UNIVERSITY OF CALIFORNIA
RIVERSIDE

Complex Query Operators on Modern Parallel Architectures

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Vasileios Zois

December 2019

The Dissertation of Vasileios Zois is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to express my sincere gratitude to my advisors Dr. Vassilis J. Tsotras and Dr. Walid A. Najjar for their support, patience, and encouragement to explore new research ideas. Besides my advisors, I would like also to thank the rest of my thesis committee: Dr. Vagelis Papalexakis and Dr. Daniel Wong.

I would also like to thank UPMEM for providing the SDK and related simulation tools to develop and evaluate the associated algorithms. Special thanks to Divya Gupta, Jean-Francois Roy, and David Furodet for their help in understanding UPMEM's tool and corresponding architecture.

My sincere thanks and gratitude to my family who provided invaluable support, encouragement and advise during the whole PhD journey. I would like to thank my parents Dr. Dimitrios Zois, and Dr. Polyxeni Stathopoulou for their unwavering belief in me. To my sister, Dr. Daphney-Stavroula Zois who offered countless days of laughter. Thank you for your unconditional love and understanding. Above all, i would like to thank my wife (soon to be Dr.) Christina Pavlopoulou for her constant love and support, for keeping my sane and motivated all these past few years. Thank you for giving me a place to stand on and move the world.

The text of this dissertation, in part or in full, is a reprint of the material as it appers in Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT18) (Massively parallel skyline computation for processing-in-memory architectures, Limassol, Cyprus, November 1-4, 2018), Proceedings of the 10th International Workshop on Accelerating Analytics and Data Management Systems Using

Modern Processor and Storage Architectures (ADMS 2019) (GPU Accelerated Top-K Selection With Efficient Early Stopping, Los Angeles, USA, Monday, August 26, 2019) and Proceedings of the 46th international Conference on Very Large Databases (VLDB 2020) (Efficient Main-Memory Top-K Selection For Multicore Architectures, Tokyo, Japan, August 31 - September 4, 2020). The co-author (Dr. Vassilis Tsotras) listed in these publications directed and superivised the research which forms the basis for this dissertation. The co-authors (Walid A. Najjar, Divya Gupta and Jean-Francois Roy) reviewed and co-wrote in part the aforementioned papers. This research was partially supported by NSF grants IIS:1447826 and IIS:1527984.

To my loving wife Christina for her eternal devotion, support, and encouragement.

To my parents and sister who inspired me to pursue and achieve more.

ABSTRACT OF THE DISSERTATION

Complex Query Operators on Modern Parallel Architectures

by

Vasileios Zois

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2019
Dr. Vassilis J. Tsotras, Chairperson

Identifying interesting objects from a large data collection is a fundamental problem for multi-criteria decision making applications. In Relational Database Management Systems (RDBMS), the most popular complex query operators used to solve this type of problem are the Top-K selection operator and the Skyline operator. Top-K selection is tasked with retrieving the k-highest ranking tuples from a given relation, as determined by a user-defined aggregation function. Skyline selection retrieves those tuples with attributes offering (pareto) optimal trade-offs in a given relation. Efficient Top-K query processing entails minimizing tuple evaluations by utilizing elaborate processing schemes combined with sophisticated data structures that enable early termination. Skyline query evaluation involves supporting processing strategies which are geared towards early termination and incomparable tuple pruning.

The rapid increase in memory capacity and decreasing costs have been the main drivers behind the development of main-memory database systems. Although the act of migrating query processing in-memory has created many opportunities to improve the associated query latency, attaining such improvements has been very challenging due to the growing gap between processor and main memory speeds. Addressing this limitation has been made easier by the rapid proliferation of multi-core and many-core architectures. However, their utilization in real systems has been hindered by the lack of suitable parallel algorithms that focus on algorithmic efficiency.

In this thesis, we study in depth the Top-K and Skyline selection operators, in the context of emerging parallel architectures. Our ultimate goal is to provide practical guidelines for developing work-efficient algorithms suitable for parallel main memory processing. We concentrate on multi-core (CPU), many-core (GPU), and processing-in-memory architectures (PIM), developing solutions optimized for high throughput and low latency. The first part of this thesis focuses on Top-K selection, presenting the specific details of early termination algorithms that we developed specifically for parallel architectures and various types of accelerators (i.e. GPU, PIM). The second part of this thesis, concentrates on Skyline selection and the development of a massively parallel load balanced algorithm for PIM architectures. Our work consolidates performance results across different parallel architectures using synthetic and real data on variable query parameters and distributions for both of the aforementioned problems. The experimental results demonstrate several orders of magnitude better throughput and query latency, thus validating the effectiveness of our proposed solutions for the Top-K and Skyline selection operators.

Contents

List of Figures

List of Tables

Chapter 1

Introduction & Motivation

The rapid proliferation of decision support systems combined with the increasing prevalence of multi-dimensional data has compelled researchers to develop various schemes for extracting useful data insights. Research in that area concentrates on enabling support for decision making by utilizing different analytical methods such as mathematical models, statistical analysis or other related data mining techniques. In the context of Relational Database Management Systems (RDBMS), methods that support discovery of hidden data patterns are crucial to the operational characteristics of many real life applications. In fact, bridging the gap between data mining and data management has been an elusive goal of the research community for the past several decades. This happens because the relational data layout hinders the efficient adaptation of relevant data mining algorithms. Therefore, it has been the ultimate goal of the community to identify those methods that work best with modern RDBMS. There has been a lot of work devoted on developing complex query operators which include but are not limited to Top-K [34], Skyline queries [17], Diversification [26] and Regret Minimization [22] queries. All of these methods aim at identifying a good representative subset of tuples which fit the user's preference or query parameters. Top-K and Skyline selection queries are amongst the most prominent solutions due to their simplicity and robustness when operating on variable data distributions.

Top-K queries are a crucial component for a wide range of real life applications which span the areas of information retrieval [35], database systems [50], sensor networks [84], spatial data analysis [24], and data stream management systems [38]. Processing Top-K queries involves ranking a large collection of tuples/objects utilizing a user-defined aggregation function combined with some preference vector and retrieving k of those that attain the highest score. Many different instances of the Top-K problem exist, including but not limited to Top-K selection [34, 65, 41, 48], Top-K aggregate [63], Top-K join [73, 49] and Top-K dominating queries [42]. Every such variant strives to attain the same goal which is to minimize tuple/object evaluations while maintaining efficient data access. In Top-K selection, these competing goals are achieved using intricate indexing techniques and auxiliary information which are intended to efficiently guide processing [41, 59, 11] and reduce the candidate maintenance cost [65]. In the context of database management systems (DBMS), the most prominent solutions fall under three categories, namely: (i) sorted-lists [34], (ii) materialized views [48], and (iii) layered-based methods (convex-hull [19], skyline [59]). 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 has been that using the main memory buffer pool to store and operate on auxiliary information is less costly than performing a full data scan.

Top-K selection compels the user to provide an aggregation function, a precondition that is not always easy to satisfy given that the user might not have a specific preference or is keen to explore the data for hidden insights. Therefore, there is no strict way of ranking the corresponding tuples in a manner that is equatable to a single aggregation function. Skyline selection was invented to address the aforementioned issue since it does not require from the user to provide a specific aggregation function. Evaluating skyline queries involves discovering a set of tuples offering (pareto) optimal trade-offs compared to every other tuple outside the skyline set. Discovering the skyline set from a given collection of items is the same as finding the Pareto optimal front. The term skyline (inspired by the Manhattan skyline example) has been introduced in [17] and has since been used extensively from the database community for a variety of large scale data processing applications including but not limited to data exploration [20], database preference queries [7], route planning [57], web-service support [97], web information [87] and user recommendation systems [12]. Database management systems are optimized on the basis of efficient per object access. Therefore, skyline queries where designed to leverage on the notion of pairwise Pareto dominance between objects/points in order to identify those points not dominated by any other point in a given dataset. A point p dominates another point q, if it is equal or better on all dimensions and there exists at least one dimension for which it is strictly better. In order to identify the dominance relationship between two points, it is common to perform a Dominance Test (DT) [21] by comparing all their attributes/dimensions. The fact that every tuple in the skyline set dominates all the others outside it, guarantees that the Top-1 result for every possible monotone aggregation function will appear in that set.

Therefore, skyline queries are suitable for scenarios in which the user does not have a specific preference function.

The apparent decrease in DRAM cost, coupled with granted high capacity and bandwidth guarantees, resulted in the migration of processing from disk to main memory and resulted in the proliferation of in-memory database systems. On the other hand, the exponential increase in data volume observed over the past decades combined with the widening gap between processor and memory speed has negated any benefits of that migration. This stimulated the development of different types of multi-core and many-core architectures having the exclusive purpose of improving processing throughput and masking data access latency. Modern processors (i.e. CPUs) leverage the integration of many compute cores and deep cache hierarchies on a single chip to mitigate the effects of processing large dataset. Many core architectures (i.e. GPUs, PNM) rely on thousands of processing cores to mask data access latency during processing and specialized memory (i.e. GDDR) increase the available memory bandwidth. The pervasiveness of such parallel architectures has contributed towards a growing need to redesign established query operators, such Top-K and Skyline selection, in order for them to execute efficiently in this new parallel environment.

In this thesis, we develop algorithms suitable for in-memory processing using parallel and massively parallel architectures to solve the Top-K and Skyline problems. We concentrate on multi-core CPUs, many-core GPUs and Processing-In-Memory (PIM) architectures. Our goal is to develop solutions that aim at achieving high parallelism, load balancing and respectable work-efficiency. We achieve these goals by (1) developing new strategies used to prune irrelevant items during processing, and (2) adopting previously proposed techniques to the aforementioned processing environments.

The rest of this thesis is organized as follows; In Chapter 2, we provide a formal definition of both the Top-K and Skyline operators including a discussion about established practices designed to enable efficient processing of these operators. Chapter 3, discusses the challenges associated with main-memory Top-K selection for multi-core architectures and develop an algorithm suitable for improving significantly query latency, and throughput for query batches while minimizing the number of tuple evaluations. Chapter 4, concentrates on how to adopt the techniques developed for in-memory CPU-based Top-K query evaluation on modern GPUs. In addition, we discuss how to take advantage of data caching on the GPU side that in order to improve query latency when the device memory capacity is low. Chapter 5, introduces the properties of an upcoming parallel architecture called Processing-In-Memory (PIM). We present a discussion concentrating on challenges and opportunities related to the development of database operators on that environment. In Chapter 6, we concentrate on how to adopt Top-K query evaluation on PIM using either Full Table Evaluation (FTE) or early termination algorithms. Chapter 7, presents a comparative study of previously proposed in-memory solutions for computing the Skyline set and propose a new algorithm to solve this problem on PIM. Finally, Chapter 8 summarizes the lessons learned from developing database operators for data analytics on in-memory parallel architectures, and provide insights on the potential for future work that is related to fusing together other well known operators (i.e. Top-K joins) or using PIM to evaluate them efficiently.

Chapter 2

Background

In this chapter, we present formal definitions for both the Top-K and Skyline problems. Furthermore, we provide a thorough review of the related literature and examine the specific properties as well as associated challenges when implementing either operator in a main memory parallel environment.

2.1 The Top-K Selection Operator

A toy example of Top-K selection is depicted in Figure 2.1. The input relation represents a collection of vehicles and their corresponding properties (i.e. warranty, MPG, price). In that example, the Top-K selection query considers the attributes warranty and MPG, requesting from the system to return the Top-2 vehicle with the highest rank, based on the weights of the provided preference vector. A simple approach of calculating the Top-2 answer will be to scan the complete relation and evaluate every tuple's score. Doing so will provide the correct answer which is depicted in green for our example.

Make	Model	Warranty	MPG	Price
Toyota	Prius	15	50	35000
Ford	Mustang	10	10	22000
BMW	M3	12	14	45000
VW	Jetta	10	30	29000
Volvo	XC60	18	27	32000
Hyundai	Accent	15	25	28000

```
SELECT  C.make, C.model
FROM Cars AS C
ORDER BY (C.warranty · 0.2 + C.mpg · 0.8) DESC
LIMIT 2
```

Figure 2.1: A Top-K selection query example.

2.1.1 Top-K Selection Definition

Let R be a relation consisting of n objects/tuples, each one having d attributes ($o = \{a_0, a_1, ...a_{d-1}\}$) ranging in $(0, 1]$ without loss of generality. Equivalently, R can 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 section 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 aggregation function has been either linear [19, 25, 48] or monotone [25, 48, 90, 100]. We formally define an arbitrary linear function as follows:

$$F(o) = \sum_{i=0}^{m} (w_i \cdot a_i) \tag{2.1}$$

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:

$$\begin{gathered} \textit{if } o_u(a_i) \geq o_v(a_i), \forall i \in [0, d-1] \\ \textit{then } F(o_u) \geq F(o_v) \end{gathered} \tag{2.2}$$

In essence, this means that objects having higher values for all attributes should also rank higher than others with smaller values. This is guaranteed for any linear function when all weights in the preference vector are non-negative. Following the majority of previous work [50], we concentrate mainly on linear monotone aggregation functions.

Well-established methods used to support efficient Top-k query evaluation include caching and managing materialized results (i.e. view-based methods), using combined attribute indexing to reorder and stream only relevant data from memory (i.e. layered-based methods), or balancing random vs sequential accesses on individual attribute indexes (i.e. list-based methods).

2.1.2 List-Based Methods

Fagin et al [34] 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) [39] improves NRA using heuristics to choose the most promising list for evaluation. LARA [65] aims at reducing the cost of maintaining the upper bounds for each seen object and improve candidate pruning. IO-Top-K [11] utilizes selectivity estimators and score predictors to efficiently schedule sorted and random accesses. TBB [75] relies on a pruning mechanism and bloom filters to efficiently process Top-k queries over bucketized sorted lists.

BPA [4] improves TA's stopping threshold by considering attributes seen both under sorted and random access. T2S [41] promotes reordering the database objects based on their first seen position in the sorted lists favoring sequential access for disk-resident data. ListMerge [104] relies on intelligent result merging to efficiently evaluate Top-k queries over large number of sorted lists.

2.1.3 View-Based Methods

PREFER [48] aims at reducing the cost associated with Top-k query processing by effectively managing and updating materialized views in-memory. LPTA [25] employs linear programming to avoid accessing the disk when the combined query attributes appear within overlapping materialized views. LPTA+ [99] aims at reducing the number of solved linear programming problems per query to improve performance. TKAP [44] combines early pruning strategies from list-based methods and materialized views to support Top-k queries on massive data.

2.1.4 Layered-Based Methods

The Onion technique [19] 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) [106, 107] orders objects according to their dominance relationship while utilizing a graph traversal algorithm to evaluate Top-k queries. The partitioned layer algorithm (PLA) [46] relies on convex skyline layering and fixed line partitioning to further improve object pruning. The HL-index [45] is a hybrid method that combines skyline layering and TA ordering within each layer to reduce object evaluations. The Dual Resolution (DL) [59] index suggest relying on skyline layering and the convex skyline properties to improve DG's graph traversal algorithm.

2.1.5 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 [92, 91, 51, 102, 29], while other methods avoid reordering the dataset and try to maximize skipping irrelevant objects during evaluation [30, 35, 28]. 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 thesis, 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.

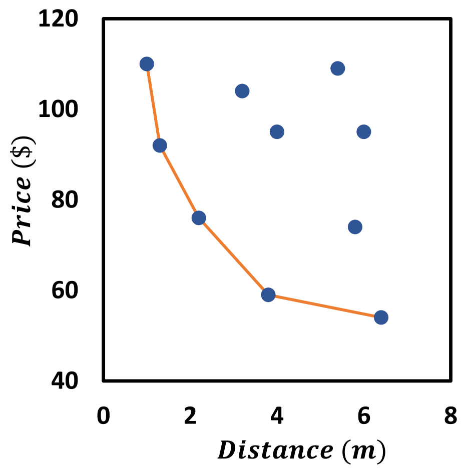
2.2 The Skyline Selection Operator

The skyline computation concentrates on identifying the Pareto front through exploration of a given data collection which cannot be formally represented using a single non-linear equation. A classic example used to demonstrate its importance is picking a hotel, given the hotel's prices and its distance to the beach. Although users prefer affordable hotels, those close to the beach are likely expensive. In this case, the skyline operator would present hotels that are no worse than any other in both price and distance to the beach (Fig. 2.2).

The term skyline (inspired by the Manhattan skyline example) has been introduced in [17] and has since been used extensively from the database community for a variety of large scale data processing applications including but not limited to data exploration [20], database preference queries [7], route planning [57], web-service support [97], web information [87] and user recommendation systems [12].

Skyline queries differ from conventional Pareto analysis which aims at discovering all or some of the Pareto optimal solutions without enumerating all of the potentially unbounded number of feasible solutions from a given collection of linear or non-linear equations and the user-provided constrains. Multi-objective optimization has been applied extensively for a number of different applications including but not limited to hardware design space exploration (DSE) [74, 85, 13], high level synthesis [101], compiler optimization ex-

Hotel	Distance(m)	Price ($)
Blue Waters	1.3	92
Empire Hotel	3.8	59
Pine Inn	6.4	54
Sunny Hotel	4	95
Sandy Beach	1	110
Holiday Inn	2.2	76
Bright Motel	6	95
Palms Hotel	3.2	104
Lakeview Inn	5.8	74
Park Hotel	5.4	109



```
SELECT  H.distance, H.price
FROM Hotels AS H
SKYLINE OF H.warranty [MIN], H.price [MIN]
```

Figure 2.2: A Skyline selection query example.

ploration [52, 47], power management [14], portfolio optimization [79]. In each case, the proposed solutions leverage on either numerical methods (e.g. linear regression), evolutionary algorithms or heuristics [33] to identify Pareto optimal solutions.

Database management systems are optimized on the basis of efficient per object access. Therefore, skyline queries where designed to leverage on the notion of pairwise Pareto dominance between objects/points in order to identify those points not dominated by any other point in a given dataset. A point p dominates another point q, if it is equal or better on all dimensions and there exists at least one dimension for which it is strictly better (see Section 2.2.1). In order to identify the dominance relationship between two points, it is common to perform a Dominance Test (DT) [21] by comparing all their attributes/dimensions.

When the input dataset is large and multidimensional, computing the skyline is costly, since in theory each unprocessed point needs to be compared against all the existing skyline points. In order to reduce this cost, most sequential algorithms rely on established optimization techniques such as in-order processing [23] and space partitioning [17], both of which aim at reducing the total number of point-to-point comparisons.

2.2.1 Skyline Selection Definition

Let D be a set of d-dimensional points such that $p \in D$ and $p[i] \in \mathbb{R}, \forall i \in [0, d-1]$. The concept of dominance between two points is used to identify those that are part of the skyline set. As mentioned, a point p *dominates* a point q, if it has *"better"* or equal value for all dimensions and there exists at least one dimension where its value is strictly *"better"*. The meaning of *"better"* corresponds to the manner in which we choose to rank the values for each dimension, being smaller or larger, although the ranking should be consistent amongst all dimensions. For this work, we regard smaller values as better, therefore the mathematical definition of dominance becomes:

Dominance: Given $p, q \in D$, p dominates q, written as $p \prec q$ if and only if $\forall i \in [0, d-1]$ $p[i] \leq q[i]$ and $\exists j \in [0, d-1]$ such that $p[j] < q[j]$.

Any point that is not dominated from any other in the dataset, will be part of the skyline set (see Fig. 2.3) and can be identified through a simple comparison called Dominance Test (DT).

Skyline: The skyline S of set D is the collection of points which are not dominated by any other point in the dataset, formally defined as:

$$S = \{\forall p \in D | \nexists q \in \ \ s.t \ q \prec p\}.$$

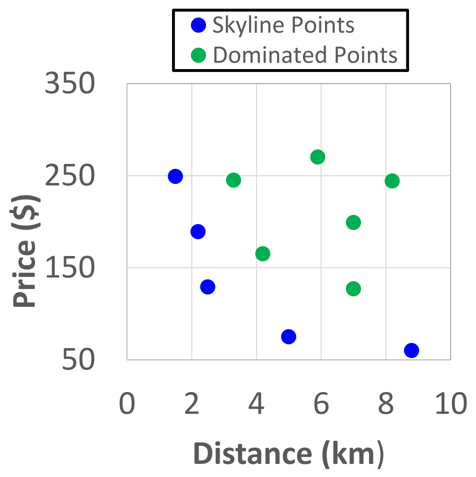


Figure 2.3: Skyline vs Dominated Points Example.

Clearly $S \subseteq D$. The definition of *dominance* acts as the basic building block for designing skyline algorithms. The BNL algorithm relies naïvely on brute force to compute the skyline set. This method is quite inefficient, resulting in $O(n^2)$ DTs and a proportional number of memory fetches. To avoid unnecessary DTs, previous solutions used in-order processing based on a user defined monotone function. It considers all query attributes, reducing the point to a single value that can be used for sorting. Such a function is formally defined as:

Monotone Function: A monotone scoring function F with respect to $\mathbb{R}^d$ takes as input a given point $p \in D$ and maps it to $\mathbb{R}$ using k monotone increasing functions $(f_1, f_2, ...f_k)$. Therefore, for $p \in D$, $F(p) = \sum_{i=1}^{k} f_i(p[i])$.

The ordering guarantees that points which are already determined to be part of the skyline, will not be dominated by any other which are yet to be processed. This effectively reduces the number of DTs by half.

$$\mathbf{p_s} = \underset{p_i \in S}{\operatorname{argmin}} \left\{ \max_{j \in [0, d-1]} \{p_i[j]\} \right\} \tag{2.3}$$

Another important optimization aimed at reducing the total number of DTs uses a so-called stopping point [10] to determine when it is apparent that no other point is going to be added in the skyline. Thus a number of DTs are avoided by stopping early. Each time a new point is added to the skyline, it is checked to see if it can be used as a stopping point. Regardless of the chosen monotone function, we can optimally select that point using the MiniMax [10] update rule depicted in Eq. 2.3.

2.2.2 Skyline Related Work

The skyline operator was first introduced by Borzsony et al. [17], who also proposed a brute-force algorithm known as Block Nested Loop (BNL) to compute it. Sort-Filter-Skyline (SFS) [23] relied on topological sorting to choose a processing order, that maximizes pruning and reduces the overall work associated with computing the skyline set. Related variants such as LESS [36] and SALSA [10] proposed the use of optimizations like pruning while sorting the data or determining when to stop early.

Sort-based solutions are optimized towards maximizing dominance and reducing the overall work by half. However, on certain distributions where the majority of points are incomparable [62], they are proven to be less effective. In contrast, space partitioning strategies [62] have been proven to perform better at identifying incomparability.

The *BSkyTree* [60] algorithm facilitates index-free partitioning by using a single pivot point. This point is calculated iteratively during processing through the use of a heuristic that aims at achieving a balance between maximizing incomparability and dominance. *BSkyTree* is the current state-of-the-art sequential algorithm for computing the skyline regardless of the dataset distribution.

Despite their proven usefulness, previous optimizations cannot be easily adapted on modern parallel platforms. Related research concentrated mainly on developing parallel skyline algorithms that are able to maintain the same level of efficiency as their sequential counterparts. The *PSkyline* algorithm [77] is based on the Branch & Bound Skyline (BBS) and exploits multi-core architectures to improve performance of the sequential BBS. For data distributions that are more challenging to process, it creates large intermediate results that require merging which causes a noticeable drop in performance. *BSkyTree-P* [60] is a parallel variant of the regular *BSkyTree* algorithm. Although, generally more robust on challenging data distributions, *BSkyTree-P* is also severely restricted during the merging of intermediate results, an operation that entails lower parallelism.

The current state-of-the-art multi-core algorithm is *Hybrid* [21] and is based on blocked processing, an idea used extensively for a variety of CPU-based applications to achieve good cache locality. Sorting based on a monotone function is used to reduce the total workload by half. For more challenging distributions, the algorithm employs a simple space partitioning mechanism, using cheap filter tests which effectively reduce the cost for identifying incomparable points. *Hybrid* is specifically optimized for multi-core platforms, the performance of which depends heavily on cache size and memory bandwidth. Data distributions that generate an arbitrarily large skyline limit processing performance. Therefore, multi-core CPUs are limited when it comes to large scale skyline computation.

Accelerators present the most popular solution when dealing with data parallel applications such as computing the skyline set. Previous solutions include using GPUs [15] or FPGAs [98]. The FPGA solution relies on streaming to implement a variant of BNL.

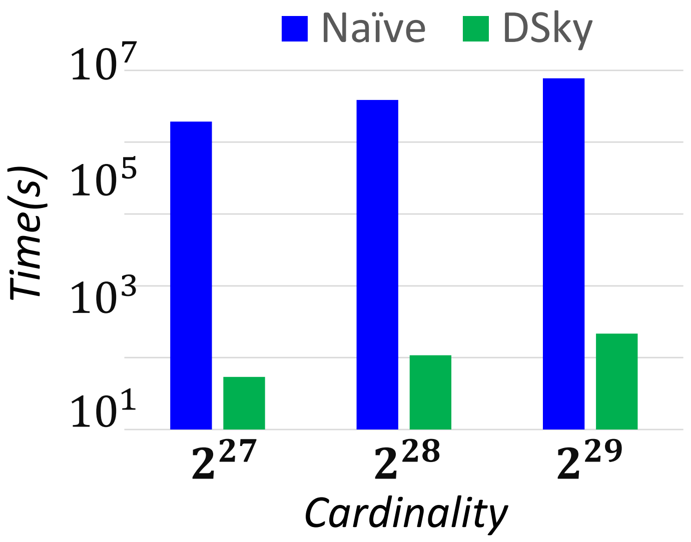


Figure 2.4: Runtime snapshot for 16 dimension skyline.

Although, it showcases better performance compared to an equivalent software solution, it is far from the efficiency achieved by *Hybrid*. On GPUs, the current state-of-the-art algorithm is *SkyAlign* [15]; it aims at achieving work-efficiency through the use of a data structure that closely resembles a quad tree. *SkyAlign* strives towards reducing the overall workload at the expense of lower throughput that is caused by excessive thread divergence. Furthermore, load balancing issues and irregular data accesses coupled with restrictions in memory size and bandwidth result in significant performance degradation when processing large dataset.

Our solution is based on PIM architectures which relies on integrating a large collection of processors in DRAM. This concept offers higher bandwidth, lower latency and massive parallelism. In short, it is perfectly tailored for computing the skyline, a data intensive application. In UPMEM's PIM architecture, each processor is isolated having access only to their local memory. This restriction makes previously proposed parallel solutions and their optimizations nontrivial to apply. In fact, our initial attempts to directly apply optimizations used in the state-of-the-art CPU and GPU solutions on UPMEM's PIM architecture, resulted in noticeable inferior performance (Figure 2.4). We attribute this behavior to low parallelism, unbalanced workload assignment and a high communication cost. In Chapter 7, we discuss these challenges in detail and describe how to design a parallel skyline algorithm suitable for this newly introduced architecture.

Chapter 3

Main-Memory Top-K Selection For Multi-core Architectures

3.1 Introduction

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 multi-threading 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 [34, 4, 11] 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 [45] or having to maintain too many candidates [65, 59], the end result of which is an increased number of memory accesses.

In this chapter, we study the related literature in order to discover suitable practices for efficient parallel main memory Top-k selection utilizing multi-core architectures. In order to identify these methods, we establish a new measure of algorithmic efficiency called *rank uncertainty*. As opposed to the number of object evaluations (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 3.2) 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 variations, thus creating the 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 3.2.2) 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 3.3.2) and data layering (Section 3.4.1) algorithms in a parallel environment (creating the VTA and SLA approaches).

- We develop a new solution (PTA) that relies on angle space partitioning (Section 3.4.3) combined with data reordering to 19 improve algorithmic efficiency while also maintaining low *rank uncertainty*.

This chapter is organized as follows. In Section 3.2.1 three parallel Top-k models are presented and Section 3.2.2 the concept of *rank uncertainty* is described. Sections 3.3 and 3.4 propose guidelines for implementing optimized algorithms for scalar, SIMD, and multi-threaded execution. Section 3.5 concludes with extensive experiments and result discussion.

3.2 Parallel Top-K Queries

Considering that previous work has focused mostly on disk-based solutions, it is nontrivial to identify which practices are best for parallel in-memory environments. In this section, we attempt to identify such practices that provide satisfactory parallelism, and efficient in-memory processing without sacrificing algorithmic efficiency.

3.2.1 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 multi-threading 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, respec-

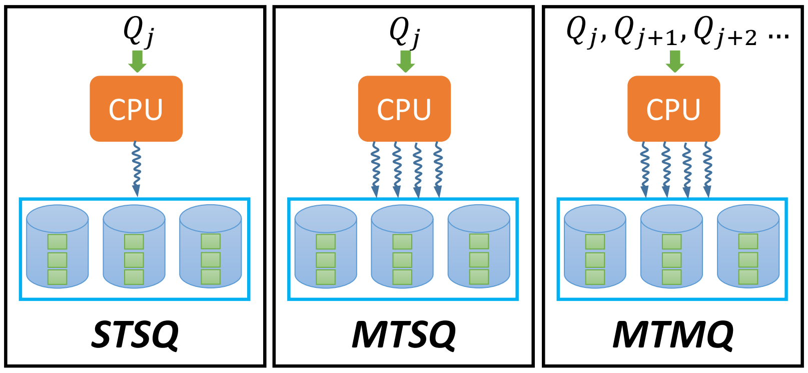


Figure 3.1: Parallel Top-k evaluation models

tively. In addition, they can be intertwined together to create three separate parallel Top-k query evaluation models as indicated in Fig. 3.1; 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 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.

3.2.2 Rank Uncertainty

Existing Top-k algorithms [34, 11, 65, 4, 45, 41] 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

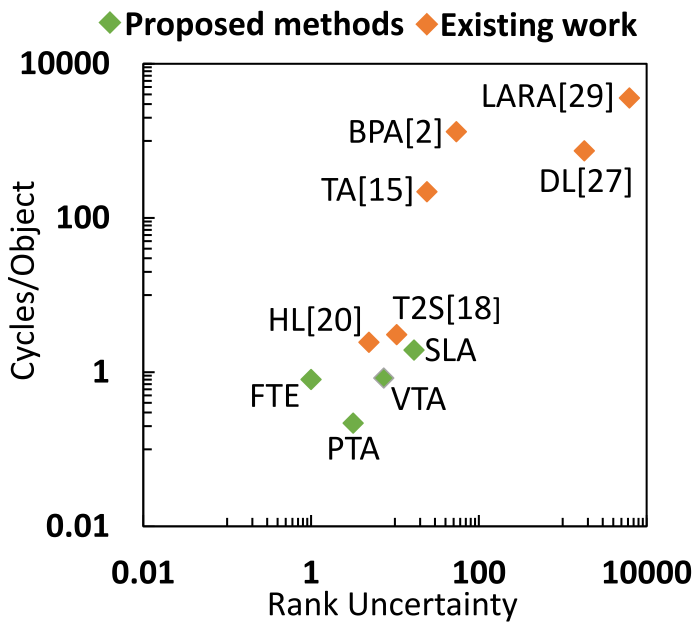


Figure 3.2: Cycles/Object vs Rank Uncertainty.

(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 uncertainty* for different threshold-based solutions (Fig. 3.2). 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 3.2 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

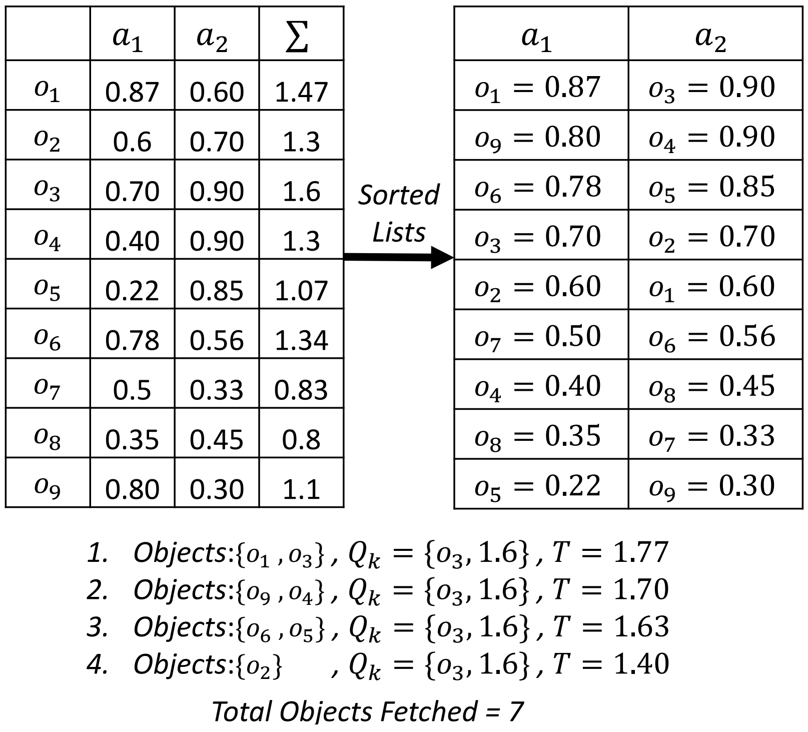
[1]Reported using *mem_uops_retired.all_loads.*

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.

3.3 Single-Thread Top-K Selection

In this section, we review TA's execution and present the concept of round robin reordering of the base relation.

TA operates on a collection of sorted-lists which are maintained using indexes (i.e. B-trees). The main algorithm retrieves the objects seen at each list level in round robin order. For example, in Fig. 3.3 objects o_1 and o_3 are accessed at the first iteration. The score of each one is computed by random access to the remaining lists. Only the k-highest ranked objects are retained using a priority queue (i.e. Q_k). Processing of new objects terminates when k objects exist in the queue and their minimum score is $\geq$ than the threshold of the given list level. In our example, o_3 remains in the queue until the 4-th list level is processed.



	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

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*:$\{o_2\}$, $Q_k = \{o_3, 1.6\}$, $T = 1.40$

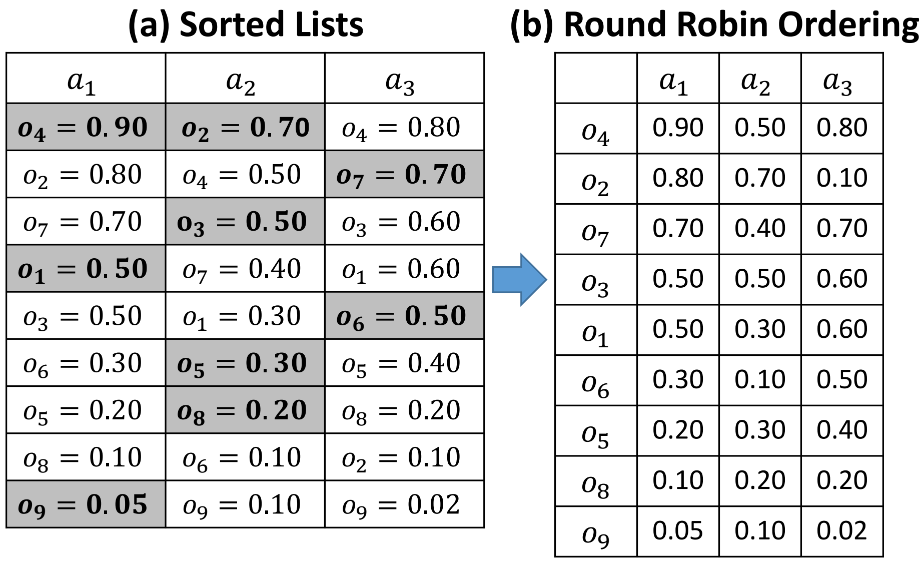
Total Objects Fetched = 7

Figure 3.3: TA execution and data access example

At that point, we can safely stop processing relying on the fact that there are no objects with score higher than 1.40 after level 4, as indicated by the threshold value.

In addition to incurring too many random accesses, TA requires keeping track of seen objects to avoid reevaluation. This results in cache pollution as the number of evaluated objects grows. For every evaluated object, TA requires $d-1$ arithmetic operations and may result to $d+2$ random memory references. 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 a solution called Vectorized Threshold Algorithm (VTA). This method relies on static object reordering as proposed by Han et al. [41]. In that work, the authors choose to reorder the base relation according to the



(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 3.4: Round robin reordering example

round-robin access indicated by the corresponding sorted-lists. Our implementation avoids using any auxiliary information such as pre-materialized results, since we try to minimize the overall data footprint. It simply relies on SIMD vectorization to accelerate processing and improve bandwidth utilization. Moreover, we develop our own easily maintainable data layout, called Threshold Block Layout (TBL; to be discussed next). This layout clusters objects together according to their first seen position and assigns them a threshold using the sorted-lists. We develop VTA in order to establish a baseline that incorporates both system specific and algorithmic optimization. This allows us to identify the right practices for developing and evaluating multithreaded methods (i.e. SLA and PTA) that promote algorithmic efficiency.

3.3.1 TBL List and TBL Node

Fig. 3.4 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 $p_i\left[j\right] \in [0, n-1]$. This position is calculated based on the following formula:

$$\widetilde{p_i} = \underset{\forall j \in [0, d-1]}{\operatorname{argmin}} \{p_i\left[j\right]\} \tag{3.1}$$

During query evaluation, not all threshold calculations are necessary since they do not contribute towards satisfying the stopping conditions. For example in Fig. 3.3, 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.

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. 3.5 (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

Algorithm 1 Build TBL List

D = Input dataset.

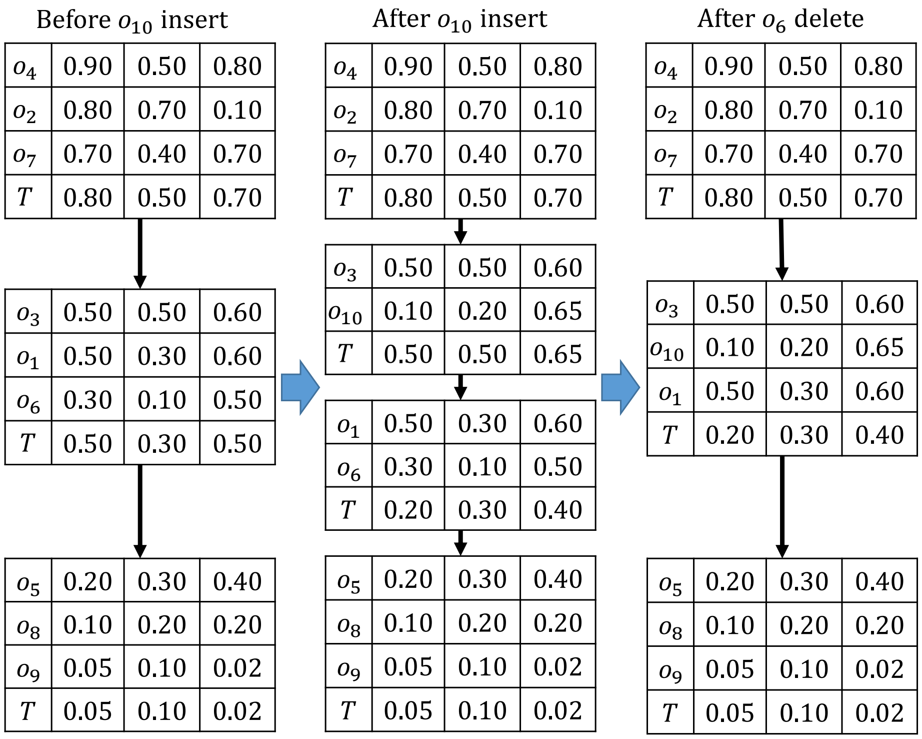
N_{TBL} = TBL node size.

```
1: for $c$ in $D$ do
2:     $c = \text{sort(c)}$
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
```

shown in Fig. 3.4. 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. 3.4 (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 they roughly approximate those seen



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 3.5: TBL list insert-delete example.

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. 3.5 (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. 3.5 (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
```

Discussion: The TBL list concept resembles that of a clustered index similar to a B+tree, having the additional requirement for keeping track of stopping thresholds. It can thus be easily integrated within a relational DBMS. The TBL list was designed explicitly to improve Top-k selection performance, which is inline with related work [34, 65, 41, 44, 80] focusing solely on selection. The idea could be extended on rank joins [50] and possibly combined with other operators (i.e. group-by, join) , but this is out of this thesis' scope.

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
```

3.3.2 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 enable 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).

3.3.3 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.

3.4 Multi-threaded 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 [45], while the second method (PTA) follows a new direction, utilizing angle space partitioning to optimally partition the data for processing.

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
```

3.4.1 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. [45], it presents the first attempt to enable parallel processing and vectorization using static reordering of each layer. Related solutions using skyline layering [59, 107] 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 3.2.2) 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 [21], 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 summarizes SLA's execution steps. SLA processes only the first k layers (Line 2) since according to Chang et al. [19] 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

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 $merge(Q_k)$
```

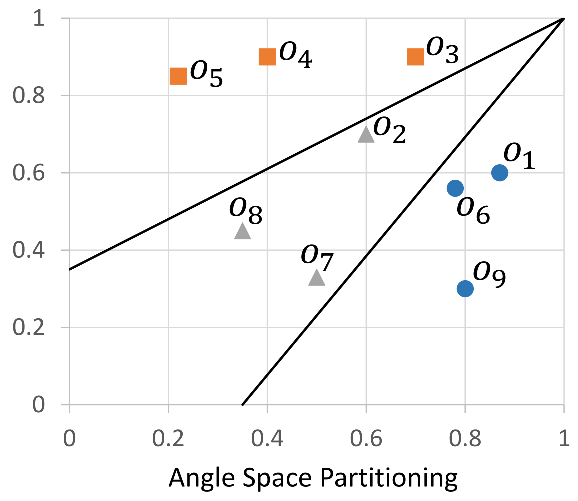
and all threads have completed their evaluations, *omp_barrier* is used to synchronize processing (Line 10). When the accrued number of objects from all queues are $\geq k$ and their minimum scored object is $\geq T$ processing terminates for the given layer (Line 11). Finally, when all k layers have been processed, the individual queues are merged together before returning the Top-k result (Line 14).

3.4.2 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).

$$\begin{aligned} tan(\phi_1) &= \frac{\sqrt{(\tilde{A}_d)^2 + (\tilde{A}_{d-1})^2 ... + (\tilde{A}_2)^2}}{\tilde{A}_1} \\ &... \\ tan(\phi_{d-2}) &= \frac{\sqrt{(\tilde{A}_d)^2 + (\tilde{A}_{d-1})^2}}{\tilde{A}_{d-2}} \\ tan(\phi_{d-1}) &= \frac{\tilde{A}_d}{\tilde{A}_{d-1}} \end{aligned} \tag{3.2}$$

PTA utilizes Angle Space Partitioning (ASP), a strategy first proposed in [96] 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 only necessitates partitioning



	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 3.6: ASP on objects from Fig 3.3.

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.

3.4.3 Angle Space Partitioning Overview

ASP maps each multidimensional object from cartesian space to hyperspherical space using the equations shown in 3.2. Commonly Top-k queries retrieve the Top-k highest ranked objects, hence the corresponding equations are applied on attributes $\tilde{A}_i = |A_i - \alpha|$ assuming the given collection (A_i) is in range $[0, \alpha]$ Due to geometric symmetry, this transformation is equivalent to calculating the angles defined opposite from the origin as shown in Figure 3.6. We partition the data using grid partitioning over the $d-1$ space defined

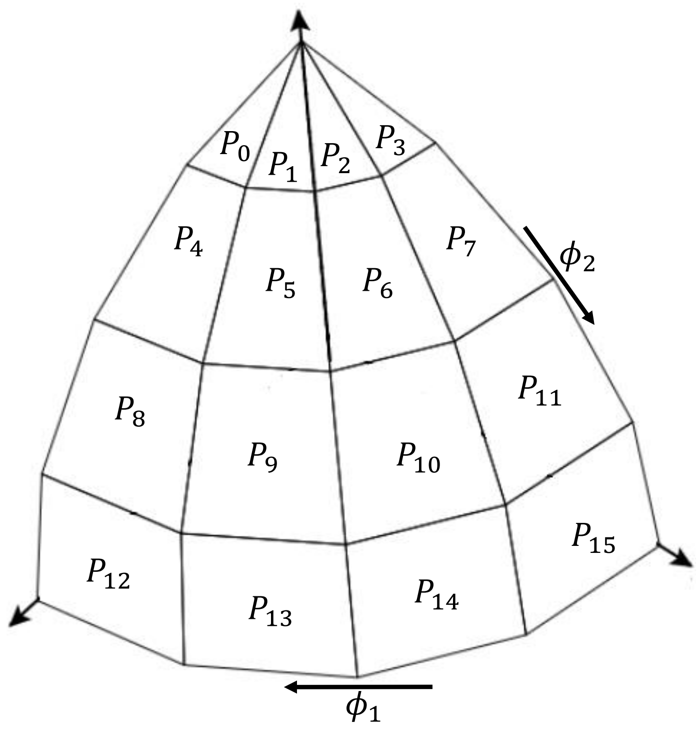


Figure 3.7: Grid partitioning on a hypersphere.

by the corresponding angular coordinates. In effect, this leads to grouping together objects that are increasingly correlated as the angle of the partition shrinks (see Figure 3.7). 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. For each partition, we build a separate TBL list following the process described in the previous sections.

ASP performs well when combined with any TA style optimization (see Section 3.4.5). This happens because the data are partitioned around an imaginary trend line as indicated by Figure 3.6. For list-based methods, this contributes towards discovering an optimal local ordering for every partition independently of any user-defined monotone function. Therefore, in theory each partition may require as little as k evaluations to discover the highest ranked objects. For example in Figure 3.6 even after reordering the data, a Top-1 query will evaluate only a single object per partition before stopping for any preference vector. Compared that to TA's and VTA's performance, we achieve at least two-fold reduction in the total number of object evaluations. In addition, PTA enables parallel evaluation without reducing algorithmic efficiency.

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 $merge(Q_k)$
```

Typically Top-k query evaluation involves only a subset of all attributes. In order to achieve the best possible algorithmic efficiency, ASP should be used only on the query attributes. However, our extensive experimentation showed that the associated processing overhead in terms of object evaluations is negligible. In the worst case, it is roughly equal to the total evaluations performed by the chosen TA style method without applying ASP. In that scenario, we still remain algorithmic efficient while evaluating queries in parallel.

3.4.4 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.

3.4.5 PTA Complexity Estimation

In this section, we demonstrate that leveraging angle space partitioning yields significant improvements over the number of object evaluations. This is apparent for any Top-k processing method when the stopping threshold is reduced. Hence, intelligent parti-

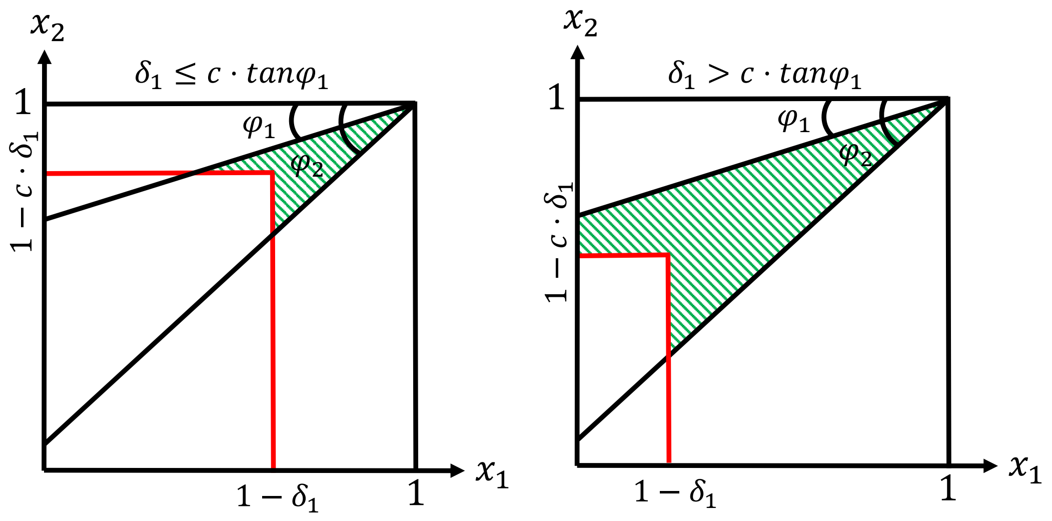


Figure 3.8: Processed areas for varying δ_i using ASP.

tioning is important for improving *rank uncertainty*, and can be used in combination with other optimizations to yield proportional improvements.

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 $\tau = (1 - \delta_1, 1 - \delta_2, \dots, 1 - \delta_m)$ ($\delta_i \in [0, 1]$) 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 $E_{TA}(t) \leq n \cdot [1 - \prod_{i=1}^{m} (1 - \delta_i)]$. 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 [34, 65, 41]. Hence, any strategy geared towards limiting such growth could be used as well to improve performance for that class of algorithms.

Let us consider the 2D case of ASP where $\delta_2 = c \cdot \delta_1, c \in [0,1]$ for simplicity. Fig. 3.8 presents the case where $0 \leq \phi_1 < \phi_2 \leq \frac{\pi}{4}$. 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 3.3, 3.4 where $\delta_1 \leq$ or $>$ to $c \cdot tan(\phi_1)$, respectively.

$$A_1 = \frac{\delta_1^2}{2} \cdot \left(tan\phi_2 - tan\phi_1 + \left(1 - \frac{1}{c \cdot tan\phi_1}\right)^2 \cdot tan\phi_1 \right) \tag{3.3}$$

$$A_2 = 0.5 \cdot \left(tan\phi_2 - tan\phi_1 - (1-\delta_1)^2 \cdot tan\phi_2 \right) \tag{3.4}$$

Fig. 3.9 presents the projected (utilizing Equations 3.3, 3.4) 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. 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

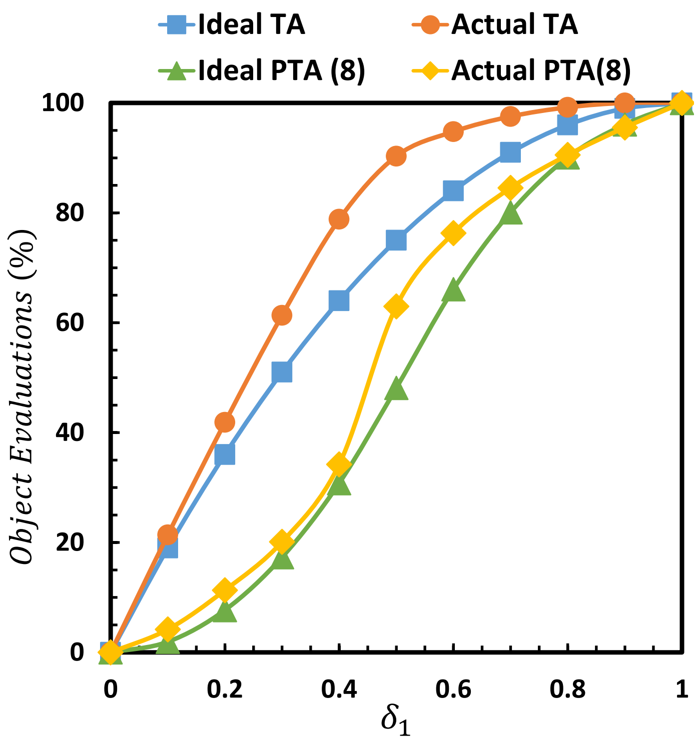


Figure 3.9: Projected vs actual object evaluations.

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.

3.5 Experimental Environment

In this section, we provide a thorough performance evaluation of the proposed solutions under different execution scenarios and processing models as indicated from Section 3.2.1. Despite the wide range of Top-*k* algorithms, there is no comprehensive study comparing the different algorithmic categories. In addition, the related literature concentrates mainly on disk-based systems. Hence, we provide a detailed experimental evaluation and comparison against hardware optimized algorithms (Full Table Evaluation (FTE)), list-based solutions optimized for random-access (TA [34]) or sorted-access (LARA [65]), and layered-based solutions geared for efficient blocked access (HL [45]) or high algorithmic efficiency (DL [107, 59]). Note that our work adopts the best practices established from the aforementioned previous work, concentrating on fine tuning it for the underlying hardware while maintain similar or better work-efficiency guarantees.

3.5.1 System Specification

All our experiments were conducted on a two socket 2.30 GHz Intel Xeon E5-2650 CPU with 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 enabling with O3 optimization flag enabled and the OpenMP framework to enable multi-threaded execution. Unless stated otherwise, all multi-threaded measurements where acquired for 16 threads. Our code is publicly available in Github [105].

3.5.2 Dataset, Query Format & Metrics

We conducted experiments using both real and synthetic data. Overall, the performance characteristics of the developed algorithms do not change between normalized

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

Table 3.1: Experimental parameters.

and regular values. For the experimental results on synthetic data we use normalized values. In contrast, the real dataset measurements where acquired using the regular values. Unless otherwise stated, the parameters of our experiments are summarized in Table 3.1. Similar to previous work [45, 107, 65], our synthetic data follow a uniform distribution and were created using the standard dataset generator from [17]. The real dataset consist of temperature measurements acquired from NOAA [66]. We gathered 524 million objects each one having 8 attributes which correspond to pairs of values indicating the maximum and minimum temperature for one day. For this reason, each object corresponds to the temperature variations for 4 consecutive days. 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 $k = 16$, $n = 2^{28}$ using our overall best performing algorithms, mainly VTA and PTA.

We evaluate the different processing models described in Section 3.2.1 utilizing the relevant implementations, mainly $STSQ$ scalar and SIMD versions, $MTSQ$ multithreaded version including SIMD, $MTMQ$ SIMD version with multiple queries per thread. Our ex-

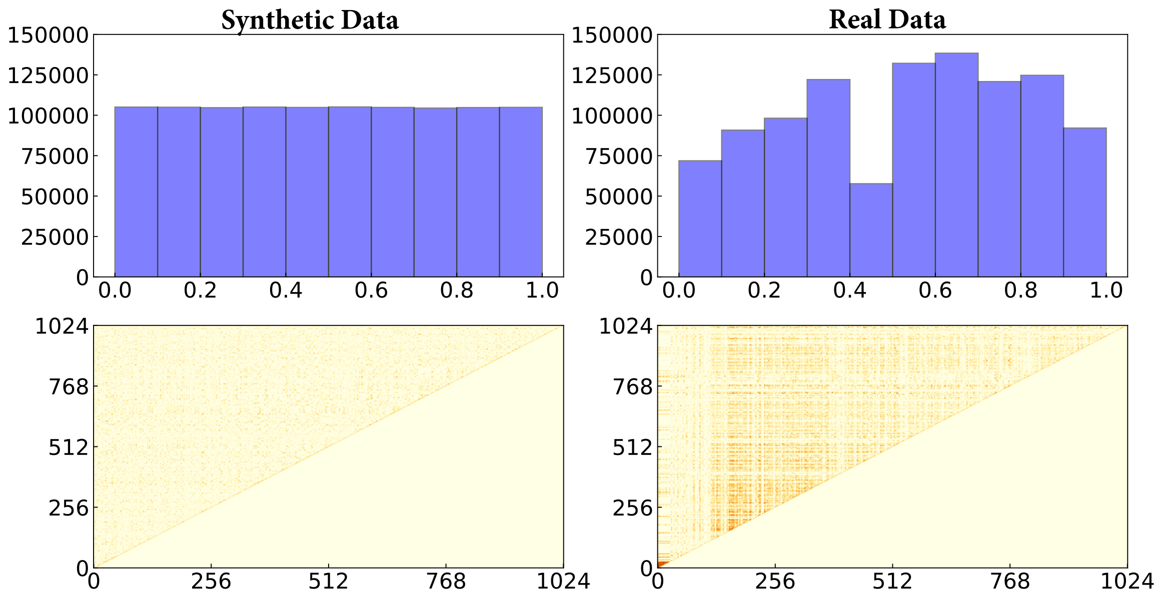


Figure 3.10: Distribution properties of synthetic vs real data (top: histogram, bottom: correlation matrix).

periments concentrate on measuring the initialization cost, throughput (queries per second), single query latency (wall clock time) and average query latency (for a batch of queries).

In Figure 3.10, 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 col-

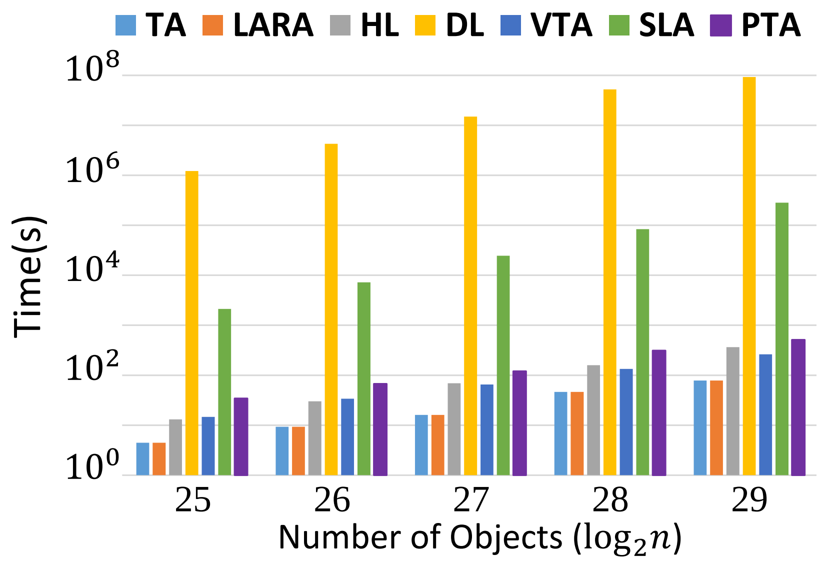


Figure 3.11: Initialization cost comparison.

lections with zero or negative linear correlation, especially for dataset that contain multiple attributes.

3.6 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.

3.6.1 Initialization Cost

In Fig. 3.11, the highest initialization cost is incurred by methods that require calculating the skyline set to construct 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.

3.6.2 TBL Node Size

Figure 3.12 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).

3.6.3 Varying Preference Vectors

Figure 3.13 presents the measured object evaluations for the preference vectors of Table 3.2. VTA exhibits little variation in performance, while PTA occasionally performs

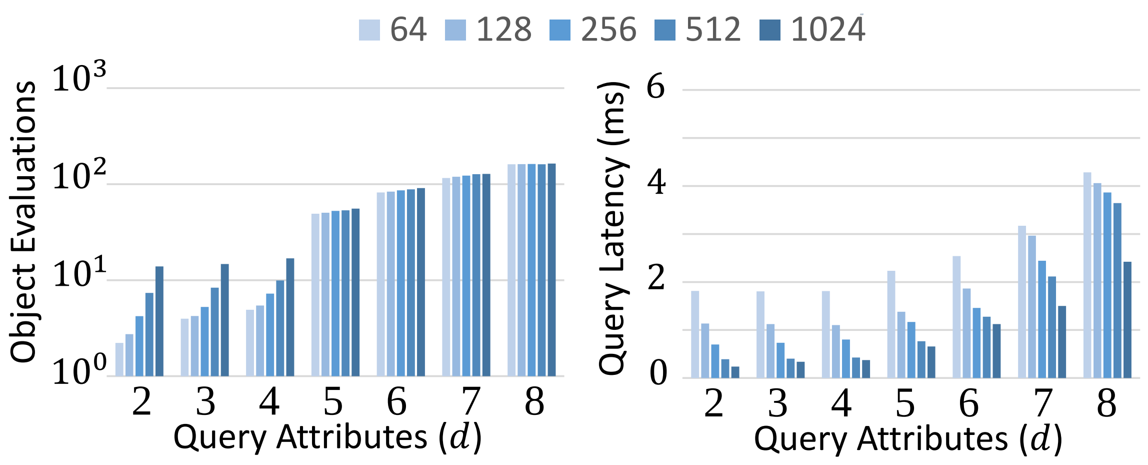


Figure 3.12: Block size vs latency-object evaluations.

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 [11] 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).

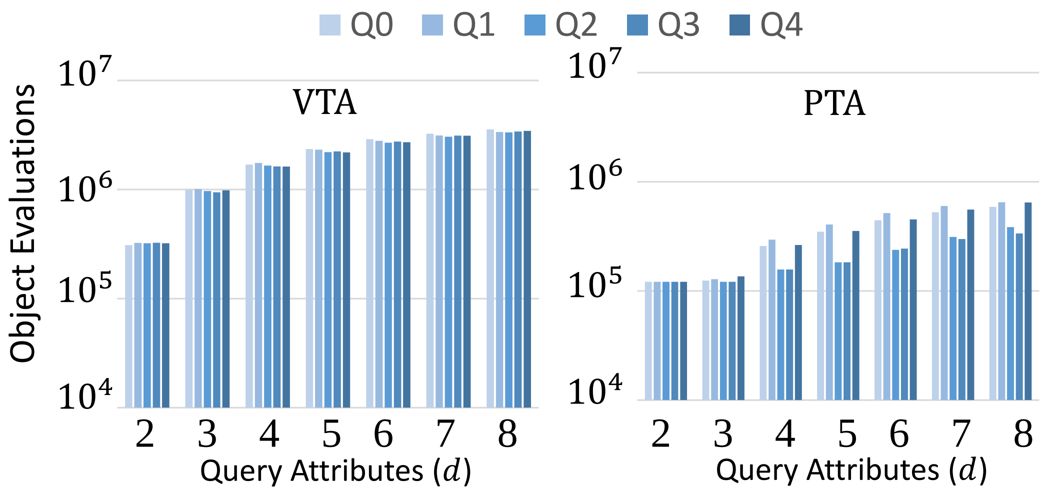


Figure 3.13: Variable weights vs object evaluations.

3.7 Synthetic Data Experiments

In this section, we concentrate on experiments using synthetic data. At first, we concentrate on scalar implementations of our proposed solutions in order to present a fair comparison against previous work which was not originally designed for in-memory execution and SIMD vectorization. Next, we evaluate the performance of our methods using hardware optimized SIMD implementations. Finally, we compare against optimized versions following the MTSQ and MTMQ models of processing.

3.7.1 Related Literature Scalar Comparison

In Figures 3.14 (a), (b), (c), we present the measured number of object evaluations for all developed scalar algorithms. Overall, PTA requires the least number of object evaluations compared to the previously developed solutions. Although, it follows a fixed order of processing within each partition, PTA manages to reduce *rank uncertainty* by

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	$(.4, .3, .2, .1, .1, .2, .3, .4)$

Table 3.2: Individual query weights.

effectively constraining the search space area. 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 exhibits lower algorithmic efficiency, especially for queries on few attributes (i.e. 2 to 4). When the query attributes contain the largest part of all indexed attributes from the relation, VTA is able to approximate the optimal order of processing of common list-based methods. Hence, the *rank uncertainty* and similarly the measured number of object evaluations are roughly equivalent to that of TA, LARA, and HL. 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 exhibit comparable performance for large number of attributes. For only few query attributes, LARA needs

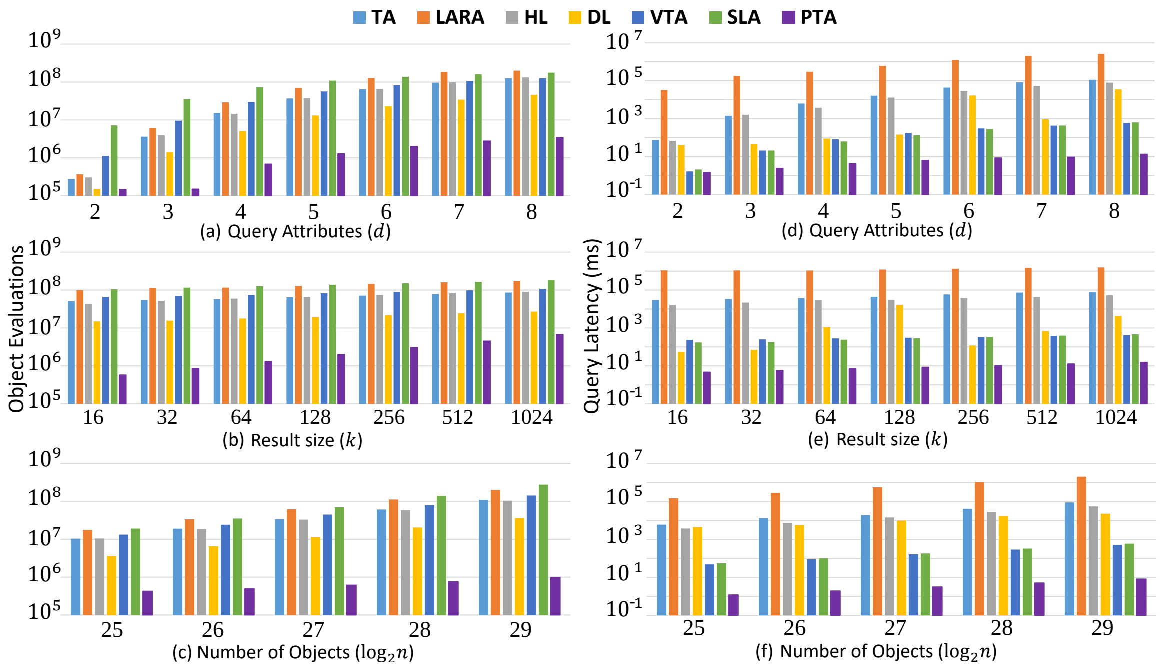


Figure 3.14: Scalar performance on synthetic data.

to evaluates more objects because it cannot approximate well the stopping depth due to having insufficient information about the actual scores of the seen objects.

In Figures 3.14 (d), (e), (f), we showcase the query latency for all scalar methods. VTA and SLA achieve query latency comparable to DL. DL requires traversing frequently the lists of dominated objects for every object within the result set, in order to update its candidate set. Although at each iteration only a few candidate objects will be identified, the process of accessing the relevant auxiliary information is extremely costly. In fact, for a relation that contains a lot of attributes these lists are quite large in length. Therefore, it is with great likelihood that every list access will be served directly from main memory and may also cause a TLB miss. Both of these actions affect negatively the expected query latency. LARA attains the worst query latency because it needs to maintain large candidate

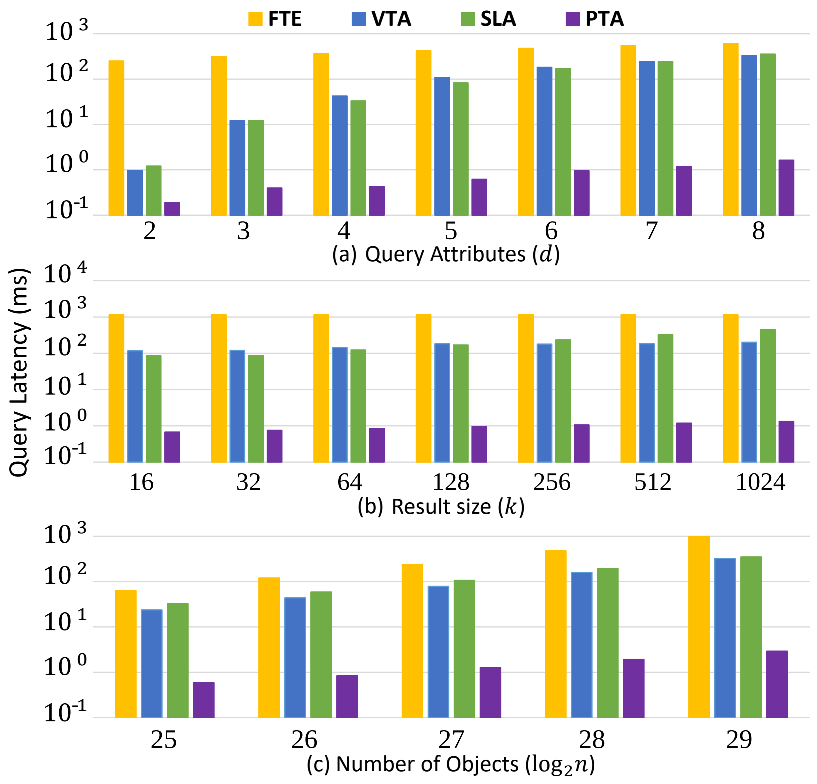


Figure 3.15: Latency using SIMD instructions.

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. In comparison, VTA and SLA are able to achieve lower query latency despite having to evaluate many more objects. Both solutions avoid maintaining a large candidate set (at most k objects) at each iteration, while also the maintenance itself is relatively cheap (i.e. only push and pop operations are executed). Furthermore, the required auxiliary information (see 3.3.1) during processing is very low compared to other methods (i.e LARA, DL). Finally, all memory accesses are sequential being also executed on blocks of data a procedure which is known to be efficient for CPU based processing.

PTA adheres to the same principles while also being work-efficient by relying on intelligent partitioning to guide effectively the order of processing. For this reason, its query latency is noticeably lower.

3.7.2 Hardware Optimized STSQ Processing

In this section, we concentrate on the evaluation of our hardware optimized implementations. We consider only algorithms designed to operate efficiently using AVX instructions.

As indicated by Fig 3.15, 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 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.

3.7.3 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 (denoted with S) of the previous section.

In Figures 3.16 (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

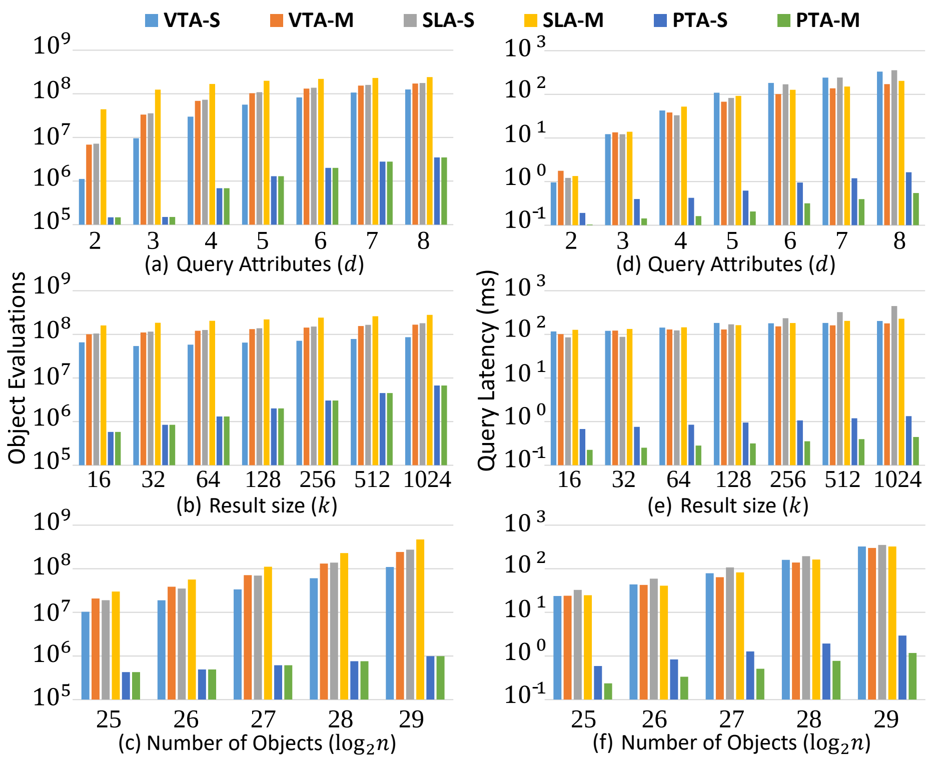


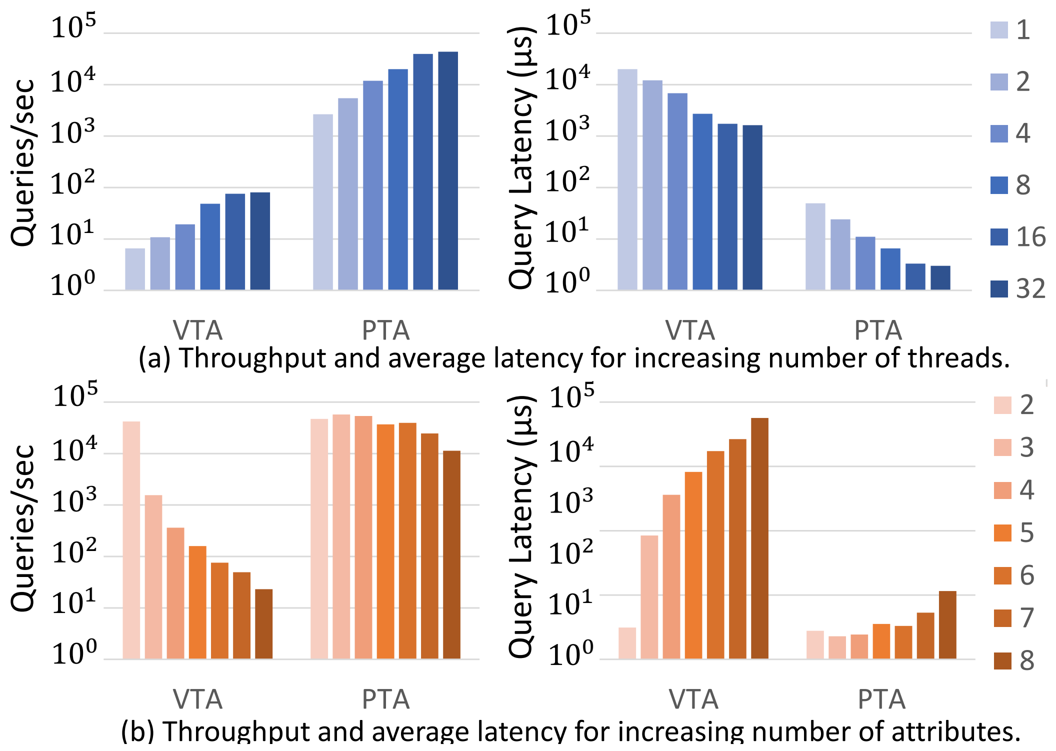
Figure 3.16: Single vs multi-thread performance on synthetic data.

partitioning increases score 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 score 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.

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 3.16 (d), (e), (f), we compare the query latency of our single-threaded and multi-threaded 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.



(a) Throughput and average latency for increasing number of threads.

(b) Throughput and average latency for increasing number of attributes.

Figure 3.17: Throughput-latency on synthetic data.

3.7.4 MTMQ Performance Evaluation

In Figure 3.17, 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 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

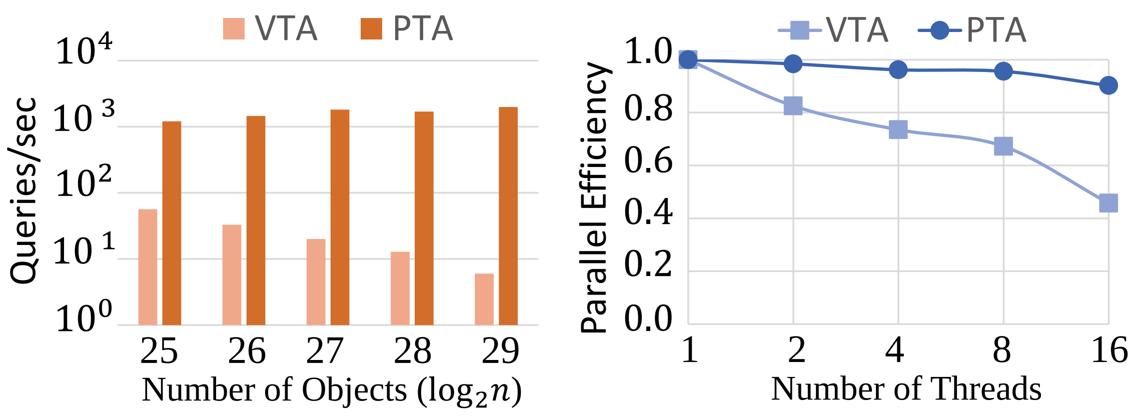


Figure 3.18: $MTMQ$ Scale-up and parallel efficiency.

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 3.18, 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.

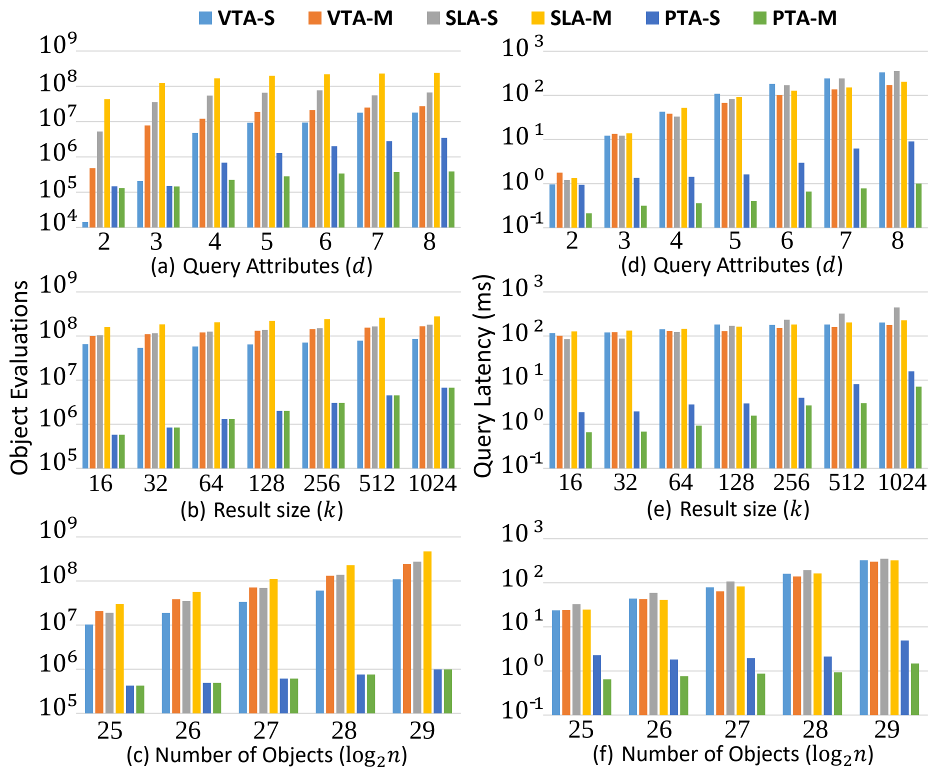
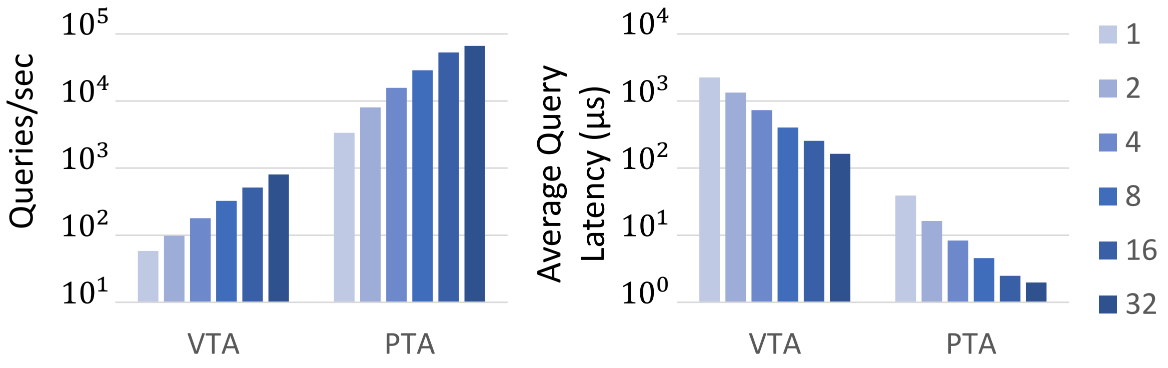


Figure 3.19: Single-thread vs multi-thread performance on real data.

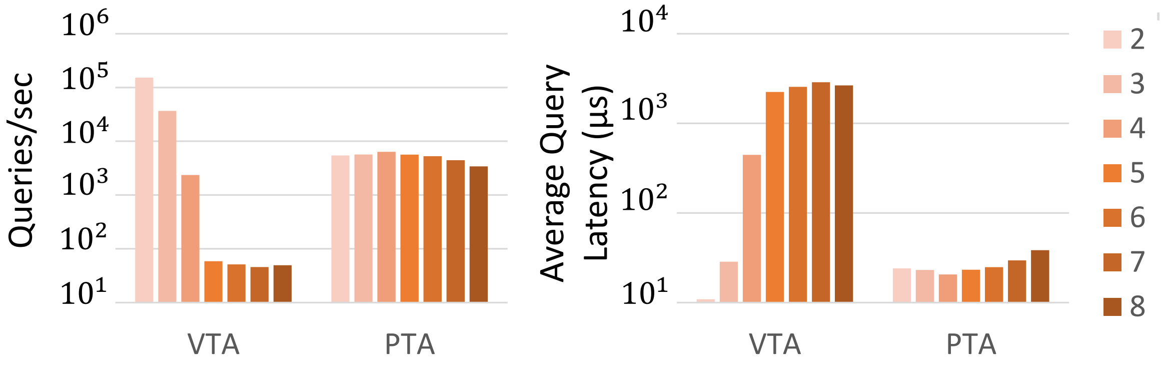
3.8 Real Data Experiments

In this section, we validate our experimental results using real data. There is no discernible difference between the experimental results on synthetic and real data when comparing our scalar implementations against the related literature. For this reason, we concentrate only on the $MTSQ$ and $MTMQ$ processing models using for both the equivalent hardware optimized implementations.

In Figures 3.19 (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 multi-threaded counterparts. PTA-S and PTA-M



(a) Throughput and average latency for increasing number of threads.



(b) Throughput and average latency for increasing number of attributes.

Figure 3.20: Throughput-latency on real data.

outperform 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 3.20, 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.

3.9 Conclusions

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).

Chapter 4

GPU Accelerated Top-K Selection With Efficient Early Termination

4.1 Introduction

A straightforward approach for answering Top-K queries involves two steps: (1) calculating the score of each tuple by summing their weighted attributes (also known as tuple score aggregation), (2) utilizing sorting or k-selection algorithms to identify those tuples having the k highest scores/rankings. The most expensive part of Top-K query evaluation is score aggregation because during that phase data movement dominates the total execution time.

Increasing memory capacity and decreasing memory costs motivated the development of in-memory database systems. Although the process of migrating in-memory has created several opportunities for improved query latency, their potential has been severely limited by the growing gap between processor and main memory speed. Further improvements in processing throughput and query latency can be obtained utilizing multi-core [9] processing or hardware acceleration [53, 1]. Related work has demonstrated the immense potential of GPU accelerated processing for filtering [86], and complex selection [15] operators. This body of work has revealed that caring about practices geared towards high throughput (i.e. coalesced memory access, minimal thread divergence) is as important as designing algorithmically efficient solutions.

GPU accelerated Top-K selection with support for early stopping has not been studied in previous work. It is a very challenging problem to tackle for two reasons: (1) Top-K query processing leverages on random accesses to resolve score ambiguity during tuple evaluation [11, 65], a practice that is inherently incompatible with GPU processing, (2) the immense compute capabilities of GPUs make it hard to justify the additional work that is required for enabling early termination. The latter point is concerned with avoiding intricate query evaluation strategies which might lead to higher query latency, despite enabling less tuple evaluations. Unless a satisfactory trade-off can be obtained there is no motivation to avoid evaluating the complete relation. Data reordering [41, 44] and layering [45, 59] are popular methods geared towards efficient sequential access. Despite being cheap to implement, their pruning abilities are severely affected for queries on relations with high number of attributes.

In this chapter, we investigate the suitability of data re-ordering and intelligent partitioning in order to enable efficient GPU based Top-K selection with support for early termination. We are concerned with Top-K selection queries that involve high number of attributes and focus on techniques that utilize clustered indices to enable early termination. These techniques involve a single initialization step to build the underlying index, after which multiple sub-queries can be efficiently executed on top of it. As established from previous work, such indices can function in a dynamic environment enabling low cost insertions/updates [45, 59]. Our ultimate goal is to improve query latency by developing solutions suitable for massively parallel architectures. The main contributions described in this chapter are summarized below:

- We develop the skeleton of a parallel threshold algorithm (see Section 4.3.3) that is designed to enable efficient GPU Top-K selection with support for early termination.
- We consider two different data partitioning strategies and evaluate their effectiveness when combined with data reordering (see Section 4.3.4).
- We study the performance characteristics of GPU-based Top-K selection and evaluate our proposed solutions for a variety of parameters, including result size, attribute number, and variable preference vectors.

The rest of the chapter is organized as follows: In Section 4.2.1, we discuss the GPU architecture and the details of previous work on GPUs, while Section 4.3 contains a thorough discussion of the proposed framework. Section 4.4 describes the experimental evaluation and Section 4.6 concludes the section.

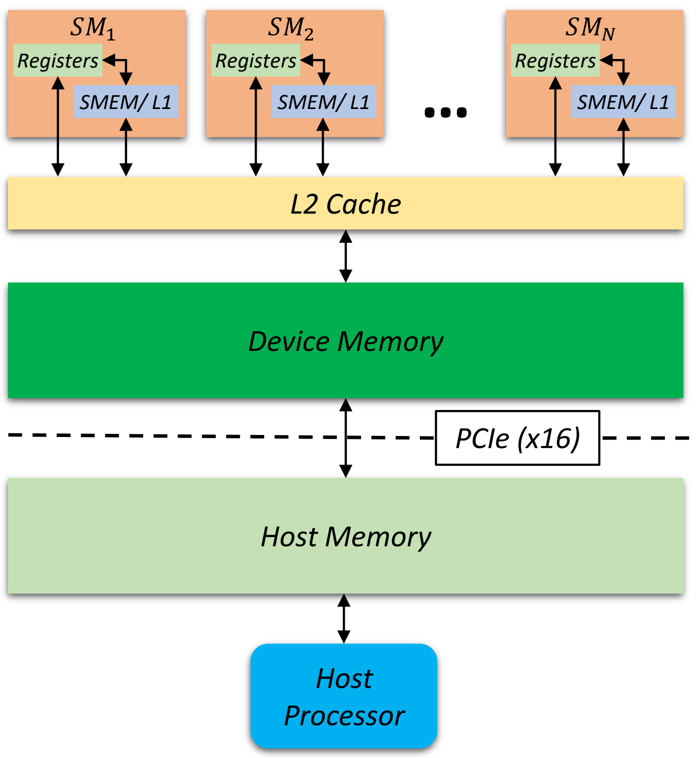


Figure 4.1: GPU Architecture Organization.

4.2 Background

4.2.1 GPU Architecture & Organization

A simplified depiction of the GPU architecture and memory hierarchy is shown in Fig. 4.1. GPUs consist of multi-core processing units known as Streaming Multiprocessors (SMs), each one containing their own set of registers, L1 cache and a software programmable cache (i.e. shared memory). In addition, each SM has direct access to a shared L2 cache and a dedicated RAM often designated as global or device memory. Programs execute on the GPU in the form of kernels. Each kernel utilizes thousand of active threads, typically grouped into thread blocks. Thread blocks share access to L1 cache and shared memory, while each thread within a block has private access to their own set of registers. Blocks are split further into warps which take turns executing in lock-step using any available SM. The threads within a warp should access data stored in global memory sequentially to ensure maximum bandwidth utilization. In addition, a sufficient number of active warps is necessary to effectively mask the latency associated with instruction dependencies (i.e. data access, synchronization).

Although the device memory offers high bandwidth its capacity is limited to only few GBs (i.e. $12-24$ GBs). For this reason, GPUs may rely on the host memory for storage, retrieving (across PCIe) the necessary data on-demand during processing. In modern GPUs, this is made possible through the use of a unified virtual memory space that is managed seamlessly either by the GPU driver or the programmer. The GPU driver facilitates data exchange across PCIe utilizing two types of memory declarations: (1) *Zero copy* memory initiates data transfers each time a GPU kernel is executed, (2) *Managed* memory utilizes heuristics and hints during runtime to prefetch the necessary data into device memory. The latter method works also as a caching mechanism being able to retain data and re-use it in future kernel calls.

4.2.2 Bitonic Top-k Selection

Typically, GPU enabled algorithms operate on data that reside in device memory. In this environment, it is important to take advantage of the immense GPU memory bandwidth by enabling coalesced data accesses when reading from and writing to the device memory. When the associated data are used multiple times during computation, it is

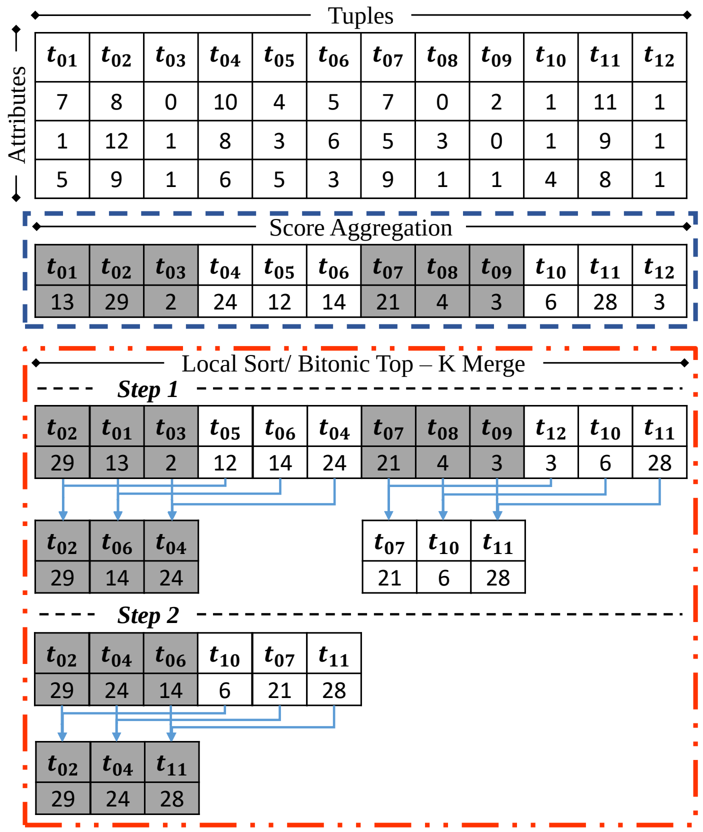

Tuples
Attributes
t_{01} t_{02} t_{03} t_{04} t_{05} t_{06} t_{07} t_{08} t_{09} t_{10} t_{11} t_{12}
7 8 0 10 4 5 7 0 2 1 11 1
1 12 1 8 3 6 5 3 0 1 9 1
5 9 1 6 5 3 9 1 1 4 8 1
Score Aggregation
13 29 2 24 12 14 21 4 3 6 28 3
Local Sort/ Bitonic Top – K Merge
Step 1
t_{02} t_{01} t_{03} t_{05} t_{06} t_{04} t_{07} t_{08} t_{09} t_{12} t_{10} t_{11}
29 13 2 12 14 24 21 4 3 3 6 28
t_{02} t_{06} t_{04}
29 14 24
t_{07} t_{10} t_{11}
21 6 28
Step 2
t_{02} t_{04} t_{06} t_{10} t_{07} t_{11}
29 24 14 6 21 28
t_{02} t_{04} t_{11}
29 24 28

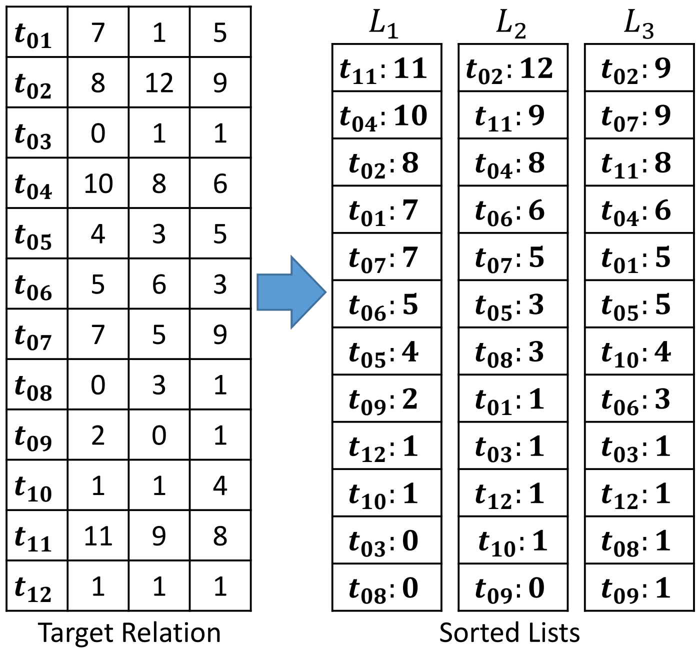
Figure 4.2: Top-3 selection using Bitonic Top-k.

commonplace to avoid unnecessary memory transactions by storing and operating on them through registers or shared memory.

A simple implementation of Top-K selection on GPUs, requires first aggregating the scores of all tuples in a given relation using the user-defined monotone function, and then utilizing a k-selection algorithm to identify those tuples with the k-highest scores. Bitonic top-K [80] is the state-of-the-art k-selection algorithm for GPUs. Its main goal is to avoid completely sorting the key-value pairs that are generated after the aggregation step. In order to achieve this, it executes bitonic sort to create k-sized groups of data which are sorted in alternating order. Consecutive groups are combined using bitonic merge, and the process repeats until a single group with the k-highest scoring tuples is created. An example of this process to calculate the Top-3 query is shown in Fig. 4.2. The bitonic top-K algorithm operates on the key-value pairs generated by summing all attributes of the input relation. Bitonic sort (a.k.a local sort) and bitonic merge execute in sequence by repeatedly sorting and extracting the maximum values until the 3-highest scoring tuples remain.

Despite its higher complexity (i.e. $O(N \log^2 k)$, bitonic top-K performs better than sorting or previous k-selection algorithms based on radix-sort because it avoids expensive scatter operations while also considerably reducing the total amount of data being written back to global memory [80]. Nevertheless, the performance of Top-K selection based on bitonic top-K drops when the target relation contains a large number of attributes. In that scenario, the aggregation phase dominates the total execution time because processing is limited by how fast the data can be read from device memory. In addition, for very large relations that cannot fit in device memory, the required attributes need to be fetched from host memory at the moment of query evaluation. In both, circumstances evaluating all the tuples is detrimental to query latency. Hence improving performance is connected to reducing the overall number of tuples being evaluated.

In relational databases the proven way to achieve this is to utilize a threshold based indexing scheme [34, 11, 65, 45, 41] able to support sub-queries and variable preference vectors. Typically, such solutions require an initialization step where the index is build, after which many queries can be executed on top of the existing data structure. Such methods also support data schemes which can be easily updated in a dynamic environment. However, these early termination solutions are not directly applicable to the GPU environment



Target Relation

t_{01}	7	1	5
t_{02}	8	12	9
t_{03}	0	1	1
t_{04}	10	8	6
t_{05}	4	3	5
t_{06}	5	6	3
t_{07}	7	5	9
t_{08}	0	3	1
t_{09}	2	0	1
t_{10}	1	1	4
t_{11}	11	9	8
t_{12}	1	1	1

Sorted Lists

L_1	L_2	L_3
t_{11}: 11	t_{02}: 12	t_{02}: 9
t_{04}: 10	t_{11}: 9	t_{07}: 9
t_{02}: 8	t_{04}: 8	t_{11}: 8
t_{01}: 7	t_{06}: 6	t_{04}: 6
t_{07}: 7	t_{07}: 5	t_{01}: 5
t_{06}: 5	t_{05}: 3	t_{05}: 5
t_{05}: 4	t_{08}: 3	t_{10}: 4
t_{09}: 2	t_{01}: 1	t_{06}: 3
t_{12}: 1	t_{03}: 1	t_{03}: 1
t_{10}: 1	t_{12}: 1	t_{12}: 1
t_{03}: 0	t_{10}: 1	t_{08}: 1
t_{08}: 0	t_{09}: 0	t_{09}: 1

Figure 4.3: An example of mapping a base relation (left) to a collection of per attribute sorted-lists (right).

because they are known to incur too many random accesses [34, 11]. Methods optimized for sequential access [65, 45, 41] exist but operate at the expense of higher number of tuple evaluations. In the next section, we review TA, a threshold based early termination solution, and describe a generic framework for developing efficient threshold based algorithms for the GPU using data preordering and layering.

4.3 GPU Threshold Algorithms

List-based algorithms utilize a collection of per attribute sorted lists to enable efficient Top-K query processing. These lists present a simple data abstraction that is resilient to different query parameters (e.g. variable preference vectors, result size, attribute number) and can be easily realized in a dynamic environment using self-balancing trees (i.e. B/B+trees). An example of those sorted lists build upon a toy relation is shown in Figure 4.3. The majority of list-based methods follow a similar execution model to the one that was established by the Threshold Algorithm (TA) [34]. The main idea is to iterate over the sorted lists in round robin order and calculate the score of each seen tuple through random access to every other list. The seen tuples with highest scores are maintained using a priority queue that is updated periodically as new tuple-score pairs are generated. Algorithm 7 summarizes the steps associated with TA's execution. We start by initializing an empty priority queue (Line 1) and set the threshold value to zero (Line 3). As stated before, we iterate through all lists in round robin order (Line 4) retrieving one tuple (i.e. tuple-id, attribute value) at a time from each sorted list (Line 5). We use the retrieved key-value pairs to update the current threshold value (Line 6). Unless the tuple was evaluated in the past (Line 7), we continue by inserting the tuple-id into a hash-table (to keep track of evaluated tuples) and initialize the score of the associated tuple equal to the value of the retrieved attribute (Line 11). An index is used to retrieve the remaining attributes of the given tuple from every other list (Line 13) which are then aggregated to the total score of that tuple (Line 14). We update the priority queue with the new tuple if its score is greater than the minimum or less than k tuples have been discovered (Lines 16 - 23). Query processing continues until k items have been discovered and the minimum scoring item has a ranking greater than or equal to the threshold of the current list level (Line 25).

Ignoring memory accesses associated with updating the hashtable (in order to avoid duplicate tuple evaluations), for every tuple evaluation, TA performs 1 sequential

Algorithm 7 Threshold Algorithm

L = *Sorted list collection.*

W = *Preference vector.*

k = *Result size.*

```
1: $Q = \{\}$                                          ▷ Initialize empty priority queue.
2: do
3:     $T = 0$                                         ▷ Initialize threshold value.
4:     for $i \in [1, d]$ do
5:         $(T_{id}, A_i) = getNextObjectFromList(L_i)$
6:         $T = T + W_i \cdot A_i$                     ▷ Update threshold.
7:         if $T_{id} \in M$ then continue             ▷ Check if object seen before.
8:         $M.push(T_{id})$
9:         $S = W_i \cdot A_i$
10:        for $j \in [1, d]$ AND $j \neq i$ do
11:            $A_j = getValueByKeyFromList(T_{id}, L_j)$
12:            $S = S + W_j \cdot V_j$
13:        end for
14:        if $Q.size() < k$ then $Q.push(T_{id}, S)$
15:        else if $Q.top() < S$ then $Q.popMin()$, $Q.push(T_{id}, S)$
16:        end if
17:    end for
18: while $Q.size() < k$ AND $Q.top() < T$
```

access (Line 5) to the corresponding sorted list and $d-1$ random accesses (Line 13) to find the remaining attributes from every other list. The number of tuple evaluations increase rapidly with respect to increasing query attributes [41], a behavior that affects proportionally the total number of random accesses. Random accesses are not compatible with GPU based processing which often relies on coalescing to fully utilize the available memory bandwidth. Data layering [45, 59] or tuple preordering strategies [41] are used to eliminate random accesses at the expense of higher tuple evaluations. In fact, their performance degrades rapidly for high dimensional relations with extreme data variability (i.e. correlated, independent, and anti-correlated data distributions [17]). As opposed to data re-ordering, data layering incurs a higher initialization cost, therefore the former method is a better candidate for GPU based processing.

In order to resolve the above issues, we present a simplistic data layout scheme based on data preordering that can be adopted to enable efficient GPU based processing. In addition, we describe the major components of a generic framework for developing efficient GPU Threshold Algorithms (GTA). Utilizing this framework, we concentrate on developing and evaluating two algorithms which use different partitioning schemes when assigning work to distinct thread blocks, namely, GTA with random partitioning (GTA-RP) and GTA with angle space partioning (GTA-ASP). We show empirically and experimentally that intelligent partitioning contributes towards high algorithmic efficiency.

4.3.1 Ordered Data-Threshold Table

Designing GPU-friendly threshold algorithms necessitates preordering the tuples of a relation to enable coalesced data access. This practice is often detrimental for queries

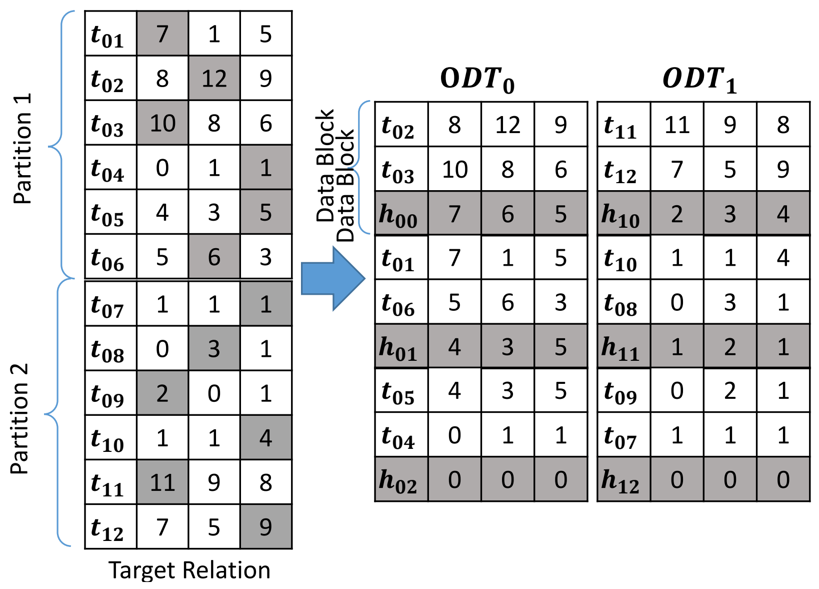


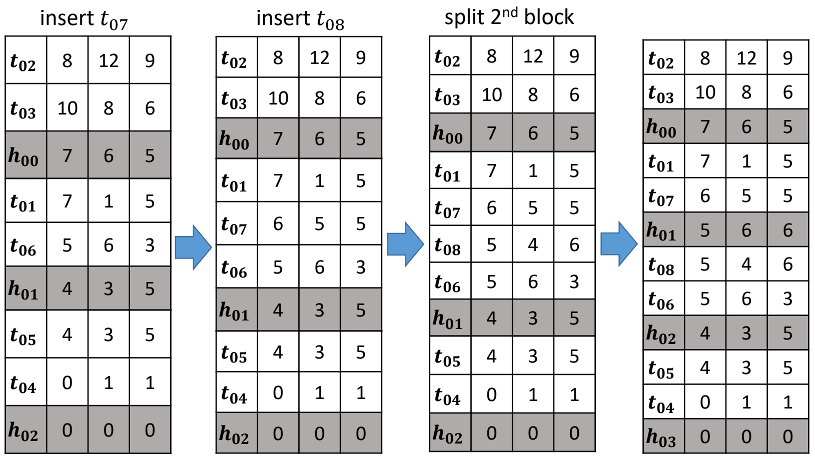
Figure 4.4: An example of mapping a base relation (left) to multiple ODT tables (right) by preordering tuples based on the maximum attribute value (indicated in gray).

that execute only on a subset of all available attributes (i.e. sub-queries). Queries on skewed distributions (i.e. anti-correlated data) or those evaluated on relations with high number of attributes become very challenging to process. Such behavior is related to the number of possible tuple orderings which grow exponentially to the number of query attributes. It is possible to overcome the aforementioned issues by restricting the value range of the associated attributes for a collection of tuples, through intelligent partitioning. Creating these range boundaries limits the number of possible tuple orderings within a partition, thus enabling a better total ordering that is beneficial for early termination.

Investigating this hypothesis requires first describing the central component of our broader data organization scheme, henceforth referred to as Ordered Data-Threshold (ODT) table. ODT tables are formulated by rearranging/layering the tuples of a target relation so as to ensure early evaluation of those with the greatest likelihood to score high. Different preordering strategies are possible including those based on skyline layering [45] or first seen position using list-based ordering [41]. In order to simplify discussion and construction of ODT tables, we order the tuples according to their largest attribute as shown in Fig. 4.4. In that example before creating each ODT table, we partition the data into two distinct collections. Although different partitioning strategies are possible (see Section 4.3.4), we group tuples based on their insertion order just for demonstration purposes. An ODT table is logically split into *ordered* data blocks, each one containing several tuples from the original relation plus one extra tuple (depicted in grey), known as the the threshold tuple. In the general case, where a relation is divided into p partitions, and b data blocks per ODT table, a single data block contains a set of relation tuples (C_{ij}) and a *threshold* tuple (H_{ij}), where $i \in [0, p-1], j \in [0, b-1]$. Let $C_{ij}[n, d]$ be the d-th attribute of the n-th tuple in C_{ij} then the threshold tuple H_{ij} is calculated as follows:

$$H_{ij} = \{a_m | a_m \geq \underset{r \in [j+1, b-1]}{\arg\max} \; C_{ir}[n, m]\} \tag{4.1}$$

The threshold tuple (H_{ij}) contains the maximum attribute values among those in the tuples of any subsequent data block. It is useful for determining when to safely stop query processing because it provides information about the maximum possible score of the tuples which are yet to be processed. When constructing the associated ODT tables, the threshold tuples are computed inexpensively by keeping track of the maximum attribute values during the assignment of every tuple to its corresponding data block. Consider the example of Fig. 4.4, for a Top-2 query on all attributes, in ODT_0 tuples $t_2 = 29$ and $t_3 = 24$



t_{02}	8	12	9
t_{03}	10	8	6
h_{00}	7	6	5
t_{01}	7	1	5
t_{06}	5	6	3
h_{01}	4	3	5
t_{05}	4	3	5
t_{04}	0	1	1
h_{02}	0	0	0

t_{02}	8	12	9
t_{03}	10	8	6
h_{00}	7	6	5
t_{01}	7	1	5
t_{07}	6	5	5
t_{06}	5	6	3
h_{01}	4	3	5
t_{05}	4	3	5
t_{04}	0	1	1
h_{02}	0	0	0

t_{02}	8	12	9
t_{03}	10	8	6
h_{00}	7	6	5
t_{01}	7	1	5
t_{07}	6	5	5
t_{08}	5	4	6
t_{06}	5	6	3
h_{01}	4	3	5
t_{05}	4	3	5
t_{04}	0	1	1
h_{02}	0	0	0

t_{02}	8	12	9
t_{03}	10	8	6
h_{00}	7	6	5
t_{01}	7	1	5
t_{07}	6	5	5
h_{01}	5	6	6
t_{08}	5	4	6
t_{06}	5	6	3
h_{02}	4	3	5
t_{05}	4	3	5
t_{04}	0	1	1
h_{03}	0	0	0

Figure 4.5: Example depicting insertion of a new tuples in a given ODT table.

will be evaluated and processing will stop at the first data block because the threshold tuple $h_{00} = 16$ guarantees that no tuples exist with higher score. Note that the chosen block size is independent of the query result size k. In a real life application, our method would operate on hundreds of partitions containing thousands of blocks each with variable query parameters including varying query attributes (m), result size (k), and tuple number (n). Note that the threshold attributes can be updated iteratively by examining only the attribute values of neighboring blocks to those where new tuples are being added or existing ones deleted (see Construction & Maintenance).

4.3.2 ODT Construction & Maintenance

ODT tables possess similar properties to that of data layering strategies [45, 41]. However, they are simpler to build and inherently well structured for coalesced data access. They can be constructed in batch using simple highly parallel GPU primitives (i.e. sort, parallel reduction). In addition, ODT tables can be easily maintained in a dynamic environ-

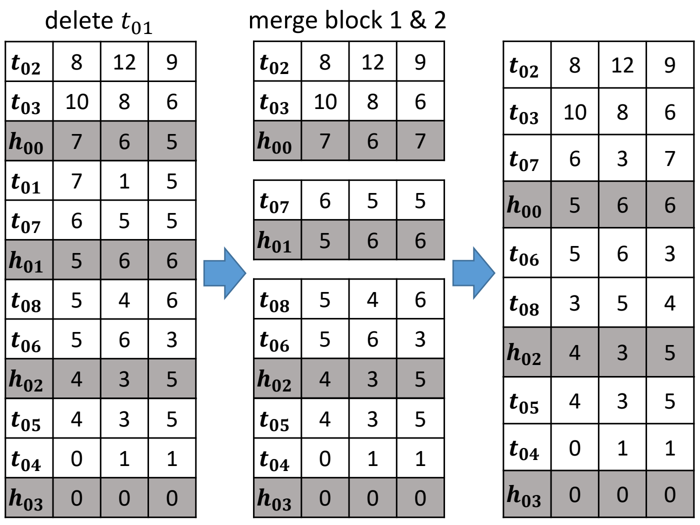


Figure 4.6: Example depicting deletion of an object from a given ODT table.

ment by retrofitting them to support insert and delete operations using a self-balancing tree structure (i.e. B/B+ tree). In that environment the data blocks of a given ODT table, correspond to the leaf pages of the associated tree structure (think of a clustered index). These leaf pages are created by indexing the maximum attribute of each tuple and are augmented with a threshold tuple as described previously. Any insert, update or delete operation will be managed on the CPU side, allowing the GPU to operate on a read-only instance of the transformed relation. Efficient GPU based processing is contingent on enabling coalesced access within any given data block, thus linking randomly pages in main memory will not degrade performance.

Below we demonstrate the capability of ODT tables to support insert and delete operations using two examples indicating their behavior during such scenarios. For these examples, we assume a minimum and a maximum block size of 2 and 3 respectively. In Figure 4.5, we showcase how an ODT table is updated during several consecutive tuple insertions which lead to a block split operation. A new tuple $t_v = \{a_0, a_1, ..a_{d-1}\}$ is inserted into an ODT table by utilizing binary search to discover block B having a threshold tuple $h_{ij} = \{t_0, t_1...t_{d-1}\}$ such that $\exists a_m \in t_v$ where $a_m > t_m$ where $m \in [0, d-1]$. In our example, t_{07} will be inserted in the second block because at least one of its attributes is greater than the equivalent attributes of h_{01}. The same happens for tuple t_{08} after the insertion of which a split operation occurs due to the current block size being larger than the maximum (3). For a block that is being split into two new ones, we preorder the tuples according to their maximum attribute value and group them into blocks of size equal to the minimum. In the previous example, this will result in two groups, one containing t_{01} and t_{07} and the other t_{08} and t_{06}. For the first group, the threshold tuple is calculated by finding the maximum attributes of the subsequent block (third block). The second group retains the threshold tuple of the original block, as it was before the split, since no changes occurred below that block.

Figure 4.6 showcases what happens when a delete operation causes the merging of two blocks. There tuple t_{01} is deleted resulting in the second block having less tuples than the minimum allowed. When merging two ordered blocks B_i and B_j, where B_j comes after B_i, we create a new block with all their tuples combined and a threshold tuple equal to that of B_j. In our example, the first and second blocks combined together utilizing the threshold tuple of the latter for that of the new block.

Algorithm 8 Aggregate-Heap Building (hbuild)

C = ODT collection.

S = Tuple-id, scores buffer.

Q = Tuple-id, scores heap.

W = Preference vector.

k = Result size.

```
1: for $i \in [1, p]$ in parallel do
2:     for $j \in [1, b]$ do
3:         for $(t \in C_{ij})$ & $(h \in H_{ij})$ in parallel do
4:             $S_{it} = \sum_{m=1}^{d} (w_m \cdot t_m)$                ▷ Score aggregation.
5:             $T_h = \sum_{m=1}^{d} (w_m \cdot h_m)$                   ▷ Threshold value.
6:         end for
7:         __syncthreads()
8:         $Q_i = hmerge(\{Q_i \cup S_i\}, k)$                         ▷ Bitonic merge.
9:         if $Q_i.min() >= T_h$ then                                   ▷ Early stopping.
10:            return $Q_i$
11:        end if
12:    end for
13: end for
```

4.3.3 Heap Build & Reduction

GTA operates in two phases: (1) Aggregate-Build Heap phase (hbuild), (2) Heap Merge phase (hmerge). Each phase is implemented using a distinct GPU kernel. A simplified description of the first phase is shown in Algorithm 8. Every partition contains a single ODT table, assigned for processing to a single thread block (Line 1). Threads within a block are responsible for aggregating in parallel the scores of several tuples (Line 3-5) and calculating the threshold value (Line 5). At the end of every data block evaluation the current collection of <tuple-id,score> pairs are combined with the k highest scoring pairs identified from previous iterations (Line 6). This heap is stored in shared memory and is constructed using Bitonic Top-K. The code responsible for merging is similar to that of the hmerge kernel (Algorithm 9). At the end of the merge step, the minimum heap score is compared to the associated threshold (Line 8) to determine when the Top-K answer for a given partition becomes available. Unless this condition is false, we write the corresponding pairs in global memory and terminate processing. The hmerge operates on the individual heaps created by hbuild. A fixed collection of ¡tuple-id, score¿ pairs is assigned to distinct thread blocks for processing. Each thread block reduces their input set to k pairs having the highest score by combining bitonic sort (only the first k iterations [80] Line 2) and parallel reduction (Line 4). Several rounds of sort-reduce operations are executed in sequence until only k pairs remain.

Algorithm 9 Heap Merge (hmerge)

```
$S$ = Tuple-id, scores collection.
$k$ = Result size.
$n$ = Score buffer size.

1: while $n > k$ in parallel do
2:     $bitonic\_sort(S, k, n)$                ▷ Sort up to $k$.
3:     $S_i = max(S_i, S_{i+k})$               ▷ Bitonic merge.
4:     $n = n/2$
5: end while
6: return $S_{0:k}$        ▷ Return $k$ highest tuple-id, score pairs.
```

4.3.4 Data Partitioning Strategies

Data partitioning is crucial for achieving efficient GPU based processing. The rationale is that good partitioning facilitates workload balance which is pivotal for masking data access latency and maximizing throughput by using an adequate number of operating threads. Likewise, in Top-K selection data partitioning influences algorithmic efficiency since it restricts access to tuples that can effectively prune the search space. Achieving a balance between these extreme cases is possible using an intelligent partitioning scheme.

$$
\begin{aligned}
tan(\phi_1) &= \frac{\sqrt{(\tilde{A}_d)^2 + (\tilde{A}_{d-1})^2 ... + (\tilde{A}_2)^2}}{\tilde{A}_1} \\
&... \\
tan(\phi_{d-2}) &= \frac{\sqrt{(\tilde{A}_d)^2 + (\tilde{A}_{d-1})^2}}{\tilde{A}_{d-2}} \\
tan(\phi_{d-1}) &= \frac{\tilde{A}_d}{\tilde{A}_{d-1}}
\end{aligned}
\tag{4.2}
$$

Random partitioning (RP) groups together tuples according to their relative position within a target relation; an example of RP is shown in Fig. 4.7 (left). This method of partitioning is beneficial for two reasons: (1) it incurs almost zero initialization cost, (2) it constructs partitions having approximately the same size, which supports workload balance during processing. However, this practice might contribute to the creation of partitions that consist primarily of anti-correlated data. In this case, algorithmic efficiency and in turn query latency are adversely impacted by the fact that an optimal query evaluation depends on different tuple re-orderings dictated by variable query parameters (i.e. attribute number, preference vector, result size).

Angle space partitioning (ASP) formulates multiple data collections by enabling grid partitioning on the polar coordinates (Eq. 3.2) of every tuple in the target relation. Considering geometric symmetry and the fact that we are interested in the highest ranked tuples, we compute the polar coordinates of $\tilde{A}_i = (\alpha - A_i)$ where A_i is the i-th attribute of each tuple having values in range $(0, \alpha]$. The number of resulting partitions is determined by the number of split points for each angular dimension. Assuming s split points, the resulting number of partitions is s^{d-1}, where d is the number of attributes in the based table. Alternatively, there exist solutions enabling equi-volume partitioning using ASP [96]. In our

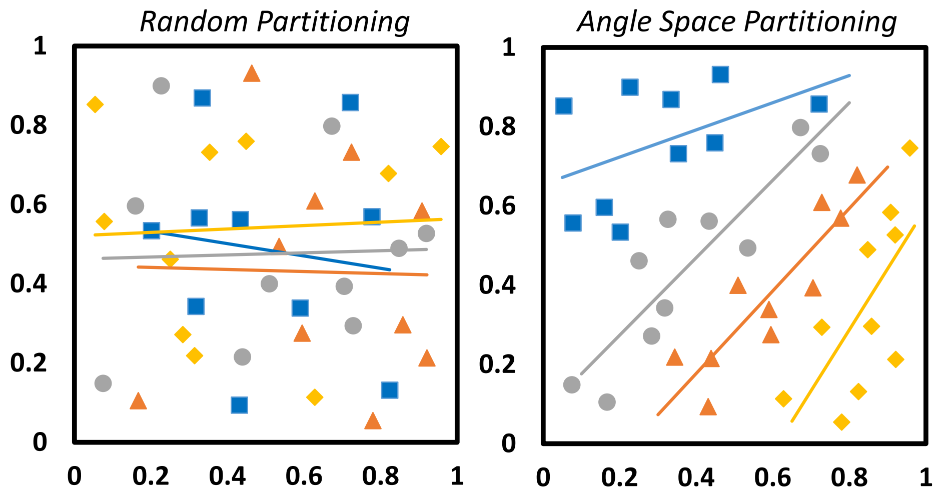


Figure 4.7: Random vs Angle Space Partitioning.

experiments, we concentrate on regular grid partitioning since it incurs lower initialization cost without noticeable difference in query latency.

A toy example indicating the different characteristics of ASP vs RP is shown in Figure 4.7. In that example, each partition consists of tuples indicated with similar point shape and color. Compared to RP, ASP is better alternative when partitioning the data for Top-K selection. This happens because the latter method creates partitions containing tuples with correlated attributes which are inherently easier to linearly order for any monotone function utilizing the ODT table concept. This conjecture is also applicable to relations with more than 2 attributes and has been showcased to be effective for skyline computation [96], where attaining a near optimal linear order is critical for improving algorithmic efficiency.

In order to demonstrate ASP's superiority in a more intuitive manner, we utilize the concept of "identical score curve" (ISC) as proposed in [89]. ISC is the line corresponding to equation $f(t) = v$, consisting of tuples (t) in the data space whose scores are equal to v. Let t_k be the minimum scoring tuple in the priority queue at some point during query evaluation, $ISC(t_k)$ the line defined by equation $f(t_k) = v_k$ and T_i the corresponding threshold tuple. Assuming the priority queue contains k tuples, query processing terminates if and only if $F(T_i) \leq F(t_k)$ has been satisfied. This indicates that T_i must be on or below $ISC(t_k)$ inside the half-space that is closer to the origin. In Figure 4.8, we concentrate on one partition from each partitioning method and plot the corresponding ISC, and threshold tuples at various points during Top-2 query processing. The example of Figure 4.8 derives from the partitions of the toy dataset depicted in Figure 4.7. Our goal with this example is to demonstrate how different partitioning strategies affect early termination by influencing $ISC(t_k)$ and T_i respectively. Note that because the tuples are reordered based on their maximum attribute value, the order in which they are visited during query processing is equivalent to performing in succession a plane sweep of each axis.

In the RP example (Figure 4.8 left), the first two tuples evaluated are $t_1 = (x_1, y_1)$ and $t_2 = (x_2, y_2)$. Query processing will terminate if the corresponding threshold tuple is below the halfspace defined by $ISC(t_1)$. In the worst case, because of random partitioning the threshold tuple will consist of attributes that are arbitrarily close to the maximum from those in at least one of t_1 and t_2. When this happens, it is very likely for the threshold tuple to reside in the halfspace above $ISC(t_1)$. In fact, in our example the threshold tuple T_1 contains x_3 from $t_3 = (x_3, y_3)$ which is very close to the value of x_1 from t_1. Thus, in the first iteration query processing does not terminate because the stopping condition is not satisfied. Because the tuples within the given partition are not restricted in any way, it takes several iterations and evaluation of tuples t_3, t_4, t_5, t_6, before Top-2 selection can safely terminate. This happens because T_2 resides above the halfspace defined by $ISC(t_4)$, and only after t_5, t_6 are evaluated, the new threshold T_3 is available for consideration, indicating that no tuples exist with better score than that of t_4.

ASP creates partitions with tuples having attributes restricted by the partition boundaries. This practice restricts the value range of the threshold attributes, subsequently reducing the threshold and enabling early termination with fewer tuple evaluations. In the ASP example (Figure 4.8 right), after evaluating $t_1 = (x_1, y_1)$ and $t_2 = (x_2, y_2)$, t_3 and t_4 are combined to create T_1 which is the current stopping threshold. At this point, because T_1 is in the halfspace below $ISC(t_2)$ processing can safely stop since we have discovered the two highest ranking tuples and satisfied the stopping condition. Therefore, for a single partition only 2 evaluations were required as opposed to 6 when using RP. When the partition angle is small, the partition boundaries restrict the threshold attributes, resulting in rapidly decreasing threshold score which contributes towards early with few tuple evaluations.

4.3.5 GTA Complexity Estimation

In this section, we analyze the complexity of GTA and the manner in which it is affected by the previously mentioned partitioning strategies (i.e. RP and ASP). Let L be the maximum attribute value for all n tuples and m be the number of query attributes. GTA evaluates each tuple within a given partition in-order of their maximum attribute. Considering that the tuples are mapped in multidimensional space, this order of processing is equivalent to a plane sweep of the axes that correspond to the actual tuple attributes.

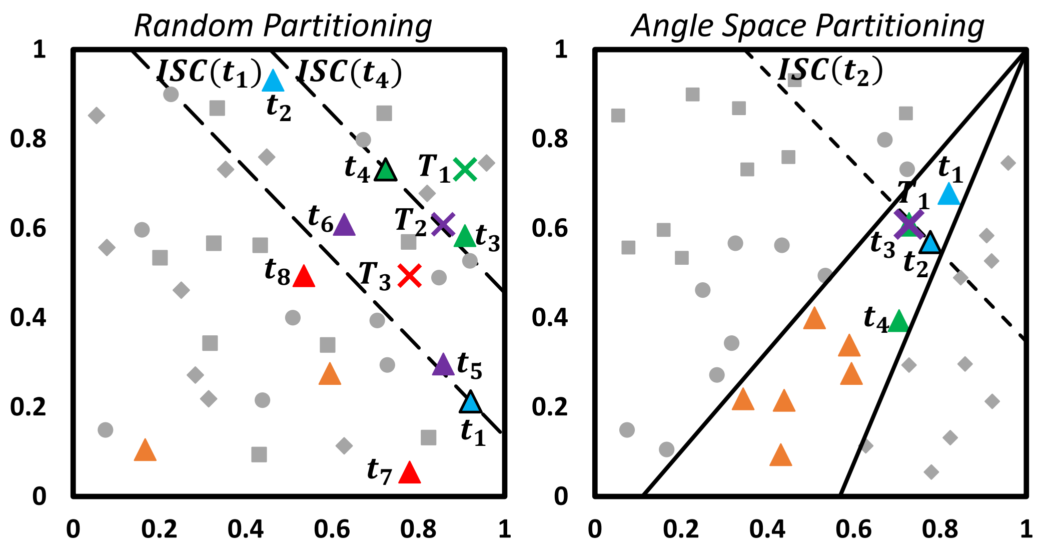


Figure 4.8: Random vs Angle Space Partitioning.

Let $[L, L - \delta_j]$ $(i \in [1, m])$, be the region processed by GTA for some axis x_i, at some point during query evaluation. Due to the order in which tuples are visited during processing, GTA would have evaluated some tuple t_i $(i \in [1, n])$, if and only if t_i contains at least one attribute $a_j \in [L, L - \delta_j]$. In this case, it is possible to realize GTA's complexity as a function of δ_i by calculating the volume (or area in 2D) of the polytope defined by hypercube $[0, L]^m$ and the intersection of every "swept" region $[L, L - \delta_j]$ $(i \in [1, m])$. Our analysis is applicable for uniformly distributed points (tuples) in space and assume the existence of a single data partition. However, it is a scenario that could arise in the worst case with many partitions when RP is used because there is no restriction on the attribute range when tuples are assigned to distinct partitions.

In order to simplify the discussion and without loss of generality, we continue our analysis by concentrating on the $2D$ case where $L = 1$. Figure 4.9 (left) provides an illustration (depicted in green) of the area that has been processed after regions $[1, 1 - \delta_1]$

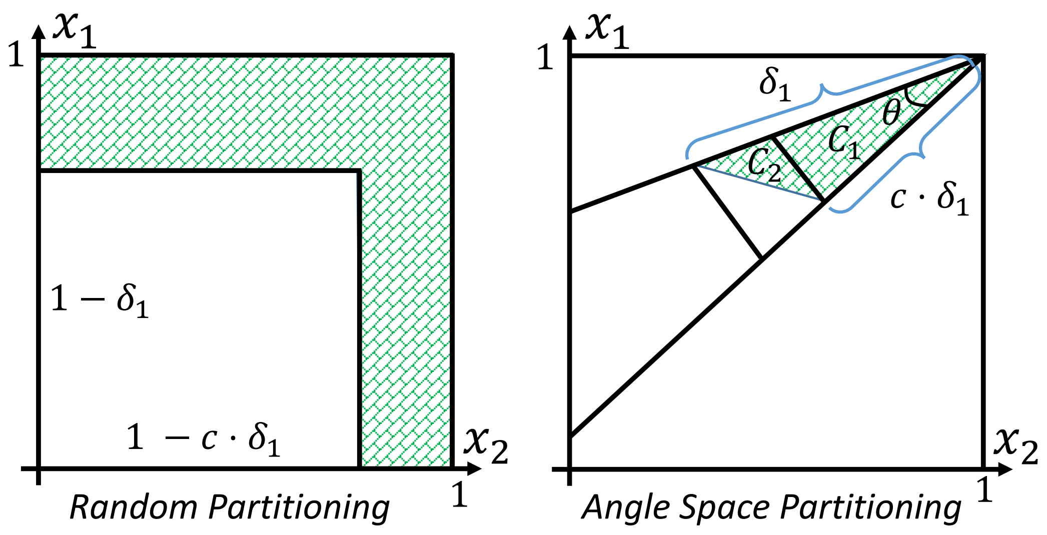


Figure 4.9: Processed Area for Varying δ_i using RP and ASP.

and $[1, 1 - c \cdot \delta_1]$ have been "swept" by GTA. We correlate $delta_2$ to δ_1 through $c \in [0, 1]$ in order to emulate the different query parameters (e.g. preference vector, result size) and data distributions that could possibly affect how they evolve during processing. The expected processed area when using RP as function of δ_1 and c equals:

$$E_{RP} = 1 - (1 - \delta_1) \cdot (1 - c \cdot \delta_1) \tag{4.3}$$

Following the same process, we showcase in Figure 4.9 (right) the area of a partition defined by angle θ that has been processed after some point during query evaluation. The partition boundaries dictate the value of δ_i because no tuple within the partition will have attributes outside that range. In this case, we can represent the processed areas as a function of δ_1 and c using the following equation:

$$E_{ASP}\delta_i^2 \cdot sin(\theta) \cdot \frac{(3 + c^2)}{4} \tag{4.4}$$

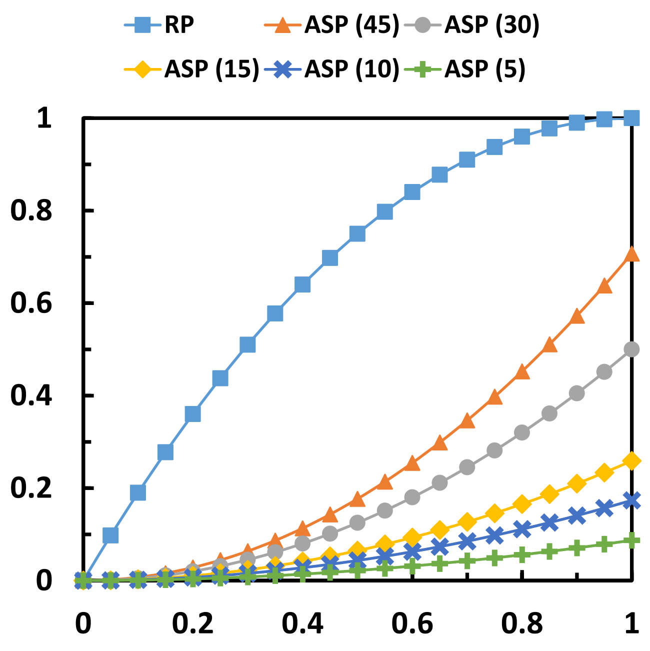


Figure 4.10: Expected processed area as function of δ_1 using RP and ASP(degrees).

The previous equation was calculated by finding the combined area of triangles C_1 and C_2. Equation 4.4 indicates that a smaller partition angle contributes towards the reduction of the total area that needs to be processed for a given δ_1 value. However, we cannot decrease the partition angle indefinitely because it may result in evaluating more tuples than necessary since from each partition at least k tuple will be evaluated. This can be realized in Figure 4.10 where we plot the corresponding area values for increasing δ_1, $c = 1$ and varying angles. In this we observe that the processed area value decreases less rapidly after 15 degrees. Therefore, there is little benefit in having too many small partitions. We discovered experimentally that utilizing ASP by having 512 to 2048 partitions is enough to attain a good trade-off between having enough work for the GPU to operate efficiently and reducing the number of tuple evaluations.

4.4 Experimental Environment

In our experimental evaluation, we consider two GTA algorithms: (1) GTA-RP which utilizes random partitioning, (2) GTA-ASP which utilizes angle space partitioning. Due to lack of existing solutions that support early termination on GPU, we compare against Bitonic Top-K [80], henceforth denoted as GFTE (GPU Full Table Evaluation).

In order to demonstrate the importance of early termination, we explore two different experimental paradigms (1) where the data reside in device memory and are directly accessible by the GPU Streaming Multiprocessors (SMs), (2) where the data reside in host memory and are managed explicitly by NVIDIA's unified memory driver. The latter form of data management has two modes of execution, one where the driver retrieves the data during the kernel's first call (Zero Copy Mode), and another where it receives a hint to prefetch the necessary data before query evaluation (Prefetching Mode).

In addition to our comparison with state-of-the-art GPU based algorithms, we implemented and evaluated the performance of CTA-ASP, an equivalent solution to GTA-ASP, that is optimized for CPU based processing using multi-threading and AVX instructions. We provide discussion comparing CTA-ASP and GTA-ASP against their corresponding full table evaluation algorithms which are optimized for CPU (CFTE) and GPU (GFTE) processing, respectively.

Our experiments were conducted using a single 12 GB NVIDIA Titan V GPU attached to a single socket Intel Xeon E5-1650 processor @ 3.5 GHz with 32 GB of RAM. All algorithms were implemented using standard C++, CUDA 10.0 and NVIDIA's CUB Library [67]. The CPU implementations utilize distinct priority queues for every thread,

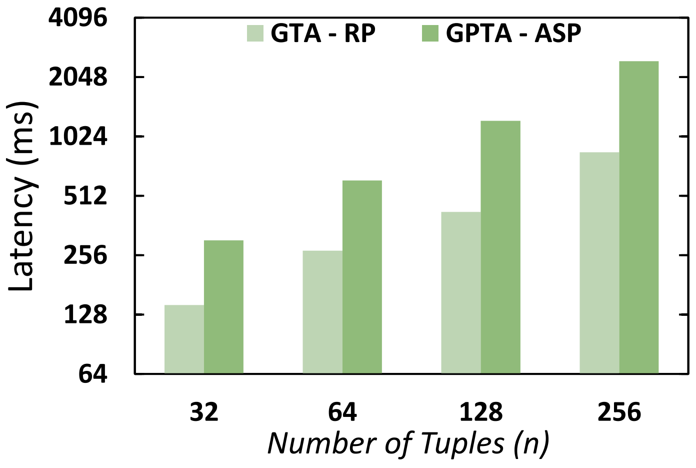


Figure 4.11: GPU Data Partitioning cost.

which are combined towards the end of query evaluation. Our code is publicly available in Github [105].

4.4.1 Dataset, Queries & Metrics

Following the example of previous work [45, 107, 65], we conducted experiments using synthetic data and three types of distributions, mainly correlated, independent and anti-correlated, generated using the readily available data-set generator [17]. Our experiments concentrate on measuring query latency for variable attribute number ($d \in [2, 8]$) and result size ($k \in [4, 8, 16, 32, 64, 128, 256]$) for a relation containing 256 million tuples with 8 attributes each (8 GB of raw data). For experiments with data that reside in host memory, we generated a relation with 8 attributes and 512 million tuples (16 GB of raw data). Finally, we measured the execution time on independent data utilizing variable preference vectors as summarized in Table 3.2.

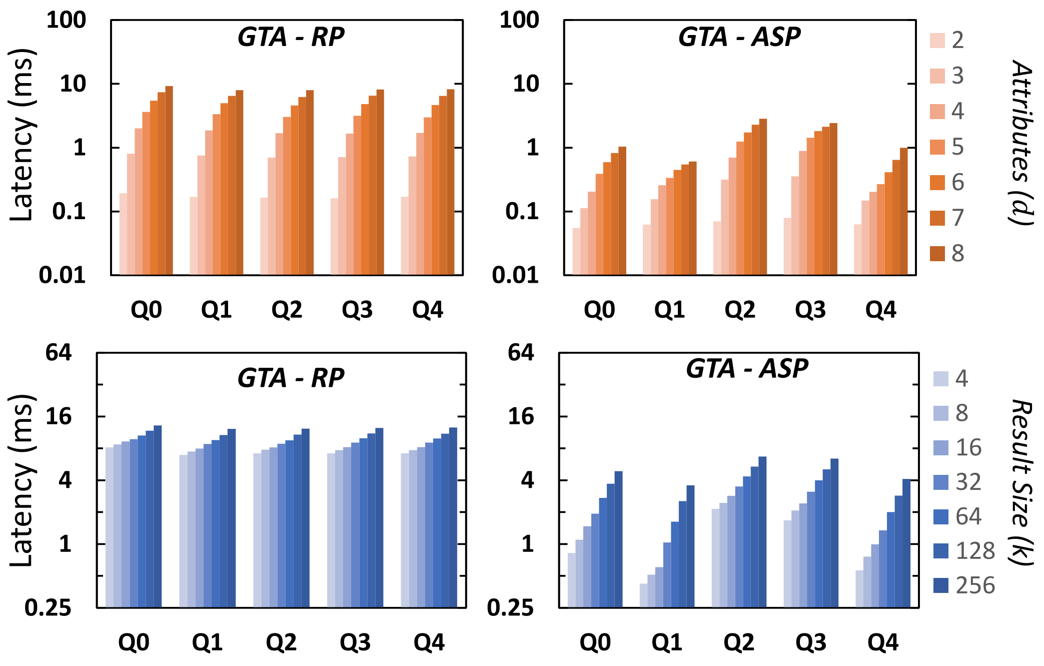


Figure 4.12: Varying Query Preference Vector.

4.5 Synthetic Data Experiments

4.5.1 Initial Cost of Indexing

Figure 4.11 indicates the total cost of initialization which includes partitioning and the host-to-device communication when building the ODT tables. ASP incurs at most twice the initialization overhead of RP while being 2× to 200× faster in terms of query latency, as it will be come clear in the following sections. Initialization occurs only once before any queries are executed and is commonplace for all list-based early stopping algorithms [34, 65, 45, 59]. Note that regardless of the chosen partitioning strategy, ODT tables can be constructed utilizing a "bulk loading" algorithm and maintained using insert/delete operations as described in Section 4.3.1. In addition, our methods are generic in that they can be applied on an arbitrary number of attributes and support queries only for a subset of them. This property is demonstrated throughout our experimental evaluation where we present the query latency for sub-queries on 2 to 8 attributes for a target totaling 8 attributes per tuple.

4.5.2 Variable Preference Vectors

As indicated in Fig. 4.12, there is no difference in execution time when using random partitioning for every tested preference vector and query parameters. On the other hand, ASP experiences somewhat noticeable change in execution time across different queries. This behavior is associated with the way query weights influence the associated processing workload of each partition. ASP operates on all attributes, though it correlates them through angular coordinate calculations which consider fewer of them in ascending attribute order (see Eq. 3.2). This ordering lessens the effects of other attributes in the corresponding angular coordinate calculations for those appearing at the end.

Q_0	$(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)$
Q_1	$(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8)$
Q_2	$(0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1)$
Q_3	$(0.1, 0.2, 0.3, 0.4, 0.4, 0.3, 0.2, 0.1)$
Q_4	$(0.4, 0.3, 0.2, 0.1, 0.1, 0.2, 0.3, 0.4)$

Table 4.1: Individual query weights.

Hence, preference vectors that favor these attributes using higher weights (see Q_1, Q_4) will naturally result in less work to process the corresponding edge (those closer to the

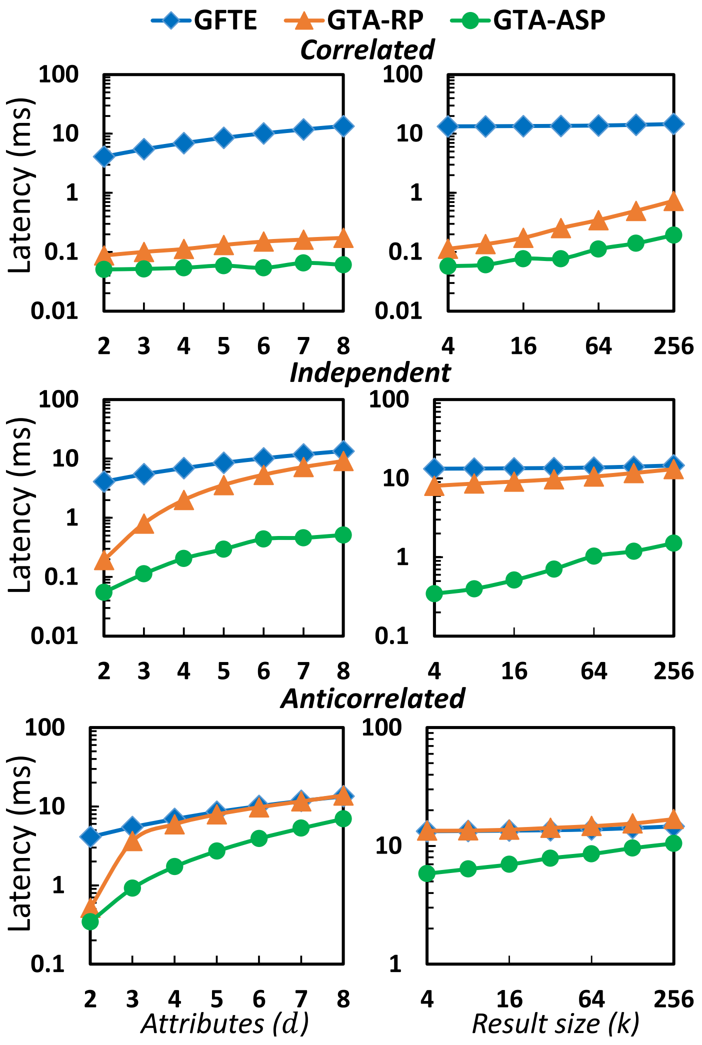


Figure 4.13: Device Memory Query Performance.

axes in high-dimensional space) partitions. In contrast, symmetrically opposite preference vectors (see Q_2, Q_3) are responsible for higher execution time since the angle coordinates are diluted more with irrelevant information from other attributes. Although this behavior is inherent to ASP, its effect on execution time are minuscule, as indicated by our measurements. Henceforth, in our remaining experiments we present results using Q_0.

4.5.3 Device Memory Query Processing

In Fig. 4.13, we present the query latency for all developed algorithms on different data distributions when the base relation resides in device memory. Early stopping solutions are very efficient when processing highly correlated data because it is possible to order the tuples linearly without any ambiguity. For this reason, both GTA-RP and GTA-ASP are respectively 60× and 150× on average faster than GFTE. Independent data are somewhat more difficult to process. This happens because the likelihood of a tuple with one high scoring attribute appearing early on during processing increases dramatically. For this reason, GTA-RP ends up evaluating almost 50% of the raw data, despite them being irrelevant to the Top-K answer. This contributes to higher query latency because of the additional synchronization cost associated with the heap merge phase. GTA-ASP remains work-efficient as it relies on ASP to restrict the answer search space (i.e. variance of tuple attribute values) within a partition. This technique enables stopping earlier, reducing the associated processing workload and query latency. Anti-correlated data are the most difficult to process. Similar to before, the high variance in attribute values results in GTA-RP processing much more data that necessary (up 90% from our measurements). Hence, its query latency is almost comparable to that of GFTE for every experimental parameter. For the same reason as before, GTA-ASP is able to adapt well exhibiting 1.7× to 4× better query latency compared to GFTE.

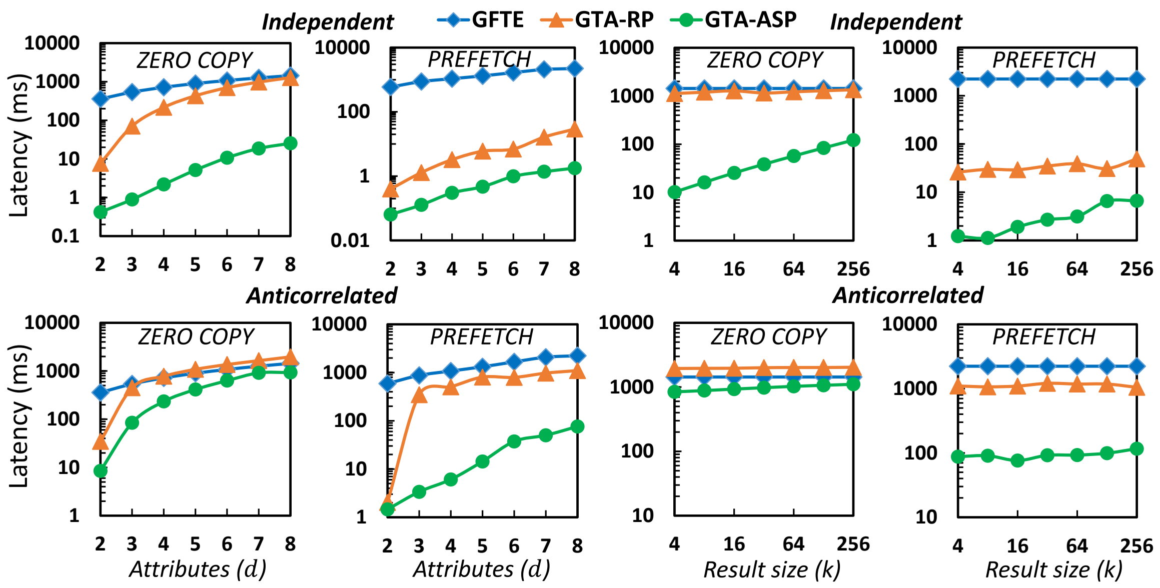


Figure 4.14: Host Memory Query Performance.

4.5.4 Host Memory Query Processing

In Fig. 4.14, we depict the query latency measurements for all developed algorithms on different data distributions when the base relation resides in host memory. We concentrate on the independent and anti-correlated distributions since they are the most challenging to process. For independent data, data prefetching as opposed to accessing them during evaluation is beneficial for both GTA variants. However, it is occasionally worse when accessing all the tuples (i.e. GFTE). We traced this behavior back to an excessive number of GPU page faults. In fact, we observed occasionally 2× the amount of the necessary data (i.e. 16GB) being transferred across PCIe. We estimate that the GPU driver is unsuccessful in detecting temporal locality during query processing, so it resorts into moving data back and forth from the host memory. On anti-correlated data, we observe similar behavior with prefetching being the best option to speed-up processing. GTA-RP

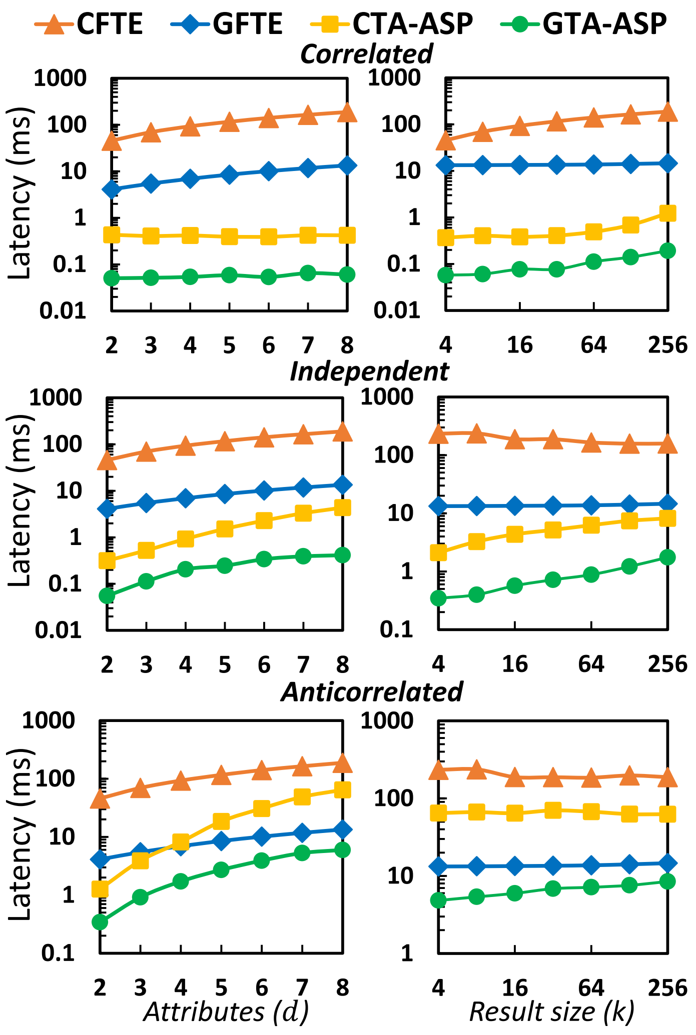


Figure 4.15: Query latency comparison against CPU.

requires processing more data blocks to find the Top-K answer and outperforms GFTE occasionally. GTA-ASP performs on average 22× to 40× better than GTA-RP. The reason is because it requires fetching less data from host memory while most of it is already cached in GPU memory due to prefetching.

4.5.5 CPU Performance Comparison

In this section, we compare the performance of CPU Top-K selection against GPU Top-K selection. We developed two CPU methods; one that evaluates the score of all tuple for the target relation (CFTE), and another that relies on ASP and data re-ordering to enable early termination during query processing (CPU-ASP). Both methods utilize multi-threading and AVX instructions to improve processing throughput. Every CPU thread keeps track of the k-highest ranking tuples in a private priority queue, merging them only towards the end of query evaluation.

In Figure 4.15, we summarize the results attained for varying query parameters and data distributions. Overall, early termination methods using ASP and data re-ordering on CPU and GPU outperform solutions that rely on full evaluation because they require less tuple evaluations to compute the query answer. In fact, early termination on CPU is able to outperform the full table evaluation algorithm on GPU despite the latter being heavily optimized and while having higher bandwidth and compute capabilities. This behavior is consistent for our experiments with correlated and independent data, showcasing improve query latency by a factor of at least 100$\times$ and 30$\times$ respectively. On anticorrelated data, it is more challenging to enable early termination because every tuple contains at least one relatively high ranking attribute. In this case, the performance of early termination drops noticeably for both CPU and GPU implementations because about 50% of all tuples are evaluated. Despite this behavior, early termination improves query latency at least 2$\times$ for both CPU and GPU solutions compared to full evaluation. In fact, subqueries referring to 2 or 3 attributes experience up to 10$\times$ lower query latency in either architecture.

4.6 Conclusions

In this section, we developed the skeleton of parallel threshold algorithms that were optimized to enable GPU accelerated Top-K selection with support for early termination. We considered two different data partitioning strategies, evaluating their effectiveness on various data distributions and query parameters. Our empirical results showcased that data preordering when combined with angle space partitioning is superior in terms of tuple evaluations compared against random partitioning. Experiments with queries that were evaluated on device memory resident data showcased $2\times$ to $100\times$ better query latency against the state-of-the-art solution that relied on evaluating the complete relation. In addition, our experiments on queries that were evaluated on host memory resident data showcased that our methods are very effective when combined with prefetching and related caching strategies. For these experiments, we showcased $10\times$ to $1000\times$ better query latency as opposed to a full table evaluation algorithm. Finally, we implemented our methods on multi-core CPUs and demonstrated proportional performance improvements compared to the corresponding state-of-the-art full table evaluation solution utilizing priority queues.

Chapter 5

Processing-In-Memory Architectures

5.1 Introduction

Modern processors leverage the integration of many compute cores and deep cache hierarchies on a single chip to mitigate the effects of processing large dataset. Despite these efforts, the widening gap between memory and processor speed contributes to a high execution time, as the maximum attainable throughput is constrained by the data movement. Processing-In-Memory (PIM) architectures [3, 31, 37, 40, 58, 71, 83, 88, 93, 103] present a viable alternative for addressing this bottleneck leveraging on many processing cores that are embedded into DRAM. Moving processing closer to where data reside offers many advantages including but not limited to higher processing throughput, lower power consumption and increased scalability for well designed parallel algorithms. In this work we rely on UPMEM's architecture [58], a commercially available PIM implementation that incorporates several of the aforementioned characteristics. UPMEM's architectural implementation follows closely the fundamental characteristics of previous PIM systems [3, 31, 37, 40, 58, 71, 83, 88, 93, 103], offering in addition an FPGA-based testing environment [94]. In this chapter, we utilize UPMEM's PIM accelerator and develop algorithmic solutions for Top-K and Skyline Selection. In Top-K selection, we provide a thorough discussion involving full table evaluation as well as early termination solutions. For Skyline selection, we design a load balanced algorithm suitable for PIM architectures and include a comprehensive evaluation comparing against state-of-the-art multi-core CPU and many-core GPU solutions.

5.2 Architecture Overview

UPMEM's Processing-In-Memory (PIM) technology promotes integration of processing elements within the memory banks of DRAM modules. UPMEM's programming model assumes a host processor (CPU), which acts as an orchestrator performing read/write operations directly to each memory module. Once the required data is in-place, the host may initiate any number of transformations to be performed on the data using the embedded co-processors. This data-centric model favors the execution of fine grained data-parallel tasks [58]. Figure 5.1 illustrates the UPMEM's PIM architecture.

A memory system of 128 GB provides access to 2048 embedded processors called Data Processing Units (*DPUs*) having a total maximum of 49152 operating threads and 2 TB/s data bandwidth to DRAM. Depending on the number of DIMMs, it is possible to have hundreds of *DPUs* operating in parallel. Each one owns 64 MBs which are part of the collective DRAM capacity, referred to as Main RAM (i.e. *MRAM*). The UPMEM DPU is a triadic RISC processor with 24 32-bits registers per thread. The *DPU* processors are highly multi-threaded, supporting a maximum of 24 threads. Fast context switching allows for effective masking of memory access latency[1]. Dedicated Instruction RAM (IRAM) allows for individual *DPUs* to execute their own program as initiated by the host. Additionally, each *DPU* has access to a fast working memory (64 KB) called Work RAM (WRAM), which is used as a cache/scratchpad memory during processing and is globally accessible from all active threads running on the same *DPU*. This memory can be used to transfer blocks of data from the MRAM and is managed explicitly by the application.

In order for PIM systems to operate at peak processing throughput, all participating embedded processors are required to operate in isolation with minimal data exchange. It is important to note that different PIM system configurations that exhibit varying levels of isolation are possible and can be classified accordingly. UPMEM's PIM is an example of physical isolation, not allowing direct communication between compute nodes requiring instead for the host CPU to be involved. PIM configurations based on 3D stacked memory (known also as Processing Near Memory(PNM) systems) utilize a Network-On-Chip(NoC) to enable support for direct access to neighboring physical memory partitions without any involvement from the host CPU [32]. Each physical memory partition can be classified as local or remote partition depending on their proximity to the corresponding embedded processor [32]. This organization indicates a form of logical isolation between the corresponding

[1]Switching is performed at every clock cycle between threads

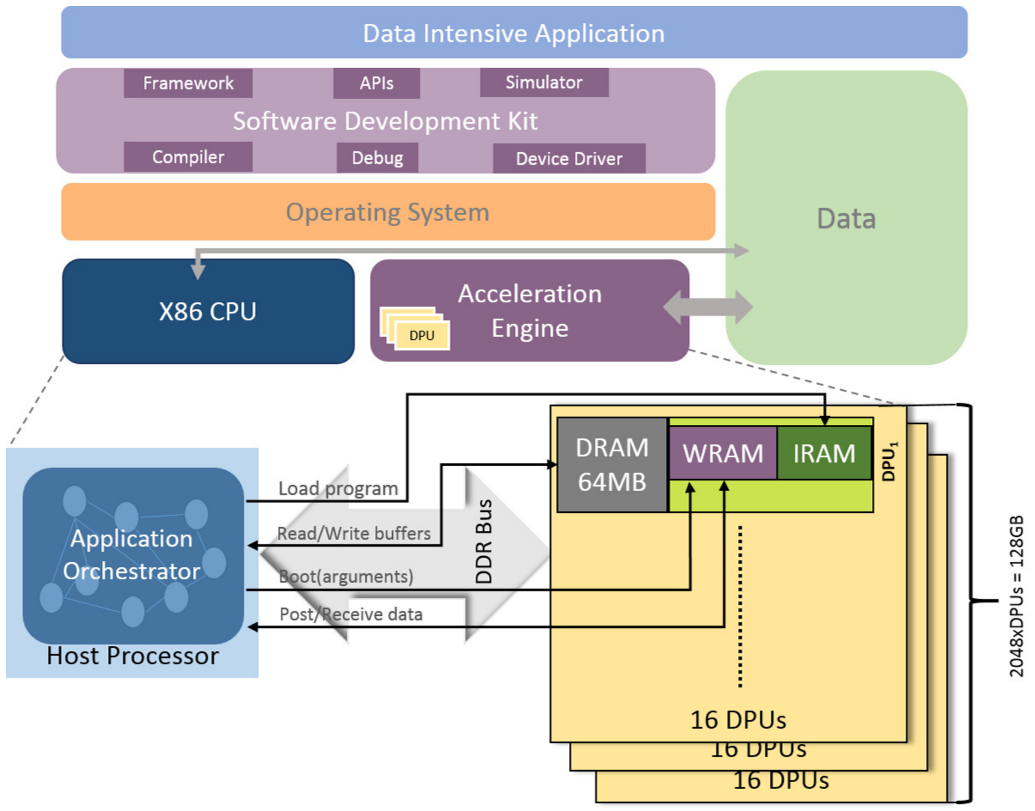


Figure 5.1: UPMEM's PIM Architecture Overview

processors affecting memory access latency since local memory partitions are significantly faster to access than a remote one [32]. Our algorithmic solution is structured around the provision of an efficient partitioning schema that enables opportunities for masking the communication overhead associated with either types of logical or physical isolation which are apparent in most PIM systems, regardless of configuration specifics.

From a programming point of view, two different implementations must be specified: (1) the host program that will dispatch the data to the co-processors' memory, sends commands, and retrieves the results, and (2) the *DPU* program/kernel that will specify any transformations that need to be performed on the data stored in memory. The UPMEM architecture offers several benefits over conventional multi-core chips including but not limited to increased bandwidth, low latency and massive parallelism. For a continuously growing dataset, it can offer additional memory capacity and proportional processing throughput since new DRAM modules can be added as needed.

5.2.1 Performance Validation

Due to lack of an actual hardware implementation (at the time of this thesis UPMEM had not publicly released their PIM hardware), we used UPMEM's cycle accurate simulator (validated through an FPGA emulation) to evaluate the performance of our proposed solutions. The simulator was designed and distributed by UPMEM and is publicly available upon request through their website [95]. It operates by taking as input the binary of an application that was generated using UPMEM's SDK [94] and compiler, and calculates the expected number of clock cycles required for completing the corresponding task. These clock cycles include time spend in the DPU pipeline as well as waiting for data to arrive from the DPU memory upon an data access request.

Chapter 6

Accelerated Top-K Selection on Processing-In-Memory

6.1 Introduction

Typically, Top-K selection involves two steps: (A) the Score aggregation step, where the individual scores of every tuple are calculated by utilizing the provided aggregation function, (B) the Ranking step, where the k-highest ranking tuples are identified by using either a sorting or a K-selection algorithm (i.e. radix-select, bitonic top-K). Considering the aforementioned processing steps, it is apparent that Top-K selection is inherently memory bound. This happens when the k value is large and the corresponding query involves many attributes. Hence, such an operator stands to achieve a significant performance improvement from utilizing processing-in-memory (PIM) architectures. This happens because PIM architectures are suitable for memory bound applications since they provide low data access latency and high processing bandwidth.

PIM accelerators offer massive parallelism and high processing bandwidth, but unlike GPUs they rely on processing units (PUs) which operate in total isolation relying on the host processor for coordination and data exchange. For this reason, the process of migrating applications on PIM systems is not as straightforward as designing a massively parallel algorithm similar to that of a GPU or another type of an accelerator. One must consider partitioning, load balancing and data exchange issues that might arise during the design phase. In this chapter, we investigate possible solutions which are geared towards evaluating efficiently Top-K queries using PIM systems. We investigate two different approaches: (1) Top-K queries using Full Table Evaluation (FTE), (2) Top-K queries with support for early termination. The former approach refers to solutions which require evaluating all the tuples from a target relation and utilize sorting or k-selection to identify those with the highest ranking. The latter approach relies on data reordering and some type of intelligent partitioning to enable processing that enables efficient early termination. In the following sections, we present a discussion analyzing first the full table evaluation solutions. We attempt to establish a fair baseline which can be used to compare against the early termination strategies that we presented in the previous chapters. We conclude by providing an extensive performance evaluation comparing the advantages and disadvantages for each one of the proposed solutions.

6.2 Top-K Selection using Full Table Evaluation

As mentioned in previous chapters, a straightforward solution to the Top-K selection problem involves two steps: (1) calculate the score for each tuple (i.e. score aggregation step) (2) utilize a sorting algorithm (see Section 6.2.1) or a priority queue (see Section 6.2.2) to identify those having the k highest scores (i.e. ranking step). Score aggregation is easy to implement on PIM accelerators since it is an embarrassingly parallel problem which requires evenly partitioning the data across the available Data Processing Units (DPUs). The ranking step is a little bit more complex because it requires optimizing calculations within every one of the participating DPUs and minimizing data exchange across them. In fact, when utilizing sorting for ranking the corresponding tuples, it is extremely important to optimize data movement during the phase in which the associated data need to be re-arranged across the participating DPU. In the following sections, we study and develop optimized sorting and k-selection algorithms for PIM accelerators and compare them against the equivalent state-of-the-art multi-core algorithms. In addition, we provide a comprehensive experimental evaluation aimed at identifying which method works best for Top-K selection.

6.2.1 Sorting on PIM Systems

Sorting a collection of keys on a PIM architecture can be achieved in two different ways. One way is to employ a comparison based approach that utilizes DPUs to create sorted sequences and the host processor to merge those sequences into a single sorted data collection. Another way is to implement a non-comparison based solution where the host processor is responsible for rearranging the corresponding keys in the associated bins (DPUs) while the available DPUs take the lead in counting digit/key occurrences locally. Both local sort and digit counting are embarrassingly parallel operations being able to take advantage of all available DPUs offered by UPMEM's PIM architecture. Merging sorted sequences on the host processor is expensive for two reasons: (1) it requires an additional buffer to store intermediate results during the actual merging, (2) it exposes limited opportunities for parallelism as the number of sorted sequences decrease. Finally, rearranging (also known as scattering) keys is less expensive though still being bounded by the write throughput of the host processor. In the following sections, we present the details for both of the aforementioned approaches providing a comprehensive complexity analysis aimed at identifying which solution is the most suitable for implementation on PIM systems.

Comparison Based Sorting Algorithms

Utilizing a comparison based method to sort a collection of keys requires first partitioning the data evenly across every available DPU. Load balancing is not really an issue even for skewed data distributions because, we can ensure that every DPU receives roughly the same number of keys. In fact, given P DPUs and having to sort N keys, the i-th DPU will receive the keys between the offsets calculated by utilizing the following equations:

$$start\_offset = \left\lfloor \frac{i \cdot N}{P} \right\rfloor \tag{6.1}$$

$$end\_offset = \left\lfloor \frac{(i+1) \cdot N}{P} \right\rfloor \tag{6.2}$$

Sorting the data locally can be achieved by utilizing different algorithms. Picking the most suitable for implementation on UPMEM's PIM architecture requires taking into consideration the corresponding processing model and hardware resources available to a single DPU. First, the processing model dictates that at least 10 threads should be active during processing to enable full utilization of the 10 stage DPU pipeline. Second, the DPU programming model follows the principles of a cache-less architecture requiring from the programmer to utilize buffers to access data from the main memory. A limited capacity scratchpad memory (known as WRAM) is used to store and manage those buffers.

Algorithm 10 Parallel Bitonic Sort

```
A = Input array of N unsorted keys.

1: $tid = me()$                                          ▷ Tasklet id.
2: for $len \leftarrow 1; len < N; len \leftarrow len << 1$ do
3:     $dir = len << 1$
4:     for $step \leftarrow len; step > 0; step \leftarrow step >> 1$ do
5:         for $t \leftarrow tid; t < (N >> 1); t \leftarrow t \leftarrow t + N_T$ do
6:             $low \leftarrow t\&(step - 1)$
7:             $i \leftarrow t << 1 - low$
8:             $reverse \leftarrow ((dir\&i) == 0)$
9:             $swap = reverse \oplus (A[i] < A[i + step])$
10:            if $swap$ then $(A[i], A[i + step]) \leftarrow (A[i + step], A[i])$
11:        end for
12:        barrier_wait()                    ▷ Wait for swapping to finish.
13:    end for
14: end for
```

Considering the previous limitations we have identified that the best candidates for implementation on DPUs are bitonic sort and merge sort. Bitonic sort is suitable for a parallel environment having no requirements for additional memory. However, it requires frequent barrier synchronizations to ensure read after write consistency. As mentioned previously, merge sort requires additional memory to buffer the corresponding data during the merging phase. In addition, it is difficult to parallelize when only few sorted sequences are available during the merging phase.

Algorithm 10 presents a high level overview of the bitonic sort DPU kernel. Typically, for an array of N keys, $N_T = N/2$ threads are required. However, a DPU can have at most 24 threads (tasklets) executing at the same time. In our DPU kernel, the required number of threads are emulated by increasing the corresponding thread-id by fixed offset (N_T line 5) to get the equivalent thread-id for the required number of operating threads. This enables executing the steps of bitonic sort with less than $N/2$ threads. In addition, it assigns more work to every participating thread minimizing the number of barrier synchronization calls (Line 12) executed in between the swapping operations. As indicated by the associated algorithm, there is no need for extra space because swapping executes in-place. In addition to the costly barrier synchronizations which are necessary to ensure read after write consistency, bitonic sort incurs higher algorithmic complexity which is equal to $O(Nlog^2N)$.

Despite requiring additional memory to perform the merging phase, merge sort attains lower algorithmic complexity that is equal to $O(NlogN)$. Algorithm 11 summarizes the individual steps required for executing merge sort. In addition to the collection of unsorted keys, the algorithm takes as input a buffer in which the merging results are stored. We utilize pointers pointing to the left and right sub-arrays (Lines 3-4) which are scheduled to be merged together. The corresponding values are compared and the minimum

Algorithm 11 Merge Sort

$A =$ *Input array of* N *unsorted keys.*

$B =$ *Buffer array.*

```
1: for $i = 1; i < N; i = i << 1$ do
2:     for $j = 0; j < N; j = j + (i << 1)$ do
3:         $(L_s, L_e) = (A + j, A + j + i)$        ▷ Pointer to start & end of left sub-array.
4:         $(R_s, R_e) = (L_e, L_e + (i << 1))$        ▷ Pointer to start & end of right sub-array.
5:         $B_s = B + i$        ▷ Buffer pointer start.
6:         while $L_s < L_e$ AND $R_s < R_e$ do
7:             if $^*L_s < {}^*R_s$ then
8:                 $^*B_s{+}{+} = {}^*L_s{+}{+}$        ▷ Write $L_s$ value to buffer and increment pointer.
9:             else
10:                $^*B_s{+}{+} = {}^*R_s{+}{+}$        ▷ Write $R_s$ value to buffer and increment pointer.
11:            end if
12:        end while
13:        while $L_s < L_e$ do $\{^*B_s{+}{+} = {}^*L_s{+}{+}\}$        ▷ Flush left sub-array to buffer.
14:        while $R_s < R_e$ do $\{^*B_s{+}{+} = {}^*R_s{+}{+}\}$        ▷ Flush right sub-array to buffer.
15:    end for
16:    swap($A, B$)
17: end for
```

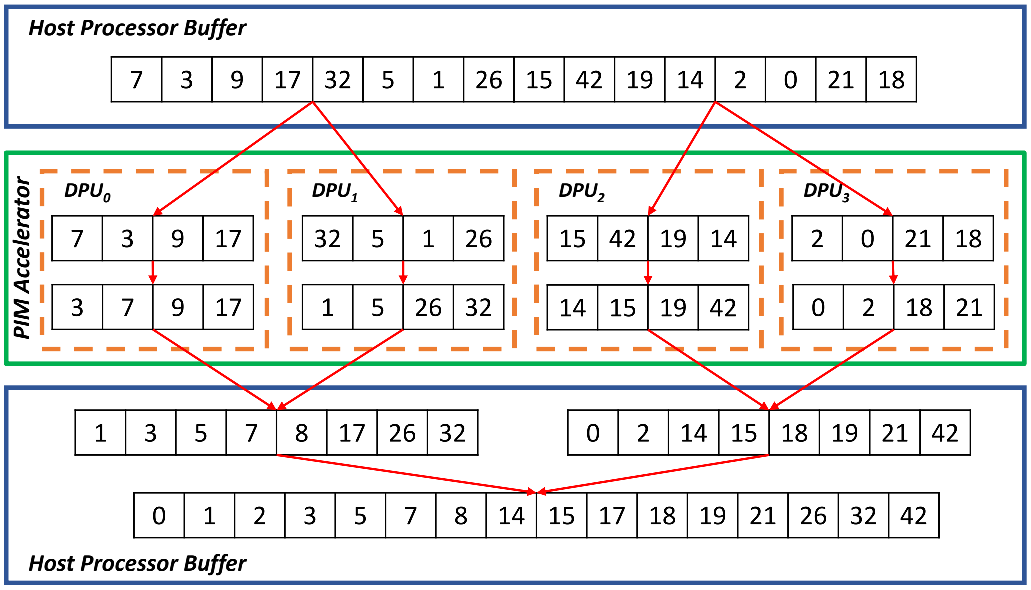


Figure 6.1: Local comparison sort on 4 DPUs combined with the host processor merging phase to produce a globally sorted sequence.

(maximum) (Lines 8,9) is selected to rearrange the associated data creating an ascending (descending) sequence of values. This process executes across $logN$ iterations creating iteratively a larger sorted sequence. At the end of every one of the $logN$ merge iterations (Line 15), the input array and buffer pointers are swapped (Line 16) to avoid expensive copy operations between the two.

The aforementioned algorithms operate locally within each DPU resulting in the creation of many distinct sorted sequences. Those sequences will be merged by the host CPU to produce a globally sorted sequence. An example of this process operating end to end to produce a globally sorted sequence is shown in Figure 6.1. Initially, the data reside in a buffer that is accessible both from the CPU and PIM accelerator. The host CPU is responsible for partitioning the data and assigning distinct batches for processing to every available DPU. This requires only transmitting the offset at the start of the corresponding data block. The DPU is able to access the buffered data directly and execute its preferred sorting algorithm (merge sort/ bitonic sort). Once local sort completes the corresponding DPU informs the host CPU using a zero overhead completion message. The host CPU will wait for all DPUs to finish before starting the merge phase. Merging can be performed using the same hierarchical approach as described either in merge sort or bitonic sort. While the former solution attains lower complexity the latter is highly parallel and should be chosen for multi-core CPUs. However, the performance of both solutions is restricted by the available data bandwidth.

Non-Comparison Based Sorting Algorithms

Non-comparison based sorting algorithms operate by counting the number of key occurrences in the input sequence using arithmetic operations and use that information to determine the offset for the position of the corresponding key in the output sorted sequence. Counting sort is the simplest example of a non-comparative based sorting algorithm As opposed to other efficient comparison based sorting algorithms, such as merge-sort, the complexity of counting sort is proportional to the size of the given input sequence and the difference between the maximum and minimum values within that sequence. Therefore, it is often suitable for sequences where the variation in key values is not significantly greater than the input size.

Radix-sort is another example of a non-comparative based sorting algorithm. It is more efficient than counting sort and can handle key values of arbitrary distribution. The algorithm has been implemented on different parallel architectures including multi-core

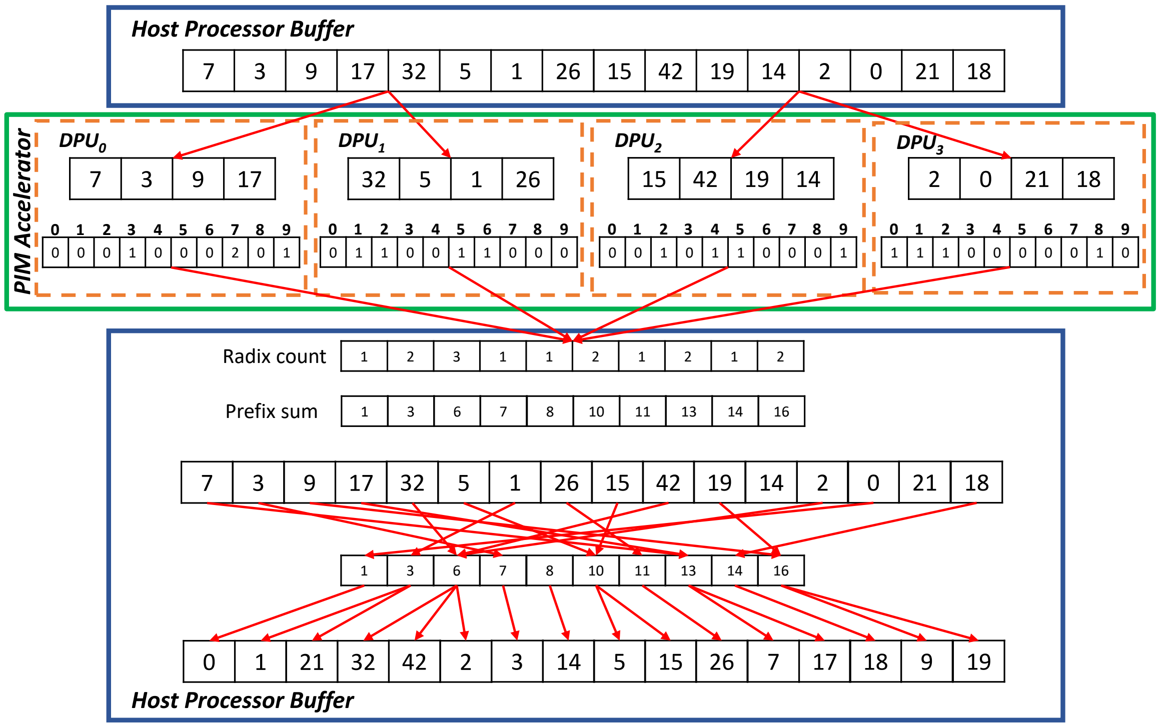


Figure 6.2: Radix sort example

CPUs and many-core GPUs. Radix-sort operates on the input sequence by counting the occurrences of non-overlapping fixed-sized digit groups. The size of the group is referred to as *radix*, which is often set of 4 or 8-bits starting from the least significant bit when sorting in ascending order. The counting step is similar to creating a histogram which is then used to rearrange the input sequence into one that is partially sorted. Counting digit occurrences and re-arranging them execute in sequence across different iterations until the sequence is fully sorted.

An example showcasing the interaction between the host CPU and each individual DPU which includes merging the counts and re-arranging the keys is show in Figure 6.2. Initially the data are loaded into memory and are accessible directly from both the host

Algorithm 12 Radix Count DPU Kernel

$A =$ *Input array of N unsorted keys.*

$P =$ *Total number of tasklets.*

$m =$ *Mask to extract corresponding digit.*

$shf =$ *Shift places to find digit occurrence*

```
1: $t =$ getTaskletId()
2: $pb = A + \lfloor (N * t)/(P) \rfloor$    ▷ Pointer to the beginning of data batch for tasklet $t$.
3: $pe = A + \lfloor (N * (t + 1))/(P) \rfloor$    ▷ Pointer to the ending of data batch for tasklet $t$.
4: $mBuff = buff + (t << 9)$    ▷ Pointer to buffer beginning for tasklet $t$.
5: $mBins = bins + (t << 8)$    ▷ Pointer to bins beginning or tasklet $t$.
6: while $pb < pe$ do
7:     $mram\_read(pb, mBuff)$
8:     $mram\_read(pb + 256, mBuff + 256)$
9:     for $i = 0; i < 512; i++$ do
10:        $d = (mBuff[i]\ \&\ m) >> shf$    ▷ Extract digit.
11:        $mBins[d]++$    ▷ Increment the bin value corresponding to the extracted digit.
12:    end for
13:    $pb += 512$
14: end while
```

CPU and all available DPUs. We assign for processing non-overlapping continuous batches of data to distinct DPUs. Every DPU will calculate their own digit count and inform the host CPU when processing ends. The resulting bins will be aggregated by the CPU who can access directly the individual bins from DRAM. After aggregating the corresponding bins, a prefix sum operation executes calculating the associated offsets. The host CPU will be responsible for scanning the input vector and using those offsets to rearrange the values as shown in the aforementioned figure.

Algorithm 12 summarizes the steps executed on the DPU. The pseudocode describes the occurrences of the corresponding digits extracted by the associated mask (m). Each DPU is assigned a distinct batch of data (which is stored in the DPU MRAM) and is responsible for counting the associated digit occurrences. Every available DPU tasklet is responsible for processing a continues sequence of values between pb and pe pointers (Line

Algorithm 13 Merge Bins DPU Kernel

```
$P$ = Total number of tasklets.
$r$ = User defined radix value.

1: $t = getTaskletId()$
2: for $i = 1; i < P; i++$ do
3:     for $j = t; j < P \cdot 2^r; j+= P$ do
4:         $bins[j] += bins[(i \cdot P \cdot 2^r) + j]$        ▷ Merge counts to first 16 bins.
5:     end for
6: end for
```

Algorithm 14 Host CPU Key Rearrange

A = *Input array of unsorted keys.*

B = *Output array of unsorted keys.*

N = *Number of input keys.*

$bins$ = *Merged bin counts.*

m = *Mask to extract corresponding digit.*

shf = *Shift places to find digit occurrence.*

```
1: for $i = N - 1; i >= 0; i - -$ do
2:     $d = (A[i] \ \& \ m) >> shf$                ▷ Extract digit.
3:     $pos = - - bins[d]$                      ▷ Calculate output offset.
4:     $B[pos] = A[i]$                          ▷ Copy element to new location.
5: end for
6: swap($A, B$)                                 ▷ Swap array pointers for next iteration.
```

2,3). We allocate space in WRAM to hold a buffer for reading the data from MRAM (Line 4) and another to store the digit occurrences (Line 5). In the pseudocode example, we assumes 16 tasklets and radix 4 which leaves enough space to store a collection of $2^4 = 16$ bins and buffer 512 values in WRAM (assuming 32-bit keys). The counting of the corresponding digit occurrences (Line 10, 11) proceeds after the data have been loaded in the corresponding buffers (Line 7,8). When all the data in the buffers have been loaded, we increment the corresponding pointers (Line 13) to process the next sequence of values. When the counting of digit occurrences completes, we merge the bin values using tasklet 0. This pseudocode for merging is shown in Algorithm 13. Every tasklet is responsible for aggregating the results of the same digit across all bins in the collection. Once this operation completes the host is responsible for merging the final count across different DPUs and using it to re-arrange the data before the next round of counting begins.

Algorithm 14 summarizes the host CPU steps required for rearranging the key elements after globally merging the individual bin counts. The algorithm operates by iterating through each element in the input array (Line 1) and extracting the corresponding digit (Line 2). The extracted digit is used to index the bins and calculate the position (Line 3) of the given element in the output array. The corresponding element is copied to that position (Line 4) and processing continues to the next available element. At the end of processing, the input and output array pointers are swapped to prepare for the next stage of processing.

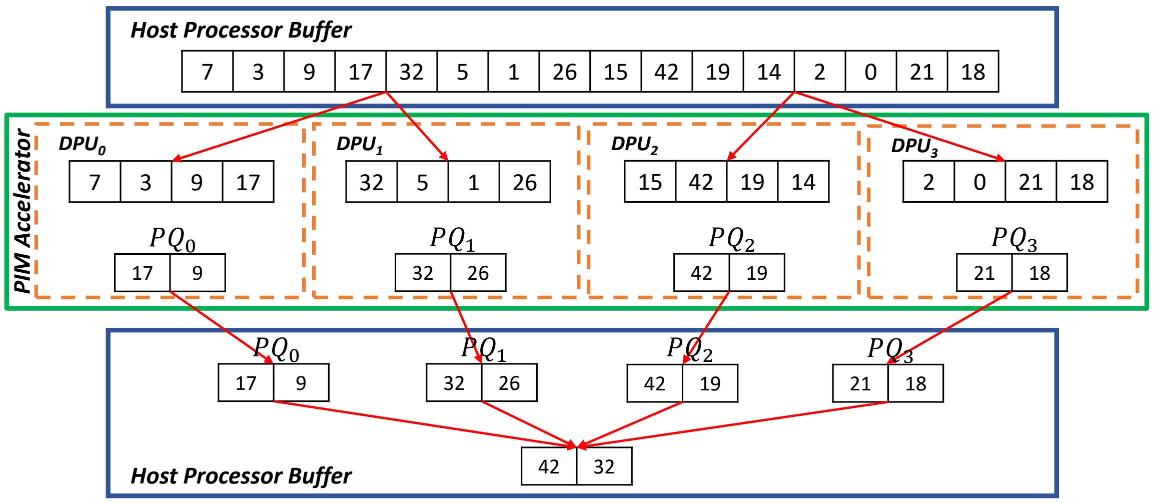


Figure 6.3: Multiple priority queue Top-K using host CPU for merging intermediate results.

6.2.2 Top-K Queries Using Multiple Priority Queues

An alternate way of evaluating Top-K queries is by utilizing a priority queue (i.e. max heap) which is used to maintain the k-highest ranking tuples during processing. This approach is beneficial because it requires scanning the input data only once while also enabling processing within on-chip memory when the corresponding k value is small (this is a common occurrence since k often ranges between 10 and 100).

On massively parallel architectures that follow the multiple instruction multiple data (MIMD) paradigm, it is common to utilize a single priority queue per available worker and merge all intermediate results at the end of processing. This paradigm is applicable on UPMEM's PIM accelerator enabling higher levels of parallelism and higher processing throughput. In the aforementioned configuration, the host CPU will be responsible for efficiently merging the intermediate results. In practice the merging operation is relatively cheap because the chosen k value is often small.

In Figure 6.3, we provide an example showcasing the individual steps associated with discovering the Top-K value from a given input vector. Following the sorting paradigm, we assign for processing non-overlapping continuous batches of data to distinct DPUs. Each DPU operates on its own local memory space which contains space for storing the priority queue data. The priority queue is used to store the k highest values during processing of the input data batch. Since each DPU encapsulates multiple operating threads (i.e. tasklet) and in order to ensure thread safety, we created an implementation where each thread operates on a different priority queue. This approach incurs an additional cost for merging and has a higher memory overhead. However, we found experimentally that enabling synchronized access (using mutexes) to a single priority queue limits the maximum attainable DPU throughput. Hence, our decision to implement a solution that utilizes a different priority queue per tasklet. After the data assignment phase, the available DPUs will begin processing their own chunk of data in parallel. Once processing completes the host processor will be informed in order to start the merging phase. Note that the merged priority queue of every DPU is stored in MRAM which is directly accessible by the host CPU. Therefore, there is no additional cost of moving the data where the CPU can access it. Merging is straightforward and can be implemented in a number of different ways. Our implementation iterates over all available priority queues each time pulling the maximum item and storing it on a CPU owned priority queue until k items have been discovered.

Algorithm 15 summarizes the steps associated with computing the Top-K values within every available DPU. Every DPU tasklet is responsible for processing a small chunk of the data input which is assigned to the given DPU (Line 2-3). A buffer local to each

Algorithm 15 Top-K using Many Priority Queues - DPU Kernel

A = *Input array of unsorted keys.*

P = *Total number of tasklets.*

$mBuff$ = *Tasklet local buffer.*

$mHeap$ = *Tasklet local priority queue*

$gHeap$ = *Pointer to global heap collection.*

```
1:  $t = getTaskletId()$
2:  $pb = A + \lfloor (N * t)/(P) \rfloor$                ▷ Pointer to the beginning of data batch for tasklet $t$.
3:  $pe = A + \lfloor (N * (t+1))/(P) \rfloor$           ▷ Pointer to the ending of data batch for tasklet $t$.
4:  while $pb < pe$ do
5:      $mram\_read(pb, mBuff)$                        ▷ Read 256 items to local buffer from $pb$ address.
6:      for $i = 0; i < 256; i++$ do
7:          if $mBuff[i] > mHeap[k-1]$ then            ▷ Condition to update heap.
8:              $mHeap.pop()$                          ▷ Pop smallest item.
9:              $mHeap.pushMax(mBuff[i])$              ▷ Push new item.
10:         end if
11:     end for
12:     $pb += 256$
13: end while
14: $barrier\_wait()$                                  ▷ Wait for swapping to finish.
15: If $t == 0$ then $rHeap = merge(gHeap)$            ▷ Merge individual heaps.
```

tasklet is used to load the data from MRAM (Line 5). In addition, a local heap is maintain containing the k-highest values. An item from the buffer will be inserted in the local heap if and only if its value is larger than the current heap minimum (Line 7). The new item is inserted into the heap (Line 9) after deleting the minimum (Line 8). Inserting new items into the corresponding max heap is a relatively cheap since it incurs a complexity of at most $O(logk)$. All of the required operations execute in WRAM which is much faster to access as opposed to MRAM. In order to ensure read after write consistency, we utilize a barrier (Line 14) to synchronize before attempting to merge the global heap collection into a single heap (Line 15). Tasklet 0 is responsible for executing the final merge operation which creates a single heap containing the k-highest ranked elements.

Algorithm 16 summarizes the steps of the standard heap merge algorithm. The algorithm iterates over all non-empty heaps in order to discover the one containing the largest element (Line 4 - 8). The corresponding element will be inserted into the result heap (Line 10) and removed from global heap collection (Line 11). This process repeats until the result heap contains k items (Line 2). Every time an item is pulled from any heap in the global collection, the corresponding heap needs to be re-balanced. The cost of this is $O(logk)$ while the total complexity of merging is $O(klogk)$.

6.3 Early Termination Top-K Selection on PIM

As discussed in previous chapters, the most expensive part of Top-K query evaluation is the process of retrieving the associated tuple attributes to calculate the corresponding tuple score. In order to mitigate this cost indexing and data re-ordering strategies are em-

Algorithm 16 Heap Merge - DPU Kernel

```
gHeap = Pointer to global heap collection.
P = Total number of tasklets.
1: $rHeap = Init()$
2: while $rHeap.size() < k$ do
3:     $pos = 0$
4:     for $i = 0; i < P; i++$ do
5:         $cHeap = gHeap[i]$
6:         if $gHeap[pos][0] > cHeap[0]$ then          ▷ Discover maximum element heap.
7:             $pos = i$
8:         end if
9:     end for
10:    $rHeap.push(gHeap[pos][0])$          ▷ Insert maximum element to result heap.
11:    $gHeap[pos][0].popMax()$          ▷ Pop maximum element.
12: end while
13: return $rHeap$
```

ployed to enable early termination during query evaluation. These strategies produce the exact answer to the given Top-K query while managing to significantly reduce the associated evaluation workload. In this section, we present a discussion focused on how to effectively adopt the best practices (as established in previous chapters) geared towards efficient Top-K selection with early termination, on processing-in-memory architectures.

6.3.1 Maximum Attribute Tuple Re-ordering

In order to enable early termination on PIM, we implemented tuple re-ordering based on the maximum attribute value considering all attributes for any given tuple. We reuse the Threshold Block Layout (TBL) configuration for our DPU-based implementation. In contrast to our CPU based implementation, we utilize the radix-sort implementation (introduced in the previous sections) to perform the actual re-ordering. A toy example of a given TBL tree is shown in Figure 6.4. It consists of a collection of ordered non-overlapping data blocks stored as leaves in a self-balancing tree. Each data block contains several tuples along with a threshold tuple that contains the maximum attribute values of any subsequent data block. This configuration resembles a clustered B+tree index thus being easily maintainable either from the host CPU or the corresponding DPU. UPMEM's PIM accelerator implements a message based communication API to received data from the host CPU. This API can be used to initiate low latency insert/update/delete operations on the corresponding TBL tree.

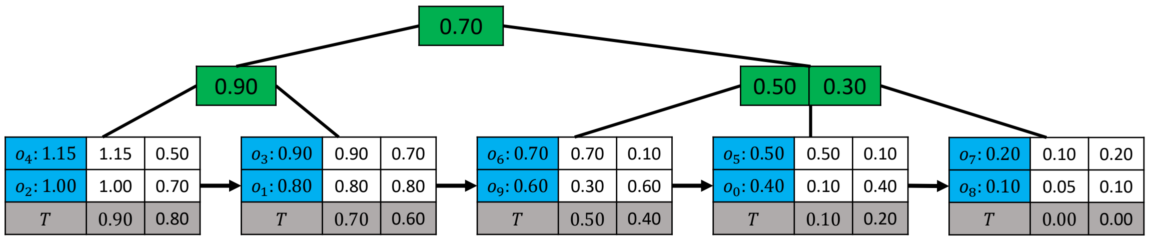


Figure 6.4: Example of TBL Tree Maintaned on a Single DPU.

6.3.2 Multi-DPU Data Partitioning

Intelligent data partitioning is fundamental to enabling efficient early termination for Top-K Selection. On PIM architectures effective data partitioning is extremely important and significantly more challenging. This happens because a given partitioning strategy should be optimized to address issues related to (1) load balancing, and (2) constraints in the memory capacity of the given embedded processor (i.e. DPU).

In the previous chapters, we established that angle space partitioning works better than random partitioning in terms of enabling load balanced and work-efficient Top-K query processing on massively parallel architectures. However, it incurs an additional initialization overhead when building the corresponding index on the target relation. In addition, due to the limited memory capacity available to every DPU (only 64 MB) care must be taken not to create large partitions when the data are skewed. In order to avoid the latter issue, we utilized equi-volume partitioning [96] increasing proportionally the number of available DPUs in order to make sure that the DPU memory capacity will not be exceeded. Doing so will not change the characteristics of our solution thus retaining the associated latency and throughput gains when answering Top-K queries for variable preference vectors and data distributions.

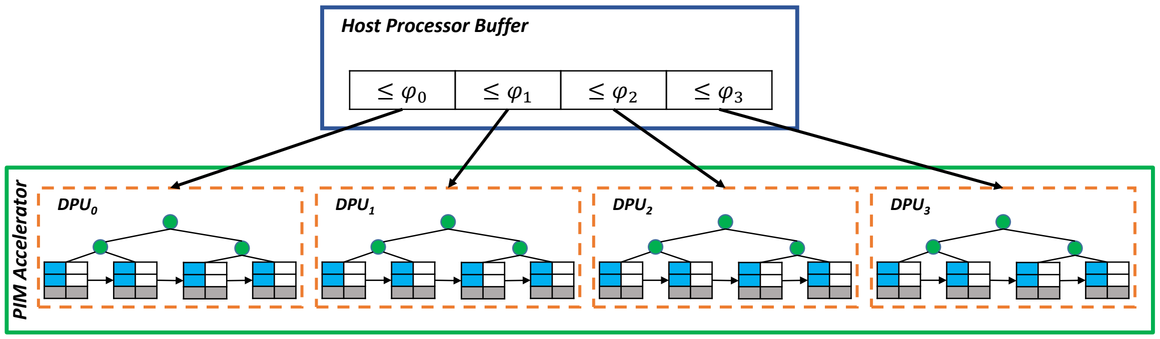


Figure 6.5: Example of a partitioned relation based on 4 angle regions that is represented using multiple distinct TBL trees.

Angle space partitioning is employed on top of the target relation after which the tuples within each partition are re-ordered based on their maximum value. The host processor retains information related to which DPU contains what portion of the space based on the partition angle. An example of this organization is shown in Figure 6.5. In that example, we partition a collection of tuples with two attributes each by splitting the space using four angles, mainly $\phi_0, \phi_1, \phi_2, \phi_3$. The host CPU facilitates insert/update/delete operations utilizing the pointers stored for every angle region. In the case where each tuple contains more than two attributes, multiple angles are needed to describe a given region. Considering that grid partitioning is employed on the angular coordinates, the grid boundaries can be maintained efficiently using R-trees. This organization does not change how the underlying TBL trees are maintained thus the insert/update/delete operations are still supported.

Algorithm 17 DPU Early Termination Kernel

mBuffer = Per tasklet read buffer., mScores = Per tasklet score buffer.

mHeap = Per tasklet heap buffer., cBytes = Bytes in column of attributes.

q = Query attribute count.

```
 1: while $b < block\_count$ do                          ▷ Iterate over data blocks.
 2:     $row = pos + (tid << 8)$           ▷ Column-wise access to the tuple attributes.
 3:     for $(i = 0; i < q; i++)$ do
 4:         $mram\_read256(row, mBuffer)$
 5:         for $(j = 0; j < 64; j++)$ do
 6:             $mScores[j] += mBuffer[j]$   ▷ Accumulate $i$-th attribute of $j$-th tuple.
 7:         end for
 8:         $row += cBytes$
 9:     end for
10:     for $(i = 0; i < 64; i++)$ do $mHeap.pushMax(mScores[i])$        ▷ Push new item.
11:     $mram\_read64(row, mBuffer)$
12:     for $(i = 0; i < q; i++)$ do $th += mBuffer[i]$             ▷ Calculate threshold.
13:     If $mHeap[k-1] \geq th$ then $break$
14:     $pos += row + 64$
15: end while
16: $barrier\_wait()$                                   ▷ Wait for swapping to finish.
17: If $t == 0$ then $rHeap = merge(mHeap)$             ▷ Merge individual heaps.
18: return $rHeap$
```

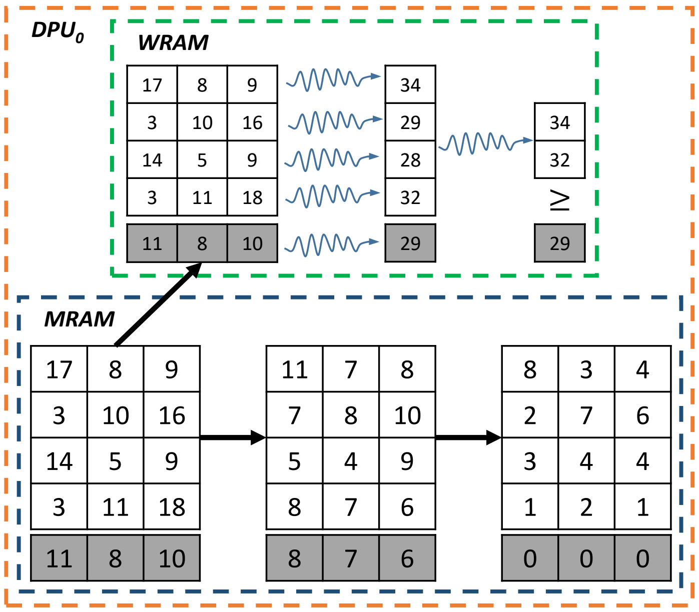


Figure 6.6: Single DPU query evaluation snapshot for top-2 query on a toy dataset.

6.3.3 DPU-Based Query Evaluation With Early Termination

Algorithm 17 summarizes the steps associated with enabling early termination on a single DPU. For every data block stored in DPU's MRAM (Line 1), we calculate the offset address for the corresponding column of attributes (Line 2). This offset is used to retrieve the query attributes in private buffer which is associated uniquely to a specific tasklet (Line 4). The retrieved attributes will then be aggregated to the scores buffer owned by the corresponding tasklet (Line 6). After calculating the scores of a given group of tuples (the size of which is determined by the result size (i.e. $k = 64$ in the example)), we update the private priority queue (Line 10) of every tasklet using the newly calculated tuple scores. In Algorithm 17, we assume that the TBL block size is equal to the result size, thus only one iteration of the main loop (Line 3-10)) is needed to calculate the scores and update the corresponding heap. Once the heap is updated, we load the threshold tuple attributes

(Line 11) and calculate the corresponding threshold (Line 12). If the threshold value is equal or less than the minimum value in the priority queue, we terminate processing (Line 13). At the end of processing tasklet 0 is responsible for merging all heaps together (Line 17).

6.4 Experimental Evaluation

In this section, we present a detailed experimental performance evaluation of all methods developed in the previous sections. The first part of our discussion concentrates on Full Table Evaluation (FTE) methods. Our analysis involves utilizing a series of microbenchmarks to identify which approaches amongst those utilizing sorting or priority queues perform the best. Following this, we compare the performance of early termination against the best FTE method. We conclude with an extensive experimental comparison against optimized FTE and early termination solution across different processing architectures, mainly multi-core CPUs and many-core GPUs.

6.4.1 Experimental Environment

All our experiments where performed on uniformly generated synthetic data. Our CPU implementations utilize 16 threads and where evaluated on a two socket Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz. Our GPU experiments were conducted using a single 12 GB NVIDIA Titan V GPU attached to a single socket Intel Xeon E5-1650 processor @ 3.5 GHz with 32 GB of RAM. The GPU algorithm was implemented using standard C++, CUDA 10.0 and NVIDIA's CUB Library [67]. The DPU kernels used for sorting, implementing the priority queue and our early termination solutions were developed using

	CPU	GPU	PIM
Cores (c)	18	5120	2048
Bandwidth (GB/s)	71	652	2048
Power (W/c)	8.5	0.11	0.04

Table 6.1: Single node specification comparison for CPU (Xeon E5-2686), GPU (TITAN V) and PIM (UPMEM) architectures.

UPMEM's C-based API. We used an AWS instance containing an FPGA-based cycle accurate emulator (emulating the parallel execution of up to 128 DPUs) that was provided by UPMEM to measure the DPU cycles (including the cost of memory access) for all PIM-based approaches. The DPU cycles were converted to ms assuming a DPU clock frequency of $750Mhz$. Every DPU implementation has been designed to utilize 16 tasklets. In Table 6.1, we summarize the specification of each parallel architecture used in our experiments.

In our experiments, we assume result size $k = 16$ for increasing input size, and the maximum input size (i.e. 2^{21} for a single DPU, 2^{30} for multiple DPUs) when presenting measurements for increasing result size. We evaluated the performance of early termination against FTE solutions using queries on 2 to 8 attributes. For increasing result and input size, we present the results gathered when using 8 query attributes.

6.4.2 Single DPU Sorting& Radix Count

In Figure 6.7, we summarize the throughput measurements obtained using a single DPU and 16 operating tasklets for mergesort, bitonic sort, and radix count. We observed

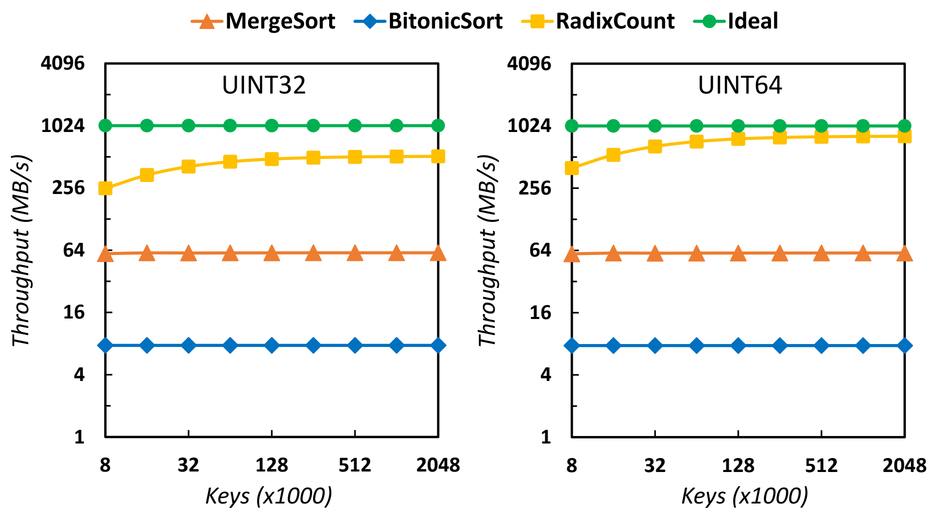


Figure 6.7: Measured throughput for sorting and radix-count using a single DPU on an input vector of 32 or 64 bits.

that regardless of the use sorting method the attainable throughput is much lower than what a single DPU can achieve in the ideal case. This happens because conventional DRAMs are designed to sustain higher write throughput by taking advantage of parallel writes across multiple DRAM chips. DPUs operate in isolation on their own DRAM chip thus their write throughput is limited. Therefore, any local sorting algorithm will be unable to sustain a high processing throughput because it is bounded by the memory chip bandwidth.

Bitonic sort performs worse than merge-sort because it has higher complexity and incurs more writes into MRAM. In addition, it requires too many barrier synchronization to ensure read-after-write consistency when rearranging the corresponding unsorted keys. Mergesort performs better because it makes use of WRAM buffers to sort the data. In the early merge stages, the associated buffers fit entirely in WRAM thus any writes to MRAM are performed efficiently in batches. When the blocks to be merged become too large, we

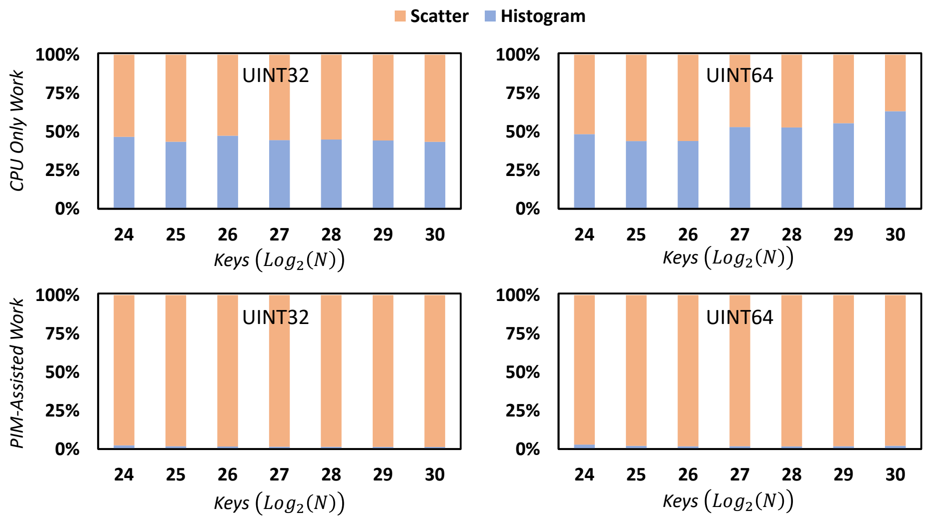


Figure 6.8: Percentage of time spent in re-arranging the unsorted keys (scatter) and calculating the digit histogram for CPU only and PIM-assisted implementation of radix sort.

utilize an external sorting algorithm that writes back to MRAM in smaller batches due to the associated limitations in WRAM memory capacity. The latter portion of mergesort's execution is what limits the maximum processing throughput. Radix-count attains much higher processing throughput because it operates entirely in WRAM when accumulating the corresponding digit occurrences. Commonly, radix-sort is implemented using radix equal to 8 which requires having to manage 256 digit bins. WRAM has more than enough space to accommodate multiple distinct groups of such bins thus enabling multi-tasklet processing. In addition, our implementation uses large buffers to sequentially read in WRAM large chunks of the unsorted key sequence.

Our previous analysis indicates that radix-sort is the best performing solution (in terms of throughput) when sorting a large sequence of keys. For this reason, henceforth we will be using radix-sort as a fair baseline for all the possible sorting based FTE solutions.

6.4.3 Radix-Sort Performance Comparison between PIM and CPU

In this section, we focus on comparing the performance between CPU only radix-sort and PIM-assisted radix-sort implementation. The latter approach is termed "assisted" because the PIM accelerator is responsible only for performing radix-count. In that configuration, the host processor takes charge of rearranging the keys according to the digit occurrences.

In Figure 6.8, we present measurements indicating the percentage of work performed on re-arranging the data (scatter) and calculating the histogram for the CPU only and PIM-assisted radix-sort implementations We observe from the CPU only implementation that the total processing time for radix-sort is split in half between data re-arranging and histogram calculations. The PIM-assisted approach spends less than 2% of its time in calculating the histogram. These measurements indicate that we should expect an improvement in the total execution time of radix-sort that is capped close to 50%

Figure 6.9 validates our expectations by showcasing an average improvement of about 2× when comparing the latency of our PIM-assisted solution over the CPU only implementation. When 64-bit keys are used and the input size is large, the observed improvement in the total execution time is more than 2×. This happens because the cost of calculating the histogram is more that 50% (as depicted Figure 6.8) and the associated read throughput achieved by UPMEM's PIM is several orders of magnitude higher than the host processor throughput.

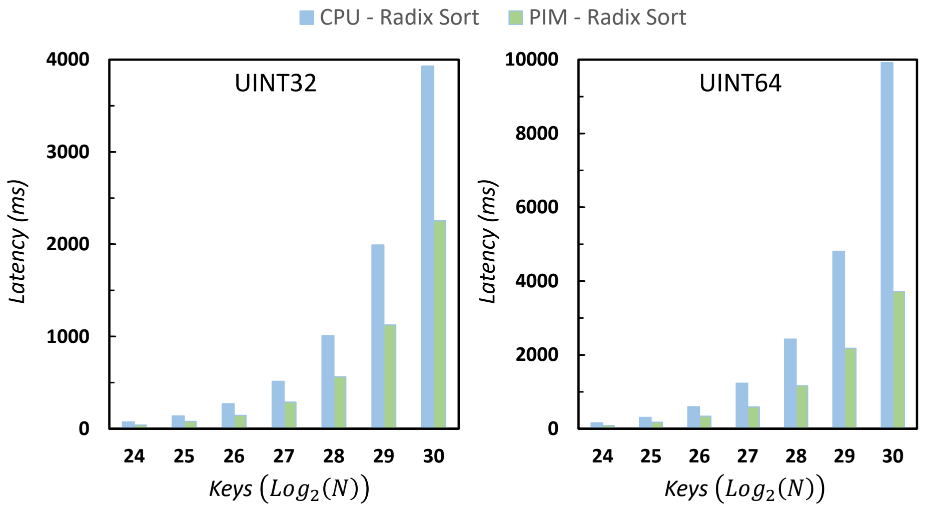


Figure 6.9: Latency measurements of sorting a sequence of 32 or 64-bit keys by using CPU only or our PIM-assisted implementation.

6.4.4 Heap-Based Top-K Performance on PIM

In Figure 6.10, we present the throughput measurements obtained for the heap-based Top-K implementation on a single DPU. We observe that the processing throughput approaches that of the ideal case when dealing with 64-bit keys when considering increasing input size. This happens because maintaining a max heap is mainly a memory intensive operation requiring only a few arithmetic operations. In that case, the throughput is better on 64-bits because we are able to maintain larger buffers which are used to load the associated data This enables us to process more information in about the same amount of time. Hence, the higher throughput which is almost 2× better than the 32-bit case.

For increasing result size, we observed a noticeable drop in the throughput when k is larger than 64. This happens because the merging phase executes on a single tasklet. In that case, we do not have enough threads executing in the DPU pipeline to mask the

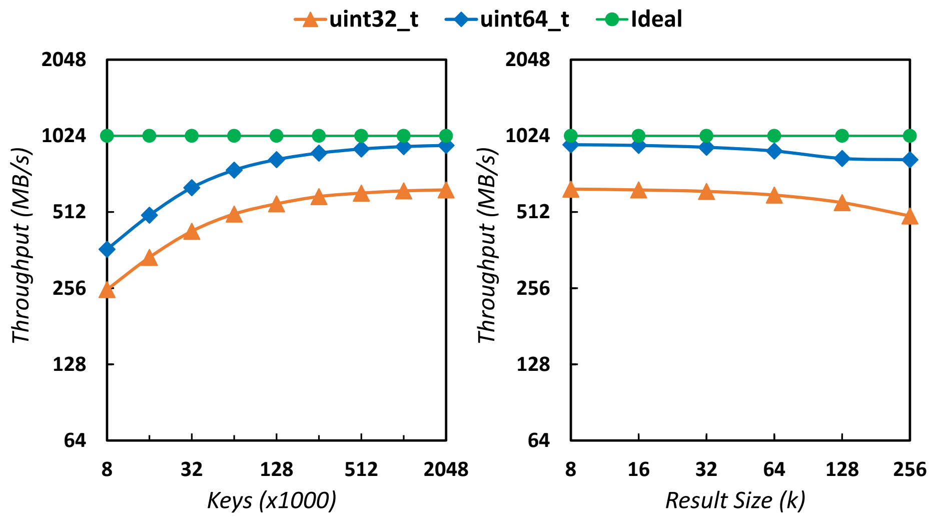


Figure 6.10: Single DPU throughput for our heap-based Top-K implementation.

latency of accessing and re-balancing multiple heaps until the k highest values are discovered. Despite this drop, the attainable throughput remains reasonably high.

In Figure 6.11, we present latency measurements for increasing number of keys and result size using our PIM optimized sorting-based and heap-based Top-K implementations. We observed that regardless of the experimental configuration, heap-based Top-K query evaluation is significantly faster than its sorting-based equivalent. This happens because using sorting requires re-arranging the input keys many times across different iterations. This operation is very expensive and usually limited by the available CPU to DRAM bandwidth. Our implementation of radix-sort utilizes the PIM accelerator to only improve radix-count. Hence, there is limited space for improving processing since the host CPU is responsible for re-arranging the corresponding keys. Our heap-based Top-K query implementation does not requires re-arranging the input keys across multiple DPUs. It operates by creating multiple heaps per DPU and merging them into a single one. This

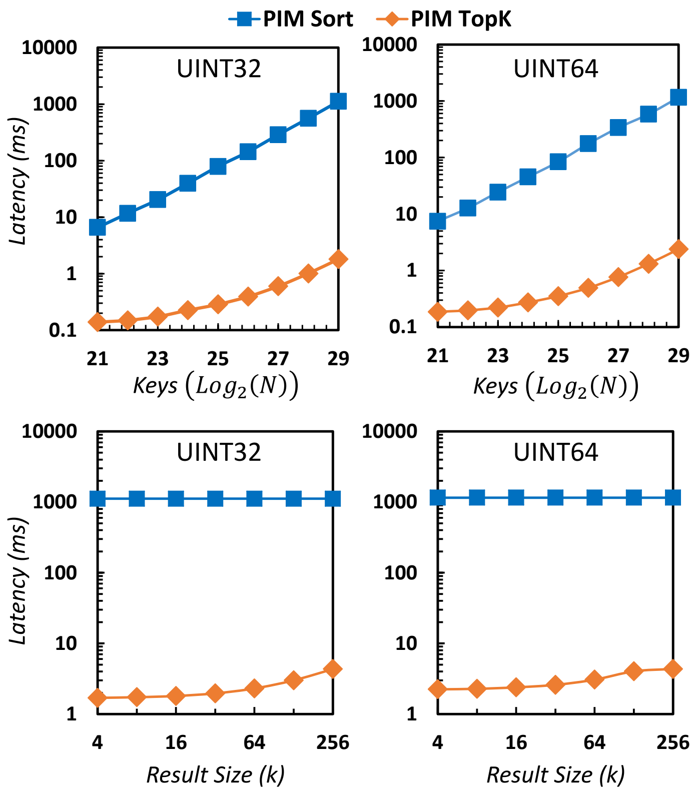


Figure 6.11: Single DPU throughput for our heap-based Top-K implementation.

merging operation executes on the host CPU and is cheap compared to re-arranging the corresponding keys across different DPUs. Therefore, the major part of the heap-based Top-K computation is spend in building multiple distinct heaps across different DPUs. This operation is embarrassingly parallel enabling us to take advantage of the full PIM processing bandwidth. In addition to that characteristic, our implementation is structured in such a way that every DPU is able to process the required information entirely into WRAM requiring only minimal write-back operations to MRAM. This reduces significantly the overall query latency since writing to MRAM has been established to be very expensive. The aforementioned performance behavior can be observed for both increasing input and result size.

Considering our analysis the heap-based Top-K query evaluation works better than completely sorting the input vector when using PIM architectures. This is evident from the fact that our heap-based Top-K implementations achieve reasonable throughput per DPU and several orders of magnitude better query latency. Based on this finding, we establish that heap-based Top-K query evaluation is the best strategy amongst every possible PIM optimized FTE solution. For this reason and henceforth, we will be using our heap-based Top-K query implementation (referred to as Top-K PIM) as the baseline solution for enabling efficient FTE Top-K query evaluation on UPMEM's PIM architecture.

6.4.5 FTE Top-K Performance on Parallel Architectures

In this section, we focus on comparing the performance of PIM Top-K against previously proposed Top-K FTE solutions that have been optimized for CPU and GPU processing. We utilize the SIMD-enabled multi-threaded implementation presented in Section 3.5 and the bitonic Top-K algorithm described in Section 4.2.2.

We conducted experiments on uniform data using 32 and 64-bit keys. In Figure 6.12, we summarize the results of our experiments indicating the measured query latency for each one of the available parallel architectures. For increasing input size, PIM Top-K performs on average 30× to 35× than the equivalent CPU implementation on the 32-bit key workload. As observed from our experiments, the performance gap between PIM and CPU for the 64-bit key workload increases significantly by almost 50% This behavior is expected and can be attributed to the fact that UPMEM's PIM architecture is designed to efficiently retrieve and operate on 64-bit keys. In addition, our implementation makes use of WRAM

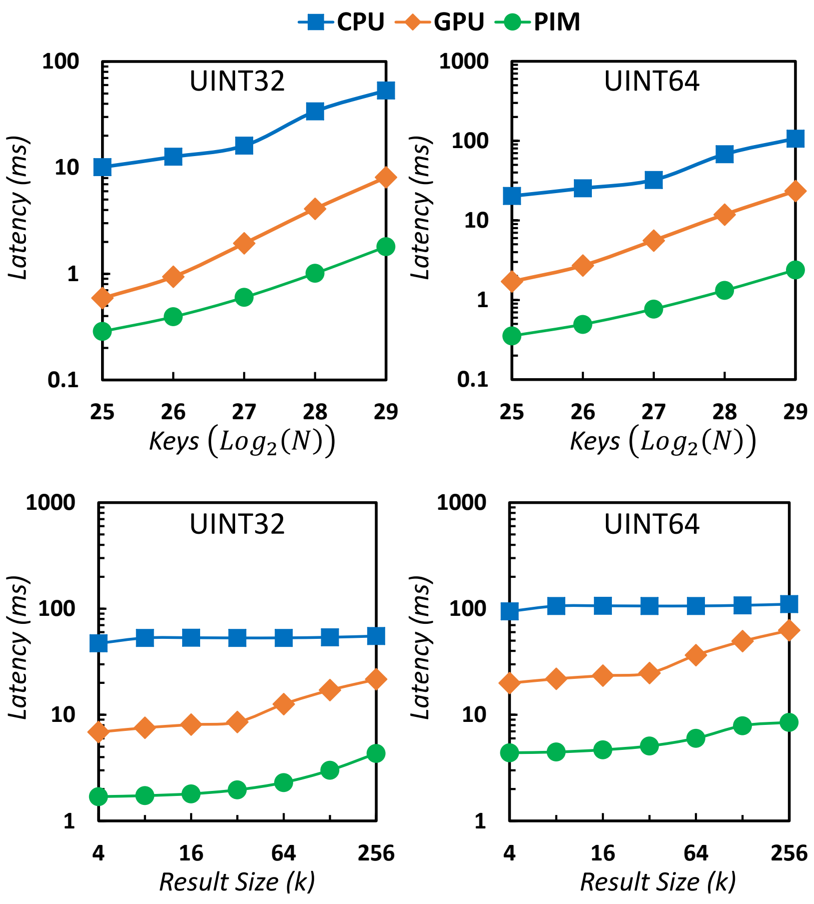


Figure 6.12: Query latency measurements for 32 and 64-bit key workloads using optimized Top-K implementations for CPU, GPU and PIM architectures.

when re-arranging the keys corresponding to the priority queue of every available DPU. Compared to the GPU implementation and for increasing input size, Top-K PIM attains on average 3× and 7× better query latency. GPUs are designed to operate efficiently on 32-bit data thus explained the noticeable performance drop when processing the 32-bit keys. As noted above, PIM will process 64-bit keys incurring an equivalent processing overhead to that of processing 32-bit keys. Hence, the performance gap doubles when focusing on 64-bit keys.

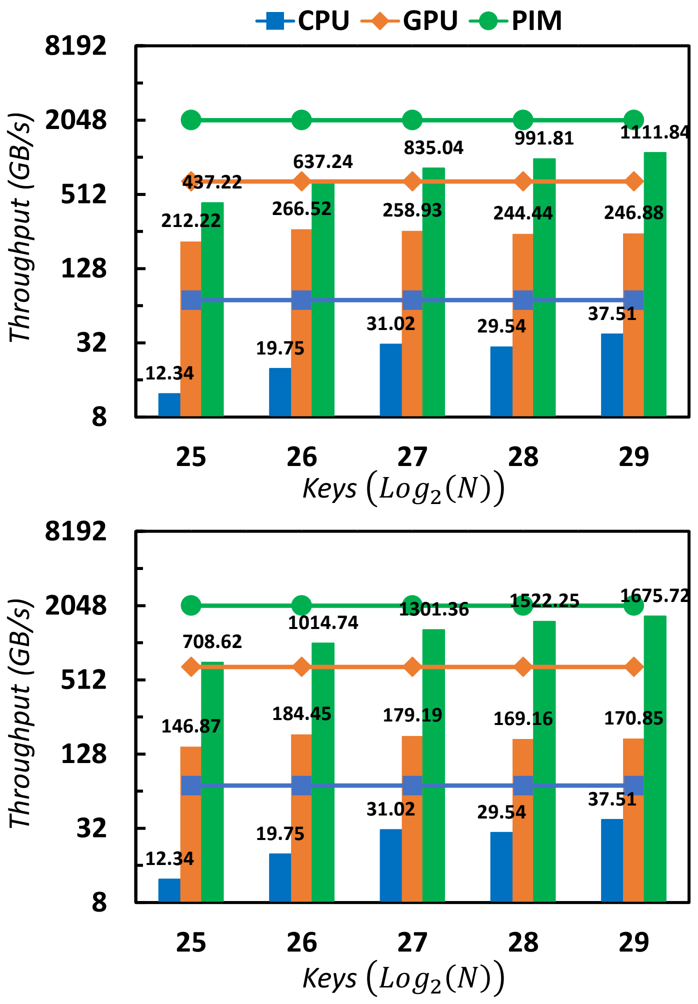


Figure 6.13: Processing throughput for 32 and 64-bit key workloads using optimized Top-K implementations for CPU, GPU and PIM architectures.

For increasing result size, all architectures suffer from increase pressure to the on-chip resources (i.e. cache, shared-memory, registers). Multi-core CPUs consist of large hierarchies which allow them to easily retain a lot of data during processing. This is a useful property which comes into play when processing Top-K queries which aim at retrieving a lot of results. GPUs and PIM architectures leverage on massive parallelism to hide data access latency. This processing model forces both architectures to invest more space on chip placing processing cores as opposed to caches and registers. Hence, they are unable to retain a lot of data during processing. This constrains affect Top-K processing when the expected result size is large. In our experimental measurements, we can clearly observe that for the CPU implementation the latency remains relatively constant for both workloads despite the increasing result size. On the other hand, for the GPU and PIM measurements we observe a noticeable increase in query latency when the result size is larger than 64 regardless of the given workload. In fact, the increase in query latency is much more striking for the GPU measurements when considering the 64-bit workload. This happens because the GPU shared memory allocated per streaming multiprocessor is limited to 48KB while the equivalent on-chip memory (WRAM) for a single DPU is 64KB. Therefore, our PIM implementation is able to maintaining more data during processing thus sustaining a better overall performance.

In Figure 6.13, we present the results of our experiments focusing on the attained processing throughput for all tested parallel architectures. The associated figure contains the maximum attainable throughput (indicated using the same color line) corresponding to the CPU, GPU and PIM based on their publicly available specification sheets which are also summarized in Table 6.1. PIM attains on average $30\times$ and $3\times$ better throughput than the equivalent CPU and GPU implementation, respectively. Similar to the latency measurements, UPMEM's PIM attains its peak performance when processing the 64-bit key workload. Concentrating on the theoretical maximum, we observe that our Top-K PIM implementation achieves between 30% to 80% utilization for increasing input size on the 64-bit workload. In comparison, the CPU maximum real utilization is at most 50% of

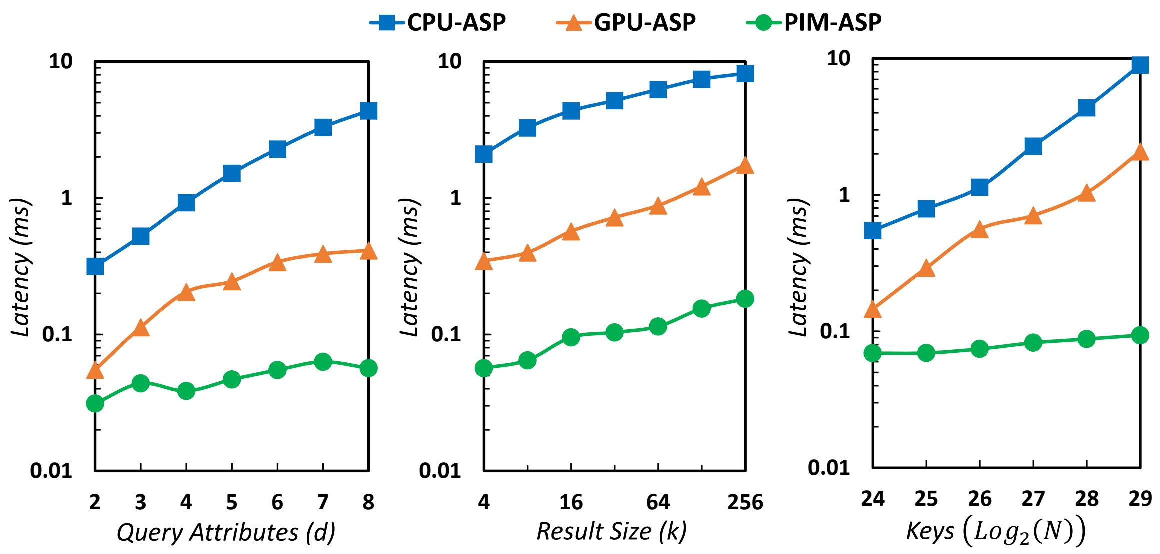


Figure 6.14: Measured query latency for experiments with increasing number of attributes, result size, and input size while using early termination Top-K query evaluation across different parallel architectures.

what is theoretically possible while the GPU attains even lower utilization the maximum of which is around 26% of the theoretical. Overall, we conclude that FTE Top-K on PIM performs better than any other equivalent solution on either multi-core CPUs or many-core GPUs.

6.4.6 Early Termination Performance Evaluation

In Figure 6.14, we present the experimental results measuring query latency for early termination algorithms developed on multi-threaded CPU, many-core GPU and PIM environments. Overall, we observed that early termination on PIM attains more than order of magnitude better query latency compared to the equivalent solution on CPU. The GPU implementation is almost an order of magnitude worse than the equivalent PIM solution.

The observed performance gap widens significantly when processing queries that access high number of attributes (i.e. 4 to 8 attributes). The difference in the maximum available read bandwidth for every tested parallel architecture (i.e. CPU, GPU, PIM) contributes to this widening gap. This happens regardless of the fact that every solution performs about the same amount of work during query evaluation.

For increasing result size, early termination on PIM consistently outperforms the equivalent CPU and GPU solutions. We observed that query latency increases significantly for large k values when using PIM or GPU processing because more pressure is put on the limited on-chip resources. Although the CPU implementation has access to comparatively larger on-chip memory, its attained query latency for large k values increases because of the additional cost that is related to aggregating the corresponding tuple scores. Finally, we observe that for increasing input size PIM outperforms again both the CPU and GPU methods. In fact, the measured query latency for PIM is somewhat as the number of input tuples increases. In comparison, the attained query latency of the CPU and GPU implementation increase linearly to the input number. This happens because PIM operates in-memory where data access latency is comparatively lower than accessing the data from DRAM (for the CPU) or GDDR (for the GPU). Hence, the overall processing cost is dominated by data movement which becomes apparent through our experimental measurements.

In Figure 6.15, we summarize the results of our experiments measuring query latency for varying query configuration while using the heap-based Top-K query evaluation (denoted as PIM-FTE) and ASP-based early termination Top-K query evaluation (denoted as PIM-ASP). The early termination solution consistently outperforms full table evaluation

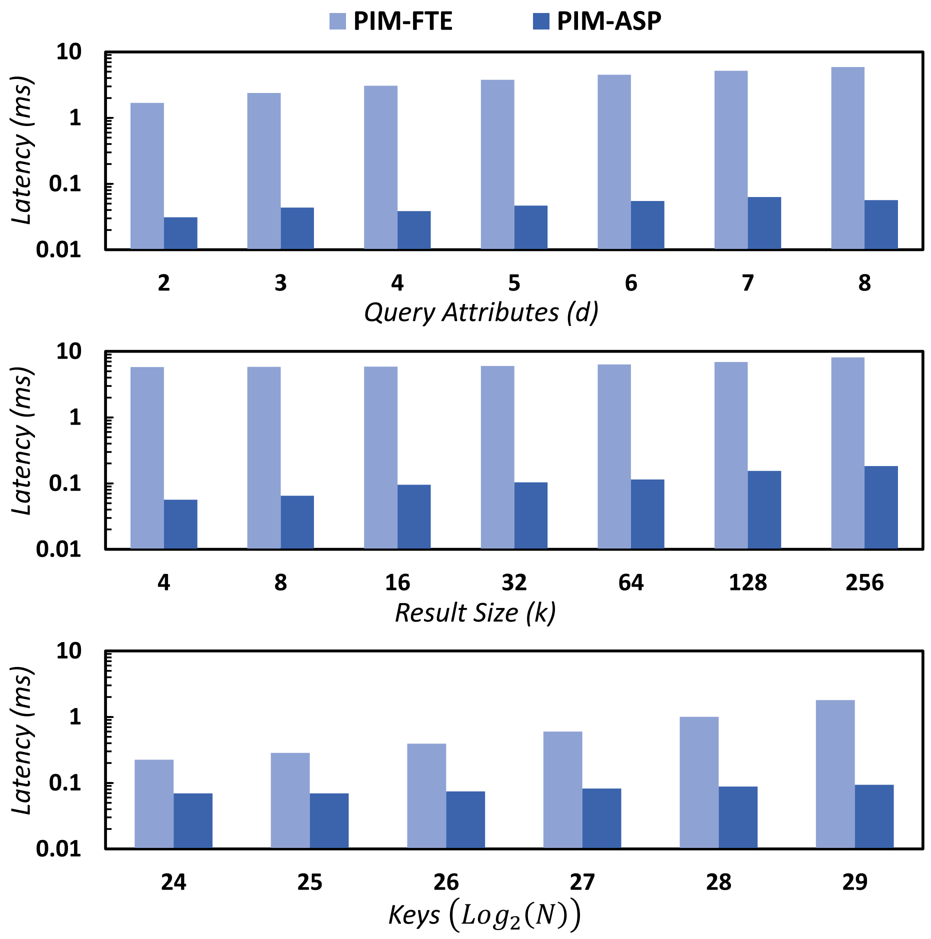


Figure 6.15: Measured query latency of FTE Top-K and early termination Top-K using PIM for experiments with increasing number of attributes, result size, and input size.

by several orders of magnitude. The performance gap between these solutions is apparent for increasing attribute number, result size, and input size. In fact, for high number of query attributes it is significant because FTE solutions requires scanning the whole dataset. In contrast, early termination requires processing about 10% of the total number of tuples, a characteristic that is consistent across different implementation platforms (i.e. CPUs or GPUs).

6.5 Conclusion

In this chapter, we developed efficient Top-K algorithms for Processing-In-Memory (PIM) architectures. We focused on optimizing query evaluation by studying strategies which leverage on either Full Table Evaluation (FTE) or early termination combined with Angle Space Partitioning (ASP). Our analysis identified that FTE methods fall under two categories; (1) sort-based, (2) heap-based. Considering the constrains imposed by UPMEM's PIM solution, we discerned that FTE heap-based solutions outperform sort-based alternatives because they are able to utilize the full bandwidth and parallelism of the corresponding PIM accelerator. Such distinction lead us towards the establishment of a fair baseline for comparing PIM performance against FTE methods that were optimized for multi-core CPU and many-core GPU processing. Our experiments showcased that PIM performs attains lower query latency, higher processing throughput and higher utilization than any of the aforementioned architectures on FTE Top-K query processing. We utilized the main principles of heap-based processing to implement efficient early termination Top-K query processing for UPMEM's PIM. Our experiments revealed that early termination combined with ASP attains lower query latency compared to FTE Top-K query evaluation. In addition, early termination on PIM attained several orders of magnitude better query latency compared to similar solutions optimized for CPUs and GPUs.

Chapter 7

Accelerated Skyline Selection on Processing-In-Memory

7.1 Introduction

Modern processors leverage the integration of many compute cores and deep cache hierarchies on a single chip to mitigate the effects of processing large dataset. This trend necessitates the redesign of popular skyline algorithms to take advantage of the additional hardware capabilities. Recent work on skyline computation relies on modern parallel platforms such as multi-core CPUs [21] and many-core GPUs [15]. These solutions attempt to address the unprecedented challenges associated with maintaining algorithmic efficiency while maximizing throughput. Despite these efforts, the widening gap between memory and processor speed contributes to a high execution time, as the maximum attainable throughput is constrained by the data movement overhead that is exacerbated by the low

	CPU	GPU	PIM
Cores (c)	10	3584	2048
Bandwidth (GB/s)	68	480	2048
Power (W/c)	10.5	0.17	0.04

Table 7.1: Single node specification comparison for CPU (Xeon E5-2650), GPU (TITAN X) and PIM (UPMEM) architectures.

computation to data movement ratio evident in the core (i.e. dominance test, Section 2.2.1) skyline computation. In addition, the skyline operator exhibits limited spatial and temporal locality because each point in the candidate set is accessed with varying frequency since it might dominate only few other points. As a result cache hierarchies will not be beneficial when processing large amounts of data.

Processing-In-Memory (PIM) architectures [3, 31, 37, 40, 58, 71, 83, 88, 93, 103] present a viable alternative for addressing this bottleneck leveraging on many processing cores that are embedded into DRAM. Moving processing closer to where data reside offers many advantages including but not limited to higher processing throughput, lower power consumption and increased scalability for well designed parallel algorithms (Table 7.1). For our work we rely again on UPMEM's architecture [58] , a commercially available PIM implementation that incorporates several of the aforementioned characteristics. Our skyline implementation presents a practical use case, that captures the important challenges associated with designing complex data processing algorithms using the PIM programming model.

Computing the skyline using a PIM co-processor comes with its own set of non-trivial challenges, related to both architectural and algorithmic limitations. Our goal is to identify and overcome these challenges through the design of a massively parallel skyline algorithm, that is optimized for PIM systems and adheres to the computational efficiency and throughput constraints established on competing architectures. Our contributions are summarized below:

- We propose a nontrivial assignment strategy suitable for balancing the expected skyline workload amongst all available PIM processors (Section 7.2.3).
- We present the first massively parallel skyline algorithm (i.e. *DSky*), optimized for established PIM architectures (Section 7.2.4).
- We provide a detailed complexity analysis, proving that our algorithm performs approximately the same amount of parallel work, as in the sequential case (Section 7.2.4).
- We successfully incorporate important optimizations, that help maintain algorithmic efficiency without reducing the maximum attainable throughput (Section 7.2.5).
- Our experimental evaluation demonstrates $2-14\times$ higher throughput (Section 7.4.3), good scalability (Section 7.4.4), and an order of magnitude better energy consumption (Section 7.4.5) compared to CPUs and GPUs.

7.2 Parallel Skyline Computation on PIM

Although PIM architectures resemble a distributed system, they are far from being one since they do not allow for direct communication between DPUs (i.e. slave-nodes). For this reason, algorithms relying on the MapReduce framework [78] are not directly applicable since they will involve excessive bookkeeping to coordinate execution and necessary data exchange for each DPU. Additionally, the MapReduce framework involves only a few stages of computation (i.e. chained map-reduce transformations) which may not be enough to effectively mask communication latency when the intermediate results between local skyline computations are prohibitively large. Despite these limitations, we can still rely on Bulk Synchronous Processing (BSP) to design our algorithm, giving greater emphasis on good partitioning strategies that provide opportunities to mask communication latency and achieve load balancing. The most prominent solutions in that field include the work of Vlachou et al. [96] and Kø̈hler et al. [55]. Both advocate towards partitioning the dataset using each points' hyperspherical coordinates. Although, this methodology is promising, it does not perform well on high dimensional data (i.e. $d > 8$), because it creates large local skylines, resulting in a single expensive merging phase [72]. Additionally, calculating each points' hyperspherical coordinates is a computationally expensive step [55]. For this reasons, we purposefully avoid using the aforementioned partitioning schemes. Instead, we present a simpler partitioning scheme which emphasizes load balancing and masking communication latency during the merging of all intermediate results.

Following, we describe the details of our approach which we call *DPU Skyline* (DSky), and present a thorough experimental analysis. The algorithm operates in two stages, the *preprocessing* stage where points are grouped into blocks/partitions and assigned to different *DPUs*, and a *main processing* stage spanning across multiple iterations within which individual blocks are compared in parallel against other previously processed blocks.

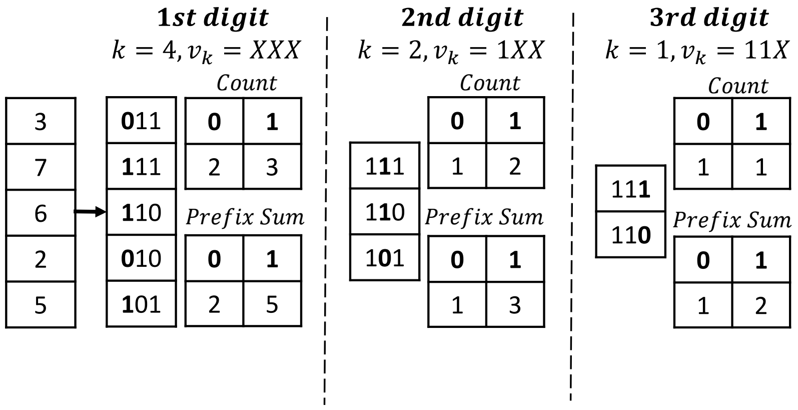


Figure 7.1: Radix-select example using radix-1.

7.2.1 Parallel Radix-Select & Block Creation

Maintaining the efficiency of sequential skyline algorithms, requires processing points in-order based on a user-defined monotone function. Due to architectural constraints, sorting the input to establish that order, contributes to a significant increase in the communication cost between host and *DPUs*. Our algorithm relies on parallel radix-select [5] to find a set of pivots which can be used to split the dataset into a collection of blocks/partitions. Radix-select operates on the ranks/scores that are generated for each point from a user defined monotone function. In our implementation, we assume the use of L_1 norm. Computing the rank of each point is relatively inexpensive, highly parallel and can be achieved by splitting the data points evenly across all available *DPUs*.

Radix-select closely resembles radix-sort, in that it requires grouping keys by their individual digits which share the same significant position and value. However, it differs as its ultimate goal is to discover the k-th largest element and not sort the data. This can be accomplished by building a histogram of the digit occurrences, for each significant position across multiple iterations, and iteratively construct the digits for the k-th largest element. An example for $k = 4$ is shown in Fig. 7.1. The digits are examined in groups of 1 (i.e. radix-1) starting from the most significant digit (MSD). At the first iteration, there are 2 and 3 occurrences of 0 and 1, respectively. The prefix sum of these values indicates that the 4-th element starts with 1. We update k by subtracting the number of elements at the lower bins. This process repeats at the next iteration for elements that match to $1XX$. After 3 iterations the k-th largest value will be $v_k = 110$.

The pseudocode for the *DPU* kernel corresponding to radix-select is shown in Algorithm 18. In our implementation, we use radix-4 (i.e. examine 4 digits at a time) which requires 16 bins per thread. For 32-bit[1] values, we require 8 iterations that consist of two phases. First, each *DPU* thread counts the digit occurrences for a given portion of the data. At a given synchronization point the threads cooperate to accumulate partial results into a single data instance. In the second phase, the host will gather all intermediate results and calculate the corresponding digit of the k-th value while also updating k. The new information is then made available to all *DPUs* at the next iteration. This whole process is memory bound, although highly parallel and with a low communication cost (i.e. only few KB need to be exchanged), fitting nicely to the PIM paradigm. Therefore, it is suitable for discovering the splitting points between partitions.

Assuming a partition size, denoted with P_{size}, and N number of points, we require $P_{vt} = P - 1 = \frac{N}{P_{size}} - 1$ pivots to create partitions $\{C_0, C_1, C_2 ... C_{P-2}, C_{P-1}\}$. In Algorithm 19, we present the pseudocode for assigning points to their corresponding partitions.

[1]Floating-point types can be processed through a simple transformation to their IEEE-754 format.

Algorithm 18 Radix-select Kernel

R = *Precomputed Rank vector.*

K = *Splitting Position.*

V_k = *Digits of Current Pivot.*

```
1: for $digit \in [7, 0]$ do
2:     $Set\ B_t = \{0\}$                          ▷ Set thread bins to zero.
3:     for all $r \in R$ in parallel do
4:         if $prefix(r, V_k)$ then                ▷ Match prefix.
5:             $B_t[digit] + +$
6:         end if
7:     end for
8:     $B = sum(Bt)$                               ▷ Aggregate Partial Counts.
9:     $(V_k, K) = search(B, K)$                   ▷ Update P & K.
10: end for
```

As indicated in Line 3, we concentrate on the rank of a given point to identify the range of pivots that contain it, after which we assign it to the partition with the corresponding index. The presented partitioning method guarantees that no two points p, q exist, such that $p \in C_i$ and $q \in C_j$, where $i < j$ and $F(p) > F(q)$. Points within a partition do not have to be ordered with respect to their rank, given a small partition which allows for parallel brute force point-to-point comparison.

Blocked processing has been used before for CPU based skyline computation [21] to improve cache locality. Our solution differs, since it supports blocking while avoiding the high cost of completely sorting the input data. Furthermore, we utilize blocking to introduce a nontrivial work assignment strategy which enables us to design a highly parallel and throughput optimized skyline algorithm for PIM architectures. This strategy aims at maximizing parallel work through maintaining good load balance across all participating *DPUs*, as compared to the optimal case.

Algorithm 19 Radix-select Partitioning

```
$D$ = Input dataset
$R_p$ = Pivots vector

1: $R_p$ = radix_select(D)                          ▷ Calculate pivots.
2: for all $p \in D$ do
3:     if $R_p[j] < F(p) \leq R_p[j+1]$ then
4:         $C_j = C_j \cup \{p\}$                   ▷ Assign $p$ to $C_j$.
5:     end if
6: end for
```

7.2.2 Horizontal Partition Assignment

In this section, we concentrate on introducing a simple horizontal assignment strategy, the performance of which motivates our efforts to suggest a better solution. Our goal is to establish the lower bound associated with the parallel work for computing the skyline, measured in partition-to-partition (*p2p*) comparisons, and suggest a strategy along with the algorithm that is able to attain it.

We start by introducing some definitions. Given a partition C_j, we define its pruned equivalent partition, the set of all points that appear in C_j which will be eventually identified as being part of the final skyline set. We denote this pruned partition as $\widetilde{C}_j \subseteq C_j$. Assuming a collection of P partitions, which can be ordered using radix-select partitioning, such that for $i, j \in [0, P-1]$ and $i < j$, then $C_i \prec C_j$ (i.e. C_i precedes C_j), it is possible to compute P pruned partitions iteratively:

a. $\widetilde{C}_0 = p2p(C_0, C_0)$

b. $\widetilde{C}_1 = p2p(\widetilde{C}_0, p2p(C_1, C_1))$

c. $\widetilde{C}_2 = p2p(\widetilde{C}_0, p2p(\widetilde{C}_1, p2p(C_2, C_2)))$

The *p2p* function denotes a single partition-to-partition comparison operation, checking if any points exist in C_i that dominate those in C_j. More details related to the implementation of *p2p*, are presented in Section 7.2.5. We observe that using the pruned partition definition, we can calculate the skyline set using the following formula:

$$S = \bigcup_{i \in [0, P-1]} \left(\widetilde{C}_i \right) \tag{7.1}$$

Eq. 7.1, indicates that it is possible to compute the skyline using the union of all pruned partitions. Therefore, it is possible to maintain and share information about the skyline *without* using a centralized data structure. Additionally, once $\widetilde{C}_j$ is generated, all remaining partitions with index larger than j may use it to prune points from their own collection. In fact, performing this work is "embarrassingly" parallel and depending on the partition size and the input dataset size, it can be scaled to utilize thousands of processing cores. However, we observe that assigning work to DPUs naïvely could potentially hurt performance, due to the apparent dependencies between partitions and the fact that latter partitions require more *p2p* comparisons to be pruned.

Assuming all partitions are processed in sequence, we can calculate the number of total *p2p* comparisons by examining each partition separately. For example, C_0 will need 1 self-comparison (i.e. $p2p(C_0, C_0)$), C_1 will need 2 *p2p* comparisons, C_2 3 and so on. In fact, the total number of *p2p* comparisons, assuming P partitions is given by the following equation:

$$M_{seq} = \frac{P \cdot (P+1)}{2} \tag{7.2}$$

Ideally, with D_p *DPUs* at our disposal, we would like to evenly distribute the workload among them, maintaining a *p2p* comparison count which is roughly equal to $\frac{M_{seq}}{D_p}$. A fairly common and easily implementable strategy, is to divide the partitions ($P_D = \frac{P}{D_p}$ per *DPU*) horizontally across *DPUs* as indicated in Figure 7.2. However, if we attempt to follow this strategy, the *DPU* responsible for the last collection of partitions will have to perform at least $(P - P_D) \cdot P_D + \frac{P_D \cdot (P_D+1)}{2}$ *p2p* comparisons a number $P \cdot P_D$ times higher than the DPU responsible for the first collection of partitions. Obviously, this assignment

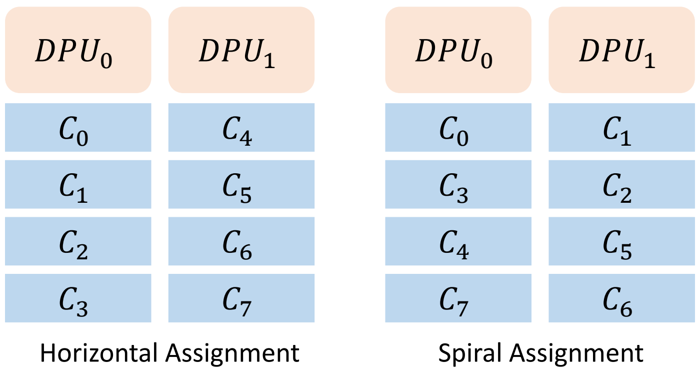


Figure 7.2: Assignment strategies of 8 partitions on 2 *DPUs*.

mechanism suffers from several issues, the most important of which is poor load balancing. In fact during processing, the majority of the participating *DPUs* will be idle waiting for pruned partitions to be calculated and transmitted. Additionally, the limited memory space available to each DPU, makes it hard to amortize the cost of communication, since processing needs to complete before exchanging any data. To overcome the problems set forth by horizontal partitioning, we introduce the concept of *spiral partition* assignment.

7.2.3 Spiral Partition Assignment

Commonly, data intensive algorithms rely on Bulk Synchronous Processing (BSP) to iteratively apply transformations on a given input across many iterations, between which a large portion of the execution time is dedicated to processing rather than communication. This process aims to maintain good load balance and reducing communication to effectively minimize each processor's idle time. In this section, we introduce a nontrivial assignment strategy which allows for the design of an iterative algorithm that follows the aforementioned properties.

Our assignment strategy relies on the observation that for a collection of $2 \cdot D_p$ ordered partitions with respect to a user-provided monotone function, we can always group them together creating non-overlapping pairs, all of which when processed individually, require the same *p2p* comparison count. The pairing process considers partitions in opposite locations with respect to the monotone function ordering, resulting in the creation of D_p pairs in total. For example, assuming the existence of partitions $\left\{C_0, C_1 ..., C_{2 \cdot D_p - 1}\right\}$, we will end up with the following pairs:

$$\left\{\left\langle C_0, C_{2 \cdot D_p - 1}\right\rangle, \left\langle C_1, C_{2 \cdot D_p - 2}\right\rangle ..., \left\langle C_{D_p - 1}, C_{D_p}\right\rangle\right\} \tag{7.3}$$

In Figure 7.2, we showcase our novel assignment strategy, which we call *spiral partitioning*, next to the naïve horizontal partitioning scheme. In contrast to the horizontal partitioning mechanism which requires $4 \cdot 4 + \frac{4 \cdot 5}{2} = 26$ *p2p* comparisons from a single DPU, our spiral partitioning scheme requires only 18 (i.e., $(1+4+5+8) = DPU_0$, $(2+3+6+7) = DPU_1$) most of which can be performed in parallel. This number is equivalent to $\frac{M_{seq}}{D_p} = \frac{36}{2}$, indicating that our spiral partitioning strategy splits evenly the expected workload across all participating $DPUs$.

In our analysis, we assumed the number of partitions P to be equal to $2 \cdot D_p$. In the general case, we can choose P and D_p, in order for P to be expressed as multiple of $2 \cdot D_p$ such that $K = \frac{P}{2 \cdot D_p}$. For each one of the K collections, we can individually apply the spiral partitioning algorithm and assign one pair from each collection to a distinct DPU. Following this assignment process, we calculate the total p2p comparison count per DPU based on the following formula:

$$\begin{aligned} M_{opt} = (1 + 2 \cdot D_p) + (1 + 6 \cdot D_p) + (1 + 10 \cdot D_p) + ... = \\ D_p \cdot (2 + 6 + 10 + 14...) + \frac{P_D}{2} = \\ D_p \cdot \frac{(4 + 4 \cdot (\frac{P_D}{2} - 1))}{2} \cdot \frac{P_D}{2} + \frac{P_D}{2} = \\ \frac{P_D}{2} \cdot \left[2 \cdot \frac{P_D}{2} \cdot D_p + 1\right] = \\ \frac{P}{2 \cdot D_p} [P + 1] \Rightarrow M_{opt} = \frac{P \cdot (P + 1)}{2 \cdot D_p} \end{aligned} \tag{7.4}$$

The aforementioned formula is based on the observation that for each collection, the number of *p2p* comparisons per *DPU* is equal to the *p2p* comparisons required for the first and last partition of that collection. Therefore, for the first collection we need $1 + 2 \cdot D_p$ *p2p* comparisons, for the second $2 \cdot D_p + 1 + 4 \cdot D_p$, for the third $4 \cdot D_p + 1 + 6 \cdot D_p$ and so on, requiring increasing number of *p2p* comparisons.

In theory, it is possible to utilize at most $\frac{P}{2}$ *DPUs* for processing when using spiral partitioning. However, in practice, it might not be beneficial to reach this limit, since at that point the work performed within each *DPU* will not be enough to amortize the cost of communication or minimize the idle time. Additionally, due to the existing dependencies between partitions, increasing the number of *DPUs* will result in less work being performed in parallel. In the next section, we present more details regarding these issues and present the intrinsic characteristics of our main algorithm.

7.2.4 DSky Main Processing Stage

Leveraging on spiral partitioning, we introduce a new algorithm for computing the skyline set on PIM architectures. Once each partition has been assigned to their corre-

Partitions/Iteration	$i = 0$	$i = 1$	$i = 2$
$\{C_0, C_3\}$	$\tilde{C}_0: (C_3)$	$\tilde{C}_1: (C_3)$	$\tilde{C}_2: (C_3)$
$\{C_1, C_2\}$	$\tilde{C}_0: (C_1, C_2)$	$\tilde{C}_1: (C_2)$	–

(A)

Partitions/Iteration	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
$\{C_0, C_3, C_4, C_7\}$	$\tilde{C}_0: (C_3, C_4, C_7)$	$\tilde{C}_1: (C_3, C_4, C_7)$	$\tilde{C}_2: (C_3, C_4, C_7)$	$\tilde{C}_3: (C_4, C_7)$	$\tilde{C}_4: (C_7)$	$\tilde{C}_5: (C_7)$	$\tilde{C}_6: (C_7)$
$\{C_1, C_2, C_5, C_6\}$	$\tilde{C}_0: (C_1, C_2, C_5, C_6)$	$\tilde{C}_1: (C_2, C_5, C_6)$	$\tilde{C}_2: (C_5, C_6)$	$\tilde{C}_3: (C_5, C_6)$	$\tilde{C}_4: (C_5, C_6)$	$\tilde{C}_5: (C_6)$	–

(B)

Figure 7.3: Number of comparisons across iterations when assigning (A) 2 partitions per DPU vs (B) 4 partitions per DPU.

sponding DPU, we can start calculating each pruned partition within two distinct phases as indicated in Algorithm 20. In the first phase, each DPU performs a "self-comparison" for all partitions assigned to it. This step is "embarrassingly" parallel and does not require any data to be exchanged. The second phase consists of multiple iterations across which the pruned partitions are computed. At iteration i, the pruned partition $\widetilde{C}_i$ has already been computed and is ready to be transmitted across all $DPUs$. Once the broadcast is complete, all $DPUs$ have access to $\widetilde{C}_i$ which they use as a window to partially prune any of their own C_j partitions in parallel, where $j > i$ is based on the established ordering of partitions.

Our implementation uses a collection of flags, denoted with F_i for partition $\widetilde{C}_i$, to mark which points have been dominated during processing. We indicate with 0 those points that have been pruned away and with 1 those that are still tentatively skyline candidates. The whole process is orchestrated by the host (CPU), who keeps track of which partition needs to be transmitted at the end of each iteration. It is important to note that broadcasting individual partitions can be expensive. For this reason, we need to carefully choose the partition size in order to overlap data exchange with actual processing. Additionally, we propose to further reduce this cost by preemptively broadcasting m partitions at each iteration before they are actually needed, thus increasing the computation-communication overlap window. Nevertheless, we still need to wait for the F_i bit-vector to become available before starting the next iteration. However, once the corresponding F_i bit-vector is calculated we can inexpensively transmit it to all *DPUs*, since it is inversely proportional to the point dimensions and partition size.

Assuming an optimal *p2p* kernel, we measure the complexity of DSky in terms of *p2p* comparisons per *DPU*. For the first phase, each *DPU* is responsible for self-comparing their assigned partitions, requiring P_D comparisons to complete. The second stage is slightly more complex. Within iteration i, the corresponding partition $\widetilde{C}_i$ will be compared against all C_j partitions having a higher index. Starting from $\widetilde{C}_0$ and for the next $D_p - 1$ iterations, each *DPU* will perform P_D comparisons. Once $\widetilde{C}_{D_p}$ is computed, only partitions with index larger than D_p will need to be considered, resulting in at most $P_D - 1$ comparisons for iterations D_p to $2 \cdot D_p - 1$. This process is repeated multiple times until all partitions within each *DPU* have been checked. Adding the complexity of each phase together, we end up with the following formula:

$$\begin{gathered} M_{par} = [(D_p - 1) \cdot P_D + D_p \cdot (P_D - 1) + \\ D_p \cdot (P_D - 2) \ \dots \ + D_p \cdot 1] + P_D \Rightarrow \\ M_{par} = D_p \cdot [\frac{P_D \cdot (P_D + 1)}{2}] \end{gathered} \tag{7.5}$$

From Eq. 7.5 and Eq. 7.4, if we replace $P_D = \frac{P}{D_p}$, we get the following ratio:

$$\frac{M_{par}}{M_{opt}} = \frac{1 + \frac{D_p}{P}}{1 + \frac{1}{P}} \tag{7.6}$$

Algorithm 20 DSky Algorithm

B_j = *Region bit vectors for* C_j.

F_j = *Flags indicating active skyline points for* C_j.

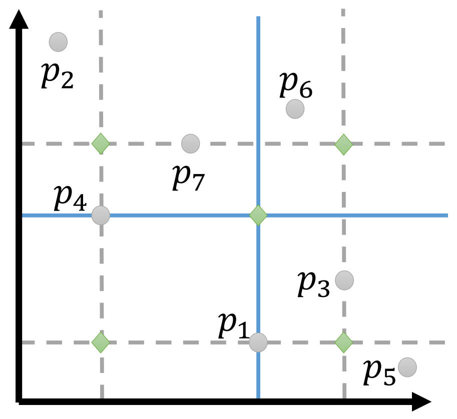
```
1: for all DPUs in parallel do
2:     for all $C_j \in DPU_i$ do
3:         $P2P(C_j, B_j, C_j, B_j)$                ▷ Self compare partitions.
4:     end for
5: end for
6: for all $i \in [0, P-1]$ do
7:     $copy(\widetilde{C}_i, \widetilde{B}_i, F_i)$      ▷ Broadcast pruned partition info.
8:     for all DPUs in parallel do
9:         for all $j > i$ do
10:            $P2P(\widetilde{C}_i, \widetilde{B}_i, C_j, B_j)$      ▷ Prune $C_j$ using $\widetilde{C}_i$
11:        end for
12:    end for
13: end for
```

From Eq. 7.6, we can observe how different values for P and D_p affect the complexity of *DSky* with respect to the optimal case. When $P \to \infty$, then $\frac{M_{par}}{M_{opt}} \to 1$. Intuitively, when the number of partitions assigned per *DPU* is significantly larger than its collective number, the observed idle time constitutes a smaller portion of the actual processing time. In Figure 7.3 using two *DPUs*, we present an example where 2 or 4 partitions are assigned per *DPU*. In the first case, we require 3 p2p comparisons and within iterations $i = 0, 2$, DPU_0 or DPU_1 will do 1 less comparison than the other, respectively. Therefore, $\frac{1}{4}$ of the time each *DPU* will be idle. In the second case, the total comparisons across iterations will be 14 and the corresponding idle time within iterations $i = 0, 2, 4, 6$ is 2. Hence, the idle time per *DPU* will be $\frac{2}{16}$, half of what was observed for the previous example. At this point, it is important to note that creating more partitions does not depend on the input size, but instead on the number of pivots calculated during radix-select partitioning. Although, this may seem like having a partition size equal to 1 is the best case, in practice there are several trade-offs to consider, such as the preprocessing time required to calculate each partition and the communication overhead when small data are transmitted frequently and not in bulk. Through experimentation, we are able to identify the specific parameters contributing to these trade-offs, allowing us to successfully fine tune the partition size.

7.2.5 P2P Kernel

In this section, we discuss three specific optimizations that can be integrated into our *p2p* kernel to ensure algorithmic efficiency.

Optimization I: The points within each partition are sorted based on their rank. This optimization can be embarrassingly parallel and less expensive than globally sorting all the points. It aims at reducing the expected number of DTs for each DPU by half [21].



	p_1	p_2	p_3	p_4	p_5	p_6	p_7
$Level_1$	10	01	10	01	10	11	01
$Level_2$	00	01	11	00	10	01	11

Figure 7.4: Median pivot multi-level partitioning example.

Optimization II: For more challenging distributions (i.e. anti-correlated), space partitioning is preferable since it can help with identifying incomparable point through cheap filter tests [21]. Similarly to previous work [15], we exploit a recursive space partitioning scheme to detect incomparability. This technique requires calculating bit-vectors for each point, indicating the region of space it resides. They are determined through a virtual pivot, constructed from the median value of its subspace.

An example of this is shown in Figure 7.4. There, we determine the values for the median level virtual pivot by taking the projection of p_1 in the x-axis and p_4 in the y-axis. Each point is assigned a bit vector based on its relative position to the virtual point. For example, p_1 is assigned 10 because it is $\geq$ and $<$ in the x and y-axis, respectively, compared to the pivot. For each quartile, we can repeat this process multiple times. However, it has been shown empirically [15] that doing it twice is sufficient to gain good algorithmic efficiency. We use radix-select to calculate the median value for each subspace and construct the corresponding pivots.

In related work [15], a centralized data structure is used to manage the bit vectors and establish a good order of processing. Due to architectural limitations (i.e. expensive global access), our implementation uses a flat array to pack both bit vectors in a single 32-bit value for each point. Our spiral partitioning scheme is responsible for maintaining the good order of processing. Additionally, it is designed around optimizing local access and minimizing communication while, also, promoting the seamless incorporation of the bit vector information within a partition.

Optimization III: Based on the work in [10], we use Eq. 2.3 to update the stopping level and point, and then compare this information with the point of the smallest rank within each partition to determine if it is dominated. The stopping information is updated locally within each DPU. The host is responsible for merging the local results at each step of *DSky's* second stage (Algorithm 20). This requires few KBs to be exchanged, thus its communication overhead is low.

Algorithm 21 presents the implementation of our *p2p* kernel. Each DPU allocates memory for P_D partitions, plus two remote partitions to support double buffering. In Line 1, we compare the smallest rank within the given partition to the global stopping value to determine if the whole partition is dominated. When this test fails, we need to check all the points within the partition. For each point in the local partition, we only examine the points that are still skyline candidates (Line 5) against those of the remote partition that satisfy the same property (Line 7). Using the corresponding bit vectors, if the two points are incomparable (Line 8) we skip to the next point in the remote partition, otherwise we need to perform a full DT (Line 11). For all points in the local partition that are not dominated

Algorithm 21 P2P Function Kernel

R_j = Rank vector for C_j, B_j = Region bit vectors for C_j.

F_j = Active skyline points for C_j, (g_s, p_s) = Global stop level and point.

```
1: if stop($g_s$,$p_s$,$R_j[0]$,$C_j[0]$) then
2:     return $F_j \leftarrow 0$                                        ▷ Prune partition.
3: end if
4: for all q $\in C_j$ in parallel do
5:     if $F_j[q] \neq 0$ then                                          ▷ $q$ is alive.
6:         for all p $\in C_i$ do
7:             if $F_i[p] \neq 0$ then                                  ▷ $p$ is alive.
8:                 If $B_i[p] \not\prec B_j[q]$ then continue           ▷ Incomparable.
9:                 If p $\prec$ q then $F_j[q] = 0$ break               ▷ Set flag for $q$ to zero.
10:            end if
11:        end for
12:    end if
13:    if $F_j[q] = 1$ then                                             ▷ Point is not dominated.
14:        $l_s[id]$ = MiniMax(q,$l_s[id]$)                             ▷ Thread stop level.
15:        $p_s[id]$ = q                                                ▷ Thread stop point.
16:    end if
17: end for
18: $(g_s, p_s)$ =update_ps($l_s[id]$ , $p_s[id]$)                      ▷ DPU stop info.
19: merge $F_j$
```

(Line 18), we update the local stop point information. At the end of the for-loop (Lines 23 − 24), we merge the local stop point information and update the local partition's flags to indicate which points have been dominated.

7.3 Experimental Environment

In this section, we present an in-depth analysis of *DSky*, comparing against the state-of-the-art sequential [60], multi-core [21] and many-core [15] algorithms.

7.3.1 Setup Configuration

CPU Configuration: For the CPU algorithms, we conducted experiments on an Intel Xeon E5-2650 2.3 GHz CPU with 64 GB memory. We used readily available C++ implementations of *BSkyTree* [60] and *Hybrid* [16].

GPU Configuration: For the GPU, we used the latest NVIDIA Titan X (Pascal) 1.53 GHz 12 GB main memory GPU with CUDA 8.0. We conducted experiments using the readily available C++ implementation of *SkyAlign* [16] which is the current state-of-the-art algorithm for GPUs. For a fair comparison, we present measurements using clock frequencies 0.75 and 1.53 GHz.

DPU Configuration: We implemented both phases of *DSky*, including the pre-processing steps, using UPMEM's C-based development framework [94] and dedicated compiler. Our experiments were performed on UPMEM's Cycle Accurate Simulator (CAS) using the binary files of the corresponding implementation. The simulation results were validated using an FPGA implementation [94] of the *DPU* pipeline. Based on the reported clock cycle count that includes pipeline stalls associated with the corresponding data accesses, and a base clock of 0.75 GHz for each *DPU*, we calculated the exact execution time for a single node system using 8 to 4096 *DPUs*. For a fair comparison against the GPU, we limit the number of *DPUs* in accordance to the available cuda cores (i.e. 3584).

7.3.2 Dataset

Similarly to previous work [15], we rely on the standard skyline dataset generator [17] to create common data distributions (i.e., correlated, independent, anticorrelated). We compare against the CPU and GPU implementations using queries with dimensionality $d \in \{4, 8, 16\}$ and for dataset of cardinality $n \in \left[2^{20}, 2^{26}\right]$ [2]. Additional experiments are presented on PIM only for cardinality $n \in \left[2^{20}, 2^{29}\right]$.

7.3.3 Experiments & Metrics

For all implementations, our measurements include the cost of preprocessing and data transfer (where it is applicable) across PCIE 3.0 (i.e. GPU) or broadcast between *DPUs*. We benchmarked the aforementioned algorithms with all of their optimizations enabled. For the performance evaluation, we concentrate on the following metrics:

Runtime Performance: This metric is used to evaluate at a high level the performance of *DSky* against previous solutions. It showcases the overall capabilities of the given architecture coupled with the chosen algorithm.

Algorithmic Efficiency & Throughput: Due to several hidden details within the runtime performance, we focus on the algorithmic efficiency by studying the number of full DTs conducted by each algorithm. Our ultimate goal is to showcase the ability of *DSky*

[2]Due to restrictions in GPU memory, the maximum dataset for comparison purposes was set to 2^{26}.

to successfully incorporate known skyline optimizations and indicate their contribution towards achieving high throughput on the UPMEM-PIM architecture.

Scaling: An important property of the UPMEM-PIM architecture is the ability to easily increase resources when the input grows beyond capacity. However, doing so requires a well designed parallel algorithm that avoids any unnecessary overheads caused by excessive communication or load imbalance. With this metric, we indicate *DSky's* ability to scale when resources increase proportionally to the input size.

In addition, our experiments on comparing the system utilization between GPU and PIM architectures, indicated an upward trend of 75% for PIM against 40% for GPUs. Moreover, we provide measurements demonstrating superior energy efficiency compared to state-of-the-art algorithms on CPUs and GPUs (Section 7.4.5).

7.4 Synthetic Data Experiments

7.4.1 Run-Time Performance

Correlated data contribute to a smaller skyline set which contains only a few dominator points. Therefore, during processing the main performance bottleneck is the memory bandwidth. Figure 7.5 illustrates the runtime performance for all algorithms on correlated data. *DSky* outperforms previous state-of-the-art algorithms for all tested query dimensions. This happens because it relies on radix-select, an inherently memory bound operation, to lower the preprocessing cost. Moreover, the main processing stage terminates early due to the discovery of a suitable stopping point. *BSkyTree* and *Hybrid* under-utilize

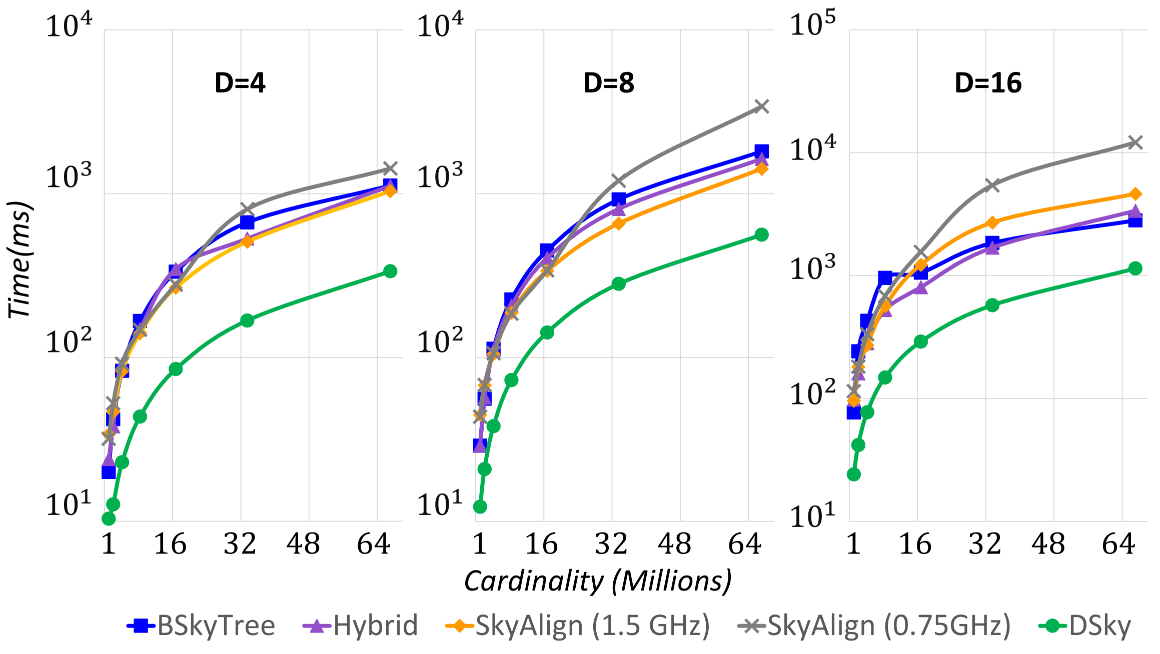


Figure 7.5: Execution time (log(t)) using correlated data.

the available bandwidth, since a single point requires only few comparisons to be pruned away. Therefore, prefetching data into cache will result in lower computation to communication ratio and higher execution time. *SkyAlign* is limited by the overhead associated with launching kernels on the GPU, which in this case is high relative to the cost of the processing and preprocessing stages.

Figure 7.6 presents the runtime performance for all methods using independent data. We observe that *DSky* outperforms previous implementations for query dimensions (i.e. $d = \{4, 8\}$) that reflect the needs of real-world applications. *Hybrid* and *BSkyTree* are restricted by the cache size, since increasing dimensionality contributes to a larger skyline. This results in a higher number of direct memory accesses leading to higher runtime. Compared to *DSky*, *SkyAlign* exhibits higher runtime on 4 and 8 dimension queries, due to

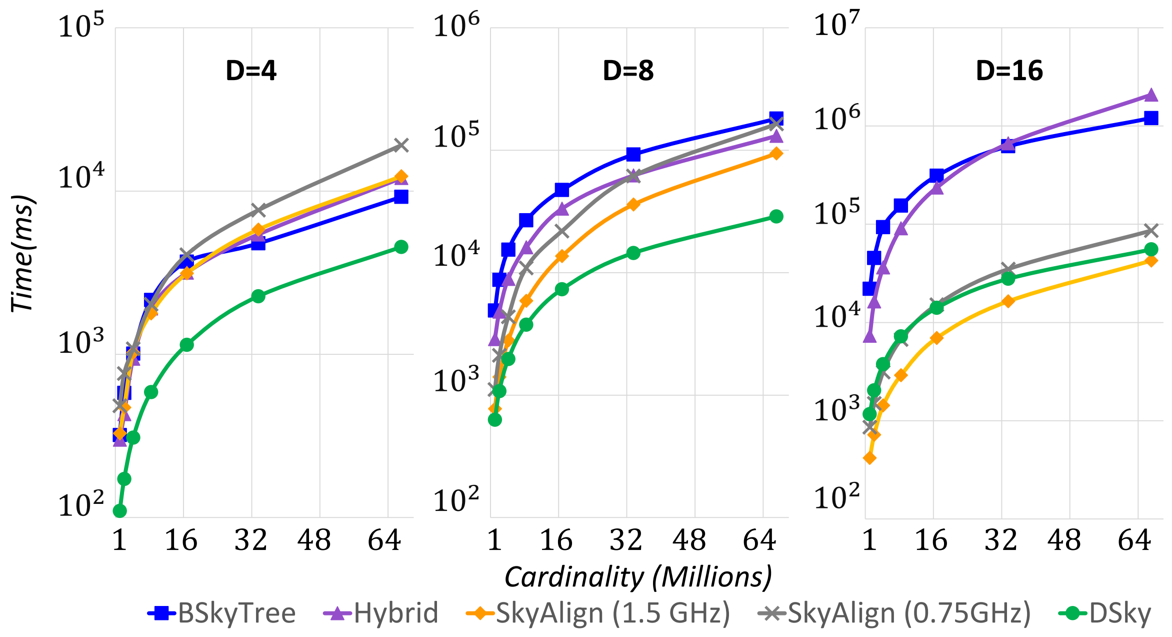


Figure 7.6: Execution time (log(t)) using independent data.

achieving lower throughput as a result of irregular memory accesses and thread divergence. On 16 dimensions, these limitations have a lesser effect on runtime, due to the increased workload which contributes towards masking memory access latency when more threads execute in parallel. However, concentrating on measurements using 750 MHz clock frequency, we observe that *DSky* outperforms *SkyAlign* approximately by a factor of 2. Intuitively, this indicates that *DSky* is throughput efficient compared to *SkyAlign*, as the latter fails to sustain same runtime for equal specification. In fact, experiments with higher frequency indicate a trend that predicts better performance for *DSky* on sufficiently large input (beyond 16 million points *SkyAlign* would crash, probably due to implementation restrictions and limited global memory).

Finally, Figure 7.7 illustrates the measured runtime for anticorrelated distributions. As before, *DSky* outperforms CPU-based methods which are restricted by the cache

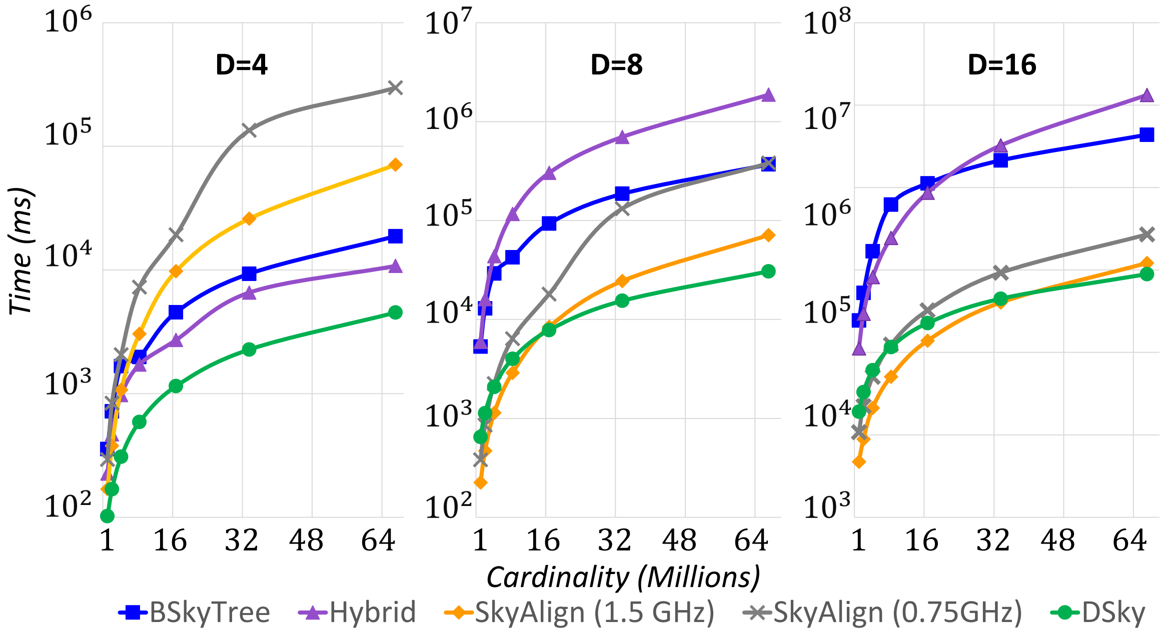


Figure 7.7: Execution time (log(t)) using anticorrelated data.

size. The only noticeable difference relates to the runtime of *SkyAlign* which is closer to that of *DSky* on 8 and 16 dimensions for higher clock frequency. The increased workload associated with anticorrelated distributions makes optimizing for work-efficiency a good strategy but only for a relatively small number of points.

7.4.2 System Throughput & Utilization

In Figure 7.8, we depict the actual to peak system throughput ratio achieved by *SkyAlign* and *DSky*. These experiments aim at quantifying the level of utilization achieved by each algorithm on the corresponding architecture. In order to attain accurate measurements, we focus on experiments with the independent and anticorrelated datasets for 8 and 16 dimensions.

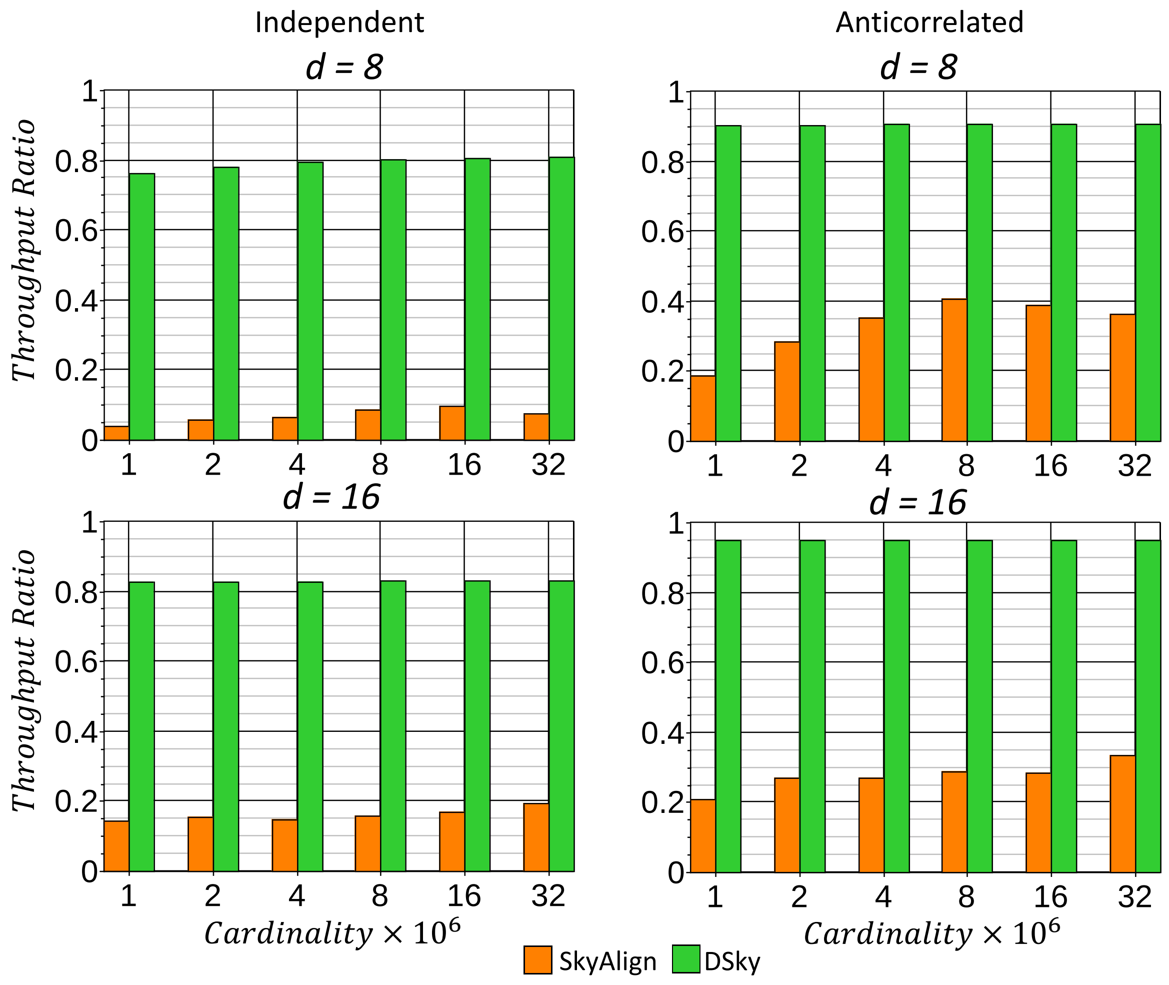


Figure 7.8: Ratio of achieved throughput over peak device throughput.

Overall, *DSky* exhibits higher and fairly constant throughput across different input size for both 8 and 16 dimension queries. Without considering communication across PCIE the expected throughput will be in the range of 90% to 97%. However, our measurements include the PCIE communication which justifies the throughput being in the worst case around 76%. In contrast, *SkyAlign* is mainly limited by the memory bandwidth and thread divergence and for 8 dimensions seems to peak around 16 million points. For 16 dimensions the system throughput increases at a constant rate, although it shows some indication of

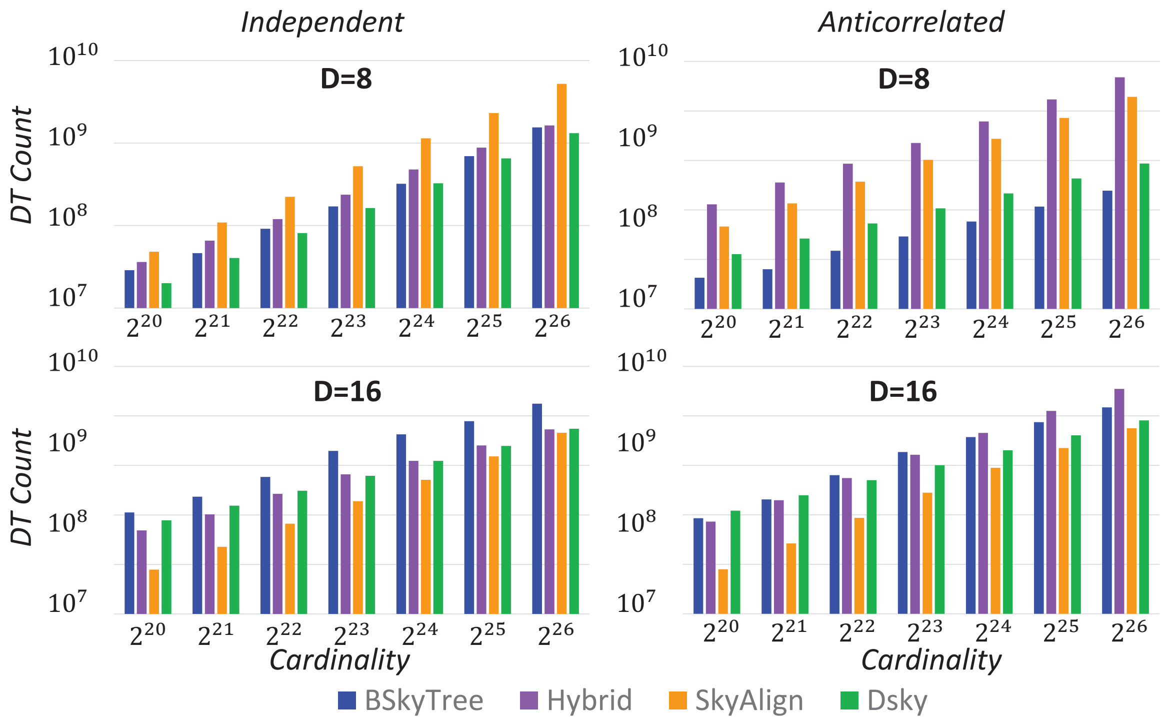


Figure 7.9: Number of executed DTs per algorithm.

reaching its peak around or after 32 million points. This behavior follows our previous claims indicating that *SkyAlign* is not throughput efficient for large scale dataset.

7.4.3 Algorithmic Efficiency & Throughput

Figure 7.9 illustrates the number of full DTs performed by all algorithms. We concentrate on independent and anticorrelated distributions and omit DTs performed on correlated data as their limited number has a lesser impact on throughput. Our experiments indicate that *DSky* exhibits remarkable efficiency for queries on 8 dimensions, outperforming the state-of-the-art parallel algorithms. In fact, its performance is closer to *BSkyTree* in terms of total DT count, indicating its ability to achieve balance between efficient pruning and detecting incomparability. This results from the optimizations related to in-order processing, early stopping and cheap filter tests using space partitioning. On 16 dimensions, *DSky* remains as efficient or slightly better than the CPU-based methods. In contrast to *SkyAlign*, *DSky* requires more DTs to compute the skyline, since the former relies on a centralized data structure to decide the ordering in which points are processed. Avoiding such a data structure comes at a trade-off, which offers opportunities for high parallelism and subsequently high throughput at the expense of doing more work.

In order to support our claims, we present in Figure 7.10 the throughput measured in million DTs per second for all implementations. We focus on the higher workload 16 dimension queries that allow for accurate throughput measurements. In our experiments, we observe that *DSky* is able to consistently maintain a higher throughput than previous state-of-the-art algorithms. Despite requiring a higher number of DTs, *DSky* maintains a higher processing rate relative to *SkyAlign* when using the same clock frequency. Intuitively, this can be attributed to a less rigid parallel execution model which allows for irregular processing, and higher bandwidth achieved through processing-in-memory. *DSky* leverages on these two properties towards being throughput efficient.

7.4.4 Scaling

We evaluate scalability by measuring the execution time, while the number of available *DPUs* increases proportionally (i.e. 8 to 4096) to the input size. Figure 7.11 contains the results of our experiments for all distributions. We focus on 8 and 16 dimension queries, which are the most compute and communication intensive case studies. Experi-

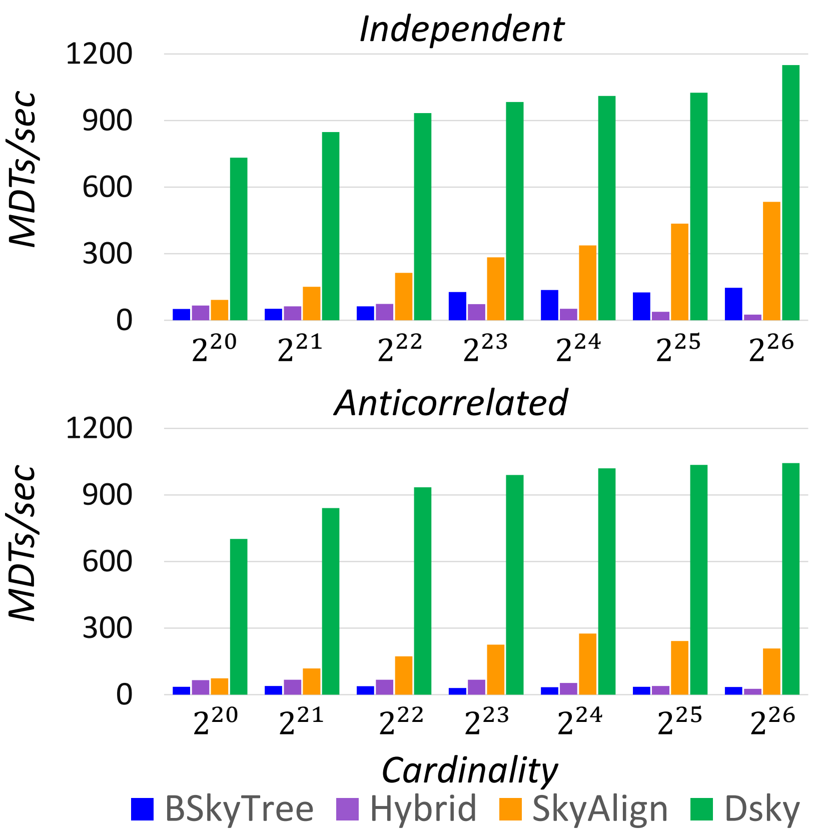


Figure 7.10: MDTs/sec for each algorithm on 16 dimensions.

ments with correlated data demonstrate a constant increase in execution time regardless of the query dimensions. We attribute this behavior to the higher cost of communication relative to processing. In practice, doubling the number of *DPUs* will improve performance only when the computation cost is sufficiently large. Low processing time offers minimal improvements over the increase in communication which dominates the overall execution time.

Independent and anticorrelated distributions require more time for processing than transmitting data, thus adding resources contributes to a higher reduction of the total execution time. In fact, as we increase the number of *DPUs* proportionally to the number

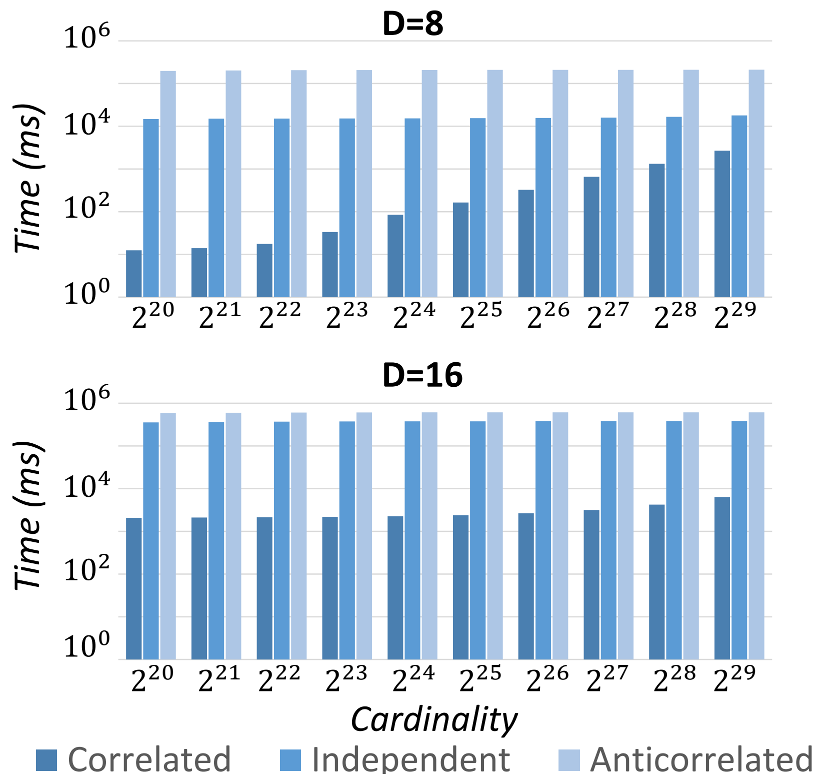


Figure 7.11: Execution time scaling with additional *DPUs*.

of points, the execution time remains fairly constant regardless of the distribution or query dimension. This showcases the ability of *DSky* to scale comfortably with respect to growing input. It is also noteworthy to mention that selecting a suitable partition size, contributes to achieving good scalability. This offers more opportunities for parallelism, while minimizing the work overhead associated with dependencies which arise from in-order processing.

7.4.5 Energy Consumption

As seen from our experimental evaluation, in most cases *DSky* achieves same or better execution time than state of the art solutions while being more throughput efficient

	CPU	GPU	PIM
Independent	0.715	1.124	0.140
Anticorrelated	1.562	2.177	0.153

Table 7.2: Energy per unit of work **(μJ/DT).**

and easily scalable. Moreover, *DSky* runs on an architecture that uses around 25% of the energy requirements (Table 7.1). Overall, this translates to more than an order of magnitude better energy consumption per unit of work in comparison to the corresponding CPU and GPU solutions, as seen in Table 7.2.

7.4.6 Fine Tuning the Partition Size

In Figure 7.12, we present the execution time for 32 million points on 16 dimension queries. Although reducing the partition size necessitates the discovery of more pivots to split the data, it contributes to a constant increase in the execution time. This happens due to radix-select which reduces data movement during preprocessing and is highly parallel. For small partition size, the communication cost grows as the associated hardware overhead is higher than the cost of data transmission. In contrast, large partitions saturate the communication channel causing a bottleneck that increases the data transmission cost. A trade-off between these extremes is achieved for partition size equal to 128. Fine tuning is important since more partitions result in more opportunities for higher parallelism and subsequently higher throughput.

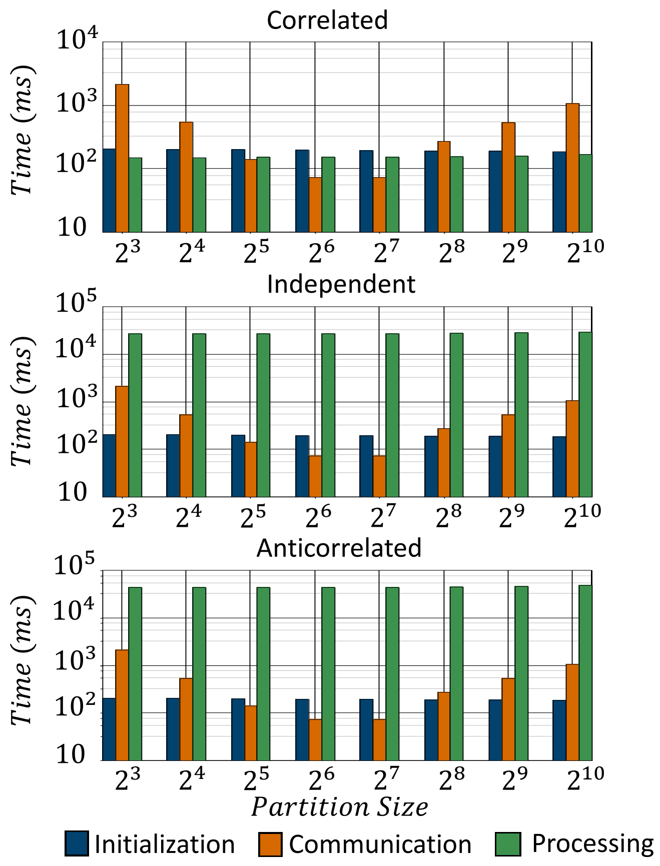


Figure 7.12: Execution time for n=32 $\times$ 10^6 d=16 on varying partition size.

7.5 Conclusion

In this work, we presented a massively parallel skyline algorithm for PIM architectures, called *DSky*. Leveraging on our novel work assignment strategy, we showcased *DSky's* ability to achieve good load balance across all participating *DPUs*. We proved that by following this methodology, the total amount of parallel work is asymptotically equal to the optimal case. Furthermore, combining spiral partitioning with blocking enabled us to seamlessly incorporate optimizations that contribute towards respectable algorithmic efficiency. Our claims have been validated by an extensive set of experiments that showcased *DSky's* ability to outperform the state-of-the-art implementations for both CPUs and GPUs. Moreover, *DSky* maintains higher processing throughput and better resource utilization. In addition, we showcased that *DSky* scales well with added resources, a feature that fits closely the capabilities of PIM architectures. Finally, our solution improves by more than an order of magnitude the energy consumption per unit of work.

Chapter 8

Conclusion and Future Work

In recent years there has been a huge influx on the number of applications responsible for generating multi-dimensional data. This trend combined with the increasing demand in discovering interesting data insights, using complex query operators puts significant pressure on the processing capabilities of modern decision support systems. In order to improve the processing capacity of such systems, researchers proposed the development of main memory databases that are tightly integrated with modern multi-core or many-core architectures. In this dissertation, we concentrated on tackling the challenges related to the deployment of complex query operators (such as the Top-K and Skyline operators), within a main memory environment that consists of either a modern multi-core CPU or certain established hardware accelerators (e.g. GPUs, PIM).

In Chapter 3, we presented an efficient main memory algorithm for multi-core CPUs which outperforms methods proposed in previous work while maintaining good work efficiency. The algorithm (termed PTA) leverages on intelligent partitioning to reduce the number of object evaluations for every possible linear monotone function when evaluating multi-attribute Top-K queries. Our analysis and extensive experimental result demonstrate that this method can operate efficiently for different types of Top-K queries and data distributions.

In Chapter 4, we presented an algorithm designed to enable Top-K selection with early termination on the GPU. The proposed early termination GPU algorithm was an adaptation of the equivalent CPU algorithm established in Chapter 3. In order to attain high throughput processing, we combined the early termination strategy (that was based on angle space partitioning) with an optimized GPU kernel called Bitonic Top-K. Our solution outperformed the previous state-of-the-art full table evaluation (FTE) solution for varying data distributions and query instances when the corresponding data reside exclusively in device memory. In addition, early termination provided several opportunities to improve query processing when the total amount of data exceed the corresponding GPU memory capacity. In that case, we were able to seamlessly integrate data caching with the concept of unified memory to enable query processing without affecting the overall cost of query evaluation. In fact, early termination enabled us to improve over the only existing solution which would have us access the data from the remote host memory incurring a significantly higher processing cost.

In Chapter 6, we studied different categories of algorithms which are suitable for enabling efficient Top-K selection on PIM. We developed two types of FTE solutions, mainly (1) sort based (2) heap based Top-K query evaluation. In addition, we adopted the practices established for GPU early termination and developed an equivalent solution for

PIM. Our detailed experimental evaluation showcased that sorting is much more expensive compared to the heap based approach. For this reason, we considered heap based Top-K query evaluation to be the best solution from all possible solutions contained in the FTE class. We compared heap based Top-K on PIM against the equivalent solutions that were specifically optimized for CPU and GPU processing and observed that PIM attains higher throughput, higher system utilization, and overall lower latency. Furthermore, we concentrated on experiments comparing the performance of early termination against FTE on PIM, and noticed that our measurements were consistent with the performance gains indicated in the performance evaluation of all the other tested architectures (i.e. CPU and GPU). Finally, early termination on PIM outperformed the equivalent CPU and GPU algorithms by more than an order of magnitude. This behavior was expected given that PIM is inherently better in providing lower data access latency during processing.

In Chapter 7, we developed a massively parallel skyline algorithm for PIM architectures. Our solution, which we called *DSky*, concentrates on maintaining good load balance while maximizing parallelism across the available PIM processors. In addition to that, we focused on efficiently adopting previously proposed early termination practices without significantly affecting the parallel work. The proposed algorithm follows the bulk synchronous processing model having as ultimate goal to overlap computation with communication in order to maximize the processing throughput. Our extensive experimental result demonstrates that *DSky* outperforms previous work optimized for CPU and GPU processing by more than an order of magnitude. In addition, we showcased that the proposed algorithm enables high system utilization in the order of 80 to 90%.

Over the recent years there has been a great amount of work focused on combining Top-K queries with other operators (e.g. Top-K joins [64, 49]). In addition several attempts have been made to merge Top-K queries with the Skyline operator [61]. Give the wealth of problems deriving from this two basic operators, we believe that our future work should focus on adopting the previously described strategies to solve the aforementioned problems. Currently, as we have focused on selection only solving such problems is very challenging since our methods will need to be amended to operate efficiently on an environment containing multiple tables. This creates additional challenges related to work assignment and load balancing especially for massively parallel architectures (i.e. GPUs or PIM).

Another promising venue of research is discovering other database operators which are primarily memory bound and attempting to implement them on PIM. This is a promising research direction because as we demonstrated PIM outperforms other architectures by more than an order of magnitude. An example of such operator which is widely popular for databases is the natural join operator. Implementing it on PIM comes with its own set of challenging mostly related to load balancing as a result of the DPU memory capacity limitations. Such a research direction will certainly have a high impact on the characteristics of the architecture itself since it is still at the early stages of development.

Bibliography

[1] Ildar Absalyamov, Prerna Budhkar, Skyler Windh, Robert J Halstead, Walid A Najjar, and Vassilis J Tsotras. FPGA-accelerated group-by aggregation using synchronizing caches. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 11. ACM, 2016.

[2] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. On the surprising behavior of distance metrics in high dimensional space. In *ICDT*, pages 420–434. Springer, 2001.

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, pages 105–117. IEEE, 2015.

[4] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Best position algorithms for top-k queries. In *Proceedings of the 33rd international conference on Very large data bases*, pages 495–506. VLDB Endowment, 2007.

[5] Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. Fast k-selection algorithms for graphics processing units. *JEA*, 17:4–2, 2012.

[6] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014.

[7] Wolf-Tilo Balke and Ulrich Güntzer. Multi-objective query processing for database systems. In *VLDB*, pages 936–947. VLDB Endowment, 2004.

[8] Wolf-Tilo Balke, Ulrich Güntzer, and Jason Xin Zheng. Efficient distributed skylining for web information systems. In *EDBT*, pages 256–273. Springer, 2004.

[9] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 29th International Conference on Data Engineering*, pages 362–373. IEEE, 2013.

[10] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Efficient sort-based skyline evaluation. *TODS*, 33(4):31, 2008.

[11] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proceedings of the 32nd international conference on Very large data bases*, pages 475–486. VLDB Endowment, 2006.

[12] Christian Beecks, Ira Assent, and Thomas Seidl. Content-based multimedia retrieval in the presence of unknown user preferences. *Advances in Multimedia Modeling*, pages 140–150, 2011.

[13] Giovanni Beltrame, Luca Fossati, and Donatella Sciuto. Decision-theoretic design space exploration of multiprocessor platforms. *TCAD*, 29(7):1083–1095, 2010.

[14] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *VLSI*, 8(3):299–316, 2000.

[15] Kenneth S Bøgh, Sean Chester, and Ira Assent. Work-efficient parallel skyline computation for the gpu. *VLDB*, 8(9):962–973, 2015.

[16] Kenneth S Bøgh, Sean Chester, Darius Šidlauskas, and Ira Assent. Template skycube algorithms for heterogeneous parallelism on multicore and gpu architectures. In *SIGMOD*, pages 447–462. ACM, 2017.

[17] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 421–430. IEEE, 2001.

[18] Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 426–434. ACM, 2003.

[19] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R Smith. The onion technique: indexing for linear optimization queries. In *ACM Sigmod Record*, volume 29, pages 391–402. ACM, 2000.

[20] Sean Chester, Michael L Mortensen, and Ira Assent. On the suitability of skyline queries for data exploration. In *EDBT/ICDT*, pages 161–166. IEEE, 2014.

[21] Sean Chester, Darius Šidlauskas, Ira Assent, and Kenneth S Bøgh. Scalable parallelization of skyline computation for multi-core processors. In *Data Engineering (ICDE), IEEE 31st International Conference on*, pages 1083–1094. IEEE, 2015.

[22] Sean Chester, Alex Thomo, S Venkatesh, and Sue Whitesides. Computing k-regret minimizing sets. *Proceedings of the VLDB Endowment*, 7(5):389–400, 2014.

[23] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting: Theory and optimizations. In *IIPWM*, pages 595–604. Springer, 2005.

[24] Gao Cong, Christian S Jensen, and Dingming Wu. Efficient retrieval of the top-k most relevant spatial web objects. *Proceedings of the VLDB Endowment*, 2(1):337–348, 2009.

[25] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd international conference on Very large data bases*, pages 451–462. VLDB Endowment, 2006.

[26] Elena Demidova, Peter Fankhauser, Xuan Zhou, and Wolfgang Nejdl. Divq: diversification for keyword search over structured databases. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 331–338. ACM, 2010.

[27] Ke Deng, Xiaofang Zhou, and Heng Tao. Multi-source skyline query processing in road networks. In *ICDE*, pages 796–805. IEEE, 2007.

[28] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. A candidate filtering mechanism for fast top-k query processing on modern cpus. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 723–732. ACM, 2013.

[29] Shuai Ding, Jinru He, Hao Yan, and Torsten 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.

[30] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 993–1002. ACM, 2011.

[31] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, et al. The architecture of the diva processing-in-memory chip. In *ICS*, pages 14–25. ACM, 2002.

[32] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The mondrian data engine. In *ISCA*, pages 639–651. ACM, 2017.

[33] Jiunn-Der Duh and Daniel G Brown. Knowledge-informed pareto simulated annealing for multi-objective spatial allocation. *Computers, Environment and Urban Systems*, 31(3):253–281, 2007.

[34] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[35] Marcus Fontoura, Vanja Josifovski, Jinhui Liu, Srihari Venkatesan, Xiangfei Zhu, and Jason Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proceedings of the VLDB Endowment*, 4(12):1213–1224, 2011.

[36] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Algorithms and analyses for maximal vector computation. *VLDB*, 16(1):5–28, 2007.

[37] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

[38] J Guntzer, W-T Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *Information Technology: Coding and Computing. Proceedings. International Conference on*, pages 622–628. IEEE, 2001.

[39] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th international conference on very large data bases*, pages 419–428. Morgan Kaufmann Publishers Inc., 2000.

[40] Qi Guo, Nikolaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze Meng Low, Larry Pileggi, James C Hoe, and Franz Franchetti. 3d-stacked memory-side acceleration: Accelerator and system design. In *WoNDP*, 2014.

[41] Xixian Han, Jianzhong Li, and Hong Gao. Efficient top-k retrieval on massive data. *IEEE Transactions on Knowledge and Data Engineering*, 27(10):2687–2699, 2015.

[42] Xixian Han, Jianzhong Li, and Hong Gao. Efficient top-k dominating computation on massive data. *IEEE Transactions on Knowledge and Data Engineering*, 29(6):1199–1211, 2017.

[43] Xixian Han, Jianzhong Li, and Donghua Yang. Supporting early pruning in top-k query processing on massive data. *Information Processing Letters*, 111(11):524–532, 2011.

[44] Xixian Han, Xianmin Liu, Jianzhong Li, and Hong Gao. Tkap: Efficiently processing top-k query on massive data by adaptive pruning. *Knowledge and Information Systems*, 47(2):301–328, 2016.

[45] Jun-Seok Heo, Junghoo Cho, and Kyu-Young Whang. The hybrid-layer index: A synergic approach to answering top-k queries in arbitrary subspaces. In *ICDE*, pages 445–448, 2010.

[46] Jun-Seok Heo, Kyu-Young Whang, Min-Soo Kim, Yi-Reun Kim, and Il-Yeol 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.

[47] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *CGO*, pages 165–174. ACM, 2008.

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

[49] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB JournalThe International Journal on Very Large Data Bases*, 13(3):207–221, 2004.

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

[51] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 253–262. ACM, 2014.

[52] Herbert Jordan, Peter Thoman, Juan J Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *SC*, pages 1–12. IEEE, 2012.

[53] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. In *Proceedings of the 8th International Workshop on Data Management on New Hardware*, pages 55–62. ACM, 2012.

[54] Hina A Khan, Mohamed A Sharaf, and Abdullah Albarrak. Divide: efficient diversification for interactive data exploration. In *SSDBM*, page 15. ACM, 2014.

[55] Henning Köhler, Jing Yang, and Xiaofang Zhou. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD*, pages 85–96. ACM, 2011.

[56] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286. VLDB Endowment, 2002.

[57] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert. Route skyline queries: A multi-preference path planning approach. In *ICDE*, pages 261–272. IEEE, 2010.

[58] Dominique Lavenier, Jean Francois Roy, and David Furodet. DNA mapping using processor-in-memory architecture. In *BIBM*, pages 1429–1435. IEEE, 2016.

[59] Jongwuk Lee, Hyunsouk Cho, Sunyou Lee, and Seung-won Hwang. Toward scalable indexing for top-k queries. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3103–3116, 2014.

[60] Jongwuk Lee and Seung-won Hwang. Bskytree: scalable skyline computation using a balanced pivot selection. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 195–206. ACM, 2010.

[61] Jongwuk Lee, Gae-won You, and Seung-won Hwang. Personalized top-k skyline queries in high-dimensional space. *Information Systems*, 34(1):45–61, 2009.

[62] Ken CK Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. Approaching the skyline in z order. In *VLDB*, pages 279–290. VLDB Endowment, 2007.

[63] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F Ilyas. Supporting ad-hoc ranking aggregates. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 61–72. ACM, 2006.

[64] Vebjorn Ljosa and Ambuj K Singh. Top-k spatial joins of probabilistic objects. In *2008 IEEE 24th International Conference on Data Engineering*, pages 566–575. IEEE, 2008.

[65] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W Cheung. Efficient top-k aggregation of ranked inputs. *ACM Transactions on Database Systems (TODS)*, 32(3):19, 2007.

[66] Matthew J Menne, Imke Durre, Russell S Vose, Byron E Gleason, and Tamara G Houston. An overview of the global historical climatology network-daily database. *Journal of Atmospheric and Oceanic Technology*, 29(7):897–910, 2012.

[67] Duane Merrill. Cub, 2016. `https://nvlabs.github.io/cub/`.

[68] Cayman Mitchell, Nelson Schoenbrot, Joshua Shor, Keith Thomas, and Sung-Hyuk Cha. Radix selection algorithm for the kth order statistic. 2015.

[69] Mohamed F Mokbel and Justin J Levandoski. Toward context and preference-aware location-based services. In *MobiDE*, pages 25–32. ACM, 2009.

[70] Michael Morse, Jignesh M Patel, and William I Grosky. Efficient continuous skyline computation. *Information Sciences*, 177(17):3411–3437, 2007.

[71] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *HPCA*, pages 457–468. IEEE, 2017.

[72] Aziz Nasridinov, Jong-Hyeok Choi, and Young-Ho Park. A two-phase data space partitioning for efficient skyline computation. *Cluster Computing*, 20(4):3617–3628, 2017.

[73] Apostol Natsev, Yuan-Chi Chang, John R Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, volume 1, pages 281–290, 2001.

[74] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Respir: a response surface-based pareto iterative refinement for application-specific design space exploration. *TCAD*, 28(12):1816–1829, 2009.

[75] Hweehwa Pang, Xuhua Ding, and Baihua Zheng. Efficient processing of exact top-k queries over disk-resident sorted lists. *The VLDB JournalThe International Journal on Very Large Data Bases*, 19(3):437–456, 2010.

[76] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478. ACM, 2003.

[77] Sungwoo Park, Taekyung Kim, Jonghyun Park, Jinha Kim, and Hyeonseung Im. Parallel skyline computation on multicore architectures. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 760–771. IEEE, 2009.

[78] Yoonjae Park, Jun-Ki Min, and Kyuseok Shim. Efficient processing of skyline queries using mapreduce. *TKDE*, 29(5):1031–1044, 2017.

[79] Antonin Ponsich, Antonio Lopez Jaimes, and Carlos A Coello Coello. A survey on multiobjective evolutionary algorithms for the solution of the portfolio optimization problem and other finance and economics applications. *TEVC*, 17(3):321–344, 2013.

[80] Anil Shanbhag, Holger Pirk, and Samuel 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.

[81] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *SIGARCH*, volume 42, pages 97–108. IEEE, 2014.

[82] Mehdi Sharifzadeh, Cyrus Shahabi, and Leyla Kazemi. Processing spatial skyline queries in both vector spaces and spatial network databases. *TODS*, 34(3):14, 2009.

[83] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In *MEMSYS*, pages 295–308. ACM, 2016.

[84] Adam Silberstein Silberstein, Rebecca Braynard, Carla Ellis, Kamesh Munagala, and Jun Yang. A sampling-based approach to optimizing top-k queries in sensor networks. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 68–68. IEEE, 2006.

[85] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio, Roberto Zafalon, Prabhat Avasare, Geert Vanmeerbeeck, et al. Multicube: Multi-objective design space exploration of multi-core architectures. In *VLSI*, pages 47–63. Springer, 2010.

[86] Evangelia A Sitaridi and Kenneth A Ross. Optimizing select conditions on gpus. In *Proceedings of the 9th International Workshop on Data Management on New Hardware*, page 4. ACM, 2013.

[87] Dimitrios Skoutas, Dimitris Sacharidis, Alkis Simitsis, and Timos Sellis. Serving the sky: Discovering and selecting semantic web services through dynamic skyline queries. In *ICSC*, pages 222–229. IEEE, 2008.

[88] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*, pages 541–552. IEEE, 2017.

[89] Yufei Tao, Vagelis Hristidis, Dimitris Papadias, and Yannis Papakonstantinou. Branch-and-bound processing of ranked queries. *Information Systems*, 32:424–445, 2007.

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

[91] Shirish Tatikonda, B Barla Cambazoglu, and Flavio P Junqueira. Posting list intersection on multicore architectures. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 963–972. ACM, 2011.

[92] Shirish Tatikonda, Flavio Junqueira, B Barla Cambazoglu, and Vassilis Plachouras. On efficient posting list intersection with multicore processors. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 738–739. ACM, 2009.

[93] Ed Upchurch, Thomas Sterling, and Jay Brockman. Analysis and modeling of advanced pim architecture design tradeoffs. In *SC*, 2004.

[94] UPMEM SAS. UPMEM SDK, 2015. `http://www.upmem.com/wp-content/uploads/2017/02/20170210_SDK_One-Pager.pdf`.

[95] UPMEM SAS. UPMEM web, 2015. `http://www.upmem.com`.

[96] Akrivi Vlachou, Christos Doulkeridis, and Yannis Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 227–238. ACM, 2008.

[97] Shangguang Wang, Qibo Sun, Hua Zou, and Fangchun Yang. Particle swarm optimization with skyline operator for fast cloud-based web service composition. *Mobile Networks and Applications*, 18(1):116–121, 2013.

[98] Louis Woods, Gustavo Alonso, and Jens Teubner. Parallel computation of skyline queries. In *FCCM*, pages 1–8. IEEE, 2013.

[99] Min Xie, Laks VS Lakshmanan, and Peter 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.

[100] Dong Xin, Chen Chen, and Jiawei Han. Towards robust indexing for ranked queries. In *Proceedings of the 32nd international conference on Very large data bases*, pages 235–246. VLDB Endowment, 2006.

[101] Sotirios Xydis, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. Spirit: spectral-aware pareto iterative refinement optimization for supervised high-level synthesis. *TCAD*, 34(1):155–159, 2015.

[102] Jeong-Min Yun, Yuxiong He, Sameh Elnikety, and Shaolei Ren. Optimal aggregation policy for reducing tail latency of web search. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 63–72. ACM, 2015.

[103] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: throughput-oriented programmable processing in memory. In *HPDC*, pages 85–98. ACM, 2014.

[104] Shile Zhang, Chao Sun, and Zhenying 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.

[105] Vasileios Zois. Top-k selection. `https://github.com/vzois/TopK`.

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

[107] Lei Zou and Lei 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.