorithm 4 Skyline layering with TBL list construction.

```
D = Relation data., LL = List of layers.
1: while $D \neq \emptyset$ do
2:     $L$ =skyline($D$)
3:     $LL$.append(build_tbl($L$))
4:     $D = D - L$
5: end while
```

6. MULTITHREADED TOP-K SELECTION

There are two ways to parallelize TBL list processing: (1) enable parallel evaluation within each TBL node, (2) create multiple TBL lists and assign each one to distinct threads for processing. Both options should be optimized to achieve high algorithmic efficiency. In the following sections, we discuss two different algorithmic solutions geared towards implementing the previous parallel query evaluation strategies. The first method (SLA) relies on the practices established in [20], while the second method (PTA) follows a new direction, utilizing angle space partitioning to optimally partition the data for processing.

6.1 The Skyline Layered Algorithm

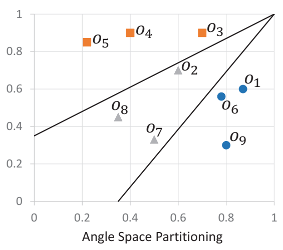
SLA combines the idea of reordering the base table, with the concept of layering data using the skyline operator. Our implementation leverages vectorization and the TBL list organization, in addition to utilizing the pruning properties of the skyline layers. Although, our solution follows the best practices established by Heo et al. [20], it presents the first attempt to enable parallel processing and vectorization using static reordering of each layer. Related solutions using skyline layering [27, 44] rely on graph traversal to improve algorithmic efficiency, a process that is often hard to vectorize. In addition, these solutions require maintenance of a high number of candidates at each processing step, a characteristic that is incompatible to our original goals (Section 4.2) and inappropriate for our current environment.

Algorithm 4 showcases the pseudo-code for calculating the skyline layers and their corresponding TBL lists. Utilizing the parallel skyline algorithm presented in [9], we calculate the skyline set (Line 2). For this collection of points, we create a TBL list which is added at the end of a list containing all layers (Line 3). Finally, we update the dataset by removing the skyline set (Line 4) and repeat the previous steps until there are no more points in D.

Algorithm 5 Skyline Layered Algorithm

```
LL = Layers List, W = Preference Vector.
$Q_k$ = Priority queues, k = Query result size., tid = Thread id
1: for $i = 0$ to $|LL| - 1$ do
2:     if $i > k$ then break
3:     for $B \in LL[i]$ in parallel do
4:         vta_kernel($B, W, Q_k[tid], k$)
5:         if $tid == 0$ then
6:             for $j = 0$ to $m - 1$ do
7:                 $T+ = B.threshold[j] \cdot W[j]$
8:             end for
9:         end if
10:        __synchronize
11:        if $T \leq Q_k.min()$ & $Q_k.size() == k$ then break
12:    end for
13: end for
14: return merge($Q_k$)
```

Algorithm 5 summarizes SLA's execution steps. SLA processes only the first k layers (Line 2) since according to Chang et al. [6] the Top-k objects are guaranteed to appear in them. Within each layer, we process the individual TBL nodes by assigning consecutive objects for evaluation to distinct threads (Line 4). Thread zero is responsible for calculating the node's threshold (Lines 5-9). In order to ensure that T has been computed and all threads have completed their evaluations, *omp_barrier* is used to synchronize processing (Line 10). When the accrued number of objects from all queues are $\geq k$ and their minimum scored object is $\geq T$ processing terminates for the given layer (Line 11). Finally, when all k layers have been processed, the individual queues are merged together before returning the Top-k result (Line 14).



	a_1	a_2
P_1	$o_3 = 0.70$	$o_3 = 0.90$
	$o_4 = 0.40$	$o_4 = 0.90$
	$o_5 = 0.22$	$o_5 = 0.85$

	a_1	a_2
P_2	$o_2 = 0.60$	$o_2 = 0.70$
	$o_7 = 0.50$	$o_8 = 0.45$
	$o_8 = 0.35$	$o_7 = 0.33$

	a_1	a_2
P_3	$o_1 = 0.87$	$o_1 = 0.60$
	$o_9 = 0.80$	$o_6 = 0.56$
	$o_6 = 0.78$	$o_9 = 0.30$

Figure 6: ASP on objects from Fig 3.

6.2 The Partitioned Threshold Algorithm

SLA relies on discovering an optimal linear ordering for all objects in the dataset to improve algorithmic efficiency. Since this is a form of global optimization, it will not work well for increasing attributes due to *the curse of dimensionality* which makes it increasingly difficult to identify similar properties between high-dimensional objects. In order to overcome this limitation, we develop a versatile solution which relies on partitioning the objects according to their attribute correlation before choosing a local optimal ordering. We call this method the Partitioned Threshold Algorithm (PTA).

PTA utilizes Angle Space Partitioning (ASP), a strategy first proposed in [37] for improving skyline computation. This strategy has never been used in the context of Top-k selection queries, hence it is a new approach. In addition, PTA necessitates partitioning the data in collections of correlated objects (i.e. objects around a given trend line), thus it is not tied to that specific partitioning strategy. In reality, our contribution with PTA revolves around the idea of minimizing the number of possible total orderings within each partition by considering object correlation. Any partitioning strategy that accomplishes these goals is suitable to overcome the limitations associated with choosing a global ordering.

Algorithm 6 Partitioned Threshold Algorithm

```
$PL = Partitions\ List,\ W = Preference\ Vector.$
$Q_k = Priority\ queues,\ k = Query\ result\ size.,\ tid = Thread\ id$

 1: for $p \in PL$ in parallel do
 2:     for $B \in p$ do
 3:         $vta\_kernel(B, W, Q_k[tid], k)$
 4:         for $j = 0$ to $m - 1$ do
 5:             $T+ = B.threshold[j] \cdot W[j]$
 6:         end for
 7:         if $T \leq Q_k.min()$ & $Q_k.size() == k$ then break
 8:     end for
 9: end for
10: return $merge(Q_k)$
```

ASP maps each multi-dimensional object from cartesian space to hyperspherical space using the geometric equations presented in [37]. The data are then partitioned using grid partitioning over the $d-1$ space defined by the associated angular coordinates. In effect, this leads to grouping together objects that are increasingly correlated as the angle of the partition shrinks (see Fig 6). Assuming a splitting factor s we create s^{d-1} distinct partitions for relations with d attributes. Through recursive splitting of each angular dimension, we are able to maintain roughly the same number of objects per partition while also building a separate TBL list for everyone.

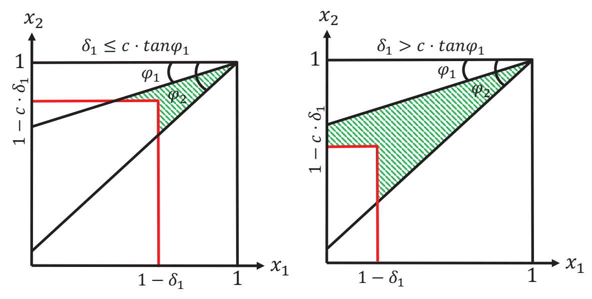


Figure 7: Processed areas for varying δ_i using ASP.

ASP partitioning contributes towards discovering an optimal per partition ordering independently of the user-defined monotone function. Hence, each partition may require as little as k evaluations to discover the highest ranked objects. This becomes apparent in Figure 6 where reordering the data per partition is almost optimal, resulting in at most one object evaluation for the corresponding Top-1 query. Typically Top-k query evaluation involves only a subset of all attributes. ASP operates optimally only when querying all of the indexed attributes. Our extensive experimentation showcased that processing Top-k sub-queries incur some minor performance degradation. Hence, our methods are suitable for solving Top-k selection efficiently.

6.3 PTA Algorithm

Algorithm 6 summarizes the execution steps of PTA. We assign each partition to a distinct thread and in parallel process their corresponding TBL lists (Lines 1 - 9). For each list assigned to a thread, the VTA kernel is utilized to evaluate one node at a time from the TBL list (Line 3). The threshold is calculated after the evaluation of each node (Lines 4-6), then the corresponding stop conditions are evaluated (Line 7). Note that stopping applies only to the partition which is currently under processing. A thread is responsible for evaluating multiple partitions. Once all partitions are processed, the individual priority queues are traversed and only the k-highest ranked objects are returned (Line 10). It is possible to employ different strategies when merging the queues together. However, the cost of merging is relatively small and is not detrimental to high performance.

6.4 PTA Complexity Estimation

Consider an algorithm leveraging on TA (i.e. VTA, SLA, T2S, HL-index, IO-Top-k) at step t of its execution where k objects have been identified. Let $\tau = (1-\delta_1, 1-\delta_2, \ldots, 1-\delta_m)$ ($\delta_i \in [0,1]$) be the combined attribute threshold. For the worst case object arrangement, the corresponding algorithm would need to evaluate all objects with at least one $a_j \geq 1 - \delta_i$. Assuming uniformly distributed values, the expected number of object evaluations can be estimated by the volume (or area in 2D) of the polytope enclosed by the threshold and hypercube $[0,1]^m$. This is $E_{TA}(t) \leq n \cdot \left[1 - \prod_{i=1}^{m} (1-\delta_i)\right]$. Typically, δ_i grows linearly with k and N, while $E(t)$ grows exponentially to the query dimensions. This growth rate is conceptually equivalent to the number of candidates maintained during processing for no random access methods [15, 29, 18]. Hence, any strategy geared towards limiting such growth could be used to improve performance for that class of algorithms as well.

Let us consider the 2D case of ASP where $\delta_2 = c \cdot \delta_1, c \in [0,1]$ for simplicity. Fig. 7 presents the case where $0 \leq$

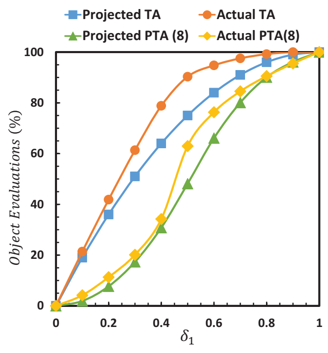


Figure 8: Projected vs actual object evaluations.

$\phi_1 < \phi_2 \leq \frac{\pi}{4}$. Given some threshold, an ASP enabled algorithm following the TA ordering would process in the worst case the depicted shaded region. This occurs because TA performs a plane sweep for each attribute while ASP restricts the associated region depending on the partition angle. For $\delta_1 \in [0, 1]$, the processed area is computed using Equations 4, 5 where $\delta_1 \leq$ or $>$ to $c \cdot tan(\phi_1)$, respectively.

$$A_1 = \frac{\delta_1^2}{2} \cdot \left(tan\phi_2 - tan\phi_1 + \left(1 - \frac{1}{c \cdot tan\phi_1}\right)^2 \cdot tan\phi_1 \right) \quad (4)$$

$$A_2 = 0.5 \cdot \left(tan\phi_2 - tan\phi_1 - (1 - \delta_1)^2 \cdot tan\phi_2 \right) \quad (5)$$

Fig. 8 presents the projected (utilizing Equations 4, 5) vs actual number of object evaluations for TA vs PTA (assuming 8 partitions). The actual number was measured through experimentation with varying k on 2^{28} uniformly generated objects. It is apparent that the projected curves are very similar to the actual ones, indicating an average improvement of at least two orders of magnitude. ASP is extremely efficient for $\delta_i \leq 0.004$ where $\frac{E_{PTA}(t)}{E_{TA}(t)} \geq 400$. This finding indicates that intelligent partitioning is pivotal to achieving high parallel efficiency and should precede the choice of a suitable Top-k implementation that favors high system performance. Note that this does not entail the selection of any specific Top-k optimization, it only acts as the foundation for the design of an efficient parallel Top-k algorithm.

Table 1: Processing environment per tested method.

	Single Threaded (Scalar)	Single Threaded (SIMD)	Multi-Threaded (SIMD)
Single Query	TA, LARA, HL, DL, VTA, SLA, PTA	FTE, VTA, SLA, PTA	VTA, SLA, PTA
Query Batch			VTA, PTA

In fact, our analysis suggests that any previously proposed Top-k method, aimed at limiting the exponential growth of candidate objects and object evaluations, can be parallelized effectively using ASP partitioning. However, according to our analysis, the in-memory execution environment dictates utilizing data reordering and layering because these techniques favor sequential access and reduced candidate object maintenance cost. As a result, we developed PTA to utilize these practices in combination with ASP. The main conclusions of our analysis are also applicable for skewed data

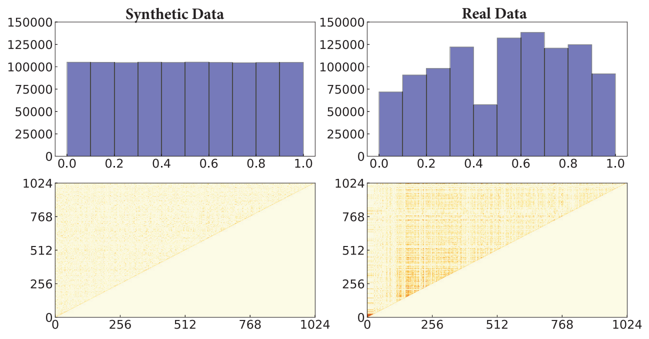


Figure 9: Distribution properties of synthetic vs real data (top: histogram, bottom: correlation matrix).

distributions. This reasoning is drawn from [37] where the authors used equi-volume instead of regular grid partitioning on polar coordinate space to adjust the partition boundaries accordingly. Following this methodology, the actual partitioning is applied on the densest regions of space where the local distribution characteristics are similar to that of a uniform dataset. Hence, the properties of early termination are still satisfied enabling improved algorithmic efficiency.

7. EXPERIMENTAL ENVIRONMENT

We provide a detailed experimental evaluation, comparing against different categories of algorithms from previous work which include list-based solutions optimized for random-access (TA) or sorted-access (LARA), layered-based solutions geared for efficient blocked access (HL) or high algorithmic efficiency (DL) and hardware optimized algorithms (i.e (Full Table Evaluation (FTE)),

We focus on three different types of experiments based on the processing models described in Section 4.1. Table 1 summarizes the environment in which every developed algorithmic was tested. In the interest of fairness, we implemented the scalar variants of our proposed solutions (i.e. FTE, VTA, SLA, PTA) and compare those to previous work which is inherently incompatible with SIMD and multithreading. Following those experiments, we present performance measurements for our proposed solutions when evaluating a single query using either single-threaded or multithreaded SIMD processing. Finally, we took the best performing solutions (i.e VTA, PTA) and evaluated their performance, measuring latency, throughput and parallel efficiency for a randomly generated query batch (see Section 7.2).

7.1 System Specification

All our experiments were conducted on a two socket 2.4 GHz Intel Xeon E5-2680 v4 CPU with hyperthreading enabled (56 cores in total) and 64 GB DDR4. We implemented each algorithm in C++ utilizing the standard priority queue implementation. FTE, VTA, SLA and PTA were designed to utilize AVX instructions and assume that the data are stored in column-major order. For these methods, we also developed a scalar version used to present a fair comparison against previous work which was not originally designed for column-major execution or to use AVX-instructions. We used GCC version 5.4.0 having the O3 optimization flag enabled and the OpenMP framework to enable multithreaded execution. The thread affinity was set

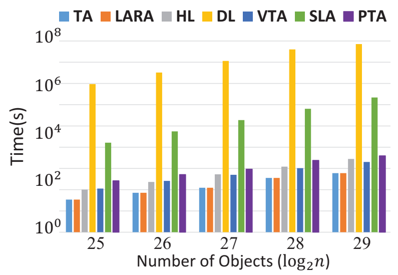


Figure 10: Initialization cost comparison.

using OMP_PLACES = sockets. Unless stated/shown otherwise, all multithreaded measurements where acquired for 32 threads. Our code is publicly available in Github [42].

7.2 Dataset & Query Format

We conducted experiments using both real and synthetic data. Unless otherwise stated, the parameters of our experiments are summarized in Table 2. Similar to previous work [20, 44, 29], our synthetic data follow a uniform distribution and were created using the standard dataset generator from [4]. We performed experiments retrieving the k objects with the highest sum (i.e. $w_i = 1, \forall i \in [0, d-1]$), unless stated otherwise. $MTMQ$ is evaluated on 131072 randomly generated queries for $k = 16$, $n = 2^{28}$. The real dataset consist of temperature measurements acquired from NOAA [30].

Table 2: Experimental parameters.

	Objects (n)	*Attributes* (d)	*Result Size* (k)
(n)	$[2^{25}, 2^{29}]$	6	128
(d)	2^{28}	$[2, 8]$	128
(k)	2^{28}	6	$[16, 1024]$

In Figure 9, we summarize the distribution characteristics for a random sample of our synthetic and real data using a single attribute histogram and correlation matrix (light = zero correlation, dark= high correlation). In contrast to the synthetic data (that follow a uniform distribution), the real dataset follow a bimodal distribution. From the correlation matrices, we observe that the synthetic data contain objects having almost no linear relationship. In contrast, the real data consist of noticeably larger clusters of strongly correlated objects. Low (High) correlation between objects is responsible for decreasing (increasing) the likelihood of constructing highly correlated partitions just by chance. Hence, we expect methods that do not utilize intelligent partitioning to perform poorly on data collections with zero or negative linear correlation, especially for queries on high number of attributes.

8. PERFORMANCE TUNING

In this section, we discuss experiments related to the cost of initialization (i.e. reordering, layering, creating sorted lists), the chosen TBL node size, and the effects of varying query weights.

8.1 Initialization Cost

In Fig. 10, the highest initialization cost is incurred by methods that require calculating the skyline set to construct

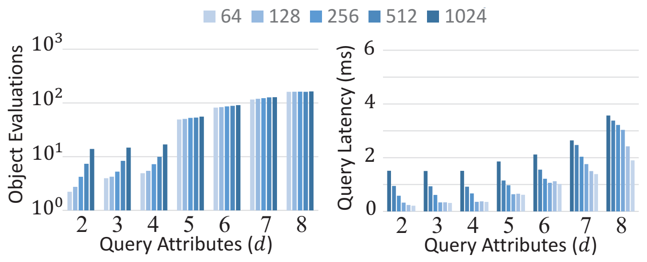


Figure 11: Block size vs latency-object evaluations.

the corresponding data layers. DL exhibits the highest initialization overhead because in addition to the skyline it requires identifying all points dominated by any point in the parent layer. Likewise, SLA attains the second highest initialization cost because it requires also reordering the layers according the first seen principle. On the other hand, HL requires discovering the skyline set and building the individual lists for each layer, which translates to incurring about the same initialization cost of VTA and PTA. Compared to TA and LARA, the previous methods exhibit at most 4× and 7× higher initialization overhead which is an acceptable trade-off considering that all of them perform 350× and 33000× better in terms of query latency. Note that initialization is executed only once, similar to any other type of index like structure.

Table 3: Individual query weights.

Q_0	$(1, 1, 1, 1, 1, 1, 1, 1)$
Q_1	$(.1, .2, .3, .4, .5, .6, .7, .8)$
Q_2	$(.8, .7, .6, .5, .4, .3, .2, .1)$
Q_3	$(.1, .2, .3, .4, .4, .3, .2, .1)$
Q_4	$(.4, .3, .2, .1, .1, .2, .3, .4)$

8.2 TBL Node Size

Figure 11 presents the measured object evaluations and query latency for varying TBL node size. We observed a noticeable increase in object evaluations for queries with 2 to 4 attributes and somewhat mediocre increase on queries with 5 to 8 attributes. In contrast, query latency follows a downward trend for increasing TBL node size. This happens because having large node size translates to less threshold evaluations and a larger pool of unordered work that favors instruction level parallelism, hence lower latency. Note that a similar downward trend is observed for the threshold memory footprint as the node size increases (i.e. at most 8 MB for 1024 vs 128 MB for 64).

8.3 Varying Preference Vectors

Figure 12 presents the measured object evaluations for the preference vectors of Table 3. VTA exhibits little variation

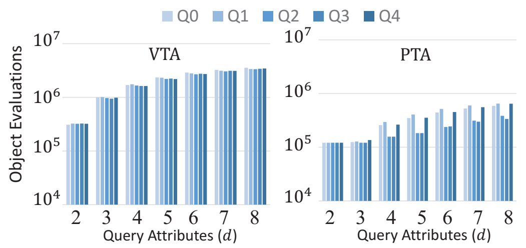


Figure 12: Variable weights vs object evaluations.

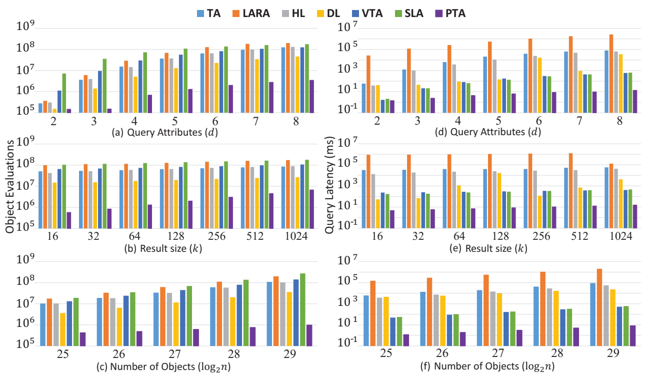


Figure 13: Scalar performance on synthetic data.

in performance, while PTA occasionally performs better for specific weight combinations. PTA's behavior is a consequence of the order in which partitions are processed (i.e. starting from ϕ_1, in ascending order of the corresponding partition angles). For preference vectors Q_2 and Q_3, the specific order of processing favors discovery of high scoring objects early, while the decreasing weight values reduce the magnitude of the threshold for each TBL node. Hence, any partition processed after these objects have been discovered will require less object evaluations due to the higher likelihood for the minimum score to be greater than the associated threshold. PTA is compatible with cost-based scheduling algorithms [3] focused on choosing the best order of evaluating the corresponding partitions. Our experiments follows the worst case order of processing (i.e. round-robin order).

9. SYNTHETIC DATA EXPERIMENTS

In Figures 13 (a), (b), (c), we present the measured number of object evaluations for all developed scalar algorithms. PTA requires the least number of object evaluations despite following a fixed order of processing within each partition. This is apparent for any instance of the Top-k problem, as indicated by our experiments with varying number of attributes, result size, and input size. VTA incurs higher number of object evaluations for queries on few attributes. This happens because it follows a fixed order of processing which forces it to evaluate all objects regardless of the chosen query attributes. SLA performs worse than any other method because it follows the same sub-optimal order of processing while also partitioning the data in few skyline layers which are quite large and often need to be evaluated completely. DL is the second best solution in terms of the number of object evaluations. However, it is not practical because it necessitates a costly initialization step and requires too much auxiliary information the processing of which negatively affects query latency (see next section). TA, LARA, and HL perform poorly for queries on large number of attributes.

In Figures 13 (d), (e), (f), we showcase the query latency for all scalar methods. Again, we observe that PTA outperforms almost every other method. VTA and SLA attain similar query latency with small variations which are inline with their corresponding object evaluations. DL requires traversing frequently the lists of dominated objects for every object within the result set, in order to update its candidate set This results in unstable query latency that occasionally is higher compared to VTA/SLA. LARA attains the worst

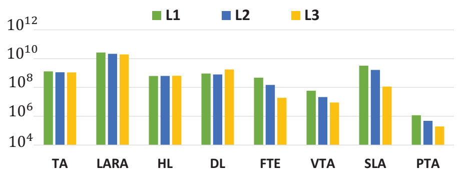


Figure 14: Average number of cache misses for variable attribute queries using performance counters.

query latency because it needs to maintain large candidate sets and update their upper bounds at every step during the shrinking phase. DL's and LARA's behavior is indicative of the performance penalties associated with maintaining too much auxiliary information while requiring also exorbitant amount of computation to avoid only few object evaluations. VTA and SLA maintain only a constant number of candidates (at most k objects). Furthermore, they require few auxiliary information (see 5.1) during processing as compared to other methods (i.e LARA, DL). PTA adheres to the same principles while also utilizing intelligent partitioning which helps with improving algorithmic efficiency and achieving noticeably lower query latency.

9.1 Profiling Cache Misses

In Figure 14, we summarize the average number of cache misses measured using synthetic data for queries having 2 to 8 attributes. We utilized the CPU event counters (i.e. mem_load_uops_retired.l1_miss) to gather the corresponding measurements. Cache misses are counted at a given level and any one above it, since any memory request would have been forwarded from the top levels first. For example, a cache miss at L3 would have been also counted on the measurements of L2 and L1 caches, since the associated data would have been requested and not found at those higher levels.

Methods that perform excessive number of random accesses cause conflict misses that often evict relevant data from every cache level. Future reference to that same data will incur a cache miss at each level, ultimately resulting into a direct memory access. Hence, when measuring the cache misses every level will incur the same amount because every data access will go through each cache level before the data become available to the CPU core. TA, LARA, HL and DL exhibit this behavior as indicated by the corresponding cache measurements.

Methods that favor sequential access incur fewer cache misses at the lowest level because multiple words are transferred from main memory for a given memory request. In addition, data prefetching primitives and SIMD load instructions are also responsible for reducing the total number of cache misses at the lowest level. This behavior is apparent by observing the cache misses for FTE, VTA, SLA and PTA. Our measurements indicate that FTE, VTA, SLA and PTA minimize the number of conflict/capacity misses by serving more memory requests from L3 cache. However, PTA is the only method that reduces cold misses by several orders of magnitudes compared to every other solution.

9.2 Hardware Optimized STSQ Processing

As indicated by Fig 15, PTA attains the best performance among all other hardware optimized solutions for varying instances of the Top-k problem. VTA and SLA achieve similar query latency, with the former being occasionally

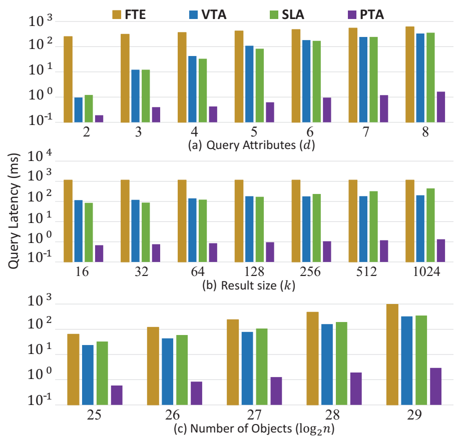


Figure 15: Latency using SIMD instructions.

slightly better than the latter. FTE performs worse that all other implementations because it requires evaluating the full dataset for every query. The above behavior indicates that achieving high algorithmic efficiency is as important as optimizing for the underlying hardware. Similar results were observed in the multithreaded query evaluation of the above algorithms (omitted due to lack of space).

9.3 Hardware Optimized MTSQ Processing

In this section, we concentrate on the evaluation of hardware optimized solutions that follow the $MTSQ$ processing model (denoted with M). We compare against the single-threaded hardware optimized implementations (shown with S) of the previous section.

In Figures 16 (a), (b), (c), we summarize the number of object evaluations for each implementation. VTA-M and SLA-M perform worse than their single-threaded counterparts because they randomly partition the data across distinct TBL lists. Random partitioning increases rank uncertainty since each partition contains objects from the complete data space, possibly omitting those that contribute towards improving the stopping threshold. SLA-M is also affected by the fact that the individual data partitions consist of objects that are weakly and possibly negatively correlated. This organization negatively affects rank uncertainty because it creates a wider gap between the maximal and minimal attribute values making it more probable to first evaluate low scoring objects which appear at the boundaries of the skyline set. Hence, the number of object evaluations increase drastically.

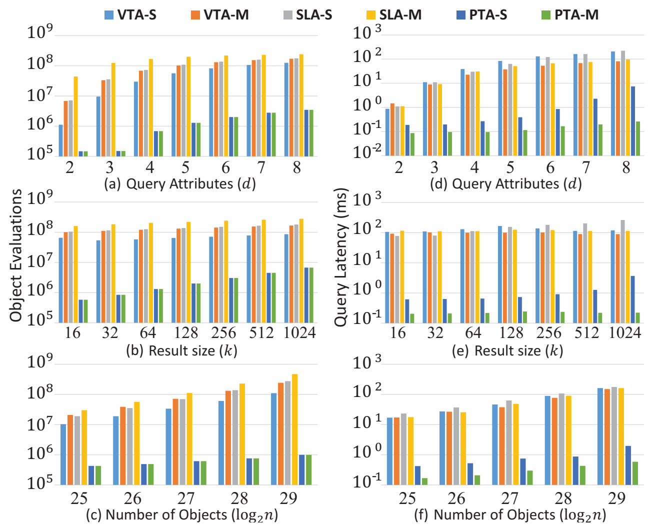


Figure 16: Single vs multithread performance on synthetic data.

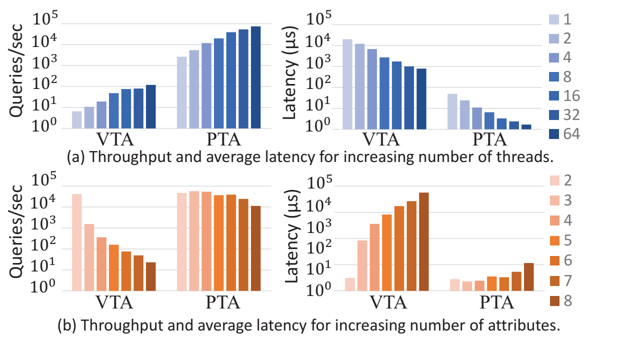


Figure 17: Throughput-latency on synthetic data.

PTA is the only method able to sustain the same algorithmic efficiency for a wide range of experimental parameters. For queries on 2 or 3 attributes, it discovers the Top-k result by evaluating just one TBL node. In fact, considering a smaller node size it can perform less object evaluations at the expense of lower processing throughput due to frequent threshold calculations. Overall, PTA's work grows linearly to the query attributes. In addition, the number of object evaluations grow linearly with respect to increasing values of k and n. This behavior suggests that PTA exhibits good scaling properties across the board and can benefit from the addition of new system resources (e.g. CPU cores, better memory bandwidth).

In Figures 16 (d), (e), (f), we compare the query latency of our single-threaded and multithreaded implementations. These measurements follow a similar trend to the observed number of object evaluations. VTA-M and SLA-M exhibit comparable performance that is overall slightly worse than their single-threaded counterparts because of the former requiring more object evaluations. PTA-M outperforms both of these solutions and the PTA-S variant. However, its performance is slightly worse than proportional to the number of threads used during processing. This happens mainly because updating the individual priority queues is an inherently sequential operation. When k is larger than 128 the combined size of all priority queues (16 threads) is larger than the size of the $L1$ cache (8 bytes for the key, 4 bytes for the score). In that case, each update operation will most likely access the priority queue from $L2$ cache the latency of which is considerably higher. Further improvements on latency and throughput are only possible through batched query processing.

9.4 MTMQ Performance Evaluation

In Figure 17, we present the measured throughput and average query latency for increasing (a) number of threads, and (b) number of attributes. PTA and VTA are both highly optimized, enabling efficient sequential processing and SIMD vectorization. For this reason, our experiments indicate that the observed throughput grows linearly to the number of processing threads. Likewise, the average query latency follows a downward pattern. Both algorithm reach their peak

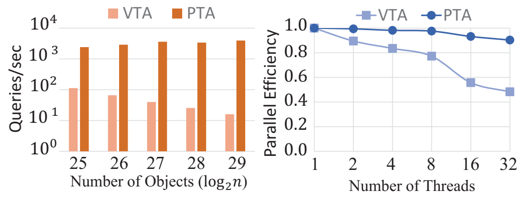


Figure 18: $MTMQ$ Scale-up and parallel efficiency.

performance when utilizing at most 16 threads, due to limitations in the $L1$ cache size. PTA achieves lower query latency because it experiences higher temporal locality during processing. For certain queries only few TBL nodes are examined, a behavior that increases the likelihood of these nodes remaining in cache thus favoring future data re-use. VTA fails to exploit this type of locality because it references many more TBL nodes during processing, thus contributing to the eviction of data useful to future queries. Overall, PTA scales well for increasing query attributes because the associated throughput and latency remain relatively stable.

In Figure 18, we showcase (a) the scale-up and (b) parallel efficiency of PTA compared to VTA. We indicate scale-up by increasing the number of processing threads to the input size. Our experiments demonstrate that PTA exhibits better scaling properties compared to VTA since the former sustains the same throughput for the corresponding experimental parameters. Parallel efficiency was measured by dividing the achieved speed-up with the number of processing threads for the same input size and $MTMQ$ workload (i.e. 512 million objects, and 131072 random queries). PTA scales almost linearly with increasing number of threads thus parallel efficiency is close to 1. On the other hand, VTA's parallel efficiency drops noticeably because it cannot effectively exploit temporal locality for a given batch of queries.

10. WEATHER DATA EXPERIMENTS

In this section, we validate our experimental results using real data. In Figures 19 (a), (b), (c), we summarize the number of object evaluations following $MTSQ$ processing. Similar to the experiments on synthetic data, VTA-S and SLA-S perform less object evaluations than their multithreaded counterparts. PTA-S and PTA-M outperform

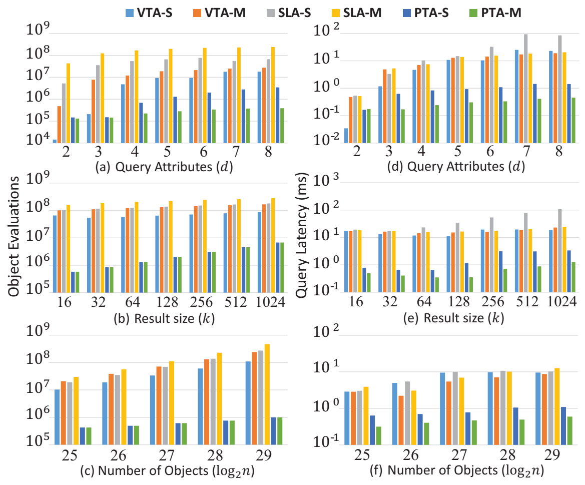


Figure 19: Single-thread vs multithread performance on real data.

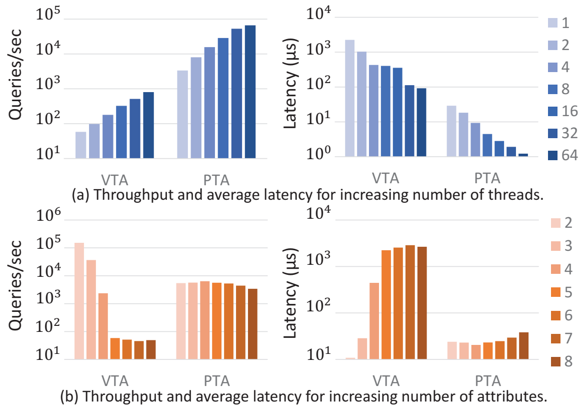


Figure 20: Throughput-latency on real data.

VTA and SLA for almost every experimental parameter, with the only exception being queries on 2 attributes. This happens because consecutive attributes within each object are often highly correlated (i.e. daily temperature values), thus their first seen position matches the ranking order of most preference vectors. The measured query latency for all methods follows a similar trend to the observed number of object evaluations. For PTA-M the major source of contention during processing is the priority queue and the fact that it does not fit completely within $L1$ cache.

In Figure 20, we present experiments measuring throughput and latency on weather data for increasing number of (a) threads and (b) attributes. PTA exhibits superior performance compared to VTA for almost every experimental parameter. This behavior is inline with our experiments on synthetic data. Again, the only exception is queries on 2 attributes in which case, the strong correlation between attributes allows VTA to stop earlier evaluating few TBL nodes. In fact, PTA suffers from the overhead of having to evaluate at least one node per partition.

11. CONCLUSION

In this work, we concentrated on developing algorithmic solutions for parallel in-memory Top-k selection. We proposed three distinct processing models that offer varying levels of parallelism. We introduced the concept of *rank uncertainty* used to discern (given a small representative subset of existing approaches) those having the highest potential to perform well for main memory processing. Based on the rank uncertainty metric, we identified HL and T2S as potential candidates for further parallel optimization (due to their early termination property). We proposed three algorithms, namely VTA and PTA (based on improving T2S), and SLA (based on improving HL). All these methods utilize a simple and easy to maintain data structure, within a conventional DBMS, called a TBL list. PTA adopts a new strategy to minimize *rank uncertainty* which relies on angle space partitioning. In its scalar form, PTA exhibits several orders of magnitude better performance compared to previous works. In addition, PTA outperforms parallel variants of previous methods that utilize reordering (VTA) and layering (SLA).

Acknowledgements. We would like to thank the anonymous reviewers for their constructive comments which contributed to the improvement of this paper. This research was partially supported by NSF grants: IIS-1838222, IIS-1527984 and IIS-1447826.

12. REFERENCES

[1] P. Ahmed, M. Hasan, A. Kashyap, V. Hristidis, and V. J. Tsotras. Efficient computation of top-k frequent terms over spatio-temporal ranges. In *Proceedings of the 2017 International Conference on Management of Data*, pages 1227–1241. ACM, 2017.

[2] R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *Proceedings of the 33rd international Conference on Very Large Databases*, pages 495–506. VLDB Endowment, 2007.

[3] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proceedings of the 32nd International Conference on Very Large Databases*, pages 475–486. VLDB Endowment, 2006.

[4] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of 17th International Conference on Data Engineering*, pages 421–430. IEEE, 2001.

[5] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *Proceedings of the 36th international Conference on Very Large Databases*, 3(1-2):373–384, 2010.

[6] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. In *ACM Sigmod Record*, volume 29, pages 391–402. ACM, 2000.

[7] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Transactions on Knowledge and Data Engineering*, 16(8):992–1009, 2004.

[8] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. In *Proceedings of the 39th International Conference on Very Large Databases*, pages 217–228. VLDB Endowment, 2013.

[9] S. Chester, D. Šidlauskas, I. Assent, and K. S. Bøgh. Scalable parallelization of skyline computation for multi-core processors. In *Proceedings of 31st International Conference on Data Engineering*, pages 1083–1094. IEEE, 2015.

[10] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd international Conference on Very Large Databases*, pages 451–462. VLDB Endowment, 2006.

[11] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *Proceedings of 24th International Conference on Data Engineering*, pages 656–665, April 2008.

[12] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. A candidate filtering mechanism for fast top-k query processing on modern cpus. In *Proceedings of the 36th International Conference on Research and Development in Information Retrieval*, pages 723–732. ACM, 2013.

[13] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance ir query processing. In *Proceedings of the 18th International Conference on World Wide Web*, pages 421–430. ACM, 2009.

[14] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International Conference on Research and Development in Information Retrieval*, pages 993–1002. ACM, 2011.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[16] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proceedings of the 37th International Conference on Very Large Databases*, 4(12):1213–1224, 2011.

[17] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 419–428. Morgan Kaufmann Publishers Inc., 2000.

[18] X. Han, J. Li, and H. Gao. Efficient top-k retrieval on massive data. *IEEE Transactions on Knowledge and Data Engineering*, 27(10):2687–2699, 2015.

[19] X. Han, X. Liu, J. Li, and H. Gao. Tkap: Efficiently processing top-k query on massive data by adaptive pruning. *Knowledge and Information Systems*, 47(2):301–328, 2016.

[20] J.-S. Heo, J. Cho, and K.-Y. Whang. The hybrid-layer index: A synergic approach to answering top-k queries in arbitrary subspaces. In *Proceedings of the 26th International Conference on Data Engineering*, pages 445–448, 2010.

[21] J.-S. Heo, K.-Y. Whang, M.-S. Kim, Y.-R. Kim, and I.-Y. Song. The partitioned-layer index: Answering monotone top-k queries using the convex skyline and partitioning-merging technique. *Information Sciences*, 179(19):3286–3308, 2009.

[22] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *ACM Sigmod Record*, volume 30, pages 259–270. ACM, 2001.

[23] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *Proceedings of the 30th International Conference on Very Large Databases*, 13(3):207–221, 2004.

[24] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

[25] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th International Conference on Research and Development in Information Retrieval*, pages 253–262. ACM, 2014.

[26] C. Jonathan, A. Magdy, M. F. Mokbel, and A. Jonathan. Garnet: A holistic system approach for trending queries in microblogs. In *Proceedings of the 32nd International Conference on Data Engineering*, pages 1251–1262, May 2016.

[27] J. Lee, H. Cho, S. Lee, and S.-w. Hwang. Toward scalable indexing for top-k queries. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3103–3116, 2014.

[28] C. Li, K. Chen-Chuan Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *Proceedings*

of the 2006 International Conference on Management of Data, pages 61–72. ACM, 2006.

[29] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-k aggregation of ranked inputs. *ACM Transactions on Database Systems*, 32(3):19, 2007.

[30] M. J. Menne, I. Durre, R. S. Vose, B. E. Gleason, and T. G. Houston. An overview of the global historical climatology network-daily database. *Journal of Atmospheric and Oceanic Technology*, 29(7):897–910, 2012.

[31] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *Proceedings of the 27th International conference on Very Large Databases*, volume 1, pages 281–290, 2001.

[32] H. Pang, X. Ding, and B. Zheng. Efficient processing of exact top-k queries over disk-resident sorted lists. *Proceedings of the 36th International Conference on Very Large Databases*, 19(3):437–456, 2010.

[33] A. Shanbhag, H. Pirk, and S. Madden. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1557–1570. ACM, 2018.

[34] Y. Tao, X. Xiao, and J. Pei. Efficient skyline and top-k retrieval in subspaces. *IEEE Transactions on Knowledge and Data Engineering*, 19(8):1072–1088, 2007.

[35] S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira. Posting list intersection on multicore architectures. In *Proceedings of the 34th International Conference on Research and Development in Information Retrieval*, pages 963–972. ACM, 2011.

[36] S. Tatikonda, F. Junqueira, B. B. Cambazoglu, and V. Plachouras. On efficient posting list intersection with multicore processors. In *Proceedings of the 32nd International Conference on Research and Development in Information Retrieval*, pages 738–739. ACM, 2009.

[37] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the 2008 International Conference on Management of Data*, pages 227–238. ACM, 2008.

[38] M. Xie, L. V. Lakshmanan, and P. T. Wood. Efficient top-k query answering using cached views. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 489–500. ACM, 2013.

[39] D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In *Proceedings of the 32nd international conference on Very Large Databases*, pages 235–246. VLDB Endowment, 2006.

[40] J.-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. In *Proceedings of the 38th International Conference on Research and Development in Information Retrieval*, pages 63–72. ACM, 2015.

[41] S. Zhang, C. Sun, and Z. He. Listmerge: Accelerating top-k aggregation queries over large number of lists. In *International Conference on Database Systems for Advanced Applications*, pages 67–81. Springer, 2016.

[42] V. Zois. Top-k selection. https://github.com/vzois/TopK.

[43] L. Zou and L. Chen. Dominant graph: An efficient indexing structure to answer top-k queries. In *Proceedings of the 24th International Conference on Data Engineering*, pages 536–545. IEEE, 2008.

[44] L. Zou and L. Chen. Pareto-based dominant graph: An efficient indexing structure to answer top-k queries. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):727–741, 2011.